1. **What is AWS Fargate and what problems does it solve?**

Answer:
AWS Fargate is a **serverless compute engine** for containers that works with **Amazon ECS and Amazon EKS**. It eliminates the need to provision, configure, or manage EC2 instances.
Problems it solves:

- **No server management:** You don't have to manage EC2 infrastructure.
- **Right-sizing:** You allocate only the required CPU/memory—no need to overprovision.
- **Scalability & security:** Each task/pod runs in its own kernel-isolated environment, improving security.

---

2. How does Fargate pricing work compared to EC2 instances? What kind of workload you will go for Fargate ?
Answer:
Fargate charges **per second** based on the **CPU, memory, and runtime duration** of your tasks or pods.
🔶 **Fargate vs EC2:**

Metric	**Fargate**	**EC2**
Pricing Model	Pay-per-use (CPU & memory secs)	Pay for full instance time
Management	Fully managed	Self-managed
Cost Efficiency	Great for short, bursty workloads	Better for long-running apps

It is like
RENT vs BUY
Decision

Use **EC2** if you have:

- Constant workloads
- Reserved instances
- Need for custom AMIs

Use **Fargate** if you want:

- Simplicity

- Auto-scaling without infra headaches

---

3. What are the supported CPU and memory configurations in Fargate?

Answer:

Fargate offers **specific combinations** of vCPU and memory. As of now:

- **CPU:** 0.25 vCPU to 16 vCPU
- **Memory:** 0.5 GB to 120 GB
- **Rules:**
  - Memory must be at least **2× vCPU** and at most **4× vCPU**

📌 Example configurations:

vCPU	Min Memory	Max Memory
0.25	0.5 GB	2 GB
1	2 GB	4 GB
4	8 GB	16 GB

Use [AWS Fargate task size documentation]() for full ranges.

---

4. Can you explain the difference between Fargate for ECS and Fargate for EKS?

Answer:

Feature	Fargate for ECS	Fargate for EKS
Orchestration	Amazon ECS	Amazon EKS (Kubernetes)
Abstraction	AWS-native abstraction	Kubernetes-native abstraction
Task type	ECS Tasks	Kubernetes Pods
Use case	Simpler, tightly-integrated apps	Complex, K8s-based apps

➡️ **Fargate for ECS** is simpler to set up and use.
➡️ **Fargate for EKS** offers more flexibility but requires knowledge of Kubernetes YAML specs and networking setup.

---

5. What are the limitations of AWS Fargate?

Answer:

While Fargate simplifies deployment, it comes with **limitations**:

- ❌ **No privileged containers** (e.g., Docker-in-Docker not allowed) (In GitHub Actions, GitLab CI, or AWS CodeBuild:
  If you try to run docker build inside a container job, it may **fail** or **hang**, because:
  The runner environment doesn't allow --privileged
  Docker daemon isn't accessible inside your container
)
- ⚠️ **No EFS support in EKS Fargate** (only ECS supports EFS volumes)
- 🔄 **Slower cold start** time vs EC2
- ⚙️ **Limited control over networking:** No custom ENIs per container
- 💰 **Can be more expensive** than EC2 for high-throughput apps
- ⛔ **No GPU support** in Fargate
- 📉 **No daemon sets support in EKS Fargate**
- 🔐 **Limited logging/monitoring customization**

DEMO FARGET

✅ Overview

We'll deploy a simple Dockerized web app (e.g., Nginx) using Amazon ECS with Fargate launch type. This walkthrough covers:

- How Fargate abstracts away EC2
- Pricing per task (vs EC2)
- Supported CPU/memory
- Fargate for ECS vs EKS
- Fargate limitations in real-time

🛠 Prerequisites

- AWS account
- AWS CLI configured
- Docker installed (for building the image)
- IAM permissions for ECS, ECR, CloudWatch, IAM, VPC

Deploy an NGINX Docker App on AWS Fargate

Step 1: Create a Docker Image & Push to AWS ECR

🔧 **Goal:**

Package your app into a Docker image and upload it to AWS Elastic Container Registry (ECR).

👣 **Steps:**

1. **Create a folder for your app:**

bash

```
mkdir fargate-demo && cd fargate-demo
```

2. **Add a basic Dockerfile using NGINX:**

bash

```
echo "FROM nginx:alpine" > Dockerfile
```

3. **Build the image:**

bash

```
docker build -t fargate-demo .
```

4. **Create an ECR repository in AWS to store this image:**

bash

```
aws ecr create-repository --repository-name fargate-demo
```

✅ This will return output like:

json

```
"repositoryUri": "906253564515.dkr.ecr.us-east-2.amazonaws.com/fargate-demo"
```

5. **Tag the Docker image to point to your ECR:**

```
bash
-----
aws_account_id=$(aws sts get-caller-identity --query Account --output text)
region=$(aws configure get region)
docker tag fargate-demo:latest
"$aws_account_id.dkr.ecr.$region.amazonaws.com/fargate-demo:latest"
```

6. **Login to ECR and push the image:**

```
bash
-----
aws ecr get-login-password | docker login --username AWS --password-stdin
"$aws_account_id.dkr.ecr.$region.amazonaws.com"
docker push "$aws_account_id.dkr.ecr.$region.amazonaws.com/fargate-demo:latest"
```

🛠️ Step 2: Create ECS Cluster

🔧 **Goal:**
Create a place where your app will run inside ECS.

👣 **Command:**

```
bash
-----
aws ecs create-cluster --cluster-name fargate-demo-cluster
```

✅ Output will show your cluster details:

```
json
-----
"clusterName": "fargate-demo-cluster", "status": "ACTIVE"
```

Step 3: Define Task Definition

🔧 **Goal:**
Tell ECS **how to run** your container (CPU, memory, image, port, etc.)

👣 **Create a file called task-definition.json with:**

```
json
-----
{
 "family": "fargate-demo-task",
 "requiresCompatibilities": ["FARGATE"],
 "networkMode": "awsvpc",
 "cpu": "256",
 "memory": "512",
 "executionRoleArn": "arn:aws:iam::<ACCOUNT_ID>:role/ecsTaskExecutionRole",
 "containerDefinitions": [
  {
   "name": "fargate-demo-container",
   "image": "<ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/fargate-demo:latest",
   "portMappings": [
    {
     "containerPort": 80,
     "protocol": "tcp"
    }
```

```
      ],
      "essential": true
    }
  ]
}
```

Replace <ACCOUNT_ID> and <REGION> with your actual values.

Step 4: Register Task Definition

Goal:

Let ECS know about the task definition you created.

👣 **Command:**

bash

```
aws ecs register-task-definition --cli-input-json file://task-definition.json
```

✅ Output shows successful registration:

json

```
"taskDefinitionArn": "arn:aws:ecs:us-east-2:906253564515:task-definition/fargate-demo-task:1"
```

Step 5: Create Security Group and Run the Task

🔧 **Goal:**

Allow web access and run the container.

👣 **Steps:**

1. **Get default VPC and subnet:**

bash

```
vpc_id=$(aws ec2 describe-vpcs --filters Name=isDefault,Values=true --query "Vpcs[0].VpcId" --output text)
subnet_id=$(aws ec2 describe-subnets --filters Name=vpc-id,Values=$vpc_id --query "Subnets[0].SubnetId" --output text)
```

2. **Create security group and allow port 80 (HTTP):**

bash

```
sg_id=$(aws ec2 create-security-group --group-name fargate-sg --description "Fargate SG" --vpc-id $vpc_id --query "GroupId" --output text)

aws ec2 authorize-security-group-ingress --group-id $sg_id --protocol tcp --port 80 --cidr 0.0.0.0/0
```

3. **Run the Fargate task:**

bash

```
aws ecs run-task \
  --cluster fargate-demo-cluster \
  --launch-type FARGATE \
```

```
  --network-configuration
"awsvpcConfiguration={subnets=[$subnet_id],securityGroups=[$sg_id],assignPublicIp=ENABLED}" \
  --task-definition fargate-demo-task
```

Step 6: Access Your App in Browser

🔗 **Goal:**

Get the **public IP** of the running container and open it in a browser.

👣 **Steps:**

1. **Get the running task ARN:**

bash

```bash
task_arn=$(aws ecs list-tasks --cluster fargate-demo-cluster --query "taskArns[0]" --output text)
```

2. **Get the network interface (ENI):**

bash

```bash
eni_id=$(aws ecs describe-tasks --cluster fargate-demo-cluster --tasks $task_arn --query "tasks[0].attachments[0].details[?name=='networkInterfaceId'].value" --output text)
```

3. **Get the public IP:**

bash

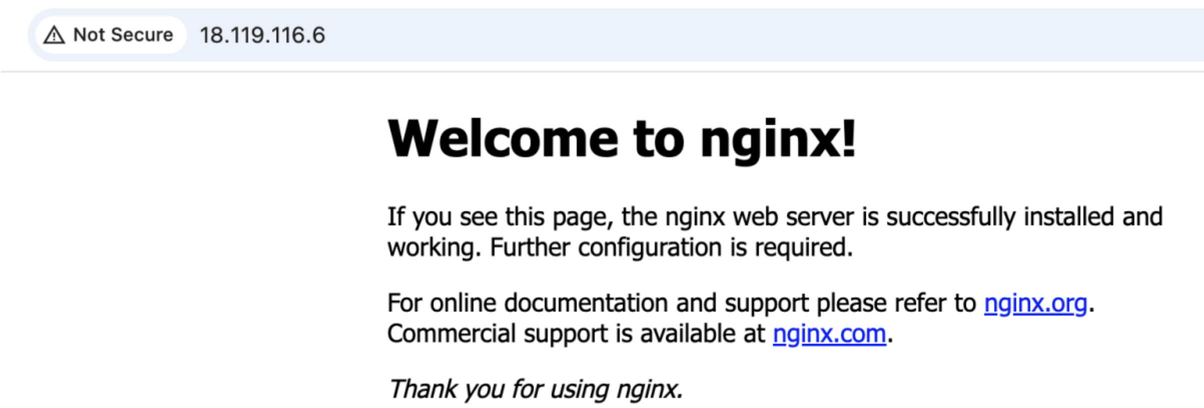
```bash
aws ec2 describe-network-interfaces --network-interface-ids $eni_id --query "NetworkInterfaces[0].Association.PublicIp" --output text
```

4. **Open in browser:**

cpp

```cpp
http://<public-ip>
```

You'll see the NGINX welcome page!



What is AWS Fargate primarily used for?

A. Hosting serverless web apps
B. Running containers without managing servers
C. Running Lambda functions
D. Managing EC2 Auto Scaling groups

B. Running containers without managing servers ✅

Which of the following is NOT a benefit of AWS Fargate?

A. Eliminates need for EC2 instance management
B. Enables overprovisioning for performance
C. Improves security through kernel isolation
D. Allocates just the required CPU and memory

C. Improves security through kernel isolation ✅

For which type of workload is AWS Fargate most cost-efficient?

A. Long-running batch jobs
B. Constant 24x7 workloads
C. Short, bursty, or unpredictable workloads
D. High-performance computing with custom kernels

C. Short, bursty, or unpredictable workloads ✅

Which of the following is a limitation of AWS Fargate?

A. Can’t scale automatically
B. Doesn’t support Kubernetes
C. Limited CPU/memory configuration options
D. Lacks integration with ECS

C. Limited CPU/memory configuration options ✅