**EC2 Architecture:**

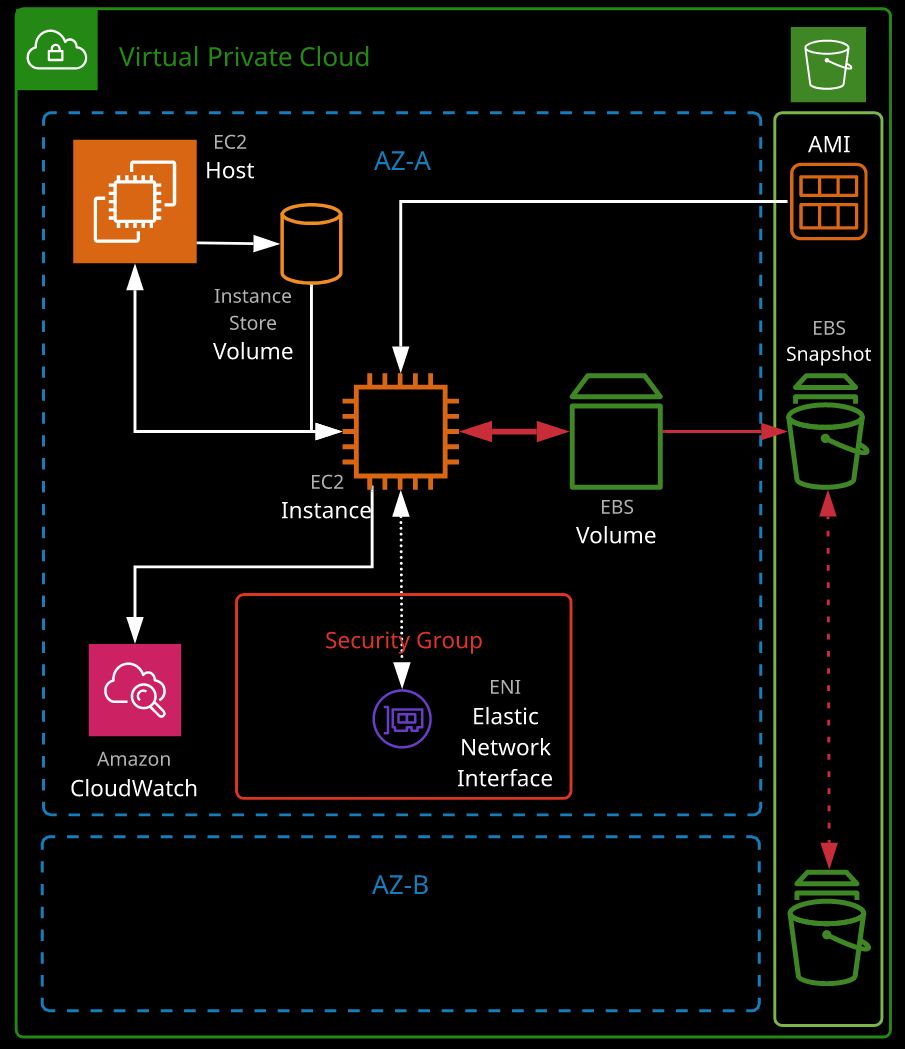
EC2 provides access to virtual servers known as EC2 instances running inside AWS with access to other AWS products and services.

**Regional Service:**

EC2 is a **regional service**. This means any EC2 resource that is created within a region is isolated to that region.

**Instances:**

**EC2 instances** are virtual machines (VM), which are operating systems which are running on EC2 hosts. **EC2 hosts** is hardware that's shared between AWS customers. The advanced virtualization technology that AWS use, enables customer to create an instance in EC2 that uses a slice of these EC2 hosts.

****So, a slice of their physical hardware and how much of a slice depends on the type and size of the instance

**VPC:**

EC2 instances do run inside a networking environment in AWS called VPC, or virtual private cloud. So, while creating an instance we need to make sure ensure that it has a network (VPC) to operate.

**Amazon Machine Image (AMI):**

After clicking on "Launch Instance", the first thing that is needed to do when creating an EC2 instance is to select the AMI.

AMI is the installation media for the EC2 instance. It contains all of the information required to build your EC2 instance, so to install the operating system on the system or the root volume. It like selecting the installation of the OS that you want to use.

The **AWS Marketplace** has lots of **commercial AMIs**.

While selecting the AMI, we can specify the CPU architecture: ARM (ARM CPUs are generally lower power and more suitable for certain types of workload) or the X86(Intel CPU).

**Type of Instance:**

The size and type that we select influences the **amount of virtual CPUs** & **memory**, as well as the **speeds of the storage** and **networking** and other capabilities, so we need to be careful so that the type & size matches your requirement.

**Networking:**

On clicking on Next "Configure Instance Details" we select various important instance options, the first two of which relate to the networking.

Every **EC2 instance** belongs to a **specific VPC**, and it lives in a **specific subnet** in a **specific availability zone** of that **VPC** so you need to select both of those, so we get to select those here.

**Storage:**

We get the option to specify the Storage by Clicking on “Add Storage. Two different types of storage can be used:

* **Instance Store Volumes:**
  + Instance store volumes are physical storage devices that are attached to the EC2 host that an instance runs on.
  + With certain types of instances, we get to add these instance store volumes.
  + (-) Instance store volumes are directly physically attached to a single EC2 host and so if the device fails, then you will lose any data on that instance store volume.
  + (-) There's no resilience.
  + (-) Additionally, if the EC2 instance has to move between physical hosts either because the host fails or that you do a stop and start of the EC2 instance which moves it to a new host, then you lose all of the data on the instance store volume because you'll be moving to a new host with new volumes.
  + (-) Instance store volumes are not persistent.
  + (-) The data is not secure. It could be lost at any time and so they're general used for temporary high-performance requirements.
  + (+) Instance store volumes tend to be the highest performance of any storage type, and again, I'll be covering that in detail in a future lesson in this topic
* **Elastic Block Storage:**
  + This is the other type of volume available are EBS volumes or elastic block store. These are provided by another service called EBS.
  + You create the volume in EBS and attach it to an EC2 instance, and that works over the network.
  + EBS is capable of providing lots of different types of volume using different types of drive to provide those volumes.
  + Based on the AMI selected, it will provide a default and this default is 8 GB of general purpose SSD storage, or gp2 EBS Volume.
  + There are lots of different ways you can change this.

**Tags:**

Tags are a way that you can apply key value pairs to AWS resources. A fairly common usage for EC2 instances is to add a key value pair where the key is name and then the value is the name of the instance.

**Security Group:**

We get to configure the Security Groups by clicking on Configure Security Group. Security groups are like a **virtual firewall and they allow/deny traffic to EC2 Instances**.

**Now, security groups are presented as being attached to an EC2 instance. That's not strictly true.**

* When you create an EC2 instance, you're creating a default ENI or elastic network interface, essentially, the instance's network card.
* The security group is attached to this network interface and because it's the default network interface it's presented as the security group is attached to the instance.

**Now, by default, the security group that you create along with the instance will allow the default protocol.**

* So, if it's a Linux instance, it'll allow SSH which is port 22 from anywhere.
* If it’s a Windows instance, it'll allow RDP.

It's recommended to change the security group name and description to something a little bit more descriptive.

**Launch:**

We can launch the instance by clicking on "Launch".

We have the option of choosing an **existing key pair** or a **new key pair** while launching the instance, which will be later used to connect to the instance.

**Key Pair:**

This is a pair of keys a public part and a private part and these will be used to authenticate to the EC2 instance.

* **Public Part:** The public part is stored by AWS and installed on any EC2 instances when we choose to use this key.
* **Private Part:** The private part the PEM part is downloaded to the local machine, from where the key was generated and it's what you'll use to connect to the instance. This private part of the key needs to be stored safely as loosing this would mean you'd no longer be able to use the key pair to connect to the instances. Then you'd need to remove the key, recreate it, and then obviously handle any existing instances which use the old key pair.

**CloudWatch Integration:**

We have the ability to monitor an EC2 instance from CloudWatch.

On the **monitoring tab** for a specific instance and get access to **CPU utilization, disk read and writes, network in and out**, and other **instance specific telemetry**.

We **don't get access to data** that's **inside the instance**. Example-**memory utilization**. For this we need to **install an agent inside the EC2 instance and connect it to CloudWatch**.

* **Standard Monitoring:** EC2 delivers this data to CloudWatch with a **5 minute** granularity.
* **Detailed Monitoring:** We can enable detailed monitoring to improve this to **1 minute** granularity. This comes at an extra cost and you generally do that for any EC2 instances involved in **production platforms**.

**Account Limits:**

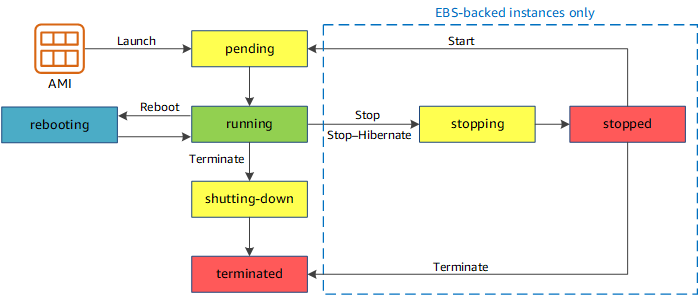
Every AWS account comes with limits and limits control the number of things that you can launch, this size of things that you can launch, and other capabilities that you're allowed to use. This is just a way the AWS can control the likelihood of bill shock or any or the large expense for new users. However most of these could be **increased by logging a support ticket**.

**Status Checks:**

Status checks relates to the **hardware that you EC2 instance runs on** and the other **relates to the actual instance** itself. So we need to check the status check for 2/2, which indicates that this instance is completely healthy.

**Cost Factors for States & Storage:**

Now, EC2 instances have two main states running & stopped. It also has intermediate ones, such as pending and stopping and, of course, you can also terminate an instance which deletes it.

Now by default, EC2 usage is billed by the second with a minimum of 60 seconds and this charge occurs whenever the instance is in the running state. If you stop the instance or terminate the instance, you aren't billed this per second cost.

* **Stopping an Instance: Stopping the instance** is like shutting it down, which means **stopping the virtual hardware**, this is **different** than shutting down from the guest OS. If you shut down from the guest operating system all you're doing is stopping the guest operating system.

When the instance is in a **stopped state**, you're **not being billed** that per second charge but if you **shut it down from the OS** and that does not actually stop the instance, you still will be getting **billed**. That's critical to understand.

* **Storage:** Another **cost factor** of EC2 is the **storage** that the instance uses. If an instance has attached EBS Volume, even when the instance is **stopped**, we are **still being charged for the space consumed on this EBS volume**. The only way we change get way from this charge is to **terminate the instance** which will delete the instance and the attached volume.

However we have option to configure the option to not delete the EBS volume while terminating instances. But generally you want to be in a position where, when you do terminate an instance, it also removes the storage. But unless you configured backups or any other snapshot technology, you will be losing the data.

* **Hibernate:** We are able to stop an instance by right clicking, going to instance state, and then stop hibernate. That **hibernate mode** is like the feature that's probably available on your laptop or desktop. It essentially **saves the memory of the instance to disk** and then it **stops it** and that means you can start it up again and have it be in the same state that it was before you stopped it. There are limitations, so certain types of instances support it and certain types don't and there are restrictions based on the memory size of the instance and the OS type.

|  |  |  |
| --- | --- | --- |
| **Instance state** | **Description** | **Instance usage billing** |
| pending | The instance is preparing to enter the running state. An instance enters the pending state when it launches for the first time, or when it is started after being in the stopped state. | Not billed |
| running | The instance is running and ready for use. | Billed |
| stopping | The instance is preparing to be stopped or stop-hibernated. | Not billed if preparing to stop Billed if preparing to hibernate |
|  |
|  |
| stopped | The instance is shut down and cannot be used. The instance can be started at any time. | Not billed |  |
| shutting-down | The instance is preparing to be terminated. | Not billed |  |
| terminated | The instance has been permanently deleted and cannot be started. | Not billed.  Note: Reserved Instances that applied to terminated instances are billed until the end of their term according to their payment option. |  |
|  |
|  |
|  |

**EC2 Use Cases:**

* Monolithic applications
* Consistent, long-running compute scenarios
* Applications that require full OS/runtime installations
* Services, endpoints, and/or applications that require high availability
* An example of this might be an **active directory**, **authentication service**, or business **network active directory**.

During off periods the CPU usage might be naught 0.1% or even less, but it does need to be active 24/7 even if that's only one instance out of, say, 100.

If you need an active, consistent compute service, then you should pick EC2.

**Instance Types:**

EC2 instances are grouped into **families**, which are designed for a **specific broad type workload**. The type determines a certain set of features, and sizes decide the **level of workload they can cope** with.

The current EC2 families are:

**General purpose:**

General purpose instances provide a balance of compute, memory and networking resources, and can be used for a variety of diverse workloads.

* **Types:** A1, T3, T3a, T2, M6g, M5, M5a, M5n, M4
* **M Series:** The M5 instance is the **general workhorse** in AWS. It's in the general purpose family, and it provides a good balance between CPU, memory, and capabilities for the price. It can be used as default for everything. The reason to default to an M is that it gives you 100% of the resources 24/7 365.

So if you want to use an application on this type of instance, where you're going to be using 100 CPU 24/7 365 that's fine.

* **T Series:** The T3 is a similar type of instance to the M5 but with T3 there's an **assumption that you won't be using its full capabilities.**  T3 has the concept of baseline performance which means if you pick a T3.medium, it's assumed you're going to be using 20% of each of the CPUs. Now if you're using lower than that, you gain CPU credits. If you consume higher than that, say you go to 100%, you allowed to do that but you consume credits. Now if the CPU credit balance goes to zero, then depending on what you select, you could be throttled.

Using a **T3 it's a lot cheaper than an M5**, but there is this **expectation of not using above a certain CPU level**.

* **Use Cases:** These instances are ideal for applications that use these resources in equal proportions such as **web servers** and **code repositories**.

**Compute optimized:**

Compute Optimized instances are ideal for **compute bound applications** that benefit from high performance processors.

* **Types:** C6g, C5, C5a, C5n, C4
* **C Series:** With C Series instances you will be able to select from advanced CPU related technologies. You get **turbo boost**; you might be able to **control CPU states**. These are powered by the **AWS Nitro System**, a combination of **dedicated hardware and lightweight hypervisor**.
* **Use Cases:** Instances belonging to this family are well suited for **batch processing workloads**, **media transcoding**, **high performance web servers**, **high performance computing (HPC), scientific modeling, dedicated gaming servers** and ad **server engines, machine learning inference** and other compute intensive applications.

**Memory optimized:**

Memory optimized instances are designed to deliver fast performance for workloads that **process large data sets in memory**.

* **Types:** R6g, R5, R5a, R5n, R4, X1e, X1, High Memory(u-6tb1.metal, u-9tb1.metal, u-12tb1.metal,u-18tb1.metal, u-24tb1.metal), z1d
* **Use Cases:** These are ideal when you have large memory uses, like **database server** or **in memory caching**.

**Accelerated computing:**

Accelerated computing is a special one with this family, we can **access specific hardware capability**.

* **FPGAs:** This comes in the form of **FPGAs** which are **field programmable gate arrays** with FPGAs you essentially get the ability to **program hardware**. So you can get hardware that functions really well for one specific type of task and by utilizing FPGAs you get to **tune the hardware specifically for a given situation**.
* **GPUs:** You've also got **GPU capability** on some of the accelerated computing instance types, and they provide CPUs or graphical processing units and these are really good for certain types of **3D modeling, machine or deep learning, high performance computing**.
* **Instance Type:** P3, P2, Inf1, G4, G3, F1
* **Use Cases:** Accelerated computing instances use hardware accelerators, or co-processors, to perform functions, such as **floating point number calculations, graphics processing, or data pattern matching**, more efficiently than is possible in software running on CPUs.

**Storage optimized:**

Storage optimized instances are designed for **workloads that require high, sequential read and write access to very large data sets on local storage**. They are optimized to deliver tens of thousands of low-latency, random I/O operations per second (IOPS) to applications.

**Instance Types:** I3, I3en, D2, H1

**Use Case:** So this is ideal for things like **noSQL databases, data warehousing, Elasticsearch, or analytic workloads where they need superfast local storage**.

**Instance types include:**

• T2 and T3: Low-cost instance types that provide burst capability (General Purpose)

• M5: For general workloads (General Purpose)

• C4: Provides more capable CPU (Compute Optimized)

• X1 and R4: Optimize large amounts of fast memory (Memory Optimized)

• I3 Delivers fast IO (Storage Optimized)

• P2, G3, and Fl: Deliver GPU and FPGAs (Accelerated Programing)

**Instance sizes**:

Include **nano, micro, small, medium, large, x.large, 2x.large, and larger**.

**Special Cases:**

• **"a"**: Use AMD CPUs

• **"A"**: Arm based

• **"n"**: Higher speed networking

• **"d"**: NVMe storage (Instance Store Volumes)

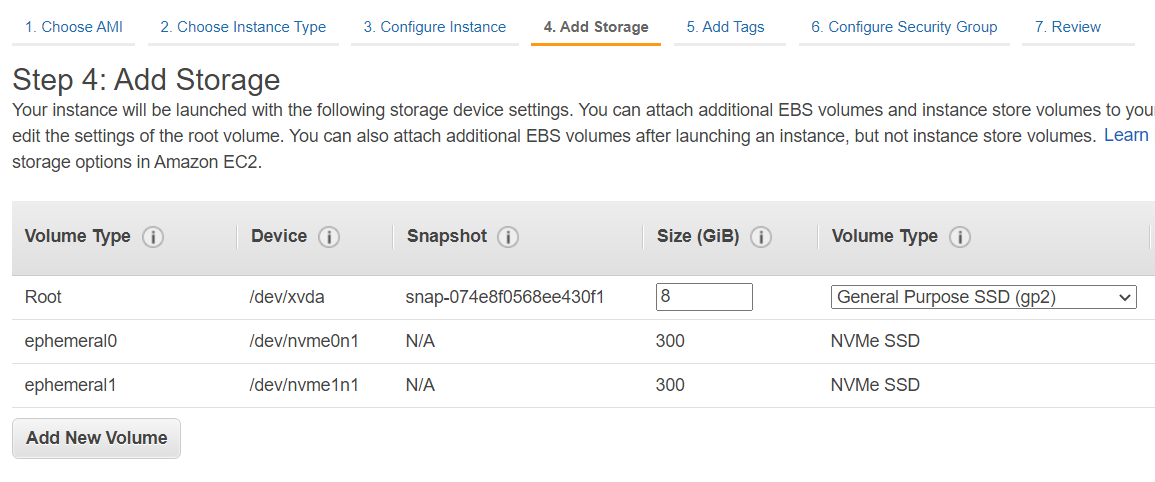
**EC2 Storage:**

Two different forms of storage: **instance store volumes** which are the physical storage devices are attached to an EC2 host and available for the instances and **EBS or elastic block store**, which provides block storage to EC2.

**Block Storage:** Now, block storage is a type of storage which virtual machines can use as attached drives. So boot volumes and data volumes. For example: Storage provided by SAN or NAS products are block storage.

**EC2 Instance Store Volumes:**

* An instance store provides **temporary block-level storage** for your instance.
* This storage is located on **disks that are physically attached to the host computer**.
* Instance store volumes offer some of the **best storage performance** available to EC2 because they're **directly connected to the EC2 host** rather than being accessed over the network.
* But that speed comes with tradeoffs because it's **attached to the host,** it means that if the **host fails or changes**, the **storage is lost**.
* Instance store volumes should be regarded as **temporary**. They're **not resilient**, and they're **not persistent**.
* Only **certain instance types** of instances come with instance store volumes, a common example being the storage optimized instance types. They're **included with the price of those instance**.
* An **instance store** consists of **one or more instance store volumes exposed as block devices**. The **size of an instance store** as well as the **number of devices available varies by instance type**.
* For Instance Store Volumes “**volume type”** is listed as “**ephemeral**[0-23]”.
* Each of the volumes is mapped inside the operating system, using a different device ID. This is known as the **block device mapping**. So for each attached volume, whether it's instance store or EBS, it's going to be mapped to a device ID inside the guest operating system.



**EC2 Instance Store Volumes Use Cases:**

Instance store is ideal for **temporary storage of information** that **changes frequently**, such as **buffers, caches, scratch data, and other temporary content**, or for **data that is replicated across a fleet of instances**, such as a **load-balanced pool of web servers**.

**Instance Store Volumes are Temporary Storage:**

For **OS level reboot** or **reboot from an instance perspective**, EC2 **instance stays on the same EC2 host**, so the instance store volume is maintained, so **data is not lost**.

However if we **stop the instance**, that would shout down the guest OS and the **Virtual Machine H/W is deallocated** from EC2 host. When I started up again, the instance going to be running on a separate EC2 host and that means new ephemeral or instance store volumes, so the data in the **instance store volume is lost**.

This is one of the downsides of using instance store volumes because they're associated with an individual EC2 host if for whatever reason that instance moves between host or if these volumes fail, then you lose access to those original volumes and you get new ones allocated on a new host.

When it starts again, it's going to have a new public IP address and a new DNS name.

**Exam Tips:**

Keep that in mind for certain performance levels you need instance store volumes, but you need to be aware that for any data that you can't reproduce so any permanent data, anything beyond temporary storage or cached volumes or temporary databases, it's not really suited to instance store volumes.

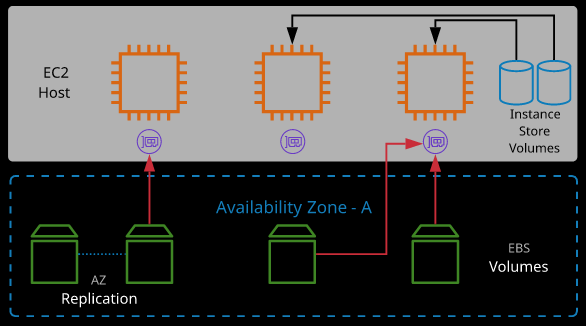
**Elastic Block Store (EBS):**

Elastic Block Store (EBS) is a storage service that creates and manages volumes based on four underlying storage types. This is the **network block storage** product available within AWS.

So that's **storage presented over the network** to EC2 instances.

Volumes are **persistent**, can be **attached and removed** from EC2 instances, and are **replicated within a single AZ**.

**EBS Availability Zone Based Service:**

* EBS is an **availability zone based service**, and you can only **attach volumes to instances which are both in the same availability zone**.
* EBS does **replicate data inside the availability zones**.
* So in this example, we've got availability zone A. We do have replication of the volume between different physical storage appliances but if the availability zone fails, you might very well lose the data on the EBS volume.
* Best case, it's going to be offline for a while the AZ is restored.

\***To protect against AZ failure**, EBS snapshots (**to S3**) can be used. Since this uses S3, data is replicated across multiple AZs in the region and (optionally) internationally.

**Encryption:**

Now you can encrypt a volume, and if you do, you'll need to pick an encryption key to use, which is provided by KMS. When you create an encrypted EBS volume and attach it to a supported instance type, the following types of data are encrypted:

* **Data at rest** inside the volume
* All data moving between the volume and the instance
* All snapshots created from the volume
* All volumes created from those snapshots

**Performance:**

Storage performance is actually measured in two main ways.

* **IOPS:** IOPS is the number of input output operations the volume can cope within a given second.
* **Throughput:** Throughput is the data rate expressed in megabytes per second and this is based on the IOPS so the number of operations and then the block size of each operation.
* **For example**, a **block size** of **256 KB** and then **IOPS** of **400** gives a **100 MiB/s** **throughput**.

**Throughput = Block Size \* IOPS**

**Dominant Performance Attribute:**

Now each of the volume types has what's known as a **dominant performance attribute**. If it's IOPS then logically, the volume type focuses on high IOPS, and that is what both of the SSD based volumes do.

* For **General Purpose SSD (gp2) & Provisioned IOPS SSD (io1)** it is **IOPS**.
* For **Throughput Optimized HDD (st1) & Cold HDD (sc1)** is **Throughput**.

If the dominant performance attributes is throughput, then you know that the IOPS will be suboptimal and the throughput will be the focus of the volume types.

**Volume Types:**

* **Mechanical Hard disk drives (HDD):** 
  + **Throughput Optimized HDD (st1):** Low cost, throughput intensive, can't be a boot volume
  + **Cold HDD (sc1):** Lowest cost, infrequent access, can't be boot volume
* **Solid-state drives (SSD):** 
  + **General Purpose SSD (gp2):** Default, balance of IOPS/MiB/s - burst pool IOPS per GB
  + **Provisioned IOPS SSD (io1):** Highest performance, can adjust size and IOPS separately

**Exam Facts and Figures: EBS**

• EBS supports a maximum per-instance throughput of **1,750 MiB/s** and **80,000 IOPS**. If you need more, you need to use instance store volumes.

**General Purpose SSD (gp2):**

* (SSD) Default for most workloads
* **3 IOPS per GiB** (100 IOPS - 16,000 IOPS)
* Bursts up to **3,000 IOPS (credit based)**
* **1 GiB - 16 TiB** size, max throughput p/vol of **250 MiB/s**
* **Use Cases:**
  + General purpose SSD volume that balances price and performance for a wide variety of workloads.
  + Default Balance of IOPS(s) per GB
  + Recommended for most workloads
  + System boot volumes
  + Virtual desktops
  + Low-latency interactive apps
  + Development and test environments

gp2 volumes have two interesting characteristics.

Performance is linked to their size. With gp2 you actually get 3 IOPS of performance for every GB of volume size. It comes with a minimum level of 100 IOPS and a maximum of 16,000 IOPS. Now this is good and bad it means that you can control the performance of the volume based on the size but it does mean for smaller volumes you might hit performance ceilings.

gp2 volumes come with what's known as a burst pool. With burst pool you get 5.4 million IO credits to start with. Any time you need IOPS above your baseline which remember, is based on the size of the volume you consume IOPS from your burst pool. Any time you're using less IOPS then your baseline, your pool replenishes. It helps give you some flexibility because even for smaller volumes, you've got this buffer that you can use to burst up to higher amounts.

Even though the baseline performance is relatively low, remember though for the burst pool, the maximum is always 3000.

So for the smaller volumes, you're always going to hit this 3000 IOPS ceiling.

Now the maximum volume size for a gp2 volume is 16384 GB and size and note if I change it to that then my IOPS is 16,000. Once you go above one TB in size then the concept of this burst pool vanishes, and you always get the baseline performance. So for any volumes above one TB, you're always going to get a baseline IOPS performance of three times the size.

Now gp2 is the default storage type because of this flexibility, if you don't have a reason to pick anything else then don't.

**Provisioned IOPS SSD (io1):**

* Used for applications that require sustained IOPS performance
* Large database workloads
* Volume size of **4 GiB - 16 TiB** up to **64,000 IOPS** per volume(Maximum ratio of 50:1 is permitted between IOPS and volume size.)
* Max throughput **p/vol** of **1,000 MiB/s**
* Maximum ratio **of 50:1** is permitted between **IOPS and volume size.**
* **Volumes with greater than 32000 IOPS** must be attached to a **Nitro based instance** to achieve provisioned performance.
* **Use cases:**
  + Highest-performance SSD volume for mission-critical low-latency or high-throughput workloads.
  + Can adjust size and IOPS separately
  + Critical business applications that require sustained IOPS performance, or more than 16,000 IOPS or 250 MiB/s of throughput per volume
  + Large database workloads, such as: MongoDB, Cassandra, Microsoft SQL Server, MySQL, PostgreSQL, Oracle

io1 is the final volume type available with EBS, and it's the most interesting it's known as provisioned IOPS.

With io1, you can set an IOPS level on a volume size independently. Which means you're paying for the size of the volume, as well as paying for the performance that you set on it.

**Throughput Optimized HDD (st1):**

* Low storage cost
* Used for frequently accessed, throughput-intensive workloads (streaming, big data)
* Cannot be a boot volume
* Volume size of 500 GiB - 16 TiB
* Per-volume max throughput of 500 MiB/s and IOPS 500
* **Use Case:**
  + Low-cost HDD volume designed for frequently accessed, throughput-intensive workloads
  + Streaming workloads requiring consistent, fast throughput at a low price
  + Big data
  + Data warehouses
  + Log processing
  + Cannot be a boot volume

**Cold HDD (sc1): (HDD)**

* Lowest cost
* Infrequently accessed data
* Cannot be a boot volume
* Volume size of 500 GiB - 16 TiB
* Per-volume max throughput of 250 MiB/s and 250 IOPS
* Use Case:
  + Lowest cost HDD volume designed for less frequently accessed workloads
  + Throughput-oriented storage for large volumes of data that is infrequently accessed
  + Scenarios where the lowest storage cost is important
  + Cannot be a boot volume

**Exam Tips:**

\*\*The first is that there is a maximum performance available on any one single EBS volume of 64,000 IOPS and that's only available on the largest high performance instances that are based on Nitro.

Anything else generally tops out at 32,000. Instances themselves have a maxim IOPS of 80,000 when using EBS.

So keep those figures in mind 64,000 for a volume and 80,000 for the instance.

If you want anything above that, then you need to use **instance store volumes** and remember, they're not persistent. So if the instance moves between hosts or you stop and start it for any other reason or there's a hardware failure, you can lose the data.

So you might face exam questions where you need to determine what type of volume to use.

These key values will help you understand that **anything above 80,000** and **it needs to be instance store volume** and then for that, you need to make sure that there is **not any limitation on the fact that you can't afford data loss**.

Most **modern instance types** support what are known as **elastic volume, so you're able to change the volume type and the size without causing instance downtime** but if you're using an older instance, type which doesn't support elastic volumes, you will need to shut down the instance to make this change.

You can create volumes attached them to instances, but remember, the **volumes are in a particular availability zone**, and if you want to **move them between availability zones or region**s you need to use **snapshots.**

**EBS snapshots:**

EBS snapshots are a **point-in-time backup of an EBS volume stored in S3**, which makes snapshots more resilient since data in S3 is replicated across availability zones in the region.

The **initial snapshot** is a **full copy of the volume**. **Future snapshots** only store the **data changed since the last snapshot**.

**Snapshots are known as Crash Consistent:** When taking the snapshot, the OS and applications running on the instance aren't aware that we are taking a snapshot, so the data that you're actually storing would be just like if you switched off a physical machine right now and the application may be left with data that could potentially be in an inconsistent state.

To perform a **truly consistent snapshot** the data needs to be in a good state and to do that we can:

* **Flush any in memory caches to disk:** Flush any in memory caches to disk so that ensures any applications that are storing any data that hasn't been written to desk write that data to disk.
* **Shutdown the instance :** Shutdown the instance, which would shut down the guest OS (write out any data that it's got in in memory caching to disk) and stop the virtual hardware. Once the instance is "Stopped" perform a Snapshot.

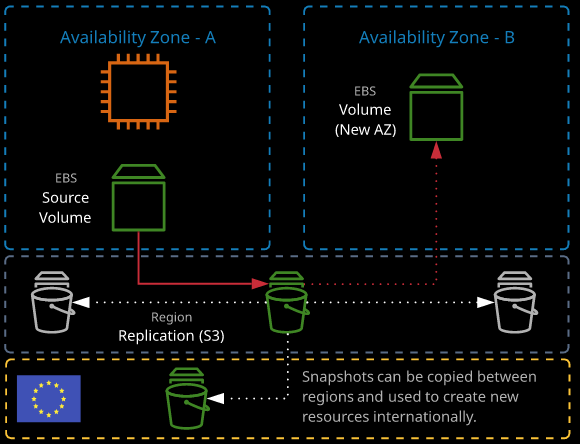
So, When creating a snapshot of the root/boot volume of an instance or busy volume, it's recommended the instance is powered off, or disks are "flushed."

Snapshots can be **used to create new volumes** and are a great way to **move or copy instances between AZs**.

**DLM:**

**Lifecycle Manager or Data Lifecycle Manager(DLM)** is an AWS Product, which allows you to **automate** the process of **creating and deleting snapshots**.

**Snapshots** can be **copied between regions**, **shared**, and **automated** using **Data Lifecycle Manager (DLM)**.

Since Snapshots are stored inside S3 they are **cheaper than the equivalent EBS storage volume.**

**Incremental snapshots:**

Also we are **storing** the **data that is different between snapshots**, which means snapshots are taken incrementally.

So if I do 100 snapshots of this 8GB volume that I'm not going to be storing 100 times 8 GB If the data change on this volume is relatively low, and I'm only going to be storing the original 8 GB and then potentially tiny components inside each snapshot.


        Snapshots capturing an initial volume state and two subsequent states after data has
          been changed.
      EBS Snapshots allows deletion of incremental snapshots by retaining **any of the data** that's used in **future versions of that snapshot**.

**Use Case:**

* **Lower RPO:** Snapshots offer a really cost effective and efficient way to achieve backups in AWS. Because of the way snapshots work, we could, in theory, do a snapshot every single minute that the system is live and by doing that, because you're only storing the difference in data, which great RPO without significant cost.

So straightaway we can see how AWS is supporting your ability to have really cutting edge RTO and RPO values without incurring significant expense. So that's a brilliant use case for snapshots.

* **Resilient:** EBS snapshots are a point-in-time backup of an EBS volume stored in S3, which makes snapshots more resilient since data in S3 is replicated across availability zones in the region.
* **Moving EBS Volume data across AZ and regions:** You can have EBS volumes attached to the instances, but remember, the volumes are in a particular availability zone, and if you want to move them between availability zones or regions you need to use snapshots “Copy” feature.

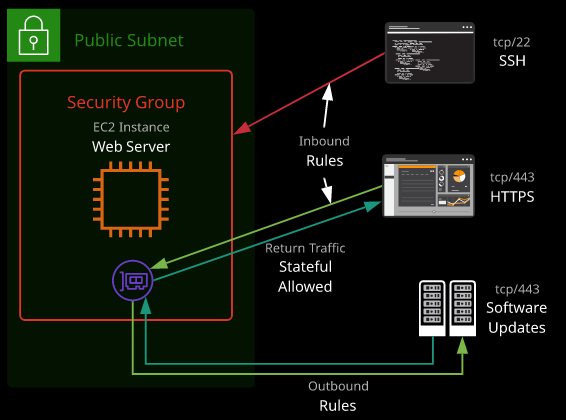
**Security Groups:**

**Security groups** are **software firewalls** that can be **attached to network interfaces** and (by association) products in AWS.

Security groups each have **inbound rules** and **outbound rules**.

A **rule allows traffic to or from a source** (IP, network, named AWS entity) and protocol.

**Implicit Deny:** By default, there is this **hidden deny rule and that applies to both the inbound and the outbound**. So all of the security groups that are associated with, in this case, an EC2 instance, they're all evaluated at the same time. All the rules or the inbound and outbound rules are all merged together, and they're evaluated at the same time, and only if none of the rules apply, this default or implicit deny take effect.

Security groups have a **hidden implicit/default deny rule** but **cannot explicitly deny traffic**.

**Inability to explicit deny Traffic:** With Security Groups, you have this implicit or hidden deny, and you're allowed to allow traffic through but you are **not allowed to explicitly specify a deny rule for traffic**.

* Now the inability to add a deny rule becomes especially important for large enterprise.
* **For example**- There is a website for which we have given access to a specific ip address range. But suppose somebody within that IP Address range with an infected laptop or somebody who was trying to cause damage to website. With security groups, we couldn't add an explicit deny for that single IP address. So in situations where you add entire IP address ranges you can't add an explicit deny for a subset of that range. It's quite crucial limitation.
* So it means that wherever possible, you need to add **allow rules for very tight groups of IPs**.
* If you want to be able to block specific IPs you **can't explicitly deny them**. You have **just to not allow them** and if you allow an entire range then by definition, you allow everyone in that range. So when you're thinking about security groups, you need to be very careful when you're opening up these allow IP address ranges.

**Security Groups are Stateful:** Now, security groups are able to see that the return traffic for a particular request is part of that original request. It's important to know, because when you allow, for example, HTTP or SSH to access your server in an inbound way. You're also allowing the outbound return traffic. You **don't need to specify it**, and in fact you **can't not allow it**.

It's known as stateful for **anything that you allow in, you're also allowing the return traffic for that out**. They are stateful — meaning for any **traffic allowed in/out, the return traffic is automatically allowed**.

**\*\***With Security Groups you can **specify other logical entities like other security groups**: Security groups can reference AWS resources, other security groups, and even themselves.

**Instance Metadata**

Instance metadata is **data relating to the instance** that can be **accessed from within the instance itself using a utility capable of accessing HTTP** and using the **URL**:

**http://169.254.169.254/latest/imeta-data**

Instance metadata is a way that scripts and applications running on EC2 can **get visibility of data related to the instance** **without making an AWS API calls**, which would typically require the instance to be connected to the internet.

So using the metadata gives you the ability to **quickly and simply access information about the running instance**.

The metadata can provide the **current external IPv4 address for the instance**, which isn't configured on the instance itself but provided by the Internet gateway in the VPC.

It provides the **Availability Zone the instance was launched in** and the security groups applied to the instance.

In the case of **spot instances**, it also provides the **approximate time the instance will terminate**.

For the exam: Remember the IP address to access metadata.

**AMI:**

AMIs (Amazon Machine Images) an object or container, and it allows you to launch EC2 instances. They store snapshots of EBS volumes, permissions, and a block device mapping, which configures how the instance OS sees the attached volumes. AMIs can be shared, free, or paid and can be copied to other AWS regions.

**Types:**

* **Instance Store Backed AMIs:** These are used when root volume of the Instance doesn't use EBS.
* **EBS Backed AMIs:** These are used when root volume of the Instance use EBS.

AMI store snapshots of **EBS volumes, launch permissions,** and a **block device mapping**, which configures how the instance OS sees the attached volumes.

* **EBS Volumes:** Generally in most cases in Production you create an AMI based on a source instance which has been configured earlier. AMI contains the necessary configurations required launch a new instance and these configurations are based off the existing instance, you are creating the AMI form. So because your source instance has EBS volumes, so while creating an AMI, it would create snapshots of the EBS attached to the instance and reference those within its configuration.
* **Block Device Mapping:** Every operating system, whether it uses directly attached or network attached storage, uses what are known as **device ID**s, which are the **unique identifier for the device** and it's **how your OS knows to mount a particular drive or a particular volume**.

So because this AMI has snapshots of these original EBS volumes, the **block device mapping** essentially stores a **link between the snapshots and how they're mapped to the new instance**. It's a list of all of the volumes that are attached to this source instance, together with their device ID.

So when you launch that instance, it'll go ahead and create new volumes, and it will know how to map them to the instance so they're accessible to the guest operating system.

You are able to add additional volumes to this AMI but generally you're creating it in AMI to be in exactly the same state as the source instance.

* **Launch Permissions:** These are related who can use your AMI to launch instances. It defaults to **private**, which means only the owning account can use the AMI. It can be changed to **public**, which means that every AWS account will be able to freely search for this AMI and launch instances from it. You can use a middle ground which is set to private but with **individual AWS accounts white listed** so that they can use this AMI to launch instances.

So at this point, because the AMI is created, what I can do is go back to instances.

**Use Cases:**

* **AMI baking or AMI prebaking:** Suppose we create an instance and do configurations(install apache server, download necessary code from repository etc.) on it to host a website, which takes 30 mins. By creating an AMI with all that work on upfront, it saves us that 30 minutes every time we deploy it and architecturally, that's known as AMI baking or AMI prebaking.

You essentially do the work in advance. You do all of the extensive installation and configuration and then you prebake that into an AMI. It can save you significant time if you needed to deploy hundreds or thousands of the same application.

* **Immutable architecture:** Immutable architecture is a technique where servers, in this case EC2 instances, are never modified after they're created. Incase the instance malfunctions due to corruption issues, instead of debugging it the instance is terminated and a brand new instance is provisioned from the known working AMI.

If you want to make a change, you make a different AMI a version two and you deploy that new version of that AMI. AMI s are generally used either to perform base installations or when you're using an immutable architecture.

Generally, organizations have processes that every time they release a new version of their application, they create a brand new AMI with that new application version prebaked into it and it just means that whenever they need to deploy that application, it's in a known working state.

That's immutability. You don't change it afterwards, you don't adjust things. You create a new version. You test that new version and then you use AMI to deploy across your entire infrastructure. Immutability is a great way of ensuring a perfect clean infrastructure, which is free of bugs.

* **High Availability:**

Now AMIs can also be used for scaling or high availability. You can scale using auto scaling groups and these could be used to deploy and scale EC2 instances based on load, and you can utilize AMIs to deploy new instances using a prebuilt application that stored inside the AMI. So it saves time when you're provisioning instances.

**Advantages of Using AMIs:**

**The biggest advantage to baking any configuration into an AMI is to reduce the time it takes to provision that instance.** So if you got any applications that take a long period of time to install and configure then they're candidates for baking into the AMI.

An example of this is if you need to get new instances into production in less than 20 minutes and an application takes 45 minutes to install and configure, then you know you can't use bootstrapping.

Bootstrapping in that scenario would be including the shell commands in the user data that would perform the installation of the application automatically but if it takes 45 minutes, it's going to be 45 minutes before your instance is ready for production. So in that case, you'd use an AMI.

**Down Side of Using AMIs:**

**The downsides to using AMI is that you don't really have the ability to do any dynamic configuration.** So if you bake everything into the AMI, it's already going to be configured. You can't really do any configuration based on the size of the instance, the type of the instance, the subnet the instance is in, or any DNS or IP addressing. Anything specific that you want to be dynamic if you use 100% AMI baking you don't have that option. With AMIs, the configuration is built in so it’s not possible to customize based on things like instance size or availability zone or subnet or IP address. That's where bootstrapping comes in handy. **Bootstrapping is when we use the user data to build an EC2 instance.**

**Cross Account AMI Copy (FAQ + Exam Tip)**

* You can share an AMI with another AWS account.
* Sharing an AMI does not affect the ownership of the AMI.
* If you copy an AMI that has been shared with your account, you are the owner of the target AMI in
* your account.
* To **copy an AMI that was shared with you from another account**, the **owner of the source AMI** must **grant you read permissions for the storage that backs the AMI**, either the associated EBS snapshot(for an Amazon EBS-backed AMI) or an associated S3 bucket (for an instance store-backed AMI).
* **Limits:**
* You **can't copy an encrypted AMI** that was shared with you from another account. Instead, if the **underlying snapshot and encryption key were shared with you, you can copy the snapshot while re-encrypting it with a key of your own**. You own the copied snapshot, and can register it as a new AMI.
* You **can't copy an AMI with an associated “billingProduct”** code that was shared with you from another account. This includes Windows AMIs and AMIs from the AWS Marketplace. To copy a shared AMI with a billingProduct code, launch an EC2 instance in your account using the shared AMI and then create an AMI from the instance.

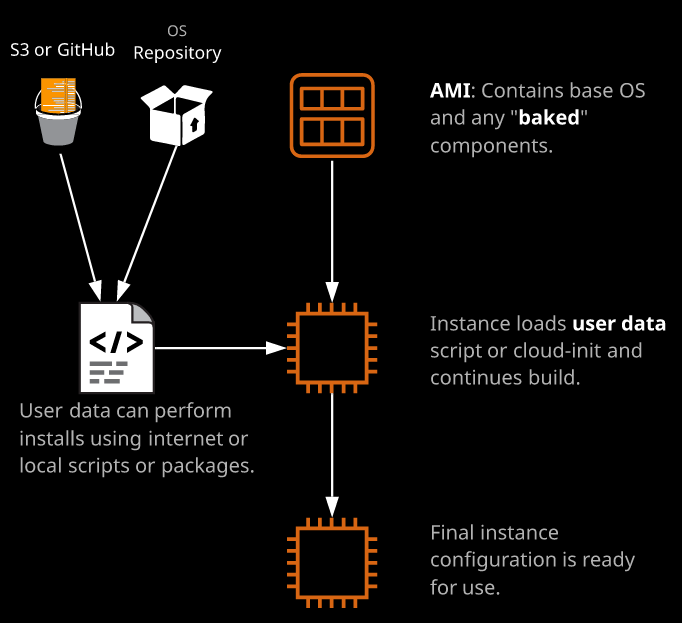
**Bootstrapping:**

Bootstrapping is a process where instructions are executed on an instance during its launch process. Bootstrapping is used to configure the instance, perform software installation, and add application configuration.

**User Data:**

User data is the EC2 implementation of bootstrapping. Essentially, whatever you paste in this user data section is used by the EC2 instance, when it's first provisioning so it can install applications, it can perform certain configurations, it can pull down data from a GitHub repository or an S3 bucket.

User Data can be specified in two forms:

* **Script:** Using **shell script** for Linux or **PowerShell script** for Windows
* **Cloud-init Directives:** Cloud-init Directives are similar to the scripts but these directives are **OS Agnostic**.

**Exam Tips:**

Choosing between AMIs and bootstrapping doesn't have to be a binary one.

You can choose to use and AMI to bake in lot of the really intensive parts of building an application. So if you're doing an installation and that installation say takes 20 or 30 minutes and then after the installation, you can do some configuration.

What you might want to do is AMI bake the installation and then perform the configuration dynamically, using bootstrapping.

So for the lengthy resource intensive part, that could impact product we can use AMI baking and for the dynamic configuration part that you want to do it at instance provisioning time you should use bootstrapping and generally, most production systems use a combination of the two.

There are exam questions on this, so when you do face an exam question that might mention AMI baking or bootstrapping, remember, AMIs are for your lengthy static configuration.

Bootstrapping is what you use for anything dynamic.

When you're facing a question that talks about, you need to get an instance into production within a certain time frame and then it says that the application might take 20 or 30 minutes to install focus on the timeframes.

It's those that will give you the clues whether you need to use AMI baking or bootstrapping.

**If you need to do the work upfront and save time, bake it into the AMI.**

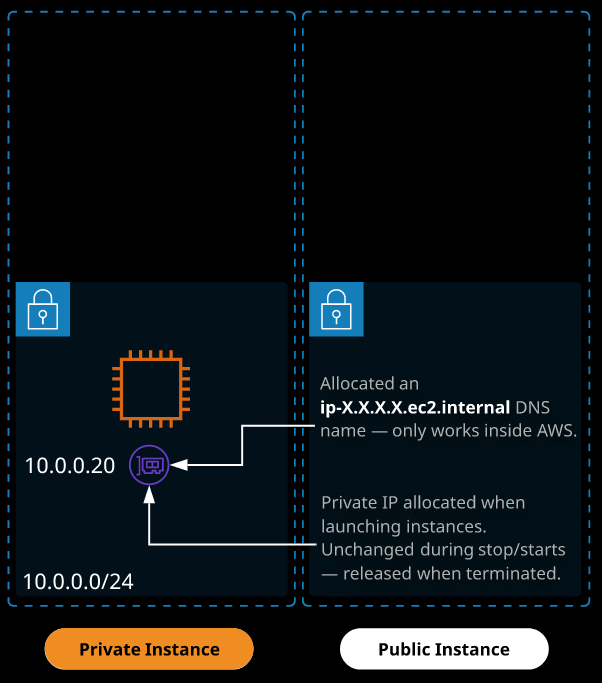
**If you need to do any dynamic configuration then use bootstrapping.**

**Private EC2 Instance:**

A private instance can be in the VPC, which have public connectivity only it's **not given a public IP address** or it's in a **subnet that's not configured to be publicly accessible.**

This can be done by changing **"Auto-assign Public IP"** to **"disable"**.

**ENI:**

By default, any instance created by EC2 starts off with a **default ENI** or **elastic network interface**. We can have **multiple network interfaces.**

**Private IP Addresses:**

* So this is the network interface card, or virtual network interface card that the instance starts with and when you launch any instance into EC2 with an elastic network interface, it's going to be allocated an IP address within the range that the subnet uses. It's known as a **private IP address**.
* **Private IP Address** is **not routable across the public internet**.
* **Private IP Address** is associated with the instance for the **lifetime of that instance**, so it stays the same. So if we stop an instance & restart it, the Private IP address remains same.
* The default network interface that's associated with an EC2 instance is known as eth0.
* We can have **one primary** & **more than one secondary Private IP addresses** associated with an ENI and the number of secondary private IP Address depends on the size of the instance.

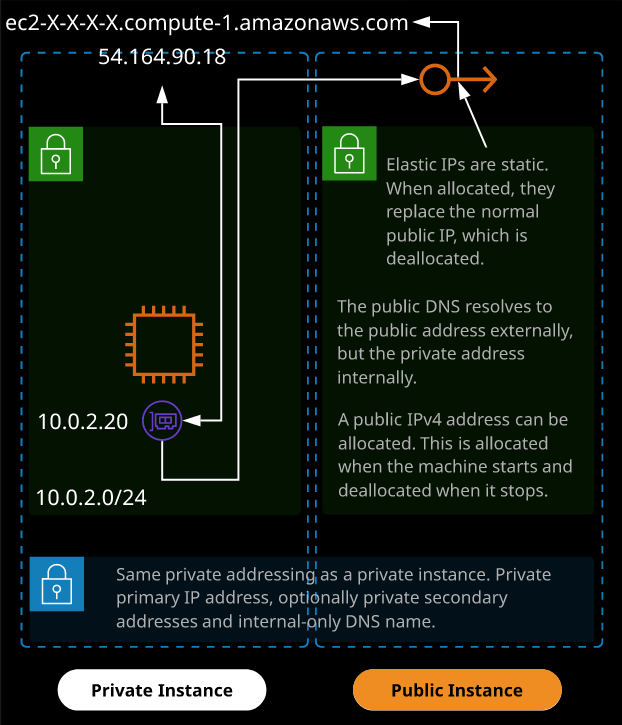
**Private DNS Name:**

* Private instances have **internal only DNS name**, known as private DNS name.
* **Private DNS Name** has got the **private IP address built into it**. So its ip- and then the IP address but instead of dots, it has dashes.
* If the **IP address were to ever change this internal DNS name would also change**. It's based on the private IP address, but because the **private IP address belongs lifetime of the private instance, so does this internal only DNS name**.
* Private DNS name **can't be pinged externally**.
* It only works inside the VPC, and it always **resolves to the private IP address of the EC2 instance**.

**For Example:**

* **Private DNS:** ip-172-31-52-57.ec2.internal
* **Private IPs:** 172.31.52.57

**Public EC2 Instance:**

Public EC2 instances apart from having the same private addressing in Private Instances receive an **IPv4 public IP address** and the subnet it's located in has a **route to the Internet Gateway configured for traffics outside the VPC**. This public IPv4 address can be **referred to from the Internet**.

**Internet Gateway:**

This **public IPv4 address** is not configured on the **Operating System**.

Every time any device is inside AWS are given public IP addresses; when a device that has an associated public IP address attempts to communicate with the internet or when any internet devices attempt to communicate with its public IP address the job of internet gateway swap those IP addresses.

The internet gateway actually uses a **process called NAT or network address translation**. It **translates** a **private IP to a public** and a **public IP to a private**.

So if the instance which doesn't have a public IP address on the operating system attempts to do let's say a ping to the internet so I ping 1.1.1.1 what will happen is this internet gateway will receive data from this private IP address and it will swap that private IP address for this public IP address. So when 1.1.1.1 receives the traffic, it will see it as from this IP address. The same process is true in reverse.

**Exam Tips:**

* You might get a question where an **EC2 instance cannot reach any machines on the public internet** or **any machines on the public internet can't reach your EC2 instance** and **one of the answers** might be you need to **apply a public IP address on the guest operating system, and that is not a valid way to fix the problem**.
* You cannot apply a public IP address on any operating system inside AWS.
* Everything in **AWS is natively private when using IPv4** and it's the job of the **internet gateway to translate that** using a process called **NAT or network address translation**. It translates a private IP to a public and a public IP to a private.

**Public DNS:**

* Along with the public IP address EC2 instances also receive a public DNS name.
* If we attempt to ping the public DNS from the **public internet**, it **resolves to the public IP address**.
* If we attempt to ping the public DNS from **within the VPC**, it **resolves** to the **private IP address**.
* This means that you can use the same DNS name, whether you're in the VPC or on the public internet and it will always talk to the **best IP address of that EC2 instance** whether it's private or public.

**Dynamic IPv4 Address:**

* Public IP address of an EC2 instance of the public IPv4 address doesn't change when you reboot an instance.
* But If we are using **dynamic IP address**, every time the **instance is stopped** the public **IPv4 address is deallocated** from the instances, so **dynamic IPv4 addresses change**. They change when the **instance moves between EC2 hosts** when it stopped and when it started.
* So when the instance is stopped, the IP address, the public dynamic IP version four address, is deallocated from the instance.
* When the instance is started again, a new public IP address will be associated to a brand new public address but in addition, because the public DNS is based on that IP address, the instance will also get a brand new public DNS as well.
* **By default all public IP version four IP addresses** that could be associated with EC2 are **dynamic**.
* They **change if the instance is stopped and started**, or if the EC2 host that in instance is running on changes.

**Elastic IP Address:**

* Elastic IP addresses these are IP addresses that are **valid for the entire AWS region**. So it's not an availability zone specific thing.
* AWS have a **pool of these elastic IP addresses** and the first thing that you need to do is to **allocate a new address with your account in this region**.
* You can then **associate this elastic IP with a private IP address on that EC2 instance or ENI**. So you're creating a **link between the private address and this elastic IP address**.
* Remember, that the public **IPv4 address doesn't change**.
* With elastic IP address with an EC2 instance, what happens is any existing dynamic or changeable IP addressing is removed so it loses that existing public address. In its place, the static elastic IP is added. So we get new public DNS name which matches this Elastic IP address.
* Unlike the previous dynamic IP address, the **Elastic IP Address & the DNS associated** is **valid for life time of the instance**.

**IPv6 Addresses:**

IPv4 is limited and it's actually running out of public IP addresses, which is why we generally use private ranges for most things.

IPv6 was designed to fix that shortage and so **all IP version six addresses inside AWS are public**.

You don't pick which addresses you want to use they're allocated by AWS but if your VPC is enabled for IPv6 and your subnet is enabled for IPv6.

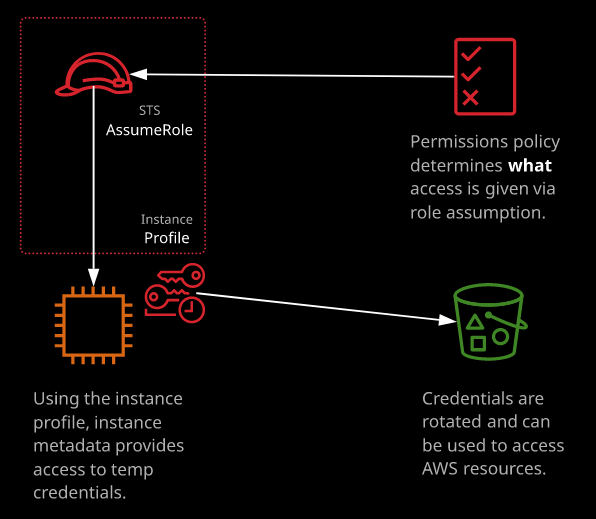
EC2 Instance Roles:

EC2 instance roles are IAM roles that can be **"assumed" by EC2** using an intermediary called an **instance profile**.

**Instance Profile:**

* Instance Profile **is a container for the role that is associated with an EC2 instance**.
* A normal IAM role can be assumed by an identity in AWS. The EC2 service is an identity, but applications running on the instance are not valid AWS identities, so to give these applications access to AWS create something that's known as an instance profile which is essentially a container for an IAM role that you can be used to pass that role information to EC2.
* It's essentially just a translating container. It allows applications on EC2 to use that role on your behalf, so you're not just letting EC2 assume the role you're allowing the applications to utilize that role as well.
* The instance profile **allows applications on the EC2 instance** to access **the credentials from the role using the instance metadata**.
* An **instance profile** is either **created automatically** **when** **using the console UI(**with the same name as the Instance Role**)** or **manually when using the CLI**.
* Now, **creating the instance role** and creating the **instance profile** are **actually two different steps, and if you do this from the command line**, you'll need to create them both separately.

A role has two things:

* **Trust policy** which defines who can assume that role.
* **Permissions policy** that gives the role permissions.

**How it works:**

Suppose we create role with permission policy for an instance role of S3 read only from the console and associate apply it to an EC2 instance. This creates an instance profile of the same name as the role and associates this EC2 instance with this instance profile and thus the IAM role that has the associated permissions policy.

**STS**, also known as the **security token service** generates short term or temporary credentials whenever a role is assumed.

Suppose we login to the EC2 console as run the aws s3 ls command, the EC2 instance begins assuming the role.

The STS service every time that occurs, which is constantly while this role is associated with the instance, the STS service begins delivering short term temporary access credentials, and they're available from within the instance metadata. Now that means that anything that's running inside the instance, has access to those temporary security credentials.

So now if I do an aws s3 ls and press enter, we get all the s3 buckets listed.

So they're using temporary access credentials that are inside the instance metadata in order to interact with AWS.

**Advantages of using Instance Roles:**

We are able to create a normal IAM role and an associated instance profile that I can associate that with 1, 10, or hundreds or even thousands of EC2 instances. Those instances work in conjunction with STS to generate temporary security credentials that are available inside that instance to applications running on that instance.

Now the key thing is that **these are temporary security credentials**, and they're **rotated every time EC2 assumes this IAM role** and so it means if this **instance is exploited** and the **temporary credentials are leaked**, it means that **the impact of that is limited** because they're not permanent credentials they expire periodically and so, even though you might have the credentials short term, once they do expire, you'd need to reassume the role to get access to new credentials and so, by using this temporary, rotating credential architecture, you offer a much improved security situation.

It also means that you don't need to worry about how you deliver credentials to these instances.

**Disadvantages of IAM Roles:**

The only scenario where **IAM roles really don't work** is when a **log in is required**. So when a user needs to log in because, of course, IAM roles don't have any long term credentials.

Other ways of authenticating to AWS from inside EC2 instances include using AWS Configure and setting Credentials in Environment Variables.

**EBS Encryption:**

Volume encryption uses EC2 host hardware to encrypt data at rest and in transit and that's between the EBS volumes and the EC2 instances.

EBS encryption is configured on a per volume basis, but it can be set it from an account perspective(Recommended), this ensures that all new EBS volumes in the region that you're currently located in will use encryption.

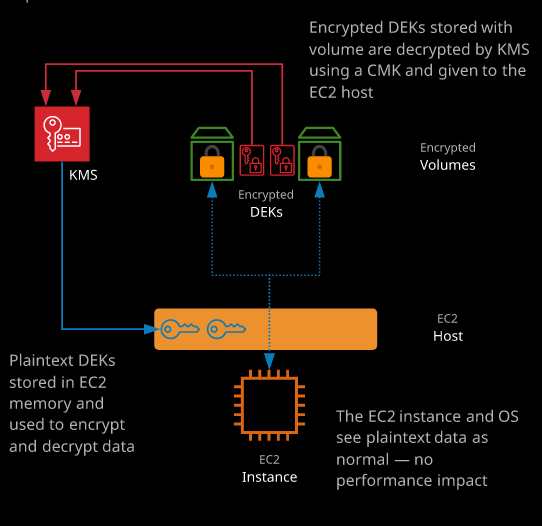
**KMS**

So the encryption and decryption the EBS performs is handled in part by KMS. It manages encryption keys. The primary type of encryption key is known as a CMK or customer master key.

* **AWS managed keys:** provided and managed by AWS. Very limited amount of control. By default, when any service within AWS wants to use encryption, it creates an AWS managed key inside the particular region.

So anything that's aws/ is an AWS managed customer master key.

* **Customer managed keys:** created and managed by customer. more control. Additional control around elements of that key, including key rotation. Only if you've got particular key rotation requirements would we use Customer Managed Key.

**Architecture:**

* Whenever encryption is required, the service asks KMS to generate what's known as a data encryption key or DEK.
* Now KMS generates two versions of these data encryption keys, an encrypted version and a plain text version.
* The encrypted version of that data encryption key could be stored along with whatever is being encrypted, so it's perfectly safe to store it along with the encrypted volume within the EBS service.
* The decrypted version of that key is sent to the EC2 host that's running this instance, and it's only ever held in memory on that EC2 host while this instance is active.
* The minute this instance moves, ie. host is rebooted, or the minute the instance is shutdown, the plain text version of this key is discarded, and you can only get back that plain text version by requesting that KMS decrypts the encrypted version and sends it back.
* So there's always this secondary entity which is KMS controlling access to these plain text keys.
* So that means in order to encrypt or decrypt disks, not only do you need the relevant IAM permissions to perform those activities, you also need to be able to interact with KMS.
* So KMS generates these customer master keys. Those never leave the device. EBS requests KMS to generate some data encryption keys, and then it delivers those keys back to EBS, an encrypted version that's stored along with the encrypted data and a plain text version that you used to actually encrypt and decrypt that data in transit.
* Now every single volume that you create inside EC2, as long as you're creating it blank from fresh uses its own dedicated data encryption key and that's generated by KMS.

**EBS Encryption is not at an OS Level:**

* Using EBS Encryption doesn't mean from OS perspective, it's an encrypted volume. From OS perspective it has no concept of the volume being encrypted.
* Now the reason for that is the encryption happens on the EC2 host, and there's no loss of performance whatsoever on the EC2 instance, as far as the EC2 instance is concerned, its running on an encrypted data.
* The data is essentially encrypted on the EC2 host, and it's transmitted in an encrypted form through to the EBS volumes where it's stored encrypted.
* When that data's needed, it's pulled off the EBS volumes into the EC2 host and these plain text data encryption keys are used to decrypt that data and present it to the operating system.

**Snapshot Encryption:**

* Every blank encrypted volume that you create inside EBS as I mentioned a second ago uses its own dedicated data encryption key.
* Creating a snapshot of an encrypted volume would create encrypted snapshot, which would use the same data encryption key as the original volume and if you create any volumes from this snapshot, they also have to be encrypted, and they use the same data encryption key.
* If you have a copy of a snapshot to another region, you'll need to specify a destination encryption key(another CMK) to use because KMS is a regional service.

**AMI Encryption:**

An AMI uses snapshots. Creating AMIs from encrypted volumes would result in AMIs also being encrypted with the same CMK. So they have the same dependencies. Since they'll utilize the same encryption key structure, and if you move them or share them between accounts, you've got the same problem with dependencies on the keys.

**Exam Tips:**

* Encryption is handled by an EC2 host.
* It doesn't cause any performance impact whatsoever on the instance.
* The Instance/OS is unaware that there is any encryption or decryption occurring. From its point of view, it is just an unencrypted file system.
* Now EBS encryption means that the data is encrypted at rest and in transit to and from the EC2 host.
* So again, it doesn't meet most encryption requirements for most organizations.
* One last thing, if you do get any exam questions that require you to manage the encryption keys and handle the encryption from an operating system perspective then EBS encryption is not going to be enough. So if you need to manage the key and if you need the operating system to manage the encryption process, then don't utilize EBS encryption utilize built-in operating system encryption.
* You are able to use both but be aware from an exam question perspective only operating system encryption will ensure that from an operating system perspective, the file's encrypted.
* Remember, from an operating system perspective, EBS encryption is transparent and you don't manage the keys, they're managed by KMS.
* So they're important exam things to keep in mind.
* Now EBS encryption is supported by most instance types. There are some older generation instances which don't support it, but any of the current modern instance generations, especially those that use the nitro platform, do support encryption but at this point

**EBS optimization:**

Legacy non-EBS-optimized instances used a shared networking path both for network data and storage data communications which means both of those different types of data contested for access to that storage path.

For a resource intensive application running on an EC2 instance, which is running heavy on storage, that could disrupt traditional networking transfer and if you had an application that was heavy on normal data transfer, it could impact storage performance.

EBS optimization created a separate, dedicated path for storage. Historically this was optional and you could enable it for an extra cost, but for modern instances this feature is included by default.

**Benefits of EBS Optimized Instances:**

* Access to better and faster storage.
* Improved network data transfer rates.
* Higher level of consistency on both.

\*\*To get maximum performance from EBS apart from selecting appropriate sized instance, you also need to make sure that using EBS optimized instances which any current generation instances because with those you get the maximum performance.

**Performance implications of restoring volumes from EBS Snapshots:**

* If we take a snapshot of an EBS volume, it's copied in S3. If we launch an instance from the snapshot it would create the new volume and allocate the space EBS but it doesn't immediately copy all that data to EBS from S3, it copies it in the background.
* This means that you don't get the maximum performance of an EBS volume created from a snapshot until all that data has been copied across in the background. If you ask for some data from the volume that's not yet been copied, it will immediately be copied.
* One way to improve the performance id to perform a read of every part of that volume in advance before moving it into production.

**Enhanced Networking:**

EC2 uses a software called a hypervisor that runs on the EC2 host, that carves up the physical resources, present them as individual virtual machines, and then control access to the physical resources for those virtual machines, so EC2 instance is a virtual machine on the EC2 host.

Since there can be multiple EC2 instance on a host, Hypervisor controls, whether our EC2 instance that currently has access to the networking card that's in the physical EC2 host. Historically, that was done using a software, that emulated a real hardware. Since EC2 instance is not accessing its own dedicated networking card but running through software, that comes with some performance implications because the software is not as fast as accessing through real hardware, and it's not as consistent.

Enhanced Networking uses a technique called **SR-IOV(single root input output virtualization)** that allows a single physical network card to appear as multiple fake physical devices, known as virtual functions.

So rather than the one physical network card presenting itself as one physical adapter, it presents itself as multiple physical adapters. For a hypervisor that supports this, each VM can be given direct access to one of these virtual functions. So instead of hypervisor controlling VM accesses hardware at a time, the VM(s) or EC2 instances are capable of interacting directly and it's the network card, which handles this interaction.

**Benefits of Enhanced Networking:**

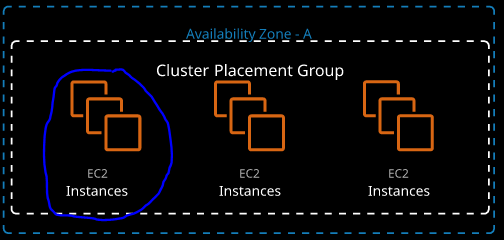
* Less CPU is used on the host during heavy utilization.
* Lower consistent latency.
* Better networking performance.

So enhanced networking when you're using EC2 uses **SR-IOV**. EC2 delivers this via the **Elastic Network Adapter (ENA)** or Intel 82599 Virtual Function (VF) interface. So **enhanced networking** is one component of **achieving maximum performance of networking inside AWS**.

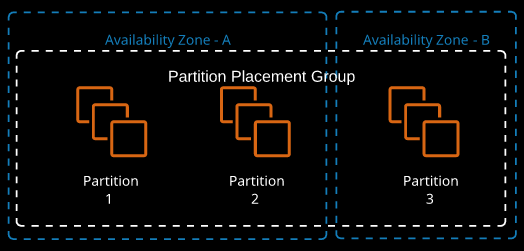
**Placement groups:**

When you launch a new EC2 instance, the EC2 service attempts to place the instance in such a way that all of your instances are spread out across underlying hardware to minimize correlated failures. You can use placement groups to influence the placement of a group of interdependent instances to meet the needs of your workload. Depending on the type of workload, you can create a placement group using one of the following placement strategies:

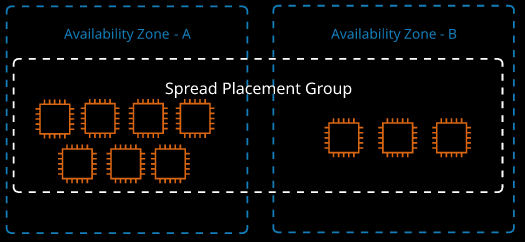
**Cluster:**

* Packs instances close together inside a **single Availability Zone**.
* This strategy enables workloads to achieve the low-latency network performance necessary for **tightly-coupled node-to-node communication** that is typical of **HPC applications**.
* A cluster placement group is a **logical grouping of instances within a single Availability Zone**. A cluster placement group can span peered VPCs in the same Region. Instances in the **same cluster placement** group enjoy a higher per-flow throughput limit of up to **10 Gbps for TCP/IP traffic** and are placed in the same high-bisection bandwidth segment of the network.
* Cluster placement groups are recommended for applications that benefit from low network latency, high network throughput, or both. They are also recommended when the majority of the network traffic is between the instances in the group.
* To provide the lowest latency and the highest packet-per-second network performance for your placement group, we need to choose an instance type that supports **enhanced networking**.
* We recommend that you launch your instances in the following way:
* Use a **single launch request to launch the number of instances** that you need in the placement group.
* Use the same instance type for all instances in the placement group. If you try to add more instances to the placement group later, or if you try to launch more than one instance type in the placement group, you increase your chances of getting an insufficient capacity error.
* If you stop an instance in a placement group and then start it again, it still runs in the placement group. However, the start fails if there isn't enough capacity for the instance. If you receive a capacity error when launching an instance in a placement group that already has running instances, stop and start all of the instances in the placement group, and try the launch again. Starting the instances may migrate them to hardware that has capacity for all of the requested instances.

**Partition:**

* Spreads your instances across logical partitions such that groups of instances in one partition do not share the underlying hardware with groups of instances in different partitions.
* This strategy is generally used to deploy **large distributed and replicated workloads**, such as **Hadoop(HDFS & HBase), Cassandra, and Kafka** across distinct racks.
* Partition placement groups help **reduce the likelihood of correlated hardware failures** for your application. When using partition placement groups, Amazon EC2 **divides each group into logical segments called partitions**. Amazon EC2 ensures that **each partition within a placement group has its own set of racks**. Each rack has its **own network and power source**. No two partitions within a placement group share the same racks, allowing you to **isolate the impact of hardware failure within your application**.
* When you launch instances into a partition placement group, Amazon EC2 tries to distribute the instances evenly across the number of partitions that you specify. You can also launch instances into a specific partition to have more control over where the instances are placed.
* A partition placement group **can have partitions in multiple AZ in the same Region**. A partition placement group can have a **maximum of 7 partitions per AZ**.
* The **number of instances** that can be launched into a partition placement group is **limited only by the limits of your account**.
* In addition, partition placement groups offer visibility into the partitions — you can see which instances are in which partitions. You can share this information with topology-aware applications, such as HDFS, HBase, and Cassandra. These applications use this information to make intelligent data replication decisions for increasing data availability and durability.
* If you start or launch an instance in a partition placement group and there is insufficient unique hardware to fulfill the request, the request fails. Amazon EC2 makes more distinct hardware available over time, so you can try your request again later.

**Spread:**

* Strictly **places a small group of instances** across **distinct underlying hardware** to **reduce correlated failures**.
* A spread placement group is a **group of instances** that are each **placed on distinct racks**, with **each rack having its own network and power source**.
* Spread placement groups are recommended for applications that have a **small number of critical instances** that should be kept separate from each other. Launching instances in a spread placement group **reduces the risk of simultaneous failures** that might occur when **instances share the same racks**.
* Spread placement groups provide access to distinct racks, and are therefore **suitable for mixing instance types** or **launching instances over time**.
* A spread placement group can span **multiple AZ in the same Region**.
* You can have a **maximum of 7** running instances per Availability Zone per group.
* If you start or launch an instance in a spread placement group and there is insufficient unique hardware to fulfill the request, the request fails. Amazon EC2 makes more distinct hardware available over time, so you can try your request again later.

**Use Cases:**

* Use cluster placement groups for maximum performance.
* Spread placement groups for maximum availability.
* Partition placement groups if you've got large infrastructure platforms where you want to have some visibility of where those instances are from a partition perspective.

**On-demand Pricing:**

With on demand billing, you pay per second charge for EC2 instances with a 60 second minimum when the instance is running it immediately gets charged for 60 seconds of usage and then continues after those 60 seconds on a per second basis.

This is based on the type and size of the instance and the region that it's launched in.

It's the default whenever you're launching instances into EC2 and you're not aware of what the expected load or the projected load is, then you should default to on demand.

With on demand there's no relationship or advanced warning between you and AWS.

**How Spot & Reserved Instances offer lower prices:**

* When you're using on demand, you're not giving AWS any advance warning until you are actually launching the instance and it's a fairly difficult process for AWS to capacity plan the advanced purchase of physical capacity.
* When AWS do purchase physical capacity, they've got two problems.
* The first is they have to do it in advance, so there aren't on going capacity issues.
* Secondly throughout a 24 hour period there would be period of high demands and times where demand is low, so would be times in a normal day where AWS have short term spare capacity and because AWS purchase infrastructure in advance there's also going to be long term spare capacity.
* Now spot pricing and reserved instances are two ways that you and AWS can make a deal or solve each other’s' problems.
* So, depending on the type of usage that you've got, you'll generally have an ability to select spot or select reserve pricing and in doing so, you're either solving AWS's capacity problem or you're informing them in advance of your expected usage and in both of those billing models, you're exchanging something.
* You're giving AWS a benefit, and in return, you're getting better instance pricing.

**Spot Instances:**

* Spot instances **allow consumption of spare AWS capacity** for a given **instance type and size in a specific AZ**.
* **Instances are provided** for as long as your **bid price is above the spot price**, and you only ever **pay the spot price**.
* **If your bid is exceeded**, instances are **terminated with a two-minute warning**.

**Spot Fleets:**

Spot fleets are a **container for "capacity needs."** You can specify pools of instances of certain types/size**s** aiming for a given "capacity." A minimum percentage of on-demand can be set to ensure the fleet is always active.

With Spot Fleets you make a request for a certain capacity an amount of compute and giving AWS a directive on how to fulfill it.

You can **define pools of certain types and sizes of instances**, and these **could be different**, and you **give AWS some flexibility** on which type and **size of instance to use and get you the best value**.

You can also **direct AWS to make sure that for a given spot fleet**, **a certain amount of that fleet is based on demand instances**. So if the fleet is **always going to be active** then using on demand is a great way of fulfilling baseload and then using spot instances to provide that variable load and you'll do that and always achieve the best cost efficiency.

**Use Cases:**

* Spot instances are perfect for **non-critical workloads, burst workloads**, or **consistent non-critical jobs that can tolerate interruptions** without impacting functionality.
* Spot is **not suitable for long-running workloads that require stability and cannot tolerate interruptions**.
* You'd utilize spot pricing when you need really cheap EC2 capacity, when you can tolerate failure, when your workloads aren't time sensitive, or you might use spot pricing as a buffer to cope with bursts, for example, during sale periods.
* If you do you spot pricing you need to make sure that your application can tolerate failure.
* If you use it for **web applications** the architecture needs to make sure that the **state is not on those servers** because you don't want to disrupt your user experience.

**Reserved Instances:**

With the reserved purchases you're giving AWS information about your future usage requirements, for example you tell AWS that you'll have a requirement to use a certain type of instance in a certain region or availability zone for a certain period of time and in exchange for that, you're getting these instances for cheaper.

Reservations can be **zonal or regional.**

Now, the way that reserved instances work is that you can pay for reserved instances for a period of **one year** or **three years** to pre-purchase the capacity for a given type and size of instance in a region or AZ for one or three years.

Purchasing it for three years means you commit to pay for that instance for three years.

* **Full Upfront:** You can either go for all upfront where you're paying for the entire term in one bulk payment that will offer you the maximum cost saving.
* **Partial Upfront:** You can go for partial upfront where you're paying a part of the fee upfront in exchange for a reduced hourly instant rate.
* **No Upfront:** You can pay for no upfront, which just gives you a discount to the on demand pricing but you're still committed to pay it.

**Regardless of the fact that you stop the instance or terminate the instance, you pay for a reservation:**

* Now the way this works on a bill is that if you start up an instance that matches the type and size that the reservation is for, then you automatically either get the discounted rate or you don't pay an hourly rate. The reservation is like a credit, and if you start matching instance, then you obviously you get a discount or you don't pay.
* Suppose you also purchased a one year reservation for the a T3 type of instance and then you decide not to use it(Not sure if you are an idiot or just a bad cloud architect or you have a lot of money). It would be wasted.
* If at that point, you started up a T3 instance which matched this wasted reservation that at that point I'd utilize this reservation,
* Now the best way to reduce your bill if you can predict your usage over long term, is to purchase three year reservations and pay in its entirety upfront.
* In doing so, you get the best discount, and you don't have to worry about any hourly charges for the instances.

Now, as well as deciding between the one year or three year and then the all upfront, partial upfront, or no upfront, you also get to decide whether you buy a zonal reserved instance so that applies to a particular availability zone or a regional reservation.

**Zonal Reservation:**

If you choose to buy a zonal reserved instance, then you only **benefit from that reservation in a particular availability zone** but you also **reserve capacity in that zone**.

A zonal reserved instance that was also **locked to a specific size**. So if you purchase an instance reservation, for example, as an M5.large. It'll only apply if you start up an M5.large.

So that's key a **zonal reserved instance reserves capacity in a particular availability zon**e, but it is **locked to a specific size**.

**Regional Reservation:**

Now regional **doesn't lock you to a particular availability zone**, but it also **doesn't reserve any capacity**.

With Regional Reservation you get **flexibility on size**. If you purchase a reservation for a T3.medium in a particular region, then you can also run two T3.smalls and have both benefit. Regional reservations also benefit in part, so if you buy a regional reservation for a T3.medium and then launch a larger T3 instance then you can get benefits on part of that instance. So it results in part reduction in hourly rate or a partial full reduction in the instance costs, but again that's only when you use regional reserved purchases.

So when you buy these reservations, remember they're zonal or they're regional.

**Reserving Capacity:**

**Zonal reserves capacity**, **Region gives you a lot more flexibility**.

**Capacity reservation** is included when you do a **zonal reservation**, so when you pick a particular availability zone, you reserve capacity. If you use **regional**, you don't reserve capacity so if AWS do **have capacity constrains in a particular availability zone**.

They **resolve** capacity constraints by **prioritizing anyone who's reserved capacity**. So anyone who's got **zonal reserved instances have guaranteed access to that capacity**.

The **second priority** is on **demand instances** and the **last priority**, are **spot instances**.

So if you've got any stringent requirements, where you need to make sure you absolutely can start up an instance in a particular availability zone. Then you need to make sure that you get that capacity.

**Key Facts for the Exam:**

* Instance size/type have an AZ spot price.
* Bid more, instance provisioned for spot price. Less = termination.
* Spot fleets are containers, allowing capacity to be managed.
* Reservations are zonal (AZ) or regional.
* One or three years, no upfront, partial upfront, all upfront.
* You pay regardless of EC2 instance using a reservation.
* Regional is more flexible — but has no capacity reservation.
* **When to Use Reserved Purchases**
  + Base/consistent load
  + Known and understood growth
  + Critical systems/components
* **When to Use Spot Instances/Fleets**
  + Burst-y workloads
  + Cost-critical, which can cope with interruption
* **When to Use On-Demand**
  + Default or unknown demand
  + Anything in between reserved/spot
  + Short-term workloads that cannot tolerate interruption

**Dedicated Hosts:**

**Dedicated hosts are EC2 hosts** for a **given type and size** that can be **dedicated to you**. The number of instances that can run on the host is fixed — depending on the type and size.

Dedicated host **come with an associated hourly charge**. So instead of paying for individual EC2 instances, you're paying for a specific EC2 dedicated host and you can pay those either on demand or using reserved purchase types.

With dedicated hosts you specify an **availability zone**, which the dedicated hosts go into.

It's **not a regional service**.

This is **physical hardware** that's **going inside an availability zone**.

You need to pick a **specific type of EC2 instance host**. Each **different type of EC2 host the AWS use is designed for a particular type and size of instance**.

* So, for example, if I want to launch a specific size of M5 utilizing one type of EC2 host, which is an M5 type. Now that EC2 host can either run for 48 M5.larges or six M54.extralarges or somewhere in between.
* So when you **allocate a dedicated host**, you need to pick not only the **instance type** but also the **instance size** and those two things together will govern **how many instances you can run on that host**.

**Deploying Instances on to EC2 Hosts:**

This is a physical service, you're actually **allocating a physical EC2 host** that you can use to **deploy instances onto**.

Now this is beneficial because it allows you to **manage your own capacity** and **control instance placement on specific hosts**.

If you got multiple EC2 dedicated hosts, you can specify which hosts your instances are placed on.

* **Auto Placement:** You can also enable auto placement then you can obviously specify no preference and allow instances to be placed on the most appropriate host.
* **Disable Auto Placement:** If you disable auto placement, then you need to specify which host to use for every instance launch.

You need to specify a type of instance that you be launching on this host and given that type, it'll control how many instances could go on each host. For example if we, specify M5.large that's 48, if we specify M5.4.xlarge that six.

**Host Recovery:**

You've also got the ability to use host recovery, which means if a dedicated host fails for some reason then your instances can automatically restart on a new host assuming that you've got one.

**Use Cases:**

Now you might be asking under what situations you would use a dedicated host and their three main business requirements that would mandate you to use a dedicated host.

* **Regulatory Compliance:** The first is **compliance and regulatory reasons**. So by using dedicated host, you're ensuring that your **instances are running on hardware that's dedicated for your use.** So there are some **governance and security benefits** to choosing a dedicated host.
* **Licensing:** The second reason is that for **certain enterprise licensing models**, they're fairly straight about the **number of CPUs and the number of cores and a given hosts that their software runs on**.
* So if you're running your software on your own dedicated host, you know exactly how many CPUs and cores that host has, and you're able to more appropriately manage licensing.
* **Control Instance Placement:** This is further improved by the fact that you can **control instance placement**, which means for any instances running on these host that's running this type of license restricted software you can often achieve cost efficiencies by managing the placement of this software running on these instances.

So the type of scenarios that you'd use dedicated hosts in are fairly specific to licensing, to governance, and to really strict instance placement requirements.

**Monolithic Applications:**

Monolithic applications, are generally a **collection of individual bits of functionality together** with **shared supporting components**. Examples of this are web servers and database servers.

**Disadvantages of Monolithic Applications:**

* Because of the monolithic nature, all of these individual components, they require & rely on each other so separating them or scaling the functionality is not possible.
* In addition to scaling, having a monolithic application also makes updates and maintenance significantly harder, because each small change to one part of the code could have implications to other parts and so it adds significant business risk.

**Microservices:**

A microservices architecture is the inverse of a monolithic architecture. Instead of having all system functions in one codebase, **components are separated into microservices and operate independently**.

* A microservice **does one thing** — and **does it well**.
* Operations, updates, and scaling can be done on a per-microservice basis.
* Microservices operate as independent applications.
* They allow direct communication between components and the end user.
* If one part of the system requires more capacity, that service can be scaled and updated as needed.

**APIs:**

An API or an application programming interface is an **interface published by a service or application, which is designed to be used or consumed by another service or application**.

There are different types of APIs including REST, SOAP, and Web Sockets, and each of these has their pros and cons.

APIs, though, have data transfer formats. For example, rest APIs generally used JSON and JSON is a format.

APIs are published in a way that allows a microservice to interact with another service so an API defines what interactions can occur with that microservice. It defines what data gets supplied and returned, and it defines any authentication or authorization that's used to interact with that microservice.

**Use Cases of APIs:**

* Now as a human being, you actually use APIs indirectly.
* When using AWS when we've been interacting with the AWS console behind the scenes that's actually using APIs to communicate with the backend AWS services and features of those services.
* If you use Microsoft Word to store files in to One Drive that's an API.
* If you used the Netflix application on your phone, tablet, or TV that's not connecting to the user interface that you might use if you go to netflix.com. Instead, it's using an API to efficiently communicate with the platform.
* When you use the AWS CLI tools, they're actually communicating with AWS using the APIs. This process uses access keys to authenticate and IAM policies to authorize its one of IAMs primary functions to control access to the AWS API and now that you understand what an API is and what a microservice is you can start to understand that AWS is a product is actually made up of lots of APIs and microservices that provide each of the different product and product features.

**API Endpoints:**

An API endpoint **hosts one or more APIs and makes them available on a network (private or public internet).** APIs remain static — they are abstracted from what the code inside the service is doing. API consumers don't care how things are done — only that the interface works. That's what allows lower-risk changes.

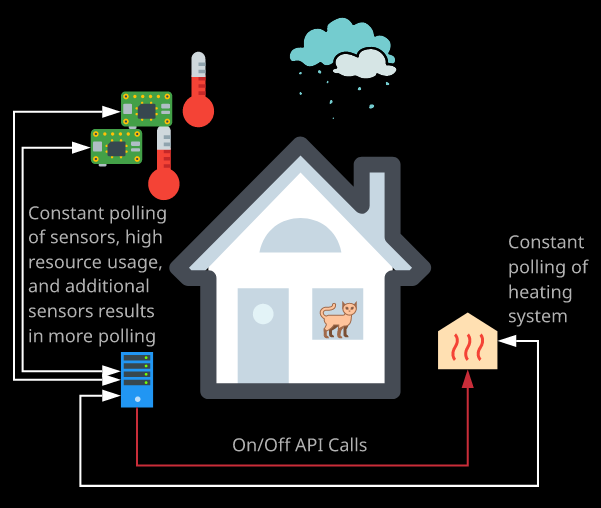
Now traditionally API endpoints have **come with a management overhead**. Generally these API endpoints run on physical or virtual servers, or even EC2 instances. So generally we still need to install and manage the operating system as well & need form of web server to host the API. But with AWS there are various ways that make this much easier and remove a lot of that management overhead

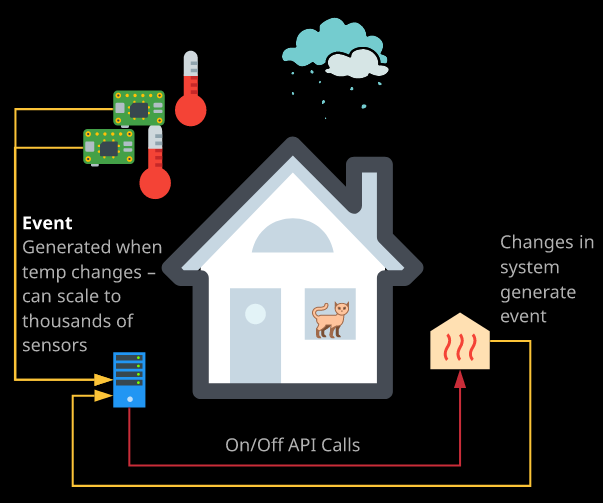
The AWS CLI tools use the AWS APIs.

**Event driven Architecture:**

When using an event-driven architecture, a system operates around "events" that represent an action or a change of state — e.g., a button being clicked, a file being uploaded, or a temperature dropping below a certain level.

It's efficient because events are generated and pushed, rather than things being polled.

****Polling requires always-on compute and doesn't scale well.

**Polling**

**Event Driven**

Using an event driven architecture allows for two things it allows for a much more efficient usage of resources because we don't have lots of entities polling for updates.

We have one entity generating these updates and sending them out, and it also allows for architectures to be developed when no compute resources are being consumed when no events are generated or received and this is one of the key benefits of the event driven architecture.

**Serverless:**

Serverless architecture has two main components.

**BaaS (or Backend as a Service)**: BaaS is an architecture or service model that prioritizes the consumption of third party services and you do that, rather than running your own services. This is essentially process whereby you as a consumer use all the services for functionality rather than managing long running compute processes of your own. means using third-party services where possible rather than running your own.

For example you might use **third party services authenticate your customers** so maybe auth0 or Cognito, which use third party ID providers such as Twitter or Google and by doing this you don't need to manage your own authentication systems or manage your own store of identities.

Another example is that you could utilize **third party database providers** such as Fir Base or Dynamodb to store and retrieve data and by doing so, you obviously don't need to run your own database infrastructure.

**Advantages of using BaaS:**

* Removes a lot of the long running infrastructure that your platform or application needs.
* Removes a lot of the management overhead, a lot of the risks, and alot of the costs that you would otherwise incur by running your own infrastructure.

**FaaS (or Function as a Service):** FaaS or function as a service, and this is a service model where you utilize short running or temporary compute rather than long running computer services like EC2 instances or containers.

With FaaS, you work with functions you give a function, which is a bit of code to a service, and it's executed as required.

This execution could be based on the schedule, so certain times of day or certain days of week or based on certain events happening.

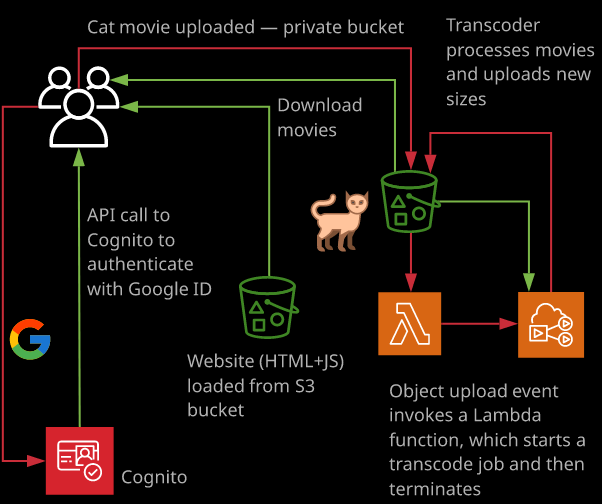
Event driven architecture supports serverless by supporting FaaS, with FaaS products you only pay for the time that your function is executing.

These functions are only active (invoked) when they are needed (when an event is received).

**Advantages of using FaaS:**

* With EC2 instances, you are paying for the EC2 instance to run for a certain amount of time but you're doing that regardless of whether any of your functions or any of your code are consuming the compute resources of that instance. With FaaS products, you only actually pay for the amount of time that your code is executing. So it is truly **on demand compute**.
* With FaaS, you don't have any other costs.
* No infrastructure to manage. No installation, no configuration, no maintenance and no patching. You give a function to a FaaS product, and it's executed when you want.
* Now functions can be used for API, and they could be run whenever a request comes into the API functions can also be run nightly toward alternate tasks in your infrastructure.
* Functions can be run on events maybe when an EC2 instances stopped, started, or terminated.

In AWS FaaS is provided by Lambda.



So with **serverless** both FaaS on BaaS are being used.

It's called serverless, not because there aren't any servers, but because you don't manage those servers and don't have to worry about them.

BaaS handles the backend by using third party services for databases, for authentication, and for much more and FaaS handles the logic of your application and only runs when needed.