COMPUTE SERVICES IN AWS

Compute: Background

In this section we will discuss fundamental concepts related to compute resources, their nature and their evolution throughout the years. Below is a brief overview of the topics to be covered:

- Compute services, and the hardware resources associated with it
- Virtualization, its purpose and financial impact
- Serverless, its advantages and disadvantages

If the reader believes themselves to already be familiar with the listed concepts, then they can feel free to skip this section and move on to the rest of the chapter.

Compute and related infrastructure

The resources that provide the computer system with processing power and are responsible for the running of applications are called compute resources. Whether it be on-prem or on the cloud, compute resources refer to three components:

- CPU (Central Processing Unit): Often called the "brain" of the computer, the CPU performs calculations and processes instructions required by applications. It handles all logical and arithmetic operations, controlling how quickly and efficiently tasks are executed.
- RAM (Random Access Memory): Already discussed in the storage chapter, RAM is the temporary holding space for information related to the programs currently being run on the computer.
- GPU (Graphics Processing Unit): Used to handle complex parallel-computing tasks like matrix and vector operations, heavily utilized in specific use cases such as gaming, cryptographic and AI operations.

Therefore, when we rent compute resources from a cloud provider what we are really renting is the CPU and the RAM, with us generally being charged based on the sophistication of the CPU Chip and the associated memory capacity.

Virtualization

Now, a layman might assume that separate physical servers would be allocated for every prospective renter but that is actually not the way cloud providers facilitate us with compute resources. In reality, cloud providers utilize **virtualization**, a technology that divides the underlying compute hardware into abstract partitions, and creating a **Virtual Machine (VM)** based off it, a virtual computer based on the partitioned resources.

Each VM has its own OS and computing resources, and operates independently of the other VMs, even if they were abstracted from the same underlying hardware enabling multiple users to share share a single physical server while maintaining the illusion of individual dedicated servers. Indeed, virtualization makes the leasing of compute resources a much more profitable venture for the cloud providers than it otherwise would be, allowing cloud providers to maximize the mileage they get out of their physical servers.

Serverless

Now, virtual machines have been around since the 1970s but there has been a major development in the field of managing compute and cloud computing in the last 10-15 years or so: **Serverless computing**.

Serverless allows us to run applications without having to manage any of the infrastructure ourselves, with the cloud providers handling the allocation of resources required by our workload. This allows us to focus only on the development of the application and not worry about operating the servers that it runs on. However, there are benefits to having control over the underlying infrastructure, it allows us to be in complete control of the security and configuration of the resources, a level of control taken away from us in the serverless faustian bargain.

EC2, ECS and Lambda

Compute services in the cloud are used for the provision of processing power/compute capacity required to run a workload. So, the virtual machines that are going to be running your applications on the cloud for example.

AWS has three main compute services, namely: **EC2**, **ECS** and **Lambda**. These are discussed further below:

EC2 (Elastic Compute Cloud): For the purposes of this book, the definition of a Virtual Machine (VM) given in the background, i.e. any software-based emulation of a physical computer that is being run on hardware infrastructure owned by the cloud provider (in our case, AWS), should suffice. VMs allow us to create and manage an isolated and portable computer with its own OS (Operating System) on the servers owned by someone other than ourselves, a “Virtual” machine if you will.

EC2 is the AWS service that allows us to rent virtual machines in the cloud, known as instances, offering a wide range of flexibility. These instances can be optimized for various use cases, including memory, compute, or storage. Said EC2 instances are highly customizable, allowing users to configure CPU, memory, storage, and networking capacity according to their needs. They also come with security features like security groups and IAM roles for access control.

Mentioned features as well as the different types of EC2 instances will be discussed in future sections.

ECS (Elastic Container Service): Containers are the “hot new thing” that’s gaining popularity throughout the IT world due to their cost-effective computations. As monolithic applications continue to be replaced by microservice applications, it’s only a matter of time before container usage becomes more prevalent than instance usage.

Docker, the most widely used containerization technology, defines containers as “a standard unit of software that packages up code and all its dependencies such that the application can run quickly and reliably, even after being transferred from one computing environment to another.” Put simply, containers are standardized, lightweight, and secure pre-packaged solutions that run efficiently no matter what situation they’re thrown into. Also worth mentioning is that these containers are usually, though not exclusively, created using Docker and its related packages.

ECS is a managed container orchestration service designed for running containerized applications. While it supports and is mostly used to run Docker containers, though it has made an active effort not to be limited to just docker containers, with the service accommodating all containers built to the Open Container Initiative (OCI) image format. Users can manage ECS clusters themselves on EC2 instances or opt for AWS Fargate, a serverless option that abstracts away infrastructure management. With Fargate, users focus solely on deploying and managing containers, without dealing with server provisioning.

A more detailed exploration of Elastic Container Service, and its related services and configurations will be discussed in future sections.

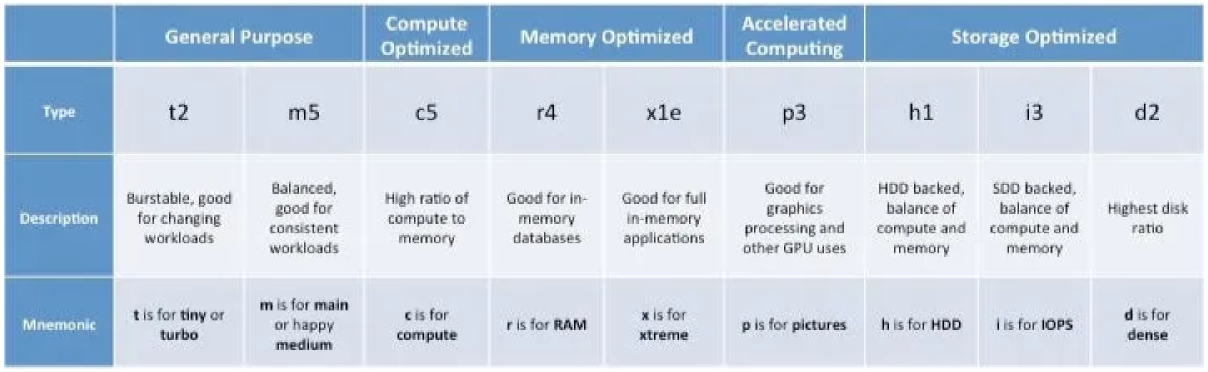
Lambda: Functions are one of the most basic yet powerful concepts in programming, and can be understood as self-contained pieces of code designed to accomplish a certain task. In order to calculate the sum of two numbers for example, a programmer could create a self-contained piece of code called *sumCalc* that would take two numbers, let us call them *num1* and *num2* as arguments, usually represented as: `sumCalc(num1, num2)`

Lambda is a serverless compute service that allows users to run function code such as the one mentioned above without managing servers. Users upload their code, and Lambda handles scaling and infrastructure management automatically. When a Lambda function is invoked, users receive an ARN (Amazon Resource Name) that uniquely identifies the function. Lambda is commonly used with API Gateway to create serverless REST APIs, enabling infinitely scalable endpoints. Lambda prioritizes simplicity, offering a streamlined development experience without the burden of managing infrastructure.

A more detailed exploration of Lambda and its associated configurations will be discussed in future sections.

Types of EC2 Instances

Because EC2 Instances are used to power a wide variety of workloads with similarly varying resource needs, Amazon provides us with different types of EC2 instances. A table showcasing all the different EC2 instance types is given below:



	General Purpose		Compute Optimized	Memory Optimized		Accelerated Computing	Storage Optimized		
Type	t2	m5	c5	r4	x1e	p3	h1	i3	d2
Description	Burstable, good for changing workloads	Balanced, good for consistent workloads	High ratio of compute to memory	Good for in-memory databases	Good for full in-memory applications	Good for graphics processing and other GPU uses	HDD backed, balance of compute and memory	SDD backed, balance of compute and memory	Highest disk ratio
Mnemonic	**t** is for **tiny** or **turbo**	**m** is for **main** or happy **medium**	**c** is for **compute**	**r** is for **RAM**	**x** is for **xtreme**	**p** is for **pictures**	**h** is for **HDD**	**i** is for **IOPS**	**d** is for **dense**

It might seem intimidating at first, but the designation names are there to help you, and allow you to make the best decision in regards to the type of EC2 instance that is best suited for your needs.

Understanding the different EC2 Instance Types

Amazon EC2 instances power a wide variety of workloads, each with unique resource needs. To accommodate these differences, Amazon provides several types of EC2 instances, each designed to optimize for a specific workload type. At first glance, the various EC2 instance designations might seem intimidating, but their naming conventions are designed to help you make the right choice for your specific requirements. Here’s an overview of the key instance types:

General Purpose Instances

General Purpose instances provide a balanced mix of compute, memory, and networking resources, making them versatile for a variety of workloads. These instances are suitable for applications such as web servers, small databases, and development environments.

Example Types:

- **T2**: Burstable instances ideal for applications with low to moderate baseline performance needs that can benefit from burst capability. The mnemonic "t" stands for tiny or turbo, reflecting the burstable performance nature.
- **M5**: Balanced instances for consistent workloads, offering a stable balance between compute, memory, and network resources. The "m" is for main or medium.

Memory Optimized Instances

These instances are designed to deliver fast performance for workloads that process large data sets in memory. They typically have a higher amount of RAM compared to other instance types, making them suitable for memory-intensive applications like in-memory databases, real-time big data analytics, and high-performance computing.

Example Types:

- **R4**: Designed for applications needing a high memory-to-CPU ratio, such as in-memory databases and real-time analytics. The "r" in R4 stands for RAM.
- **X1e**: Optimized for extremely large memory needs, often used for full in-memory applications like SAP HANA. The "x" represents xtreme memory (slightly childish I know, but it is what it is).

Compute Optimized Instances

These instances are optimized for compute-bound applications that require high-performance processors. They typically have a high ratio of vCPUs to

RAM/Memory, making them suitable for applications that require intensive computational processing such as gaming servers, scientific modeling, batch processing, and media transcoding.

Example Type:

- **C5**: These instances provide a high ratio of compute to memory. The "c" stands for compute, making them ideal for tasks that need powerful processors but do not require a lot of memory.

Storage Optimized Instances

These instances are designed to deliver high storage performance for workloads that require high I/O performance. They come with local instance storage optimized for high-speed, low-latency access, making them suitable for applications that require frequent and fast access to large data sets, such as NoSQL databases, distributed file systems, data warehousing, and data processing.

Example Types:

- **H1**: These instances use HDD-backed storage and offer a balance between compute and storage performance. The "h" stands for HDD.
- **I3**: SSD-backed instances, ideal for applications needing high IOPS, such as transactional databases. The "i" stands for IOPS.
- **D2**: Offers the highest disk throughput with dense storage for applications like data warehousing. The "d" stands for dense, representing the storage capacity.

Accelerated Computing Instances

Accelerated Computing instances are designed for applications that benefit from hardware accelerators (such as GPUs). These instances are perfect for computationally expensive applications, such as machine learning, graphics rendering, and gaming.

Example Type:

- **P3**: Ideal for GPU-heavy workloads like graphics processing or deep learning. The "p" stands for pictures, reflecting its role in image and graphics processing.

Choosing the Right Instance

Now, with so many different EC2 instance types one might think that the process of selecting an EC2 instance might be tedious and confusing, but it is actually quite simple once the workload requirements have been decided upon. Are you running a memory-intensive in-memory database? An **R4** instance might be ideal. Need to process large datasets stored locally? Consider a **D2** or **I3** instance. By understanding the key characteristics of each EC2 instance type, you can optimize performance and cost efficiency for your applications. Not to mention, in cases where workload requirements are yet to be identified one can simply start off with a general purpose instance with the intention of shifting to another instance type in the future.

Hibernation and Stop

Sometimes it might not be technically and/or economically prudent for us to have an EC2 instance running 24/7, and we might instead wish to halt an instance at a specific moment in time, with the intention of resuming it in the future as we see fit. AWS provides us with two methods of halting EC2 instances: **Hibernation** and **Stop**. Both methods are discussed in greater detail below:

Stopping an EC2 instance

What we must understand before we differentiate between Hibernation and Stopping an EC2 instance is that all EC2 instances have an EBS (Elastic Block Store) volume attached to them upon creation, with the specifics of the EBS volume being decided by the administrator who performed the creation operation in the first place. Said EBS volumes serve as the default secondary storage device for the created EC2 instances with the files, directories and programs utilized by the EC2 instance stored in it.

Stopping an EC2 instance means temporarily shutting down the virtual machine while preserving its configuration and data on the attached EBS volume. When an instance is stopped, its CPU and RAM are no longer running, but its storage and settings remain intact. The instance goes through a normal shutdown process when stopped, similar to when we turn off our personal computers with all running processes and applications terminated.

Once the instance stops, the account will not be charged for the EC2 compute resources (CPU, RAM) though the account will still be charged for the EBS volumes and any Elastic IP Addresses attached to the instance, that is, until the instance is restarted of course.

Hibernating an EC2 instance

Hibernation is similar to stopping an EC2 instance though it may be best understood as the pausing and resuming of EC2 instances rather than the stopping and restarting of them. When we hibernate an EC2 instance, we cease its operation just like with the stopping of an EC2 instance with the key difference being that the data stored in the primary memory/RAM is retained when an EC2 instance is hibernated, which is not the case with the stopping of an EC2 instance.

This is particularly relevant in environments that are time-sensitive, have long startup times or must retain sticky sessions because it saves the administrator the effort of setting up the environment or applications all over again every time an EC2 instance has to be halted. Instead of having to rebuild the memory footprint from the ground up, hibernation allows applications to pick up exactly where they left off often reducing both the startup time taken by the Operating system as well as the frictions associated with setting up a work environment.

Also worth noting is that just like with the stopping of an EC2 instance, while the instance is in hibernation, you pay only for the EBS volumes and Elastic IP Addresses attached to it; there are no other hourly charges (just like any other stopped instance). However, unlike with stopping which is a function available to all EC2 instances, hibernation is a feature which the user must opt-in to when an EC2 instance is being created, as it is not possible to enable or disable hibernation for an instance after it has been launched.

TLDR;

Stopping: Halts the EC2 instance while retaining the data in the secondary storage; Equivalent to Shut Down / Turn Off options in personal computers.

Hibernation: Halts the EC2 instance while retaining both the data in the secondary storage as well as the data in the memory (RAM). Equivalent to Sleep / Hibernate options in personal computers.

Autoscaling Group and Spanning

Now, EC2 instances are a finite resource capable of handling a very finite amount of traffic and load, with the overburdening of EC2 instances causing performance degradation and bottlenecks at best, and leading to the whole application or program crashing at worst. Autoscaling groups allow us to avoid such scenarios, and ensure that our application can accommodate itself with the right amount of resources, no matter the traffic load.

An Auto Scaling Group is essentially the administrator's best friend when it comes to managing EC2 instances, automatically adjusting the number of instances running at any given time, based on criterion defined by the administrator. Whether your app is facing a sudden spike in traffic or things are slowing down during off-peak hours, an ASG can scale out or scale in to meet the demand, ensuring that your app stays both performant and cost-effective.

At its core, an ASG monitors your application's performance through metrics like CPU usage or network traffic (with a little help from CloudWatch, of course). When traffic increases, the ASG adds more EC2 instances to balance the load. When things quiet down, it reduces the number of instances, saving you money. This automatic scaling makes sure that your app is never under-provisioned or over-provisioned.

Another one of the really cool features of ASGs is the ability to perform **health checks** on your instances. If an instance starts acting up or fails, ASGs detect this and immediately replace it with a healthy one. This keeps your app running smoothly without you having to manually intervene. Plus, ASGs work hand in hand with Elastic Load Balancing (ELB). As new instances are added, ELB distributes incoming traffic evenly across your instances. So, no single instance gets overwhelmed while others sit idle, ensuring that your app performs consistently no matter how many users are hitting it.

But what if you know in advance that you're going to need more resources, say for a big product launch or during holiday shopping peaks? ASGs also allow for **scheduled scaling**, meaning you can set specific times for when additional instances should be added or removed. This proactive approach is especially useful for companies and organizations that have already identified time-periods when the product may be overloaded.

Also worth noting is that ASGs can be fine-tuned with CloudWatch alarms and custom scaling policies. Let's say you want your ASG to add more instances if CPU utilization hits 80%. Or maybe you want it to scale back once CPU usage drops below 50%. These kinds of scaling policies let you control exactly how and when your instances are scaled based on real-time data.

Finally, ASGs can span across multiple Availability Zones (AZs) and is an easy and convenient way for us to ensure high availability. If one AZ is experiencing issues for example, your ASG might automatically distribute the load to instances in other AZs. This cross-zone scaling provides the instances comprising the ASG with resilience against potential failures in a single geographic area.

TLDR;

Auto Scaling Groups (ASGs) automatically adjust the number of EC2 instances based on real-time demand, ensuring your app stays performant without overspending. They monitor health, distribute traffic evenly through Elastic Load Balancing, and can even schedule scaling for predictable traffic spikes. Plus, they span multiple Availability Zones to keep your app highly available.

Benefits of Overprovisioning EC2 instances

Consider a scenario where the administrator is designing an ASG for a system which calls for a minimum of 2 instances operating at all times, then common sense would tell us to create an ASG with a minimum of 2 instances. However, in the real world, you will often find that administrators go a step further and have at least 3 instances running at all times.

This might seem odd for the uninitiated, after all, is not the entire point of autoscaling for us to be saving costs by using only the required amount of resources? So then why would we wish to provision more EC2 instances than absolutely necessary?

This is because if an Availability Zone (AZ) outage occurs, your Auto Scaling Group (ASG) will spin up a new instance in an unaffected AZ to compensate for the lost one. However, this new instance doesn’t come online immediately, so for a short while, you’ll be running with only 1 active instance—leaving your application below the minimum threshold required to operate.

Therefore, over-provisioning by just one instance can be the difference between maintaining your app’s availability and leaving it vulnerable during an AZ outage making it a simple, proactive approach that buys you time during unexpected events, and ensures that your application stays resilient and continues to readily meet user demand.

Reserved Instances and On-Demand Capacity Reservation

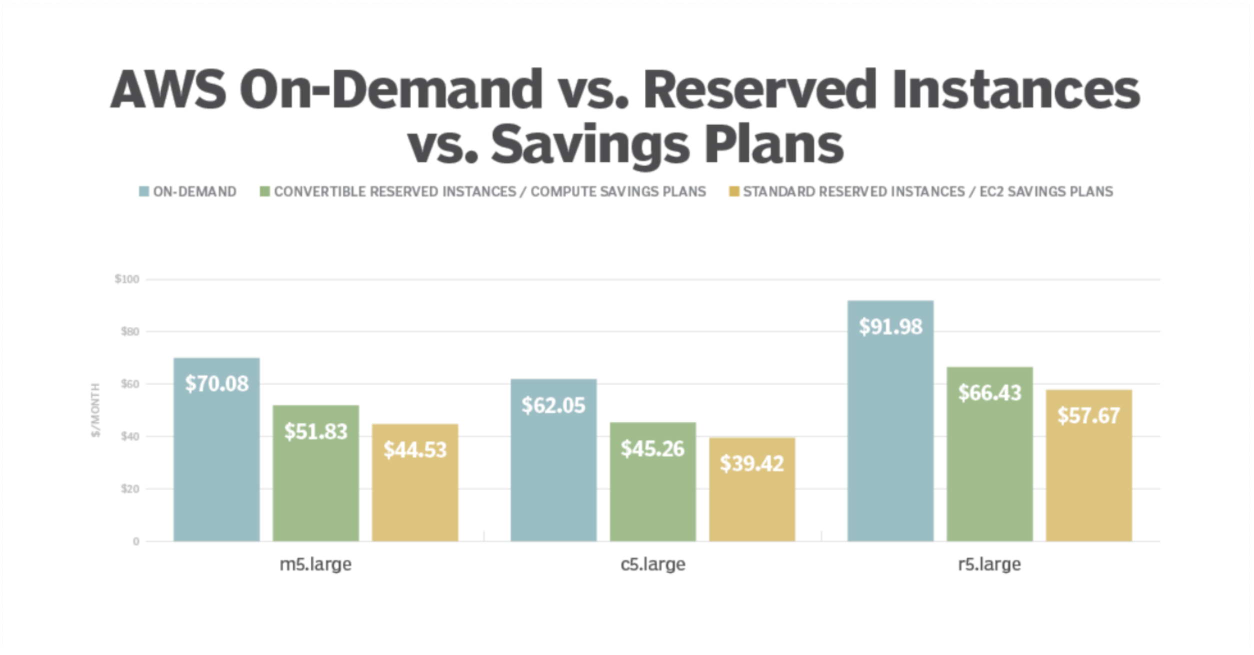
When we use EC2 instances, what we are essentially doing is renting computer infrastructure from AWS. The default renting agreement, i.e. On-Demand EC2 instances provides us with EC2 instances on-the-go, provisioning us with resources according to the load being put on their computers. However, this is not the only renting agreement available to us, and if we are already aware of the compute capacity that is required for our workload then AWS provides us with two other methods of renting EC2 instances: **Reserved Instances** and **On-Demand Capacity Reservation**.

Leveraging these pricing models allow us to get a fixed number of EC2 instances of a specific hardware configuration potentially for a fraction of the price usually extracted when utilizing On-Demand EC2 instances.

Reserved Instances (RIs) in AWS are a cost-efficient way to secure compute capacity for long-term, predictable workloads. By committing to a one- or three-year term, RIs offer significant discounts (up to 75%) compared to On-Demand instance pricing. This pricing model is especially useful for workloads that have steady, consistent demand, such as databases, backend services, or enterprise applications. When you know your infrastructure needs in advance, RIs allow you to plan ahead and reduce overall cloud costs without sacrificing performance or availability.

RIs come in two types: **Standard** and **Convertible**. Standard RIs offer the deepest discounts but require a fixed commitment to a specific instance type, operating system, and tenancy. In contrast, Convertible RIs provide more flexibility by allowing changes to the instance family, OS, or tenancy during the reservation period, albeit with slightly lower savings. Additionally, users can choose different payment options (All Upfront, Partial Upfront, or No Upfront) to balance between immediate cost savings and cash flow management.

While Reserved Instances can drastically reduce costs, it is crucial to remember that they do come with certain caveats which must be taken into consideration. If the workload pattern changes and the user does not fully utilize the reserved capacity for example, they are still obligated by contract to pay for the reservation. Therefore, careful planning is essential while utilizing Reserved Instances to avoid any unnecessary financial commitments.



Source: TechTarget and AWS Pricing Calculator

On-Demand Capacity Reservations offer AWS users the flexibility to reserve EC2 capacity for any fixed duration of time that is comfortable to them, without the need to specify exact instance types or Availability Zones (AZs). This approach provides much greater control over resource management, particularly for workloads requiring precise instance placement and configuration.

Unlike Reserved Instances (RIs), which necessitate a fixed one-year or three-year commitment, On-Demand Capacity Reservations are more suitable for tasks that have known capacity requirements but a shorter duration. If a workload demands reserved capacity for a temporary period, these reservations allow you to secure resources without long-term contracts, making them an efficient alternative to RIs. However, do note that On-Demand Capacity Reservation is simply a guarantee of compute capacity, and does not offer any reduction in cost when compared to normal On-Demand pricing, unlike reserved instances.

TLDR;

AWS provides us with two methods of reserving compute capacity: **Reserved Instances** and **On-Demand Capacity Reservation**.

Reserved instances: Provide the most savings but the contract must be for a 1 or 3 year duration.

On-Demand Capacity Reservation: Provide capacity without any long-term contracts, making them better for more temporary workloads though they are slightly pricier than Reserved instances.

Elastic Container Service (ECS)

Another one of the three aforementioned major compute options made available by AWS, **Elastic Container Service (ECS)** is Amazon's fully managed container orchestration service, designed to simplify the deployment and management of Docker containers. Since we've already discussed containers and Docker in previous sections, I will spare the reader from a repeat explanation.

ECS allows you to run containerized applications in two main ways: on a scalable cluster of EC2 instances, where you control the underlying infrastructure, or using **AWS Fargate**, a serverless compute option that abstracts away infrastructure management, allowing administrators/developers to focus solely on the containers themselves without having to worry about the hardware it is running on.

Especially useful for administrators that wish to avoid the operational overhead of managing the container's compute infrastructure (such as provisioning and scaling of EC2 instances), AWS Fargate is a relatively painless method of operating containerized workloads. However, this does mean relinquishing a great deal of control over the infrastructure used over to AWS. However, Fargate is neither universally desirable, nor universally adoptable, as organizations and workloads that desire a certain amount of control over the infrastructure or need specialized configurations, will always find running ECS on EC2 Instances to be the more appropriate option.

Now, while ECS is an extremely powerful tool for container orchestration, many organizations prefer to use Kubernetes, an open-source container orchestration tool and platform. Kubernetes offers more flexibility and extensibility, with a rich ecosystem of plugins and integrations which has led to the platform becoming somewhat of an industry standard. To meet this demand for Kubernetes workloads, AWS also started offering **Elastic Kubernetes Service (EKS)**, a fully managed Kubernetes service that allows you to run Kubernetes clusters on AWS without the complexity of managing the Kubernetes control plane yourself.

With EKS, much like ECS, you can run Kubernetes workloads on either EC2 instances or AWS Fargate, providing flexibility based on your operational needs. When working with a team or organization which is already experienced with Kubernetes or when working with workloads that require Kubernetes-specific features and integrations, using EKS over ECS might be the right choice.

TLDR;

AWS provides two main services for managing containers: **Elastic Container Service (ECS)** and **Elastic Kubernetes Service (EKS)**.

Both services simplify the deploying and managing of Docker containers, with the option to run workloads on EC2 instances (which offers more control and flexibility) or AWS Fargate.

An extension on the ECS and EKS offerings, **AWS Fargate** hands over the responsibility of managing the infrastructure required to operate the containers, and is ideal for teams avoiding the operational overhead associated with such tasks.

Lambda

The final one of the aforementioned major compute services, AWS Lambda is a FaaS (Functions as a Service) offering provided by AWS, and allows us to run small self-contained pieces of code without having to provision and manage hardware infrastructure yourself. Usually marketed as both infinitely scalable and incredibly cheap, Lambda is particularly useful for minute tasks that need to be run on-demand in an irregular fashion, where renting a server or EC2 instance 24/7 may not be desirable.

It is also worth noting that whilst AWS Lambda is a serverless offering, we do have to specify the memory capacity allocated for every invoked lambda function (usually chosen during the creation stage of the lambda function). By default, all Lambda functions have 128 MB of memory allocated to them though we can increase that value up to 10 GB.

Though we usually deliver our function code to Lambda through packaged zip files, Lambda is also capable of running containerized images, which we can either upload directly or pull from **Amazon ECR (Elastic Container Registry)**, an AWS service designed to store and manage container images. The ability to run containerized images is especially useful in deployments which require external dependencies to be installed on the system (For example, imagine running python code in a system where python was not even installed to begin with).

ENV variables for Lambda functions

All modern computer operating systems utilize a set of key-value pairs called **Environment Variables** to store configuration data that applications may require access to during their runtime, allowing developers to set certain settings and values without embedding them directly into the code. This is especially useful when managing sensitive information such as database credentials, API keys, or authentication tokens whereby instead of hard-coding these details into your application, we can simply store them as environment variables that your application can reference during execution.

Though we do not have control over the hardware infrastructure using which Amazon in its infinite wisdom (*sarcastic*) decides our Lambda function will be run, Amazon does provide us with the ability to set the environment variables which might be referenced by our function.

Do remember however that when we create or update Lambda functions that use environment variables, AWS Lambda encrypts them using the AWS Key Management Service (often shortened to KMS, will be discussed deeply in later sections) with the intention of decrypting them and making said values available to the Lambda code whenever the function is invoked.