AWS Storage

THE AWS CERTIFIED SOLUTIONS ARCHITECT ASSOCIATE EXAM OBJECTIVES COVERED IN THIS CHAPTER MAY INCLUDE, BUT ARE NOT LIMITED TO, THE FOLLOWING:

✓ **Domain 1: Design Resilient Architectures**

- 1.1 Design a multi-tier architecture solution
- 1.2 Design highly available and/or fault-tolerant architectures
- 1.4 Choose appropriate resilient storage

✓ **Domain 2: Design High-Performing Architectures**

- 2.1 Identify elastic and scalable compute solutions for a workload
- 2.2 Select high-performing and scalable storage solutions for a workload

✓ **Domain 3: Design Secure Applications and Architectures**

- 3.1 Design secure access to AWS resources
- 3.2 Design secure application tiers
- 3.3 Select appropriate data security options

✓ **Domain 4: Design Cost-Optimized Architectures**

- 4.1 Identify cost-effective storage solutions

Introduction

Amazon Simple Storage Service (S3) is where individuals, applications, and a long list of AWS services keep their data. It's an excellent platform for the following:

- Maintaining backup archives, log files, and disaster recovery images
- Running analytics on big data at rest
- Hosting static websites

S3 provides inexpensive and reliable storage that can, if necessary, be closely integrated with operations running within or external to AWS.

This isn't the same as the operating system volumes you learned about in the previous chapter; those are kept on the *block storage* volumes driving your EC2 instances. S3, by contrast, provides a space for effectively unlimited *object storage*.

What's the difference between object and block storage? With block-level storage, data on a raw physical storage device is divided into individual blocks whose use is managed by a file system. NTFS is a common filesystem used by Windows, and Linux might use Btrfs or ext4. The filesystem, on behalf of the installed OS, is responsible for allocating space for the files and data that are saved to the underlying device and for providing access whenever the OS needs to read some data.

An object storage system like S3, on the other hand, provides what you can think of as a flat surface on which to store your data. This simple design avoids some of the OS-related complications of block storage and allows anyone easy access to any amount of professionally designed and maintained storage capacity.

When you write files to S3, they're stored along with up to 2 KB of metadata. The metadata is made up of keys that establish system details like data permissions and the appearance of a filesystem location within nested buckets.

In this chapter, you're going to learn the following:

- How S3 objects are saved, managed, and accessed
- How to choose from among the various classes of storage to get the right balance of durability, availability, and cost
- How to manage long-term data storage lifecycles by incorporating Amazon Glacier into your design
- What other AWS services exist to help you with your data storage and access operations

S3 Service Architecture

You organize your S3 files into buckets. By default, you're allowed to create as many as 100 buckets for each of your AWS accounts. As with other AWS services, you can ask AWS to raise that limit.

Although an S3 bucket and its contents exist only within a single AWS region, the name you choose for your bucket must be globally unique within the entire S3 system. There's some logic to this; you'll often want your data located in a particular geographical region to satisfy operational or regulatory needs. But at the same time, being able to reference a bucket without having to specify its region simplifies the process.

Here is the URL you would use to access a file called `filename` that's in a bucket called `bucketname` over HTTP:

```
s3.amazonaws.com/bucketname/filename
```

Naturally, this assumes you'll be able to satisfy the object's permissions requirements.

This is how that same file would be addressed using the AWS CLI:

```
s3://bucketname/filename
```

Prefixes and Delimiters

As you've seen, S3 stores objects within a bucket on a flat surface without subfolder hierarchies. However, you can use prefixes and delimiters to give your buckets the *appearance* of a more structured organization.

A prefix is a common text string that indicates an organization level. For example, the word `contracts` when followed by the delimiter / would tell S3 to treat a file with a name like `contracts/acme.pdf` as an object that should be grouped together with a second file named `contracts/dynamic.pdf`.

S3 recognizes folder/directory structures as they're uploaded and emulates their hierarchical design within the bucket, automatically converting slashes to delimiters. That's why you'll see the correct folders whenever you view your S3-based objects through the console or the API.

Working with Large Objects

Although there's no theoretical limit to the total amount of data you can store within a bucket, a single object may be no larger than 5 TB. Individual *uploads* can be no larger than 5 GB. To reduce the risk of data loss or aborted uploads, AWS recommends that you use a feature called Multipart Upload for any object larger than 100 MB.

As the name suggests, Multipart Upload breaks a large object into multiple smaller parts and transmits them individually to their S3 target. If one transmission should fail, it can be repeated without impacting the others.

Multipart Upload will be used automatically when the upload is initiated by the AWS CLI or a high-level API, but you'll need to manually break up your object if you're working with a low-level API.

An application programming interface (API) is a programmatic interface through which operations can be run through code or from the command line. AWS maintains APIs as the primary method of administration for each of its services. AWS provides low-level APIs for cases when your S3 uploads require hands-on customization, and it provides high-level APIs for operations that can be more readily automated. This page contains specifics:

`docs.aws.amazon.com/AmazonS3/latest/dev/uploadobjusingmpu.html`

If you need to transfer large files to an S3 bucket, the Amazon S3 Transfer Acceleration configuration can speed things up. When a bucket is configured to use Transfer Acceleration, uploads are routed through geographically nearby AWS edge locations and, from there, routed using Amazon's internal network.

You can find out whether Transfer Acceleration would actually improve transfer speeds between your location and a particular AWS region by using the Amazon S3 Transfer Acceleration Speed Comparison tool (`s3-accelerate-speedtest.s3-accelerate.amazonaws.com/en/accelerate-speed-comparsion.html`). If your transfers are, in fact, good candidates for Transfer Acceleration, you should enable the setting in your bucket. You can then use special endpoint domain names (like `bucketname.s3-accelerate.amazonaws.com`) for your transfers.

Work through Exercise 3.1 and create your own bucket.

EXERCISE 3.1

Create a New S3 Bucket and Upload a File

1. From the AWS S3 dashboard, create a new bucket and use either the dashboard or the AWS CLI (`aws s3 cp mylocalfile.txt s3://mybucketname/`) to upload one or more files, giving public read access to the object.
2. From the file's Overview page in the S3 dashboard (displayed when the bucket name is clicked), copy its public link, paste it into the URL field of a browser that's not logged into your AWS account, and confirm that you can open the file.

Encryption

Unless it's intended to be publicly available—perhaps as part of a website—data stored on S3 should always be encrypted. You can use encryption keys to protect your data while it's at rest within S3 and—by using only Amazon's encrypted API endpoints for data transfers—protect data during its journeys between S3 and other locations.

Data at rest can be protected using either server-side or client-side encryption.

Server-Side Encryption

The “server-side” here is the S3 platform, and it involves having AWS encrypt your data objects as they’re saved to disk and decrypt them when you send properly authenticated requests for retrieval.

You can use one of three encryption options:

- Server-Side Encryption with Amazon S3-Managed Keys (SSE-S3), where AWS uses its own enterprise-standard keys to manage every step of the encryption and decryption process
- Server-Side Encryption with AWS KMS-Managed Keys (SSE-KMS), where, beyond the SSE-S3 features, the use of an *envelope key* is added along with a full audit trail for tracking key usage. You can optionally import your own keys through the AWS KMS service.
- Server-Side Encryption with Customer-Provided Keys (SSE-C), which lets you provide your own keys for S3 to apply to its encryption

Client-Side Encryption

It’s also possible to encrypt data before it’s transferred to S3. This can be done using an AWS KMS–Managed Customer Master Key (CMK), which produces a unique key for each object before it’s uploaded. You can also use a Client-Side Master Key, which you provide through the Amazon S3 encryption client.

Server-side encryption can greatly reduce the complexity of the process and is often preferred. Nevertheless, in some cases, your company (or regulatory oversight body) might require that you maintain full control over your encryption keys, leaving client-side as the only option.

Logging

Tracking S3 events to log files is disabled by default—S3 buckets can see a lot of activity, and not every use case justifies the log data that S3 can generate.

When you enable logging, you’ll need to specify both a source bucket (the bucket whose activity you’re tracking) and a target bucket (the bucket to which you’d like the logs saved). Optionally, you can also specify delimiters and prefixes (such as the creation date or time) to make it easier to identify and organize logs from multiple source buckets that are saved to a single target bucket.

S3-generated logs, which sometimes appear only after a short delay, will contain basic operation details, including the following:

- The account and IP address of the requestor
- The source bucket name

- The action that was requested (`GET`, `PUT`, `POST`, `DELETE`, etc.)
- The time the request was issued
- The response status (including error code)

S3 buckets are also used by other AWS services—including CloudWatch and Cloud-Trail—to store their logs or other objects (like EBS Snapshots).

S3 Durability and Availability

S3 offers more than one class of storage for your objects. The class you choose will depend on how critical it is that the data survives no matter what (durability), how quickly you might need to retrieve it (availability), and how much money you have to spend.

Durability

S3 measures durability as a percentage. For instance, the 99.999999999 percent durability guarantee for most S3 classes and Amazon Glacier is as follows:

> . . . corresponds to an average annual expected loss of 0.000000001% of objects. For example, if you store 10,000,000 objects with Amazon S3, you can on average expect to incur a loss of a single object once every 10,000 years.
>
> *Source:* `aws.amazon.com/s3/faqs`

In other words, realistically, there's pretty much no way that you can possibly lose data stored on one of the standard S3/Glacier platforms because of infrastructure failure. However, it would be irresponsible to rely on your S3 buckets as the only copies of important data. After all, there's a real chance that a misconfiguration, account lockout, or unanticipated external attack could permanently block access to your data. And, as crazy as it might sound right now, it's not unthinkable to suggest that AWS could one day go out of business. Kodak and Blockbuster Video once dominated their industries, right? You should always back up your data to multiple locations, using different services and media types. You'll learn how to do that in Chapter 10, "The Reliability Pillar." The high durability rates delivered by S3 are largely because they automatically replicate your data across at least three availability zones. This means that even if an entire AWS facility was suddenly wiped off the map, copies of your data would be restored from a different zone.

There is, however, one storage class that isn't quite so resilient. Amazon S3 Reduced Redundancy Storage (RRS) is rated at only 99.99 percent durability (because it's replicated across fewer servers than other classes). The RRS class is still available for historical reasons, but it's officially not recommended that you actually use it.

You can balance increased/decreased durability against other features like availability and cost to get the balance that's right for you. While all currently recommended classes are designed to withstand data loss 99.999999999% (11 nines) of the time, most will be maintained in at least three availability zones. The exception is S3 One Zone-IA that, as the name suggests, stores its data in only a single zone. The difference shows up in its slightly lower availability, which we'll discuss next.

Availability

Object availability is also measured as a percentage; this time, though, it's the percentage you can expect a given object to be instantly available on request through the course of a full year. The Amazon S3 Standard class, for example, guarantees that your data will be ready whenever you need it (meaning it will be available) for 99.99% of the year. That means there will be less than nine hours each year of downtime. If you feel downtime has exceeded that limit within a single year, you can apply for a service credit. Amazon's durability guarantee, by contrast, is designed to provide 99.999999999% data protection. This means there's practically no chance your data will be lost, even if you might sometimes not have instant access to it.

S3 Intelligent-Tiering is a relatively new storage class that saves you money while optimizing availability. For a monthly automation fee, Intelligent-Tiering will monitor the way you access data within the class over time. It will automatically move an object to the lower-cost infrequent access tier after it hasn't been accessed for 30 consecutive days.

Table 3.1 illustrates the availability guarantees for all S3 classes.

TABLE 3.1 Guaranteed availability standards for S3 storage

	S3 Standard	S3 Standard-IA	S3 One Zone-IA	S3 Intelligent-Tiering
Availability guarantee	99.99%	99.9%	99.5%	99.9%

Eventually Consistent Data

It's important to bear in mind that S3 replicates data across multiple locations. As a result, there might be brief delays while updates to existing objects propagate across the system. Uploading a new version of a file or, alternatively, deleting an old file altogether can result in one site reflecting the new state with another still unaware of any changes.

To ensure that there's never a conflict between versions of a single object—which could lead to serious data and application corruption—you should treat your data according to an eventually consistent standard. That is, you should expect a delay (usually just two seconds or less) and design your operations accordingly.

Because there isn't the risk of corruption when creating *new* objects, S3 provides read-after-write consistency for creation (PUT) operations.

S3 Object Lifecycle

Many of the S3 workloads you'll launch will probably involve backup archives. But the thing about backup archives is that, when properly designed, they're usually followed regularly by more backup archives. Maintaining some previous archive versions is critical, but you'll also want to retire and delete older versions to keep a lid on your storage costs.

S3 lets you automate all this with its versioning and lifecycle features.

Versioning

Within many file system environments, saving a file using the same name and location as a preexisting file will overwrite the original object. That ensures you'll always have the most recent version available to you, but you will lose access to older versions—including versions that were overwritten by mistake.

By default, objects on S3 work the same way. But if you enable versioning at the bucket level, then older overwritten copies of an object will be saved and remain accessible indefinitely. This solves the problem of accidentally losing old data, but it replaces it with the potential for archive bloat. Here's where lifecycle management can help.

Lifecycle Management

In addition to the S3 Intelligent-Tiering storage class we discussed earlier, you can manually configure lifecycle rules for a bucket that will automatically transition an object's storage class after a set number of days. You might, for instance, have new objects remain in the S3 Standard class for their first 30 days, after which they're moved to the cheaper One Zone IA for another 30 days. If regulatory compliance requires that you maintain older versions, your files could then be moved to the low-cost, long-term storage service Glacier for 365 more days before being permanently deleted.

NOTE

You can optionally use prefixes to apply lifecycle rules to only certain objects within a bucket. You should also be aware that there are minimum times (30 days, for instance) an object must remain within one class before it can be moved. In addition, you can't transition directly from S3 Standard to Reduced Redundancy.

Try it yourself with Exercise 3.2.

EXERCISE 3.2

Enable Versioning and Lifecycle Management for an S3 Bucket

1. Select your bucket and edit its properties to enable versioning.
2. Upload a file to that bucket, edit the copy on your local computer, and upload the new copy (keeping the filename the same). If you're working with a file from your static website, make sure you give the new file permissions allowing public access.
3. With the contents of the bucket displayed in the dashboard, select Show Versions. You should now see two versions of your file.
4. Add a couple of directories with files to your bucket.
5. From the bucket's Management tab, select Lifecycle and specify a prefix/tag filter that matches the directory name of one of the directories you uploaded.
6. Add a lifecycle rule by adding transitions and configuring the transition timing (in days) and target for each one.

You'll need to be patient to test this configuration because the minimum lag between transitions is 30 days.

Accessing S3 Objects

If you didn't think you'd ever need your data, you wouldn't go to the trouble of saving it to S3. So, you'll need to understand how to access your S3-hosted objects and, just as important, how to restrict access to only those requests that match your business and security needs.

Access Control

Out of the box, new S3 buckets and objects will be fully accessible to your account but to no other AWS accounts or external visitors. You can strategically open up access at the bucket and object levels using access control list (ACL) rules, finer-grained S3 bucket policies, or Identity and Access Management (IAM) policies.

There is more than a little overlap between those three approaches. In fact, ACLs are really leftovers from before AWS created IAM. As a rule, Amazon recommends applying S3 bucket policies or IAM policies instead of ACLs.

S3 bucket policies—which are formatted as JavaScript Object Notation (JSON) text and attached to your S3 bucket—will make sense for cases where you want to control access to a single S3 bucket for multiple external accounts and users. On the other hand, IAM policies—because they exist at the account level within IAM—will probably make sense when you're trying to control the way individual users and roles access multiple resources, including S3.

The following code is an example of an S3 bucket policy that allows both the root user and the user Steve from the specified AWS account to access the S3 `MyBucket` bucket and its contents. Both users are considered *principals* within this rule.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": ["arn:aws:iam::xxxxxxxxxxxx:root",
        "arn:aws:iam::xxxxxxxxxxxx:user/Steve"]
      },
      "Action": "s3:*",
      "Resource": ["arn:aws:s3:::MyBucket",
                   "arn:aws:s3:::MyBucket/*"]
    }
  ]
}
```

NOTE You can also limit access within bucket policies by specifying time of day or source Classless Inter-Domain Routing (CIDR) IP address blocks.

When it's attached to an IAM entity (a user, group, or role), the following IAM policy will accomplish the same thing as the previous S3 bucket policy:

```
{
  "Version": "2012-10-17",
  "Statement":[{
    "Effect": "Allow",
    "Action": "s3:*",
    "Resource": ["arn:aws:s3:::MyBucket",
                 "arn:aws:s3:::MyBucket/*"]
    }
  ]
}
```

IAM roles and policies will be discussed in greater detail in Chapter 6, "Authentication and Authorization—AWS Identity and Access Management."

You can also closely control the way users and services access objects within your buckets by using Amazon S3 Access Points. An access point is a hostname that can point to a carefully defined subset of objects in a bucket. Depending on how you configure your access points, clients invoking the hostname will be able to read or write only the data you allow, and only as long as you allow it.

A simple AWS CLI command to request an access point might look something like this:

```
aws s3control create-access-point --name my-vpc-ap \
   --account-id 123456789012 --bucket my-bucket \
   --vpc-configuration VpcId=vpc-2b9d3c
```

Presigned URLs

If you want to provide temporary access to an object that's otherwise private, you can generate a presigned URL. The URL will be usable for a specified period of time, after which it will become invalid. You can build presigned URL generation into your code to provide object access programmatically.

The following AWS CLI command will return a URL that includes the required authentication string. The authentication will become invalid after 10 minutes (600 seconds). The default expiration value is 3,600 seconds (one hour).

```
aws s3 presign s3://MyBucketName/PrivateObject --expires-in 600
```

Try it yourself with Exercise 3.3.

EXERCISE 3.3

Generate and Use a Presigned URL

1. Use the complete URL of a private object in an S3 bucket belonging to you to generate a presigned URL using a variation of this command:

   ```
   aws s3 presign s3://MyBucketName/PrivateObject --expires-in 600
   ```

2. Copy the full URL from the command output and, from a browser that's not logged into your AWS account, try to open the file.
3. Wait for the URL to expire and try again. This time, it should not work.

Static Website Hosting

S3 buckets can be used to host the HTML files for entire static websites. A website is static when the system services used to render web pages and scripts are all client- rather than server-based. This architecture permits simple and lean HTML code that's designed to be executed by the client browser.

S3, because it's such an inexpensive yet reliable platform, is an excellent hosting environment for such sites. When an S3 bucket is configured for static hosting, traffic directed at the bucket's URL can be automatically made to load a specified root document, usually named `index.html`. Users can click links within HTML pages to be sent to the target page or media resource. Error handling and redirects can also be incorporated into the profile.

If you want requests for a DNS domain name (like `mysite.com`) routed to your static site, you can use Amazon Route 53 to associate your bucket's endpoint with any registered name. This will work only if your domain name is also the name of the S3 bucket. You'll learn more about domain name records in Chapter 8, "The Domain Name System and Network Routing: Amazon Route 53 and Amazon CloudFront."

You can also get a free SSL/TLS certificate to encrypt your site by requesting a certificate from AWS Certificate Manager (ACM) and importing it into a CloudFront distribution that specifies your S3 bucket as its origin.

Build your own static website using Exercise 3.4.

EXERCISE 3.4

Enable Static Website Hosting for an S3 Bucket

1. From the S3 dashboard, select (or create) an S3 bucket that contains at least one file with some simple text named `index.html`. Any files you want to be accessible should be readable by the public.
2. From the bucket Properties tab, enable static website hosting and specify your `index.html` file as your index document.
3. Paste the static website endpoint into the URL field of a browser that's not logged into your AWS account and confirm you can access the website.

Note that you can also enable static website hosting from the AWS CLI using a variation of these two commands:

```
aws s3api put-bucket-acl --bucket my-bucket --acl public-read
aws s3 website s3://my-bucket/ --index-document index.html --error-document
error.html
```

AWS provides a different way to access data stored on either S3 Select or Glacier Select. The feature lets you apply SQL-like queries to stored objects so that only relevant data from within objects is retrieved, permitting significantly more efficient and cost-effective operations.

One possible use case would involve large comma-separated values (CSV) files containing sales and inventory data from multiple retail sites. Your company's marketing team might need to periodically analyze only sales data and only from certain stores. Using S3 Select, they'll be able to retrieve exactly the data they need—just a fraction of the full data set—while bypassing the bandwidth and cost overhead associated with downloading the whole thing.

Amazon S3 Glacier

At first glance, Glacier looks a bit like just another S3 storage class. After all, like most S3 classes, Glacier guarantees 99.999999999 percent durability and, as you've seen, can be incorporated into S3 lifecycle configurations.

Nevertheless, there are important differences. Glacier, for example, supports archives as large as 40 TB, whereas S3 buckets have no size limit. Its archives are encrypted by default, whereas encryption on S3 is an option you need to select. And unlike S3's "human-readable" key names, Glacier archives are given machine-generated IDs.

But the biggest difference is the time it takes to retrieve your data. Retrieving the objects from an existing Glacier archive can take a number of hours, compared to nearly instant access from S3. That last feature really defines the purpose of Glacier: to provide inexpensive long-term storage for data that will be needed only in unusual and infrequent circumstances.

NOTE Glacier does offer Expedited retrievals, getting you your data in minutes rather than hours. Expedited will incur a premium charge.

In the context of Glacier, the term *archive* is used to describe an object like a document, video, or a TAR or ZIP file. Archives are stored in *vaults*—the Glacier equivalent of S3's buckets. Glacier vault names do not have to be globally unique.

There are currently two Glacier storage tiers: Standard and Deep Archive. As you can probably guess, Glacier Deep Archive will cost you less but will require longer waits for data retrieval. Storing one terabyte of data for a month on Glacier standard will, at current rates, cost you $4.10, while leaving the same terabyte for the same month on Glacier Deep Archive will cost only $1.02. Retrieval from Deep Archive, however, will take between 12 and 48 hours.

Table 3.2 lets you compare the costs of retrieving 100 GB of data from Glacier using each of its five retrieval tiers.

TABLE 3.2 Sample retrieval costs for Glacier data in the US East region

Tier	Amount retrieved	Cost
Glacier Standard	100 GB	$0.90
Glacier Expedited	100 GB	$3.00
Glacier Bulk	100 GB	$0.25
Deep Archive Standard	100 GB	$2.00

Storage Pricing

To give you a sense of what S3 and Glacier might cost you, here's a typical usage scenario. Imagine you make weekly backups of your company sales data that generate 5 GB archives. You decide to maintain each archive in the S3 Standard Storage and Requests class for its first 30 days and then convert it to S3 One Zone (S3 One Zone-IA), where it will remain for 90 more days. At the end of those 120 days, you will move your archives once again, this time to Glacier, where they will be kept for another 730 days (two years) and then deleted.

Once your archive rotation is in full swing, you'll have a steady total of (approximately) 20 GB in S3 Standard, 65 GB in One Zone-IA, and 520 GB in Glacier. Table 3.3 shows what that storage will cost in the US East region at rates current as of this writing.

TABLE 3.3 Sample storage costs for data in the US East region

Class	Storage amount	Rate/GB/month	Cost/month
Standard	20 GB	$0.023	$0.46
One Zone-IA	65 GB	$0.01	$0.65
Glacier	520 GB	$0.004	$2.08
Total			$3.19

Of course, storage is only one part of the mix. You'll also be charged for operations including data retrievals; `PUT`, `COPY`, `POST`, or `LIST` requests; and lifecycle transition requests. Full, up-to-date details are available at `aws.amazon.com/s3/pricing`.

Exercise 3.5 will introduce you to an important cost-estimating tool.

EXERCISE 3.5

Calculate the Total Lifecycle Costs for Your Data

Use the AWS Simple Monthly Calculator (`calculator.s3.amazonaws.com/index.html`) to estimate the total monthly costs of the scenario described at the beginning of this section. Even better, use a scenario that fits your own business. Try to include a full usage scenario, including requests, scans, and data retrieval. Note that you access the S3 part of the calculator by clicking the Amazon S3 tab on the left, and you can keep track of your itemized estimate using the Estimate Of Your Monthly Bill tab along the top.

Other Storage-Related Services

It's worth being aware of some other storage-related AWS services that, while perhaps not as common as the others you've seen, can make a big difference for the right deployment.

Amazon Elastic File System

The Elastic File System (EFS) provides automatically scalable and shareable file storage to be accessed from Linux instances. EFS-based files are designed to be accessed from within a virtual private cloud (VPC) via Network File System (NFS) mounts on EC2 Linux instances or from your on-premises servers through AWS Direct Connect connections. The goal is to make it easy to enable secure, low-latency, and durable file sharing among multiple instances.

Amazon FSx

Amazon FSx comes in two flavors: FSx for Lustre and Amazon FSx for Windows File Server. Lustre is an open source distributed filesystem built to give Linux clusters access to high-performance filesystems for use in compute-intensive operations. Amazon's FSx service brings Lustre capabilities to your AWS infrastructure.

FSx for Windows File Server, as you can tell from the name, offers the kind of file-sharing service EFS provides but for Windows servers rather than Linux. FSx for Windows File Server integrates operations with Server Message Block (SMB), NTFS, and Microsoft Active Directory.

AWS Storage Gateway

Integrating the backup and archiving needs of your local operations with cloud storage services can be complicated. AWS Storage Gateway provides software gateway appliances (based on an on-premises hardware appliance or virtual machines built on VMware ESXi, Microsoft Hyper-V, Linux KVM, VMware Cloud on AWS, or EC2 images) with multiple virtual connectivity interfaces. Local devices can connect to the appliance as though it's a physical backup device like a tape drive, and the data itself is saved to AWS platforms like S3 and EBS. Data can be maintained in a local cache to make it locally available.

AWS Snowball

Migrating large data sets to the cloud over a normal Internet connection can sometimes require far too much time and bandwidth to be practical. If you're looking to move terabyte- or even petabyte-scaled data for backup or active use within AWS, ordering a Snowball device might be the best option.

When requested, AWS will ship you a physical, 256-bit, encrypted Snowball storage device onto which you'll copy your data. You then ship the device back to Amazon, where its data will be uploaded to your S3 bucket(s).

Choosing the best method for transferring data to your AWS account will require a bit of arithmetic. You'll have to know the real-world upload speeds you get from your Internet connection and how much of that bandwidth wouldn't be used by other operations. In this chapter, you've learned about Multipart Upload and Transfer Acceleration for moving larger objects into an S3 bucket. But some objects are just so large that uploading them over your existing Internet connection isn't practical. Think about it; if your Internet service provider (ISP) gives you 10 MB/second, then, assuming no one else is using the connection, uploading a one-terabyte archive would take you around 10 days!

So, if you really need to move that data to the cloud, you're going to have to either invest in an expensive AWS Direct Connect connection or introduce yourself to an AWS Snowball device (or, for really massive volumes of data, AWS Snowmobile at `aws.amazon.com/snowmobile`).

AWS DataSync

DataSync specializes in moving on-premises data stores into your AWS account with a minimum of fuss. It works over your regular Internet connection, so it's not as useful as Snowball for really large data sets. But it is much more flexible, since you're not limited to S3 (or RDS as you are with AWS Database Migration Service). Using DataSync, you can drop your data into any service within your AWS account. That means you can do the following:

- Quickly and securely move old data out of your expensive data center into cheaper S3 or Glacier storage.
- Transfer data sets directly into S3, EFS, or FSx, where it can be processed and analyzed by your EC2 instances.
- Apply the power of any AWS service to any class of data as part of an easy-to-configure automated system.

DataSync can handle transfer rates of up to 10 Gbps (assuming your infrastructure has that capacity) and offers both encryption and data validation.

AWS CLI Example

This example will use the AWS CLI to create a new bucket and recursively copy the `sales-docs` directory to it. Then, using the low-level s3api CLI (which should have been installed along with the regular AWS CLI package), you'll check for the current lifecycle configuration of your new bucket with the `get-bucket-lifecycle-configuration` subcommand, specifying your bucket name. This will return an error, of course, since there currently is no configuration.

Next, you'll run the `put-bucket-lifecycle-configuration` subcommand, specifying the bucket name. You'll also add some JSON code to the `--lifecycle-configuration` argument. The code (which could also be passed as a file) will transition all objects using the `sales-docs` prefix to the Standard-IA after 30 days and to Glacier after 60 days. The objects will be deleted (or "expire") after a full year (365 days).

Finally, you can run `get-bucket-lifecycle-configuration` once again to confirm that your configuration is active. Here are the commands you would need to run to make all this work:

```
$ aws s3 mb s3://bucket-name
$ aws s3 cp --recursive sales-docs/ s3://bucket-name
$ aws s3api get-bucket-lifecycle-configuration \
   --bucket bucket-name
$ aws s3api put-bucket-lifecycle-configuration \
   --bucket bucket-name \
   --lifecycle-configuration '{
    "Rules": [
        {
            "Filter": {
                "Prefix": "sales-docs/"
            },
            "Status": "Enabled",
            "Transitions": [
                {
                    "Days": 30,
                    "StorageClass": "STANDARD_IA"
                },
                {
                    "Days": 60,
                    "StorageClass": "GLACIER"
                }
            ],
            "Expiration": {
                "Days": 365
            },
            "ID": "Lifecycle for bucket objects."
        }
    ]
}'
$ aws s3api get-bucket-lifecycle-configuration \
   --bucket bucket-name
```

Summary

Amazon S3 provides reliable and highly available object-level storage for low-maintenance, high-volume archive and data storage. Objects are stored in buckets on a "flat" surface. However, through the use of prefixes, objects can be made to appear as though they're part of a normal filesystem.

You can—and usually should—encrypt your S3 data using either AWS-provided or self-serve encryption keys. Encryption can take place when your data is at rest using either server-side or client-side encryption.

There are multiple storage classes within S3 relying on varying degrees of data replication that allow you to balance durability, availability, and cost. Lifecycle management lets you automate the transition of your data between classes until it's no longer needed and can be deleted.

You can control who and what get access to your S3 buckets—and when—through legacy ACLs or through more powerful S3 bucket policies and IAM policies. Presigned URLs are also a safe way to allow temporary and limited access to your data.

You can reduce the size and cost of your requests against S3 and Glacier-based data by leveraging the SQL-like Select feature. You can also provide inexpensive and simple static websites through S3 buckets.

Amazon Glacier stores your data archives in vaults that might require hours to retrieve but that cost considerably less than the S3 storage classes.

Amazon Virtual Private Cloud

THE AWS CERTIFIED SOLUTIONS ARCHITECT ASSOCIATE EXAM OBJECTIVES COVERED IN THIS CHAPTER MAY INCLUDE, BUT ARE NOT LIMITED TO, THE FOLLOWING:

✓ **Domain 1: Design Resilient Architectures**

- 1.3 Design decoupling mechanisms using AWS services

✓ **Domain 3: Design Secure Applications and Architectures**

- 3.1 Design secure access to AWS resources
- 3.3 Select appropriate data security options

Introduction

Amazon's Virtual Private Cloud service provides the networking layer of EC2. A VPC is a virtual network that can contain EC2 instances as well as network resources for other AWS services. By default, every virtual private cloud (VPC) is isolated from all other networks. You can, however, connect your VPC to other networks, including the Internet and other VPCs.

In addition to EC2, VPCs are foundational to many AWS services, so understanding how they work is fundamental to your success on the exam and as an AWS architect. Don't assume you can ignore VPCs just because you're not using EC2.

A VPC can exist only within an AWS region. When you create a VPC in a region, it won't show up in any other regions. You can have multiple VPCs in your account and create multiple VPCs in a single region. To keep things simple, we'll start by assuming only one VPC in one region. Later, we'll cover considerations for multiple VPCs.

If you're familiar with the components of a traditional network, you'll recognize many VPC components. But although VPCs function like a traditional TCP/IP network, they are scalable, allowing you to expand and extend your network without having to add physical hardware. To make this scalability possible, some components that you'd find in a traditional network—such as routers, switches, and VLANs—don't exist in VPCs. Instead, they're abstracted into software functions and called by different names.

VPC CIDR Blocks

Like a traditional network, a VPC consists of at least one range of contiguous IP addresses. This address range is represented as a *Classless Inter-Domain Routing* (CIDR) block. The CIDR block determines which IP addresses may be assigned to instances and other resources within the VPC. You must assign a primary CIDR block when creating a VPC. After creating a VPC, you divide the primary VPC CIDR block into subnets that hold your AWS resources.

There are different ways to represent a range of IP addresses. The shortest way is by CIDR notation, sometimes called *slash notation*. For example, the CIDR 172.16.0.0/16 includes all addresses from 172.16.0.0 to 172.16.255.255—a total of 65,536 addresses!

You may also hear the CIDR block referred to as an *IP prefix*. The /16 portion of the CIDR is the *prefix length*. The prefix length refers to the length of the subnet mask, which in the case of a VPC CIDR can range from /16 to /28. An inverse relationship exists between the prefix length and the number of IP addresses in the CIDR. The smaller the prefix length, the greater the number of IP addresses in the CIDR. A /28 prefix length gives you only 16 possible addresses, whereas a /16 prefix length gives you 65,536 possible addresses!

The acronym IP refers to Internet Protocol version 4 (IPv4). Valid IPv4 prefix lengths range from /0 to /32. Although you can specify any valid IP range for your VPC CIDR, it's best to use one in the following RFC 1918 (`tools.ietf.org/rfc/rfc1918.txt`) range to avoid conflicts with public Internet addresses.

- 10.0.0.0–10.255.255.255 (10.0.0.0/8)
- 172.16.0.0–172.31.255.255 (172.16.0.0/12)
- 192.168.0.0–192.168.255.255 (192.168.0.0/16)

If you plan to connect your VPC to another network—whether an on-premises network or another VPC—be sure the VPC CIDR you choose doesn't overlap with addresses already in use on the other network.

You can't change the primary CIDR block after you create your VPC, so think carefully about your address requirements before creating a VPC.

Secondary CIDR Blocks

You may optionally specify secondary CIDR blocks for a VPC after you've created it. These blocks must come from either the same address range as the primary or a publicly routable range, but they must not overlap with the primary or other secondary blocks. For example, if the VPC's primary CIDR is 172.16.0.0/16, you may specify a secondary CIDR of 172.17.0.0/16 because it resides in the 172.16.0.0/12 range (172.16.0.0–172.31.255.255). But you may not specify 192.168.0.0/16 as a secondary CIDR.

If you think you might ever need a secondary CIDR, be careful about your choice of primary CIDR. If you choose 192.168.0.0/16 as your primary CIDR, you won't be able to create a secondary CIDR using any of the other RFC 1918 ranges.

IPv6 CIDR Blocks

You may let AWS assign an IPv6 CIDR to your VPC. Unlike the primary CIDR, which is an IP prefix you choose, you can't choose your own IPv6 CIDR. Instead, AWS assigns one to your VPC at your request. The IPv6 CIDR will be a publicly routable prefix from the global unicast IPv6 address space, so all IPv6 addresses are reachable from the Internet. For example, AWS may assign you the CIDR 2600:1f18:2551:8900/56. Note that the prefix length of an IPv6 VPC CIDR is always /56.

Complete Exercise 4.1 to create your own VPC.

EXERCISE 4.1

Create a New VPC

Create a VPC with the primary CIDR 172.16.0.0/16.

To complete this exercise using the AWS Command Line Interface, enter the following command:

EXERCISE 4.1 *(continued)*

```
aws ec2 create-vpc
--cidr-block 172.16.0.0/16
```

If the command succeeds, you should see output similar to the following:

```
{
    "Vpc": {
        "CidrBlock": "172.16.0.0/16",
        "DhcpOptionsId": "dopt-21500a47",
        "State": "pending",
        "VpcId": "vpc-0d19e8153b4d142ed",
        "OwnerId": "158826777352",
        "InstanceTenancy": "default",
        "Ipv6CidrBlockAssociationSet": [],
        "CidrBlockAssociationSet": [
            {
                "AssociationId": "vpc-cidr-assoc-0a99f2470e29710b7",
                "CidrBlock": "172.16.0.0/16",
                "CidrBlockState": {
                    "State": "associated"
                }
            }
        ],
        "IsDefault": false,
        "Tags": []
    }
}
```

Note that the VPC is initially in the pending state as AWS creates it. To view its state, note the VPC identifier and issue the following command:

```
aws ec2 describe-vpcs --vpc-ids [vpc-id]
```

You should see output similar to the following:

```
{
    "Vpcs": [
        {
            "CidrBlock": "172.16.0.0/16",
            "DhcpOptionsId": "dopt-21500a47",
            "State": "available",
                "VpcId": "vpc-0d19e8153b4d142ed",
                "OwnerId": "158826777352",
                "InstanceTenancy": "default",
                "CidrBlockAssociationSet": [
                    {
                        "AssociationId": "vpc-cidr-assoc-0a99f2470e29710b7",
                        "CidrBlock": "172.16.0.0/16",
                        "CidrBlockState": {
                            "State": "associated"
                        }
                    }
                ],
                "IsDefault": false
            }
        ]
    }
```

Your new VPC is now ready to use. Keep it handy, as you'll be using it in subsequent exercises.

Subnets

A subnet is a logical container within a VPC that holds VPC resources, including your EC2 instances. A subnet lets you isolate instances from each other, control how traffic flows to and from your instances, and organize them by function. For example, you can create one subnet for public web servers that need to be accessible from the Internet and create another subnet for database servers that only the web instances can access. In concept, subnets are similar to virtual LANs (VLANs) in a traditional network.

Every instance must exist within a subnet. You'll often hear the phrase "launch an instance into a subnet." After you create an instance in a subnet, you can't move the instance. By extension, this also means you can't move an instance from one VPC to another. You can, however, terminate it and create a different instance in another subnet. To preserve the data on the instance's EBS volume, you can snapshot the volume, create an AMI, and then use the AMI to create a new instance in another subnet.

Subnet CIDR Blocks

From the VPC CIDR block, you carve out a smaller CIDR block for each subnet. For example, if your VPC has a CIDR of 172.16.0.0/16, one of your subnets may have a CIDR of 172.16.100.0/24. This range includes all IP addresses from 172.16.100.0 to 172.16.100.255, which yields a total of 256 addresses.

AWS reserves the first four and last IP addresses in every subnet. You can't assign these addresses to any instances. Assuming a subnet CIDR of 172.16.100.0/24, the following addresses would be reserved:

- 172.16.100.0
- 172.16.100.1–Implied router
- 172.16.100.2–Amazon-provided DNS server
- 172.16.100.3–Reserved
- 172.16.100.255

The restrictions on prefix lengths for a subnet CIDR are the same as VPC CIDRs. Subnet CIDR blocks in a single VPC can't overlap with each other. Also, once you assign an IP prefix to a subnet, you can't change it.

It's possible for a subnet and VPC to share the same CIDR. For example, you might assign the CIDR 192.168.0.0/16 to both a VPC and a subnet within the VPC. This is uncommon and won't leave you room for additional subnets, which means your VPC will be effectively limited to one availability zone. We explain availability zones in the next section. More commonly, each subnet's prefix length will be longer than the VPC's CIDR block to allow for multiple subnets to exist in the same VPC. For example, if you assign the CIDR 192.168.0.0/16 to a VPC, you might assign the CIDR 192.168.3.0/24 to a subnet within that VPC, leaving room for additional subnets.

A subnet can't have multiple CIDRs. Unlike a VPC that can have secondary CIDRs, a subnet can have only one. However, if a VPC has a primary CIDR and a secondary CIDR, your subnet's CIDR can be derived from either. For example, if your VPC has the primary CIDR of 172.16.0.0/16 and a secondary CIDR of 172.17.0.0/16, a subnet in that VPC could be 172.17.12.0/24, as it's derived from the secondary VPC CIDR.

Availability Zones

A subnet can exist within only one *availability zone* (AZ, or *zone* for short), which is roughly analogous to a relatively small geographic location such as a data center. Although availability zones in an AWS region are connected, they are designed so that a failure in one zone doesn't cause a failure in another.

You can achieve resiliency for your applications by creating two subnets each in a different availability zone and then spreading your instances across those zones. Table 4.1 provides an example of two subnets in different availability zones.

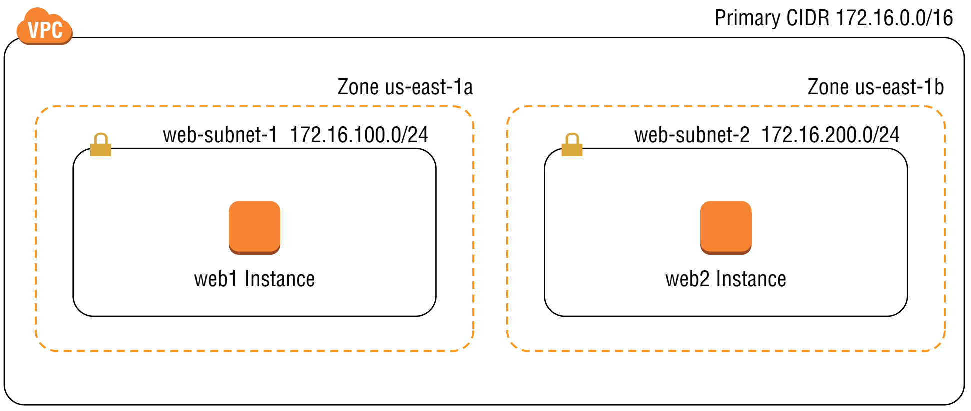
TABLE 4.1 Subnets in different availability zones

Subnet	Availability zone	Instance
`web-subnet1`	`us-east-1a`	`web1`
`web-subnet2`	`us-east-1b`	`web2`

In this example, if the `us-east-1a` zone fails, the instance `web1` will fail because it's in that zone. But `web2`, which is in the `us-east-1b` zone, will remain available.

Having subnets in different availability zones is not a requirement. You can place all of your subnets in the same zone, but keep in mind that if that zone fails, all instances in those subnets will fail as well. Refer to Figure 4.1 for an example of two instances in different availability zones.

FIGURE 4.1 VPC with subnets and instances


VPC
Primary CIDR 172.16.0.0/16
Zone us-east-1a
Zone us-east-1b
web-subnet-1 172.16.100.0/24
web-subnet-2 172.16.200.0/24
web1 Instance
web2 Instance

Complete Exercise 4.2 to practice creating a subnet.

EXERCISE 4.2

Create a New Subnet

Create a subnet in the VPC you created earlier. Choose an availability zone and assign the CIDR block 172.16.100.0/24.

To complete this using the following AWS Command Line Interface commands, you'll need the resource ID of the VPC:

```
aws ec2 create-subnet
--vpc-id [vpc-id]
--cidr-block 172.16.100.0/24
--availability-zone us-east-1a
```

You should see output similar to the following:

```
{
    "Subnet": {
        "AvailabilityZone": "us-east-1a",
```

EXERCISE 4.2 *(continued)*

```
            "AvailabilityZoneId": "use1-az2",
            "AvailableIpAddressCount": 251,
            "CidrBlock": "172.16.100.0/24",
            "DefaultForAz": false,
            "MapPublicIpOnLaunch": false,
            "State": "pending",
            "SubnetId": "subnet-0e398e93c154e8757",
            "VpcId": "vpc-0d19e8153b4d142ed",
            "OwnerId": "158826777352",
            "AssignIpv6AddressOnCreation": false,
            "Ipv6CidrBlockAssociationSet": [],
            "SubnetArn": "arn:aws:ec2:us-east-1:158826777352:subnet/subnet-
                    0e398e93c154e8757"
        }
```

Wait a few moments, and then check the status of the subnet as follows:

```
aws ec2 describe-subnets --subnet-ids [subnet-id]
```

You should see the subnet in an `available` state, like so:

```
{
    "Subnets": [
        {
            "AvailabilityZone": "us-east-1a",
            "AvailabilityZoneId": "use1-az2",
            "AvailableIpAddressCount": 251,
            "CidrBlock": "172.16.100.0/24",
            "DefaultForAz": false,
            "MapPublicIpOnLaunch": false,
            "State": "available",
            "SubnetId": "subnet-0e398e93c154e8757",
            "VpcId": "vpc-0d19e8153b4d142ed",
            "OwnerId": "158826777352",
            "AssignIpv6AddressOnCreation": false,
            "Ipv6CidrBlockAssociationSet": [],
            "SubnetArn": "arn:aws:ec2:us-east-1:158826777352:subnet/subnet-
                    0e398e93c154e8757"
        }
    ]
}
```

IPv6 CIDR Blocks

If you've allocated an IPv6 CIDR to your VPC, you can assign IPv6 CIDRs to subnets within that VPC. The prefix length for an IPv6 subnet is fixed at /64. If your VPC's IPv6 CIDR is 2600:1f18:2551:8900/56, you can assign a subnet the CIDR 2600:1f18:2551:89*00*/64 through 2600:1f18:2551:89*FF*/64.

NOTE You must always assign an IPv4 CIDR block to a subnet, even if you intend to use only IPv6.

Elastic Network Interfaces

An *elastic network interface* (ENI) allows an instance to communicate with other network resources, including AWS services, other instances, on-premises servers, and the Internet. It also makes it possible for you to use Secure Shell (SSH) or Remote Desktop Protocol (RDP) to connect to the operating system running on your instance to manage it. As the name suggests, an ENI performs the same basic function as a network interface on a physical server, although ENIs have more restrictions on how you can configure them.

Every instance must have a *primary network interface* (also known as the *primary ENI*), which is connected to only one subnet. This is the reason you have to specify a subnet when launching an instance. You can't remove the primary ENI from an instance, and you can't change its subnet.

Primary and Secondary Private IP Addresses

Each instance must have a *primary private IP address* from the range specified by the subnet. The primary private IP address is bound to the primary ENI of the instance. You can't change or remove this address, but you can assign secondary private IP addresses to the primary ENI. Any secondary addresses must come from the same subnet that the ENI is attached to.

It's possible to attach additional ENIs to an instance. Those ENIs may be in a different subnet, but they must be in the same availability zone as the instance. As always, any addresses assigned to the ENI must come from the subnet to which it is attached.

Attaching Elastic Network Interfaces

An ENI can exist independently of an instance. You can create an ENI first and then attach it to an instance later. For example, you can create an ENI in one subnet and then attach it to an instance in the same subnet as the primary ENI when you launch the instance. If you disable the Delete On Termination attribute of the ENI, you can terminate the instance without deleting the ENI. You can then associate the ENI with another instance.

You can also take an existing ENI that's not attached to an instance and attach it to an existing instance as a secondary ENI. This lets you redirect traffic from a failed instance to a working instance by detaching the ENI from the failed instance and reattaching it to the working instance. Complete Exercise 4.3 to practice creating an ENI and attaching it to an instance.

EXERCISE 4.3

Create and Attach a Primary ENI

Create an ENI in the subnet of your choice.

Using the subnet ID from the previous exercise, issue the following AWS CLI command:

```
aws ec2 create-network-interface
--private-ip-address 172.16.100.99
--subnet-id [subnet-id]
```

You should see output similar to the following:

```
{
    "NetworkInterface": {
        "AvailabilityZone": "us-east-1a",
        "Description": "",
        "Groups": [
            {
                "GroupName": "default",
                "GroupId": "sg-011c3fe63d256f5d9"
            }
        ],
        "InterfaceType": "interface",
        "Ipv6Addresses": [],
        "MacAddress": "12:6a:74:e1:95:a9",
        "NetworkInterfaceId": "eni-0863f88f670e8ea06",
        "OwnerId": "158826777352",
        "PrivateIpAddress": "172.16.100.99",
        "PrivateIpAddresses": [
            {
                "Primary": true,
                "PrivateIpAddress": "172.16.100.99"
            }
        ],
        "RequesterId": "AIDAJULMQVBYWWAB6IQQS",
            "RequesterManaged": false,
            "SourceDestCheck": true,
            "Status": "pending",
            "SubnetId": "subnet-0e398e93c154e8757",
            "TagSet": [],
            "VpcId": "vpc-0d19e8153b4d142ed"
        }
```

After a few moments, verify the status of the network interface as follows, using the `NetworkInterfaceId` from the preceding output:

```
aws ec2 describe-network-interfaces
--network-interface-ids [network-interface-id]
```

Enhanced Networking

Enhanced networking offers higher network throughput speeds and lower latency than ENIs. Enhanced networking uses single-root input/output virtualization (SR-IOV). SR-IOV allows multiple instances on the same physical host to bypass the hypervisor, resulting in lower CPU utilization and better network performance. You can take advantage of enhanced networking in two ways:

Elastic Network Adapter The elastic network adapter (ENA) supports throughput speeds up to 100 Gbps. Most instance types support it.

Intel 82599 Virtual Function Interface The Intel 82599 virtual function (VF) interface supports speeds up to 10 Gbps. It's available for only a few instance types that don't support ENAs.

Your instance's operating system must include the appropriate drivers to support enhanced networking. Amazon Linux and Ubuntu hardware virtual machine (HVM) AMIs have ENA support enabled by default.

Internet Gateways

An *Internet gateway* gives instances the ability to receive a public IP address, connect to the Internet, and receive requests from the Internet. The default VPC has an Internet gateway associated with it by default. But when you create a custom VPC, it does not have an Internet gateway associated with it. You must create an Internet gateway and associate it with a VPC manually. You can associate only one Internet gateway with a VPC, but you may create multiple Internet gateways and associate each one with a different VPC.

An Internet gateway is somewhat analogous to an Internet router an Internet service provider may install on-premises. But in AWS, an Internet gateway doesn't behave exactly like a router.

In a traditional network, to give your servers access to the Internet, you might configure your core router with a default route pointing to the Internet router's internal IP address. An Internet gateway, however, doesn't have an IP address or network interface. Instead, AWS identifies an Internet gateway by its resource ID, which begins with `igw-` followed by an alphanumeric string. To use an Internet gateway, you must create a *default route* in a *route table* that points to the Internet gateway as a target.

Route Tables

To control how traffic ingresses, egresses, and moves within your VPC, you need to use routes stored in route tables. Rather than using physical or virtual routers that you configure, the VPC architecture implements IP routing as a software function that AWS calls an *implied router* (also sometimes called an *implicit router*). This means there's no virtual router on which to configure interface IP addresses or dynamic routing protocols such as BGP. Rather, you only have to manage the route table that the implied router uses.

Each route table consists of one or more routes and at least one subnet association. Think of a route table as being connected to multiple subnets in much the same way a traditional router would be. When you create a VPC, AWS automatically creates a default route table called the *main route table* and associates it with every subnet in that VPC. You can use the main route table or create a custom one that you can manually associate with one or more subnets.

A subnet cannot exist without a route table association. If you do not explicitly associate a subnet with a custom route table you've created, AWS will implicitly associate it with the main route table.

Routes

Routes determine how to forward traffic to or from resources within the subnets associated with the route table. IP routing is destination-based, meaning that routing decisions are based only on the destination IP prefix, not the source IP address. When you create a route, you must provide the following elements:

- Destination IP prefix
- Target resource

The destination must be an IPv4 or IPv6 prefix in CIDR notation. The target must be an AWS network resource such as an Internet gateway or an ENI. It cannot be an IP prefix.

Every route table contains a *local route* that allows instances in different subnets to communicate with each other. Table 4.2 shows what this route would look like in a VPC with the CIDR 172.31.0.0/16.

TABLE 4.2 The local route

Destination	Target
172.31.0.0/16	Local

The local route is the only mandatory route that exists in every route table. It's what allows communication between instances in the same VPC. Because there are no routes for any other IP prefixes, any traffic destined for an address outside of the VPC CIDR range will get dropped.

The Default Route

To enable Internet access for your instances, you must create a default route pointing to the Internet gateway. The default route is what allows Internet traffic to ingress and egress the subnet. After adding a default route, you would end up with Table 4.3.

TABLE 4.3 Route table with default route

Destination	Target
172.31.0.0/16	Local
0.0.0.0/0	igw-0e538022a0fddc318

The 0.0.0.0/0 prefix encompasses all IP addresses, including those of hosts on the Internet. This is why it's always listed as the destination in a default route. Any subnet that is associated with a route table containing a route pointing to an Internet gateway is called a *public subnet*. Contrast this with a *private subnet*, which does not have a route with an Internet gateway as a target.

Notice that the 0.0.0.0/0 and 172.31.0.0/16 prefixes overlap. When deciding where to route traffic, the implied router will route based on the closest match. Put another way, the order of routes doesn't matter. Suppose an instance sends a packet to the Internet address 198.51.100.50. Because 198.51.100.50 does not match the 172.31.0.0/16 prefix but does match the 0.0.0.0/0 prefix, the implied router will use the default route and send the packet to the Internet gateway.

AWS documentation speaks of one implied router per VPC. It's important to understand that the implied router doesn't actually exist as a discrete resource. It's an abstraction of an IP routing function. Nevertheless, you may find it helpful to think of each route table as a separate virtual router with a connection to one or more subnets. Follow the steps in Exercise 4.4 to create an Internet gateway and a default route.

EXERCISE 4.4

Create an Internet Gateway and Default Route

In this exercise, you'll create an Internet gateway and attach it to the VPC you used in the previous exercises.

1. Create an Internet gateway using the following command:

```
aws ec2 create-internet-gateway
```

You should see the following output:

```
{
    "InternetGateway": {
        "Attachments": [],
        "InternetGatewayId": "igw-0312f81aa1ef24715",
        "Tags": []
    }
}
```

2. Attach the Internet gateway to the VPC you used in the previous exercises using the following command:

```
aws ec2 attach-internet-gateway
--internet-gateway-id [internet-gateway-id]
--vpc-id [vpc-id]
```

3. Retrieve the route table ID of the main route table for the VPC:

```
aws ec2 describe-route-tables
--filters Name=vpc-id,Values=[vpc-id]
```

You should see something like the following:

```
{
    "RouteTables": [
        {
            "Associations": [
                {
                    "Main": true,
                    "RouteTableAssociationId": "rtbassoc-00f60edb255be0332",
                    "RouteTableId": "rtb-097d8be97649e0584",
                    "AssociationState": {
                        "State": "associated"
                    }
                }
            ],
            "PropagatingVgws": [],
            "RouteTableId": "rtb-097d8be97649e0584",
            "Routes": [
                {
                    "DestinationCidrBlock": "172.16.0.0/16",
                    "GatewayId": "local",
                    "Origin": "CreateRouteTable",
                    "State": "active"
                }
            ],
            "Tags": [],
            "VpcId": "vpc-0d19e8153b4d142ed",
            "OwnerId": "158826777352"
        }
    ]
}
```

4. To create a default route in the main route table, issue the following command:

```
aws ec2 create-route
--route-table-id rtb-097d8be97649e0584
--destination-cidr-block "0.0.0.0/0"
--gateway-id igw-0312f81aa1ef24715
```

If the command succeeds, you should see the following output:

```
{
    "Return": true
}
```

5. Confirm the presence of the route by rerunning the command from step 3.

Security Groups

A *security group* functions as a firewall that controls traffic to and from an instance by permitting traffic to ingress or egress that instance's ENI. Every ENI must have at least one security group associated with it. One ENI can have multiple security groups attached, and the same security group can be attached to multiple ENIs.

In practice, because most instances have only one ENI, people often think of a security group as being attached to an instance. When an instance has multiple ENIs, take care to note whether those ENIs use different security groups.

When you create a security group, you must specify a group name, description, and VPC for the group to reside in. Once you create the group, you create inbound and outbound rules to specify what traffic the security group allows. If you don't explicitly allow traffic using a rule, the security group will block it.

Inbound Rules

Inbound rules specify what traffic is allowed into the attached ENI. An inbound rule consists of three required elements:

- Source
- Protocol
- Port range

When you create a security group, it doesn't contain any inbound rules. Security groups use a default-deny approach, also called *whitelisting*, which denies all traffic that is not explicitly allowed by a rule. When you create a new security group and attach it to an instance, all inbound traffic to that instance will be blocked. You must create inbound rules to allow traffic to your instance. The order of rules in a security group doesn't matter.

Suppose you have an instance running an HTTPS-based web application. You want to allow anyone on the Internet to connect to this instance, so you'd need an inbound rule to allow all TCP traffic coming in on port 443 (the default port and protocol for HTTPS). To manage this instance using SSH, you'd need another inbound rule for TCP port 22. However, you don't want to allow SSH access from just anyone. You need to allow SSH access only from the IP address 198.51.100.10. To achieve this, you would use a security group containing the inbound rules listed in Table 4.4.

TABLE 4.4 Inbound rules allowing SSH and HTTPS access from any IP address

Source	Protocol	Port range
198.51.100.10/32	TCP	22
0.0.0.0/0	TCP	443

The prefix 0.0.0.0/0 covers all valid IP addresses, so using the preceding rule would allow HTTPS access not only from the Internet but from all instances in the VPC as well.

Outbound Rules

Outbound rules specify what traffic the instance may send out of the attached ENI. The format of an outbound rule mirrors an inbound rule and contains three elements:

- Destination
- Protocol
- Port range

In many cases, the outbound rules of a security group will be less restrictive than the inbound rules. When you create a security group, AWS automatically creates the outbound rule listed in Table 4.5.

TABLE 4.5 Outbound rule allowing Internet access

Destination	Protocol	Port range
0.0.0.0/0	All	All

The main purpose of this rule is to allow the instance to access the Internet and other AWS resources. You may delete this rule, but if you do, the security group won't permit your instance to access the Internet or anything else!

Sources and Destinations

The source or destination in a rule can be any CIDR. The source can also be the resource ID of a security group. If you specify a security group as the source, the rule will permit the traffic to any instance with that security group attached. This makes it easy to allow instances to communicate with each other by simply assigning the same security group to all of them.

The source security group you use can exist in a different AWS account. You'll need to specify the AWS account owner ID in order to specify a source security group in a different account.

Stateful Firewall

A security group acts as a stateful firewall. Stateful means that when a security group allows traffic to pass in one direction, it intelligently allows reply traffic in the opposite direction. For instance, when you allow an instance outbound access to download updates from a software repository on the Internet, the security group automatically allows reply traffic back into the instance.

Security groups use connection tracking to determine whether to allow response traffic to pass. For TCP and UDP traffic, connection tracking looks at the flow information for each packet that is allowed by a security group rule. By tracking flows, the security group can identify reply traffic that belongs to the same flow and distinguish it from unsolicited traffic. Flow information includes:

- Protocol
- Source and destination IP address
- Source and destination port number

Default Security Group

Each VPC contains a default security group that you can't delete. You don't have to use it, but if you do, you can modify its rules to suit your needs. Alternatively, you may opt to create your own custom group and use it instead. Complete Exercise 4.5 to practice creating a custom security group.

EXERCISE 4.5

Create a Custom Security Group

In this exercise, you'll create a new security group that allows SSH, HTTP, and HTTPS access from any IP address.

1. Create a security group named **`web-ssh`** in the VPC you created previously:

```
aws ec2 create-security-group
--group-name "web-ssh"
--description "Web and SSH traffic"
--vpc-id [vpc-id]
```

   Make a note of the security group ID in the output:

```
{
    "GroupId": "sg-0f076306080ac2c91"
}
```

2. Using the security group ID, create three rules to allow SSH, HTTP, and HTTPS access from any IP address.

```
aws ec2 authorize-security-group-ingress
--group-id [group-id]
--protocol "tcp"
--cidr "0.0.0.0/0"
--port "22"
```

```
aws ec2 authorize-security-group-ingress
--group-id [group-id]
--protocol "tcp"
--cidr "0.0.0.0/0"
--port "80"
aws ec2 authorize-security-group-ingress
--group-id [group-id]
--protocol "tcp"
--cidr "0.0.0.0/0"
--port "443"
```

3. Verify the rules by viewing the security group:

```
aws ec2 describe-security-groups
--group-id [group-id]
```

Network Access Control Lists

Like a security group, a *network access control list* (NACL) functions as a firewall in that it contains inbound and outbound rules to allow traffic based on a source or destination CIDR, protocol, and port. Also, each VPC has a default NACL that can't be deleted. But the similarities end there.

An NACL differs from a security group in many respects. Instead of being attached to an ENI, an NACL is attached to a subnet. The NACL associated with a subnet controls what traffic may enter and exit that subnet. This means that NACLs can't be used to control traffic between instances in the same subnet. If you want to do that, you have to use security groups.

A subnet can have only one NACL associated with it. When you create a new subnet in a VPC, the VPC's default NACL is associated with the subnet by default. You can modify the default NACL, or you can create a new one and associate it with the subnet. You can also associate the same NACL with multiple subnets, provided those subnets are all in the same VPC as the NACL.

Unlike a security group, which is stateful, an NACL is stateless, meaning that it doesn't use connection tracking and doesn't automatically allow reply traffic. This is much like an access control list (ACL) on a traditional switch or router. The stateless nature of the NACL is why each one is preconfigured with rules to allow all inbound and outbound traffic, as discussed in the following sections.

Inbound Rules

Inbound rules determine what traffic is allowed to ingress the subnet. Each rule contains the following elements:

- Rule number
- Protocol
- Port range
- Source CIDR
- Action—Allow or deny

The default NACL for a VPC that has no IPv6 CIDR assigned comes prepopulated with the two inbound rules listed in Table 4.6.

TABLE 4.6 Default NACL inbound rules

Rule number	Protocol	Port range	Source	Action
100	All	All	0.0.0.0/0	Allow
*	All	All	0.0.0.0/0	Deny

NACL rules are processed in ascending order of the rule number. Rule 100 is the lowest-numbered rule, so it gets processed first. This rule allows all traffic from any source. You can delete or modify this rule or create additional rules before or after it. For example, if you wanted to block only HTTP (TCP port 80), you could add the rule in Table 4.7.

TABLE 4.7 Blocking rule

Rule number	Protocol	Port range	Source	Action
90	TCP	80	0.0.0.0/0	Deny

This rule denies all TCP traffic with a destination port of 80. Because it's the lowest-numbered rule in the list, it gets processed first. Any traffic not matching this rule would be processed by rule 100, which allows all traffic. The net effect of rules 90 and 100 is that all inbound traffic except for that destined to TCP port 80 would be allowed.

NOTE After traffic matches a rule, the action specified in the rule occurs. That means if traffic matches a rule with a deny action, that traffic will be denied, even if there's a subsequent (higher-numbered) rule that would allow the traffic.

The last rule in Table 4.6 is the default rule. It's designated by an asterisk (*) instead of a number and is always the last rule in the list. You can't delete or otherwise change the default rule. The default rule causes the NACL to deny any traffic that isn't explicitly allowed by any of the preceding rules. Complete Exercise 4.6 to create a custom NACL.

EXERCISE 4.6

Create an Inbound Rule to Allow Remote Access from Any IP Address

NACL rule order matters! Create a new NACL and attach it to the subnet you used in Exercise 4.3.

1. Create a new network ACL using the following command:

```
aws ec2 create-network-acl
--vpc-id [vpc-id]
```

Note the network ACL ID in the output, which should look something like the following:

```
{
    "NetworkAcl": {
        "Associations": [],
        "Entries": [
            {
                "CidrBlock": "0.0.0.0/0",
                "Egress": true,
                "IcmpTypeCode": {},
                "PortRange": {},
                "Protocol": "-1",
                "RuleAction": "deny",
                "RuleNumber": 32767
            },
            {
                "CidrBlock": "0.0.0.0/0",
                "Egress": false,
                "IcmpTypeCode": {},
```

EXERCISE 4.6 *(continued)*

```
                "PortRange": {},
                "Protocol": "-1",
                "RuleAction": "deny",
                "RuleNumber": 32767
            }
        ],
        "IsDefault": false,
        "NetworkAclId": "acl-052f05f358d96cfb3",
        "Tags": [],
        "VpcId": "vpc-0d19e8153b4d142ed",
        "OwnerId": "158826777352"
    }
}
```

Notice that the network ACL includes the default ingress and egress rules to deny all traffic. Even though the rule number is 32767, it shows up in the AWS Management Console as an asterisk.

2. Create an inbound rule entry to allow SSH (TCP port 22) access from any IP address. We'll use the rule number 70.

```
aws ec2 create-network-acl-entry
--ingress
--cidr-block "0.0.0.0/0"
--protocol "tcp"
--port-range "From=22,To=22"
--rule-action "allow"
--network-acl-id [network-acl-id]
--rule-number 70
```

The From= and To= values after the `--port-range` flag refer to the range of ports to allow, *not* to the source port number. In this rule, we just want to allow traffic to TCP port 22.

3. Modify the command from step 2 to create rule number 80 that allows RDP (TCP port 3389) access from your public IP address (Use `https://ifconfig.co` to find it if you don't already know it). Use a /32 prefix length.

4. Use the following command to verify that the rules made it into your NACL:

```
aws ec2 describe-network-acls
--network-acl-id [network-acl-id]
```

Outbound Rules

As you might expect, the outbound NACL rules follow an almost identical format as the inbound rules. Each rule contains the following elements:

- Rule number
- Protocol
- Port range
- Destination
- Action

Each default NACL comes with the outbound rules listed in Table 4.8. Notice that the rules are identical to the default inbound rules except for the Destination element.

TABLE 4.8 Default NACL outbound rules

Rule number	Protocol	Port range	Destination	Action
100	All	All	0.0.0.0/0	Allow
*	All	All	0.0.0.0/0	Deny

Because an NACL is stateless, it won't allow return traffic unless you add a rule to permit it. Therefore, if you permit HTTPS traffic with an inbound rule, you must also explicitly permit the return traffic using an outbound rule. In this case, rule 100 permits the return traffic.

If you do need to restrict access from the subnet—to block Internet access, for example—you will have to create an outbound rule to allow return traffic over *ephemeral ports*. Ephemeral ports are reserved TCP or UDP ports that clients listen for reply traffic on. As an example, when a client sends an HTTPS request to your instance over TCP port 80, that client may listen for a reply on TCP port 36034. Your NACL's outbound rules must allow traffic to TCP port 36034.

The range of ephemeral ports varies by client operating system. Many modern operating systems use ephemeral ports in the range of 49152–65535. But don't assume that allowing only this range will be sufficient. The range for TCP ports may differ from the range for UDP, and older or customized operating systems may use a different range altogether. To maintain compatibility, do not restrict outbound traffic using an NACL. Instead, use a security group to restrict outbound traffic.

NOTE

If your VPC includes an IPv6 CIDR, AWS will automatically add inbound and outbound rules to permit IPv6 traffic.

Using Network Access Control Lists and Security Groups Together

You may want to use an NACL in addition to a security group so that you aren't dependent on AWS administrators to specify the correct security group when they launch an instance. Because an NACL is applied to the subnet, the rules of the NACL apply to all traffic ingressing and egressing the subnet, regardless of how the security groups are configured.

When you make a change to an NACL or security group rule, that change takes effect immediately (practically, within several seconds). Avoid changing security groups and NACLs simultaneously. If your changes don't work as expected, it can become difficult to identify whether the problem is with a security group or an NACL. Complete your changes on one before moving to the other. Additionally, be cautious about making changes when there are active connections to an instance, because an incorrectly ordered NACL rule or missing security group rule can terminate those connections.

NOTE

In an NACL rule, you can specify only a CIDR as the source or destination. This is unlike a security group rule, for which you can specify another security group for the source or destination.

Public IP Addresses

A public IP address is reachable over the public Internet. This is in contrast to RFC 1918 addresses (e.g., 192.168.1.1), which cannot be routed over the Internet but can be routed within private networks.

You need a public IP address for an instance if you want others to directly connect to it via the Internet. Naturally, this requires an Internet gateway attached to the VPC that the instance resides in. You *may* also give an instance a public IP address if you want it to have outbound-only Internet access. You *don't* need a public IP address for your instances to communicate with each other within the VPC infrastructure, as this instance-to-instance communication happens using private IP addresses.

When you launch an instance into a subnet, you can choose to automatically assign it a public IP. This is convenient, but there are a couple of potential downsides to this approach.

First, if you forget to choose this option when you launch the instance, you cannot go back and have AWS automatically assign a public IP later. Second, automatically assigned public IP addresses aren't persistent. When you stop or terminate the instance, you will lose the public IP address. If you stop and restart the instance, it will receive a different public IP address.

Even if you don't plan to stop your instance, keep in mind that AWS may perform maintenance events that cause your instance to restart. If this happens, its public IP address will change.

If your instance doesn't need to maintain the same IP address for a long period of time, this approach may be acceptable. If not, you may opt instead for an *elastic IP address*.

Elastic IP Addresses

An elastic IP (EIP) address is a type of public IP address that AWS allocates to your account when you request it. After AWS allocates an EIP to your account, you have exclusive use of that address until you manually release it. Outside of AWS, there's no noticeable difference between an EIP and an automatically assigned public IP.

When you initially allocate an EIP, it is not bound to any instance. Instead, you must associate it with an ENI. You can move an EIP around to different ENIs, although you can associate it with only one ENI at a time. When you associate an EIP with an ENI, it will remain associated for the life of the ENI or until you disassociate it.

If you associate an EIP to an ENI that already has an automatically assigned public IP address allocated, AWS will replace the public IP address with the EIP. Complete Exercise 4.7 to practice allocating and using an EIP.

An EIP is tied to an AWS region and can't be moved outside of that region. It is possible, however, to transfer any public IP addresses that you own into your AWS account as EIPs. This is called bring your own IP address (BYOIP). You can bring up to five address blocks per region.

EXERCISE 4.7

Allocate and Use an Elastic IP Address

Allocate an elastic IP address and associate it with the instance you created earlier.

1. Allocate an EIP using the following command:

   ```
   aws ec2 allocate-address
   ```

   You should see a public IP address and allocation ID in the output:

   ```
   {
       "PublicIp": "3.210.93.49",
       "AllocationId": "eipalloc-0e14547dac92e8a75",
       "PublicIpv4Pool": "amazon",
       "NetworkBorderGroup": "us-east-1",
       "Domain": "vpc"
   }
   ```

2. Associate the EIP to the elastic network interface you created earlier:

   ```
   aws ec2 associate-address
   --allocation-id eipalloc-0e14547dac92e8a75
   --network-interface-id eni-0863f88f670e8ea06
   ```

EXERCISE 4.7 *(continued)*

3. Verify that the EIP is associated with the elastic network interface:

```
aws ec2 describe-network-interfaces
--network-interface-ids eni-0863f88f670e8ea06
```

You should see the public IP address in the output:

```
{
    "NetworkInterfaces": [
        {
            "Association": {
                "AllocationId": "eipalloc-0e14547dac92e8a75",
                "AssociationId": "eipassoc-045cc2cab27d6338b",
                "IpOwnerId": "158826777352",
                "PublicDnsName": "",
                "PublicIp": "3.210.93.49"
            },
            "AvailabilityZone": "us-east-1a",
            "Description": "",
            "Groups": [
                {
                    "GroupName": "default",
                    "GroupId": "sg-011c3fe63d256f5d9"
                }
            ],
            "InterfaceType": "interface",
            "Ipv6Addresses": [],
            "MacAddress": "12:6a:74:e1:95:a9",
            "NetworkInterfaceId": "eni-0863f88f670e8ea06",
            "OwnerId": "158826777352",
            "PrivateIpAddress": "172.16.100.99",
            "PrivateIpAddresses": [
                {
                    "Association": {
                        "AllocationId": "eipalloc-0e14547dac92e8a75",
                        "AssociationId": "eipassoc-045cc2cab27d6338b",
                        "IpOwnerId": "158826777352",
                        "PublicDnsName": "",
                        "PublicIp": "3.210.93.49"
                    },
                    "Primary": true,
                    "PrivateIpAddress": "172.16.100.99"
                }
            ],
            "RequesterId": "AIDAJULMQVBYWWAB6IQQS",
            "RequesterManaged": false,
            "SourceDestCheck": true,
            "Status": "available",
            "SubnetId": "subnet-0e398e93c154e8757",
            "TagSet": [],
            "VpcId": "vpc-0d19e8153b4d142ed"
        }
    ]
}
```

AWS Global Accelerator

If you have AWS resources in multiple regions, having separate EIPs for each region can be cumbersome to manage. AWS Global Accelerator solves this problem by giving you two anycast static IPv4 addresses that you can use to route traffic to resources in any region. Unlike EIPs, which are tied to an AWS region, Global Accelerator static addresses are spread across different AWS points-of-presence (POPs) in over 30 countries. These static addresses are also called anycast addresses because they are simultaneously advertised from multiple POPs.

Users connecting to a static address are automatically routed to the nearest POP. The Global Accelerator listener receives the TCP or UDP connection, and then proxies it to the resources you've specified in an endpoint group. An endpoint group can contain elastic IP addresses, elastic load balancers, or EC2 instances.

Global Accelerator routes traffic to the fastest endpoint. Because the static addresses it uses are anycast, the failure of one POP doesn't cause an interruption of service. Users are automatically and transparently routed to another POP.

Network Address Translation

When you associate an ENI with a public IP address, the ENI maintains its private IP address. Associating a public IP with an ENI doesn't reconfigure the ENI with a new address. Instead, the Internet gateway maps the public IP address to the ENI's private IP address using a process called *network address translation* (NAT).

When an instance with a public IP connects to a host on the Internet, the host sees the traffic as originating from the instance's public IP. For example, assume an instance with a private IP address of 172.31.7.10 is associated with the EIP 35.168.241.48. When the instance attempts to send a packet to the Internet host 198.51.100.11, it will send the following packet to the Internet gateway:

Source IP address: 172.31.7.10

Destination IP address: 35.168.241.48

The Internet gateway will translate this packet to change the source IP address to the instance's public IP address. The translated packet, which the Internet gateway forwards to the host, looks like this:

Source IP address: 35.168.241.48

Destination IP address: 198.51.100.11

Likewise, when a host on the Internet sends a packet to the instance's EIP, the Internet gateway will perform network address translation on the incoming packet. The packet that reaches the Internet gateway from the Internet host will look like this:

Source IP address: 198.51.100.11

Destination IP address: 35.168.241.48

The Internet gateway will translate this packet, replacing the destination IP address with the instance's private IP address, as follows:

Source IP address: 198.51.100.11

Destination IP address: 172.31.7.10

Network address translation occurs automatically at the Internet gateway when an instance has a public IP address. You can't change this behavior.

NOTE Network address translation as described here is also sometimes called one-to-one NAT because one private IP address gets mapped to one public IP address.

Network Address Translation Devices

Although network address translation occurs at the Internet gateway, there are two other resources that can also perform NAT:

- NAT gateway
- NAT instance

AWS calls these *NAT devices*. The purpose of a NAT device is to allow an instance to access the Internet while preventing hosts on the Internet from reaching the instance directly. This is useful when an instance needs to go out to the Internet to fetch updates or to upload data but does not need to service requests from clients.

When you use a NAT device, the instance needing Internet access does not have a public IP address allocated to it. Incidentally, this makes it impossible for hosts on the Internet to reach it directly. Instead, only the NAT device is configured with a public IP. Additionally, the NAT device has an interface in a public subnet. Refer to Table 4.9 for an example.

TABLE 4.9 IP address configuration when using a NAT device

Name	Subnet	Private IP	Public IP
db1	Private	172.31.7.11	None
db2	Private	172.31.7.12	None
NAT device	Public	172.31.8.10	18.209.220.180

When db1 sends a packet to a host on the Internet with the address 198.51.100.11, the packet must first go to the NAT device. The NAT device translates the packet as follows:

Original Packet's Source IP Address	Original Packet's Destination IP Address	Translated Packet's Source IP Address	Translated Packet's Destination IP Address
172.31.7.11 (db1)	198.51.100.11	172.31.8.10 (NAT device)	198.51.100.11

The NAT device then takes the translated packet and forwards it to the Internet gateway. The Internet gateway performs NAT translation on this packet as follows:

NAT Device Packet's Source IP Address	NAT Device Packet's Destination IP Address	Translated Packet's Source IP Address	Translated Packet's Destination IP Address
172.31.8.10 (NAT device)	198.51.100.11	18.209.220.180 (NAT device's EIP)	198.51.100.11

Multiple instances can use the same NAT device, thus sharing the same public IP address for outbound connections. The function that NAT devices perform is also called *port address translation* (PAT).

Configuring Route Tables to Use NAT Devices

Instances that use the NAT device must send Internet-bound traffic to it, and the NAT device must send Internet-bound traffic to an Internet gateway. Hence, the NAT device and the instances that use it must use different default routes. Furthermore, they must also use different route tables and hence must reside in separate subnets.

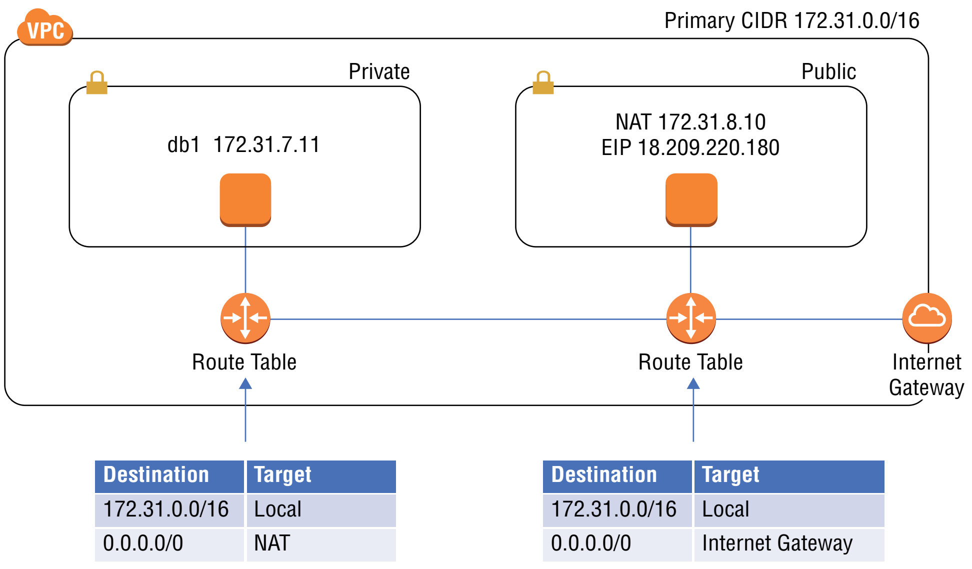
Refer to Table 4.9 again. Notice that the instances reside in the Private subnet, and the NAT device is in the Public subnet. The default routes for these subnets would follow the pattern in Table 4.10.

TABLE 4.10 Default routes for the Private and Public subnets

Subnet	Destination	Target
Private	0.0.0.0/0	NAT device
Public	0.0.0.0/0	igw-0e538022a0fddc318

Refer to the diagram in Figure 4.2 to see the relationship between both of the route tables. Recall that a route target must be a VPC resource such as an instance, Internet gateway, or ENI. The specific target you choose depends on the type of NAT device you use: a NAT gateway or a NAT instance.

FIGURE 4.2 Network address translation using a NAT device



Destination	Target
172.31.0.0/16	Local
0.0.0.0/0	NAT

Destination	Target
172.31.0.0/16	Local
0.0.0.0/0	Internet Gateway

NAT Gateway

A NAT gateway is a NAT device managed by AWS. Like an Internet gateway, it's a one-size-fits-all resource. It doesn't come in a variety of flavors, and there's nothing to manage or access. It automatically scales to accommodate your bandwidth requirements. You set it and forget it.

When you create a NAT gateway, you must assign it an EIP. A NAT gateway can reside in only one subnet, which must be a public subnet for it to access the Internet. AWS selects a private IP address from the subnet and assigns it to the NAT gateway. For redundancy, you may create additional NAT gateways in different availability zones.

After creating a NAT gateway, you must create a default route to direct Internet-bound traffic from your instances to the NAT gateway. The target you specify will be the NAT gateway ID, which follows the format `nat-0750b9c8de7e75e9f`. If you use multiple NAT gateways, you can create multiple default routes, each pointing to a different NAT gateway as the target.

Because a NAT gateway doesn't use an ENI, you can't apply a security group to it. You can, however, apply an NACL to the subnet that it resides in.

NAT Instance

A NAT instance is a normal EC2 instance that uses a preconfigured Linux-based AMI. You have to perform the same steps to launch it as you would any other instance. It functions like a NAT gateway in many respects, but there are some key differences.

Unlike a NAT gateway, a NAT instance doesn't automatically scale to accommodate increased bandwidth requirements. Therefore, it's important that you select an appropriately robust instance type. If you choose an instance type that's too small, you must manually upgrade to a larger instance type.

Also, a NAT instance has an ENI, so you must apply a security group to it. You also must remember to assign it a public IP address. Lastly, you must disable the *source/destination check* on the NAT instance's ENI. This allows the NAT instance to receive traffic addressed to an IP other than its own, and it also allows the instance to send traffic using a source IP that it doesn't own.

One advantage of a NAT instance is that you can use it as a *bastion host*, sometimes called a *jump host*, to connect to instances that don't have a public IP. You can't do this with a NAT gateway.

You must create a default route to direct Internet-bound traffic to the NAT instance. The target of the default route will be the NAT instance's ID, which follows the format `i-0a1674fe5671dcb00`.

If you want to guard against instance or availability zone failures, it's not as simple as just spinning up another NAT instance. You cannot create multiple default routes pointing to different NAT instances. If you need this level of resiliency, you're better off using NAT gateways instead.

VPC Peering

You can configure VPC peering to allow instances in one VPC to communicate with VPCs in another over the private AWS network. You may want to do this if you have instances in different regions that need to communicate. You may also want to connect your instances to another AWS customer's instances.

To enable VPC peering, you must set up a *VPC peering connection* between two VPCs. A VPC peering connection is a point-to-point connection between two and only two VPCs. You can have at most one peering connection between a pair of VPCs. Also, peered VPCs must not have overlapping CIDR blocks.

With one exception, a VPC peering connection allows only instance-to-instance communication. This means an instance in one VPC can use the peering connection only to connect to another instance in the peered VPC. You can't use it to share Internet gateways or NAT devices. You can, however, use it to share a network load balancer (NLB).

If you have more than two VPCs you need to connect, you *cannot* daisy-chain VPC peering connections together and route through them. You must create a peering connection between each pair. This configuration is called *transitive routing.*

To use a peering connection, you must create new routes in both VPCs to allow traffic to travel in both directions. For each route, the destination prefix should exist within the destination VPC. The target of each route must be the peering connection's identifier, which begins with pcx–. Refer to Table 4.11 for examples of two routes you would create to enable peering between a pair of VPCs.

TABLE 4.11 Routes for VPC peering

Source VPC CIDR	Destination VPC CIDR	Target
172.31.0.0/16	10.0.0.0/16	pcx-076781cf11220b9dc
10.0.0.0/16	172.31.0.0/16	pcx-076781cf11220b9dc

Notice that the routes are mirrors of each other. Again, this is to allow traffic in both directions. The destination CIDR doesn't need to exactly match that of the destination VPC. If you want to enable peering only between specific subnets, you can specify the subnet CIDR instead.

Interregion VPC peering is not available for some AWS regions. Peering connections between regions have a maximum transmission unit (MTU) of 1,500 bytes and do not support IPv6.