IJCRT.ORG ISSN : 2320-2882
INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)
An International Open Access, Peer-reviewed, Refereed Journal

Study on Fault-Tolerance for Serverless Computing

FNU Aayoush Dhara Bhadani Siddhesh Gawde
Department of Computer Engineering and Computer Science
California State University, Long Beach, United States

Abstract—Serverless Computing, most often referred as Functionas- a-Service (FaaS), has become popular in recent years because of their operation such as ease-of-use, autoscaling and pay-for-whatyou-use features. FaaS infrastructure provides some fault tolerant measures like automatic retry functions in case of function failure - whether because of application error or infrastructure failure. This retries mechanism ensures that functions are executed at-least once. It encourages developers to write idempotent program which ensures at-most once execution. Combining retry-based atleast once execution and idempotent at-most once execution is insufficient to guarantee exactly once execution. To address this challenge, Vikram Sreekanti et al. have introduced AFT, an atomic fault tolerance shim for serverless applications. It interposes between a commodity FaaS platform and storage engine and ensures atomic visibility of updates by enforcing the read atomic isolation guarantee. In this paper, we are discussing fault-tolerance measures for Serverless Computing and how AFT guarantees read atomic isolation and scales thousands of requests per second smoothly. We also discussing alternative approach for fault tolerance in FaaS based on Active-Standby failover.

Index Terms—Serverless Computing, Faas, Autoscaling, AFT, Active- Standby failover approach

1. Introduction

For cloud solution architects and developers, serverless computing is becoming increasingly appealing. It's advantages varies from simplicity to deployment. Serverless computing most commonly refers to Functions-as-a-Service in today's public clouds (FaaS). FaaS systems allow users to build applications in high-level languages while restricting the functionality of those apps. The requirement that programs be stateless is one of the most significant limitations—requests are not guaranteed to be sent to any particular instance of a program, thus developers must plan accordingly.Clients often re-issue requests after a timeout, and FaaS platforms provide some amount of fault tolerance by automatically retrying functions if they fail. The use of retries ensures that functions are run at least once. Combining retry-based at-least-once execution with idempotent at- leastone execution appears to provide exact once execution.We argue that in a retry-based faulttolerance model, atomicity is required to address these issues: either all or none of an application's updates should be displayed.Serializable transactions are a straightforward solution in this case. Because functions have well-defined beginnings and finishes, a transactional architecture is an obvious choice for ensuring atomicity in FaaS platforms.Providing high availability for deployed functions is one of the primary difficulties for FaaS providers. Commercial FaaS solutions are marketed as having high availability and built-in fault tolerance.

1.1. Serverless computing

Serverless computing unlike its name do have physical servers present and their services are based on it but the user or developers do not need to worry about them or be aware of it. Serverless computing is fairly a new and booming concept that is growing exponentially in the recent years. When it comes to technology in serverless computing, the user is billed only on the amount of usage.In other words serverless computing is an as-used basis service. It is a provider empowers a user to write/deploy a piece of code or host an API without the pain of establishing, maintain scale an underlying infrastructure. It can be considered as a cloud computing execution model where the service provider provides machine resources and does not occupy resources in volatile memory, however the processing is done with the help of short busts with results continue steadily to memory or storage

1.2. Advantages of serverless computing

Since Serverless computing is only billed based on your usage and until the resource is used, it definitely is an edge in cost effectiveness and is generally very cost effective as compared to cloud providers who bill user for idle resources as well, hence serverless computing is more affordable. Secondly,it would simplify scalability since the user does not need to worry about good practices to scale the system and policies governing scaling. Serverless computing developers can create independent functions that may perform only one purpose for instance an API call hence offers simplified backend code. It is time efficient and has quicker turnaround times as it does not include a complex deploy process to roll out bugs. Users or developers can modify, add code characterized by non-systematic fractional measures

1.3. Faas

Function As a Service is a cloud computing service that gives user the freedom to execute program or code without the hassle of complicated infrastructure with regards to development and launching of applications like microservices to host an application on the world wide web three things are required the physical hardware, virtual machine operating system, and web server software management. Faas provides all of these three things and is handled and provided by cloud service provider so the user can only focus on individual function that are present in your application code.

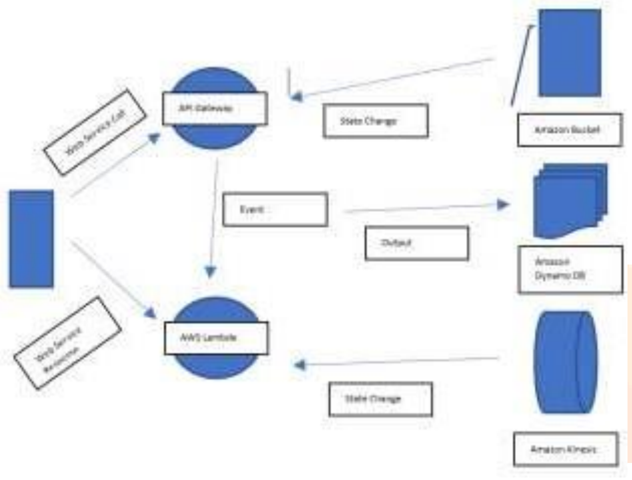

Figure 1. Serverless architecture

Faas and Serverless Computing are often confused with one another but actually Faas is nothing different from serverless computing and is a subset of serverless. serverless in itself is a bigger term that focuses on a variety of service category be it storage , database ,computing , API, gateways etc, where configuration management is hidden or invisible from the user. While in the case of Faas which is one of the most fundamental technology in serverless architecture focusing on events, and is event driven where application code and containers only run in case of requests.

1.4. Advantages of FaaS

The advantages of FaaS includes advantages of serverless computing that it allows user or developers to focus on writing application logic without the burden of deploying on servers hence faster development turnaround, hence Scalable. Developers don't need to create contingencies for traffic, therefore, its cost effective. Since it is a subset of serverless computing, users are only billed on usage and there is no need to spend for extra cloud resources.

2. Retry-based Fault Tolerance Mechanism

Failures occur while communicating with components or services. The issues can be due to unavailability of the service/component, network failure, timeouts or overload. In such cases, if we call the process again, it may succeed. Such type of failures are called transient failures. In such cases, we can implement the retry mechanism to resolve the issue.

1) Identify if the fault is transient
2) Declare maximum retry count
3) Retry the process and increment the retry counter
4) If the process is successful, acknowledge the caller and return result
5) If the fault persists, repeat step 3 until maximum retry counter is reached.
6) If maximum retry count reached, inform the caller that the service is unavailable.

Incomplete action/ process is similar to an undelivered event. When the exception or the error occurs, the retry mechanism analyzes it and identifies whether the process should be retried automatically or not. When the system decides to retry it, it moves the process on queue with some defined delay.

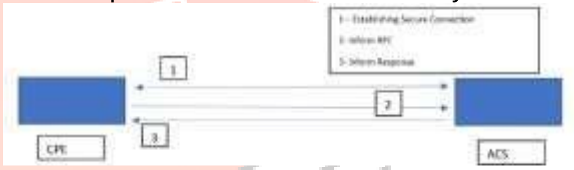

Figure 2. CPE ACS Connection

The CPE tries to send an event to ACS. When the ACS responds and a successful connection is established, the action is complete and the event is considered successful. The event is unsuccessful if the Inform RPC is not completed. After every successful inform RPC, the Retry Counter is set to zero. For an unsuccessful event, it is set to one. In this CPE model, there are two variables of RetryMinimumWaitInterval and RetryIntervalMultiplier. After an unsuccessful event, the CPE waits for some time within the interval of the two parameters which prevents number of CPE reconnection attempts at the same time. If it fails again, the counter is again incremented. This leads to exorbitantly high values of

waiting interval time calculated using the below formula- Previous Maximum
Wait Interval to
$(RetryMinimumWaitInterval * (IntervalMultiplier)^{RetryCount})$
The old maximum range becomes the new minimum.
The new maximum range now equals to the RetryMinimumWaitInterval multiplied by IntervalMultiplier (in seconds) to the power of RetryCount.

2.1. Challenge in Retry-based Fault-Tolerance

FaaS platform provides fault tolerance with retries. If function fails, it retries function until maximum retry counter is reached. It ensures that function executes at-least once. FaaS platform providers enforce developers to write idempotent program as idempotence logically ensures atmost once execution [1]. Combination of retry-based atleast once execution and idempotent at-most once execution is no sufficient to guarantee exactly-once execution. To see why, consider a function *f* write two keys, *k* and *l* to storage. If *f* fails after write of *k*, we have new version of *k* and old version of *l* so parallel read requests see partially updated data. Despite function *f* is idempotent, the application is exposed to fractional execution where some updates are visible and other are not. [1] propose that in retry-based fault tolerance, atomicity is required to solve this problem: Either all the updates made by an application should be visible or none of them should.

3. A Fault-Tolerance Shim

AFT, an Atomic Fault Tolerance shim provides fault tolerance for FaaS application by interposing between a FaaS platform (e.g. AWS Lambda, Azure Functions etc.) and a cloud storage engine (e.g. AWS S3, Google Cloud BigTable etc.). It enforces read atomic isolation guarantee, ensuring that transaction never see partial side effects. All updates to storage are buffered by AFT and committed to storage at the end of request automatically.

1) The design of AFT, a low-overhead, transparent fault tolerance shim for serverless applications that is flexible enough to work with many combinations of commodity compute platforms and storage engines
2) A new set of protocols to guarantee read atomic isolation for shared, replicated storage systems.
3) A garbage collection scheme for our protocols that significantly reduces the storage overheads of read atomic isolation
4) A detailed evaluation of AFT, demonstrating that it imposes low latency penalties and scales smoothly to hundreds of clients and thousands of requests per second.

In sections 3.1–3.2, we present how AFT achieves atomic reads and writes at a single node. Next, we discuss how AFT achieves the same in distributed environment in section 4.

3.1. Architecture and API

Figure 3 shows a high-level overview of the AFT architecture. Each AFT node has a transaction manager, an

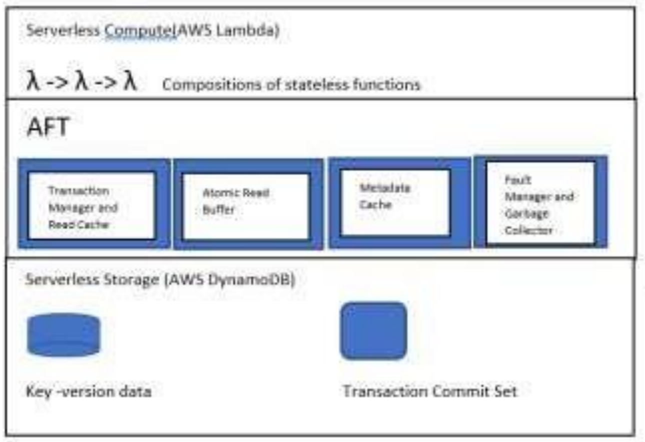
Serverless Compute(AWS Lambda)
λ -> λ -> λ
Compositions of stateless functions
AFT
Transaction Manager and Read Cache
Atomic Read Buffer
Metadata Cache
Fault Manager and Garbage Collector
Serverless Storage (AWS DynamoDB)
Key -version data
Transaction Commit Set

Figure 3. A high level overview of AFT shim

atomic write buffer and a local metadata cache. A transaction manager is responsible for read atomicity. It keeps a track of key versions read by each transaction. The atomic write buffer stores each transaction's write set and is responsible for atomically persisting them at commit time. AFT maintains a Transaction Commit Set storage, which holds the ID of each committed transaction and its corresponding write set. AFT caches the IDs of recently committed transactions and locally maintains an index that maps from each key to the recently created versions of that key. When an AFT node starts (e.g., after recovering from failure), it reads the latest records in the Transaction Commit Set to warm its metadata cache. In addition to a metadata cache, AFT has a data cache, which stores values for a subset of the key versions in the metadata cache and improves performance by avoiding storage lookups for frequently accessed versions.

API	Explanation
StartTransaction()→txid	This API begins a new transaction and returns a transaction ID.
Get(txid, key)→value	This API retrieves key in the context of the transaction keyed by txid.
Put(txid, key, value)	This API performs an update for transaction txid.
AbortTransaction(txid)	This API aborts transaction txid and discards any updates made by it.

CommitTransaction(txid)	This API commits transcation txid and persists its updates; only acknowledges after all data and metadata has been persisted.

TABLE 1. AFT APIS

Table 1 lists transactional key-value store API offered by AFT. Each logical request, which might span multiple FaaS functions, is referred as a transaction. When a client calls StartTransaction, AFT starts a new transation and a globallyunique UUID is assigned to a transaction. Clients uses Get(txid, key) and Put(txid, key, value) to read and write keyvalue respectively. When a client calls CommitTransaction, AFT assigns a commit timestamp to transaction, persists all of the transaction's updates, and only acknowledges the request once the updates are durable. If a client calls AbortTransaction, none of its updates are made visible, and the data is deleted from atomic write buffer.

3.2. Read Atomic Isolation

The read atomic isolation guarantee, introduced by Bailis et al. in [2], ensure that transactions do not view partial effects of other transactions. A system provides Read Atomic isolation (RA) if it prevents fractured reads anomalies and also prevents transactions from reading uncommitted, aborted, or intermediate data. "uncommitted, aborted, or intermediate" data is referred as dirty reads. A fractured read happens when transaction T_i writes two key versions x_m and y_n, and T_j [later] reads version x_m and y_k and $k < n$.

To prevent dirty reads, AFT guarantees that if transaction T_i reads key version k_j written by transaction T_j, T_j must have successfully committed. As described in section 3.1 Atomic Write Buffer sequesters all the updates for each transaction. Below steps shows how AFT implements atomic updates via a simple write-ordering protocol:

1) AFT writes the transaction's updates to storage When CommitTransaction is invoked.
2) AFT updates Transaction Commit Set in storage by
adding transaction's write set, timestamp, and UUID.
3) AFT acknowledges the transaction as committed to the client and make the transaction's data visible to other requests only after the Commit Set is updated
4) If a client calls AbortTransaction, its updates are simply deleted from the Atomic Write Buffer, and no state is persisted in the storage engine.

This simple write-ordering protocol ensures that transactions never read dirty data.

To avoid fractured reads, each transaction's read set must form an *Atomic Readset* defined below: Definition 1 (Atomic Readset). Let R be a set of key versions. R is an Atomic Readset if $\forall k_i \in R, \forall l_i \in k_i.\text{cowritten}, l_j \in R \Rightarrow j \geq i$.
AFT implements atomic read protocol which guarantees that after every consecutive read, the set of key versions read forms an Atomic Readset. It uses local committed transaction metadata and recent versions of keys.

In addition to read atomicity, AFT also ensures two other properties: *r*ead-your-writes and *r*epeatable read.

4. Scaling AFT

One of the key requirements of serverless computing is scalability. In section 3, we have discussed how AFT ensures read atomicity for single node. In this section, we discuss how AFT ensures the same in distributed environment.

Coordination-based techniques are mostly used in distributed environment to scale to hundreds or thousands of parallel requests. But it has issues with performance and scalability. AFT nodes do not coordinate with each other while serving requests. The write protocol described in
3.2 allows each transaction to write to separate storage locations, ensuring that different nodes do not accidentally overwrite each others' updates. Each AFT nodes is allowed to commit transaction without coordination which improves performance. Hence it requires that each nodes are aware of transactions committed by others. A background thread that periodically broadcasts all transactions committed recently to others and listens for messages from other replicas. When it receives a new commit set, it adds all those transactions to its local Commit Set Cache and updates its key version index.

In a distributed setting, we might be processing thousands of transactions a second. So transaction commit metadata can grow monotonically. To avoid communicating unnecessary metadata, there is a process
which proactively prune the set of transactions that each node multicasts. Any transaction that is locally superseded does not need to be broadcast. A transaction T_i is locally superseded if, $\forall k_i \in T_i$.writeset, $\exists k_j | j > i$ - that is, for every key written by T_i, there are committed versions of those keys written by transactions newer than T_i. Each node's background multicast protocol checks whether a recently committed transaction is superseded before sending it to other replicas. If the transaction is superseded, it is omitted entirely from the multicast message. Similarly, for each transaction received via multicast, the receiving node checks to see if it is superseded by transactions stored locally; if it is not merged into the local metadata cache.

4.1. Fault Tolerance

Distributed deployments of AFT have a fault manager (see Figure 3) that lives outside of the request critical path. The fault manager receives every node's committed transaction set without our pruning optimization. It periodically scans the Transaction Commit Set in storage and checks for persisted commit records that it has not received via broadcast. It notifies all AFT nodes of any such transactions, ensuring that data is never lost once it has been committed. The fault manager is itself stateless and faulttolerant: If it fails, it can simply scan the Commit Set again.

4.2. Garbage Collection

There are two kinds of data that would grow monotonically if left unchecked: *transaction commit metadata* and *set of key versions*. Each transaction's updates are written to unique keys in the storage engine and are never overwritten. In 4.2.1, we describe how each node clears its local metadata cache, and in 4.2.2, we describe how we reduce storage overheads by deleting old data globally

4.2.1. Local Metadata Garbage Collection.

There is a background garbage collection (GC) process, periodically sweeps through all committed transactions in the metadata cache. It checks if it is superseded and ensures that no currently-executing transactions have read from that transaction's write set. If both conditions are met, it removes that transaction from the Commit Set Cache and evict any cached data from that transaction. But it cannot make decisions about whether to delete key versions because a transaction running at another node might read the superseded transaction's writes. In next section, we describe a global protocol that communicates with all replicas to garbage collect key versions.

4.2.2. Global Data Garbage Collection.

The fault manager discussed in 4.1 also serves as a global garbage collector (GC). The fault manager already receives commit broadcasts from AFT nodes. This process is combined to reduce communication costs. Each individual replica maintains a list of all locally deleted transaction metadata. If all nodes have deleted a transaction's metadata, we can be assured that no running transactions will attempt to read the deleted items. The global GC process asks all nodes to send list of locally deleted transactions. When the GC process receives acknowledgements from all nodes, it deletes the corresponding transaction's writes and commit metadata. Separate cores are allocated for the data deletion process, which allows us to batch expensive delete operations separate from the GC process.

5. Active-Standby based mechanism

In active standby, the redundancy is introduced in the fission mechanism by creating two instances of the same function service namely the first instance called as the active instance and secondly the standby instance. One instance stays active and the other on standby. This is accomplished by marking first instance as ready and second instance as not- ready. To implement this approach, a popular open source framework, namely Fission, is used due to its ease of deployment and flexibility.

5.1. Fission Architecture

Fission is a well-known open source architecture used in retry based mechanism. Fission is built atop of Kubernetes' basic abstractions like deployments, pods, and services. Fission makes it simple to build HTTP services on Kubernetes using functions. It abstracts away container pictures at the source level. It also shortens the learning curve for Kubernetes by allowing you to create usable services without understanding much about the platform. Deployments are declarative objects that describe an application that has been deployed. Pods are groups of application containers that share a similar execution environment. Services are groups of policies for gaining access to specific pods, including load balancing, naming, and discovery. To utilize Fission, we need to create functions and add them via a command line interface. Functions can be linked to HTTP routes, Kubernetes events, and other triggers. Fission currently supports NodeJS and Python. When their trigger fires, functions are called, and they only use CPU and memory while they're running. Idle functions use no resources other than storage.

Fission is made up of two primary parts: an Executor and a Router. The Executor is in charge of managing the lifespan of function pods. Also, it creates and controls the lifecycle of the function pods.

PoolManager and NewDeploy are the two sorts of Executors. PoolManager keeps a pool of generic warm containers to let functions start faster when they are cold. Autoscaling is not supported by this executor type.

NewDeploy is built on top of Kubernetes deployments, services, and a Horizontal Pod Autoscaler that allows autoscaling function pods.

When a function call is made, the Router sends it to the appropriate function pod and retries if it fails. The NewDeploy executor is used to create two replicas of the service function pod. The state of the function pods is managed by the Kubernetes Readiness Probe. The probe helps to mark the state of the active pod. The active pod is marked in ready state and is therefore ready to receive and serve traffic. The passive pod is marked in not-ready state hence does not receive any traffic. The DNS Server of Kubernetes 'CoreDNS' is used to retrieve the IP address of active instance. In case of failure, then the new pods created and NewDeploy executor ensures that always the two replicas of the functions are running.

The implemented Active-Standby mechanism in Fission works as follows (see Figure 7).The Kubernetes CoreDNS receives function call and returns IP address. Later, the user forwards his request to active pod. At the same time, the active and passive pods send heartbeats continuously to each other to perform health check. The heartbeats are created by the Kubernetes readiness probes which are performed every second. When active pod is up and running perfectly, the passive pod fails readiness probe and stays does not stay in ready state. If the active pod is down, the passive pod gets ready and becomes active. One replica of the pod is created which would serve as passive pod later. If passive pod is down, replica of the active pod is created and keep in not-ready state.

5.2. Experiment

In this section, we evaluate the effectiveness of the active-standby approach and comparing it with retry mechanism used in fission based on experiment performed in [3].

5.3.1. Test Environment.

Grid’5000 testbed, Five nodes to deploy Kubernetes, Fission AS, Fission Vanilla 1.5.0, two CPU’s Intel Xeon E52620 v4, 8 cores/CPU and 64 GB memory, two additional nodes for invoking functions and the other one for inserting faults.

5.3.2. Test Scenario.

Two failure scenarios are defined.

1) Application failure due to pod failure: PowerfulSeal tool is used in this case for inserting faults between 30 sec to 60 sec of workload execution.
2) Application failure due to node failure: A script is used to cause failure in this case at 30 seconds after the beginning of workload execution.

5.3.3. Applications.

Two HTTP-Triggered functions were used. The first is Fibonacci, which is a CPU-intensive function for computing the Fibonacci sequence. The second application is the Guestbook, which consists of two functions, GET and ADD, for reading and writing text messages in a Redis database.

5.3.4. Evaluation.

Evaluation is based on Performance, Availability and Resource Consumption.

5.3. Experimental Results

[3] performed three different sets of experiments: (1) Experiments without failures; (2) Experiments with pod failures; (3) Experiments with node failures.

5.4.1. Experiments without failures.

Figure 8 and Figure 9 present the throughput and average response time of the Fibonacci and Guestbook applications deployed with both Fission AS and Fission vanilla without failures. Throughput is similar for both functions in their versions. Both show a capacity of 11 seconds processing per request. Both functions, Fibonacci and Guestbook, have lower response time with Fission AS.

5.4.2. Experiments with pod failures.

Figure 10 and Figure 11 show the throughput and average response times of the Fibonacci and Guestbook applications with Fission AS and vanilla, with pod failures. Vanilla retries the function execution numerous times until the maximum number of attempts is reached, after which it deletes the function instance from the cache and recreates a new one. This wastes time and resources because it effectively re-executes a request that will most likely fail in the end. Active-Standby method allows for speedier recovery than the vanilla retry system.

5.4.3. Experiments with node failure.

Figure 12 and Figure 13 show the throughput and average response times of Fibonacci and Guestbook applications with Fission AS and vanilla, with node failures. In terms of availability, we can easily observe that AS outperforms vanilla. Another observation is that vanilla tolerates shortterm failures better than long-term failures like node breakdowns.

6. Conclusion

In this paper, we have presented how Atomic Fault Tolerance shim (AFT) achieves fault tolerance by guaranteeing read atomic isolation. It adds minimum overhead to prevailing architectures and gauges linearly with the size of the cluster in just one execution. We have also discussed how the high availability is achieved using ActiveStandby approach where system makes sure that there are two replicas of same function services. In case one become nonoperational, the other one is there to back it up by handling the service requests which provide fault tolerance. If we want to achieve atomicity in serverless computing, we can use AFT with function retries mechanism for fault tolerance. But if the requirement is high availability, then Active-Standby approach is better for fault tolerance.

References

[1] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, Jose M. Faleiro, “A fault-tolerance shim for serverless computing,” in *the Fifteenth European Conference on Computer Systems (EuroSys ’20)*, Apr. 2020, pp. 1–7.

[2] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, Ion Stoica, “Scalable atomic visibility with ramp transactions,” in *2014 ACM SIGMOD International Conference on Management of Data, SIGMOD ’14*, New York, NY, USA, Jan. 2014, p. 27–38.

[3] Yasmina Bouizem, Djawida Dib, Nikos Parlavantzas, Christine Morin, “Active-standby for highavailability in faas,” in *WoSC6 2020 Sixth International Workshop on Serverless Computing*, Delft, Netherlands, Dec. 2020, pp. 1–6. [4] Serverless Functions for Kubernetes - Fission. [Online]. Available: https://fission.io/

[5] Bruce Wu. (2019, Jun.) Fission: A Deep Dive Into Serverless Kubernetes Frameworks. Alibaba Cloud. [Online]. Available: https://www.alibabacloud.com/blog/594902

[6] Soam Vasani. (2017, Jan.) Fission: Serverless Functions as a Service for Kubernetes. Kubernetes Blog. [Online]. Available: https://kubernetes.io/blog/2017/01/fission-serverlessfunctionsas- service-for-kubernetes/

[7] What is serverless computing? — serverless definition. Cloudflare. [Online]. Available: https://www.cloudflare.com/learning/serverless/what-is-serverless/

[8] What is Serverless Computing? IBM Cloud Education. [Online]. Available: https://www.ibm.com/cloud/learn/serverless

[9] Function as a service. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Function as a service [10] Serverless computing. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Serverless computing