**Chapter 7. AWS Compute Services**

**A NOTE FOR EARLY RELEASE READERS**

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

You love programming and spend your free time designing and creating new software applications on your personal computer. You recently created an application that converts input images to cartoon images and demoed this to your friend who happens to be a software engineer at a big tech company. The friend really liked the idea and suggested sharing the application with other people—plus it can also help you to earn some extra dollars. But how can you achieve this task? How can you make a program running on a personal computer accessible to other people?

For this specific example, you need some mechanism that allows people to send an image to a personal computer and the personal computer returns output as a cartoon image via the internet. Here, in this scenario, the personal computer is acting as a server. A server is nothing but a computer that is running most of the time (i.e. a highly available application) responsible for serving any kind of request. This server can be accessible just to you or any other computer or other users who all are referred to as clients.

Servers are an essential part of any application and AWS offers multiple compute platforms to run applications. A compute platform in AWS Cloud is a virtual server hosted in AWS data center which can be accessed over the internet. We’ll start off our discussion with a compute platform similar to our personal computer, called the Amazon Elastic Compute Cloud (EC2) machine and then move off towards AWS Lambda and containerization services present in AWS Cloud.

**Amazon Elastic Compute Cloud**

Amazon EC2 is a scalable virtual server hosting service provided by AWS. We can leverage [Amazon EC2](https://aws.amazon.com/ec2/) service to create virtual machines in AWS Cloud with a required set of configurations such as CPU, memory, storage, networking, etc. These virtual machines or virtual computing environments are referred to as instances. AWS provides the option to select bare metal servers or go with a virtualized environment maintained via hypervisor, which should be determined by identifying your business and compliance requirements. Here is a general description about hypervisor and bare metal servers:

*Hypervisor*

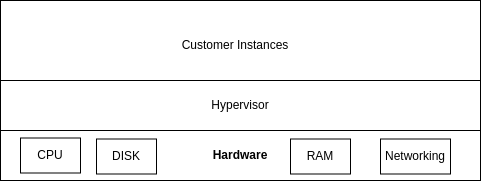
A virtualization component helpful in running multiple virtual machines on a single physical server as shown in [Figure 7-1](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch07.html#fig_1_xen_hypervisor). Hypervisor helps in allocating computing resources such as CPU and memory to virtual machines running their own operating system and applications. AWS supports two hypervisors based on available instance types – [Xen](https://xenproject.org/" \t "_blank) and [Nitro](https://aws.amazon.com/ec2/nitro/). [Nitro](https://www.allthingsdistributed.com/2020/09/reinventing-virtualization-with-nitro.html) system is a new generation hypervisor with enhanced security and performance.

*Bare Metal Server*

Bare metal servers are physical servers dedicated to a single tenant. They’re helpful for workloads which require direct access to Intel Xeon processor infrastructure such as [Intel VT-x](https://en.wikipedia.org/wiki/X86_virtualization#Intel-VT-x) or a strict compliance requirement or a need to run a custom hypervisor of your own. For example, run [Oracle hypervisor](https://docs.aws.amazon.com/whitepapers/latest/running-oracle-hypervisors-ec2-bare-metal/running-oracle-hypervisors-ec2-bare-metal.html) instead of Nitro..

**NOTE**

One Amazon EC2 instance doesn’t necessarily map to one physical server in AWS Cloud.



**Figure 7-1. Xen Hypervisor**

Launching and running instances in AWS Cloud is easy and removes the headache of maintaining physical hardware. Amazon EC2 instances can be launched in just a few clicks on the AWS console following the steps on the [AWS page](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/LaunchingAndUsingInstances.html). The instances are launched with a specific set of configurations determined by Amazon Machine Image (AMI).

**Amazon Machine Image**

We launch an Amazon EC2 instance with the different configurations, such as: the Operating System the instance is launched with; storage volumes attached to an instance; install dependencies before instance boot up, custom security configurations, etc. These configurations can be defined via a pre-configured template, referred to as Amazon Machine Image (AMI), which enables easy and efficient provisioning of EC2 instances with desired software stack. Consider a scenario where you as an engineer have responsibility to install security dependencies to all the EC2 instances launched in organization. One way is to run scripts on these EC2 instances after they are launched but a more optimal way could be to create a base AMI with all required dependencies installed. This essentially saves a lot of time and ensures 100% confidence that none of EC2 instances is missed for required dependency.

We, as customers, can launch EC2 instances using freely available AMIs from the AWS community, purchase AMIs from third parties such as Red Hat, or create our own custom AMIs depending on our needs. For example, Amazon Linux and Amazon Linux 2 AMIs are available free of cost and are fully supported and maintained by AWS. AWS provides a marketplace, where users can choose from a wide variety of publicly available AMIs. Here are few considerations for choosing the best AMI for your business use-case:

*Region*

AMIs are created specific to the region. If an AMI is not available in a specific region, you can [copy the AMI](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/CopyingAMIs.html) to the required region and further use it to launch EC2 instances.

*Operating System & Architecture*

EC2 instance runs on Operating System and you can choose from multiple Operating System flavors(32-bit or 64-bit architectures) [supported by AWS](https://docs.aws.amazon.com/systems-manager/latest/userguide/prereqs-operating-systems.html).

*Launch Permissions*

An AMI can be owned by AWS, businesses or individuals. As per the ownership, an AMI can be available to the general public, a set of users or just a single individual.

*Root Device Storage*

The AMIs are either backed by Amazon EBS or by Instance Store (refer Chapter 9–Storage Services for more details) as root device storage. The root device storage contains the image used to boot the EC2 instance. There are significant [differences](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ComponentsAMIs.html#storage-for-the-root-device) in both of these storage choices, such as size limit (64 TB for io2 EBS Block Express and 10 GB for instance store), data persistence, boot time (usually less than 1 minute for EBS and less than 5 minute for Instance Store), etc.

**NOTE**

You can check complete details on Amazon EC2 instance root device storage [here](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/RootDeviceStorage.html).

As we finalize AMI for an EC2 instance, the next decision we need to make is the instance type. AWS offers a wide range of instance types based on different capabilities and we can choose which best suits our requirements.

**Instance Type**

The hardware requirements can vary from customer to customer and with this in mind, EC2 service provides different instance types. Instance types in AWS EC2 define the hardware and performance characteristics of an EC2 instance. We might not always have a concrete answer to which instance type is most suitable and in these scenarios, we can start by installing our application on any general purpose instance and then perform load testing. This direction will help determine the limits where this specific instance breaks and it can eventually help in figuring out instance type and number of instances needed for the workload.

Different instance types offer varying combinations of CPU, memory, storage, and networking capacities. Here are different instance type families available in AWS Cloud:

*General Purpose Instances*

General purpose instances are suitable for most of the workloads in the industry and they keep a fine balance between compute, memory and network resources.

*Compute Optimized Instances*

Compute Optimized Instances are suitable for CPU intensive workloads such as high performance computing, big data analytics, etc.

*Memory Optimized*

Memory Optimized Instances are suitable for workloads with requirements for large data sets processing in RAM such as in-memory caches.

*Storage Optimized*

Storage Optimized Instances are suitable for workloads with requirements for high, sequential read and write access to large data sets on local storage such as databases.

*Accelerated Computing Instances*

Accelerated Computing Instances are GPU-based instances and suitable for workloads with requirements to high performance computing, computational finance, machine learning, etc.

Choosing the appropriate instance type is crucial for achieving optimal performance and cost-efficiency for specific workloads. For updated guidance on instance types and instance families, refer to the [AWS page](https://aws.amazon.com/ec2/instance-types/) on Amazon EC2 instance types. There are some additional key considerations as we launch an EC2 instance:

* To establish connection with an instance after it has launched, you should select a key pair to enable secure shell(SSH) access to the instance.
* EC2 instances are launched at the AZ level within a specific region. EC2 allows you to modify default network configurations, such as the VPC instance it should be launched in, subnet in VPC, security group, etc. These configurations help in determining how the connection to launched instances will be established.
* We can choose to assign public IP to an instance. Public IP assignment helps in connecting to an instance directly from the public internet given that we’ve appropriate permissions either via security groups or NACLs.

**NOTE**

For production environments, we recommend using AWS CloudFormation or any other tool to launch infrastructure via code instead of AWS Console. This will help in better maintainability and replicating the infrastructure in any other required region or AWS account.

For running the application at scale, we need a good balance of vertical scaling and horizontal scaling for a number of instances in the entire infrastructure. By looking at usage metrics, we can figure out the minimum number of instances that will be required throughout a longer period of time. For such scenarios, cost can be reduced by using [Reserved](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-reserved-instances.html) EC2 instance capacity to get up to 72% discount compared to On-Demand instance pricing based on the usage commitment– one-year or three-year commitment. Note that On-Demand instances are instances that we launch as needed from AWS Console, while we pre-commit to use EC2 instances for a period of time in case of Reserved instances.

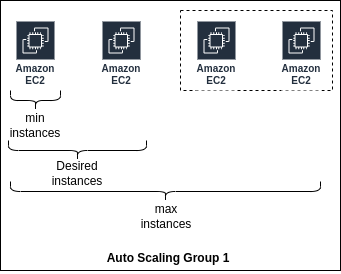
In contrast to Reserved Instances, there may be use-cases where we’re not much worried about losing an instance while the workload is running or the workload can be resumed later on once instances are available in the EC2 pool. For such scenarios, we can leverage something called Spot[instances](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html). Spot instances are less expensive than On-Demand instances. Let’s take an example, assume us-east-1a AZ has a total capacity of 100 m4.large instances. Out of these 100, half of them are already in use by customers. Further, based on AWS historic analysis, 30 instances will be requested by customers shortly as dedicated capacity. In this EC2 pool of 100 instances, there are 20 such instances left that no-one is getting benefit from—neither AWS nor customers. So these instances are made available to customers as spot instances at a lesser cost. For scenarios such as AWS customers requiring more than the 30 as predicted earlier, any of the spot instances can be turned off and assigned to requesting customers.

Traffic on applications running on top of EC2 instances can vary across the day and we might not always require a fixed number of running EC2 instances. AWS offers an Auto Scaling feature which helps in automatically adding and removing instances for the application. For example, the application requires 10 instances at noon but only two at midnight. Let’s dive into that next.

**Auto Scaling**

Auto scaling is an AWS feature that automatically adjusts the number of EC2 instances in the collection in response to changing workload demands. We can create a collection of EC2 instances based on any factor, such as AZ, instance type, etc., and these collections are referred to as Auto Scaling groups(ASG) when auto scaling is enabled on them. In these groups, we can specify minimum and maximum number of instances and apply an auto scaling policy for automatically adding and removing the instances within this limit. It helps maintain application performance and availability by dynamically scaling the capacity up or down based on predefined scaling policies. Autoscaling can be configured to scale based on metrics like CPU utilization, network traffic, or custom metrics defined by the user. It ensures that the application scales seamlessly to handle increased traffic and reduces costs during periods of low demand.  For example, refer to [Figure 7-2](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch07.html#fig_2_auto_scaling) for how a new instance is added as cumulative CPU Utilization limit is breached for ASG:

* For Auto Scaling Group 1, the minimum number of instances configured is one and the maximum is four.
* Assume that the auto scaling policy states to add a new instance to the group if maximum CPU Utilization exceeds 75% for continuous 5 minutes.
* Now as the limit is breached, auto scaling kicks in with the desired number of instances as two from one and it launches a new EC2 instance.



**Figure 7-2. Auto Scaling**

Large applications in AWS Cloud might be running thousands of EC2 machines to serve the customer traffic or internal system processing and failures are bound to happen. The EC2 instances might go down due to unforeseen hardware failures and we as customers should have proper mechanisms in place to overcome these failures. There are multiple ways to handle these issues such as:

* Auto Scaling policies to always maintain the desired number of instances within a group.
* EC2 instances launched in multiple AZs to withstand AZ failure.
* EC2 Auto Recovery is one mechanism to relaunch the instance in case of failure. The instance is relaunched with the same configurations, such as instance Id, metadata, attached EBS volumes, private IP, and elastic IP addresses.

EC2 is the oldest and most widely used service for running applications in AWS Cloud. EC2 provides control to maintain the software stack and maximum visibility of hardware operations. However, it might not always be a requirement to know how application software is deployed or you might want to avoid any operational maintenance of systems such as OS patching. For these scenarios, AWS offers serverless services where customers don’t need to worry about instance provisioning or capacity planning. Serverless, as the name dictates, abstracts out server details and directly allows us to run our workloads. AWS Lambda operates on a serverless model allowing code execution with required scale. Let’s dig deeper into AWS Lambda next.

**AWS Lambda**

AWS Lambda is a fully managed serverless compute service that lets users run code without overhead of server maintenance. The only thing we need in order to run our applications on AWS Lambda is code in our preferred language, referred to as Lambda function. It supports multiple programming languages and enables rapid development and deployment of event-driven applications. The backend architecture of Lambda is provisioned as per function runtime and executed on customer invocation events. Lambda automatically scales the underlying infrastructure to match the workload, providing high availability and reliability. Here are the main concepts in relation to the AWS Lambda service that you’ll need to understand:

*Function*

Function is an application’s codebase in your preferred programming language supported by AWS Lambda and is invoked with the help of a trigger to execute specific logic.

*Trigger*

Trigger is any resource or configuration which holds the responsibility of invoking Lambda function—for example, [invocation](https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html) via AWS console or via any AWS services such as [Simple Queue Service](https://aws.amazon.com/sqs/)(SQS), [DDB streams](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html), [Kinesis](https://aws.amazon.com/kinesis/), etc.

*Event*

Event is a custom JSON-formatted document or any other structure specific to AWS service, such as SNS notification, which is used to pass on information to Lambda function for processing.

*Lambda Execution Environment*

Lambda backend infrastructure internally creates a secure and isolated runtime environment for function execution. This runtime environment is referred to as the execution environment and is set up with help of user inputs such as function language runtime, available memory and maximum execution time (maximum supported configuration is 15 minutes) for function. The lambda execution shuts down as the execution time elapses. The execution environment is launched with specific instruction set architecture and Lambda offers two type of architectures to determine computer processor type:

* arm64 with 64-bit ARM architecture for AWS [Graviton2](https://aws.amazon.com/ec2/graviton/) processor.
* x86\_64 with 64-bit x86 architecture for x86-based processors. We recommend using arm64 architecture for cost and performance efficiency over x86.

*Deployment Package*

The Lambda function code can be deployed via deployment package in below two ways:

* A .zip file with code and its dependencies stored in Amazon S3 with maximum allowed size of 250MB. Lambda holds responsibility for providing operating system and runtime for function.
* A container image stored in [Amazon Elastic Container Registry](https://aws.amazon.com/ecr/) (ECR) with code and its dependencies with maximum allowed size of 10GB. The image should be compatible with [Open Container Initiative](https://opencontainers.org/) and should have the operating system and runtime included in itself.

*Layer*

Common code such as utilities or libraries can be zipped as layers instead of adding them to each Lambda function. Layers can be attached to Lambda and help in code sharing, reducing overall Lambda function code bundle size and the startup time. Layers additionally support versioning to maintain multiple versions of code libraries and ensuring backward compatibility.

*Destination*

Destination is another AWS service that is configured to receive invocation records on completion (success/failure) of asynchronous Lambda execution. AWS supports standard SQS queue, standard SNS topic, Lambda function and EventBridge event bus as destination.

**NOTE**

AWS continually adds new language runtimes—check for updated guidance on supported language runtimes [here](https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html).

The way Lambda function should be invoked for processing depends on your  use-case—some options include an [object addition to S3 bucket](https://docs.aws.amazon.com/lambda/latest/dg/with-s3-example.html) or scheduled weekly via [Cloudwatch events](https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/WhatIsCloudWatchEvents.html" \t "_blank). You may choose a specific trigger for lambda invocation depending on your use-case. Lambda supports three types of [invocation mode](https://docs.aws.amazon.com/lambda/latest/operatorguide/invocation-modes.html)s–synchronous, asynchronous or poll-based invocation:

*Synchronous Invocation*

Synchronous invocation can be achieved in multiple ways—via  API Gateway, Application Load Balancer, AWS CLI, etc. In this invocation mode, the customers wait for a response until Lambda completes its execution. It is preferred for latency sensitive workloads with a maximum of 15 minutes execution timeout. In case Lambda is triggered via any AWS service, the execution timeout could be less as well, such as 29 seconds for API gateway.

*Asynchronous Invocation*

Asynchronous invocation can be achieved via S3, Simple Notification Service(SNS), CloudWatch events, etc. In this invocation mode, response is returned immediately and request is queued by Lambda for processing. It can additionally be configured to handle retries and direct invocation response to configured [destination](https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#invocation-async-destinations) unlike synchronous mode of invocation.

*Polling Invocation*

This is the preferred mode for stream or queue based services such as[DDB streams](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html), [Kinesis](https://aws.amazon.com/kinesis/), SQS queues or [Kafka](https://aws.amazon.com/msk/). In this invocation mode, Lambda holds responsibility to poll the AWS service and synchronously invoke Lambda function for processing. The good part is AWS doesn’t charge customers for message polling and we only pay for Lambda function invocation. The retry mechanism is dependent on the data expiration in AWS service, such as from one minute to 14 days for SQS queue.

We mentioned Lambda provision resources on invocation, which could result in more request processing time due to cold start problems. Cold start refers to the time Lambda takes to provision resources and prepare an execution environment involving function code download from S3 or ECR, creating an environment with memory and runtime configurations and executing any initialization code out of main function code. After this complete set up, Lambda starts serving customer requests. Cold start time could vary from 100 ms to 1 second and typically occurs for 1% of total invocations as analyzed by AWS on production workloads. AWS recommends multiple ways to reduce cold start problems as described below:

*Warm Start*

The execution environment is not deleted as soon as the function completes its execution, it is retained for some time so as to serve continuous requests invoking the same Lambda function. This ensures skipping the set up of execution time on each Lambda invocation, referred to as warm start. The cold start is not in the customer’s hands and Lambda can create an execution environment on a need basis. For example, Lambda creates a fresh execution environment for concurrent invocation to handle increasing traffic.

We can use warm start for our own advantage by intentionally invoking the Lambda function with dummy input before it serves actual requests. This approach is only recommended for scheduled or low-traffic workloads and not for production environments operating at large scale.

*Provisioned Concurrency*

Provisioned Concurrency ensures the execution environment is set up in advance ready to serve requests. For example, Lambda function with Provisioned Concurrency of four will create four execution environments in parallel and it can be extremely helpful for functions with large initialization code-base.

*Memory Configurations*

AWS provides an option to configure memory that is available for Lambda function varying from 128 MB to 10,240 MB. The compute power allocated to Lambda function is proportional to configured memory so it is advisable to tune memory settings to improve performance. You can come up with the best memory configuration for your workload via running test runs on Lambda function. Additionally you can explore[AWS Lambda Power Tuning](https://github.com/alexcasalboni/aws-lambda-power-tuning) tool for this purpose.

*Static Initialization Optimization*

As lambda prepares the execution environment, it configures any required connections or download code dependencies. A large number of dependencies can definitely add to the cold start time, so it is recommended to import only required dependencies. Additionally if the function code base becomes too large where such a scenario is not feasible, other avenues can be explored such as breaking a function into multiple child functions.

*SnapStart*

For Java11 runtime environments, snapstart improves function start-up time by taking an encrypted snapshot of memory and disk state of the initialized execution environment. The new function invocation resumes its execution from the cached snapshot,  therefore improving on cold start time.

As we mentioned, Lambda supports a variety of invocation modes and scales automatically for required traffic. The key deciding factor when deciding whether to choose Lambda as a compute platform is deciding whether you’re fine with too much abstraction of infrastructure. Consider: Is there a requirement to fine tune system efficiency at the infrastructure level? Is it ok if there is a latency spike due to cold start issues? These are some of the questions you should try to answer in order to make your decision.

If you’ll recall, we took a deep dive into different containerization concepts in Chapter 6. Here we’ll explore how we can use containers as a compute platform and launch our systems on AWS Cloud.

**Containerization Services**

AWS offers container orchestration service for running Docker containers via two services, referred to as Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS). ECS is a fully managed container orchestration service offered by AWS while EKS is a managed service for running open-source [Kubernetes](https://kubernetes.io/) on AWS cloud.

**NOTE**

This section is built on top of concepts covered in Chapter 7—please make sure you’ve read through that chapter first.

**Amazon Elastic Container Service**

Amazon ECS is a fully managed highly available service that helps with deployment, operational management and scaling containerized applications for the required traffic load. ECS integrates with other AWS services, such as EC2, ECR (Elastic Container Registry), and Elastic Load Balancer, to provide a complete container management solution. Let’s cover some important terminology related to ECS:

*Task*

Task is the basic unit of deployment in ECS responsible for running one or more containers as shown in [Figure 7-3](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch07.html#fig_3_ecs_task_service_and_cluster_representation).

*ECS Service*

ECS service groups identical tasks to scale and monitor in a single place.

*ECS Cluster*

The infrastructure is registered to an ECS cluster, which acts as logical grouping of ECS tasks or services running via ECS Fargate or ECS EC2 launch types, or both types in the single cluster.

*Task Definitions*

Task Definition defines various [configurations](https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definition_parameters.html) for a task, such as runtime platform, task size, launch type, data volumes, environment variables, container definition for configuration specific to container launched within a task, etc. Task definitions enable you to define complex multi-container applications and specify how they should be orchestrated. Here is an example with Fargate as the launch option:

{

   "family": "fargate-task-definition",

   "containerDefinitions": [

      {

         "image": "aws\_account\_id.dkr.ecr.us-west-2.amazonaws.com/repository:tag",

         "name": "my-fargate-application",

         "portMappings": [

            {

               "containerPort": 80,

               "hostPort": 80,

            }

         ]

      }

   ],

   "essential": true,

   "cpu": "256",

   "memory": "512",

   "executionRoleArn": "arn:aws:iam::aws\_account\_id:role/ecsTaskExecutionRole",

   "networkMode": "awsvpc",

   "runtimePlatform": {

        "operatingSystemFamily": "LINUX"

    },

   "requiresCompatibilities": [

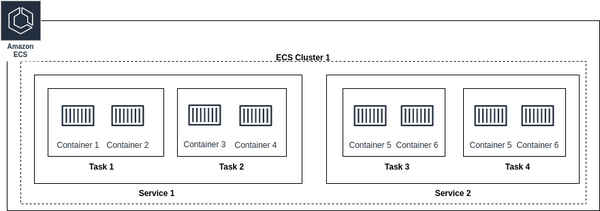
       "FARGATE"

    ]

}

*Amazon Elastic Container Registry (ECR)*

[ECR](https://docs.aws.amazon.com/AmazonECR/latest/userguide/Registries.html) is an AWS managed container image registry which can be used to push, pull and manage docker or Open Container Initiative (OCI) images.



**Figure 7-3. ECS task, service, and cluster representation**

There are two types of launch options available with ECS, the first being ECS EC2 where we manage our containers on a fleet of EC2 machines, and the second being ECS Fargate, a serverless option where we hand over complete operational management responsibility to AWS. The key differences between these two deployment options are captured in Table 11-1.

|  |  |  |
| --- | --- | --- |
| Comparison Factor | Amazon ECS EC2 | Amazon ECS Fargate |
| Operational Management | Compute layer is managed by customers—such as instance type and number of EC2 instances, application scaling, OS level patching, etc. ECS EC2 supports auto scaling feature to adjust number of instances based on resource utilization or application demand. | Serverless, meaning customers don’t have to worry about server management tasks such as applying OS security patches. We can specify OS, [vCPU, memory](https://aws.amazon.com/blogs/containers/how-amazon-ecs-manages-cpu-and-memory-resources/) and auto scaling configurations. |
| Pricing | There is no separate cost for ECS, customers are [charged](https://aws.amazon.com/ec2/pricing/) for EC2 instances running time, plus any storage selection, such as EBS volumes. | [Cost](https://aws.amazon.com/fargate/pricing/) is based on vCPU, memory, CPU architecture, and storage selection for ECS tasks or EKS pods, and customers are charged only when container workloads are active. |
| Business use-cases | AWS recommends ECS EC2 for workloads with high CPU or memory requirements, cost optimization requirements, application requirements for persistent storage access or compliance and organization requirements to manage infrastructure on your own. | AWS recommends ECS Fargate for large workloads to be optimized for low overheads, small workloads with occasional bursts, and batch workloads. |
| Limitations | Extra operational overhead at the customer’s end for selection of EC2 instances, maintaining fleets, OS patching, etc. | No support for GPU and EBS volumes for persistent storage. You can use [EFS volumes](https://docs.aws.amazon.com/AmazonECS/latest/userguide/using_data_volumes.html) for persistent storage and bind mounts for ephemeral storage. In short, there are less customizations available for customers as compared to ECS EC2. |
| Table 7-1. Amazon ECS Launch Types | | |

ECS seamlessly integrates with other AWS services such as CloudWatch for observability, IAM for security and access control, [AWS Cloud Map](https://aws.amazon.com/cloud-map/) and Application Load Balancer (ALB) to provide service discovery and load balancing capabilities. Cloud Map allows users to register and discover services dynamically, making it easier for containers to communicate with each other. ALB provides load balancing across containers, distributing traffic based on configured rules, helping to achieve high availability and scalability. Here are some additional key benefits of AWS ECS:

*Scalability*

ECS allows you to scale your containerized applications seamlessly based on demand. It automatically adjusts resources to handle traffic spikes efficiently via auto scaling or serverless compute options.

*High Availability*

ECS ensures high availability of your containers by spreading them across multiple Availability Zones within a region. It automatically recovers failed containers and keeps your applications running smoothly.

*Cost-Effective*

With ECS, you only pay for the resources you use. It optimizes resource allocation and scales based on actual demand, helping to reduce costs.

*Security and Compliance*

ECS integrates with AWS Identity and Access Management (IAM), allowing you to control access to your containers and resources. It also provides encryption options for data in transit and at rest.

As mentioned earlier, EKS is another service offered by AWS to launch applications on containers. Let’s explore how it differs from ECS and in which use-cases customers generally prefer to go with EKS instead of ECS.

**Amazon Elastic Kubernetes Service**

EKS is a fully managed Kubernetes service that simplifies the deployment, management, and scaling of containerized applications using Kubernetes on AWS Cloud. AWS EKS eliminates the need for manual setup and configuration of Kubernetes clusters, allowing developers to focus on their applications rather than the underlying infrastructure. EKS is available with EKS EC2 and Fargate launch types, similar to ECS. Now you might wonder, why are there two different containerized services and how do I decide which is best for my use-case? Some of the key benefits of AWS EKS include:

*Kubernetes Managed Control Plane*

Amazon EKS manages the Kubernetes cluster control plane which includes the API server and [etcd persistence database](https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/" \t "_blank). This eliminates the administrative overhead of managing and scaling the control plane from the customer’s plate, ensuring high availability and reliability guarantee from AWS.

*Multi-AZ Support*

Amazon EKS supports running Kubernetes clusters across multiple availability zones, providing enhanced availability and fault tolerance. By distributing cluster resources across different AZs, applications running on EKS can tolerate failures and continue to operate seamlessly. The EKS control plane runs in three AZs with support for automatic detection and replacement of unhealthy nodes.

*Automated Kubernetes Version Upgrades*

EKS automatically handles Kubernetes version upgrades for the control plane, simplifying the process of staying up-to-date with the latest features and security patches. This helps ensure that clusters are running on stable and secure Kubernetes versions without requiring manual intervention.

*Hybrid Deployments*

For highly latency sensitive workloads, EKS on [AWS Outposts](https://docs.aws.amazon.com/outposts/latest/userguide/what-is-outposts.html) can be used to operate applications at on-premise data centers. AWS Outposts extends AWS infrastructure, services and tools to customer’s personal data center so essentially making it feasible to run AWS EKS workloads there.

*Integration with AWS Services*

EKS seamlessly integrates with various AWS services, enabling developers to leverage the extensive ecosystem of AWS resources and capabilities. It integrates with services like Elastic Load Balancing, Amazon VPC, AWS Identity and Access Management (IAM), AWS CloudTrail, and more, providing a unified experience for managing and securing applications.

*Scalability and Elasticity*

With Amazon EKS, scaling Kubernetes clusters to accommodate increased workloads is effortless. It leverages Amazon EC2 instances and Auto Scaling Groups to dynamically scale worker nodes based on demand or we can use Fargate launch as a serverless option. This allows applications to handle traffic spikes and changing resource requirements seamlessly. We can also leverage EC2 spot instances to reduce our cost of EKS clusters or use GPU optimized instances for high performance computing.

*Security and Compliance*

EKS incorporates several security features to protect applications and data. It integrates with AWS Identity and Access Management (IAM) to control access to cluster resources, supports fine-grained network policies using Amazon VPC, and allows encryption of data at rest using AWS Key Management Service (KMS). EKS is also compliant with various industry regulations, such as  [SOC](https://aws.amazon.com/compliance/soc-faqs/), [PCI](https://aws.amazon.com/compliance/pci-dss-level-1-faqs/), [ISO](https://aws.amazon.com/compliance/iso-certified/), [FedRAMP-Moderate](https://aws.amazon.com/compliance/fedramp/" \t "_blank), [IRAP](https://aws.amazon.com/compliance/irap/), [C5](https://aws.amazon.com/compliance/bsi-c5/), [K-ISMS](https://aws.amazon.com/compliance/k-isms/), [ENS High](https://aws.amazon.com/compliance/esquema-nacional-de-seguridad/), [OSPAR](https://aws.amazon.com/compliance/OSPAR/), [HITRUST CSF](https://aws.amazon.com/compliance/hitrust/), and [HIPAA](https://aws.amazon.com/compliance/hipaa-compliance/).

*Monitoring and Logging*

EKS provides integrations with popular monitoring and logging services, including Amazon CloudWatch and AWS CloudTrail. These services enable developers to gain insights into cluster performance, monitor resource utilization, and track API calls and events for auditing and troubleshooting purposes.

*Ecosystem and Community*

As an open-source platform, Kubernetes has a vibrant ecosystem and community. EKS benefits from this ecosystem by providing compatibility with Kubernetes-native tools, frameworks, and operators. Developers can leverage these tools to enhance their application deployment, management, and observability within the EKS environment.

Your love for open-source could be just another reason to choose AWS EKS, where you have complete visibility about the ongoing development and Kubernetes internal architecture. EKS on AWS offers the benefits of both worlds, open-source support, as well as the offloading of management tasks to AWS, along with easier integration with other AWS services.

**Conclusion**

Selecting a compute platform for launching applications could be a difficult choice, so you should clearly lay out all the application requirements or expectations in order to help you to make a wise decision. There is a great philosophy of a ‘two way door’ decision-making process at Amazon which states, ‘you can always revert back your decision if things are not working out as expected’. In a similar way, the compute platform can always be reiterated as application scope or traffic pattern changes for your application. Here are few recommendations we think will be helpful in making your decision about your compute platform:

*Flexibility*

EC2 offers maximum flexibility to users with customizations at the hardware level, in addition to with how software applications are deployed. In contrast, Lambda abstracts the resources completely and customers only have to focus on the code with restrictions such as a maximum 15 minutes invocation time. The high flexibility brings large operational overhead of system maintenance such as OS patching for security issues, figuring out the most efficient instance type for application, configuring auto scaling policies, deploying applications, etc.

*Learning Curve*

As we start out building and launching applications, familiarity with the platform is another factor to consider. More familiarity means less unknowns and quicker application launches. For example, if you’re completely new to any compute platform, it’s very easy to start off with Lambda and launch applications in the minimum time possible as you just have to focus on writing code and skip all headache of application deployment and maintenance for required traffic.

*Traffic Patterns*

All compute platforms offered by AWS can operate at high scale. You should understand requirements in hand and how a specific platform is best for serving the business use-case along with cost efficiency. For example, EC2 or ECS EC2 provides more hardware control so could be a better choice for high latency sensitive operations or GPU accelerated computing.

*Cost*

It’s important to consider the cost we pay for resources in AWS Cloud. The pricing model for each of the compute platforms varies and there is no direct head to head comparison. Cost could be dependent on traffic patterns, size of workload, resource selection such as EC2 instance type or ECS Fargate task size or Lambda memory configuration,  etc. We recommend considering operational overhead as well, because eventually there will be people maintaining the operations.

While you might not know the best solution right off the bat, don’t forget, it’s always a two-way door decision and can always be reiterated. In the next chapter, we’ll be exploring orchestration services offered by AWS Cloud, such as Simple Queue Service (SQS) and AWS Step Functions, along with how application health can be monitored via metrics and logging on AWS Cloudwatch.