

AWS

Q1. How you manage Concurrency in Lambda?

There are two types of concurrency controls available:

Reserved concurrency – Reserved concurrency guarantees the maximum number of concurrent instances for the function. When a function has reserved concurrency, no other function can use that concurrency. There is no charge for configuring reserved concurrency for a function.

Provisioned concurrency – Provisioned concurrency initializes a requested number of execution environments so that they are prepared to respond immediately to your function's invocations. Note that configuring provisioned concurrency incurs charges to your AWS account.

This topic details how to manage and configure reserved concurrency. If you want to decrease latency for your functions, use provisioned concurrency.

Concurrency is the number of requests that your function is serving at any given time. When your function is invoked, Lambda allocates an instance of it to process the event. When the function code finishes running, it can handle another request. If the function is invoked again while a request is still being processed, another instance is allocated, which increases the function's concurrency. The total concurrency for all of the functions in your account is subject to a per-region quota.

To learn about how concurrency interacts with scaling, see [Lambda function scaling](#).

How do you manage in Disaster management in your application

Disaster recovery strategies available to you within AWS can be broadly categorized into four approaches, ranging from the low cost and low complexity of making backups to more complex strategies using multiple active Regions. Active/passive strategies use an active site (such as an AWS Region) to host the workload and serve traffic. The passive site (such as a different AWS Region) is used for recovery. The passive site does not actively serve traffic until a failover event is triggered.

It is critical to regularly assess and test your disaster recovery strategy so that you have confidence in invoking it, should it become necessary. Use [AWS Resilience](#)

Hub to continuously validate and track the resilience of your AWS workloads, including whether you are likely to meet your RTO and RPO targets.

For a disaster event based on disruption or loss of one physical data center for a well-architected, highly available workload, you may only require a backup and restore approach to disaster recovery. If your definition of a disaster goes beyond the disruption or loss of a physical data center to that of a Region or if you are subject to regulatory requirements that require it, then you should consider Pilot Light, Warm Standby, or Multi-Site Active/Active.

When choosing your strategy, and the AWS resources to implement it, keep in mind that within AWS, we commonly divide services into the data plane and the control plane. The data plane is responsible for delivering real-time service while control planes are used to configure the environment. For maximum resiliency, you should use only data plane operations as part of your failover operation. This is because the data planes typically have higher availability design goals than the control planes.

Backup and restore

Backup and restore is a suitable approach for mitigating against data loss or corruption. This approach can also be used to mitigate against a regional disaster by replicating data to other AWS Regions, or to mitigate lack of redundancy for workloads deployed to a single Availability Zone. In addition to data, you must redeploy the infrastructure, configuration, and application code in the recovery Region. To enable infrastructure to be redeployed quickly without errors, you should always deploy using infrastructure as code (IaC) using services such as AWS CloudFormation or the AWS Cloud Development Kit (AWS CDK). Without IaC, it may be complex to restore workloads in the recovery Region, which will lead to increased recovery times and possibly exceed your RTO. In addition to user data, be sure to also back up code and configuration, including Amazon Machine Images (AMIs) you use to create Amazon EC2 instances. You can use AWS CodePipeline to automate redeployment of application code and configuration.

Configuring a Streaming Source

At the time that you create an application, you specify a streaming source. You can also modify an input after you create the application. Amazon Kinesis Data Analytics supports the following streaming sources for your application:

A Kinesis data stream

A Kinesis Data Firehose delivery stream

Note

If the Kinesis data stream is encrypted, Kinesis Data Analytics accesses the data in the encrypted stream seamlessly with no further configuration needed. Kinesis Data Analytics does not store unencrypted data read from Kinesis Data Streams. For more information, see [What Is Server-Side Encryption For Kinesis Data Streams?](#).

Kinesis Data Analytics continuously polls the streaming source for new data and ingests it in in-application streams according to the input configuration.

Note

Adding a Kinesis Stream as your application's input does not affect the data in the stream. If another resource such as a Kinesis Data Firehose delivery stream also accessed the same Kinesis stream, both the Kinesis Data Firehose delivery stream and the Kinesis Data Analytics application would receive the same data.

Throughput and throttling might be affected, however.

Your application code can query the in-application stream. As part of input configuration you provide the following:

Streaming source – You provide the Amazon Resource Name (ARN) of the stream and an IAM role that Kinesis Data Analytics can assume to access the stream on your behalf.

In-application stream name prefix – When you start the application, Kinesis Data Analytics creates the specified in-application stream. In your application code, you access the in-application stream using this name.

You can optionally map a streaming source to multiple in-application streams. For more information, see [Limits](#). In this case, Amazon Kinesis Data Analytics creates the specified number of in-application streams with names as follows: `prefix_001`, `prefix_002`, and `prefix_003`. By default, Kinesis Data Analytics maps the streaming source to one in-application stream named `prefix_001`.

There is a limit on the rate that you can insert rows in an in-application stream. Therefore, Kinesis Data Analytics supports multiple such in-application streams so that you can bring records into your application at a much faster rate. If you find

that your application is not keeping up with the data in the streaming source, you can add units of parallelism to improve performance.

Mapping schema – You describe the record format (JSON, CSV) on the streaming source. You also describe how each record on the stream maps to columns in the in-application stream that is created. This is where you provide column names and data types.

Note

Kinesis Data Analytics adds quotation marks around the identifiers (stream name and column names) when creating the input in-application stream. When querying this stream and the columns, you must specify them in quotation marks using the same casing (matching lowercase and uppercase letters exactly). For more information about identifiers, see [Identifiers in the Amazon Kinesis Data Analytics SQL Reference](#).

You can create an application and configure inputs in the Amazon Kinesis Data Analytics console. The console then makes the necessary API calls. You can configure application input when you create a new application API or add input configuration to an existing application. For more information, see [CreateApplication](#) and [AddApplicationInput](#). The following is the input configuration part of the [CreateApplication](#) API request body:

```
"Inputs": [
  {
    "InputSchema": {
      "RecordColumns": [
        {
          "Mapping": "string",
          "Name": "string",
          "SqlType": "string"
        }
      ],
      "RecordEncoding": "string",
      "RecordFormat": {
        "MappingParameters": {
          "CSVMappingParameters": {
```

```

        "RecordColumnDelimiter": "string",
        "RecordRowDelimiter": "string"
    },
    "JSONMappingParameters": {
        "RecordRowPath": "string"
    }
},
"RecordFormatType": "string"
}
},
"KinesisFirehoseInput": {
    "ResourceARN": "string",
    "RoleARN": "string"
},
"KinesisStreamsInput": {
    "ResourceARN": "string",
    "RoleARN": "string"
},
"Name": "string"
}
]

```

Configuring a Reference Source

You can also optionally add a reference data source to an existing application to enrich the data coming in from streaming sources. You must store reference data as an object in your Amazon S3 bucket. When the application starts, Amazon Kinesis Data Analytics reads the Amazon S3 object and creates an in-application reference table. Your application code can then join it with an in-application stream.

You store reference data in the Amazon S3 object using supported formats (CSV, JSON). For example, suppose that your application performs analytics on stock orders. Assume the following record format on the streaming source:

Ticker, SalePrice, OrderId

AMZN \$700 1003

XYZ \$250 1004

This article looks at a few approaches Amazon has taken to manage API requests to its systems to avoid overload by implementing API rate limiting (also referred to as “throttling” or “admission control”). Without these kinds of protection, a system becomes overloaded when more traffic flows into the system than it is scaled to handle at that time. API rate limiting lets us shape incoming traffic in different ways, such as prioritizing the client workloads that remain within their planned usage, while applying backpressure to the client workload that spikes unpredictably. In this article, I’ll cover topics ranging from admission control algorithms to operational considerations for safely updating quota values in a production system. I also focus on ways Amazon has designed APIs with fairness in mind to provide predictable performance and availability and to help customers avoid the need for workloads that could lead to rate limiting.

Do you have any exposure in multitenant application

Fairness in multi-tenant systems

Any multitenancy service works in concert with systems to ensure fairness.

Fairness means that every client in a multi-tenant system is provided with a single-tenant experience. The systems that ensure fairness in multi-tenant systems are similar to systems that perform bin-packing algorithms, which are classic algorithms in computer science. These fairness systems do the following things:

Perform placement algorithms to find a spot in the fleet for new workload.

(Similar to finding a bin with room for the workload.)

Continuously monitor the utilization of each workload and each server to move workloads around. (Similar to moving workloads between bins to ensure that no bin is too full.)

Monitor the overall fleet utilization, and add or remove capacity as needed.

(Similar to adding more bins when they’re all getting full, and removing bins when they’re empty.)

Allow workloads to stretch beyond hard-allocated performance boundaries as long as the underlying system isn’t being fully utilized, and hold workloads to their boundaries when the system is fully utilized. (Similar to allowing workloads to stretch within each bin as long as they’re not crowding out other workloads.)

Q5. How do you optimize query in Dynamo DB? If one query is taking 40 seconds to sharing data?

Performance considerations for scans

In general, Scan operations are less efficient than other operations in DynamoDB. A Scan operation always scans the entire table or secondary index. It then filters out values to provide the result you want, essentially adding the extra step of removing data from the result set.

If possible, you should avoid using a Scan operation on a large table or index with a filter that removes many results. Also, as a table or index grows, the Scan operation slows. The Scan operation examines every item for the requested values and can use up the provisioned throughput for a large table or index in a single operation. For faster response times, design your tables and indexes so that your applications can use Query instead of Scan. (For tables, you can also consider using the GetItem and BatchGetItem APIs.)

Alternatively, design your application to use Scan operations in a way that minimizes the impact on your request rate.

What Is Elasticsearch?

ElasticSearch is an open-source, broadly distributable, readily scalable, enterprise-grade search engine based on Lucene and released under the terms of the Apache License. It is Java-based and designed to operate in real-time. It can search and index document files in diverse formats. It was designed to be used in distributed environments by providing flexibility and scalability. Now, ElasticSearch is the most popular enterprise search engine followed by Apache Solr, also based on Lucene.

ElasticSearch is able to achieve fast search responses because instead of searching the text directly, it searches an index instead. This is more or less like searching for a keyword by scanning the index at the back of a book as opposed to searching every word of every page of the book. ElasticSearch can scale up to thousands of servers and accommodate petabytes of data. Its enormous capacity results directly from its elaborate, distributed architecture.

Instead of the typical full-text search setup, ElasticSearch offers ways to extend searching capabilities through the use of APIs and query DSLs. There are clients

available so that it can be used with numerous programming languages, such as Ruby, PHP, JavaScript, and others.

ElasticSearch is used for a lot of different use cases as well, for example, “classic” full-text search, analytics storage, auto-complete, spell checker, alerting engine, and a general purpose document store.

Advantages

Advantages of ElasticSearch include the following:

Lots of search options. ElasticSearch implements a lot of features when it comes to search such as customized splitting text into words, customized stemming, faceted search, full-text search, autocomplete, and instant search. Also, fuzzy search is good for spelling errors. You can find what you are searching for even though you have a spelling mistake. Autocompletion and instant search refer to searching while the user types. It can be simple suggestions of existing tags, trying to predict a search based on search history, or just doing a completely new search for every keyword.

Document-oriented. ElasticSearch stores real-world complex entities as structured JSON documents and indexes all fields by default, with a higher performance result.

Speed. Speaking of performance, ElasticSearch is able to execute complex queries extremely fast. It also caches almost all of the structured queries commonly used as a filter for the result set and executes them only once. For every other request containing a cached filter, it checks the result from the cache.

Scalability. Software development teams favor ElasticSearch because it is a distributed system by nature and can easily scale horizontally, providing the ability to extend resources and balance the loading between the nodes in a cluster.

Data record. ElasticSearch records any changes made in transactions logs on multiple nodes in the cluster to minimize the chance of data loss.

Query fine tuning. Elastic search has a powerful JSON-based DSL, which allows development teams to construct complex queries and fine tune them to receive the most precise results from a search. It provides also a way of ranking and grouping results.

RESTful API. Elasticsearch is API-driven, so actions can be performed using a simple RESTful API.


Distributed approach. Indices can be divided into shards, with each shard able to have any number of replicas. Routing and rebalancing operations are done automatically when new documents are added.

Multi-tenancy. Often, you have multiple customers or users with separate collections of documents, and a user should never be able to search documents that do not belong to them. This often leads to a design where every user has their own index. Often, this leads to having too many indexes. One larger Elasticsearch index is actually better.

Explain security services which you utilized in your application

Identity & access management

Securely manage access to services and resources

 AWS Identity & Access Management (IAM)


Cloud single-sign-on (SSO) service

 AWS IAM Identity Center (successor to AWS SSO)


Identity management for your apps

 Amazon Cognito

Managed Microsoft Active Directory

 AWS Directory Service

Simple, secure service to share AWS resources

 AWS Resource Access Manager

Central governance and management across AWS accounts

 AWS Organizations

Explain event hub based solution?

Amazon EventBridge is a serverless event bus that makes it easier to build event-driven applications at scale using events generated from your applications, integrated Software-as-a-Service (SaaS) applications, and AWS services.

EventBridge delivers a stream of real-time data from event sources such as Zendesk or Shopify to targets like AWS Lambda and other SaaS applications. You can set up routing rules to determine where to send your data to build application

architectures that react in real-time to your data sources with event publisher and consumer completely decoupled.

Explain RabbitMQ and MSMQ?

RabbitMQ is the most widely deployed open source message broker.

With tens of thousands of users, RabbitMQ is one of the most popular open source message brokers. From T-Mobile to Runtastic, RabbitMQ is used worldwide at small startups and large enterprises.

RabbitMQ is lightweight and easy to deploy on premises and in the cloud. It supports multiple messaging protocols. RabbitMQ can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements.

RabbitMQ runs on many operating systems and cloud environments, and provides a wide range of developer tools for most popular languages.

See how other people are using RabbitMQ:

Basis of Comparison RabbitMQ MSMQ

Platform Open Source Platform. Available free for all. Supported by the community. Proprietary platform by Microsoft. Supported by Microsoft.

Operating System Works in multiple OS (Linux, Mac, Windows). It works only in Windows OS.

Protocol Communication between applications takes place through a platform-agnostic wire-level protocol named Advanced Message Queuing Protocol (AMQP). It uses multi various Proprietary protocols, and there is no standard one.

Functionality A sender cannot directly communicate with the receiver, and messages are communicated to the receiver through a central exchange server. The exchange manages message queues. It operates in a centralized queuing method. Sending applications manage the messaging queue in their machine, and the data to be communicated is placed in the queue and the receiver pulls the data from this queue. It operates in a decentralized mode.

How does It work? The sending application submits the message to exchange and forgets it. Like the post office, the exchange sends it to the receivers queue exchange from where the receiver pulls the information. Exchange does not store any messages in it. RabbitMQ thrives on the exchange model. MSMQ is a default feature in windows, and it should be enabled on both the sender and receiver

side. Sending machine to have full control over the messages placed in the queue and messages remain in queue till the receiver machine becomes active.

Messages are retrieved by the receiver as and when required by it. Exchange is a new concept in MSMQ.

Specializations Exchange enables sending the same message to multiple receivers simultaneously, and it supports the publish/subscribe model as an out of the box solution. It manages to send the messages to multiple queues and exchanges through bindings and routings. Exchange is configured to handle different types of communications.

1. Direct: One to one based on the routing key.
2. Fanout: One to many. Broadcasting model.
3. Topic: Messages to a group of receivers.
4. Headers: Instead of routing key, it relies on headers. MSMQ specializes in sending the incoming message to a receiver queue. MSMQ supports multicasting in its latest 3.0 version. There are few limitations in managing transactional messages in multicast resulting in data loss affecting delivery. Limited options like multicasting are only available in this MQ.

Queue Characteristics Queue Characteristics cannot be changed post its creation. Each queue has 2 major attributes: lifetime (Durability) and Auto delete. Since the sender does not manage the queues, messages expiry is managed by instructing the exchange to purge (dead lettering) messages in the exchange server. Normally dead lettering taking place under the following circumstances.

- a. Upon reaching the expiry limit
 - b. Rejected by receiver
 - c. The queue limit is exceeded
- Queue characteristics are fully under the control of the Sender application. MSMQ has an inbuilt facility for managing Dead lettering queues. It tracks the expired, un-retrieved and rejected messages and removes them periodically to save disk space.

Policy Queue characteristics cannot be changed once created, but they can be changed through policies. One needs to create the policies first and apply them to all queues wherever applicable. If there are multiple policies applied over a queue, the highest priority policy will be applied. Since the respective applications manage the queues, it is simpler to manage the policy on queues.

Distributed Brokers Normally it operates in a single broker environment where multiple hosts are connected to it. Managing multiple brokers in RabbitMQ is a little complicated, and this facility is not available as out of a box solution. It is managed by using Federation and Shovel as plugins to RabbitMQ. Federation is a one-way exchange connecting queues and exchanges. Shovel moves messages from one system to another system. The distributed broker's functionality is inbuilt in MSMQ and will not be visible to its users as a separate facility. It is managed by simply changing the receiver address in the queue manager in the local servers. Queues in MSMQ stores the messages until they are delivered to the receiver system. If the receiving system or network is down, the queue keeps on trying until the attempt is successful.

Development Aspects Creating a queue is just a declaration, and it is automatically created if it is not there, and the programmer needs not to check for its existence before declaring. The same things apply to exchange also. This has a message acknowledgement mechanism (ack – acknowledge, nack – Issues – negative) to track the status of the message sent to the broker through the network. Bindings can be short-circuited by declaring the destination in the routing key. A mandatory flag can be enabled to prevent an important message from getting lost. In MSMQ, the queue in respective applications needs to be set up by a configuration process, and it resides there permanently till deleted. MSMQ tracks the messages through its local services not applicable.

<https://www.educba.com/msmq-vs-rabbitmq/>

What is the Event Loop?

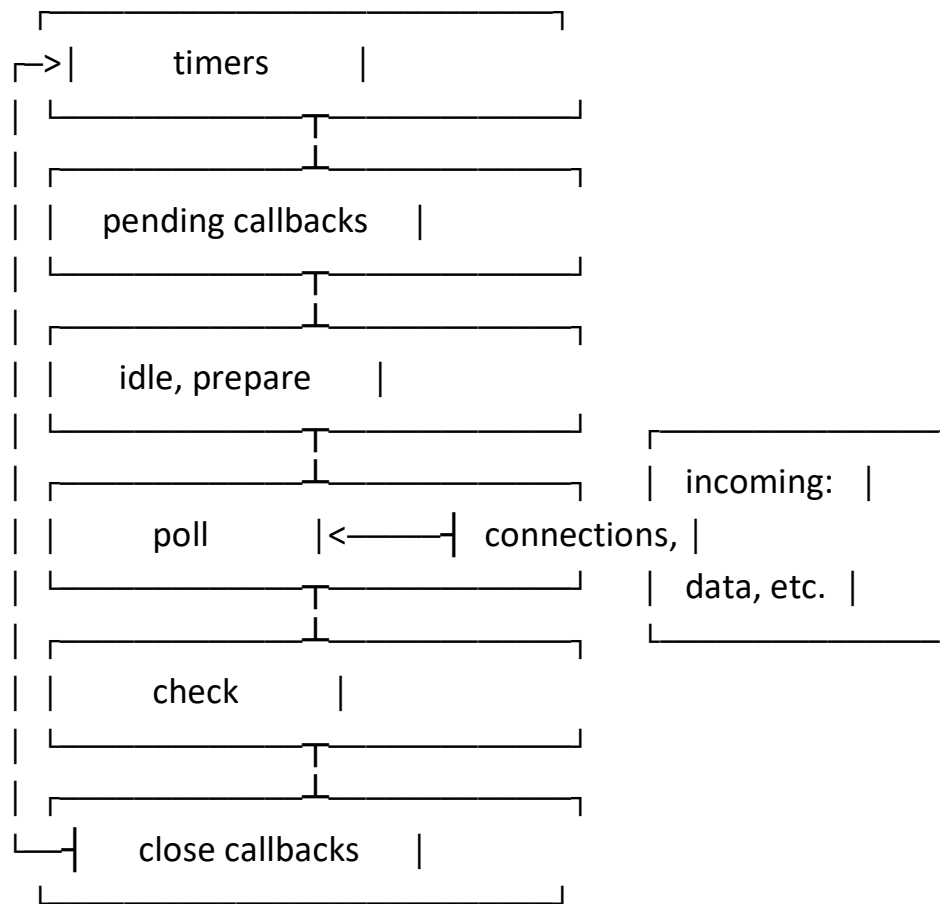
The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.

Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed. We'll explain this in further detail later in this topic.

Event Loop Explained

When Node.js starts, it initializes the event loop, processes the provided input script (or drops into the REPL, which is not covered in this document) which may make async API calls, schedule timers, or call `process.nextTick()`, then begins processing the event loop.

The following diagram shows a simplified overview of the event loop's order of operations.



Each box will be referred to as a "phase" of the event loop.

Each phase has a FIFO queue of callbacks to execute. While each phase is special in its own way, generally, when the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue until the queue has been exhausted or the maximum number of callbacks has executed. When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.

Since any of these operations may schedule more operations and new events processed in the poll phase are queued by the kernel, poll events can be queued

while polling events are being processed. As a result, long running callbacks can allow the poll phase to run much longer than a timer's threshold. See the timers and poll sections for more details.

There is a slight discrepancy between the Windows and the Unix/Linux implementation, but that's not important for this demonstration. The most important parts are here. There are actually seven or eight steps, but the ones we care about — ones that Node.js actually uses - are those above.

Phases Overview

timers: this phase executes callbacks scheduled by `setTimeout()` and `setInterval()`.

pending callbacks: executes I/O callbacks deferred to the next loop iteration.

idle, prepare: only used internally.

poll: retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and `setImmediate()`); node will block here when appropriate.

check: `setImmediate()` callbacks are invoked here.

close callbacks: some close callbacks, e.g. `socket.on('close', ...)`.

Between each run of the event loop, Node.js checks if it is waiting for any asynchronous I/O or timers and shuts down cleanly if there are not any.

Phases in Detail

timers

A timer specifies the threshold after which a provided callback may be executed rather than the exact time a person wants it to be executed. Timers callbacks will run as early as they can be scheduled after the specified amount of time has passed; however, Operating System scheduling or the running of other callbacks may delay them.

Technically, the poll phase controls when timers are executed.

For example, say you schedule a timeout to execute after a 100 ms threshold, then your script starts asynchronously reading a file which takes 95 ms:

```
const fs = require('fs');
function someAsyncOperation(callback) {
  // Assume this takes 95ms to complete
  fs.readFile('/path/to/file', callback);
}
```

```

const timeoutScheduled = Date.now();
setTimeout(() => {
  const delay = Date.now() - timeoutScheduled;
  console.log(`${delay}ms have passed since I was scheduled`);
}, 100);
// do someAsyncOperation which takes 95 ms to complete
someAsyncOperation(() => {
  const startCallback = Date.now();
  // do something that will take 10ms...
  while (Date.now() - startCallback < 10) {
    // do nothing
  }
});

```

When the event loop enters the poll phase, it has an empty queue (`fs.readFile()` has not completed), so it will wait for the number of ms remaining until the soonest timer's threshold is reached. While it is waiting 95 ms pass, `fs.readFile()` finishes reading the file and its callback which takes 10 ms to complete is added to the poll queue and executed. When the callback finishes, there are no more callbacks in the queue, so the event loop will see that the threshold of the soonest timer has been reached then wrap back to the timers phase to execute the timer's callback. In this example, you will see that the total delay between the timer being scheduled and its callback being executed will be 105ms.

To prevent the poll phase from starving the event loop, libuv (the C library that implements the Node.js event loop and all of the asynchronous behaviors of the platform) also has a hard maximum (system dependent) before it stops polling for more events.

pending callbacks

This phase executes callbacks for some system operations such as types of TCP errors. For example if a TCP socket receives `ECONNREFUSED` when attempting to connect, some *nix systems want to wait to report the error. This will be queued to execute in the pending callbacks phase.

poll

The poll phase has two main functions:

Calculating how long it should block and poll for I/O, then

Processing events in the poll queue.

When the event loop enters the poll phase and there are no timers scheduled, one of two things will happen:

If the poll queue is not empty, the event loop will iterate through its queue of callbacks executing them synchronously until either the queue has been exhausted, or the system-dependent hard limit is reached.

If the poll queue is empty, one of two more things will happen:

If scripts have been scheduled by `setImmediate()`, the event loop will end the poll phase and continue to the check phase to execute those scheduled scripts.

If scripts have not been scheduled by `setImmediate()`, the event loop will wait for callbacks to be added to the queue, then execute them immediately.

Once the poll queue is empty the event loop will check for timers whose time thresholds have been reached. If one or more timers are ready, the event loop will wrap back to the timers phase to execute those timers' callbacks.

check

This phase allows a person to execute callbacks immediately after the poll phase has completed. If the poll phase becomes idle and scripts have been queued with `setImmediate()`, the event loop may continue to the check phase rather than waiting.

`setImmediate()` is actually a special timer that runs in a separate phase of the event loop. It uses a libuv API that schedules callbacks to execute after the poll phase has completed.

Generally, as the code is executed, the event loop will eventually hit the poll phase where it will wait for an incoming connection, request, etc. However, if a callback has been scheduled with `setImmediate()` and the poll phase becomes idle, it will end and continue to the check phase rather than waiting for poll events.

close callbacks

If a socket or handle is closed abruptly (e.g. `socket.destroy()`), the 'close' event will be emitted in this phase. Otherwise it will be emitted via `process.nextTick()`.

`setImmediate()` vs `setTimeout()`

`setImmediate()` and `setTimeout()` are similar, but behave in different ways depending on when they are called.

`setImmediate()` is designed to execute a script once the current poll phase completes.

`setTimeout()` schedules a script to be run after a minimum threshold in ms has elapsed.

The order in which the timers are executed will vary depending on the context in which they are called. If both are called from within the main module, then timing will be bound by the performance of the process (which can be impacted by other applications running on the machine).

For example, if we run the following script which is not within an I/O cycle (i.e. the main module), the order in which the two timers are executed is non-deterministic, as it is bound by the performance of the process:

```
// timeout_vs_immediate.js
setTimeout(() => {
  console.log('timeout');
}, 0);
setImmediate(() => {
  console.log('immediate');
});
$ node timeout_vs_immediate.js
timeout
immediate
$ node timeout_vs_immediate.js
immediate
timeout
```

However, if you move the two calls within an I/O cycle, the immediate callback is always executed first:

```
// timeout_vs_immediate.js
const fs = require('fs');
fs.readFile(__filename, () => {
  setTimeout(() => {
    console.log('timeout');
  });
});
```

```

    }, 0);
    setImmediate(() => {
      console.log('immediate');
    });
  });
$ node timeout_vs_immediate.js
immediate
timeout
$ node timeout_vs_immediate.js
immediate
timeout

```

The main advantage to using `setImmediate()` over `setTimeout()` is `setImmediate()` will always be executed before any timers if scheduled within an I/O cycle, independently of how many timers are present.

`process.nextTick()`

Understanding `process.nextTick()`

You may have noticed that `process.nextTick()` was not displayed in the diagram, even though it's a part of the asynchronous API. This is because `process.nextTick()` is not technically part of the event loop. Instead, the `nextTickQueue` will be processed after the current operation is completed, regardless of the current phase of the event loop. Here, an operation is defined as a transition from the underlying C/C++ handler, and handling the JavaScript that needs to be executed. Looking back at our diagram, any time you call `process.nextTick()` in a given phase, all callbacks passed to `process.nextTick()` will be resolved before the event loop continues. This can create some bad situations because it allows you to "starve" your I/O by making recursive `process.nextTick()` calls, which prevents the event loop from reaching the poll phase.

Why would that be allowed?

Why would something like this be included in Node.js? Part of it is a design philosophy where an API should always be asynchronous even where it doesn't have to be. Take this code snippet for example:

```

function apiCall(arg, callback) {
  if (typeof arg !== 'string')

```

```

    return process.nextTick(
      callback,
      new TypeError('argument should be string')
    );
  }
}

```

The snippet does an argument check and if it's not correct, it will pass the error to the callback. The API updated fairly recently to allow passing arguments to `process.nextTick()` allowing it to take any arguments passed after the callback to be propagated as the arguments to the callback so you don't have to nest functions.

What we're doing is passing an error back to the user but only after we have allowed the rest of the user's code to execute. By using `process.nextTick()` we guarantee that `apiCall()` always runs its callback after the rest of the user's code and before the event loop is allowed to proceed. To achieve this, the JS call stack is allowed to unwind then immediately execute the provided callback which allows a person to make recursive calls to `process.nextTick()` without reaching a `RangeError: Maximum call stack size exceeded` from v8.

This philosophy can lead to some potentially problematic situations. Take this snippet for example:

```

let bar;
// this has an asynchronous signature, but calls callback synchronously
function someAsyncApiCall(callback) {
  callback();
}
// the callback is called before `someAsyncApiCall` completes.
someAsyncApiCall(() => {
  // since someAsyncApiCall hasn't completed, bar hasn't been assigned any value
  console.log('bar', bar); // undefined
});
bar = 1;

```

The user defines `someAsyncApiCall()` to have an asynchronous signature, but it actually operates synchronously. When it is called, the callback provided to `someAsyncApiCall()` is called in the same phase of the event loop because

`someAsyncApiCall()` doesn't actually do anything asynchronously. As a result, the callback tries to reference `bar` even though it may not have that variable in scope yet, because the script has not been able to run to completion.

By placing the callback in a `process.nextTick()`, the script still has the ability to run to completion, allowing all the variables, functions, etc., to be initialized prior to the callback being called. It also has the advantage of not allowing the event loop to continue. It may be useful for the user to be alerted to an error before the event loop is allowed to continue. Here is the previous example using

`process.nextTick()`:

```
let bar;
function someAsyncApiCall(callback) {
  process.nextTick(callback);
}
someAsyncApiCall(() => {
  console.log('bar', bar); // 1
});
bar = 1;
```

Here's another real world example:

```
const server = net.createServer(() => {}).listen(8080);
server.on('listening', () => {});
```

When only a port is passed, the port is bound immediately. So, the 'listening' callback could be called immediately. The problem is that the `.on('listening')` callback will not have been set by that time.

To get around this, the 'listening' event is queued in a `nextTick()` to allow the script to run to completion. This allows the user to set any event handlers they want.

`process.nextTick()` vs `setImmediate()`

We have two calls that are similar as far as users are concerned, but their names are confusing.

`process.nextTick()` fires immediately on the same phase

`setImmediate()` fires on the following iteration or 'tick' of the event loop

In essence, the names should be swapped. `process.nextTick()` fires more immediately than `setImmediate()`, but this is an artifact of the past which is

unlikely to change. Making this switch would break a large percentage of the packages on npm. Every day more new modules are being added, which means every day we wait, more potential breakages occur. While they are confusing, the names themselves won't change.

We recommend developers use `setImmediate()` in all cases because it's easier to reason about.

Why use `process.nextTick()`?

There are two main reasons:

Allow users to handle errors, cleanup any then unneeded resources, or perhaps try the request again before the event loop continues.

At times it's necessary to allow a callback to run after the call stack has unwound but before the event loop continues.

One example is to match the user's expectations. Simple example:

```
const server = net.createServer();
server.on('connection', (conn) => {});
server.listen(8080);
server.on('listening', () => {});
```

Say that `listen()` is run at the beginning of the event loop, but the listening callback is placed in a `setImmediate()`. Unless a hostname is passed, binding to the port will happen immediately. For the event loop to proceed, it must hit the poll phase, which means there is a non-zero chance that a connection could have been received allowing the connection event to be fired before the listening event.

Another example is inheriting from `EventEmitter` and emitting an event from within the constructor:

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {
  constructor() {
    super();
    this.emit('event');
  }
}
const myEmitter = new MyEmitter();
```

```
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
```

You can't emit an event from the constructor immediately because the script will not have processed to the point where the user assigns a callback to that event. So, within the constructor itself, you can use `process.nextTick()` to set a callback to emit the event after the constructor has finished, which provides the expected results:

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {
  constructor() {
    super();
    // use nextTick to emit the event once a handler is assigned
    process.nextTick(() => {
      this.emit('event');
    });
  }
}
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
```

<https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/#:~:text=The%20event%20loop%20is%20what,operations%20executing%20in%20the%20background.>

Blocking vs Non-blocking

Blocking refers to operations that block further execution until that operation finishes while non-blocking refers to code that doesn't block execution. Or as Node.js docs puts it, blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes.

Blocking methods execute synchronously while non-blocking methods execute asynchronously.

// Blocking

```

const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
console.log(data);
moreWork(); // will run after console.log
// Non-blocking
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log

```

In the first example above, `console.log` will be called before `moreWork()`. In the second example `fs.readFile()` is non-blocking so JavaScript execution can continue and `moreWork()` will be called first.

In Node, non-blocking primarily refers to I/O operations, and JavaScript that exhibits poor performance due to being CPU intensive rather than waiting on a non-JavaScript operation, such as I/O, isn't typically referred to as blocking. All of the I/O methods in the Node.js standard library provide async versions, which are non-blocking, and accept callback functions. Some methods also have blocking counterparts, which have names that end with `Sync`.

Non-blocking I/O operations allow a single process to serve multiple requests at the same time. Instead of the process being blocked and waiting for I/O operations to complete, the I/O operations are delegated to the system, so that the process can execute the next piece of code. Non-blocking I/O operations provide a callback function that is called when the operation is completed.

Callbacks

A callback is a function passed as an argument into another function, which can then be invoked (called back) inside the outer function to complete some kind of action at a convenient time. The invocation may be immediate (sync callback) or it might happen at a later time (async callback).

```

// Sync callback
function greetings(callback) {
  callback();
}

```



```

}
greetings(() => { console.log('Hi'); });
moreWork(); // will run after console.log
// Async callback
const fs = require('fs');
fs.readFile('/file.md', function callback(err, data) { // fs.readFile is an async
method provided by Node
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log

```

In the first example, the callback function is called immediately within the outer greetings function and logs to the console before moreWork() proceeds.

In the second example, fs.readFile (an async method provided by Node) reads the file and when it finishes it calls the callback function with an error or the file content. In the meantime the program can continue code execution.

An async callback may be called when an event happens or when a task completes. It prevents blocking by allowing other code to be executed in the meantime.

Instead of the code reading top to bottom procedurally, async programs may execute different functions at different times based on the order and speed that earlier functions like http requests or file system reads happen. They are used when you don't know when some async operation will complete.

You should avoid "callback hell", a situation where callbacks are nested within other callbacks several levels deep, making the code difficult to understand, maintain and debug.

Events and event-driven programming

Events are actions generated by the user or the system, like a click, a completed file download, or a hardware or software error.

Event-driven programming is a programming paradigm in which the flow of the program is determined by events. An event-driven program performs actions in response to events. When an event occurs it triggers a callback function.

Now, let's try to understand Node and see how all these relate to it.

Node.js: what is it, why was it created, and how does it work?

Simply put, Node.js is a platform that executes server-side JavaScript programs that can communicate with I/O sources like networks and file systems.

<https://www.freecodecamp.org/news/node-js-what-when-where-why-how-ab8424886e2/#:~:text=Non%2Dblocking%20I%2FO%20operations,the%20next%20piece%20of%20code.>

Non-blocking I/O operations allow a single process to serve multiple requests at the same time. Instead of the process being blocked and waiting for I/O operations to complete, the I/O operations are delegated to the system, so that the process can execute the next piece of code. Non-blocking I/O operations provide a callback function that is called when the operation is completed.

Callbacks

A callback is a function passed as an argument into another function, which can then be invoked (called back) inside the outer function to complete some kind of action at a convenient time. The invocation may be immediate (sync callback) or it might happen at a later time (async callback).

// Sync callback

```
function greetings(callback) {
  callback();
}
greetings(() => { console.log('Hi'); });
moreWork(); // will run after console.log
```

// Async callback

```
const fs = require('fs');
fs.readFile('/file.md', function callback(err, data) { // fs.readFile is an async
method provided by Node
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log
```

In the first example, the callback function is called immediately within the outer greetings function and logs to the console before moreWork() proceeds.

In the second example, `fs.readFile` (an async method provided by Node) reads the file and when it finishes it calls the callback function with an error or the file content. In the meantime the program can continue code execution.

<https://www.freecodecamp.org/news/node-js-what-when-where-why-how-ab8424886e2/#:~:text=Non%2Dblocking%20I%2FO%20operations,the%20next%20piece%20of%20code.>

In this getting started exercise, you create a Lambda function using the console. The function uses the default code that Lambda creates. The Lambda console provides a code editor for non-compiled languages that lets you modify and test code quickly. For compiled languages, you must create a .zip archive deployment package to upload your Lambda function code.

You can create a web API with an HTTP endpoint for your Lambda function by using Amazon API Gateway. API Gateway provides tools for creating and documenting web APIs that route HTTP requests to Lambda functions. You can secure access to your API with authentication and authorization controls. Your APIs can serve traffic over the internet or can be accessible only within your VPC. sqs is simple queue service it is essential for transferring the photos like 20 million photo processes using sqs service

AWS is a cloud computing service offered by Amazon. AWS lets you build, test, deploy and manage applications and services. All this is done via the data-centers and the hardware managed by Amazon. AWS provides you a combination of Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) offerings.

You can use AWS to create Virtual Machines which can be armed with processing power, storage capacity, and analytics along with networking and device management. AWS offers you a pay-as-you-go model, which helps to avoid upfront costs and pay based on the usage monthly.

Find the list of the top asked AWS Interview Questions and answers below.

AWS Basic Interview Questions

1. What is EC2?

EC2, a Virtual Machine in the cloud on which you have OS-level control. You can run this cloud server whenever you want and can be used when you need to deploy your own servers in the cloud, similar to your on-premises servers, and when you want to have full control over the choice of hardware and the updates on the machine.

2. What is SnowBall?

SnowBall is a small application that enables you to transfer terabytes of data inside and outside of the AWS environment.

AWS Snowball

3. What is CloudWatch?

CloudWatch helps you to monitor AWS environments like EC2, RDS Instances, and CPU utilization. It also triggers alarms depending on various metrics.

AWS Cloudwatch

You can download a PDF version of Aws Interview Questions.

[Download PDF](#)

4. What is Elastic Transcoder?

Elastic Transcoder is an AWS Service Tool that helps you in changing a video's format and resolution to support various devices like tablets, smartphones, and laptops of different resolutions.

5. What do you understand by VPC?

VPC stands for Virtual Private Cloud. It allows you to customize your networking configuration. VPC is a network that is logically isolated from other networks in the cloud. It allows you to have your private IP Address range, internet gateways, subnets, and security groups.

6. DNS and Load Balancer Services come under which type of Cloud Service?

DNS and Load Balancer are a part of IaaS-Storage Cloud Service.

7. What are the Storage Classes available in Amazon S3?

Storage Classes available with Amazon S3 are:

Amazon S3 Standard

Amazon S3 Standard-Infrequent Access

Amazon S3 Reduced Redundancy Storage

Amazon Glacier

8. Explain what T2 instances are?

T2 Instances are designed to provide moderate baseline performance and the capability to burst to higher performance as required by the workload.

9. What are Key-Pairs in AWS?

Key-Pairs are secure login information for your Virtual Machines. To connect to the instances, you can use Key-Pairs which contain a Public Key and a Private Key.

10. How many Subnets can you have per VPC?

You can have 200 Subnets per VPC.

11. List different types of Cloud Services.

Different types of Cloud Services are:

Software as a Service (SaaS)

Data as a Service (DaaS)

Platform as a Service (PaaS)

Infrastructure as a Service (IaaS)

Advanced AWS Questions

12. Explain what S3 is?

S3 stands for Simple Storage Service. You can use the S3 interface to store and retrieve any amount of data, at any time and from anywhere on the web. For S3, the payment model is “pay as you go”.

13. How does Amazon Route 53 provide high availability and low latency?

Amazon Route 53 uses the following to provide high availability and low latency:

Globally Distributed Servers - Amazon is a global service and consequently has DNS Servers globally. Any customer creating a query from any part of the world gets to reach a DNS Server local to them that provides low latency.

Dependency - Route 53 provides a high level of dependability required by critical applications.

Optimal Locations - Route 53 serves the requests from the nearest data center to the client sending the request. AWS has data-centers across the world. The data can be cached on different data-centers located in different regions of the world depending on the requirements and the configuration chosen. Route 53 enables any server in any data-center which has the required data to respond. This way, it

enables the nearest server to serve the client request, thus reducing the time taken to serve.

Amazon Route

As can be seen in the above image, the requests coming from a user in India are served from the Singapore Server, while the requests coming from a user in the US are routed to Oregon region.

14. How can you send a request to Amazon S3?

Amazon S3 is a REST Service, and you can send a request by using the REST API or the AWS SDK wrapper libraries that wrap the underlying Amazon S3 REST API.

15. What does AMI include?

An AMI includes the following things:

A template for the root volume for the instance.

Launch permissions to decide which AWS accounts can avail the AMI to launch instances.

A block device mapping that determines the volumes to attach to the instance when it is launched.

16. What are the different types of Instances?

Following are the types of instances:

Compute Optimized

Memory-Optimized

Storage Optimized

Accelerated Computing

General Purpose

17. What is the relation between the Availability Zone and Region?

An AWS Availability Zone is a physical location where an Amazon data center is located. On the other hand, an AWS Region is a collection or group of Availability Zones or Data Centers.

This setup helps your services to be more available as you can place your VMs in different data centers within an AWS Region. If one of the data centers fails in a Region, the client requests still get served from the other data centers located in the same Region. This arrangement, thus, helps your service to be available even if a Data Center goes down.

18. How do you monitor Amazon VPC?

You can monitor Amazon VPC using:

CloudWatch

VPC Flow Logs

19. What are the different types of EC2 instances based on their costs?

The three types of EC2 instances based on the costs are:

On-Demand Instance - These instances are prepared as and when needed.

Whenever you feel the need for a new EC2 instance, you can go ahead and create an on-demand instance. It is cheap for the short-time but not when taken for the long term.

Spot Instance - These types of instances can be bought through the bidding model. These are comparatively cheaper than On-Demand Instances.

Reserved Instance - On AWS, you can create instances that you can reserve for a year or so. These types of instances are especially useful when you know in advance that you will be needing an instance for the long term. In such cases, you can create a reserved instance and save heavily on costs.

20. What do you understand by stopping and terminating an EC2 Instance?

Stopping an EC2 instance means to shut it down as you would normally do on your Personal Computer. This will not delete any volumes attached to the instance and the instance can be started again when needed.

On the other hand, terminating an instance is equivalent to deleting an instance. All the volumes attached to the instance get deleted and it is not possible to restart the instance if needed at a later point in time.

21. What are the consistency models for modern DBs offered by AWS?

Eventual Consistency - It means that the data will be consistent eventually, but may not be immediate. This will serve the client requests faster, but chances are that some of the initial read requests may read the stale data. This type of consistency is preferred in systems where data need not be real-time. For example, if you don't see the recent tweets on Twitter or recent posts on Facebook for a couple of seconds, it is acceptable.

Strong Consistency - It provides an immediate consistency where the data will be consistent across all the DB Servers immediately. Accordingly. This model may take some time to make the data consistent and subsequently start serving the

requests again. However, in this model, it is guaranteed that all the responses will always have consistent data.

22. What is Geo-Targeting in CloudFront?

Geo-Targeting enables the creation of customized content based on the geographic location of the user. This allows you to serve the content which is more relevant to a user. For example, using Geo-Targeting, you can show the news related to local body elections to a user sitting in India, which you may not want to show to a user sitting in the US. Similarly, the news related to Baseball Tournament can be more relevant to a user sitting in the US, and not so relevant for a user sitting in India.

23. What are the advantages of AWS IAM?

AWS IAM enables an administrator to provide granular level access to different users and groups. Different users and user groups may need different levels of access to different resources created. With IAM, you can create roles with specific access-levels and assign the roles to the users.

It also allows you to provide access to the resources to users and applications without creating the IAM Roles, which is known as Federated Access.

24. What do you understand by a Security Group?

When you create an instance in AWS, you may or may not want that instance to be accessible from the public network. Moreover, you may want that instance to be accessible from some networks and not from others.

Security Groups are a type of rule-based Virtual Firewall using which you can control access to your instances. You can create rules defining the Port Numbers, Networks, or protocols from which you want to allow access or deny access.

25. What are Spot Instances and On-Demand Instances?

When AWS creates EC2 instances, there are some blocks of computing capacity and processing power left unused. AWS releases these blocks as Spot Instances. Spot Instances run whenever capacity is available. These are a good option if you are flexible about when your applications can run and if your applications can be interrupted.

On the other hand, On-Demand Instances can be created as and when needed. The prices of such instances are static. Such instances will always be available unless you explicitly terminate them.

26. Explain Connection Draining.

Connection Draining is a feature provided by AWS which enables your servers which are either going to be updated or removed, to serve the current requests. If Connection Draining is enabled, the Load Balancer will allow an outgoing instance to complete the current requests for a specific period but will not send any new request to it. Without Connection Draining, an outgoing instance will immediately go off and the requests pending on that instance will error out.

27. What is a Stateful and a Stateless Firewall?

A Stateful Firewall is the one that maintains the state of the rules defined. It requires you to define only inbound rules. Based on the inbound rules defined, it automatically allows the outbound traffic to flow.

On the other hand, a Stateless Firewall requires you to explicitly define rules for inbound as well as outbound traffic.

For example, if you allow inbound traffic from Port 80, a Stateful Firewall will allow outbound traffic to Port 80, but a Stateless Firewall will not do so.

28. What is a Power User Access in AWS?

An Administrator User will be similar to the owner of the AWS Resources. He can create, delete, modify or view the resources and also grant permissions to other users for the AWS Resources.

A Power User Access provides Administrator Access without the capability to manage the users and permissions. In other words, a user with Power User Access can create, delete, modify or see the resources, but he cannot grant permissions to other users.

29. What is an Instance Store Volume and an EBS Volume?

An Instance Store Volume is temporary storage that is used to store the temporary data required by an instance to function. The data is available as long as the instance is running. As soon as the instance is turned off, the Instance Store Volume gets removed and the data gets deleted.

On the other hand, an EBS Volume represents a persistent storage disk. The data stored in an EBS Volume will be available even after the instance is turned off.

30. What are Recovery Time Objective and Recovery Point Objective in AWS?

Recovery Time Objective - It is the maximum acceptable delay between the interruption of service and restoration of service. This translates to an acceptable time window when the service can be unavailable.

Recover Point Objective - It is the maximum acceptable amount of time since the last data restore point. It translates to the acceptable amount of data loss which lies between the last recovery point and the interruption of service.

31. Is there a way to upload a file that is greater than 100 Megabytes in Amazon S3?

Yes, it is possible by using the Multipart Upload Utility from AWS. With the Multipart Upload Utility, larger files can be uploaded in multiple parts that are uploaded independently. You can also decrease upload time by uploading these parts in parallel. After the upload is done, the parts are merged into a single object or file to create the original file from which the parts were created.

32. Can you change the Private IP Address of an EC2 instance while it is running or in a stopped state?

No, a Private IP Address of an EC2 instance cannot be changed. When an EC2 instance is launched, a private IP Address is assigned to that instance at the boot time. This private IP Address is attached to the instance for its entire lifetime and can never be changed.

33. What is the use of lifecycle hooks in Autoscaling?

Lifecycle hooks are used for Auto-scaling to put an additional wait time to a scale-in or a scale-out event.

34. What are the policies that you can set for your user's passwords?

Following are the policies that can be set for user's passwords:

You can set a minimum length of the password.

You can ask the users to add at least one number or special character to the password.

Assigning the requirements of particular character types, including uppercase letters, lowercase letters, numbers, and non-alphanumeric characters.

You can enforce automatic password expiration, prevent the reuse of old passwords, and request for a password reset upon their next AWS sign-in.

You can have the AWS users contact an account administrator when the user has allowed the password to expire.

Aws MCQs

AWS IoT Core lets you connect billions of IoT devices and route trillions of messages to AWS services without managing infrastructure.

Securely transmit messages to and from all of your IoT devices and applications with low latency and high throughput.