Project Report- Final Project

CSCE 689-607

Cloud Computing

Project Name: Function As a Service (Faas)

Github Link: https://github.tamu.edu/adil-hamid/689-18-a-P2/tree/master

Members:

Adil Hamid Malla (425008306)

Shubham Jain (525004338)

Abhishek Sharma (226001364)

Timeline of the project (Approximation for 3 person efforts) :

1)	Problem Literature and Research	:	10 hours
2)	Design Discussion	:	6 hours
3)	Core Service Implementation	:	36 hours
4)	Database Designing & Implementation	:	6 hours
5)	UI Mockups	:	3 hours
6)	UI Implementation	:	12 hours
7)	Integration with UI	:	12 hours
8)	Additional Feature Addition	:	15 hours
9)	System run on Local system	:	3 hours
10)	Integration with Docker	:	13 hours
11)	Testing with different platforms	:	6 hours
	Total Time Spent	:	122 hours

About the Project:

In this project, we have implemented a cloud-based service that lets you run code without provisioning or managing servers. Function as a Service(FaaS) provides a stateless server which doesn't need to be created, configured and maintained in the cloud. When the data arrives, FaaS triggers the function and executes. There can be different kinds of triggers to which the service could take events from but we used Kafka Queue for the project. A User Interface is also made to upload the lambda functions, add triggers and run them when needed. One of the most important use case of FaaS is data processing, eg. real time file/stream processing, or in extract, transform, load.

Motivation:

Having seen the trend to go from the bare-metal to virtual machines was the first step to the cloud and distributed computing. Then, came the concept of making these resources available on the cloud. Thus we came to know about the Platform as a Service and Hardware as a Service. Making life easy for the users, the new concepts kept on adding on this structural base. One of the most popular was the Software as a Service, which made the distribution, maintenance, and security of an application as easy as one button click. But having seen the application as service, the stakeholders are assuming that the whole functionality of the software application will be used which is not the case majority of the times. Majority of the end users need the application to run a single functionality of the cloud-based application.

Even though the user is not using all the application, the resources are assigned based on the application level requirement hence making the resource management costly and tedious job.

Making Function as a Service (Faas) helps in tackling all the problems and making the resources more manageable and not having to tackle the problem of managing the whole application as well as the hardware resources.

Background Research:

There are many Function as a Service implementations present in the market today. The popularity of serverless architectures started with the AWS Lambda but there are many other big players in market and they are all evolving very fast. Some of these FaaS are provided by Microsoft, Google, etc. Microsoft has Azure functions, Google has Google Cloud Functions, IBM has Openwhisk and so on. In these, only IBM has open source code of Openwhisk.

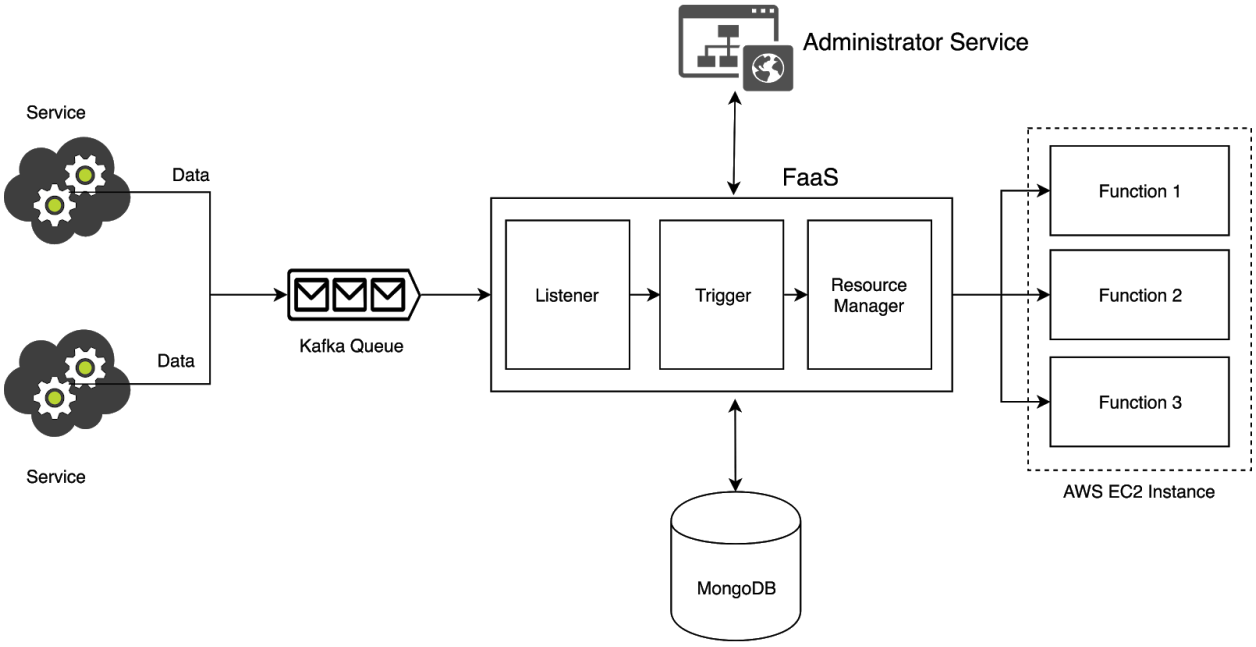
We first try some hands on AWS Lambda which is most popular of these all and we have architected our system taking inspirations from AWS Lambda. For starting with, we also research with IBM Openwhisk since it is open-source. We also looked into some more

implementations present online to get the brief idea of the implementation. We figure out several things after our research, like the components of our system, how each components is connected to each other, etc.

The research of looking into some of the existing solutions help us made the system that is present today. Our solution differs from all the previous existing solution in one dimension that our system is made very easy to deploy and hence it can be used by anyone. We are making it open source. The customers we are targeting for our Service are enterprises which can use this code and use their machine in the backend to use the Function as a Service.

Proposed Architecture :

The architecture of the Function as a Service(FaaS) system looks like this:



1. **Queue Waiting System:** For our system, we assumed that all the events are stored in queueing system and our system would listen to this queue. We used a queueing system to make sure all the individual requests are processed atomically since all the requests have to be handled individually. We used Kafka queue to implement the same.
2. **Listener:** The listener would listen to all requests from all the Kafka topics one at a time and send them to the trigger, where they would be mapped to the functions. So, its job is to subscribe to all new topics added in Kafka and listens to all the events from them.
3. **Trigger:** Trigger had a mechanism to map from the requests from the listener to the corresponding functions. Trigger handled this by querying the mongo database given the

request and its associated data(like kafka topic name). Once the Trigger get the function name from the database, it will call the resource manager to provide the infrastructure to handle the execution of the function on the server.

4. **Database:** All the requests are handled by the trigger mechanism to invoke the right function depending on the requests as well as data passed. We need to have a mapping system available and it needs to be fast so that we can have small latency. Since the mapping is not just single key based, we will need to have the multiple key based data structure for handling the matching. Also, we need a place to store all the current Kafka topics so that listeners would know which all kafka topics it should listen to. For serving all these purposes, we used the document based database, mongodb for storing this data.
5. **Resource Manager:** Given the request mapped function and its corresponding resource it needs, the resource manager provides the infrastructure to the request with all prerequisites from the resource pool. After the execution is done, the resource manager gets the resource back to the pool and the result of the executed function is returned to the user. For our use case, we used a AWS EC2 instance for running all these functions.
6. **User Interaction:** The user interface will allow users to request execution of the functions hosted by the system. This also gives the interface for the service providers to add functions to be executed by the users. This interface will also facilitate the user to put data to be used by the functions onto the system and get results from the system. Also, it provides a user a way to edit function and to see the output of the functions(i.e. history of all commands ran).

Functionalities Planned and Completed:

Following are some functionalities that we planned to add in the FaaS and are able to complete them all:-

1. A listener which would listen to all events from different queues. Eg. listening the data from Kafka queue, etc.
2. A Trigger which would take actions based on the event received. It would execute the appropriate function based on the event type received.
3. Resource Manager which would make sure to utilize the resource optimally. It would do so by analyzing the request traffic and accordingly allocate the machines for Lambda function.
4. An UI interface for the user which would provide the upload feature to make it easy to use.

Apart from the planned functionalities, we are also able to add some more features in our system which are as follows:

1. An UI interface for the source service to send some events to Kafka queue.

2. Added a functionality to create the kafka queue from the UI.
3. Add lot of error handling at each point of the system. Eg. Checking the syntax errors in the python code uploaded, making sure there are no duplicates of functions or kafka queue, created.

We will provide more details on each of the above functionalities that we have added in our system in the subsequent topics.

Tasks Planned and Done:

Planned - Setup the virtual machine for Lambda service, which comprises of a listener to listen to the user based requests, a trigger and provisioning mechanism.
Done - Since the number of services involved in our system are quite many, so we decided to run most of the things in local.

Planned - Setup Kafka and create a Kafka Topic.
Done - We setup a Kafka and provided a UI to create Topic. Eg. How AWS Lambda does.

Planned - Implement a listener to listen events from Kafka Topic.
Done - We are able to create a listener which keep running and checks for any messages and it also subscribes to new topics if it notices there is a new topic created in Kafka.

Planned - Implement trigger mechanism which would take the decision of executing the particular function based on the event type.
Done - We are are able to create a trigger which basically connects with mongodb and finds the corresponding function to the message.

Planned - Implement Resource Manager which would allocate the machines and configure the allocated machines.
Done - We have implemented a resource manager which runs the function uploaded on AWS EC2 instance.

Planned - Create a simple UI/CLI interface which would have functionality for uploading the function and executing it.
Done - We have created a UI interface to upload the functions and map it to the Kafka Topic. This makes it very easy for the users to interact with our system without taking care of anything happening inside the system.

Implementation Details:

Implementation of this system includes development of lot of components since the system is made up of many components as can be seen in architecture above. We have implemented our code in different services. Management of the different services helps us to achieve the realization of the project. Majority of the backend implementation is done using the Python 2.7. Moreover, the web application shown to the customers using the Function as a Service (FaaS) is implemented using the flask framework of the python. We have used HTML5 with JS to handle the front end of both the user service as well as the admin service for adding the functions and other functionalities.

The main services of our project are:

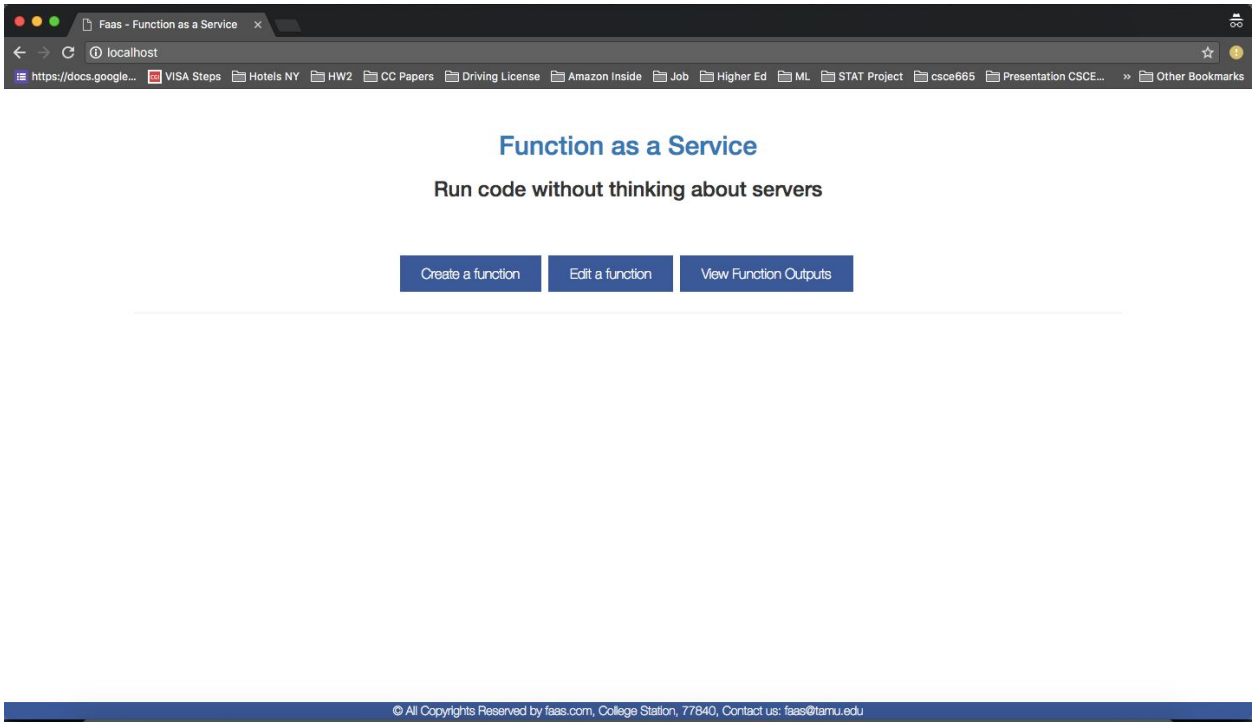
1) User Service
2) Listener Service
3) Queuing Service - Kafka Queues
4) Database Service
5) Trigger and Resource Manager Service
6) Administrator Service.

Administrator Service:

The service is required to add the kafka topic, manage the functions that is served by our system and also the output the result that are obtained when a user sends the request for running the function with the input data. The web application service is implemented using the python flask framework. This is a web-service that can be invoked using the http request calls. We have implemented the front end using the HTML5 with JS to handle the interaction with our web-service. The web service in background interacts with the database service and kafka client to manage the addition of the topics, and functions. Moreover, it also handles the task of showing the output of all the functions run in past with their data as well as output after running the function on the user data.

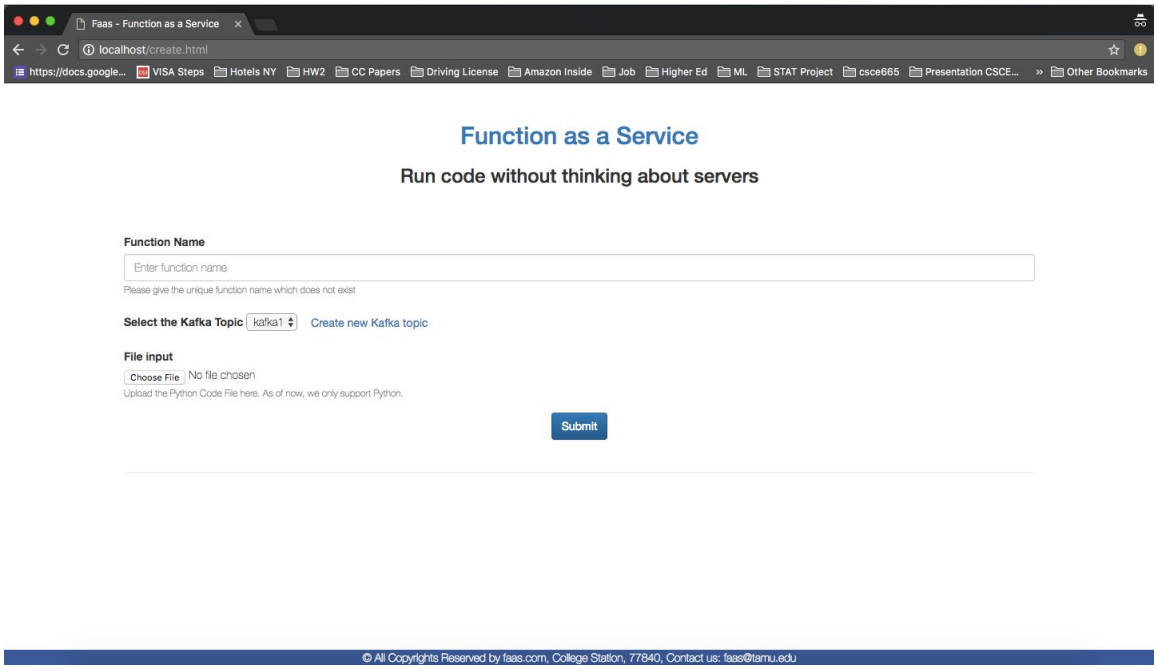
Please note that the name of the service is little misleading. By administrator, we do not mean that this service should only be visible to only one person or something like that. It's just that we are calling the person who can create the functions or kafka topics as administrator here but that can be anyone. Normally, administrator here would be a developer who wants to use this system to upload some functions which runs when the event arrives. Currently, we do not have the account concept of user in the website, so do not maintain any user related data like who is creating the function and all. Hence it is open to everyone.

The main page of our administrator website looks like below:

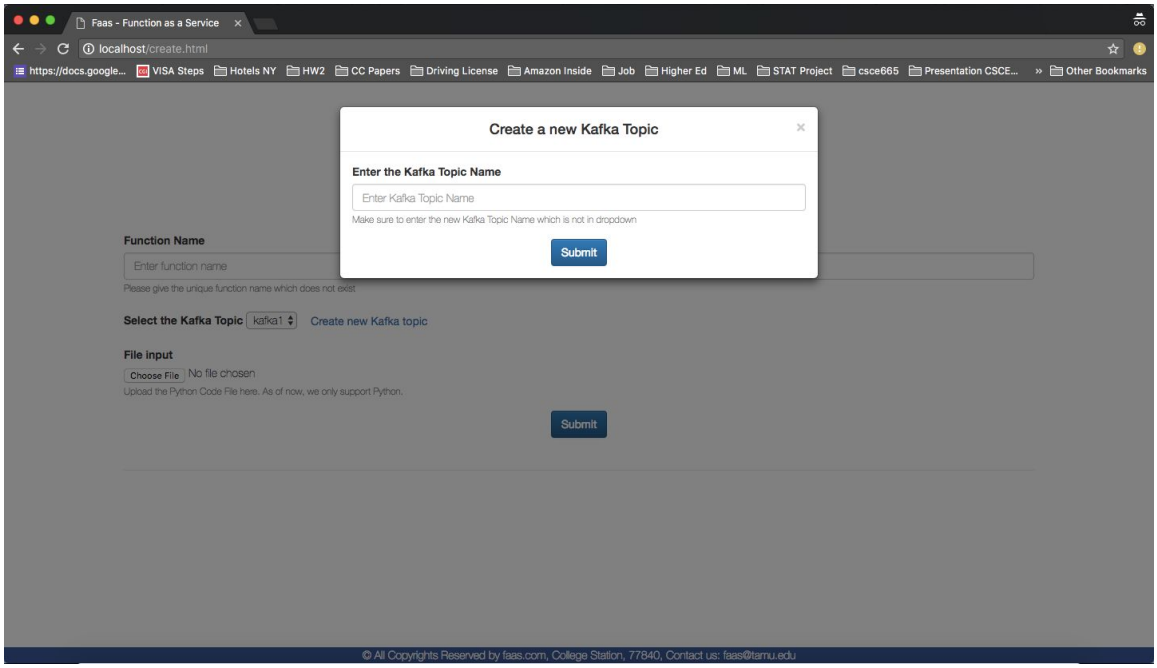


As you can see in above figure, we have three main components in this service:

1) **Function-Service Creation:**
   Since this service is admin only, if a service provider wants to add the functionality in our system, he has to provide a python file that contains the function and the parameters that the function expects like kafka topic name and the function name. This component also contains the functionality to create the kafka topic if it did not exist. Following is the screenshot of the webpage to upload new functions in the system.

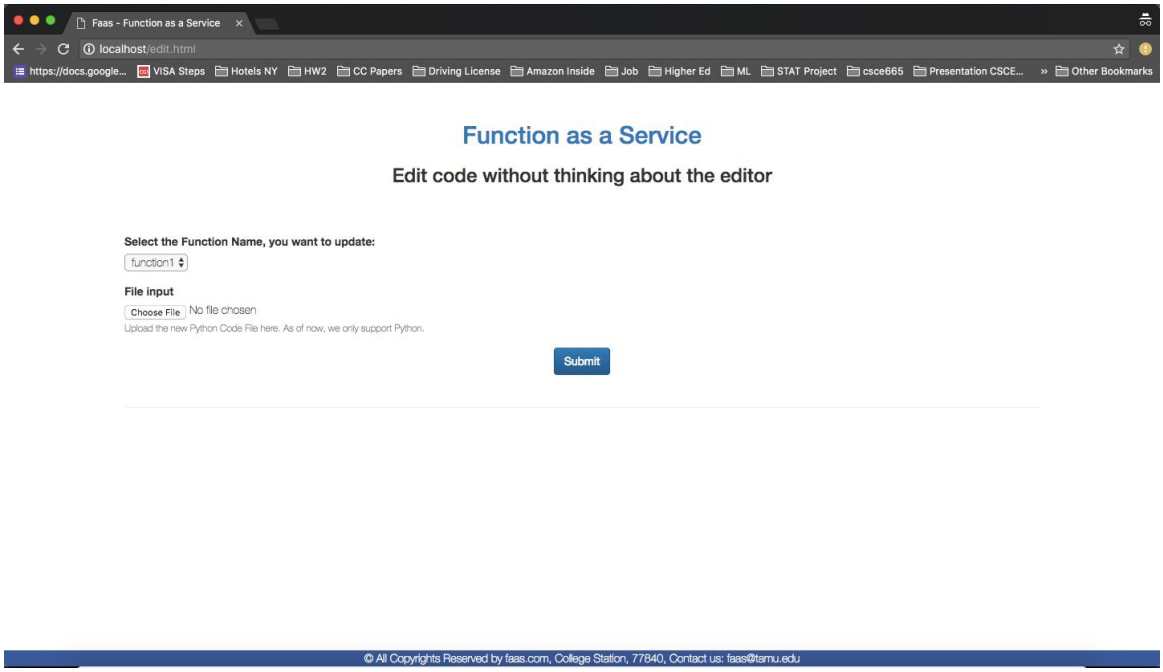


For now, we only supports Python 2.7 files to be uploaded. We have done necessary error handling in the frontend website and backend so that no other files can be uploaded. Eg. apart from normal file extension check, we used pyflakes library to check the syntax errors in the python file. Once the user submits the function creation request, we store the python file in our server and creates a mapping of Kafka and function name in mongodb. And if the user creates the request to create a new Kafka Topic, we insert a new kafka topic name in another mongodb collection so that listener is informed a new kafka topic has been created. Following is the webpage to create the new Kafka topic.



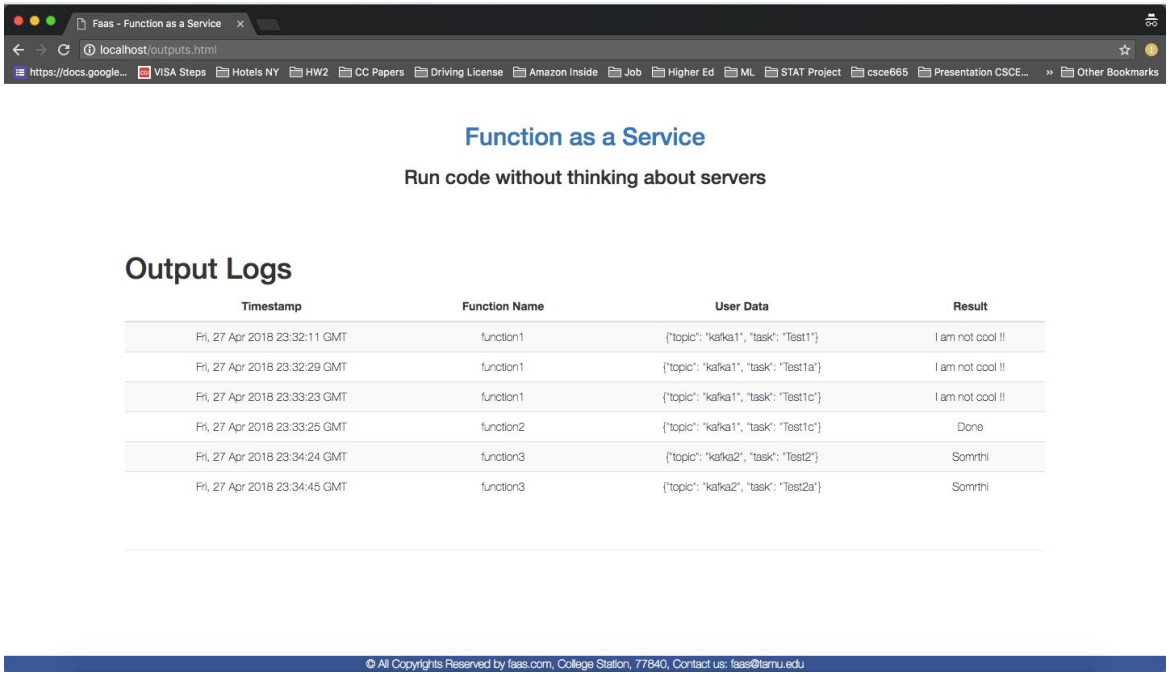
2) Function-Service Updation:

For some reason if the functional service provider feels like to upgrade the functionality algorithmically or wants to update the function with some optimization, he can use this sub service to do so without changing anything else. The service provider just needs to give the updated python file with new parameters and we are good to include the updated functionality in our system. Following is the webpage in our website to edit the code of the functions already uploaded.



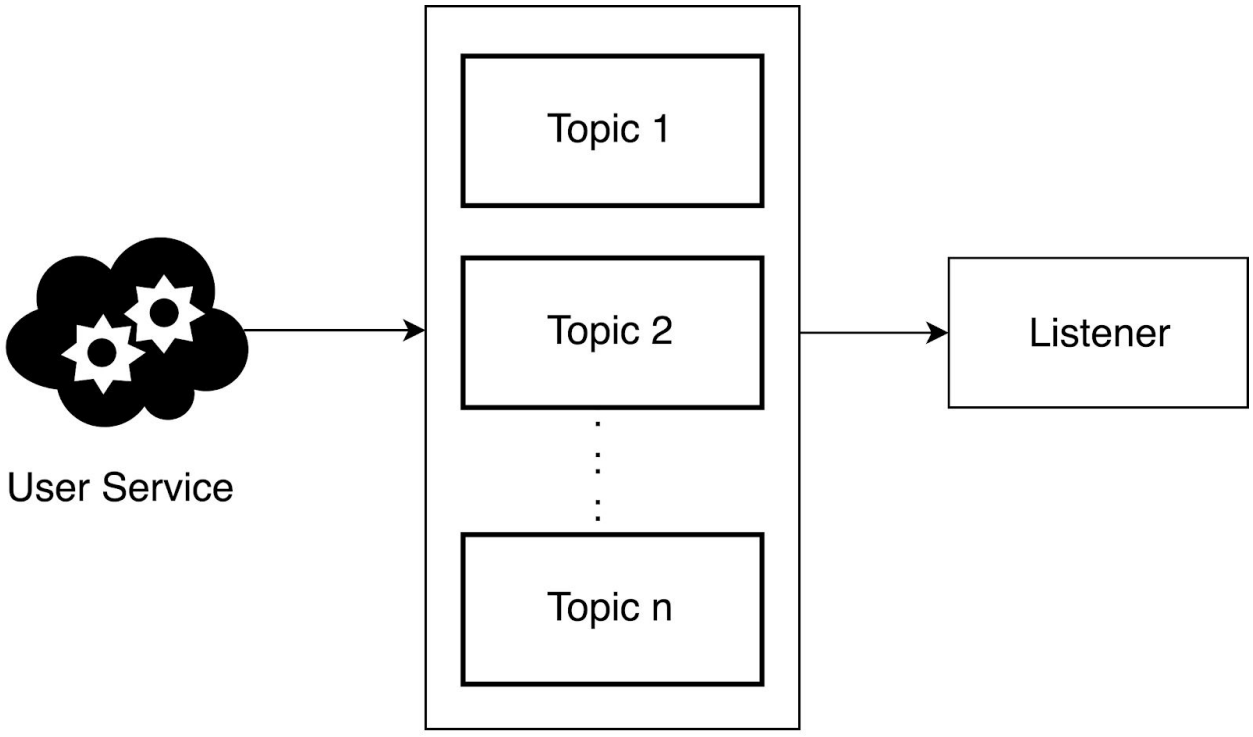
3) **Output Display for Admin:**

Each function is made available to the customers to use using our other web application that only faces the actual users. This web application takes the input data and function name from the user. Once the request is generated it is handled using the Listener, Trigger and Resource Manager. Once the function gets a resource to run, the results are stored in the mongodb to be showed both to the admin for debugging and maintenance purposes as well as to the user who requested for running the service.



Queuing Service-Kafka:

As mentioned earlier in the architecture, we wanted to implement the queue service to make sure the functions are executed on basis of first come first serve. We implemented the Kafka Queue for serving the same. All the events initiated by the user for running the function is pushed to the Kafka Queue. Each event consists of the parameters of the functions that are mapped to this kafka queue. The kafka topics available are the ones created by the admins using our administrator services. Moreover, the input parameters are also defined by the admins. We push the input params/events into queue to be listened by the Listener service to take it forward for execution. The below figure shows Kafka Queue in the blue box. We can see that, it receives events from User Service(the services that pushes the data into queue) and listener fetches the events from queue. Also, it consists of multiple topics to which user service can send the data to.



Kafka-topics are the broad category of the functions. For example if we have three functions, like add, subtract, and multiply we can assemble them in one category called Maths Functions, thus the name of the kafka-topic is majorly decided on functionality. Although, this being said, we can also prioritize the running of the functions based on priority and the premium account system based on the kafka-topics. If we have a premium kafka-topic that is run on high power server in background, then service provider can get a kafka-topic for fast track system and thus the function will be run faster as compared to other functions in other kafka-topics. Another thing to note is that a single kafka topic can be subscribed by many functions and a function can subscribe to many kafka topics. So, it's a many to many mapping.

Listener Service:

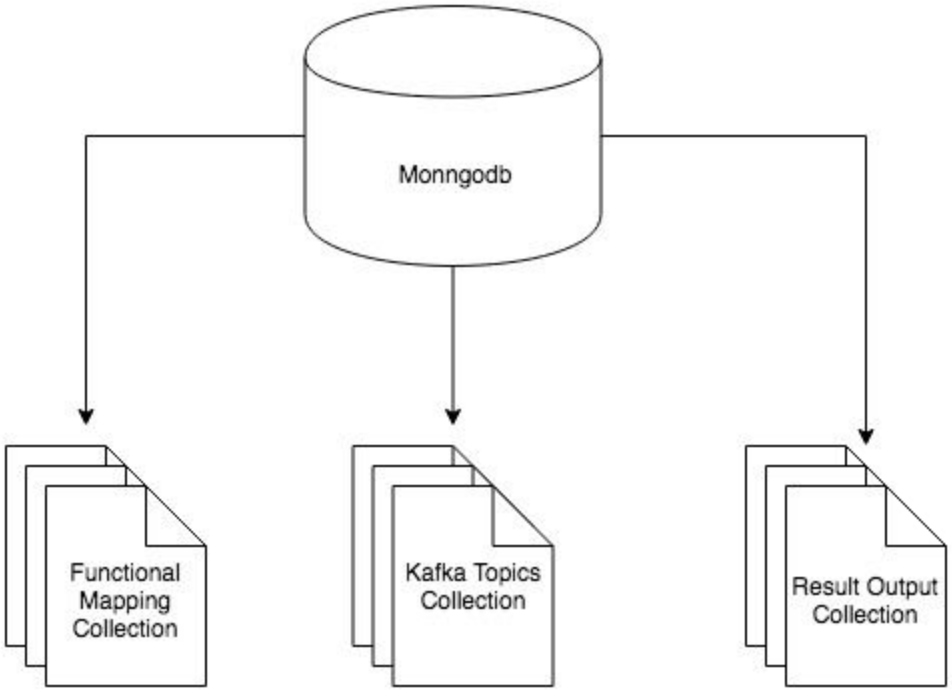
Listener service is very important service that listens to the request fed to the kafka-queue. Listener is running indefinitely and keeps on reading mongodb so that it can listen to the new kafka-topics added dynamically. Listener Service is implemented in the python that runs continuously and keeps on waiting for the new requests. This service is very important to track the requests coming from the users using the functionality.

Once the listener gets the new message generated by the requests, it accepts it and passes it to the trigger. The function of the listener is to listen to the kafka topics and report it to trigger for future processing. It uses Kafka Consumer client to subscribe to kafka topics and read the requests present in the kafka topic.

Database Service:

For storing the mapping between the function name and the actual function that needs to be invoked is stored in the database. We are using mongodb as our database system. Mongodb is a collection based noSQL database system. A database client is made which is exposed to all the services to interact with the database. We are organizing our database in three collections.

1) Function Mapping Collection.
2) Kafka Topic Collection.
3) Result Output Collection.



Function Mapping Collection: It stores the mapping of where we are saving the function corresponding to the function name. We are also storing the kafka topic it belongs to. Once we get the request from the user for running a function with input params, the listener catches the request and the trigger invokes the database call to know which file needs to be executed and then transfers this information to the resource manager.

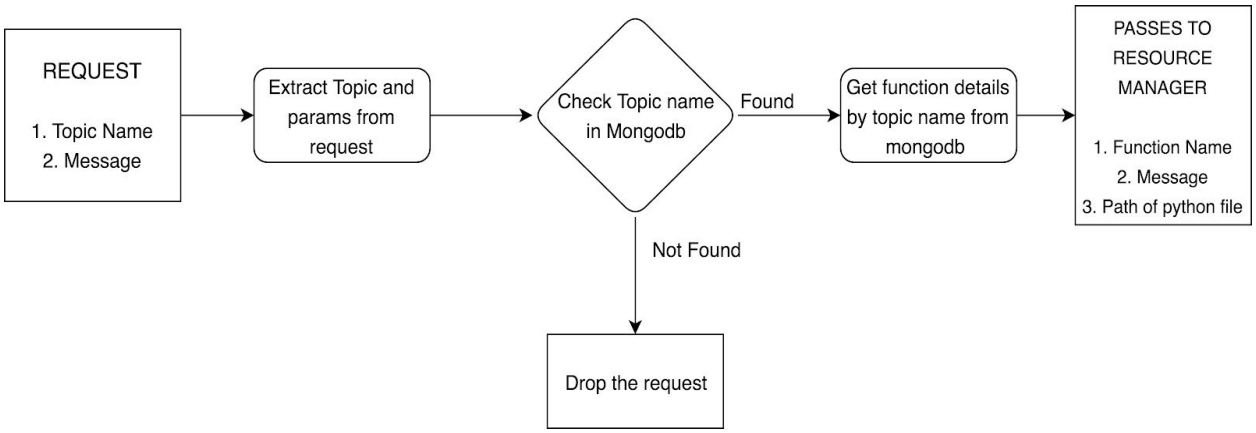
Kafka Topic Collection: This collection of the mongodb stores the kafka topics that our system supports. This helps us to keep the track of the topics the listener is listening and listener polls to this database to check if new kafka topic has been created at regular intervals of the time.

Result Output Collection: This collection of the mongodb stores the output of all the function served and run for each request by the user. This makes sure that we have the proof as well as the

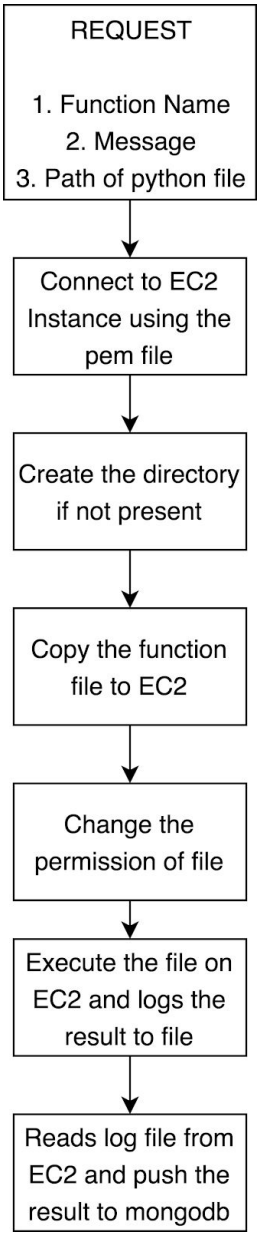
output result for function requests. This will help us to keep the outputs ready for the user requests.
We connect to the mongodb using the pymongo package of the python.

Trigger and Resource Manager Service:

Trigger and the resource manager is one of the most important part of our project. Trigger handles all the request captured by the listener. Once the trigger get called, it invokes database call to mongodb on function mapping collection to get the function's actual file that needs to be run in the input data from the user. Once we get the location of the function, we send the request to the resource manager to get the hardware instance for running the concerned file. The resource manager helps to get the resource from a pool and then transfers the function to the concerned instance and runs the function. The entire flow of the trigger is shown in the following flowchart:



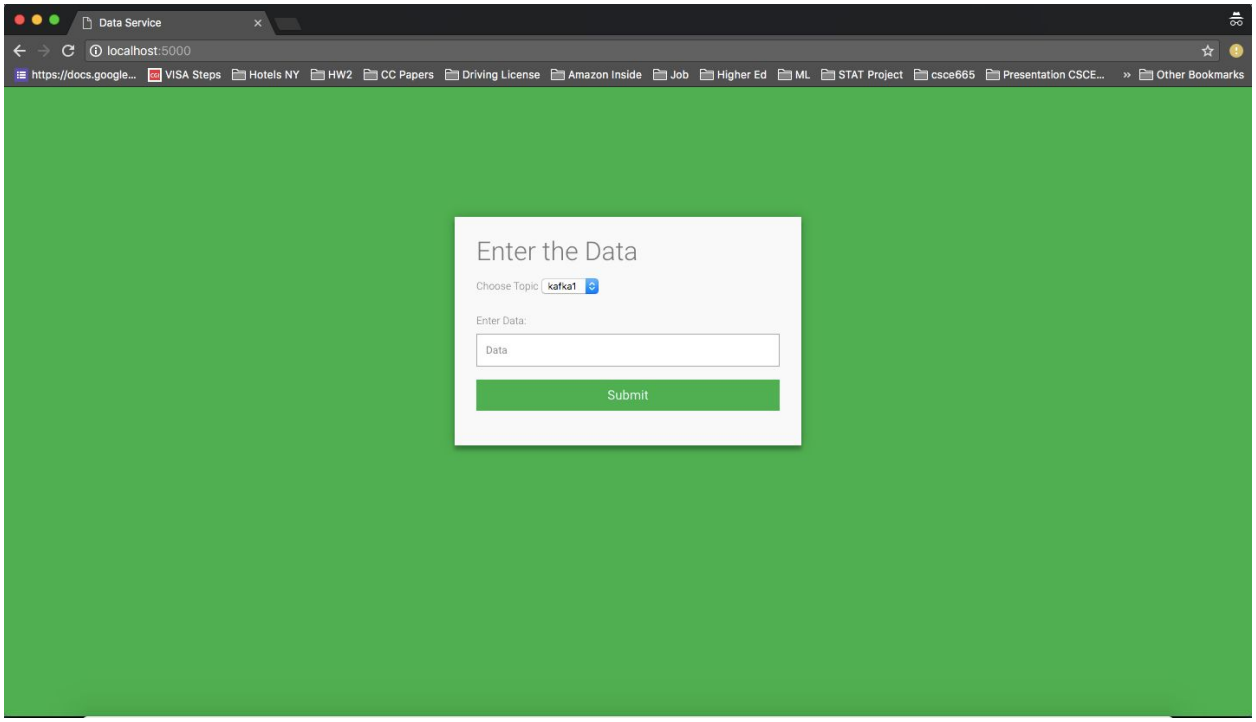
Once the resource manager provides the resource for running the function, the output of the function is stored in the mongodb for maintenance as well as for showing to the user. After this the function is executed properly the resource is freed and the process continues.
Currently the infrastructure we are using is hosted on Amazon EC2, due to lack of free credits and high charging, we tested using just one EC2 instance to run the function through the resource manager. The entire flow inside the resource manager is shown in the diagram below:



User Services:

This is the front end service that will be exposed to the users who wants to run the function served through our system by some service provider. The user will access the Web UI that will help them to choose among the functions available and then also ask them to add the data needed to run the function. Once the topic is selected and the data is added, the request is send to the kafka queue topic, where all the processing that we have discussed above will happen. Once the function is executed and result is saved, we will technically notify the user about the completion, that part we haven't done as it is trivial and doesn't add any technical complexity of our system.

This part can be easily done by adding the user information and keep track of the requests generated and then populate the results correspondingly.



Example of Transformation Logic with our system

Say, there are multiple services that are collecting data from many sources and we want to aggregate this whole data in one place(eg. Hbase) for data analytics purpose. Also, these services are collecting this data in different formats, so before aggregating this data, we need to transform this data so as to bring all data in same format. This is where our system would be very useful.

Steps to achieve this system using our system:

1. A developer can write a function which would contain the code to transform the data to a single format and pushes this data in Hbase.
2. Once, developer writes the function, he can create the kafka topic(to which all the services would push their data) and the function in our system using our UI shown above.
3. Developer job is done now.
4. Now, different services which have this data in different formats can starting sending their data to Kafka queue. And that's all.

So, here, we can see how easy it is now to build things. One can do many such things. Above is just one application of our system.

Technologies Used: We have used a plethora of technologies to implement our system. Our major coding effort was done in Python 2.7 for its widespread use and library support. Following are the technologies that we have used in our system.

1) Kafka Queues
2) Zookeeper for maintaining the Kafka
3) Apache HTTP Server
4) Flask Framework
5) Mongodb
6) Docker
7) Bootstrap
8) Programming Languages: Python, HTML, CSS, JS, Bash
9) Python Libraries Used:
    a) Subprocess
    b) Kafka
    c) Datetime
    d) Pymongo
    e) Flask python
    f) Flask cors
    g) Pyflakes
    h) Json

Steps to run our system:

Since we have a lot of components in form of services that needs to be run for setting up things properly for our system. We have set up our system in two modes.

1) **Docker Mode:**

    Docker Mode is the easiest one, where you need not to worry about setting up the environment for each service. Docker is already used widely towards the deployment of applications. It only needs two things to run the whole system, Docker application and the github code where all the code is implemented.

    Please take the code from this branch to run the code in Docker mode :

    https://github.tamu.edu/adil-hamid/689-18-a-P2/tree/master

    Steps to run using the docker method:

    a) First make sure you have the docker application.
    b) Connect to the internet so that the docker can download the images required to run the system.
    c) Pull the code from the github.

d) After pulling the code, make sure to change the permission of faas.pem file to 400. It is present in function-as-service/listener/faas.pem. This file is needed to connect to AWS EC2 instance.
e) Run a single command and sitback and enjoy the docker magic. Run following command according to your operating system inside the folder where you can see the docker-compose.yml file.
    i) **Mac**: "docker-compose up --build"
    ii) **Linux** : Install docker
        sudo apt install docker.io
        sudo apt install docker-compose
        sudo docker-compose -f <docker.yml file> up --build
f) Now, all the services should be up and running.
g) Go to http://localhost for website where you can add/edit/see results of the functions.
h) Go to http://localhost:5000 for website where user is going to push data to Kafka.

2) **Local System Mode:**
If you have plenty of time to play around with a lot of services and want to see how many services docker is creating, please follow the following steps after pulling the code from this branch : https://github.tamu.edu/adil-hamid/689-18-a-P2/tree/localhost-prod

Steps to run using local system mode:
a) Download and install the mongodb
b) Download and install kafka and zookeeper
c) Start Zookeeper and Kafka:
    brew services start zookeeper
    brew services start kafka
d) Start MongoDB
    sudo mongod
e) Start the backend application in function-as-a-service/webservice/
    python application.py
f) Open the webpage by right-clicking on webapp/index.html
g) Start the listener inside function-as-a-service/listener/
    python listener.py
h) Start the user service inside the source-service/
    python userService.py
i) Now, all the services should be up and running
j) Go to http://localhost for website where you can add/edit/see results of the functions.

k) Go to http://localhost:5000 for website where user is going to push data to Kafka.

Conclusion:

In conclusion, we present Function as a service for a hassle free service to run functions. Currently we only support the python file for execution, but the modularity of our implementation makes it possible to add functions in other programming languages as well in future. The Function as a service helps to make the functionality as services available to the users who don't want to invest in infrastructure, platform or application. This functionality is similar to that of Lambda functionality of AWS.

We further made the project in such a way that it would be possible to make various amendments in the project according to the user. A user can run various functionalities without considering the infrastructure and implementation, e.g. machine learning functionality. The other users of our system are the function providers, who specialize in implementing the algorithm or functionality. So both the users of our system specialize in their kind of work and we provide them a safe environment to make the transaction between them possible.

What we learnt from this project are enormous technologies and their integration and application. As mentioned in the Technologies used section, we learnt a plethora of technologies. Broadly we learnt following things:

1) Kafka Queue
2) Zookeeper
3) Resource Management
4) Docker
5) Using Virtual Instances
6) Mongodb
7) Interaction of user services and handling the infrastructure.

We faced a lot of road blocks. The main ones were setting up the technological services like Kafka, Zookeeper, and many others. We spent a lot of time reading about their implementation and how people have used them to solve their problems. To avoid these kinds of problem during the deployment we spent a lot of time on Docker, so that the user didn't have to go through all the pain of setting up the environment.

During setting up the Docker also we faced many problems of communication between the docker containers and how to handle the filesystem of the container and share the data. First we thought of doing the message passing between the containers but then if the data is huge then we would face a problem of network bandwidth getting used, so we choose the shared system architecture for two containers sharing the underlying filesystem or data volumes.

To make sure we make our resource manager highly dynamic we thought of using the openstack

with nodes being added from our machines and one of our teammates lab machines so that we can do the resource management. Due to restrictions on the networking, permission and other reasons we were not able to implement that and we resorted to using the Amazon EC2 system for testing purposes. Our implementation is ready to handle hardware scaling by adding a simple load balancer at resource manager.

Demo Links:

Docker Setup : https://www.youtube.com/watch?v=v1QisIVjj3g
Kafka Topic Creation: https://www.youtube.com/watch?v=XDuwZnWmmtI
Function Creation: https://www.youtube.com/watch?v=7B8BmZ5_73M
Function Update: https://www.youtube.com/watch?v=q5cj1GQqPgc

References:

1. https://aws.amazon.com/lambda/
2. https://www.cloudkarafka.com/blog/2016-12-13-part2-3-apache-kafka-for-beginners_example-and-sample-code-python.html
3. https://github.com/apache/incubator-openwhisk
4. https://cloud.google.com/functions/
5. http://kafka-python.readthedocs.io/en/master/
6. White Paper on “Serverless Architectures with AWS Lambda”
7. https://github.com/PyCQA/pyflakes
8. https://getbootstrap.com/
9. https://aws.amazon.com/ec2/
10. https://api.mongodb.com/python/current/
11. https://www.perlmonks.org/?node_id=916491
12. https://en.wikipedia.org/wiki/Function_as_a_service
13. https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/
14. https://www.usenix.org/legacy/event/atc10/tech/full_papers/Hunt.pdf
15. Kafka Paper - http://notes.stephenholiday.com/Kafka.pdf
16. Docker Codes: https://github.com/docker
17. http://flask.pocoo.org/docs/0.12/htmlfaq/
18. https://www.sciencedirect.com/science/article/pii/S004579061500275X
19. https://www.stratoscale.com/blog/cloud/7-alternatives-aws-lambda/
20. https://www.openstack.org/assets/pdf-downloads/virtualization-Integration-whitepaper-2015.pdf

and large number of Google Searches and Stack Overflow references.