POLITECNICO
MILANO 1863

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
Corso di Laurea Magistrale in Computer Science and Engineering

Workflow based orchestrations of serverless workloads with ephemeral statestore

Supervised by: Prof. Alessandro Margara

Submitted by:

Nandaja Varma Nandakumar

894333

Academic Year 2019/2020

Abstract

Serverless Computing is an up and coming Platform as a Service(PaaS) offering where the cloud provider manages and allocates resources needed to keep the application running. This lets the developer focus on the application development and not on server maintenance. Alongside off loading the provisioning and maintenance of the server, Serverless computing also reduces resource waste by scaling up and down the allocation depending on the load and the configurations. The users only pay for the resources that were used by the application thereby saving huge operational cost on their infrastructure hosting.

Although Serverless might sound like the holy grail of application hosting, the current state of art technology fall short in several places to meet the industrial requirements. Data intensive applications, streaming applications, and distributed computing are some of the fields that could be benefited heavily by implementation on Serverless platforms in terms of ease of development, efficiency and cost. But all the existing platforms offer very poor performance in these fields and works mostly via workarounds and numerous third party tools.

This thesis analyses the Serverless paradigm in depth, pointing out the reasons for this reduced adaptability. To solve these issues, we propose a lightweight extension to OpenFaaS, an Open Source Serverless platform, that provides flexibility, scalability and adaptability, while making sure not to violate the notion of functions. Our implementation tries to reduce the operational gap between the industrial applications and theoretical ideas put forward by researches in the past few years. This thesis also offers a deep study of the full potential and limitations of Serverless thereby making it clear to the reader why more innovation is necessary in this field.

Sommario

Le piattaforme di Serverless Computing sono soluzioni emergenti Platform as a Server (PaaS) dove il fornitore di servizi cloud gestisce e alloca le risorse necessarie per mantenere in esecuzione l'applicazione. Ciò consente allo sviluppatore di concentrarsi sullo sviluppo del software e non sulla manutenzione del server. Oltre a diminuire il carico di lavoro per il provisioning e la manutenzione del server, questa tecnologia riduce anche lo spreco di risorse aumentando o diminuendo l'allocazione a seconda del traffico e delle configurazioni. In questo modo l'utente paga solo per le risorse che sono state utilizzate dall'applicazione risparmiando così enormi costi operativi sulla loro infrastruttura ospitante.

Sebbene il Serverless Computing possa sembrare il santo Graal dell'hosting di applicazioni, attualmente le tecnologie all'avanguardia non sono all'altezza in molti casi di soddisfare i requisiti industriali. Applicazioni con uso intensivo di dati, applicazioni di streaming e di elaborazione distribuita sono alcuni dei campi che potrebbero trarre grandi vantaggi da un'implementazione su piattaforme Serverless in termini di facilità di sviluppo, efficienza e costo. Tuttavia tutte le piattaforme esistenti offrono soluzioni alternative combinando strumenti di terze parti, con scarse prestazioni.

La presente tesi analizza in profondità il paradigma Serverless, sottolineando le ragioni della sua ridotta adattabilità. Per risolvere questi problemi, proponiamo una leggera estensione di OpenFaaS, una piattaforma Serverless open source che fornisce flessibilità, scalabilità e adattabilità assicurandosi nel contempo di non violare la nozione di funzioni. La nostra implementazione cerca di ridurre il divario operativo tra le applicazioni industriali e le idee teoriche prodotte dalle ricerche negli ultimi anni. Questa tesi offre anche uno studio approfondito del pieno potenziale e dei limiti del Serverless Computing, rendendo così chiaro al lettore la necessità di innovazione in questo campo.

Contents

List of Figures

List of Tables

1 Introduction

Serverless can easily be considered as the new generation of Platform as a Service(PaaS). It is a deployment solution where instead of having continuously running servers, application instances come up and execute on predefined events. While the developers worry about the logic of handling the requests/events, the infrastructure provider takes care of receiving the request, responding to them, capacity planning, task scheduling, and operational monitoring [1] This has huge economical and architectural implications that is still waiting to be explored in its full potential.

In the current industrial workloads, Applications are increasingly becoming data intensive day by day paving way to adopt several resource heavy tools to do stream processing, distributed processing, etc. More than often CPU and memory loads in these machines tent to vary a lot and rather than having a dedicated server to accommodate the whole range of requirements, it makes perfect sense to convert it into a Serverless workload thereby saving up on operational cost, resource waste, and ease of development. However, the current commercial offerings of Serverless do not work very well with such workloads.

This is mostly due to the sheer nature of the Serverless paradigm of being completely stateless, thereby forcing the developers to use external block storages for data store and communication. By the design of it, Serverless applications are deployed as isolated entities which are hard to address directly via the network. This makes the composition of functions a tad bit complicated. The most commercial Cloud Service Providers currently offer wrapper solutions to workaround the composition problem. A very notable commercial solution here is AWS stepfunction. AWS stepfunction provide an API to define function compositions, which eventually get executed in AWS Lambda, the pioneer in serverless platforms. Other than enforcing vendor lock in to AWS, stepfunction comes with numerous limitations like 20s timeout on the API gateway, 5 minutes limit to lambda execution, a limit of 2 executions per second etc.

Another major requirement usually for heavy multi-staged processing pipelines is fault tolerance. If at all a stage fails, the system should be able to restart from where it failed with the correct intermediate data and complete the workflow. In orchestrations like stepfunction where data is passed over the gateway among each other, if a failure occur the processed data till the failure function is gone for good. This makes the system unreliable for certain heavy load applications.

In this thesis, we propose an approach to improve the performance of big data workloads on the Serverless platform by introducing the provision to provide a computational graph to the Serverless platform which defines the control flow and data flow in the orchestration. The intermediate data transfer between the functions will be taken care with the help of maintaining a scalable in memory distributed cache and storing the intermediate data in them as ephemeral data. Our system also provides fine grained control over resource allocation and scaling rules for each individual function. Alongside, it provides extensive monitoring, function level tracing and visualization, and out of box setup and deployment. Because of the usage of the intermediate ephemeral storage and fine grained monitoring the system provides automatic fail overs. Meaning, if at any step the operation fails, the pipeline will be able to restart from the point of failure without redoing the whole process till then. The system will offer an at-least-once guarantee in the request handling. This is mostly because of the message queue that is being used by the FaaS system we built on called NATS streaming [47].

It is worth mentioning that our implementation focuses on reducing the gap that currently exists with a lot of research ideas and the industry level applications, making it an easily adaptable solution. We propose a very secure and multi-tenant implementation of a state-ful Serverless setup which can be easily used for production quality applications. The possibility to efficiently do application performance and usage monitoring makes fine grained billing an out of the box functionality.

Managing intermediate data via additional infrastructure instead of altering the stateless nature of the function was a conscious choice. The serverless computing abstraction, despite its many advantages, exposes several low-level operational details that make it hard for programmers to write and reason about their code [4]. This is related to misusing/misunderstanding state in serverless environment. Since the same function is reused again and again to avoid latency, the cache or state persists across invocations leading to faulty results. If the state store is handled by an external party mapping to the invocation id, a lot of this faulty management can be handled.

As for the implementation, we proceed by extending an existing, widely used Serverless platform called OpenFaaS so as to make it readily adaptable as a FaaS infrastructure for production quality applications. As for the function composition, we use an existing library faas-flow to support event driven workflow based function composition pattern. We make sure that the notion of a function or serverless platform will not be violated in this process since with the current state of art of infrastructure deployment, autoscaling and concurrency happen by leveraging the notions of statelessness and functional coding. The proof of concept of our solution was successfully implemented [59] and has been Open Sourced.

There has been several academic researches on ephemeral autoscaling storages in the past couple of years. Pocket project is one that has received appreciation and we will be analyzing this platform in depth. Our thesis implementation adapts an existing ephemeral storage platform.

Using our proposed Serverless setup, we try to efficiently run an Extract-Transfer-Load(ETL) workload on streaming data. ETL basically is a pipelined workflow that involves receiving data from source, cleaning and transforming it, and loading it to a sink. We will split the whole operation into multiple functions as per the Serverless notion and have them communicate data internally via the ephemeral data store to complete the pipeline thereby reducing the latency and external bottlenecks.

This document describes more on Serverless paradigm, the shortcomings of it, the ones we are trying to solve, our solution and evaluation. It is split into several sections as follows:

In Section 2, we go a bit in depth to understand the history of cloud infrastructure and the technological innovations that led to Serverless paradigm. We also look in detail at the characteristics and nature of Serverless. We look at some commercial Serverless offerings and understand how in the programming world Serverless has affected even the way of coding. We will also see what limitations it holds at its current state of art and the problem statement of the thesis.

In Section 3, We elaborate on the proposed solution for our Serverless setup going into detail about how certain crippling limitations can be overcome.

In Section 4, we present the implementation of the system including the architecture and the tools used.

In Section 5, we go on with the evaluation of our system.

In Section 6, we look at the current state of research in the field of Serverless technologies and the related works.

In Section 7, we lay down the future works that the system has got planned moving forward.

We conclude by pointing out what went well and what did not with our solution in Section 8.

2 Background and Motivation

The term Serverless has been vaguely thrown around the domain of cloud infrastructure in the past decade as the breakthrough resource(and hence money) saving tool that lets the developers focus on application logic rather than the deployment and server maintenance. However, it is often hard to define what exactly serverless is since the service offering tend to change based on the cloud provider and the interpretations of the users. It is fair to say that serverless is a huge leap in the direction of using computational power as a resource which is paid for, according to the usage. Although the terminology is irrelevant, we will be focusing on the serverless offering called

Function-as-a-Service(FaaS) where the cloud providers offer a platform to which we can upload our application code to(complying to the API rules) and get uninterrupted service of the same at an endpoint no matter how the traffic or data load might be. Paying only for what resources has been used adds to the attraction of the domain. In this section, we will understand more about this technology, the popular commercial offerings of the same, and its limitations and the current state of research. We will also analyze the popular data processing and streaming pipelines in the industry these days and why Serverless computing fall short in being the right tool of development and deployment in some cases.

2.1 Evolution of cloud resource management

In the past three decades, software deployment and infrastructure management has seen a lot of innovation and evolution. Before diving into the current industrial standards, it is important to understand the evolution in this field to get a better grasp on the technological innovations that brought about this change.

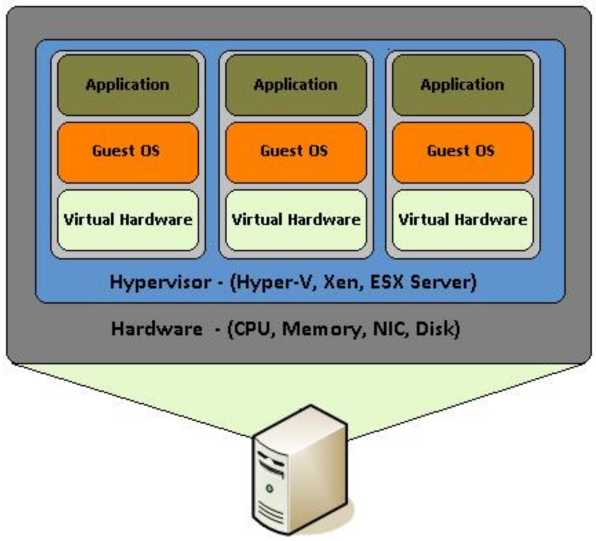
2.1.1 Dedicated servers

Even as recent as fifteen years ago, using dedicated servers was the industry standard for deployments. Dedicated servers are physical machines. The general practice was to have server racks on the premise of the company which are maintained by system administrators and all your software is hosted there. Although this method offers advanced security and high availability, it is often common that a lot of physical resources were underutilized and each resource was for single client. Not to mention the environmental impact of the reserved heavy hardware which leaves a heavy carbon footprint and e-wastes.

2.1.2 Dedicated virtual machines(BaaS)

Virtualization technology changed the face of software infrastructure by decoupling applications from the underlying hardware. Virtualized servers are not physical machines, they are a software construct. Virtual servers run on dedicated servers, the resources of which are divided between several virtual servers. To get slightly technical, virtualization usually involves installing a virtualization software(Hypervisor) on an existing operating system and then having multiple operating systems on it, sharing all the resources of the host operating system, yet providing great security and isolation.

Figure 1: Virtualization through hupervisors



Although applications hosted on the virtual machine suffer from a heavy input/output and network overload because of the added layer of indirection, this technology reduces the resource waste to a great extend. The enterprises could partition their hardware into multiple virtual machines and have different hosting and computation in each of the them. System administrators started splitting up their bare metal resources among multiple Virtual Private Servers(VPS) by the help of virtualization software. Each VPS would give you the feeling of having a real system although it is a virtualized system which is sharing the resources with other VPSs. This reduced a lot the amount of work and energy spent on maintaining server racks along with the terrible underutilization of resources.

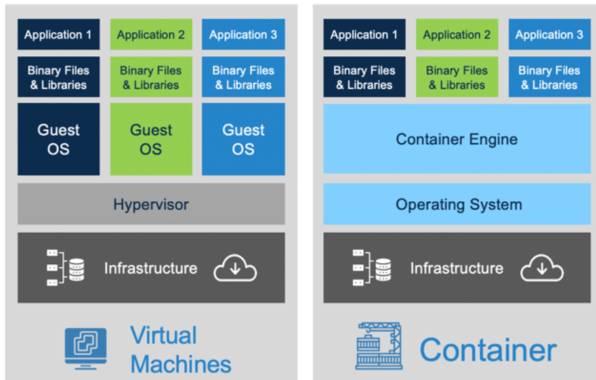
More and more companies started adapting this technology and in early 2006 Amazon Web Services(AWS) re-launched themselves as a platform that offers computing and storage space to developers and enterprises on an on-demand basis revolutionizing how companies were designing their system architecture. Soon after Google and Microsoft followed suit with their cloud infrastructure platforms offering similar services. All these providers function by maintaining huge, dedicated server farms across the globe to provide the necessary resources to the customers.

These kind of services, generally called as Infrastructure as a Service(IaaS) or Platform as a Service(PaaS), went through a series of changes during the past decade. On-demand compute instances to completely managed deployment services(eg: Google App Engine), Pay per use block storages(AWS S3) to fully managed dedicated relational databases(Google Cloud SQL, AWS RDS, etc.) a lot of really efficient and interesting services started to be available for the developers disposition. The billing scheme of these services also started to be quite flexible even allowing a per second billing plan in the past couple of years by Google.

It is also worth noting that with the advent of virtualization, the job profiles in several companies shifted from having a system administrator role to having profiles called DevOps(development and operations) who are application developers focusing on the provisioning of the virtual machines to deploy their applications. Although IaaS solved a lot of hassle around infrastructure provisioning, the systems and load of the applications still remained independent. Applications always had dedicated virtual machines even if the load/traffic to and fro the application is not constant. This meant that a lot of resources were still being wasted.

Linux Containers A game changer in the world of virtualization was containerization. Containers are yet another packaged computing environment that combine various IT components and isolate them from the rest of the system just like a virtual machine would. It was developed to solve a lot of problems with virtual machines. The purpose of the containers is to encapsulate an application and its dependencies within its own environment. This allows them to run in isolation while they are using the same system resources and the same operating system. Since the resources are not wasted on running separate operating systems tasks, containerization allows for a much quicker, lightweight deployment of applications. Each container image could be only a few megabytes in size, making it easier to share, migrate, and move. Figure 2 shows the difference in the isolation levels of containers and virtual machines. Even though Linux Containers [5] have existed for a very long time, in the past decade, containers were made a lot more approachable and adaptable as a technology by the advent of communities like Docker and rkt.

Figure 2: Virtual Machines Vs Containers



The light weight of the containers made it the ideal candidate for running applications. What makes container based deployments special as opposed to the ones deployed directly on the host is the consistency of the environment. The application execution environment can be recreated and ported from one system to another without affecting the functionality of the application or having to reinstall the whole binary dependencies on the new machine. Reproducability of the production environment even in the local exactly, meant that the development/testing cycle became much more efficient. The isolated package of the application, enveloped as a container image, is agnostic of the operating system it runs on opening new possibilities for the deployment. One could also limit and fine tune the resources used by a running containers giving a lot more control over the application.

Autoscaling The ease in which one can limit the resources and tweak the runtime parameters externally contributed heavily to the service offering called autoscaling which basically meant resources for an application runtime were added or removed as per the usage. All the commercial cloud providers started offering the aforementioned service in different flavors. Autoscaling on EC2 or Google Compute, AWS Fargate, etc. are some examples.

In the past two years, innovations have taken a leap in the field of isolation environments, introducing solutions like AWS Firecracker, Cloudflare workers, etc. to the community. These solutions aim at mitigating the shortcomings of Containers which we will discuss in Section 2.2.4

2.1.3 Serverless

Like mentioned earlier, in the past two years the terms Serverless and Function-as-a-Service are quite often used interchangeably. In terms of the resource reservation, Serverless can be considered as a platform as a service solution that scales. Your application will always have enough and only enough resources dedicated to it. It will scale up and down based on the load and traffic and the developer only pays for the usage. This paradigm of autoscaling has been hence applied even to database storage solutions by major cloud providers such that even the block storage is allocated based on usage and there will be a burst of reservation as soon as a certain limit is reached. The pioneers of this technology can be considered as the proprietary service Lambda by Amazon Web Services[6]. Several other cloud providers followed suit with similar platforms specific to their infrastructure. The nature of serverless makes it attractive for both developers and the cloud providers since in the case of former, it means paying much less and in case of the latter, it means they can easily provide shared tenant resource allocation units.

We will dive more into the properties and nature of the solution Function-as-a-Service(FaaS) in the following session.

2.2 FaaS

So far, we have covered the infrastructure management style of FaaS or Serverless in general. Let us discuss in detail the specifics of the hosting platform that provides the FaaS functionality.

Most FaaS platforms being closed source, provides the client API for developers to supply a package including their code and dependencies to. Most platforms supports a limited set of programming language runtime although it is usually possible to do workarounds to deploy custom runtime. Behind the screen, the platform containerizes the application and deploy it so as to get triggered via pre-defined hooks specified by the developer. The infrastructure also provides endpoints or interfaces to specify the maximum and minimum CPU and memory allocated for the application, the maximum timeout for the application(although there is a hard bound on this imposed by the infrastructure provider usually). To understand the flow of FaaS workloads, it is important to be aware of the following properties of the platform.

2.2.1 Properties of FaaS

Statelessness Statelessness in deployments is a conscious decision that was taken during the conception of the Serverless infrastructure model to make the management of the platform straight forward and less cumbersome. Statelessness simply means that the applications that are to be deployed on the said platform exists as independent functions that are pure in nature. As in, the same data input given to the function always produces the same output at any point in time. This is what is termed as the lack of side effects. The data source and sink of the function can be any supported platform or tool as per the requirement, but there will not be any intermediate state or cache for the function. This means that the function at any execution will have no information about the previous execution unless explicitly specified.

The main advantage with this method for the infrastructure manager is pretty obvious. The fact that there are no volumes necessary to store any internal state means that the function can be scaled up and down independently and the whole infrastructure can stay elastic. Along with this, the provider can schedule the function in any node in the cluster that they use to host the application, move it around as per the usage burst, have multi-tenant deployments in a single machine ensuring the proper isolation for maximum profitability, and the list goes on.

In short, the notion of function is of prime importance in a Function-as-a-Service workload like the name suggests.

Triggers The functions that are hosted on a FaaS solution need to get triggered on a timely basis or based on an event. Usually most cloud providers provide more than a few ways to trigger the functions which the developer can choose from. Some of the most common triggers for FaaS applications are

- HTTP requests: An endpoint will be provided by the platform for the function that was deployed.

This endpoint can be called as an REST API endpoint and the event handler of the function will get the payload from the call.

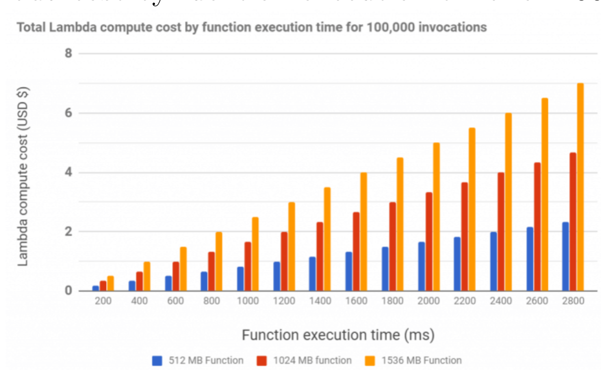
- Data arrival in a storage or data broker system: This is the most popular and heavily used triggering mechanism in FaaS. The idea is that the function gets triggered as soon as a new data arrives in whatever format at a particular storage setup. This can be arrival of a file object in the S3 block storage, arrival of streamed data in Kafka message broker system, etc. This method is the most suited for big data and streaming data applications since the function can be activated as soon as the new data is detected in the source. Usually the FaaS infrastructure provide supports more than a bunch of source storage to be used as the sources for the trigger.

- Cron: Another very common way to trigger function is based on a schedule. The

programmer can choose how often the function should be triggered on what days of the week, month, year, etc.

Billing One of the most attractive features of the FaaS service is the 'pay for what you use' policy. Billing model is an important constituent in the equation. Generally the commercial cloud providers charge you on the amount of memory that was reserved for the function, the execution time of the function in relation to the number of invocations that the function incurred. In most of the platforms, the developer can configure a maximum amount of memory that need to be dedicated to a function during its invocation. To save on the billing, if the user reserve less memory for the function, at the end of the day the execution time ends up being longer and there will not be much notable difference in the money spent [7] Figure 3 shows more on how billing varies as a function of execution time [8].

Figure 3: Lambda cost by fucntion execution time for 100,000 executions



When looking at the price per function invocation, currently at $0.0000002 for AWS Lambda and Azure Functions, it's very easy to get the impression that FaaS is incredibly cheap (20 cents for 1 million invocations). However, the price based on the number of invocations alone does not truly reflect the cost of providing this sort of service like mentioned earlier. With the current AWS Lambda price at $0.00001667 for every GB-second used (Azure Functions cost $0.000016 for every GB-second), you can see how the cost mounts quickly.

Since the amount of allocated memory is configurable between 128 MB and 1.5 GB, the total cost of function execution will vary depending on the configuration, and the cost per 100ms of the execution time for the most powerful specification will be roughly 12 times more expensive than the basic 128 MB option. Even with this it is easy to see that FaaS is a pretty cheap option.

If we compare this to an IaaS solution we can realize the fact that FaaS is not the right tool for all kind of applications. In the past couple of years, cloud prices has fallen that keeping up a small cloud instances all the time would cost comparable amounts. For example, the micro instance of EC2 costs $4.25 in average to keep it on for the entire month. In fact, simple math shows that running a tiny EC2 instance would be cheaper than having a function running continuously for the entire month. The saving comes up in the case of heavy yet variable load applications. In this case, if we reserve the memory needed at the peak load time, it is going to stay up with that capacity even during zero load which is very expensive and a huge waste of resources. And this is where FaaS shines.

2.2.2 How programming models are getting affected by this

Faas + Microservices In Software Systems Design, a widely discussed discussed topic is if to design the application in a monolithic fashion or a micro-services fashion. Monolith is the kind of design pattern where you have one big application doing multiple functions and maintained as one solid stack. On the contrary, when one designs their app in a microservices pattern, they will have to split up their application into multiple smaller parts which can be independently built and deployed, and yet working together with inter app communications. Both of these methods has its advantages and challenges. When monoliths are easier to develop and maintain, it can be very hard to test and manage due to the size, and usually if one part is buggy, it tends to break the whole system. On the other hand, microservices, since they work as independent units do not usually affect each others working and can be very easily tested and maintained. It is although often a very tedious task developing a system that fragmented and maintaining it that way.

With the advent of FaaS, a very interesting pattern has been adapted in the industry. The pattern pushed microservices one step further. The idea is that instead of having microservices that are available and on at all time, the huge applications are split up into functions that can be deployed to a FaaS infrastructure and triggered with the help of HTTP endpoints to act as a part of web application setup. This method is very effective resource usage wise and much easier to deploy and manage compared to vanilla microservices which have to be built and deployed independently. A very notable reason for mixing up FaaS and Microservice is that, Microservices usually embeds a local state alongside the application. This is usually one or more provisioned volume in the local file system. This is not possible in a purely FaaS based infrastructure. Deploying it alongside microservices offer a lot functionalities.

Statelessness or Functional programming model Like mentioned earlier, the notion of function is very important for the serverless platforms. It is intrinsically linked with functional programming. It is very interesting to note that Amazon named their FaaS solution Lambda which is a very basic concept of functional programming. Stateless clean functions that produce no side effect was objectively the perfect choice for an infrastructure solution of this scale.

What this change bought about is a thriving interest in functional programming languages. A lot of the functional programming languages belonging to the LISP family and some purely functional ones have seen a very increasing adaptation in the past few years in Serverless platforms. Since these languages are perfectly suited for stateless program it is only natural that they can be efficiently used to code for this environment.

2.2.3 Popular commercial offerings

Now that we have seen what makes FaaS an attractive field for cloud providers, developers, and researchers alike, it is interesting to understand the popular FaaS services out there.

AWS was the first big player in the field of Serverless introducing their platform AWS Lambda in 2014 [9]. Soon Google followed suite with their cloud functions and then Microsoft and IBM entered the game with Azure Functions and cloud functions respectively. In the past couple of years, Cloudflare [11] , Edge [12], etc. has started providing similar services but the former offerings still continue to lead the industry.

Although all the aforementioned commercial offerings contribute in strengthening the vendor locked in nature of the FaaS paradigm, it is worth understanding to see what kind of services a developer gets to have from each of these platforms.

The leading giants like AWS, Azure and Google tend to focus on configurability and ease of use. Their FaaS platforms are easily triggerable from their other cloud services, making it a very convenient yet monopolizing way of development. To understand the nature of the leading commercial service providers, in this section we go into looking at their characteristics.

AWS Lambda AWS lambda became publicly available in 2015 and currently dominates the landscape of AWS lambda. AWS Lambda has a free-tier under which it covers first 1M function requests and 400,000 GB-secs per month. AWS Lambda functions can be written in a handful of popular languages including Python, Javascript, Golang, C++, etc. The code is supposed to be bundled as a zip file and uploaded using API operations provided by AWS. One of the key issues that were noted often about AWS lambda at this point is the dependency management. The dependencies are expected to be bundled inside this zip file and there is a size limit to the zip. This is not a very great way to manage dependent libraries especially for data processing algorithms which deals with mathematical toolkits. Lambda provides guidelines for the way code and dependencies are to be organized in the zip file.

The idea of statelessness takes an interesting approach in AWS Lambda. We already saw how statelessness is a key aspect in FaaS platforms. To ensure that the corrupted caches are lying around, AWS do not have any extra garbage collecting processing. Instead it relies on the user not using any variables while writing the function. This is a very functional way of programming indeed but can be rather crippling when dealing with a lot of data. The way they suggest the developers take care of this is by using an external block storage like s3 to store these variables. The idea of AWS stepfunction was introduced briefly in the introduction section. For enabling state in a stateless architecture and orchestrate functions, AWS created Step Functions. This module logs the state of each function so it can be used by subsequent functions or for root-cause analysis.

Access management is managed by the IAM policies that are inherently used by AWS to manage access to any cloud service. AWS Lambda provides you with the facility to create your own custom IAM policies and attach them with your Lambda functions. This allows permissions for AWS Lambda API actions, users, groups, roles and resources.

Aws Lambda provides an API gateway and an HTTP endpoint to trigger the function in standard way. Other than this AWS support a huge list of AWS services that the developer can configure as the event source. Lambdas can also be invoked using the AWS SDK.

Another aspect worth noting is concurrency support and the execution support. AWS Lambda currently supports 1000 parallel executions of function instances and each function has a maximum runtime of 15 minutes. It is worth noting that concurrency often depends on the dependent resources that are used in the lambda function which may not be scalable by nature. AWS Lambda generally increases the number of concurrent functions running as soon as there is a rise in traffic. If there is no predefined limit they keep increasing it by 500 per minute until the demand is met.

Google cloud functions Google Cloud Providers entire the FaaS race very recently, in July 2018. Currently Google cloud functions do not support a lot of language runtimes. This includes NodeJS, Python3, Go and Java 11. The functions written can be uploaded to the service via the CLI, zip upload, inline editore, and cloud storages. So far Google cloud provides the most flexible workflow in dependency management. The developer just have to specify the dependent libraries in a package.json file and the cloud provider installs them for you avoiding the heavy package that needs to be uploaded like we saw in AWS lambda's case. This is really good because if the developer is building the package with libraries included in a Windows machine there will be huge incompatibilities for the package in the AWS lambda.

For state or for sharing data between functions google cloud recommends similar approach as of AWS Lambda, that is to use a cloud storage. The events for the trigger can be triggered by HTTP requests, and a bunch of google storage services like cloud storage, cloud pub/sub, cloud firebase, strackdriver logging, etc. Access control is managed in a similar fashion to AWS, by using IAM roles.

Google cloud functions really lags a bit behind when it comes to function orchestration. It does not offer any kind of orchestration mechanism that for the user to programatically chain functions via HTTP gateway.

When coming to the execution time, GCF have maximum hard limit of 9 minutes on this. The concurrency of functions in GCF is measured at a per function level that at an account level as opposed to AWS Lambda.

Fine grained scalability is not at its best yet on Google Cloud Functions. The functions are known to be scaled pretty slow depending on their size. It is seen to have a maximum cold start of around 500ms [10], which is in fact quite significant.

All in all Google Cloud Functions has to go a longer way to be a more flexible solution.

Azure functions Joining the world of Faas in 2016, Azure shines in a lot of places with its Functions where Google Cloud Function falls short. To start with Azure functions have a rich runtime almost comparable to AWS Lambda. They support a lot of very popular languages. Contrary to AWS Lambda, Azure Functions provides you with multiple options for deploying your function, such as GitHub, DropBox, Visual Studio, Kudu Console, Zip deployment and One Drive.

The dependency management in Azure is very similar to AWS in that, the system expects you to bundle all the dependencies together and upload it to the system.

In Azure, there is a tricky way to handle state by keeping static variables as cache data. Although if someone needs persistent storage they will have to use block storages.

While Azure Functions lets you control your function policies through Resource Based Access Control. It is supported at Subscription and Resource-Group. Though at the moment, you can give permission to read/write access both to your functions as read-only access disables some of the app's features.

As for the function triggers, Azure too supports a bunch of Microsoft services. But along with this, Azure lets you trigger the function using webhooks from Github, external HTTP, APIM, function proxy and bindings. For the orchestration, Azure functions provide Durable functions which basically is a bloated queueing service to pass event triggers between functions. It is a weaker form of AWS stepfunction.

The execution time is usually capped at 10 minutes. The number of concurrent activity is apparently 10x the number of cores in the machine. Azure Functions' free tier covers 1M requests and 400,000 GB-secs on the monthly basis. Afterwards, you will pay \$0.000016/GB-secs and \$0.20 per 1M executions. Azure functions have an embarrassingly long cold start period which is in the range of 3640ms on median.

2.2.4 Where Serverless computing fall short

Although serverless computing might sound like the silver bullet of the deployment solutions, it is a field that is still being rapidly grown and researched on. There are several staggering shortcomings for this technology that makes it unsuitable for certain applications. The current offering have the following noticeable limitations.

Lack of state As mentioned earlier, statelessness is a primary nature for serverless workloads making it easy to deploy and port agnostic of the environment and server. Hence serverless/auto-scaling paradigm generally push for a development style involving no state to make the infrastructure simple, encouraging a functional style of development. Although this can contribute to easily scalable and parallelisable applications, it often limits the technology from being adapted in applications that are data intensive. The fact that serverless functions do not store any intermediate state requires the application developers to use a block storage to store the data and state after the execution. This basically means communication via slow storage and adds a lot to the latency. This discourages the use of serverless in distributed computing which is actually a domain that needs very fine grained communication between the functions and usually a lot of resources are wastefully dedicated to ensure high availability.

A function during execution has no clue of the previous executions and its results. Which is something that is usually very basic for data analysis operations. The developers in this case are forced to send the data after each execution to a block store and retrieve the data from the block store before the next execution. Other than the input output overhead and the network latency this adds, it is a violation of the elastic nature of the Serverless paradigm.

I/O Latency Like was mentioned earlier, FaaS have had a lot of influences in the system architecture and programming paradigms like would with any new infrastructure management system. It is quite unfortunate though that, even with a paradigm with such huge potential, FaaS is very conventional when it comes to its data engineering architecture. Functions are run in isolated units separate from the data or data store. This is actually a very huge system design anti-pattern because Input/Output have and will remain to be a bottle neck even with heavy memory and huge number of dedicated cores to a function. The pattern where the data is taken to code as opposed to code to data adds to the latency, cost, and inconvenience. For the clarity of the reader, an example of a code shipping architecture is procedures that you run in databases. The code is moved to the data than the other way around in this.

Coordination issues among functions FaaS workloads are usually containerized by the cloud provider to deploy it easily in their node pool or cluster. By nature, docker containers are indiscoverable units that need to be opened up explicitly to the network of the host machine. Meaning that, we cannot explicitly address the docker container directly using an IP address or an endpoint. Cloud providers do not open up the container to the network consider the potential security issues this can cause and the necessity of state in this case. They provide handles to communicate with the function or trigger-able entry points, but no direct network addressability.

What this implies is that, if the developer has multiple functions that has to be composed together to form a pipeline, rather than triggering each other internally and directly, the developer will have to hack around by either triggering it via an HTTP endpoint if the provider allows that, or like was mentioned in the previous point via an external block storage, or other external queueing systems they provide, etc. In either of these scenarios, it is hard to avoid added latencies.

This makes FaaS particularly inefficient for applications like distributed computing when it depends on very fine grained communication between the functions. With FaaS we can only ensure very weak consistency across function storages making it a pretty bad candidate. What this also means is that there is no way we can actually have efficient parallelism even if we have many powerful cores installed over the current state of FaaS since the block storage will always be a bottleneck.

It goes without saying that most big data applications that need ephemeral storages between function executions suffers from the very similar kind of latencies as well. This includes function compositions like ETL on streaming and batch data alike [13]

Vendor lock-in It is no secret that the most widely used FaaS/serverless offerings are the ones by proprietary cloud providers where they hand twist the developers into complying to their programming environment and runtime thereby forcing devs to use their technologies. What such practices contribute to is limited innovations and development around the paradigm of Function as a service itself and people re-inventing the wheel by creating custom made code and hack to fit each of these provider runtime.

In a system like FaaS, where you are basically out-sourcing the whole setup of your application to a vendor, the fact that the whole ecosystem is closed source and uses the tools developed by the vendor only means that the user has near to zero control over the infrastructure and the pipeline is not transparent at all for any kind of performance optimization or fine tuning.

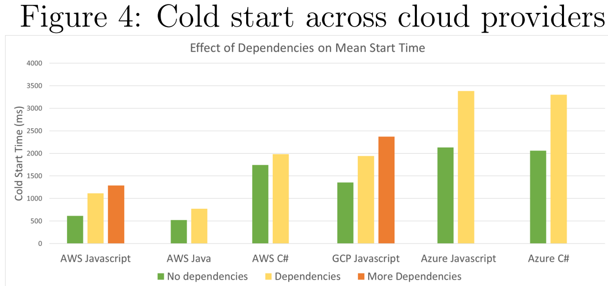
Fixed timeouts This is the one of the other bigger reasons that hinder the usage of FaaS in big data applications. In applications that involve heavy number crunching algorithms, there are chances that often the function needs to run for a longer period of time. Current commercial FaaS offerings has a fixed timeout, exceeding which the function execution is automatically terminated irrespective of the stage of the execution. The fact that the platform offer little to no control over this discourages the developers to use the tool.

Currently the maximum timeout for function execution in AWS and GCP platforms for the FaaS setups are 15 minutes and for Azure functions it is 10 minutes. These are all extremely bounding as conditions especially for functions that are composed and a function should wait for the other functions to finish executing.

Cold Start Cold start it the delay that the function incurs after the invocation or triggering of the function till the execution of the function. In the background, FaaS uses containers to encapsulate and execute the functions. When an user invokes a function, FaaS keeps the container running for a certain time period after the execution of the function (warm) and if another request comes in before the shutdown, the request is served instantaneously. Cold start is about the time it takes to bring up a new container instance when there are no warm containers available for the request [14]. In most platforms serverless latency on average is measure to as 1-3 second [15], which can have very dramatic impacts when it comes to certain workloads. According a 2018 survey, this is the third biggest concern developers have regarding the serverless platform [16].

The cold start time in-fact is overblown by several factors in the infrastructure. All the popular commercial FaaS offerings suffer from a cold start time. It can referred that irrespective of the language runtime used, the start time tend to be almost the same on a platform. The main deciding factor is the dependencies that were packaged for the application which obviously makes the container slower to start because of the heaviness. Figure 4 shows the cold start time differences across different commercial cloud providers under different runtime and different dependencies.

Figure 4: Cold start across cloud providers



A solution for this problem, other than keeping the dependencies small, is to have a warm function up at all times so it can handle the request right away for time sensitive applications. The problem here though is that most commercial offerings do not offer this option. Instead the developers are forced to keep pinging the function to keep it warm for the next trigger. This is a very hacky solution and reduces the whole efficiency of the platform in general. Most of the cloud providers are although aware of this problem and are trying to be innovative and introduce lighter alternatives to Linux containers in the FaaS platform these days.

Parallelism Current FaaS offerings are not known to have the right support for heavily parallel computations. In the most popular commercial platforms, the presence of cold starts delays some invocations and increases the runtime. This impedes the parallelism. In case of multiple simultaneous requests, maximum parallelism that was achieved in handling them on average was less than 50% [17] in Google Cloud Functions. The reason for this can be further elaborated as follows:

- Virtualization technology: If a FaaS system has to run multiple functions in parallel when triggered, the most import thing that comes up is the ability of the platform to boot up more instances of the function instantaneously. The quickness of the creation of the instances depends on the virtualization technology that is being used. This is basically the cold start latency that is affecting the parallelization. For example, if Docker is used as the virtualization technology the system is seen to have a bit more latency, but if a virtual machine is used the latency goes up exponentially. This calls for the need for more lightweight isolation solutions. AWS firecracker is a step in this direction.

- Reactive scheduling: In FaaS systems the kind of scheduling that happens is extremely reactive. Reactive model s seen to be too slow to scale. Achieving high levels of parallelism requires being able to provide resources rapidly. So how the system deals with the incoming invocation is very important. It is seen that the current event based triggers are less that optimal for such applications. This calls for a proactive approach in dealing with invocations. It could be a more push based approach as opposed to the former.

Security issues in a multi-tenant environment Like was previously mentioned, the whole FaaS infrastructure offering is economical for the cloud provider because they get to share their node pool among all their standard customers making the resource cost for them very low. The problem with this practice though is that this introduces safety issues for the data that is executed in the machines. Linux containers are not particularly secure as an isolation mechanism since they share a Kernel with the host operating system. This means that any bug or back door introduced to the Kernel get affected to all the containers as well exposing the customer data at a very high risk. This is an issue that is actively being worked on by companies. Till a while ago, the solution for this was to encapsulate the containers in a light weight VM which unfortunately contributed to the heavy cold start time. But recently the innovative new alternatives for Linux containers are also aimed at to fix these issues.

Function caches Along with the above mentioned issue with multi-tenancy across customers, a similar issue can occur under the same customer who runs an application across multiple of their client. The problem is that each function has an inaccessible cache that get cleaned up at an arbitrary time hidden from the user. There is a chance that somehow cache from the previous execution of the function somehow lingered and the data from one client got leaked on to another or got corrupted by the other. If the developers are not cautious enough while coding and usage of variables, there is a high chance for data corruption and leakage on the platform.

Developer friendliness In a recent survey [16], developers were asked about the challenges they face when using Serverless platforms. This is a very significant data to look into since at the end of the day the gap of the research and the end user experience is something we are trying to mitigate with this project. The following were some key takeaways from the study.

- Debugging and testing: Even though FaaS setup modularizes the code a lot, when we consider most commercial offerings of FaaS, there is low to zero possibility to actually follow the conventional testing and debugging methodology. It is mostly because of the fact that the runtime of the FaaS environment is not known to the developer at the time of the development. Along with this, by the sheer nature of FaaS, it is often hard to mock exactly the events like would in the production setup locally. So a full functional testing of the platform is often pretty difficult to make happen.

More than often the developers have to depend on deployed setup of the FaaS function and try debugging on production. This costs resources and on issues involves re-deploying it and testing again. This has a huge impact on the productivity and slows down the whole development workflow.

- Logging and monitoring: Most of the current commercial platforms asks the developer to user an external tool like AWS cloudwatch which costs more for this service. Considering logging is the only way to debug the function, it becomes a bit of an inconvenience if the developer is expected to pay for it. As for the monitoring the same story applies. For each metric that is being tracked extra is expected to be paid. If one is composing the functions, it gets even more difficult to understand the cumulated runtime monitoring along with the transfer details on the block storage, if any.

- Standardizing development practices The problem basically boils down to this one tag. The idea is that each of the FaaS operator has a different kind of interface or way of dealing with the events hence introducing a lack of standard dev practices. The problems are more so prevalent when it comes to the building and deployment of the function since the user management and the CLI access to do deployment are all delegated to external tools.

2.3 Extract-Transform-Load(ETL) pipelines

In the previous sections, we talked about how serverless is the most suited but inefficient(with the current state of art) tool for ETL pipelines and that it is a standard practice when dealing with today's data driven workloads. In this section, we look in detail into the characteristics of ETL workloads and their applications.

ETL is the type of data integration process that is used to process data from multiple sources to build a Data Warehouse or similar sinks. It integrates three distinct but interrelated steps namely Extract, Transform and Load.

The main advantage of having ETL pipelines in the splitting of functionalities in the data processing programs that would have otherwise been a single huge monolith - hard to manage and extremely bloated.

Extract In the present day production workloads, data can be arriving from numerous sources of very varied nature and behavior. Depending on their origin source, this data can have different formats or organizational structure. Some examples for this are relational databases, XML, JSON, flat text files, etc. To allow scalability in a software solution it is always necessary to have the tool working for multitude of data types. This is where having a dedicated extract process shine. Extract process accepts data from any of the data source and format it to a unified data type. In general, in extraction phase developers try to convert the data into a single format that is understandable by the transformation phase. Another import thing the extract step take care of is the validation of the data. The data that is coming from the source can be of the wrong format as expected, even missing some columns or corrupted somehow. In the extract step, these bad data are reported and the process is aborted.

Transform This step can be considered as the brain of the whole workflow. This is the stage where we convert the data that was received into meaningful information. Transform operation often happen in multiple stages where in each stage a certain transformation logic is applied to the data. These logic can be simple text formatting steps like splitting the data, cleaning the data, deduplication, replacing codewords with meaningful entities etc. or more complex arithmetic or logic operations like machine learning models. This step often ends up being the bottleneck in a lot of ETL pipelines since data processing can be a very resource heavy task and if the code is written with no optimization, the whole pipeline will end up eating up the all the resources.

Load Like the name suggests this is the step where the data coming after the extraction and transformation processed get loaded into the target data store. This is a rather interesting process because the nature of the target store and the communication API need to be analyzed efficiently to write the code. The code often contains certain validation parameters to see if the current data is suitable for insertion into the target. Common problems that occur might be format difference from the target, duplication of primary or foreign keys, other integrity violation issues, etc. Monitoring is critical in this step since there is a chance that the target endpoint can go unavailable and the developers have to make sure the data after the entire processing is not lost. More than often the data output overwrites the existing data in the source, in other cases the new data gets appended to the older ones or aggregated with the existing data.

2.3.1 How Serverless can make a difference in ETL

Let us now look at some of the characteristics of the ETL processes that can be derived from the definition.

- Atomicity of functions: In the pipeline, if a failure occurs in any of the above stage, it should stop from proceeding to the next stage for that data without disrupting the pipeline that is dealing with any other batch of data.
- Each stage can have different loads depending on the operations and these stages should be able to independently scale without scaling the other stages. This helps in saving resources. Along with that it also saves from making any one step a bottleneck because of low resource availability by scaling the resources up as per necessity.
- Intermediate data transfer need to handled properly by the system. We need to have temporary data between each of the processes stored temporarily. This is to make sure that if one of the stages fails, the data the came out of the previous step is still available which can be processed again after the developer fixes the issue with the current stage. This means that at any stage you can restart the system and the pipeline can continue without issues.
- There should be proper monitoring support make sure that we can easily see when the errors are happening in the system. Along with that, the performance of the pipeline should be quantifiable. We should be able to tell which stage in the whole pipeline uses more resources, etc. and the overall performance of the ETL workload.

It is clear from the above description why Serverless might be a good fit for ETLs considering its elastic architecture and functional style of coding. Each stage in ETL can be separate functions than can be independently scaled and monitored. Although the aforementioned issues with the Serverless makes the function composition and data transfer quite inefficient making it an ill suited candidate for ETL applications. Also in the current Serverless solutions, it is hard to achieve the atomicity of functions.

2.4 Problem statement

From the above set of evaluations, there is no doubt that Serverless Computing is a solution with great potential as a tool for deployment and infrastructure provisioning. Even with the current state of art FaaS offerings, 21% of the entire workload is Data processing applications that include heavy batch and streaming Extract, Transform and Load operations [16]. However, the implementation usually involves numerous hacks in this setup, even after which the latency of the I/O, network and the platform itself slows from leveraging the full potential of the idea. All the existing commercial offerings being closed source and vendor locked in, implies that the limitations are set for you by the cloud provider and is often very difficult to fiddle with it or to extend the system so as to support an extra runtime, increase the running time, etc. Along with this, the way current FaaS offerings deal with function compositions and parallelism are extremely clumsy and almost always explicit. While this lets the providers have a very generic way of dealing with the platform and holds to the one way to code them all paradigm, the gateways often tend to be a bottleneck. Also the data transfer between functions always depend on a storage based off of Block IO which contribute to the latency immensely.

When we look at the academic research in and around this area, in the past couple of years a handful of ideas has been thrown into introducing state in serverless. A very interesting proposal was Cloudburst [2] which introduces a consistent cache storage between functions to store and retrieve intermediate data in wire speed. Although the project succeeds in proposing a very elastic system architecture that co-locates data alongside functions across the cluster, it is seen that the system does not scale really well along with the requirement making it a bad adaptation for streaming and big data workloads. Alongside, it lacks provisioning to define branches or conditionals in the function composition making it less flexible from the point of view of the orchestration.

A similar idea was SAND [3], where a hierarchical message bus is used to allow function composition and inter process calls. Other than being a closed source project and pretty abandoned in the past couple of years, the resource allocation is not tracked or controlled by the system breaking the per usage billing notion of Serverless paradigm. The system also does not offer a proper isolation mechanism.

The focus of the thesis is mostly to propose a solution for the aforementioned issues. We are proposing a Open Source infrastructure, infrastructure that can be maintained by the companies which can offer a multi-tenant and completely elastic platform to deploy their data intensive and high throughput applications on. By nature, these data intensive applications can be a composition of multiple functions, that would pass along data between them. The setup would user ephemeral in memory storage to keep intermediate data. This infrastructure would comply perfectly with the notion of Serverless in the sense that, each element in the system would be independently elastic and scalable. Function composition based on conditionals and branching should be supported by the system along with independent scaling of the functions based on the load, so there would not be any bottlenecks. An easily adaptable programmable API is required for defining this composition.

According to the aforementioned survey, the developer community is concerned about the monitoring and debugging of the functions during the development stage due to the lack of reproducability of the runtime. Our system should give a lot more flexibility and traceability when it comes to the development process. Along with that, we should aim at building a system that is easily adaptable and stable enough for production workloads, and easily integratable with the common development tools like Github, CI/CD pipelines etc.

3 Proposed Solution

In this section we dig in deeper into the specifications of our proposal to build a production ready FaaS infrastructure stack that is completely elastic and not locked into any vendor. The idea is that, any party or enterprise should be able to reproduce this stack easily and developers should be able to deploy their application code from any git hosting service or command line to this platform without worrying about the server management. The platform we build also should be provider agnostic, in the sense that it should work with constant efficiency on any cloud provider the user may choose. The developer should be able to monitor the usage and performance of the application easily.

In the light of the above discussion we propose the following extensions to the existing Serverless platforms:

- Provision to compose functions by defining a computational graph
- Ephemeral in-memory storage to store intermediate data
- Multi-tenancy support by separating function instances using namespaces
- Fine grained tracing and monitoring of the functions and the compositions

To clarify how the above mentioned steps will help solve major limitations of Serverless paradigm, we will have a platform agnostic look at how the above steps change the current state of art FaaS systems. In the section 5, we will get into platform specific study by implementing these suggestions on a flexible open source FaaS solution for our proof of concept.

3.1 Function composition

Big data processing is pretty inevitable as an application scenario. The nature of these data can be very varied including streaming, semi-regular burst streams, etc. making it a very good space to apply Serverless paradigm to, to save up resources and have fine grained scaling of the resources based on requirement. The aforementioned complexity in the application logic suggests that it make a lot of sense to split the application into multiple functions and compose them efficiently. If applied to the Serverless logic, this means that each function can be scaled independently based on the load in that logic.

The above requirement exposes some issues that were discussed in the section 2.2.4 of FaaS. Function composition is not something that has been cleanly supported by popular commercial FaaS offerings. The popular infrastructure today do not have any information about the dependencies between multiple functions. It is up to the developer to programatically call functions from each other which are packaged and deployed separately. If there are any heavy data to be transferred among these functions, which we can refer to as intermediate data, the developers are expected to use a block storage of some sort(eg: S3, google data store, etc.) adding heavily to the Input/Output latency of the service, not to mention the network latency if the infrastructure is in a different VPC.

In a recent case study [18], Autodesk claims their FaaS-ification of their whole platform. Unfortunately, their account creation platform, which was implemented as a composition of multiple small functions on AWS lambda incurred a round trip time of 10 minutes. This is horrendous especially considering the vitality of the task in discussion. Overhead of Lambda in task management and the state management is explained as the causes.

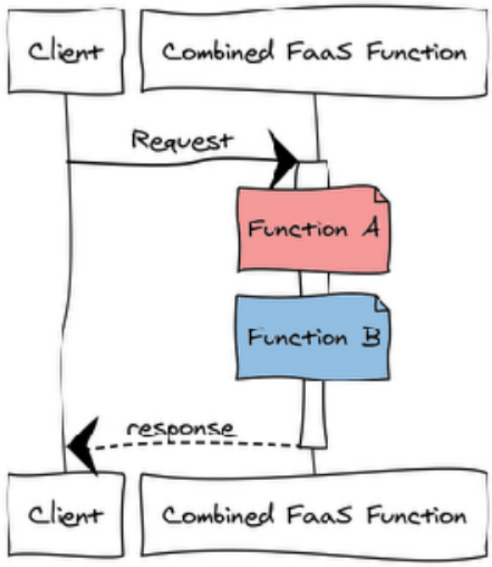
More products has been introduced by cloud providers, like AWS step functions [19] , instead of fixing the inherent architecture of FaaS solutions to help create data intensive workflows in FaaS. These systems work by introducing an event queue like AWS SQS to the equation. The problem with such solutions is that they violate the notion of Serverless in a way by introducing an element that is practically non scalable and cannot be debugged easily. It becomes extremely difficult to develop and test the system locally. Not the mention, the fact that this introduces more lock in to the vendor.

Another approach can be found here [20], where the function composition is done by triggering the other functions by pushing intermediate data to s3, which the following function considers as the trigger. The example in question is a very simple map reduce which is not very intensive computationally even with a heavy load of data. Even with that the setup takes around 2 minutes to complete the task for a dataset of size 25GB. It can be seen that the majority of the running time was spent on pushing and pulling data and not on the compute.

It is quite clear that the ability of functions to call each other are rather important. There should be a way to define programatically the relationship between the functions in a FaaS infrastructure along with the data flow dependencies. If cloud provider exposes an API that would let the developer feed a computational graph for this function composition, this would not just improve the performance, but also would be useful for better function and data placement so the latency for data and control transfer would be minimum. This can be a very tricky thing conceptually since, containers are not directly addressable network wise.

Before getting into the technicalities of the platform itself, let us look at different approaches in which functions can be composed in a serverless workload.

Figure 5: Merged in the source code



3.1.1 Manual Compilation

This the most basic and inefficient way of compiling the functions. This basically involves merging all the functions together to form a huge function. From FaaS executor's point of view, it is one big function.

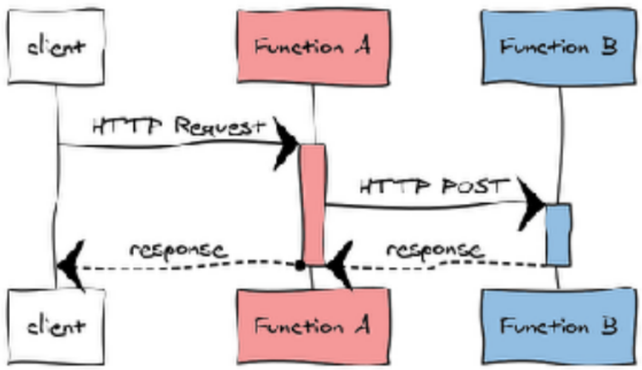
```
def funcA():
  doStuff()

def funcB():
  doStuff()

def main():
  funcA()
  funcB()
```

The above code block and Figure 5 explains how the control flow works in this kind of compilation scheme. As is pretty obvious, with this method one cannot scale individual functions independently and function can get really

Figure 6: Direct function chaining



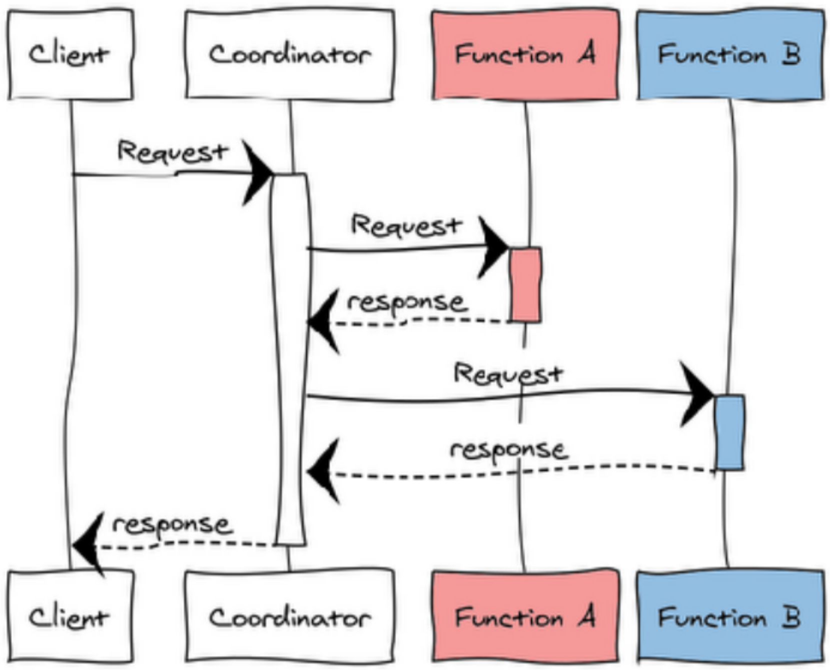
big. There is no necessity to store intermediate data or serialize and deserialize data between functions. But the problem is that this kind of violates the notion of serverless since each application is not an atomic functional unit. If the compute is complex, function might not even completely run because of the hardbound limit to the running time set on most FaaS platforms.

3.1.2 Direct function chaining

Like can be seen from Figure 6, here each task is a separate function. Each function directly call the succeeding function in a chain. Meaning the code is written so that the current knows the details of the next function, but not any further. Even here like before, there is no need for any serialization deserialization overhead since functions can directly send each other data. No external components are used either. Although the problem arises when the data load increases. The load on the network to transfer data via HTTP rises. Along with that each function will have to wait for the next function. If a function fails then the logic to retry/fallback etc. will have to be coded into each function. The following pseudo code shows how the function design would be.

```
def funcA():
  doStuff()
```

Figure 7: Coordination functions



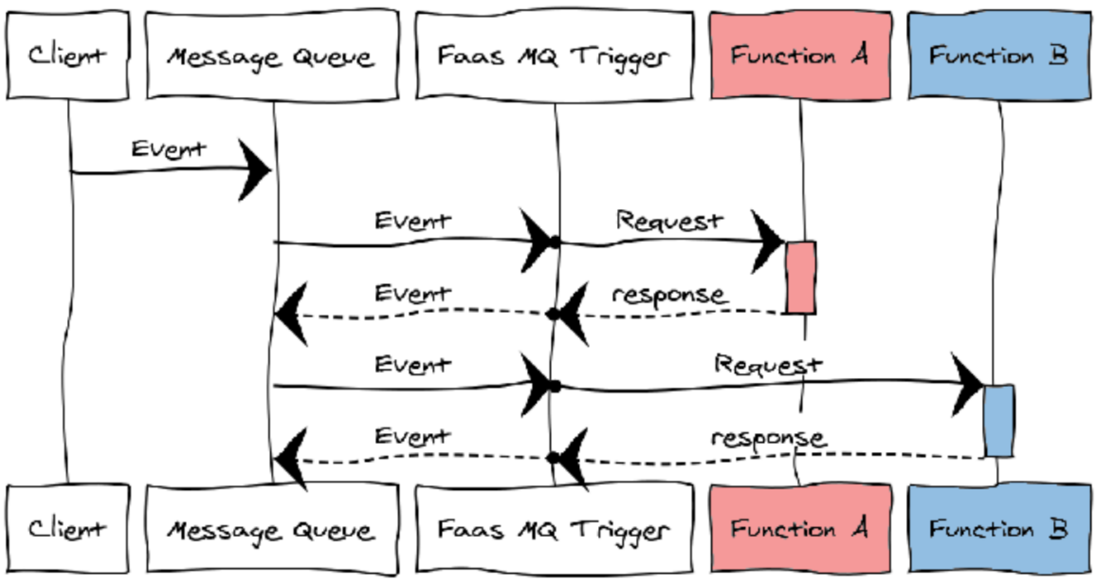

```
def funcB():
  doStuff()
```

3.1.3 Composition via coordinator functions

In this method, a coordinator function will be used which manage the execution of all the functions by calling them directly. The individual functions will be unaware of each other.

The win over the previous method here is that, the house keeping code need not be present in each individual task. Also it is very flexible in the sense that, each function can be tested independently and then the user can properly write the control flow in one place, that being the coordinator function. This comes at a cost of adding an extra function which is the coordinator function. This function will continue running the whole time, costing more

Figure 8: Event driven function composition



and violating the FaaS paradigm a bit. An example of this kind of coordination can be found here [21].

3.1.4 Event driven composition

This is a powerful design pattern that supports a lot more fault tolerance and involves changing or extending the infrastructure of the FaaS platform. In this method, one introduces message queues in the architecture as can be referred from Figure 8. Functions emit events to these message queues. Alongside, all the functions listen to the same queues. So on receiving certain events, they react in the programmed ways. Contrary to all the previous methods, it is very interesting to note that in this method, the stress is given to the data flow instead of the control flow among functions. The intermediate data between the functions has to be managed separately by using a storage.

This is a very commonly used and popular architecture. Message queues like Kafka or MQTT brokers like rabitMQ offer a lot of functionalities and features like fault tolerance, error handling, alerting, backup, etc. Functions can be completely decoupled. This is a very good solution for big data and streaming data applications.

The problem with this method is though the very heavy dependencies which are very hard to manage. The fact that message queues are not inherently serverless makes the platform less elastic and thereby billing and usage tracking can be troublesome of the infrastructure manager. Alongside, message queues usually only supports limited control flow structures. Probably just conditional and on-error handles. It will be terribly complicated to do dynamic branching, iterations, etc. Along with this, since functions are so tightly dependent on the message queues, it will be slightly challenging to upgrade or version them.

3.1.5 Workflows

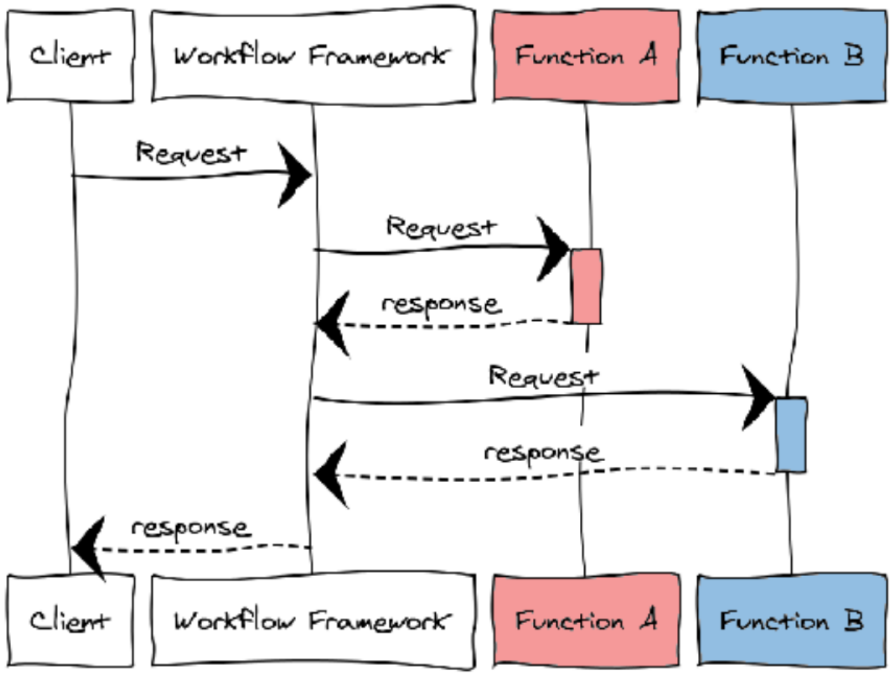
Workflows are a very interesting architectures pattern where the system supports the creation of a sort of flowchart of the functional interaction. Workflows are a very widely used pattern these days in a lot of big data processing tools.

An workflow is designed as a directed acyclic graph (DAG). This means that a new runtime has to be introduced in the FaaS system to manage the execution of the functions. When authoring a workflow, one should think how it could be divided into tasks which can be executed independently. The workflow runtime would let one to merge these tasks into a logical whole by combining them into a graph.

This definitely adds the overhead of writing a runtime for the FaaS platform, providing an API to define the DAG to the runtime and then managing and executing the workflow based on the DAGs. But once the platform is in place, it provides numerous flexibility. One can get done dynamic branching, iteration, etc. very easily on this platform along with individual upgrade of the functions. The fact that no external infrastructure tool has to be managed to work as a triggering mechanism maintains the elastic nature of the tool. The only thing is that there has to be a storage unit to manage the state of the DAG for the workflow framework. Similarly just the event driven composition, the intermediate data store has to be handled separately.

Logically, this method resembles the coordinate function setup, just that

Figure 9: Workflows

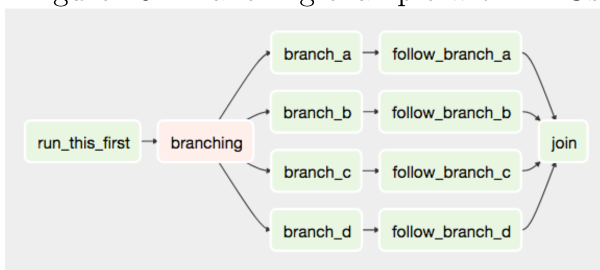


instead of a simple coordinator function, in this case we have a month more powerful framework that is added permanently to the infrastructure. This can be referred from Figure 9.

The shape of the graph decides the overall logic of the workflow. A DAG can include multiple branches and you can decide which of them to follow and which to skip at the time of workflow execution. This creates a very resilient design, because each task can be retried multiple times if an error occurs. To give the reader clarity on what a DAG looks like, the Figure 10 from the Airflow's operator might shed some light.

With this setup, we can get a lot more centralization to the compositional logic, making logging and visualization lot more easier. With this method the function scheduling and placement can also be improved. Meaning, functions that have compositions with each other can be scheduled in the same node, if we have a cluster or the intermediate data can be placed nearer, etc. One downside to this method is that the user will have to use the workflow

Figure 10: Branching example with DAGs



specific language or DSL and not just the programming language used for the function implementation.

It is arguably clear that workflows offer the most flexible and application independent solution as a composition pattern. Of course the concern of having a storage for the running state of the workflow framework remains along with the storage of the intermediate data. We will look into the solution to this in section 4.2.

3.2 Ephemeral Storage

In the previous section, we saw that flexible function composition can be achieved via workflow pattern. However, efficient state storage is necessary to make this efficient. The problem is that we have to not violate the notion of elasticity when it comes to Serverless. The resources involved in Serverless should be scalable up and down, only when we can have a per usage payment and resource conservation. Scaling up also affects the availability of the tool since one should be able to have all the requested served without much latency. Along with storing the state of the workflow or DAG, if function has to pass around data from one function to another, we should introduce some sort of intermediate storage since there is no direct communication between functions. The workflow framework take care of triggering each function based on its state and the data transferred between the functions will be via this intermediate storage as well.

In traditional analytics framework, long running process in nodes takes care of managing the intermediate data in local storages. On contrary to this conventional approach, Serverless workloads do not have any long running processes. Because of the network addressing problem of containers, direct transfer of data is also pretty impossible between functions.

In all the commercial service offerings of FaaS this intermediate storage is done via a block storage like S3. This is a very inefficient approach since a block storage adds a lot of I/O latency to the system. Along with that, it adds a non scalable entity to the equation. Conventional storage systems are not designed to meet the demands of serverless applications in terms of elasticity, performance, and cost. We are talking about data that has limited life span, which we can refer to as ephemeral data [22].

Traditional storages like RDBMS, NoSQL, block storage, etc. are not made for short lived data because of the latency involved in writing to the disk. An in-memory key value store seem like the most obvious choice. But unfortunately the industry standard key value stores like Redis does not scale very easily. One has to take care of the scale of the storage cluster, network configuration, maintenance, etc. Per use billing can also be very tricky in this case.

We should be looking into innovative new ideas to use for serverless platforms when it comes to data storage because of the ephemeral and scalable nature of it. Since Serverless functions are deployed on clusters that exist across multiple nodes, a distributed key value cache that is scalable is the desirable option we are looking for.

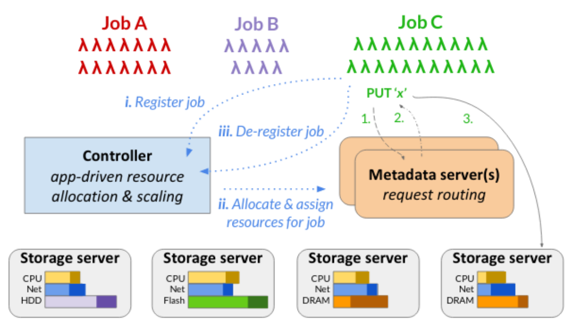
In our preferred storage medium, we should have automatic scaling, fine grained usage tracking & billing, low latency, high throughput, low cost, and unlimited availability. Key value stores like Redis and memcache offer low latency and high throughput but at the higher cost of DRAM. They also require users to manage their own storage instances and manually scale resources [22]. We look into two different storage solutions for the adaption to our FaaS extension: Pocket and Orlic

3.2.1 Pocket

Pocket [22] is an ephemeral storage build for the Serverless workflows. It is a key value store suited for storing and exchanging data between hundreds of fine-grained, short-lived tasks. Pocket is an elastic distributed storage service for ephemeral data that automatically and dynamically right sizes storage cluster resource allocations to provide high I/O performance while minimizing cost. Pocket is not completely an in-memory storage infrastructure like expected. Instead, pocket has a smart data allocation system that leverages different storage media(DRAM, Flash, Disk) to store the data depending on the requirement of the application while minimizing the cost.

Pocket has a tiered architecture. It has three planes - A control plane, a meta data plane and the data plane. Like the name suggests data plane stores the data ultimately. Meta data plane tracks the presence of the data distributed across this data plane. Finally the control plane manages cluster scaling and data placement. This layer keeps the platform elastic, in that it scales the storage resources based on the usage. Each of the aforementioned layers can scale independently. The project claims to have a sub-millisecond latency for I/O operations.

Figure 11: Pocket system architecture



Architecture Like Figure 11 represents, Pocket system has one centralized controller server, one or more meta data servers, and multiple data plane storage servers. The meta data plane according to us is the most interesting in the architecture, since it enforces coarse-grain data placement policies generated by the controller. It manages data at the granularity of blocks whose size is configurable, defaulted to 64KB. Objects larger than this size is divided into blocks and are distributed across storage servers by the meta data server. Client access data blocks directly from storage servers.

Client API Pocket provides an API to communicate with the system. There are system calls to each of the three planes. First of all it lets the client register and un-register of the jobs(control plane). The client gets to communicate with the meta data server multiple times during its lifetime.

Figure 12: Pocket Client API

Client API Function	Description
register_job(jobname, hints=None)	register job with controller and provide optional hints, returns a job ID and metadata server IP address
deregister_job(jobid)	notify controller job has finished, delete job's non-PERSIST data
connect(metadata_server_address)	open connection to metadata server
close()	close connection to metadata server
create_bucket(jobid, bucketname)	create a bucket
delete_bucket(jobid, bucketname)	delete a bucket
enumerate(jobid, bucketname)	enumerate objects in a bucket
lookup(jobid, obj_name)	return `true` if obj_name data exists, else `false`
delete(jobid, obj_name)	delete data
put(jobid, src_filename, obj_name, PERSIST=`false`)	write data, set PERSIST flag if want data to remain after job finishes
get(jobid, dst_filename, obj_name, DELETE=`false`)	read data, set DELETE `true` if data is only read once

The data in pocket is stored as objects that goes in buckets. They are identified using names. Meta data plane provides system calls to create and delete these buckets, look up objects and delete these objects.

Client put and get data directly to/from the object at a byte granularity. The put and get operations invoke the meta data layer with the Job ID of the client. This is to do the meta data look up operation to get the data placement of the object that is being looked up. When a put call is invoked, with a PERSIST flag to be true, the object will remain in the data even after the job terminates despite the ephemeral nature of the storage. It will remain until it is explicitly deleted or after a configurable timeout period. The get call with a DELETE flag set will get deleted right away after returning the value of the object. The nature of the ephemeral storage in discussion is assumed to be write and read once only. Figure 12, describes the system calls in detail.

Implementation

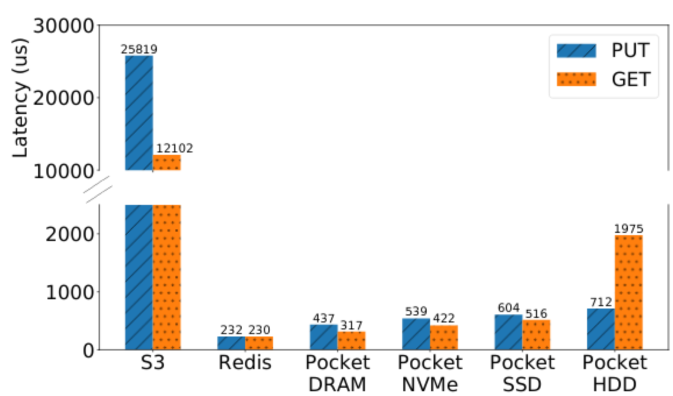
1. Controller: Pocket is run on Kubernetes with each layer as separate docker containers. A resource monitoring daemon is run on each node

in the cluster sending resource utilization info to the controller. The controller right sizes the cluster by launching new nodes and sending the info of the existing meta data servers to it. The load is balanced using data steering new active job data to the newer server than balancing out existing data since this can add a heavy overhead especially since the data is short lived. The container also keeps the meta data server resource usage under the target limit by precalculating the load a job would put on the meta data server from its throughput and capacity allocation. Based on this estimate the controller select the meta data server.

2. Meta data and Storage tier: These are implemented on top of Apache Crail distributed data store [23]. Crail is designed for low latency, high throughput storage of arbitrarily sized data with low durability requirements. Crail provides a unified namespace across a set of heterogeneous storage resources distributed in a cluster. Its modular architecture separates the data and meta data plane and supports pluggable storage tier and RPC library implementations. As of the storage tier, Pocket project implements it on DRAM, NVMe on top of ReFlex and then on generic block storage.

3. Client library: The API is written in Python to provide better adaptability of the tool. The core library although is in C++

Figure 13: Pocket Performance for get and put requests



Analysis Pocket is seen to have pretty good performance almost comparable to Redis but much better economically when set up on DRAM. It is seen to be almost 300% faster than S3 storage for the GET requests. It can be seen from Figure 13. So considering that DRAM will be used as the storage tier, it can be the right tool for the ephemeral storage in Serverless platforms.

3.2.2 Olric

Olric [24] is a distributed in-memory key/value data store. The idea is that we can create a shared pool of RAM across a cluster of computers to store the data in, in a scalable manner. The design motives for Olric is to share between servers fast-changing transient data. At the time of writing, Olric supports multiple serialization formats including JSON, MessagePack, etc. By utilizing the heuristics of Kubernetes, Olric provides horizontal scalability to the RAM pool available. Olric uses a consistent hashing algorithm [25] to distribute the load fairly among the cluster members. Olric best-effort consistency guarantees without being a complete CP (indeed PA/EC) solution. This thread safe in-memory cache comes with replica support and a command line interface.

The data is stored inside distributed maps which can be thought of as a bucket. Inside each distributed map, there can be numerous key - value pairs. As for the operations, currently Olric support atomic operations and the lookup has a complexity of O(1). Olric uses SETNX algorithm to implement locking primitives inspired from Redis protocol [26]. Olric can be used either as a Go library or as a language independent service.

The architecture of Olric is rather sophisticated. Olric distributes data among partitions that are distributed across the cluster using the consistent hash algorithm. Every partition is being owned by a cluster member and may have one or more backups for redundancy. When a distributed map entry is being written, the communication is to the partition owner. In a stable cluster, the query hits the most up-to-date version of that data entry. In order to find the partition which the key belongs to, Olric hashes the key and mod it with the number of partitions.

There is an elected coordinator in each cluster. The coordinator election is done via a very simple heuristics. All the machines share their birthdate in the cluster. The oldest machine gets elected as the coordinator. When the coordinator leaves, the second oldest gets elected. It manages the partition table. This can involve registering new partitions if a new machine joins, removes outdated data, pushes new partition table to all the members and to the cluster.

There is a rebalancer binary running in each node that takes care of relocating the partition from the backup to the new host when one of the hosts leaves. Along with this it merges fragmented partitions.

The idea of fragmented partitions are rather curious. Each partition has an owner. There can be multiple owners in which case the partition is called a fragmented partition. The last added owner is called a primary owner. Write operation is only done by the primary owner. The previous owners are only used for read and delete. When you read a key, the primary owner tries to find the key on itself, first. Then, queries the previous owners and backups, respectively. The delete operation works the same way. The data(distributed map objects) in the fragmented partition is moved slowly to the primary owner by the rebalancer. Until the move is done, the data remains available on the previous owners. The distributed map methods use this list to query data on the cluster.

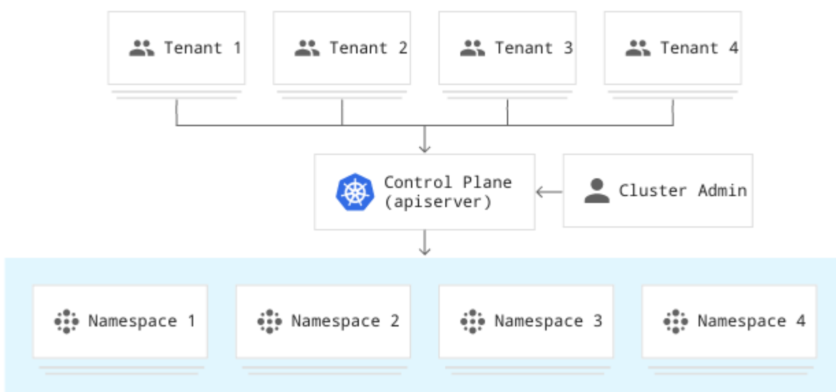
3.3 Multi-tenant security and isolation

A multi-tenant cluster is shared by multiple user and/or workloads which are referred to as tenants [27]. It is very crucial to make sure that data is completely segregated between the tenants and in no circumstances can data of one tenant be visible to the other. This is considered one of the biggest security risks [28] of the cloud multi-tenancy. Along with this it is critical to make sure that the resource is distributed equally among the tenants and in no way a starvation occur in the cluster. It is very important to keep in mind the security implications of sharing certain kind of resources among tenants.

In the case of FaaS, we already saw that how the same function instance can be reused for different tenants and the cache can be lying around corrupting the data or worse, exposing it. Along with this if there are functions that deal with very sensitive information of a client, it is often very unsafe to schedule other tenants' operations in the same node.

In this thesis, we leverage Kubernetes to guarantee isolation of data in a multitenant setup. We separate each tenant and their resources into their own namespace [29]. Along with this we leverage policies [30] by Kubernetes to enforce resource limits to each client and manage access. By default, compute resources on a Kubernetes cluster are unbound. With Policies we can set quotas or restrict completely any or all kind of resources available to a pod. A LimitRange is a policy to constrain resource allocations (to Pods or Containers) in a namespace. A resource quota, defined by a ResourceQuota object, provides constraints that limit aggregate resource consumption per namespace. Then we have Pod Security Policies [31]. This is used to make sure that each pod that runs comply by certain security policies and permissions. This is what will be used in our system to ensure that extremely sensitive pods will be scheduled separately from the rest of the pods. Figure 14 shows how a multitenant system looks like controlled by Kubernetes policies.

Figure 14: Multi-tenant infrastructure



3.4 Monitoring and tracing

According to a survey of 2018 done by Serverless(the company) [16], the second most thing the devs are worried about the development process of a FaaS application is monitoring and logging. Monitoring, logging and tracing are all ways to ensure correctness in your system. Monitoring serverless application is very complex. In a traditional application, we usually focus on monitoring the execution of code, while in serverless, we also need to monitor the integrations between the different services and make sure that we can follow a request end to end in our distributed system.

The demonstrate the issue a bit better, we consider a real AWS Lambda workload and look at the logs [32]. We deploy a function with a clear bug as a zip file with the CLI of AWS. We get a HTTP endpoint for the function. This endpoint can be tried hitting with an API client like postman to get a 200 OK result as seen in Figure 15.

Figure 15: Trying to hit the AWS Lambda function endpoint via REST API client

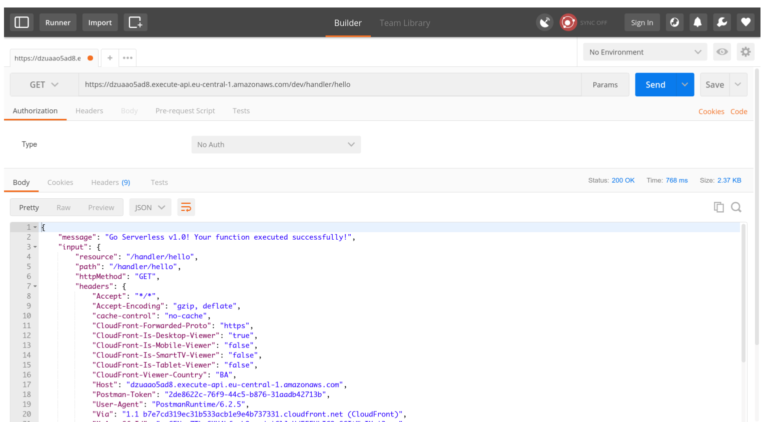
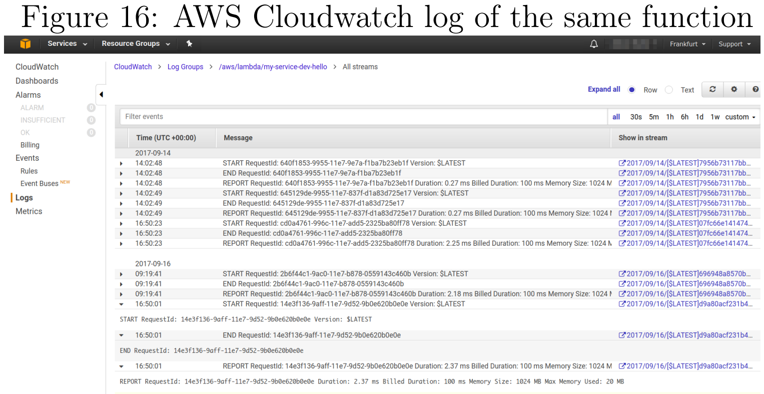


Figure 16: AWS Cloudwatch log of the same function



AWS Lambda provides the logs via AWS Cloudwatch. We now look at the logs produced by AWS Cloudwatch upon invocation. This can be found in

Figure 16. As you can see the logs are very fine grained, but the problem is that the logs make no sense or helm the developer in the debugging process. Error messages for failing functions are not verbose enough, so they often go unnoticed. We also are having a hard time finding functions that timed out. This basically clears up why we need more innovations in the monitoring of Serverless functions.

Let us understand first what monitoring, logging and tracing entails as functionalities.

3.4.1 Logging

Logging is used to track errors that were encountered in an app and other debugging information of the running app. Even if the application is distributed or otherwise, a good logging system will accumulate the logs and provide it in a centralized way for the ease of the developer. Log files can show any discrete event within an application or system, such as a failure, and error, or a state transformation. When something inevitably goes wrong, such transformations in state help indicate which change actually caused an error.

Since log files can grow exponentially, it is very important to analyze before setting the logging framework in place what are the things that need logging. We only need to know the crucial information but be mindful not to omit the ones that might contribute to the debugging of the system. Even with this logging system often tend to eat up all the storage in the systems. There are several strategies that can be adopted to evict this problem. One way to deal with this is to set a retention period to logs and clear up the log entries that are older than this date. In most cases this would work really well since there is seldom need for looking into significantly older logs. Another way to deal with bloated logs is to rotate the log files. This is the practice where you write to a different log file in a time window. This time window can be a day or a week or a month and so on. The older log files are compressed and backed up someplace if need arises to use them. The most recent logs will be available in the current log files. This is a heavily adopted log handling mechanism.

A very good logging system will have a clean and standardized structure that lets the developer read through it and debug easily. keep in mind that logging should be precise and on point. It is also important to keep in mind whom is the log for [33].

3.4.2 Tracing

Traces are intentionally a noisier set of data that logs. When logs document discrete events happening in the application, traces document a much wider continuous control flow in the application. The idea is to track the data flow and the control flow in the application completely. There is a lot more information available in the traces as opposed to the logs.

The goal of tracing is not debugging but optimization. Traces often track the whole lifecycle of one single request. This makes it easier for the developers to understand the bottlenecks and other performance issues in the application. It can often be used along with the logs when a problem occur. Traces can tell you what has lead to that problem and how the previous functions have contributed to the issue.

Traces need not be very reactive as opposed to the logs. Considering the amount of data involved, it is easy to see that how resource intensive can tracing be. It is often very hard to manage and it involves writing a lot more code to make sure the framework catches everything we need to. But in microservice or FaaS architectures, traces can be crucial since there are a lot of connected separate parts involved in the pipeline and traces let you have a complete overview of your workflow in action.

3.4.3 Monitoring

Monitoring is a wider term that can be applied for both tracing and logging. But in this context we talk about more complex monitoring systems. Monitoring here helps the developer understand how their overall system works. It involves instrumenting an application and then collecting, aggregating and analyzing the metrics involved in the system. The main purpose of the monitoring system can be considered as alerting the developers issues going on in the platform before the users find out. These issues can be more system specific like out of memory, out of storage, or other system failures.

There are various metrics from the system or the application that you can feed to your monitoring platform. Recently systems developers have started feeding application logs as well to monitoring platform so that they can get alerts as soon as an error appears in the application log.

3.4.4 Adaptation in FaaS

Logs While working with an OpenSource FaaS platforms where the isolation is managed by Docker, we can use the inbuilt logging system by Docker which are standard systemd logs [34]. We will not get into details of that here.

With the kind of distributed system that we have in our hands, in this thesis we propose adapting distributed tracing to trace and monitor the function instances.

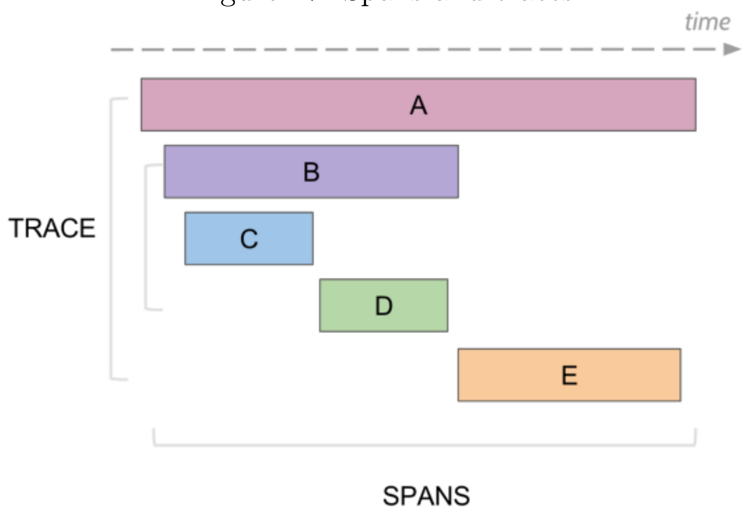
Tracing Distributed tracing is a method used to profile and monitor applications, especially those built using a microservices architecture. Distributed tracing helps pinpoint where failures occur and what causes poor performance [35]. Opentracing [36] is a standard specification for the definition of tracing information. Systems that are written following this specification can be ported from one tracing framework to another without having to change the implementation. There are some fundamental attributes of Opentracing API that are worth understanding. Refer Figure 17.

- Span It represents the most atomic unit of logic in a pipeline. This unit will have a name, a start time, and the duration
- Trace A trace is basically a collection of spans. It represents the workflow of the entire pipeline. Each distributed component in the pipeline contribute their own spans to form the trace from the aggregation.

As the definition goes from the documentation, OpenTracing is a way for services to "describe and propagate distributed traces without knowledge of the underlying OpenTracing implementation.". The idea of tracing was well covered before. What Opentracing adds to it is the capability to make tracing infrastructure independent and standard across platforms. With the span and trace form of specification, OpenTracing makes it easier to:

- Spans of services
- Time taken by each service

Figure 17: Spans and traces



- Latency between the services
- Hierarchy of services
- Errors or exceptions during execution of each service.

Monitoring Monitoring of distributed systems can be heavily challenging but yet highly necessary. The best industry recommended way is to do Time series monitoring suggested by Google via the proprietary tool Borgman [37]. It is basically an in-memory database that scrape different kind of metric from the system and applications. Then it does a rule based extraction from the data and provides a queryable time-series database. Borgmon relies on a common data exposition format; this enables mass data collection with low overheads and avoids the costs of subprocess execution and network connection setup. The data is used both for rendering charts and creating alerts, which are accomplished using simple arithmetic. Because collection is no longer in a short-lived process, the history of the collected data can be used for that alert computation as well.

In our FaaS application, such a system basically can scrape the system information from the Kubernetes cluster since each function is a pod. Then we can specify appropriate rules for the kind of data aggregation we want to see, visualize it and setup alerts. We will be using an Open Source alternative which we discuss in the next section.

4 Implementation

For implementing the aforementioned strategies and verifying its effectiveness in making the Serverless workflows efficient, it is important to trial it on a

are closed in its source and vendor locked is practically impossible and stalling the growth of Serverless as a paradigm. Instead several Open Source FaaS infrastructure were analyzed for this thesis for the implementation of our ideas. Along with the platform, choosing the right orchestration and clustering tools, the workflow implementation tool, the right monitoring tools, etc. are also vital in the implementation. So before going in detail about the architectural specifics of the implementation, let us analyze the tools used in the process and the reasoning behind their choosing.

4.1 Tools

4.1.1 Container Orchestration

We are going to work with a containerized setup like was hinted at the beginning of the thesis description. Each function that is being written will be containerized and brought up and down, scaled up and down based on the configurations and usage requirement. We have to go with the right containerization platform and an clustering tool that would take care of managing, scheduling, scaling up and down, etc. of these containers across a cluster of nodes agnostic of the application specifics or the underlying systems specifics. We of course go with the industry standard here which are Docker and Kubernetes especially because all of the leading FaaS solutions these days work on both of these technologies. A gentle introduction to both tools before proceeding to FaaS specific solutions.

Docker Docker [38] is one of the leading Linux Containers solution that is being adopted very widely across all kinds of software infrastructure maintenance environments. According to the Docker Inc., over 3.5 million applications have been placed in containers using Docker technology and over 37 billion containerized apps have been downloads [39]. Advantages of using containers for application shipping was already seen in Section 2.1.2. Docker made the whole Linux Containerization landscape a lot more approachable as a packaging technology by the introduction of namespaces.

Without delving too much into the technicalities of containerization, we would like to quickly explain the life of a containerized application with Docker. Some terminologies that would help with understanding the concept:

- **Docker image**: Like in any virtual machine environment, images can be thought of somewhat a snapshot of the current state of an execution environment(which is basically a stripped down operating system with applications installed on it, ready to run). What makes Docker images unique is its immutability. You cannot modify a docker container. You can create copies or delete and recreate but not change the state. This helps in guaranteeing that once your Docker image has reached a working stage, it will always continue working no matter what. You can try an add changing to the running instance of this image, but none of these changes are persistent from the point of view of the image. You can shut it down and start from the same image state as was created.

  Sharing these images is an extremely easy process. There are container registries which are hosting services for docker images like Github is for git tracked code. Popular publicly available container registry is DockerHub [40]. Developers can push their docker image to docker with a simple 'docker push' command from their command like and share or make it publicly available for other developers or software tools.

  To create a docker image, the most straightforward way is via a configuration file called Dockerfile. According to the reference from [41],

  > "Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession."

  For example, the following code block shows a Dockerfile written to dockerize a simple Python app, that runs a simple flask HTTP server.

```
FROM python:3.6.1-alpine
WORKDIR /project
ADD . /project
RUN pip install -r requirements.txt
CMD ["python","app.py"]
```

- **Docker Container**: If Docker image is a digital photograph, a docker container is like a printout of the photograph [42]. Containers can be thought of as a running instance of the image. Each container is run separately and unlike the images, you can change the running container. If you want to persist these changes though, you will have to commit the running container it its running state by committing it as a new image. Your host operating system isolated the running container from the others in the computer. Each container instance will have its process namespace, limits on the resource usage, allowed system calls, etc. Communication across containers can be setup explicitly. Most production applications, usually have multiple containers running together with communication internally so as to isolate each process environment, to avoid cascaded application damage, etc. A container is inherently not addressable directly from external network, although one can open it up by exposing corresponding service port to that of the host system, provided necessary security precautions are taken.

- **Orchestration tools**: Docker by default ship a couple of orchestration tools that are specific to Docker. An significant one among this is a docker-compose. Docker-compose lets one tie up multiple docker containers, expose certain ports in each docker containers, pass environment variables, define the communication and storage usage rules, etc with the help of a configuration file by default to be name docker-compose.yml. This is a very simple tool to use that helps in most basic usages of the application deployment. One can connect multiple nodes together and deploy the containers across these nodes via docker-compose using a library called docker-swarm. Docker swarm takes care of very basic scaling up and down of containers etc.

Kubernetes Now that we have seen a popular containerization solution called Docker, it is time to see the most popular orchestration solution. Docker swarm was already mentioned but when it comes to modern applications, the requirement goes far beyond this. The application needs better scaling heuristics, version rollout policies, cluster management, better networking and application discoverability, better monitoring and alerting systems, etc. This takes up to a much more advanced container orchestration platform called Kubernetes. It is worth noting that Kubernetes is not just built for Docker but for multiple flavors of Linux Container technologies.

Kubernetes is an Open Source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation [43]. Contrary to the traditional deployment setups where applications ran on physical servers, we have moved to an era where we deploy packaged applications and are deployed across clusters of virtual nodes provided by cloud providers. We require smarter tools for this to manage complexities in different levels starting from application packaging to cluster management. Kubernetes can be considered as the most popular solution that deals with these complexities.

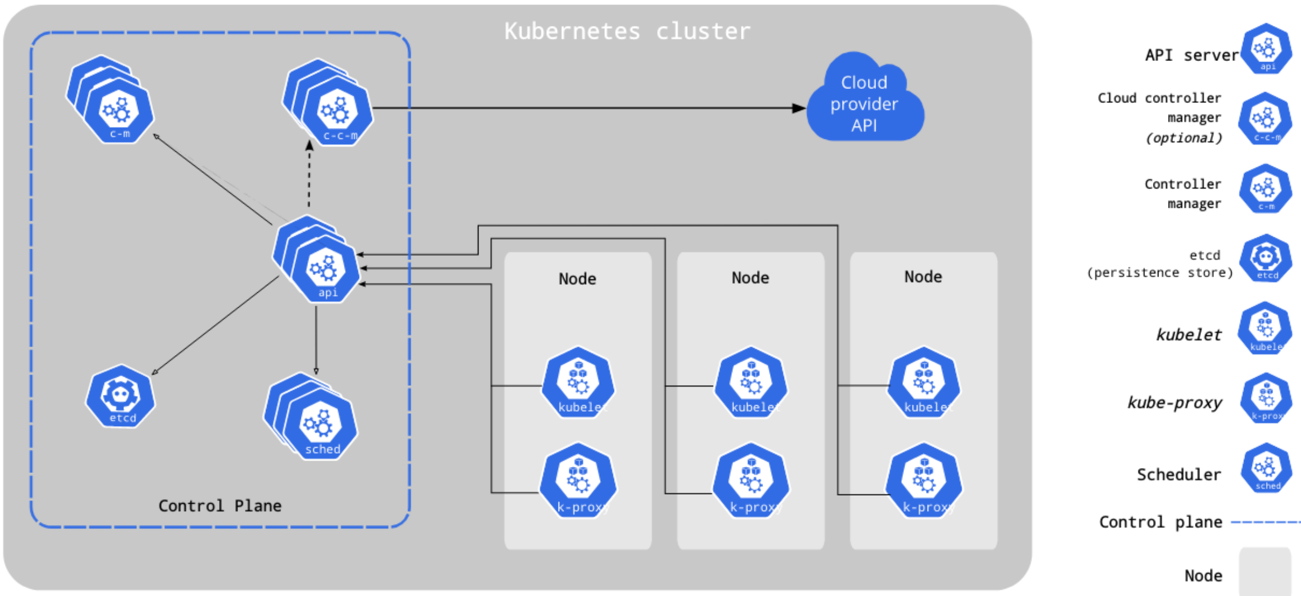
Kubernetes provides the framework to run these applications along with the tools for the following purposes [43]:

- Service discovery and load balancing
- Storage orchestration
- Automated rollouts and rollbacks
- Automatic bin packing to make sure optimal resource usage
- Secret and configuration management
- Monitoring the usage and load to the cluster and applications

A great thing about Kubernetes as project these days, is the community support. It has a very large and widely adopted community. Along with that most cloud providers now support out of the box kubernetes engines making the development of infrastructure agnostic applications very easy. This is the way to go to be away from a deployment cycle that is not completely vendor locked in.

Kubernetes is an immensely complex piece of software with numeral tools and add-ons. Figure 18 depicts the architecture of Kubernetes. The control plane is the core component of the setup. It consists of the API server to the platform, etcd to store the state of the cluster, scheduler to deploy the

Figure 18: Kubernetes infrastructure



Pods(collections of containers that makes up an application) to the corresponding node in the cluster evaluating the usage requirements and availability, cloud controller manager that links the logic of the cluster to the API of the cloud provider, etc. At the risk of getting out of this scope of the thesis, we do not analyze more of the technicalities of kubernetes.

In our implementation, we will use the heuristics of scaling provided by kubernetes in multiple cases. Alongside, the web UI kubernetes provides lets us visualize the resource usage by the platform and applications to a great extending helping with the monitoring of the setup.

- **Namespaces**: It is very useful to understand the concept of Namespaces in Kubernetes though. We can logically divide each cluster into multiple virtual clusters called namespaces. It is a way to divide the existing cluster into separate logical partitions. The implications of this provision is huge. We will be utilizing this feature of kubernetes to logically partition the function executors to support multi tenancy.

Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace [29].

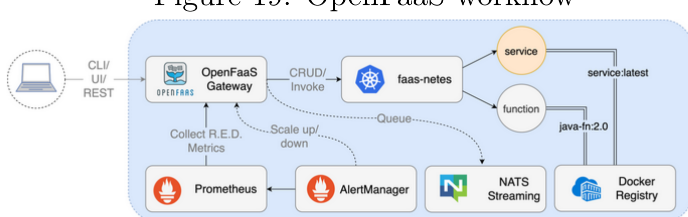
4.1.2 OpenFaaS

Now that we have seen an overview of the packaging and clustering management systems, it is time to look at the right platform to test out our state-ful FaaS idea on. Considering that one main aim of the thesis is to move away from vendor locked platforms, it all makes sense that we investigate the available open source FaaS solutions to extend on.

A survey was done comparing multiple open source FaaS offerings as explained in the section 2.2.3. From [43] , it is quite clear that one of the most simplistic approach to architecture and flexibility belongs to OpenFaaS. The ease of setup and the community support also is a huge add on for the OpenFaaS to be chosen as out tool of preference.

OpenFaaS was a one person project that was initially developed just to test out the power of vanilla docker orchestration tools to deploy event driven functions on demand and scale. The clean and scalable architecture soon put the project in spotlight. The best thing about OpenFaaS is that, the core modules of OpenFaaS are very light weight and all the other units can be added on to this core as necessary. The tool soon got to using Kubernetes as the default deployment platform due to the increased popularity and to make the best of Kubernetes heuristics for scaling.

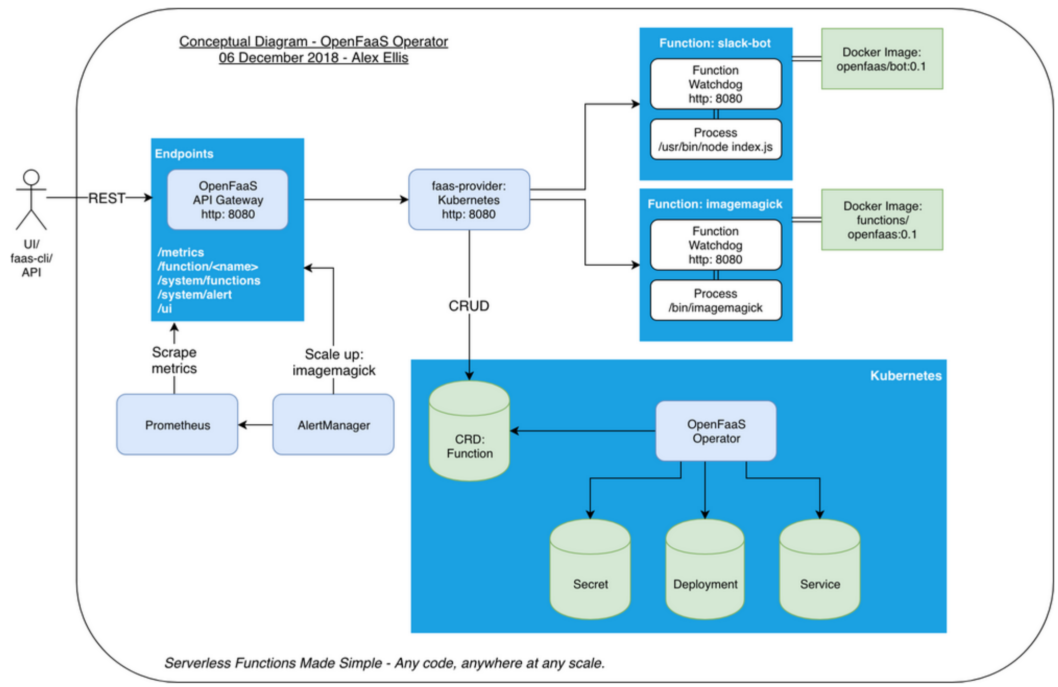
Figure 19: OpenFaaS workflow



The following are the main components on an OpenFaaS setup to give the user a bit more intuition on how functions are scheduled, executed and scaled in the platform. Figure 19 goes along with the following description.

OpenFaaS Gateway The gateway is the entrypoint to the FaaS infrastructure. It provides an API which opens an external route into the functions. The gateway does a lot of the main functions in the infrastructure. The gateway is basically responsible for collecting the metric information and scaling the functions accordingly. It has a built in UI portal for ease of deployment and invocations of functions for the user. When kubernetes is used as the orchestration platform, the conceptual design of OpenFaaS can be visualized as Figure 20.

Figure 20: OpenFaaS conceptual design with Kubernetes



As can be noted in the image, Prometheus and Alertmanager are connected to the OpenFaaS Gateway API.

- Prometheus is a monitoring system and time series database. Prometheus is now the de-facto monitoring solution for Cloud Native projects. It combines a simple interface with a powerful query language to monitor and observe microservices and functions, which are the two primitives of any FaaS. Prometheus basically does two functions. It gets metrics from machines in your cluster. These machines can be actual nodes or virtual machines or containers. One can define custom rules to check on these metrics and if any of the rules are triggered, Prometheus will fire off alerts via AlertManager. OpenFaaS Gateway exposes a lot of these collected metrics via Prometheus for visualization and monitoring. We will be using these metrics for our monitoring.

faas-provider faas-provider is a very flexible interface that provides CRUD(create, read, update, delete) to functions and the invoke capability. The information about the function that need to be created/updated/invoked gets fed directly from the OpenFaaS gateway which is the endpoint to which external world communicates to.

The design of faas-provider makes OpenFaaS a unique platform. One can their own faas-provider and hence change the backend of the OpenFaaS infrastructure very easily. There are design guidelines available to develop your own faas-provider backend [44] , which basically is defining how CRUD and invoke operations are handled by the backend. The most stable and popularly used faas-provider that is maintained by the community is faas-netes, which is the Kubernetes backend for OpenFaaS.

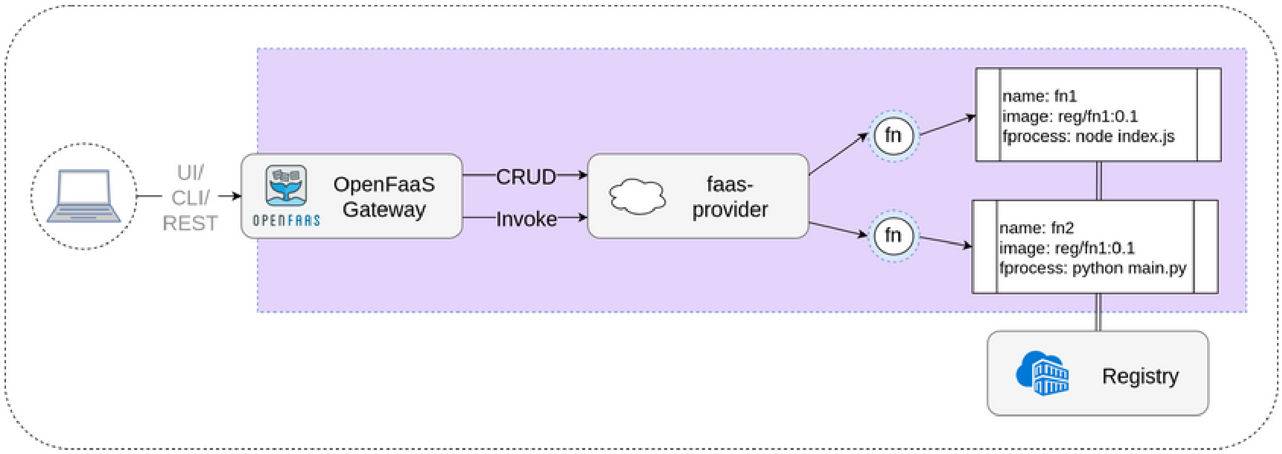
faas-provider takes care of scheduling the functions in the right node based on the availability and requirement. It also does the scaling up and down of the function instances based on the information from the gateway that it gathered via Prometheus. Figure 21 shows the conceptual view of just faas-provider stripping away the rest of the complexities.

OpenFaaS watchdog The OpenFaaS watchdog [45] is responsible for starting and monitoring functions in OpenFaaS. Any binary can become a function through the use of watchdog.

The watchdog becomes an "init process" with an embedded HTTP server written in Golang, it can support concurrent requests, timeouts and healthchecks. The newer of-watchdog mentioned below is similar, but ideal for streaming use-cases or when an expensive resource needs to be maintained between requests such as a database connection, ML model or other data.

Auto-scaling OpenFaaS ships with a single auto-scaling rule defined in the mounted configuration file for AlertManager. AlertManager reads usage

Figure 21: faas-provider



(requests per second) metrics from Prometheus in order to know when to fire an alert to the API Gateway.

The API Gateway handles AlertManager alerts through its /system/alert route.

The auto-scaling provided by this method can be disabled by either deleting the AlertManager deployment or by scaling the deployment to zero replicas.

One can specify the minimum number of replicas and the maximum replicas to be available. If minimum replicas is defined to be >0 then a warm copy of the function will always be idle-ing there by avoiding the cold start issue. Although this comes with an added cost of a docket container always up in the memory although the resource usage during the idle time is super low. We can also fine tune several other parameters like the factor by which the function should be scaled up or down when there is a burst or decline of the traffic, etc. This makes OpenFaaS extremely powerful and yet in the most simplistic way possible.

When Kubernetes is used as the backend, instead of AlertManager the built-in Horizontal Pod Autoscaler [46]. This is a lot more matured as a scaler scheduler and we will be using that for the thesis implementation.

NATS streaming A curiously lightweight application that has been adapted into OpenFaaS is NATS. NATS provides simple and secure messaging functionality to the setup. It does event and data streaming in the cluster. OpenFaaS uses NATS Streaming which builds on top of the base NATS protocol to offer data streaming or a queue [47] . NATS streaming provides Queue worker in which the function invocation requests can be queued up by the API Gateway, and processed in parallel when the capacity becomes available. Asyncronous invocations can be very easily done since it is built in without making any changes to the gateway. Each function will have a separate endpoint that can be used to invoke it asynchronously.

NATS streaming is a Pub Sub protocol implementation like Kafka but with very high throughput compared to the latter. Publish-subscribe pattern corresponds to a mechanism where in producers publish messages that are grouped into categories and consumers subscribe to categories which they are interested [48]. NATS is extremely lightweight as a technology making it the right candidate for an elastic Serverless infrastructure, compared to a full blown message broker system like Kafka. Along side, considering the ephemeral nature of state in FaaS setup, an in-memory message delivery protocol like NAT could be extremely useful.

Triggers OpenFaaS functions can be triggered easily by any kind of event. A small piece of code will convert from the event-source and trigger the function using the OpenFaaS Gateway API. Some of the most used triggers are:

- HTTP: One can send POST requests to the function endpoint which follows the patter ‘`https://`<gateway URL>:<port>/function/<function name>‘
- Cron
- NATS streaming/Async: You can execute a function or microservice asynchronously by replacing *function* with *async-function* when accessing the endpoint via the OpenFaaS gateway.
- CLI: we can trigger user faas-cli which is a command line application to communicate with faas gateway
- Apache Kafka

- AWS SQS
- Redis
- Minio/S3
- RabbitMQ

Runtime supports and templates OpenFaaS is one of the unique engines that supports any and all programming languages to write functions in because of its architecture. The way OpenFaaS works, it dockerize the application by adding an of-watchdog to the application container and deploy it to the kubernetes cluster. To make the process easier, OpenFaaS does not expect you to write the Dockerfile. Instead, it provides already packaged versions of language bundles called templates. The developer can just pull the right template from the template store and just edit the entrypoint script to add their application logic.

Like was briefly hinted earlier, OpenFaaS provides a command line tool called faas-cli. This tool can be used to build, push and deploy the docker images from the code. With build, it build an image into the local Docker library. With push, it pushes that image to a remote container registry. With deploy, it deploys your function into a cluster.

As an example, to build a simple python function, the developer will follow the proceeding commands:

```
faas-cli template store pull python3
faas-cli new funcname --lang python3

faas-cli build -f funcname.yml
faas-cli push funcname
faas-cli deploy -f funcname.yml
```

4.1.3 FaaS-flow

In Section 3.1, we analyzed different possible ways to do function composition. We saw that workflow pattern is the most efficient and flexible design for a FaaS application to composite functions. What this means is, the best way to go about it is by keeping a Distributed Acyclic Graph in memory that is logically sort of a flowchart defining the conditionals, branches and the loops in a function composition.

FaaS-flow [60] is a library written in Golang that lets the developers fiddle with the runtime of OpenFaaS via an SDK. We will use it to orchestrate a pipeline with long running and short running ETL jobs without having to orchestrate them manually or maintain a separate application. Faas-flow ensures the execution order of several functions running in parallel or dynamically and provides rich construct to aggregate results while maintaining the intermediate data [60]. Using Faas-flow you can combine multiple OpenFaaS functions with little codes while your workflow will scale up/down automatically to handle the load.

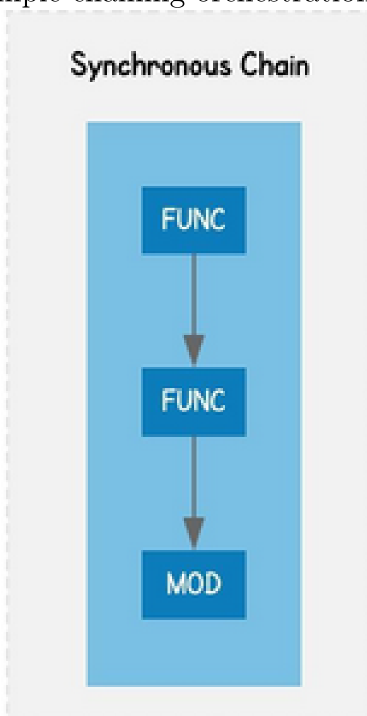
The main motivation behind the development of faas-flow is building a pipeline that is very loosely coupled. We adapt faas-flow in this thesis considering its extremely stateless nature. This means that the fundamental notions of FaaS architecture is not violated here. Faas flow provides the possibility to reuse the same function in multiple workflows which will execute in parallel agnostic of each others execution. Along with this, in the DAG generation, faas-flow supports multiple operations to orchestrate pipelines with conditionals, branching, iterations, etc.

Faas flow only supports Go as a programming language for the development at the moment which is a hard constraint. In any case this can help us build the workflow DAGs for our proof of concept in this thesis. Now we will see some codeblocks [60] written using faasflow library in Go that forms certain pipeline orchestrations.

```
func Define(flow *faasflow.Workflow, context *faasflow.Context)
    (err error) \{
    flow.SyncNode()
        .Apply("func1")
        .Apply("func2")
        .Modify(func(data []byte) ([]byte, error) \{
            // do something
            return data, nil
        \})
    return nil
\}
```

The above code block defines a composition that begins by applying a function called func1 to the start node. Then it goes ahead and apply func2. After this we call a modify operator on the pipeline to get the data from the last pipeline and do something with the data and return it as the end result of the whole pipeline to the invoker. A workflow diagram of the dag can be referred from the Figure 22 In the above codebase though, like one could in-

Figure 22: Simple chaining orchestration with openfaas

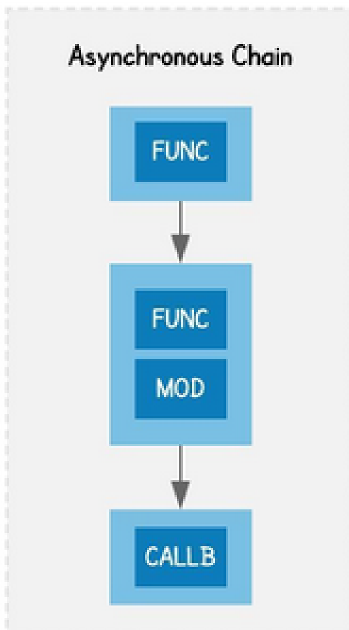


fer, the functions are in a blocking stage. Meaning, the whole pipeline waits until each of the function that is executing to finish before moving further down the DAG with the execution. If there is no intermediate data to be passed along, this can actually slow down the whole cycle. To avoid this, faas-flow supports asynchronous chaining. The codeblock below implements an asynchronous cycle and Figure **??** represents the DAG structure of the same.

```
func Define(flow *faasflow.Workflow, context *faasflow.Context)
    (err error) \{
```

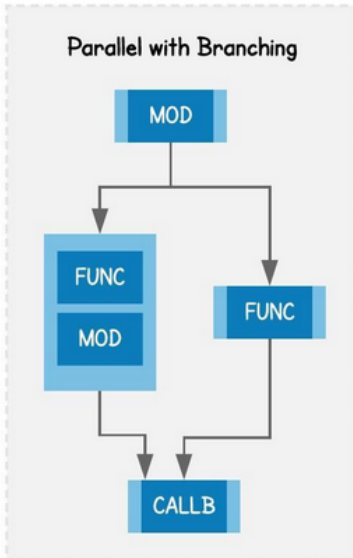
```
    dag := flow.Dag()
    dag.Node("n1").Apply("func1")
    dag.Node("n2")
        .Apply("func2")
        .Modify(func(data []byte) ([]byte, error) \{
            // do something
            return data, nil
        \})
    dag.Node("n3").Apply("func4")
    dag.Edge("n1", "n2")
    dag.Edge("n2", "n3")
    return nil
\}
```

Figure 23: Asynchronous function chaining



Aside from these basic chaining operations, with faas-flow we can even design orchestration that are a lot more complex like parallel branching and dynamic branching. The following codeblock creates a parallel branching orchestration and Figure 24 represents the DAG that is created.

Figure 24: Parallel execution function chaining



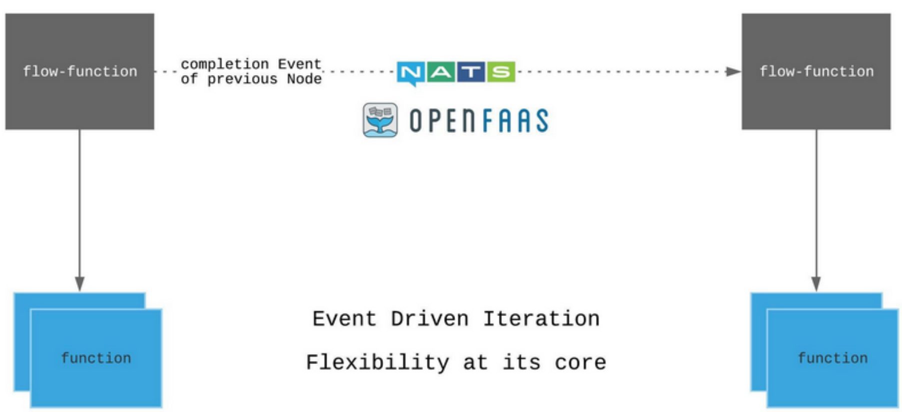

```
func Define(flow *faasflow.Workflow, context *faasflow.Context)
    (err error) \{
    dag := flow.Dag()
    dag.Node("n1").Modify(func(data []byte) ([]byte, error) \{
        // do something
        return data, nil
    \})
    dag.Node("n2").Apply("func1")
    dag.Node("n3").Apply("func2").Modify(func(data []byte) ([]byte,
        error) \{
        // do something
        return data, nil
    \})
    dag.Node("n4", faasflow.Aggregator(func(data map[string][]byte)
        ([]byte, error) \{
        // aggregate branch result data["n2"] and data["n3"]
        return []byte(""), nil
    \}))

    dag.Edge("n1", "n2")
```

```
    dag.Edge("n1", "n3")
    dag.Edge("n2", "n4")
    dag.Edge("n3", "n4")
    return nil
\}
```

Faas flow does not use a complete workflow based approach for the orchestration. Instead it mixes it with an event based workflow. Meaning that, internally faasflow keeps a DAG, but the completion of each node in the DAG is broadcasted with the help of an event queue. faas-flow uses NAT streaming [47] as the event bus. Node execution in Faas-flow starts by a completion event of one or more previous nodes. A completion event denotes that all the previous dependent nodes have completed. The event carries the execution state and identifies the next node to execute. With events Faas-flow asynchronously carry-on execution of nodes by iterating itself over and over till all nodes are executed. Figure 25 logically represents how function orchestration happen via event propagation.

Figure 25: Event based workflows



The most notable thing about faasflow is its flexibility and extensibility as a tool. We can extend faas flow to add different kind of storage infrastructures to the workflow. Faas flow basically has two kind of data. First is the state of the DAG that is being processed by faas flow and then is the intermediate data that need to be transferred between two functions. The former is called the state store and the latter is called the data store. We can write our own libraries to define our own data store and state store logic and how we want to deal with data in the pipeline. Figure 26 and Figure 27 represents the logical workflow when we add external storage suits.

Figure 26: State store logical view

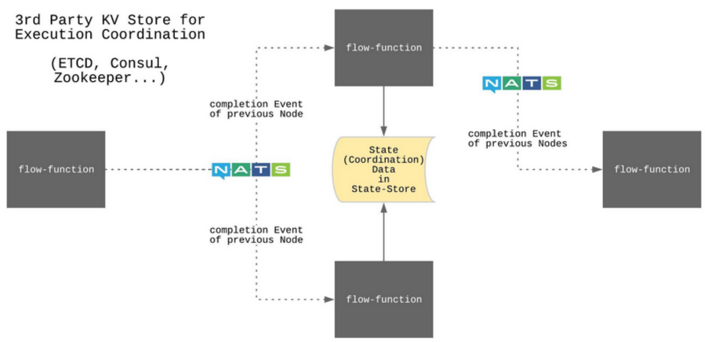
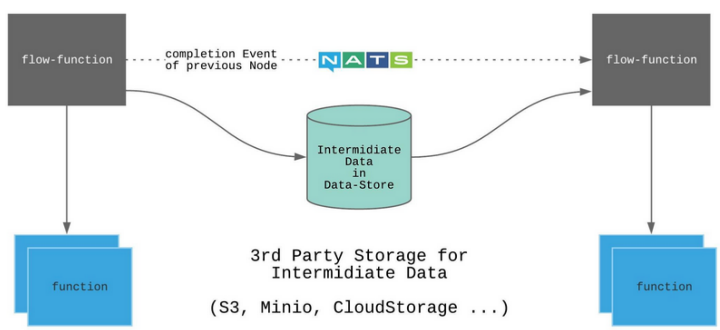


Figure 27: Data Store logical view



Another important thing about faas flow is that faas flow at the end of the works as yet another function in the FaaS infrastructure letting it leverage the autoscaling policies of the system than handling it itself. There rich features make us adapt faas-flow as the choice of orchestration engine for our proof of concept.

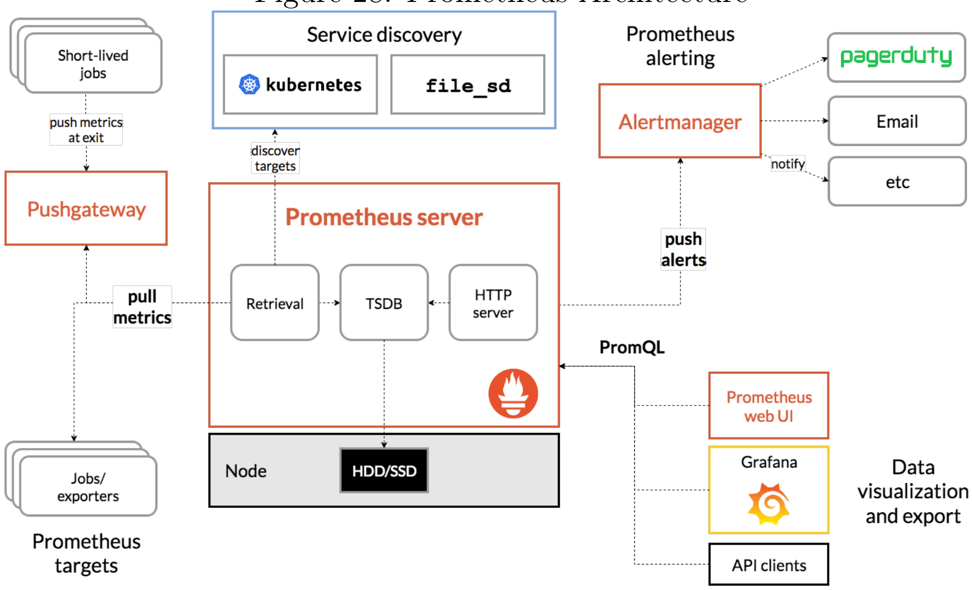
4.1.4 Prometheus

In the previous section, we discussed briefly about the time-series monitoring approach adapted by Google. Prometheus is an Open Source monitoring and alerting toolkit that works exactly the way Google Borgman works. The working of Prometheus is rather curious. We can statically configure Prometheus to detect certain metric sources. For example, the CPU usage of the nodes in the Kuberenetes cluster. Prometheus pulls these metric as a time-series over HTTP. Prometheus need not be setup in the same cluster as the applications. Prometheus does not even expect to be run as a distributed system. At this stage, Prometheus parses the data and keep a a multi-dimensional data model with time series data identified by metric name and key/value pairs. There are streams of timestamped values belonging to the same metric and the same set of labeled dimensions [49] . Prometheus also comes with a query language called PromQL that can query over streaming, multi-dimensional time series data.

Prometheus is a tool that is written in Go programming language. The main component of Prometheus is a server that scrapes the time series data. Prometheus packs several client libraries that need to be used in application code if data has to be pushed to Prometheus. For ephemeral jobs that are shortlived, as is the case with most FaaS functions, Prometheus provides a push gateway. The Prometheus Pushgateway exists to allow ephemeral and batch jobs to expose their metrics to Prometheus. Since these kinds of jobs may not exist long enough to be scraped, they can instead push their metrics to a Pushgateway. The Pushgateway then exposes these metrics to Prometheus [50]. There are several other exporters for various services. Along with various other support tools, Prometheus also has an inbuilt alert manager to handle alerts. Figure 28 shows the overall system architecture of Prometheus.

There are numerous tools that can be used to connect to Prometheus like Grafana that lets us visualize the data with more meaning in a dashboard. Prometheus can be used with any kind of numeric data. May it be machine-centric monitoring or monitoring of highly dynamic service-oriented architectures. We can track the usage of the system resources via Prometheus in a very fine grained manner. Memory, CPU, and execution time of the application, all can be accounted for as a function of time. The person maintaining our system can do fine grained billing using these usage metric.

Figure 28: Prometheus Architecture



OpenFaaS Gateway component exposes several metrics to help you monitor the health and behavior of the functions. We will leveraging that to have clean usage tracking for our system. For example, to get the total number of successful function invocations from the gateway, we can run the PromQL query $sum(\ gateway_{functioninvocationtotal}\ \{\ code=\backslash\ "200\backslash\ "\}$

4.1.5 Jaeger

We discussed how OpenTracing API helps identify the bottlenecks and allows easy debugging in a distributed setup. One of the most popular implementations of OpenTracing API is Jaegar [51]. Jaeger identifies that the difficulty in dealing with debugging in microservices or FaaS setups are an order of magnitude away from simple monoliths. The majority of operational problems happen in such platforms either as an issue of networking or that of observability.

We already mentioned Spans and Traces as a part of OpenTracking API in the previous section. One think special about Jaeger is that, it handle trace as a directed acyclic graph of spans. This basically visualizes the data/execution path through the system.

Figure 29: Jaeger Architecture

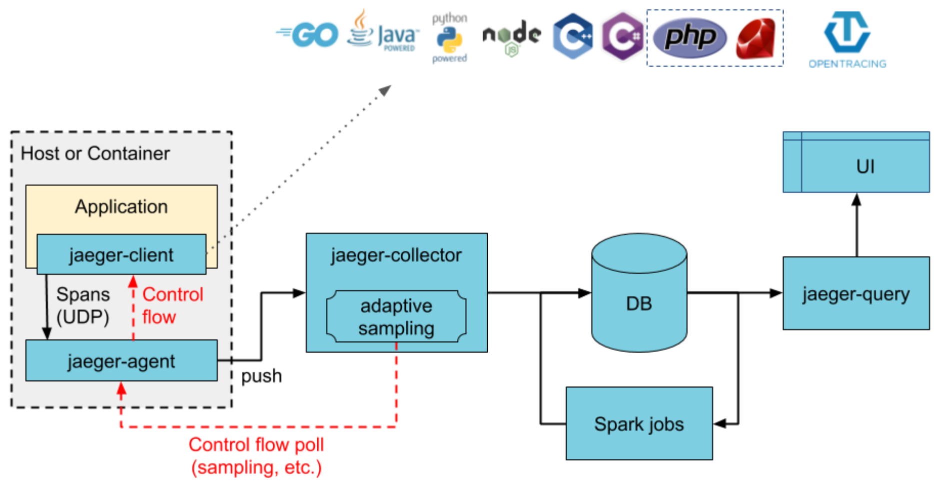
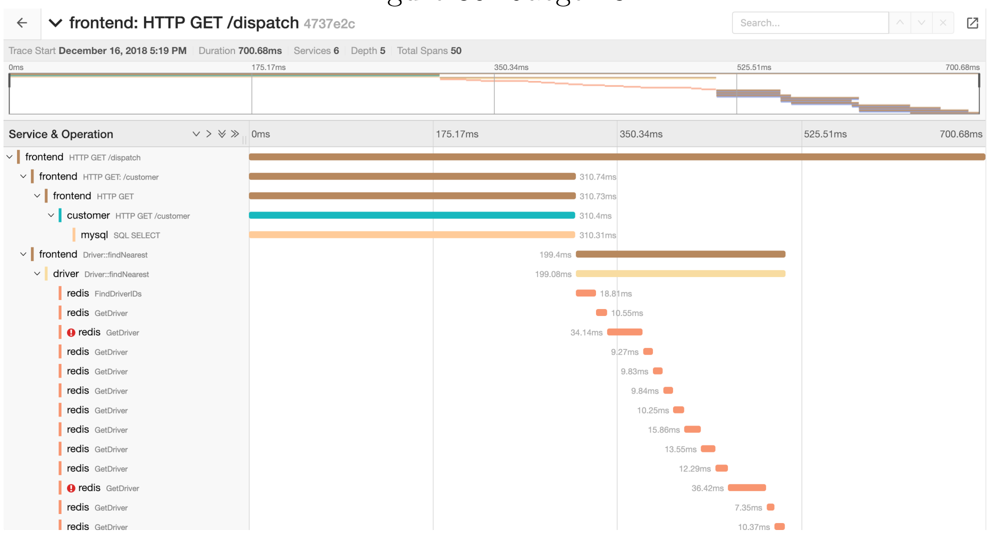


Figure 29 shows the architecture of Jaeger. Jaegar has code facing client libraries that are compliant with OpenTracing API. It can be integrated very easily with any tool that is integrated with OpenTracing. This includes frameworks like Flask, gRPC, and many more. Basically an instrumented service creates spans when receiving new requests and attaches context information (trace id, span id, and baggage) to outgoing requests [52]. Sampled spans are propagated out of the process asynchronously to Jaeger Agents. This process has very little overhead. The Jaeger agent is a network daemon that listens for spans sent over UDP, which it batches and sends to the collector. It is designed to be deployed to all hosts as an infrastructure component. Collective receive these traces and process it to validate, index, transform and store them. Like Prometheus, Jaeger also supports a Query language via a Query component and visualizes the output in a UI. Figure 30 shows what the UI looks like for a web app on an HTTP request to the endpoint.

4.2 Architecture Overview

In this section, we will see how the tools mentioned above were efficiently used to build an orchestration system for pipelined FaaS workloads that

Figure 30: Jaeger UI



work much better than the out of the box solutions. Great care was given into building a system that is very easy to deploy and maintain for the infrastructure providers. We believe that the practicality, adaptability and the accordance with the core philosophies of serverless architecture make our system unique.

We can separate our system logically into 3 parts. The FaaS runtime, the workflow framework and the workflow defining client API. The functions that are (written in any language of preference) that does the computations exist outside the logic.

Figure 31 depicts the overall architecture of the setup. To begin with, we have the OpenFaaS runtime which takes care of the execution of the function instances upon triggers using Kubernetes and packaging the functions as Docker containers. We configure the setup to not scale down to zero to avoid the cold start time during the initial start. We might still incur a bit of cold start when the number of instances is scaling up upon heavy load.

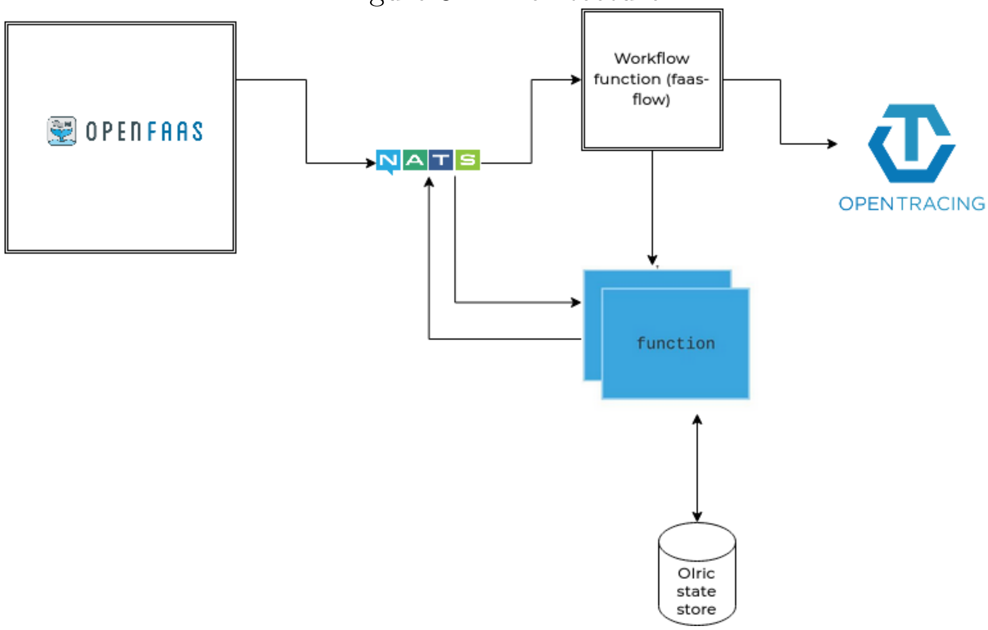
By default in OpenFaaS, functions are coupled with a tiny Go based server and a function watchdog [45]. The watchdog takes care of parsing the incoming requests and forwards it to the function via standard IO. There is no latency between this call transfer since it happens internally and the watchdog process is co-located with the function instance. In our implementation though we remove the watchdog component from the function runtime since our added framework for the workflows take care of handling the requests and passing it between the functions. We try to avoid the added latency by the watchdog process since we already serializes and de-serializes the data as a part of data store library. This also means that we can test the efficiency of the orchestration setup without the added latency by the platform. OpenFaaS' runtime driver for Kubernetes is faas-netes, which deploys functions as Kubernetes pods, and then delegates scheduling decisions to the Kubernetes scheduler.

We keep the NATS streaming message queue that is used by faas-netes for queuing of requests. The interesting thing about the NATS streaming here is that, the storage is in-memory only of the queued requests. This is rather interesting because it means that the message queue is completely autoscalable as well since we have the RAM pool controlled by the Kubernetes autoscaling logic. This helps in keeping our platform completely elastic.

The requests received by the gateway are queued in the NATS streaming. The workflow function orchestrated using faas-flow will receive these messages from the queue. The workflow framework takes care of handling the request arriving, formatting it in the way needed. Then the DAG is processed with the input data by the framework. The DAG or the workflow is a part of the state that the OpenFaaS runtime need to process. The whole workflow is basically stored in the state store along with the current state of the workflow which can be retrieved by the workflow framework from time to time to trigger the next function that needs to be executed. Whenever a function finishes its execution a FINISH event is sent to the message queue which is the NATS streaming, which triggers the workflow framework to fetch the next node to be executed from the state store and trigger it over HTTP via the OpenFaaS gateway. At the end of each function, a call is made to the OpenTracing setup to register the span of each function invocation which can be used for evaluation and optimizations. Once the execution of a particular node is over, the information about that node in the flow is removed from the state store and there by automatically freeing up memory.

The data returned from any of the function invocations are stored in an intermediate data store provided by us. This data is ephemeral in nature and exists in memory only until the workflow has successfully finished execution. There is a request id parameter associated with the particular invocation of

Figure 31: Architecture



a node or instance. The data stored will use this request ID to identify it uniquely and is fetched by the following function accordingly.

4.2.1 Workflow framework

The workflow framework is the most important part of our orchestration platform. We discussed in section 3.1 different theoretical solutions for composing functions. From the analysis it was quite clear that the workflow based orchestration provides the most amount of flexibility and scalability to the system. The problem of introducing something like that directly to Serverless platform is the complication in maintaining the state. Since we have multiple functions getting triggered(asynchronously at times), it gets really hard to realize if a function has terminated and might even lead us to a polling sort of methodology which is kind of inefficient. Instead we choose to mix event based orchestration to workflow orchestrations.

The user communicates with the system via the client library. This is a form of domain specific format to specify the structure of the Directed Acyclic Graph. We use the faas-flow interface language to provide the client library. This was explained in detail in section 4.1.3. By default, OpenFaas runtime takes caring handling events from function like function termination, function error, etc. We cannot use this inherent ability since we want our workflow to proceed differently that the conventional way. Hence we tweak the function runtime to override the default event handler and the function executor in the OpenFaas platform. We make each functions sent out events to the event queue after the execution completion, on errors, etc. Upon receiving these events, our workflow runtime will look inside the state store for the next node/function to be executed according to the DAG specified via the client library and the respective function is triggered. The workflow library globally keeps a context map for each invocation. The context map contains the current node which is the current node/function of execution, state which represents the orchestration structure and the requestID which is a unique identification for the current request invocation. Like was mentioned earlier, this is used while storing the function intermediate data of each invocation.

Along with the orchestration, workflow framework takes care of the trace handling. Each function can take three states during their lifetime - Running, Paused, Stopped. On each of this state update via the events, the framework sends logs a corresponding entry on the OpenTracing platform. This is how the spans and traces are logged in the system.

Using the standard visual interface of OpenTracing, we can visualize these tracing information which would look similar to Figure 32. Like is visible, there is a request ID representing the trace of the entire orchestration. Then you can see the spans of each node in the graph(function in composition). It is easy to measure the latency of each step via this interface.

Along with this interface, we use a D3 based library setup to visualize the DAG structure that represents our function orchestration. Figure 33 shows how the DAG looks like in our setup for a sample code that modify data based on a conditional "odd or even". The blocks composing the modify nodes are the ones that change the data or have side effects if we talk in a functional language paradigm.

There are certain configurable parameters in the system. We can enable or disable the tracing, we can set the whole system to scale from 0 than from 1, we can set an upper limit to the amount of memory and CPU used by the system, etc. The standard OpenFaaS config [63] support is used for this but we found it quite flexible and easy to use.

Figure 32: Trace of a sample workload

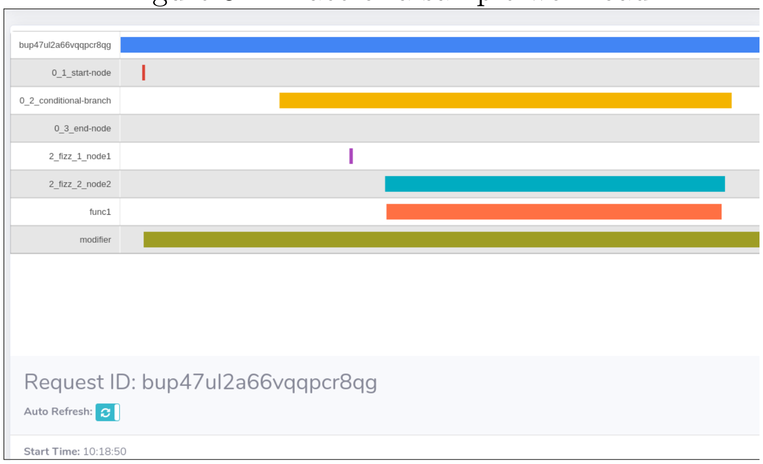
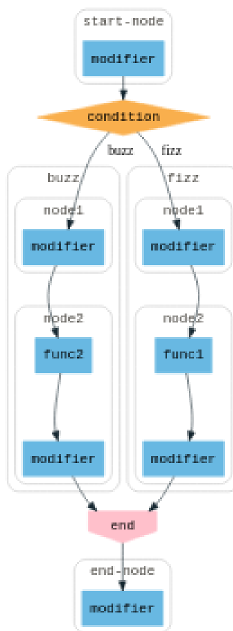


Figure 33: DAG of a sample workload



4.2.2 Data store library

To store the ephemeral data between the functions like was mentioned earlier we propose the usage of distributed in-memory cache which can be easily scaled up and down. Our consideration was both Pocket and Olric. For the implementation though, due to technical difficulties we opted the usage of Olric.

Olric was deployed as Kubernetes pods one per each node in the cluster to ensure that the data transfer time among the storage and the functions are minimum. To communicate to this library, we added a data store management library for Orlic extending the faas-flow library [64].

The library integrates effortlessly with the DataStore object of faas-flow library. The DataStore interface of faas-flow first looks for a Init endpoints that initializes the connection with the deployed Olric setup, configures the timeouts and maximum number of connections. The Olric Go library is used to build the interface. During this initialization step we also specify the serialization library that needs to be used. We use Go's NewMsgPack as the serialization library. MessagePack [65] is a binary serialization format that functions a bit like JSON. Small integers are encoded into a single byte, and typical short strings require only one extra byte in addition to the strings themselves. This makes it a bit better than JSON in the performance.

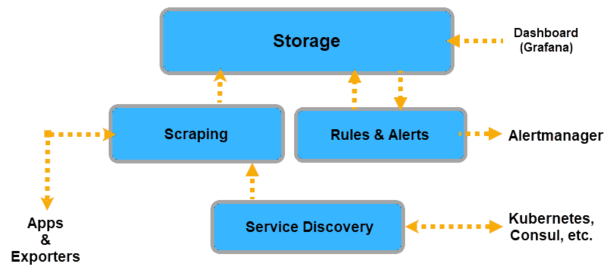
For each new request that is created on the faas-flow pipeline, our library creates a new distributed map to store the key value pairs. On the termination of the request, this distributed map is deleted. Alongside we take of the logic to set and get values to and from the distributed map during the FINISH and RUNNING event changes on the workflow framework. The handling of the events are done inherently by faas-flow as was explained earlier.

4.2.3 Monitoring & usage tracking

We looked in detail at Prometheus as a monitoring tool in the previous section. As a part of the thesis implementation, we deployed Prometheus and Grafana to monitor the usage of the resources and scaling of the platform. The data needs to be appropriately exposed and formatted so that Prometheus can collect it. Prometheus can access data directly from the app's client libraries or by using exporters. An exporter is a piece of software sitting next to the application that sends data to prometheus to scrape. Exporter basically accepts HTTP request from Prometheus and provide the data to Prometheus. For Prometheus to know what data to pull, it uses a Service Discovery. For example, In the case of Kubernetes cluster, this Service

Discovery is done via Kubernetes API since it already has labels, names, etc. for all the applications in it. Figure 34 shows the way Prometheus connects with applications and process the data.

Figure 34: Prometheus workflow



Along with the application information, we should get information about the nodes such as disk usage, CPU usage, etc. We use node exporter [66] for this. Along with this, we also have to export the information about the cgroups that make up Docker containers. For this we use Google's cadvisor project [67]. Once this connection is made, we can use the PromQL language to query from the scraped time series data.

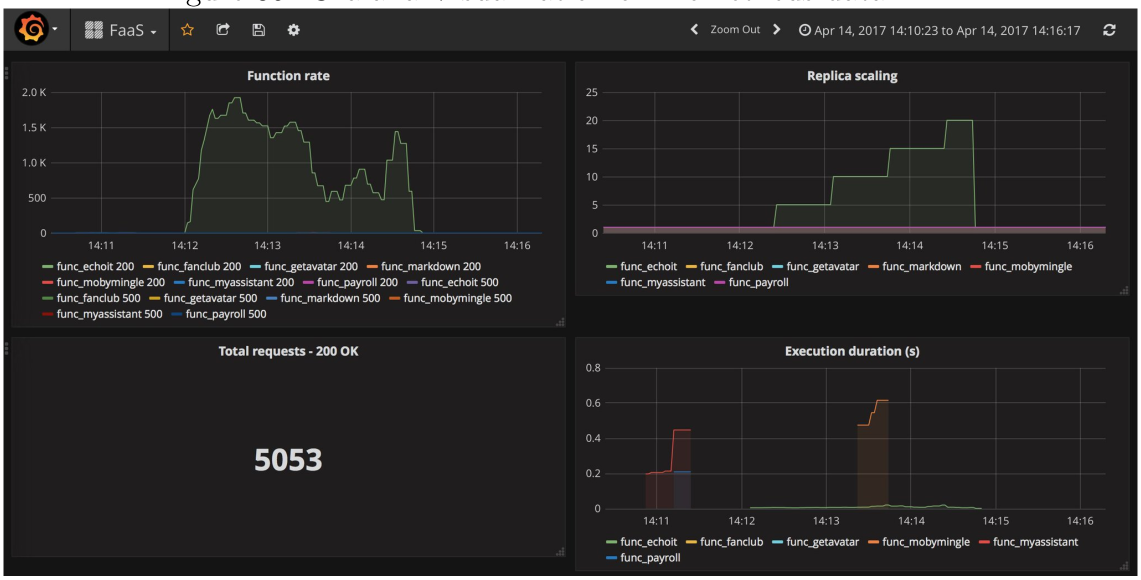
Prometheus can be easily installed on Kubernetes with the help of YAML configurations. Then in the configuration to specify the scraper called $\text{scape}_{\text{config}}$, provide the Kubernetes API, kubectl. The output of the PromQL queries can be used to track the usage by the applications in the infrastructure along with the responsiveness of the system with respect to the scaling, etc. Figure 35 represents a monitoring visualization setup for a serverless infrastructure.

5 Evaluation

To analyze the efficiency of our system, we thought of the right kind of metrics that would quantitatively measure the improvement in the issues we were trying to resolve. We wanted to understand how the composition technique and the ephemeral storage affects the following aspects of the system:

- Overall pipeline execution
- Individual function execution
- Scaling of individual functions

Figure 35: Grafana Visualization of Prometheus data



- Resource usage

We are not measuring the cold start latencies of the system since the mechanism we proposed would not alter anything in this scenario. Also our system maintain a warm pool of instances to avoid the initial cold start latency.

To empirically evaluate the above mentioned aspects, we track several metrics via the monitoring and tracing systems we have setup for the infrastructure. For each workload we test, we record the following details:

- **Trace of the whole function orchestra (Overall execution time):** Overall execution of a function orchestra comprises usually comprises the cold start time, the time to store and retrieve intermediate data and the function execution of the whole pipeline. We are hoping to find a reduction in the execution time on our proposed solution.
- **Span of individual function (Single function execution time):** Our tracing platform effectively records the span of each function. We expect a reduction here as well.
- **The percentage of requests that were handled by the system:**

We measure the number of requests that hit the OpenFaaS gateway the number of successful requests. This measures if our system ever drops a request during heavy load situation. A successful execution is considered as the ones that return 200 status code. We can measure this value for each second and calculate the throughput of the system. We will set a 30s timeout on open requests which will be marked as failed.

- **Memory and CPU usage:** We measure the Memory and CPU usage by each pipeline. This is to learn if our method produced any unexpected spikes in CPU usage and memory and also to test our theory on having a easily billable infrastructure.
- **Scaled parameter:** We define a scaling logic for our setup. We will measure the responsiveness of the platform to heavy load, how easily it scales up and down.

For measuring the improvement our thesis has made to the platform, we compare the platform with the standard practices used for function orchestrations using OpenFaaS. We choose to stick to testing workload on the same platform since certain latencies are very much platform dependent. It would not be very enlightening if we compare a setup of our thesis on OpenFaaS to a commercial setup like AWS Lambda. We design our workloads on OpenFaaS in three different ways:

- **Manual orchestration:** We compose the functions by issuing an HTTP request from each other which is the suggested way of doing it by AWS in the absence of step functions [21].
- **Orchestration with faas-flow and object store:** We compose the functions using faas-flow but use an object store for storing the intermediate data that needs to transferred between the functions
- **Orchestration with faas-flow and state store:** Here we compose the functions with faas-flow but use the in-memory data store for the intermediate storage

5.1 Setup and tools

For the proof of concept, We setup OpenFaaS on single node Kubernetes cluster setup on AWS. Each machine has 2 vCPUs and 8GB of dedicated memory. We used the helm charts [68] of OpenFaaS to setup the FaaS platform. We extended or removed memory limits / quotas for each service and function. For the test setup we turn off the debug flag so that the logs for the function are kept sparse. Like was mentioned in the Implementation section, we connect Prometheus, Grafana and Jaeger tracing with the setup to get fine grained usage and runtime data from the environment. It is worth noting that we use minikube [72] to setup the cluster which technically runs over KVM hypervisor. This introduces some latency to the setup.

In the same cluster we setup Olric, the in-memory distributed cache of our choice, with one pod in each of the nod in the cluster. We connect Olric to Prometheus to get the usage of the memory by the state store. To test our second type of orchestration, the one with the object store, we chose an Open Source object store called Minio [69]. Minio has the functionalities very similar to AWS s3 with an almost similar CLI library. We use the pre-built helm charts of Minio as well for the setup.

One we had the whole setup ready, we configured the platform to have no limits over scaling factor of the function. Along with this we define the rule for scaling the function instances. We make this a function of the number of invocations to have a load based scaling strategy. Like was discussed during the description of OpenFaaS, AlertManager is responsible for firing requests for scaling up and down to the OpenFaaS gateway as per the metrics tracked by Prometheus. So we write the custom AlertManager rule on Prometheus with PromQL. The logic we specify is as follows:

```
alert: APIHighInvocationRate
expr: sum(rate(gateway-function-invocation-total\{code=
   "200"\}[10s])) BY (functionName) > 5
for: 5s
```

The above query groups the invocation by the function name. If the sum of successful invocations is above 5 within 5 seconds, the gateway scales up the function instances running by a certain factor. We define this factor to be 20% in the OpenFaaS configuration file for our functions later.

For load testing we use the HTTP load mocking library called hey [71]. hey is a lightweight replacement for ApacheBench. hey runs provided number of requests in the provided concurrency level and prints stats. It can be used as a command as follows:

```
hey -q 1 -c 1 -z 15m 'http://<server-ip>/function/nodeinfo'
```

The above command sends 1 request per second for a total of 15 minutes to the function called nodeinfo in the local OpenFaaS setup.

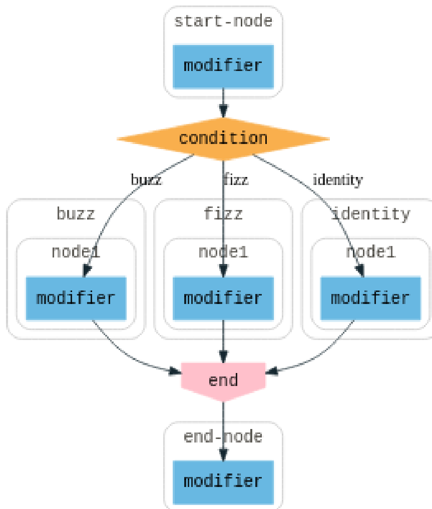
5.2 Workload

We use a very simple workload to do the function composition with. We implement the famous Fizz buzz children's puzzle [70]. It is a simple function that checks if a number is divisible by 5, in which case it replaces the number with buzz and if it is divisible by 3, the number is replaces by fizz. We split this into multiple functions. The first function checks if the number is divisible by 3 or by 5 and conditionally it chooses the next function which either is: the function that return fizz, the function that returns buzz, or the function that returns the same number. This function is great because we can test the conditional branching in the composition. It is not particularly CPU intensive or memory intensive. We do not want such workloads since our scaling heuristics depend on the number of requests.

Figure 36 shows the DAG generated from the composition. Like one can notice, based on the conditional(divide by3, divide by 5, or neither) the flow can take one of three branches. At the end, the end node returns the value in a unified format.

To further understand how the variable number of functions can affect the performance of our proposed system, we tried next to increase the number of functions that are chained together. We wrote a composition that would get passed in a number, and the system will create a DAG with that many nodes(functions) in it. To compare the performance to standard setups, we wrote a manual composition where the the function is pass in a number, which in turn calls another instance of itself with a number one less that what it got. If the number it got is zero, it just returns the number and do nothing. This dynamically creates a chain of the number of functions we want. This workload is much less CPU and memory intensive. We are only trying to see how well it perform for long function compositions.

Figure 36: Fizz buzz workload



5.3 Results and Analysis

To begin the experiment with we used hey command line tool to produce three different kind of load for the above three different workloads. We tried a load of 1 request every 1 second for 1 minute, a load of 50 requests concurrently for 40 seconds. We noted the metrics on hey, average spans on the tracing framework, scaling frequency and the CPU and memory usage for all the mentioned workloads as follows:

5.3.1 faas-flow with ephemeral storage

In the first setup we try emulate an HTTP load of one request per second for a whole minute using the testing tool hey like mentioned earlier. The command would be as follows:

```
hey -q 1 -c 1 -z 1m 'http://<server-ip>/function/fizz-buzz-olric'
```

We notice that all 60 requests that were sent were successfully handled by the platform. We get the benchmark information from hey like is shown in Figure 37.

Figure 37: Benchmark summary from hey for composition with ephemeral storage

```
Summary:
  Total:        60.0204 secs
  Slowest:      0.0628 secs
  Fastest:      0.0129 secs
  Average:      0.0168 secs
  Requests/sec: 0.9997


Response time histogram:
  0.013 [1]     |■
  0.018 [49]    |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.023 [6]     |■■■■■
  0.028 [1]     |■
  0.033 [0]     |
  0.038 [2]     |■■
  0.043 [0]     |
  0.048 [0]     |
  0.053 [0]     |
  0.058 [0]     |
  0.063 [1]     |■


Latency distribution:
  10% in 0.0134 secs
  25% in 0.0140 secs
  50% in 0.0145 secs
  75% in 0.0172 secs
  90% in 0.0211 secs
  95% in 0.0331 secs
  0% in 0.0000 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0000 secs, 0.0129 secs, 0.0628 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0003 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0001 secs
  resp wait:    0.0166 secs, 0.0127 secs, 0.0620 secs
  resp read:    0.0001 secs, 0.0000 secs, 0.0001 secs
```

It is seen that on average the pipeline took 0.0168 seconds.

Further looking at the function span, it can be seen that the starting node in the composition is the one that takes up the most time in the execution. On average this is seen as 0.013s.

We try increasing the load on the function so we can see how well it scales. We invoke an HTTP mock load of around 200 requests per second for a period of 1 minute. All of the requests were handled by the platform. The hey benchmarking details are as in Figure 38

Figure 38: Benchmark summary from hey for composition with ephemeral storage with higher load

```
Summary:
  Total:        2.1332 secs
  Slowest:      0.6185 secs
  Fastest:      0.0142 secs
  Average:      0.1895 secs
  Requests/sec: 234.3870

  Total data:   3836 bytes
  Size/request: 7 bytes

Response time histogram:
  0.014 [1]     |
  0.075 [7]     |■■
  0.135 [134]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.195 [167]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.256 [86]    |■■■■■■■■■■■■■■■■■■■■■
  0.316 [65]    |■■■■■■■■■■■■■■■■
  0.377 [27]    |■■■■■■
  0.437 [8]     |■■
  0.498 [4]     |■
  0.558 [0]     |
  0.618 [1]     |


Latency distribution:
  10% in 0.1017 secs
  25% in 0.1271 secs
  50% in 0.1819 secs
  75% in 0.2345 secs
  90% in 0.3060 secs
  95% in 0.3388 secs
  99% in 0.4740 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0001 secs, 0.0142 secs, 0.6185 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0014 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0008 secs
  resp wait:    0.1893 secs, 0.0141 secs, 0.6184 secs
  resp read:    0.0000 secs, 0.0000 secs, 0.0015 secs
```

We can see that there is a bit of heavy variance in the response times of the requests. On average the response time is about 0.1895 seconds which is more than the previous case. This latency can be attributed to the cold start delay during the scaling.

We now try and increase the number of functions in the chain dynamically. We write a script that will pass a number into the flow function, and the flow function dynamically creates a DAG of echo functions that serially chains the that many number of functions. The DAG that was dynamically created looks like Figures 39 and 40 for 2 and 6 nodes respectively. In each of these compositions, we will load test by variable load of input messages and see how well they perform in comparison to the manual compositions. We present the benchmark found on this comparison in the Analysis section.

Figure 39: Dynamically created composition with 2 nodes

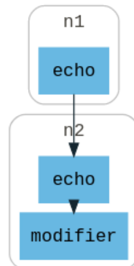
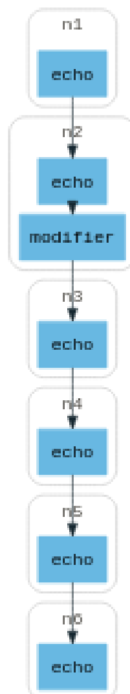


Figure 40: Dynamically created composition with 6 nodes



We can find an interesting corelation here between the scaling and the function execution time. We start the system by maintaining a warm pool of 30 nodes for higher number of requests. When we send out 50 requests at once, the system scale up according to the alermanager rule that we defined. It can be seen from the charts of the function rates and the execution duration how when the function is scaling up the execution time starts increasing steadily. This can be referred from Figure 41 and Figure 42. These are charts from Grafana monitoring taken directly. In the former you can see two plots under the execution time. The yellow one belongs to the time spent on the start node and the lower one is of the function that tracks the trace metrics of the same function. Although this adds some latency, this was necessary to easy tracking of data. In the latter you can see the scaling rate of the same node.

Figure 41: Execution time - Composition with ephemeral storage

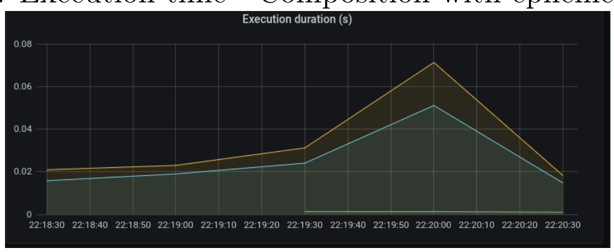
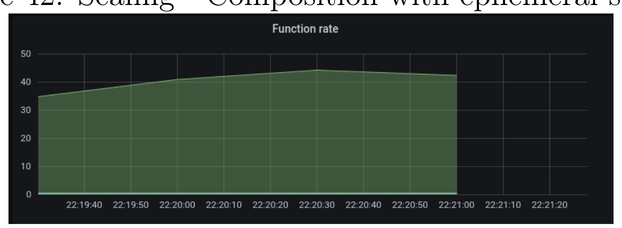


Figure 42: Scaling - Composition with ephemeral storage



5.3.2 faas-flow with block storage

The first scenario as was done with the previous workload, we use one request per second for a minute HTTP request load as follows.

```
hey -q 1 -c 1 -z 1m 'http://<server-ip>/function/fizz-buzz-minio'
```

In this case as well all the requests were successfully handled by the platform (200OK response). The benchmark details from hey CLI is as in Figure 43

```
Summary:
  Total:        60.0342 secs
  Slowest:      0.0912 secs
  Fastest:      0.0125 secs
  Average:      0.0184 secs
  Requests/sec: 0.9994


Response time histogram:
  0.013 [1]     |■
  0.020 [50]    |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.028 [3]     |■■
  0.036 [3]     |■■
  0.044 [1]     |■
  0.052 [1]     |■
  0.060 [0]     |
  0.068 [0]     |
  0.075 [0]     |
  0.083 [0]     |
  0.091 [1]     |■


Latency distribution:
  10% in 0.0134 secs
  25% in 0.0138 secs
  50% in 0.0153 secs
  75% in 0.0175 secs
  90% in 0.0289 secs
  95% in 0.0394 secs
  0% in 0.0000 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0000 secs, 0.0125 secs, 0.0912 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0002 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0001 secs
  resp wait:    0.0183 secs, 0.0124 secs, 0.0910 secs
  resp read:    0.0001 secs, 0.0000 secs, 0.0001 secs
```

Figure 43: Benchmark summary from hey for composition with block storage

On an average the requests took 0.0184 seconds to complete the whole pipeline. When we look at the function spans of each function, it can be seen that on average the modifier node which represents the data modifier takes up quite a bit of time along with the start node. This was shown as 0.00716s and 0.00932s respectively.

Similar to the above section, we repeat the experiment with a heavier load to see how well the scaling is managed by the system. We get an average response time of 0.3646 seconds. But in this experiment we had 3 timeouts. According to the trace, there timeouts happened during the writing phase to the Minio. Figure 44 represents the benchmark information. The 3 requests that exceeded 1 minute in execution are the ones that timed out.

```
Summary:
  Total:        2.1556 secs
  Slowest:      1.0821 secs
  Fastest:      0.0047 secs
  Average:      0.3646 secs
  Requests/sec: 231.9573

  Total data:   5480 bytes
  Size/request: 10 bytes

Response time histogram:
  0.005 [1]     |
  0.112 [31]    |■■■■■■■■■■
  0.220 [116]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.328 [113]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.436 [88]    |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.543 [69]    |■■■■■■■■■■■■■■■■■■■■■■■
  0.651 [26]    |■■■■■■■■
  0.759 [26]    |■■■■■■■■
  0.867 [20]    |■■■■■■
  0.974 [7]     |■■
  1.082 [3]     |■


Latency distribution:
  10% in 0.1496 secs
  25% in 0.2016 secs
  50% in 0.3101 secs
  75% in 0.4976 secs
  90% in 0.6821 secs
  95% in 0.7961 secs
  99% in 0.8890 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0004 secs, 0.0047 secs, 1.0821 secs
  DNS-lookup:   0.0001 secs, 0.0000 secs, 0.0070 secs
  req write:    0.0002 secs, 0.0000 secs, 0.0021 secs
  resp wait:    0.3640 secs, 0.0046 secs, 1.0821 secs
  resp read:    0.0000 secs, 0.0000 secs, 0.0005 secs
```

Figure 44: Benchmark summary from hey for composition with block storage - higher load

We do notice in this case as well a general spike in execution time when the functions are getting scaled up after the warm pods we have kept on.

5.3.3 Manual composition with block storage

For the manual composition, basically we wrote simple python functions of the same functionality that would call each other via simple HTTP REST calls using requests [73] module. This is quite comparable with the recommended setup recommended by some of the commercial FaaS offerings [20]. The very same HTTP request load was applied first on manual composition to yield a result as in Figure 45

```
hey -q 1 -c 1 -z 1m 'http://<server-ip>/function/fizz-buzz-manual'
```

Figure 45: Benchmark summary from hey for manual composition with block storage

```
Summary:
  Total:        60.3179 secs
  Slowest:      0.7717 secs
  Fastest:      0.3007 secs
  Average:      0.3540 secs
  Requests/sec: 0.9947

  Total data:   3600 bytes
  Size/request: 60 bytes

Response time histogram:
  0.301 [1]     |■
  0.348 [38]    |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.395 [12]    |■■■■■■■■■■■■■
  0.442 [2]     |■■
  0.489 [4]     |■■■■
  0.536 [0]     |
  0.583 [0]     |
  0.630 [1]     |■
  0.678 [1]     |■
  0.725 [0]     |
  0.772 [1]     |■


Latency distribution:
  10% in 0.3033 secs
  25% in 0.3085 secs
  50% in 0.3141 secs
  75% in 0.3687 secs
  90% in 0.4543 secs
  95% in 0.5967 secs
  0% in 0.0000 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0000 secs, 0.3007 secs, 0.7717 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0003 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0001 secs
  resp wait:    0.3539 secs, 0.3006 secs, 0.7716 secs
  resp read:    0.0001 secs, 0.0001 secs, 0.0002 secs
```

We can see that on average the function takes 0.33 seconds to complete a request. Even in this case, we could see that all the requests were handled by the system successfully. We could not measure spans of individual functions exactly in this case because of the missing tracing infrastructure for this.

In the second phase of the test on this setup, as with the previous two cases, we mock above 200 requests per second on the manual composition pipeline. There was a serious slowness in the scaling of the function in this scenario. The function that is being called from another function did not scale well since it is not being called simultaneously and would not trigger the Alertmanager rule. Hence there were quite a few timeouts.

Figure 46: Benchmark summary from hey for manual composition with block storage and heavy load

```
Summary:
  Total:        10.6331 secs
  Slowest:      10.6321 secs
  Fastest:      7.0268 secs
  Average:      8.7022 secs
  Requests/sec: 1.8809

  Total data:   1200 bytes
  Size/request: 60 bytes

Response time histogram:
  7.027 [1]     |■■■■■■■■■■■■■■■■■■■■
  7.387 [2]     |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  7.748 [2]     |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  8.108 [2]     |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  8.469 [2]     |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  8.829 [2]     |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  9.190 [2]     |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  9.551 [1]     |■■■■■■■■■■■■■■■■■■■■
  9.911 [2]     |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  10.272 [2]    |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  10.632 [2]    |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■


Latency distribution:
  10% in 7.2230 secs
  25% in 7.8208 secs
  50% in 8.7201 secs
  75% in 9.8203 secs
  90% in 10.4514 secs
  95% in 10.6321 secs
  0% in 0.0000 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0006 secs, 7.0268 secs, 10.6321 secs
  DNS-lookup:   0.0004 secs, 0.0000 secs, 0.0008 secs
  req write:    0.0002 secs, 0.0000 secs, 0.0006 secs
  resp wait:    8.7013 secs, 7.0254 secs, 10.6315 secs
  resp read:    0.0001 secs, 0.0001 secs, 0.0001 secs
```

Figure 46 shows the staggering slowness and network clogging that happened during this test. On an average a request took around 8 seconds to complete a round trip. We could not notice clean scaling either in this setup since only the coordinator functions kept scaling and not the individual functions.

Like with the faas-flow composition with ephemeral storage, we tried to dynamically compose variable number of functions and see how the chaining factor increases or decreases the composition efficiency. We found that the manual composition fall short in independently scaling the functions and the

Table 1: Execution times of the compositions

Composition	Workload	Average response time(s)	Fastest response time(s)	Slowest response time(s)	Timeouts
Manual Composition	~1 request per second	0.3540	0.3007	0.7717	1
Manual Composition	~230 request per second	8.7822	7.0268	10.6321	4
Faas-flow with Object Store	~1 request per second	0.0184	0.0125	0.0912	0
Faas-flow with Object Store	~230 request per second	0.3646	0.0047	1.0821	0
Faas-flow with in-memory cache store	~1 request per second	0.0168	0.0129	0.0628	0
Faas-flow with in-memory cache store	~230 request per second	0.1895	0.0142	0.6185	0

error handling and network latency affects the efficiency a lot in the system. We present the values we found in the next section.

5.3.4 Analysis

Consolidating the above collection data, we create the Table 1

The initial analysis is driven from the first scenario of our testing setup. When we look at the scenario with one request per second, we can understand that on average each function takes an average of 0.0168 seconds in a setup with faas-flow + ephemeral storage, 0.0184 seconds for faas-flow + block storages and a staggering 0.3540s for the manual composition. These patterns were repeated even when we increased the load to have heavy concurrent requests.

The problem that was noticed mainly with our proposed solution is the cold start during the scaling up scenario. The reasons of this can be attributed to the slow virtualization mechanism we are using. Although we did see that we had an almost immediate scaling down when the load goes down. This is actually really great for resource preservation.

With the setup using block storage, the modifier functions seem to be taking longer which is the added latency of accessing the block storage. We could not really achieve clean scaling up with the manual composition setup of the same.

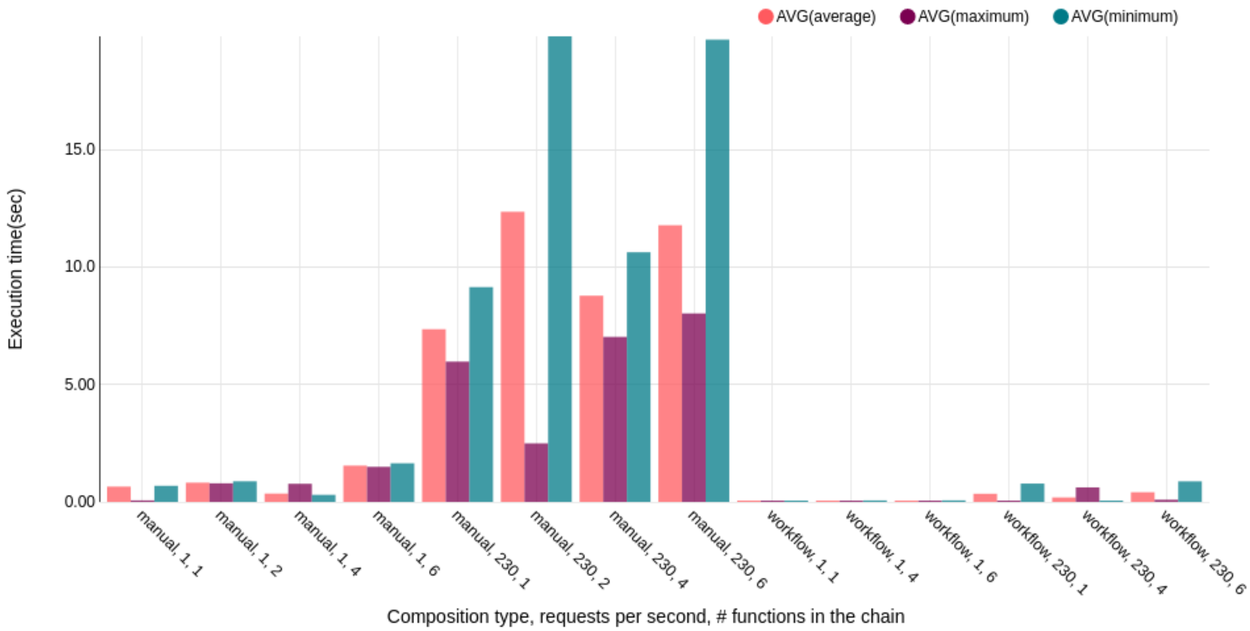
To see the effects of increasing the number of functions in the chain has on the overall efficiency, we created a script that dynamically creates DAGs on our platform. We present results we got from using 1 function, 2 function and 6 function chains on simple load and heavy load in the faas-flow and manual composition setups. We got results that we were actually expecting. We saw that in manual composition, the longer the chain the longer the chances for timeouts and broken pipes. We saw that our compositional method works exceptionally well with longer chains, especially in regards to scaling up each layer independently. We present the results we got in Table 2.

We consolidate the data from the tables by plotting it using a bar chart. We used Apache Superset to plot the chart as a function of execution time with respect to composition type, requests per second and the number of functions in the composition chain. The plot can be referred in Figure 47 We found that the compositions worked especially when irrespective of the number of chains in it. The functions were independently scaling. There was always a slight latency that we encountered on node 1 which can be attributed to the cold start latency. The warm pools of functions kept ready improved the performance of our system quite a lot especially in the compositions that has larger number of chains. We could see that in the case of manual compositions, as the chains got longer, the functions started having a very high average response time. This is because the functions are technically waiting until the last of the functions returns to terminate. In the case of the faas-flow compositions, once the functions have passed on the SUCCESS event to the NATstreaming event bus, they can basically idle or be killed and hence saving resources. It is to be noted that although efficient, our system had about 3 timeouts when the number of requests became too many on a longer composition. This is because of the openfaas gateway did not handle certain requests in the case where more requests were being passed around between functions.

Table 2: Different composition lengths - Execution times of the compositions

Composition	Number of Functions	Workload	Average response time(s)	Fastest response time(s)	Slowest response time(s)	Timeouts
Manual Composition	1	~1 request per second	0.6516	0.6394	0.6851	1
Manual Composition	1	~100 request per second	7.3540	5.9746	9.1473	1
Manual Composition	2	~1 request per second	0.8195	0.7920	0.8783	0
Manual Composition	2	~100 request per second	12.3560	2.4926	19.8284	19
Manual Composition	6	~1 request per second	1.5463	1.4946	1.6452	5
Manual Composition	6	~100 request per second	11.7822	8.0268	19.6878	25
Faas-flow with in-memory cache store	1	~1 request per second	0.0172	0.0121	0.0498	0
Faas-flow with in-memory cache store	1	~100 request per second	0.3463	0.0431	0.7810	0
Faas-flow with in-memory cache store	2	~1 request per second	0.0162	0.0141	0.0323	0
Faas-flow with in-memory cache store	2	~100 request per second	0.3882	0.0605	1.1607	0
Faas-flow with in-memory cache store	6	~1 request per second	0.0189	0.0140	0.0647	0
Faas-flow with in-memory cache store	6	~100 request per second	0.4117	0.0927	0.8759	3

Figure 47: Execution time of different compositional strategies under different conditions



6 Related work

Serverless has gained a lot of attention and traction from the scientific community in the past few years because of its massive implications in resource conservation and innovative programming when one does not have to worry about compute management anymore. The issues that were discussed in sessions above are being studied by various studies and the most significant ones are worth noting.

Before getting into the studies that focus on the issues that was covered in this paper, it is interesting to have a look at a very recent literature review [53] . In the paper the authors analyze 112 different academic papers and grey journals in and around the paradigm of FaaS were analyzed. The researchers found a staggering lack in the practicability of the work that were proposed by the scientific community. Along with the lack of reusability and reproducability, it was found that 88% of these proposals were worked in and around AWS lambda, which is not very universal as FaaS solution especially considering its vendor locked in and closed source attributes. The study also mentions how most of these works being done focus on unrealistic workloads that are not very common in the production setups in the industry. The paper also says how the current research lacks methods to chain and branch functions in a meaningful way.

In [54], the authors interestingly look at the issues that the state of art isolation mechanisms in FaaS infrastructure bring forward as was mentioned earlier. These include the lack of security and the heavy cold start time. It introduces faaslets, an alternate isolation policy to be used instead of containers. With this, faaslets can share data across instances there by reducing data transfer costs. In a contemporary study [55], an orchestration mechanism called TriggerFlow is introduced. It is a really interesting tool to manage the lifecycle of a cloud function. In this smart triggering system, function composition is allowed using Distributed Acyclic Graphs(DAG) to define control flow and data flow in the pipeline. This has huge potential as an idea, although currently the usability of the platform is terrible and it can be quite bloated as a entry point to a FaaS system especially since it is not a very elastic platform. In an older research, and idea was proposed to schedule events based on tags which was quite similar. But in a comparison, it is stated that the solution has a heavier memory footprint than the former.

Cloudburst [2] and SAND [3] are projects that were mentioned in the previous section. In the former, they suggest adding a key value cache along with a limited DAG based language to specify the composition was specified before. Although a very interesting idea, the issues with this systems were discussed previously. SAND is a very interesting idea as well where they use a different kind of isolation scheme to allow function composition as opposed to containers.

In yet another recent paper [4], a theoretical model for a composition language called serverless composition language(SPL) which lets the programmer define function compositions(even can be higher order functions). This paper has some very interesting formal foundations for serverless as a technology which was used as a reference.

A very intriguing idea that has been proposed in the research community is to change the programming model of serverless paradigms completely and introduce a function shipping architecture for serverless. The idea is that it is suggested that the way FaaS functions are designed is actually a architectural anti-patterns that system designers make [13]. Currently the pattern can be referred as data shipping. Meaning that data is shipped to the function as opposed to a function shipping architecture. An example for a function shipping architecture would be procedures in databases where data is not moved from its storage location. The reason why the data shipping pattern is bad is because of the fact that across different storage layers and network layers, there is a vast spectrum in the memory hierarchy which adds heavy latencies. Shredder [56] was a work towards adopting a function coding pattern by adopting v8 isolation mechanism to boot up light weight instances of the function near to the storage layer of the system. The problem with this method is the fact that the current data loads are extremely heterogeneous and it is hard to support this system on all the storage platforms. But it is a very ambitious idea that has a lot of potential.

Coming to the domain of ephemeral scalable storage, Pocket is a very significant project which was described in detail earlier. Anna KVS [57], is a similar idea which was adopted in the Cloudburst project. The tool was not adopted in this project mostly because of the low elasticity the tool offers.

In InfiniCache [58], a memory object cache is used to store the ephemeral state in the system. It uses erasure coding and data backup to ensure high availability. They try to get this system working on AWS lambda by connecting the runtime to a priority based queue. In a very recent paper [61], a shared filesystem is being introduced that can be shared among the functions to transfer intermediate data among themselves. Currently a very theoretical suggestion, FaasFS has the potential to be an interesting mode of handling the intermediate data issue.

7 Future work

The idea we presented here has a lot of potential for innovation mostly due to its ease of adaptability and simplistic design. A very obvious improvement to the platform would be the adaptation of a different isolation mechanism instead of Docker containers. Since Docker containers are rather heavy as an isolation mechanism, the cold start penalty in case of no warm nodes is still high. The log during the scale up and down is due to this latency. A much lighter isolation mechanism like v8 isolates [62] might help in making the system faster. The direction taken by FaaSM [54] project is very applicable in this scenario. This only tightens the security of the infrastructure since Docker containers can inherits issues of the host operating system.

Furthermore, we plan on extending the project to provide better fault tolerance. Since the workflow inherently supports conditional branching, we can extend the API to accept callback functions in case of a function failure. With this the developer can define the retry logic or the failure logic, thereby avoiding an entire restart of the pipeline or worst - data corruption. We could also improve the message queuing system to get an exactly one guarantee on the event handling.

Another improvement that we would like to make to our system is to introduce intelligent data locality for the intermediate data. In the present architecture, the data is being stored in the distributed cache which might store it in any of the partition across the cluster. But we can introduce logic that would place the data in the same machine as that of the function. This is a very simplistic adaptation of the "porting function to the data logic". This would reduce the latency spent of data transfer before the function invocation.

For developer friendliness, we can support multiple client libraries for the application that would allow development of the workflow code in different languages.

8 Conclusion

In this paper, we analyzed a function composition solution and extended it to use in-memory cache to store intermediate data between the functions. Our initial thesis was that this would be a much better solution for big data function pipelines, in which the developer would be bound to use a third party block storage or pass via network both of which costs quite a bit on latency. The proposed solution provides a very neat at efficient way to compose the functions, reducing the latency of the manual orchestrations which has a higher rate of gateway timeouts or the latency of block storages that adds the I/O bottleneck. The system is very flexible allowing numerous operations while composing the function like branching, looping, etc. This adds the possibility of extending the platform for to support automatic roll-backs, error correction, etc. Having said that, the system has limitations like slow scaling up at certain instances due to the cold start latency. The system uses NATS streaming for message queuing which is extremely simple and elastic but does not guarantee exactly once semantics for the message delivery. Like was mentioned in the section above, we believe that the improvements in the virtualization technology can improve the latencies further.

9 References

1. Gojko Adzic and Robert Chatley. "Serverless Computing: Economic and Architectural Impact". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, 884–889. ISBN: 9781450351058. DOI: 10.1145/3106237.3117767. URL: https://doi.org/10.1145/3106237.3117767

2. Vikram Sreekanti et al. "Cloudburst". In: *Proceedings of the VLDB Endowment* 13.12 (2020), 2438–2452. ISSN: 2150-8097. DOI: 10.14778/3407790.3407836. URL: http://dx.doi.org/10.14778/3407790.3407836

3. Istemi Ekin Akkus et al. "SAND: Towards High-Performance Serverless Computing". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 923–935. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/akkus

4. Abhinav Jangda et al. "Formal Foundations of Serverless Computing". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360575. URL: https://doi.org/10.1145/3360575

5. Antonia Pollock. *Virtualization vs. Containerization.* URL: https://www.liquidweb.com/kb/virtualization-vs-containerization/

6. *The economics of serverless computing: A real-world test.* URL: https://techbeacon.com/enterprise-it/economics-serverless-computing-real-world-test

7. Rafal Gancarz. *The economics of serverless computing: A real-world test*. June 2019. URL: https://techbeacon.com/enterprise-it/economics-serverless-computing-real-world-test

8. Rohit Akiwatkar. *AWS Lambda Pricing: How Much it Costs to Run a Serverless Application?* Apr. 2019. URL: https://www.simform.com/aws-lambda-pricing/

9. Alex Handy. *Amazon introduces Lambda, Containers at AWS re:Invent*. Nov. 2014. URL: https://sdtimes.com/amazon/amazon-introduces-lambda-containers/

10. Rohit Akiwatkar. *AWS Lambda vs Azure Functions vs Google Cloud Functions: Comparing Serverless Providers*. Nov. 2020. URL: https://www.simform.com/aws-lambda-vs-azure-functions-vs-google-functions/

11. *Cloudflare*. URL: https://www.cloudflare.com/

12. *Edge computing*. URL: https://en.wikipedia.org/wiki/Edge_computing

13. Joseph M. Hellerstein et al. *Serverless Computing: One Step Forward, Two Steps Back*. 2018. arXiv: 1812.03651 [cs.DC]

14. Krish. *On the Serverless cold start problem*. June 2019. URL: https://medium.com/faun/on-the-serverless-cold-start-problem-2fc0797da5cc

15. Mikhail Shilkov. *Serverless: Cold Start War*. Aug. 2018. URL: https://mikhail.io/2018/08/serverless-cold-start-war/

16. Andrea Passwater. *2018 Serverless Community Survey: huge growth in serverless usage*. 2018. URL: https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage

17. Daniel Barcelona-Pons and Pedro García-López. *Benchmarking Parallelism in FaaS Platforms*. 2020. arXiv: 2010.15032 [cs.DC]. URL: https://arxiv.org/abs/2010.15032

18. Alan Williams. *Autodesk Goes Serverless in the AWS Cloud, Reduces Account-Creation Time by 99%*. 2019. URL: https://aws.amazon.com/solutions/case-studies/autodesk-serverless/

19. *AWS Step Functions*. 2019. URL: https://aws.amazon.com/step-functions

20. Sunil Mallya. *Ad Hoc Big Data Processing Made Simple with Serverless MapReduce*. Nov. 2016. URL: https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/

21. V. Giménez-Alventosa, Germán Moltó, and Miguel Caballer. "A framework and a performance assessment for serverless MapReduce on AWS Lambda". In: *Future Generation Computer Systems* 97 (Mar. 2019). DOI: 10.1016/j.future.2019.02.057

22. Ana Klimovic et al. "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, pp. 427–444. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/klimovic

23. *Apache Crail*. URL: https://crail.apache.org/

24. *Olric*. URL: https://github.com/buraksezer/olric

25. *Consistent Hashing Algorithm*. URL: https://github.com/buraksezer/consistent

26. *Redis SETNX, url = https://redis.io/commands/setnx,*

27. *Cluster multi-tenancy , url = https://cloud.google.com/kubernetes-engine/docs/concepts/mult overview,*

28. Isaac Odun-Ayo et al. "Cloud Multi-Tenancy: Issues and Developments". In: Dec. 2017, pp. 209–214. DOI: 10.1145/3147234.3148095

29. *Kubernetes namespaces, url = https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/,*

30. *Kubernetes Policies, url = https://kubernetes.io/docs/concepts/policy/limit-range/,*

31. *Kubernetes Pod Security Policy, url = https://kubernetes.io/docs/concepts/policy/pod-security-policy/,*

32. Adnan Rahic. *Serverless monitoring - the good, the bad and the ugly.* 2019. URL: https://www.serverless.com/blog/serverless-monitoring-the-good-the-bad-and-the-ugly

33. Chrissy Kidd. *Tracing vs Logging vs Monitoring: What's the Difference?* Mar. 2019. URL: https://www.bmc.com/blogs/monitoring-logging-tracing/

34. *Systemd - Wikipedia Article.* URL: https://en.wikipedia.org/wiki/Systemd

35. Valetio Technologies. *A Comprehensive Tutorial to Implementing OpenTracing With Jaeger.* Feb. 2019. URL: https://medium.com/velotio-perspectives/a-comprehensive-tutorial-to-implementing-opentracing-with-jaeger-a01752e1a8ce

36. *Vendor-neutral APIs and instrumentation for distributed tracing.* URL: https://opentracing.io/docs/

37. Jamie Wilkinson. *Practical Alerting from Time-Series Data.* URL: https://landing.google.com/sre/sre-book/chapters/practical-alerting/

38. Steven J. Vaughan-Nichols. *What is Docker and why is it so darn popular?* 2018. URL: https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/

39. Vaughan-Nichols, *What is Docker and why is it so darn popular?*

40. *Docker Hub.* URL: https://hub.docker.com/

41. *Dockerfile reference.* URL: https://docs.docker.com/engine/reference/builder/

42. ERIC BOERSMA. *Docker Image vs Container: Everything You Need to Know.* May 2019. URL: https://stackify.com/docker-image-vs-container-everything-you-need-to-know/

43. Junfeng Li et al. "Understanding Open Source Serverless Platforms: Design Considerations and Performance". In: *Proceedings of the 5th International Workshop on Serverless Computing.* WOSC '19. Davis, CA, USA: Association for Computing Machinery, 2019, 37–42. ISBN:

9781450370387. DOI: 10.1145/3366623.3368139. URL: https://doi.org/10.1145/3366623.3368139

44. *Faas provider*. URL: https://github.com/openfaas/faas-provider/
45. *OpenFaaS watchdog*. URL: https://docs.openfaas.com/architecture/watchdog/
46. *Horizontal Pod Autoscaler*. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
47. *Introducing the PLONK Stack for Cloud Native Developers*. URL: https://www.openfaas.com/blog/plonk-stack/
48. Sharvari T and Sowmya Nag K. *A study on Modern Messaging Systems-Kafka, RabbitMQ and NATS Streaming*. 2019. arXiv: 1912.03715 [cs.DC]
49. *Prometheus Data Model*. URL: https://prometheus.io/docs/concepts/data_model/
50. *Prometheus Push Gateway*. URL: https://github.com/prometheus/pushgateway
51. *Jaeger Tracing*. URL: https://www.jaegertracing.io/
52. *Jaeger Architecture*. URL: https://www.jaegertracing.io/docs/1.14/architecture/
53. Joel Scheuner and Philipp Leitner. "Function-as-a-Service performance evaluation: A multivocal literature review". In: *Journal of Systems and Software* 170 (2020), p. 110708. ISSN: 0164-1212. DOI: 10.1016/j.jss.2020.110708. URL: http://dx.doi.org/10.1016/j.jss.2020.110708
54. Simon Shillaker and Peter Pietzuch. *Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing*. 2020. arXiv: 2002.09344 [cs.DC]
55. Pedro García López et al. "Triggerflow". In: *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems* (2020). DOI: 10.1145/3401025.3401731. URL: http://dx.doi.org/10.1145/3401025.3401731

56. Tian Zhang et al. "Narrowing the Gap Between Serverless and Its State with Storage Functions". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, 1–12. ISBN: 9781450369732. DOI: 10.1145/3357223.3362723. URL: https://doi.org/10.1145/3357223.3362723

57. C. Wu et al. "Anna: A KVS for Any Scale". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 401–412. DOI: 10.1109/ICDE.2018.00044

58. Ao Wang et al. "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, pp. 267–281. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/conference/fast20/presentation/wang-ao

59. **github**

60. *Faas Flow*. URL: https://github.com/s8sg/faas-flow

61. Johann Schleier-Smith et al. "A FaaS File System for Serverless Computing". In: *ArXiv* abs/2009.09845 (2020)

62. *v8 isolates*. URL: https://v8docs.nodesource.com/node-0.8/index.html

63. *OpenFaaS configurations*. URL: https://docs.openfaas.com/architecture/autoscaling/

64. *Olric Data Store Library*. URL: https://github.com/nandajavarma/faas-flow-olric-datastore

65. *MessagePack*. URL: https://msgpack.org/

66. *Prometheus Node Exporter*. URL: https://github.com/prometheus/node_exporter

67. **cadvisor**

68. *Helm Charts for OpenFaaS*. URL: https://github.com/google/cadvisor

69. *Minio Object Store.* URL: https://min.io/

70. *Fizz buzz Algorithm.* URL: https://en.wikipedia.org/wiki/Fizz_buzz

71. *HTTP load generator.* URL: https://github.com/rakyll/hey

72. *Minikube.* URL: https://minikube.sigs.k8s.io/docs/

73. *Python requests modules.* URL: https://requests.readthedocs.io/en/master/