Function as a Service on Openstack

Vision and Goals of The Project:

Function as a service is a utility that allows users to run their code on the cloud without worrying about creating, configuring, and maintaining machines on the cloud. The resources are provisioned and the code is executed on an event and the resources are released as soon as execution ends.

It will allow users focus on development rather than managing the environment.

The vision is to develop this capability for MOC using Openstack which can later be packaged so that it can be deployed onto other OpenStack clouds and eventually be published in an App Catalog like Murano.

High-Level Goals include:

- Providing an interface for users to upload their code for a specified computation tasks that run in response to events/messages/streams.
- Real-time resource provisioning based on some predefined, user parameters for every event.
- Execution of the uploaded code on provisioned resources and returning the resources to free pool on execution.

Users of The Project

This provides the user to use the cloud as a platform or a service. It targets end users as third-party or in-built cloud applications.

1. **Platform:** The end users who can host their code as a service to be called on an event.
2. **Service:** The cloud application that will use this service to execute the code on an event.

Scope and Features Of The Project:

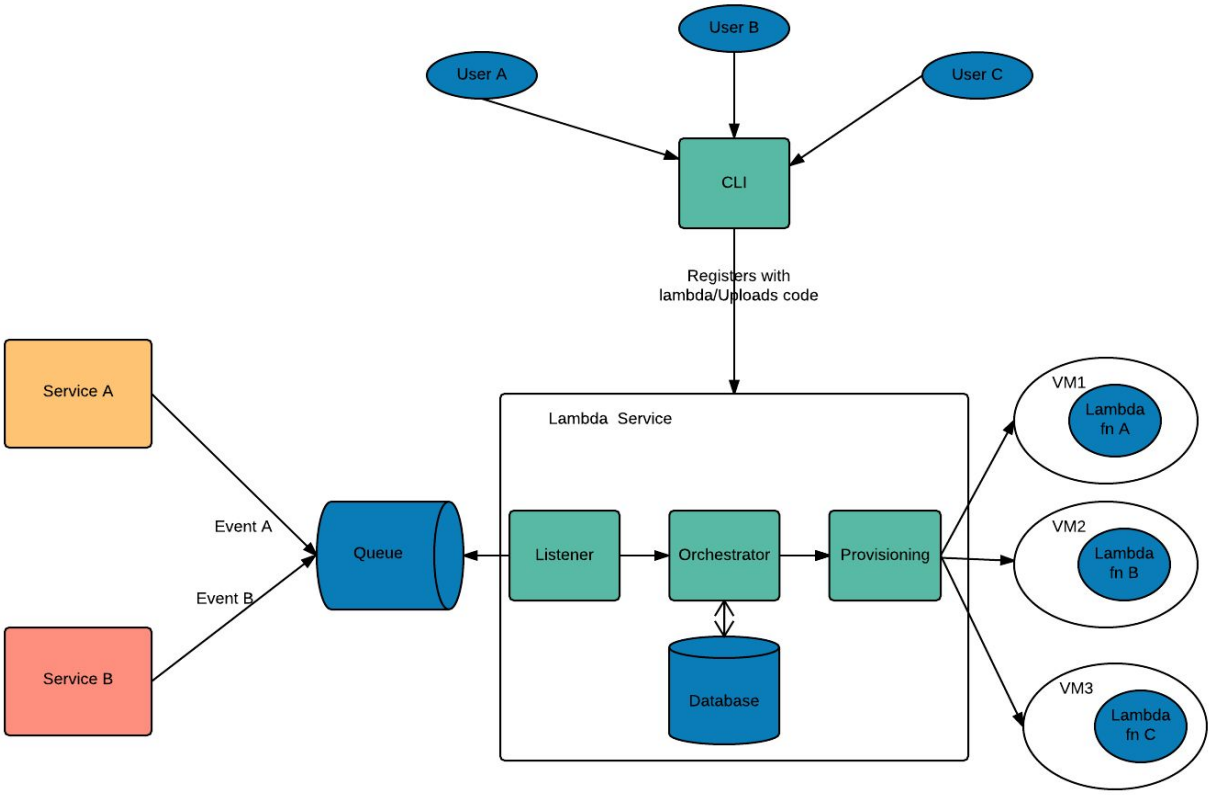
1. Create a service called “Lambda Service” which will allow to:
    - Expose an application or code as a service
    - Listen to events from an application in cloud or a service defined by the user.
    - Decide which event should trigger the execution of which code for a user and compute parameters for provisioning machines based on user input and requirements.
    - Provision machines on cloud based on computed parameters.
    - Configure the provisioned machines and run the user code on them.
    - Release the resources at the end of execution.

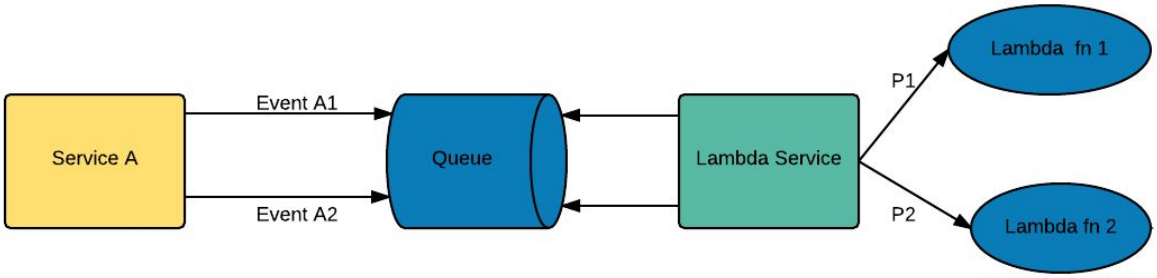
2. Provide a simple user interface to the users for:
    - Creating and managing user accounts for Lambda service.
    - Uploading code for a predefined running environment(javascript, Node.js).
    - Define events on which the provided code should run and what streams should generate those events.
    - Defining parameter(s) on which the machines will be provisioned.

Solution Concept

This section provides a high-level outline of the solution.

Global Architectural Structure Of the Project:





Each event is handled uniquely, i.e there is no maintaining of state. In the above diagram for example, two events from the same service i.e (Event A1= Event A2) execute in isolated environment from each other. Additional VM's can be provisioned based on requirement of Event A2 based on utilization.

There are 4 components to Lambda Service:

1) Listener - Listens to events from user specific services assuming each event is unique and should be handled differently. A queue based approach to handle incoming events will be implemented
2) Orchestrator - This component will decide the code to execute based on the event it receives and also compute parameters for provisioning resources on the cloud. The mapping between events and users will be picked up from the database. On successful provisioning of resources, it executes the code in the cloud.
3) Provisioning component - Provisions the resources based on a set of parameters given by the orchestrator. (eg - minimum 128 MB RAM, minimum 64GB storage)
4) Command line Interface(CLI) - The user will have to upload the code and specify what events his code should be triggered on through a CLI. Capstone account will be created for testing purposes.

The services could be any cloud specific service hosted by specific clouds like DynamoDB, S3 in Amazon's case or a third party service capable of:

1. Writing messages to a cloud supported by the cloud in which lambda service will be deployed
2. Should provide an interface for the Lambda service to read the queue through an API.

<u>Design Implications and Discussion:</u>

1. Queuing mechanism for listening to the events generated from service: Lambda service will support one way request-only event handlers(applications/code). So to have a reliable communication, we will have a persistent queue mechanism to receive events from the consumers. This will also keep the registered lambda functions independent of the consumers and have and ensured execution in case of any downtimes.

2. Separation of components in the Lambda Service: The lambda service performs the execution of a code in three states. listening to event, orchestrating, provision resources and execute the code. These stages are abstract components which can be deployed separately. This enables stateless workers to process the events as each event is independent to each other and has isolated running environment in terms of provisioned resources. Also single execution environment(e.g java or Node or python running env).
3. Database need in the lambda service: There is a need to map events to user specific code. The user specific code can be persisted along with the event in a database. The orchestrator queries the database based on the event it receives and maps it to the corresponding code, which is later handed off to the provisioning component
4. Execution verification: Once the code is deployed on the provisioned VMs, there should be a mechanism to check if the code deployed is actually executing. For this there is a need for us to develop a service which can be used by the developer to verify his code's results. The code will log its execution in the VM at the beginning. Just before the VM is destroyed we will need a mechanism to push the logs into a repository which the user can later query for correctness

Probable Tech Stack:

- Openstack - For provisioning and managing resources.
- Chef - Configuring the provisioned resources for running the code.
- NodeJs/Python - For event listening, orchestration, and provisioning
- RabbitMQ/Kafka for queuing mechanism.
- Trove/MySQL/MongoDB for persistence

Minimum Acceptance Criteria

1. A command-line interface to upload the code in order to host it as a service.
2. Provisioning minimum resources based on user specified/system determined parameters for an event.
3. Execute the code on a single execution environment provided by the user in response to an event.

Release Planning

Detailed description on trello board:
https://trello.com/b/81Fhk6yE/lambda-on-openstack-or-functions-as-a-service

Release #1

- Creating a deployable service capable of interfacing with a queue and put events in it on requirement basis. Create the event Listener component of the lambda Service which will listen to events raised.

- A component which should be able to provision machines through Openstack API's given provisioning parameters.

Release #2

- Configuring the provisioned machine environment ready for execution of the lambda function uploaded by user.

Release #3

- Orchestration component which will receive data from event listener and map the uploaded lambda function to execute. Integration between the components is done to make sure an event could be received and processed till execution of lambda function.

Release #4

- The orchestrator should be able to compute the provisioning parameters based on some user defined and system defined data. The computed parameters will be passed to the provision component for provisioning machines.

Release #5

- Simplistic user Interface for interacting with the Lambda Service to upload Lambda function and any other detail that the user needs to upload for running the lambda function.