Lambda

AWS Lambda allows code to execute in response to triggers caused by activity from other AWS resources, services, and applications. AWS Lambda provides this capability by allowing code written specifically for use in Lambda (called Lambda functions) to execute in an environment where the infrastructure becomes totally invisible and irrelevant. Scaling and server management are handled transparently by AWS. The user isn't even aware of, and has no visibility into, how the servers are organized to execute the functions – this is all hidden from view by AWS.

EKS

Amazon EKS is a managed service that you can use to run Kubernetes on AWS without needing to install, operate, and maintain your own Kubernetes control plane or nodes. Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications. Amazon EKS runs Kubernetes control plane instances across multiple Availability Zones to ensure high availability. Amazon EKS automatically detects and replaces unhealthy control plane instances, and it provides automated version upgrades and patching for them.

Fargate

AWS Fargate is a managed service offered by Amazon, which behaves as a compute engine for AWS ECS. It helps in running containers without the need to manage servers and clusters. Fargate provides, configures, and scales clusters of virtual machines which help in running containers. The overhead of choosing server types, scaling clusters at the right point in time and optimizing the cluster pack need not be looked after by the user. Fargate completely eliminates the need of the user requiring to interact with or worry about servers and clusters. It eliminates the user worrying about infrastructure management. It allows the user to focus completely on efficiently designing the application and building high level applications.

When to Use Fargate vs Lambda vs EKS

AWS Fargate can be classified as a tool in “**Container as a Service**” category, while AWS Lambda comes under **Serverless/Task processing** and AWS EKS is a **managed service for Kubernetes.**

As Fargate only runs docker containers, we can create our docker image locally and use Fargate to run docker containers using images. It is a serverless container management service which we can use to deploy our applications and we have to focus only on our application model and not on infrastructure. It allows us to build and manage applications in a serverless containerized environment and works in both ECS and EKS. We use Fargate when we don’t want the overhead to manage clusters or scale the environment manually. AWS Fargate does this operation manually.

AWS Lambda largely operates on customer events which gets triggered through some event which originates through other medium, like upload a file to S3 bucket. It is used where developers has to build custom applications and need to run in a serverless environment on immediate basis. There is no dedicated server based environment which we have to manage while running lambda functions.

AWS EKS is a managed service for Kubernetes provided by AWS which we can use to set up, run and manage Kubernetes clusters on AWS. This service leverages open source tool kubernetes, we can use EKS when we have to deploy applications over Kubernetes environment. We can also use Fargate type in AWS EKS itself while deploying and creating infrastructure.

Cost Comparison

With Fargate, there is no upfront cost and we have to pay for the vCPU and memory used in AWS Fargate. ECS and EKS does not allow us to configure Fargate to any amount of vCPU and memory. The amount of memory being specified in Fargate depends on the actual vCPU we are taking. Fargate pricing is not based on number of requests or request length, we are only paying for memory and CPU consumed by our containerized applications. The pricing is charged on the basis of per second and minimum is 1 minute. The main factors on which Fargate calculates cost is:

1. Total amount of CPU resources
2. Total amount of memory resources
3. Time running

AWS Lambda is basically billed on a combination of the number of requests, memory, and seconds of function execution. When we work on Lambda, it's very difficult to manage function memory requirements properly since it affects our cost. We will be charged based on the memory which we have provided in the Lambda function, no matter whether we are using the same amount or less.

AWS EKS is charged on $0.10 per hour for each EKS cluster we create. EKS is available in two models: EC2 and Fargate.
If you are using EC2 (including with EKS managed node groups), you pay for AWS resources (e.g. EC2 instances or EBS volumes) you create to run your Kubernetes worker nodes.
If you are using AWS Fargate, pricing is calculated based on the vCPU and memory resources used from the time you start to download your container image until the Amazon EKS pod terminates, rounded up to the nearest second. Minimum billing is of 1minute in case of Fargate.

Monitoring Comparison

Fargate - Fargate exports monitoring to cloudwatch directly. We can monitor it using cloudwatch and see all of the available metrics provided by Fargate. We can also monitor and send logs to cloudwatch. AWS provides container insights which provides even more visibility into your ECS Tasks with improved logging and metrics. While this feature is only in beta, it looks promising that Amazon is willing to put more effort to increase visibility with ECS.

Lambda - For Lambda monitoring, we can also use AWS Cloudwatch services and see all the available metrics and logs of every execution of the lambda function.

EKS - We can use Cloudwatch to fetch metrics and logs for AWS EKS also and take advantage of Container insights which collect metrics from AWS EKS and Kubernetes.

Security Comparison

EKS, Fargate and Lambda all three are deeply integrated with AWS IAM service, enabling customers to assign granular access permissions for each container and using IAM to restrict access to each service and delegate the resources that a container can access. EKS, conversely, does not have this integration.

How can we achieve faster cold start times for java applications for lambdas ?

A cold start happens when you execute an inactive function. The delay comes from your cloud provider provisioning your selected runtime container and then running your function.Cold starts can be a killer to Lambda performance, especially if you're developing a customer-facing application that needs to operate in real time. They happen because if your Lambda is not already running, AWS needs to deploy your code and spin up a new container before the request can begin. This can mean a request takes much longer to execute, and only when the container is ready can your lambda start running.

We cannot entirely avoid cold starts, but you can reduce their duration and frequency by using the following tips.

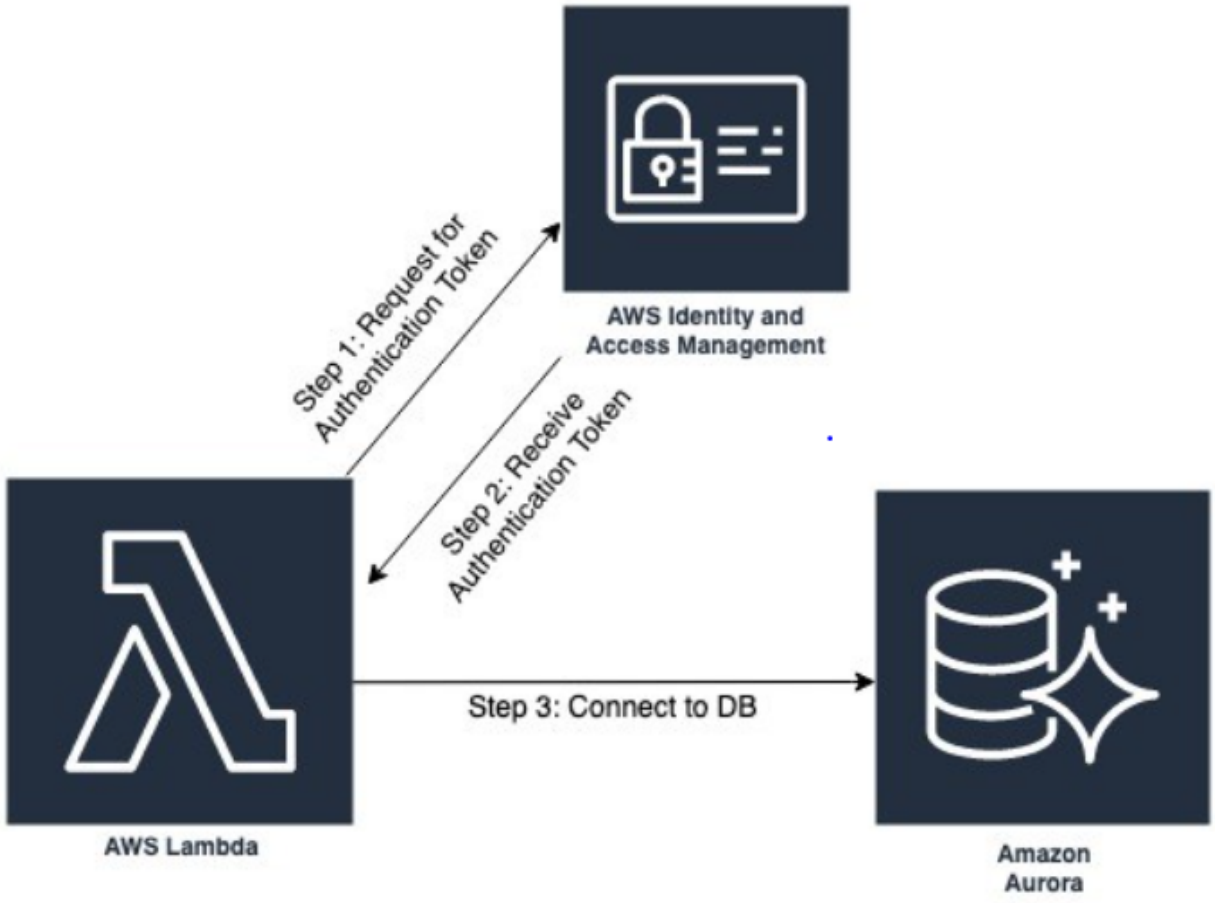
1. **Reduce the size of your function's artifact** - Use a different package for each Lambda function and only include the code necessary for that function. Only include the libraries necessary for each Lambda function in its artifact.
2. **Prefer dynamically typed languages**—use languages like Node.js or Python instead of statically typed programming languages like C# and Java. Dynamically typed languages check what you type during run-time as opposed to compile-time in statically typed languages.
3. **Avoid using Lambdas in Virtual Private Cloud (VPC)**—A VPC is an isolated, secure, private cloud hosted on a public cloud. VPC isolates your computing resources from each other, which can increase the delay time and cause cold starts.
4. **Avoid HTTPS calls inside your lambda**—SSL handshake and other security-related calls can create cold starts since they are limited by CPU power.
5. **Avoid dependencies**—Java dependencies that scan classpath like Spring can cause cold starts. In addition, loading Java classes can take some time and may lead to a cold start.
6. **Increase the memory on AWS Lambda to get more CPU capacity**—this can make the execution time of Lambda faster, and also reduce costs compared to lower memory settings.
7. Reduce the number of packages - We've seen that the biggest impact on AWS Lambda cold start times is not the size of the package but the initialization time when the package is actually loaded for the first time.The more packages you use, the longer it will take for the container to load them.

What is the right security pattern to use an application deployed on EKS or Lambda to connect to Elasticsearch ?

To connect with Lambda function with elasticsearch, we have to follow below steps:
1. Create and setup elasticsearch cluster environment on AWS
2. Write lambda function based on the application and install necessary packages required to run the same
3. Configure your lambda function to connect with ElasticSearch Service using the endpoint URL of elasticsearch cluster
4. Create an IAM role and attach permissions based on elasticsearch and lambda.
5. Modify the IAM role attached on Lambda function and attach elasticsearch permission accordingly.
6. Test the Lambda function

What is the right security pattern to use for an application deployed on EKS or Lambda to connect to the Aurora Postgres (Serverless) without using the data API ?

You can use IAM Database Authentication to connect to the DB cluster. Using this method, you can access the database with an authentication token generated instead of storing the password in a configuration file. Amazon Aurora generates an AWS Signature Version 4 authentication token that is valid for 15 minutes to create a connection from your application. As authentication is fully managed externally by IAM, you do not need to create credentials in the database.



Create a lambda function and specify the runtime based on application. Create an IAM role and attach permissions on it.

By default, a Lambda function has access to your public internet address and public AWS APIs. To access the private resources like Amazon RDS located in private subnets, you must enable your Lambda function for VPC. It's a best practice to configure the Lambda function in the same VPC where the database instance resides. Additionally, ensure that the private subnets have a NAT gateway added in their route tables to grant the Lambda function internet access.

Here's how you can access Amazon Aurora from a Lambda function using IAM authentication:

- You need the AWS Command Line Interface (AWS CLI) installed and configured on your machine before moving to the next steps.
- Enable IAM database authentication in the DB cluster using the AWS CLI as follows. You could also use the AWS Management Console or Amazon RDS API.
- aws rds modify-db-cluster --db-cluster-identifier <cluster-identifier> --enable-iam-database-authentication --apply-immediately
    - `cluster-identifier`: name of your DB cluster.
      Example:
    - aws rds modify-db-cluster --db-cluster-identifier mysql-demo --enable-iam-database-authentication --apply-immediately
- Connect to the DB cluster, and create a user with login privileges and grant IAM role access to the user:

  PostgreSQL: Grant `rds_iam` privilege to the user.

  CREATE USER <db_user_name> WITH LOGIN;

  GRANT rds_iam TO <db_user_name>;

Now, create the policy which allows the user created in the previous step to access the database using IAM database authentication. For more information, see IAM Policy and IAM Database Access.

Policy Document:

```
{
   "Version": "2012-10-17",
```

```
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "rds-db:connect"
         ],
         "Resource": [
            "arn:aws:rds-db:<region>:<account-id>:dbuser:<DbResourceId>/
<db_user_name>"
         ]
      }
   ]
}
```

Use the following settings.

- region: The region of the database instance.
- account-id: The AWS account number of the database instance.
- DbiResourceId: The DB instance identifier, or * to allow all the database instances in the region of the account.
- db_user_name: The user that has been created in the database. Mention a specific user, or * to allow IAM authentication for all users in the database.

Now attach this policy to the IAM role associated with the Lambda function:

- aws iam attach-role-policy --role-name <role_name> --policy-arn <policy_arn>
    - role_name: IAM role associated with the Lambda function (for example, demo-function-role-ipqlab4h).
    - policy_arn: ARN of IAM policy created for IAM database authentication (for example “arn:aws:iam::123456789012:policy/my-policy”).
- Set the following environment variables in Lambda.

- DBEndPoint: The DB instance endpoint (for example, ab9c3efgh31k.us-east-1.rds.amazonaws.com).
- DatabaseName: The name of the database.
- DBUserName: The user name created in the previous steps (for example, demouser).

To access the Amazon Aurora PostgreSQL database from your Lambda function, you must include the pg8000 (PostgreSQL Driver) library as part of your deployment package.

EKS Security

EKS is a fully managed service provided to AWS , that offers easy out of the box integrations with services like IAM, KMS and monitoring through CloudTrail, GuardDuty amongst others. These integrations greatly simplify the ability to implement encryption, authentication, authorization, and policy-based security models.

1. Secret Encryption – It is very important and can be done at a cluster launch. Kubernetes secrets are a way of storing sensitive configuration information so it can be accessed by the pods in your cluster. Commonly used secrets are passwords for applications, TLS certificates, and license files. By default, all secrets are base-64 encoded in ETCD to prevent them from being readily exposed to human eyes. There are several third-party Kubernetes secrets encryption tools that you can deploy, but when using AWS EKS the easiest way is to enable Secrets Encryption via AWS KMS. This will encrypt your secrets inside ETCD so only someone with the key can decrypt the data.

2. Private Control Plane – Controlling access to your control plane at the network layer should be the first thing you consider while launching your cluster. Best practice would be to limit access to the control plane to internal networks only. This way only users on VPNs or inside your corporate network will be able to administer the cluster.

3. RBAC – Role-based access control is essential to Kubernetes' security. Kubernetes offers native objects Roles and ClusterRoles to control how administrators and service accounts can interact with the API. Always follow the least privilege principles. Using namespaces in your environment to segregate different applications in your cluster allows for more granularity when creating RBAC Roles.

4. Audit Logging – Any security engineer would tell you audit logs are essential to any system and Kubernetes is no different. EKS has several logging options:

a. API Server – all cluster API requests

b. Audit – all changes and requests to Kubernetes

c. Authenticator – authentication requests

d. Controller Manager – state and actions of cluster controllers

e. Scheduler – Scheduling decisions

It would be best to enable all logging options and ship them to a SIEM. However, since EKS ships the logs to CloudWatch, when cost must be considered, enabling Audit Logs should be the bare minimum for security in EKS.

5. Deny- All Default Network Policy – Zero-trust architectures are becoming the new standard for security. The simplest way to implement zero-trust is to start by denying all inter-pod communication with a Network Policy (kind of like AWS Security Groups for Kubernetes), and add allow network policies for each individual service that needs to access another service – e.g. NGINX pods communication with MySQL pods.

6. Limit access to AWS API – There may be times when individual pods will need to interact with AWS services to perform their function. Every pod on a worker node may not require the same level of access to the AWS API, thus relying on the instance profile may grant pods unnecessary access. AWS recommends its native IRSA (IAM Role for Service Accounts) which allows you to create a service account, assign the service account to a pod, and allow the service account to assume an IAM role over OIDC. However, this usually requires updating the container images to the latest AWS SDK. When that is not feasible, consider deploying a tool such as kube2iam that will allow each individual pod in the cluster to use its own IAM role rather than using the instance profile of your worker nodes.

7. Use encrypted persistentVolumes and storageClasses – It should go without saying but encrypting data at rest should be the default for any application these days. This is no different when storing persistent data in Kubernetes. If you are using the EBS CSI, make sure the volumes are encrypted; if you are using EFS, make sure the file system is encrypted; any data at rest should be encrypted. This can be a deal-breaker for many compliance auditors especially in HIPAA and PCI scenarios.

8. Update your control plane and workers regularly – Like all systems in the IT world, vulnerabilities in Kubernetes will arise and they will require to be patched. Updating your Kubernetes environment to the latest version supported by your application manifests is essential. And if your manifests use very old Kubernetes APIs, it may be time to update them.

9. Use Managed Node Groups – Managed Node Groups provide excellent security posture – you can even configure them to block all remote access and can be upgraded seamlessly using the AWS API. Worker nodes require a very specific sequence to properly upgrade while minimizing downtime. Even the most seasoned Kubernetes Admins can make mistakes. When using Managed Node Groups, AWS automates the cordoning, draining, and scheduling of pods on your worker nodes during an upgrade. Don't reinvent the wheel.

10. Scan your container images- It is a good idea to regularly scan your images for vulnerabilities. Current AWS is working on a building solution for its container registry (ECR).

Maintenance on EKS

AWS EKS provides Fargate which makes it easy for you to focus on building your applications. 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.

Using Farget, there is no need for us to manage clusters or server, utilisation of resources become efficient, hence leads to less cost, seamless scaling, containerized applications, a standard unit for delivering software, serverless environment

Monitoring on EKS

Monitoring the environment is very important for business goals. EKS also provides a wide range of monitoring environment using AWS native service Cloudwatch using which we can monitor metrics and logs of Kubernetes cluster.
Cloudwatch provides lots of metrics to monitor and using Container insights functionality we can monitor, collect, aggregate and summarize metrics and logs based on containerized applications and microservices.

The cloudwatch container insights dashboard gives us access to the following information:

1. CPU and memory utilization
2. Task and service counts
3. Read/write storage
4. Network Rx/Tx
5. Container instance counts for clusters, services and tasks

We can also Integrate with CloudWatch Logs Insights to dynamically query and analyze container application and performance logs