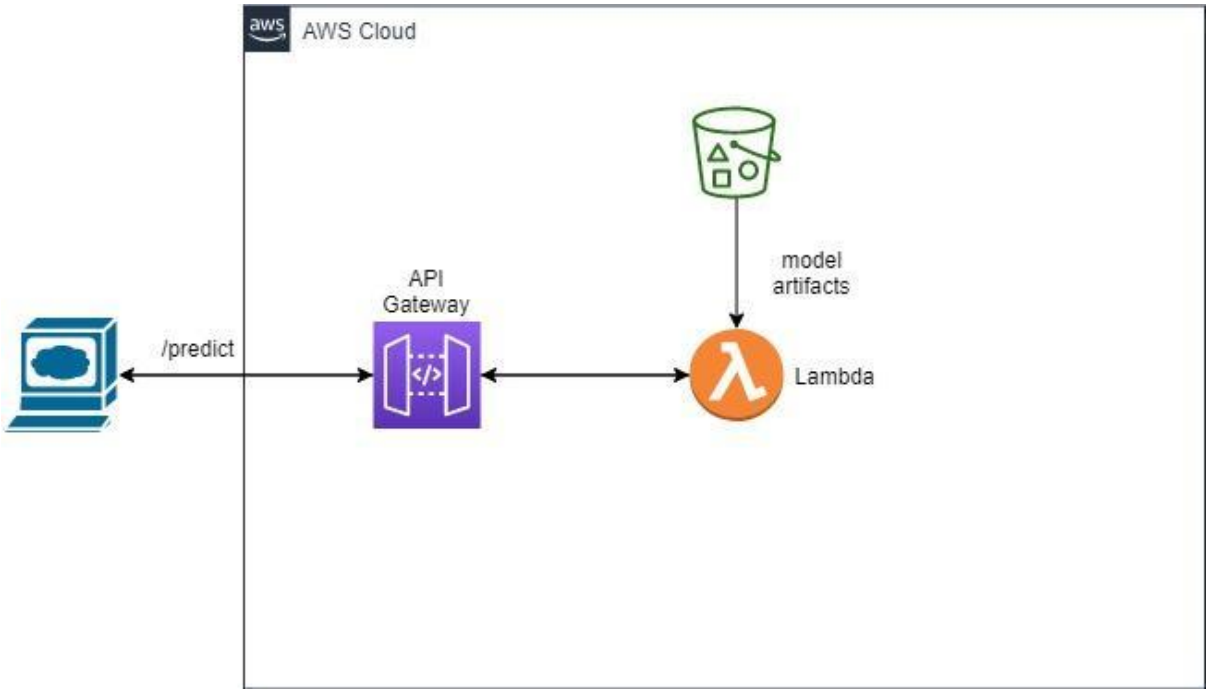
Serverless Architecture for Machine Learning

AWS API Gateway and AWS Lambda:

AWS lambda is serverless and FaaS (Function as a Service) which is suitable for data processing, ETL, web app, IOT backends etc. It automatically scales application by running code in response to the trigger. Your code runs in parallel and processes each trigger individually, scaling precisely with the size of the workload. All these happens without overhead of managing servers.



AWS Lambda:

1. Reads the model artifact from S3 bucket
2. With features given as input from the request, it returns back the predict using loaded model from S3.

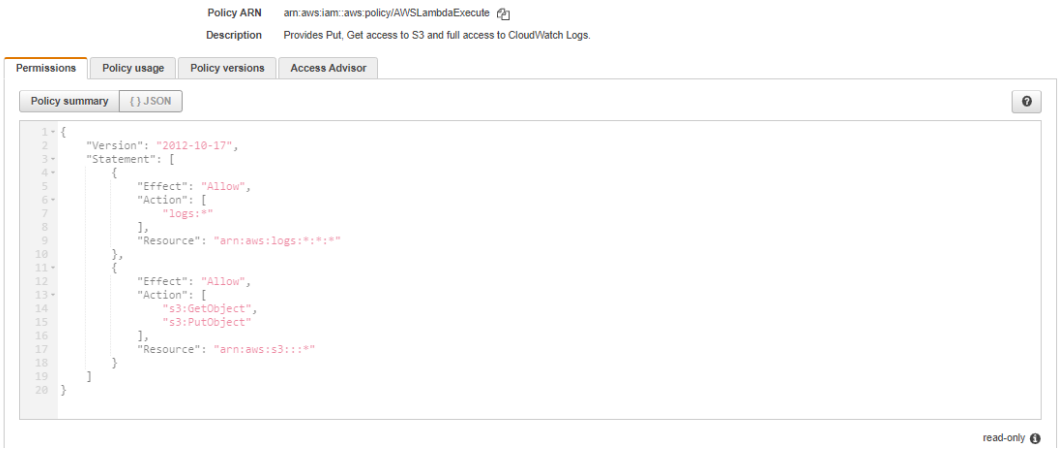
Layers:

To add the dependencies for lamda function, we can create layers and attach it to the lamba function. We zip the dependencies and upload it to s3 if its size is larger and create the layer. Once we create a layer and attach it lamda function, dependent libraries are placed in /opt directory. For instance, we zipped python/lib/python3.7/sitepackages/* and created a layer. Libraries will be unzipped as

/opt/python/lib/python3.7/sitepackages/

Permissions:

Above lambda function needs to access S3 bucket for model artifact. So, we need to create IAM role with policy **AWSLambdaExecute.** Below is the policy summary



API Gateway:

Trigger:

To call the lambda function, we need to create a trigger. In our case, we need to create API gateway. Post that create method and integration type as lambda. While creating method, it creates the permission to access lambda function.

Test API:

AWS API Gateway console has option to test the integration. All you need to do is give the request body.

Limitations:

1. Memory and CPU limit in AWS Lambda (3008 MB / 2 vCPUs)
2. Size of deployment package (max 250MB unzipped)

Pricing:

Pricing is for Request and Compute Charges.

Request Charges:

$0.20 per 1M requests

Total requests – Free tier request = Monthly billable requests

Monthly request charges = Monthly billable requests * $0.2/M

Compute Charges:

$0.0000166667 for every GB-second

Total Compute(seconds) = Number of Request * Bill Duration for every request

Total Compute (GB-s) = Total Compute(seconds) * Allocated Memory/1024 (For instance 128 MB)

Monthly Billable = Total Compute – Free tier compute (400,000 GB-s)

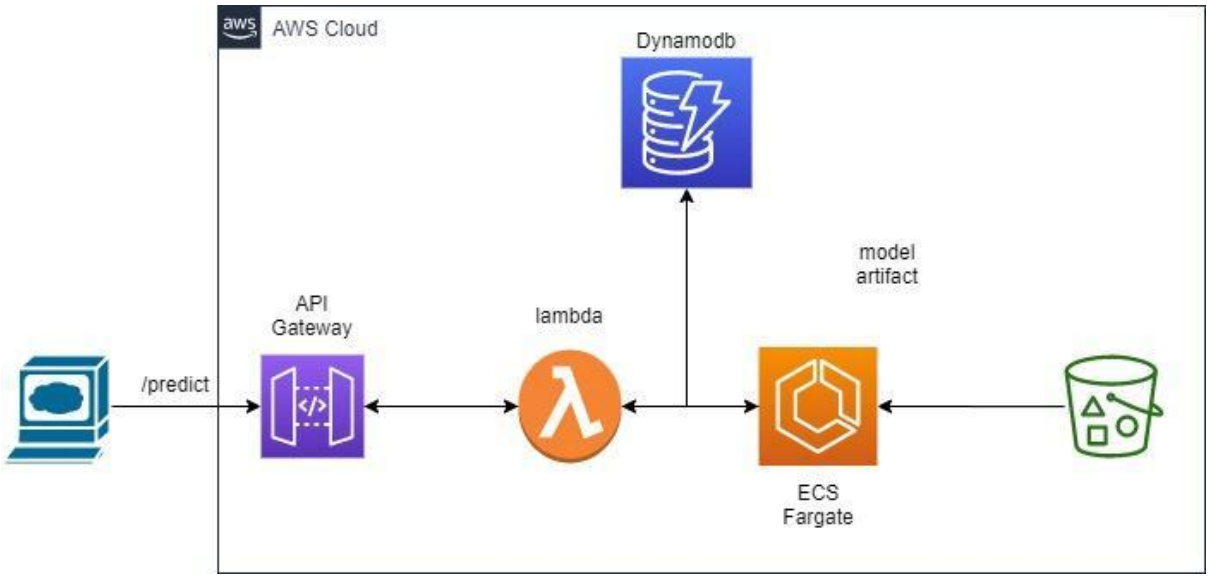
Monthly compute charges = Monthly Billable (GB-s) * $0.00001667

Total Compute:

Monthly compute charges + Monthly request charges

AWS Lambda and AWS ECS:

Since lambda has the limitations of cpu and memory, we need a different architecture that supports heavy compute and large size libraries.



Container:

Create the docker image that has all the scoring code and its dependencies and push it ECR. Scoring code updates the predictions in the dynamodb.

AWS ECS Fargate:

ECS is a great choice to run containers for several reasons. You can choose to run your ECS clusters using AWS Fargate, which is serverless compute for containers. Fargate removes the need to provision and manage servers, lets you specify and pay for resources per application, and improves security through application isolation by design.

Task Definition:

We need to create task definition that holds the configuration like task role, task size, container definition etc. Let's go through

Task Role: ecsTaskExecutionRole

We need to attach a role that has permission to execute ecs task and access dynamodb to write the incoming api request and its predictions. Role has below policy

AmazonDynamoDBFullAccess (Full access not needed. Can add whatever neccesary)

AmazonECSTaskExecutionRolePolicy (Full access not needed. Can add whatever neccesary)

Task Size:

The task size allows you to specify a fixed size for your task. Task size is required for tasks using the Fargate launch type and is optional for the EC2 launch type. Container level memory settings are optional when task size is set

Container Definitions:

Attach the scoring image that was uploaded to ECR.

ECS Cluster:

Create ECS cluster where we run the task that was created.

Integration between AWS Lambda and AWS ECS:

Now that we created the task definition, we call the ecs task with launch type as Fargate from lambda function as per the architecture.

Lambda function:

1. Inserts the request info into dynamo db
2. Runs the task that was created with lauch type as Fargate
3. Polls for the status from ECS
4. Updates the request record in dynamodb with FAILED or SUCCEDED

Permissions:

Create IAM role that has below policy and attach it to lamda function

AmazonDynamoDBFullAccess (Full access not needed. Can add whatever neccesary)

AmazonECS_FullAccess (Full access not needed. Can add whatever neccesary)

Limitations:

1. High latency due to ecs takes some time to run the container.
2. Lambda functions has to poll the ecs for status of the task to route back the response to api call. Due to this, extra cost gets added up.

Pricing:

For example, your service uses 5 ECS Tasks, running for 10 minutes (600 seconds) every day for a month (30 days) where each ECS Task uses 1 vCPU and 2GB memory.

Monthly CPU charges

Total vCPU charges = # of Tasks x # vCPUs x price per CPU-second x CPU duration per day (seconds) x # of days

Total vCPU charges = 5 x 1 x 0.000011244 x 600 x 30 = $1.01

Monthly memory charges

Total memory charges = # of Tasks x memory in GB x price per GB x memory duration per day (seconds) x # of days

Total memory charges = 5 x 2 x 0.000001235 x 600 x 30 = $0.22

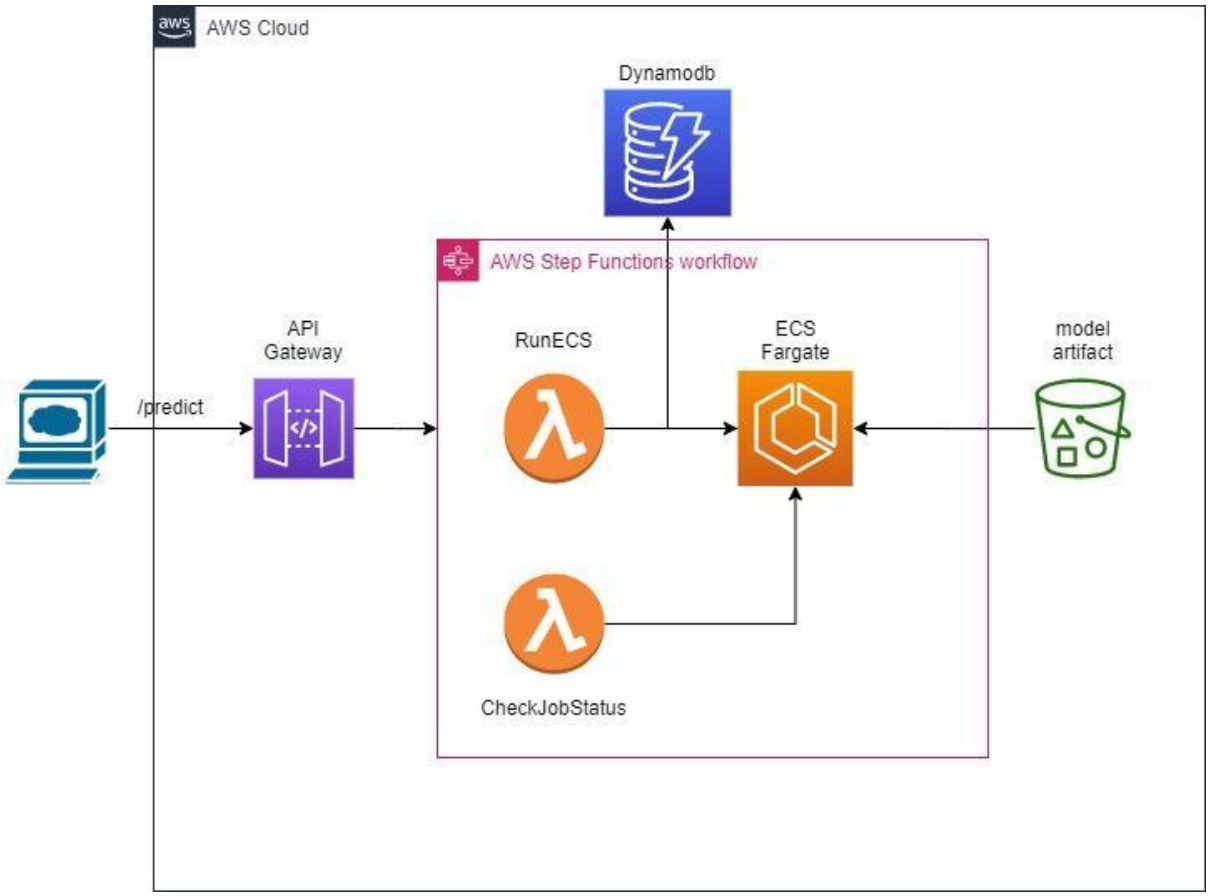
Monthly Fargate compute charges

Monthly Fargate compute charges = monthly CPU charges + monthly memory charges

Monthly Fargate compute charges = $1.01 + $0.22 = $1.23

AWS Step Functions and AWS Lamda:

Since AWS Lambda function has to poll for the status of ecs task, we need a service that takes care of this caveat and it should be cost effective. AWS step function does the polling using lambda function.



AWS Step Functions:

AWS Step Functions lets you coordinate multiple AWS services into serverless workflows so you can build and update apps quickly.

To create a step function, you need to have permission to execute the aws services. In our case, step function should have permission to call lambda function.

Permission:

Step Functions lets us create a new role for you based on your state machine's definition and configuration details. Moreover, you can also use existing role.

Type:

We chose **Standard** type for our workflow since polling takes sometime. Other option includes **Express** which is ideal for short duration and high-event-rate workloads.

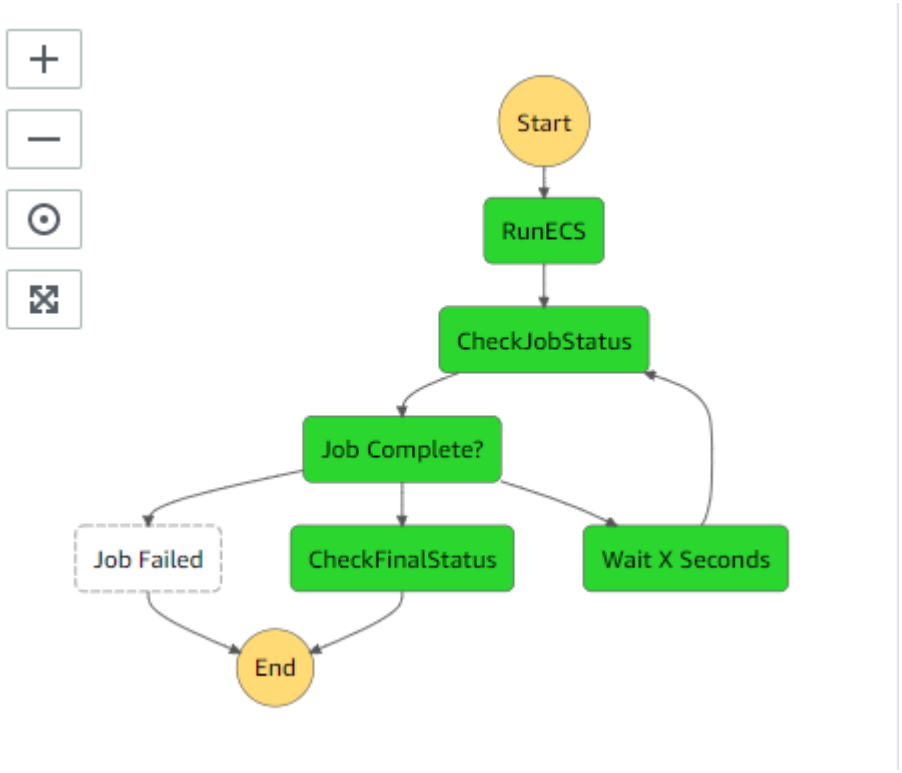
Definition:

You can define the workflow using **Amazon States Language**. State is nothing but the task/logic you want to execute. States if group of tasks that forms the workflow. In our case, States are as follows

RunECS: execute the lambda function that calls the ecs task with launch type as fargate

CheckJobStatus: execute lambda function that checks status of ecs task

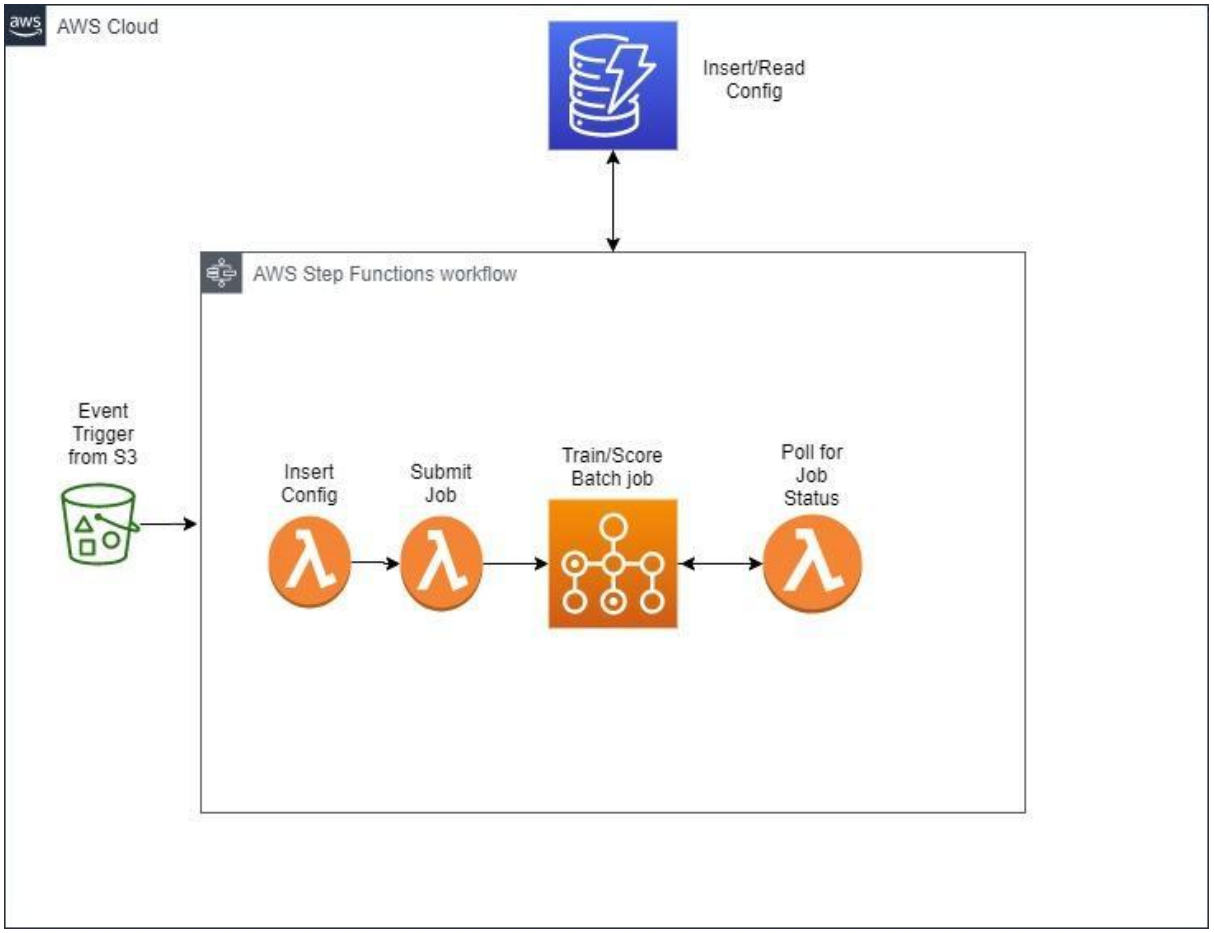
Step function Summary:



Pricing:

$0.025/1000 state transitions

AWS Step Functions and AWS Batch:



AWS Batch:

AWS Batch enables you to run batch computing workloads on the AWS Cloud. Batch computing is a common way for developers, scientists, and engineers to access large amounts of compute resources,

and AWS Batch removes the undifferentiated heavy lifting of configuring and managing the required infrastructure, similar to traditional batch computing software.

Job Definition:

AWS Batch job definitions specify how jobs are to be run. While each job must reference a job definition, many of the parameters that are specified in the job definition can be overridden at runtime.

Job Role:
AWS Batch internally creates the ECS cluster and Task definition while we include container image. To create the ecs cluster and run the task we need to have below IAM policy in a role. Moreover, since we use dynamodb to insert/read the job config, we need to policy for dynamodb

AmazonECSTaskExecutionRolePolicy

AmazonDynamoDBFullAccess

read-write-playground-s3-bucket

Job Attempts:

Specify the maximum number of times to attempt your job (in case it fails)

Environment:

Specify the CPU and memory to reserve for the container

Container Image:

For Container image, choose the Docker image to use for your job from a repository like ECS or Dockerhub.

Command:

Command to pass to the conainer.

Job Queues:

Jobs are submitted to a job queue, where they reside until they are able to be scheduled to run in a compute environment. An AWS account can have multiple job queues. For example, you might create a queue that uses Amazon EC2 On-Demand instances for high priority jobs and another queue that uses Amazon EC2 Spot Instances for low-priority jobs.

Job State:
The state of the job queue. If the job queue state is ENABLED (the default value), it is able to accept jobs.

Job Priority:
The priority of the job queue. Job queues with a higher priority (or a higher integer value for the priority parameter) are evaluated first when associated with the same compute environment.

Compute Environment Order:
The set of compute environments mapped to a job queue and their order relative to each other. The job scheduler uses this parameter to determine which compute environment should execute a given job. Compute environments must be in the VALID state before you can associate them with a job queue. You can associate up to three compute environments with a job queue.

Job Scheduling:

The AWS Batch scheduler evaluates when, where, and how to run jobs that have been submitted to a job queue. Jobs run in approximately the order in which they are submitted as long as all dependencies on other jobs have been met.

Compute Environment:
Job queues are mapped to one or more compute environments. Compute environments contain the Amazon ECS container instances that are used to run containerized batch jobs. A given compute environment can also be mapped to one or many job queues. Within a job queue, the associated compute environments each have an order that is used by the scheduler to determine where to place jobs that are ready to be executed. If the first compute environment has free resources, the job is scheduled to a container instance within that compute environment. If the compute environment is unable to provide a suitable compute resource, the scheduler attempts to run the job on the next compute environment.

Design:

1. Once job config file placed in S3, event will trigger a lambda function that inserts job config to dynamodb.
2. After job config inserted into dynamodb, a lambda function will submit the batch job to AWS Batch.
3. Lambda function will poll for batch job status.

Above workflow is orchestrated using AWS Step functions.

Step Function Summary: