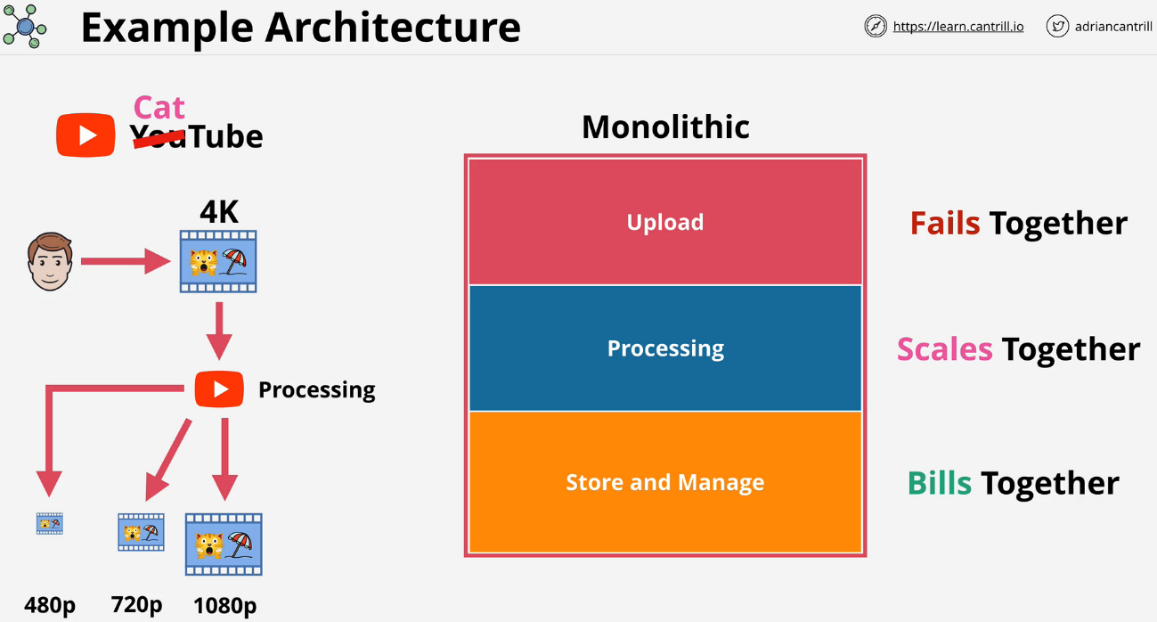
**Serverless-and-App-Services**

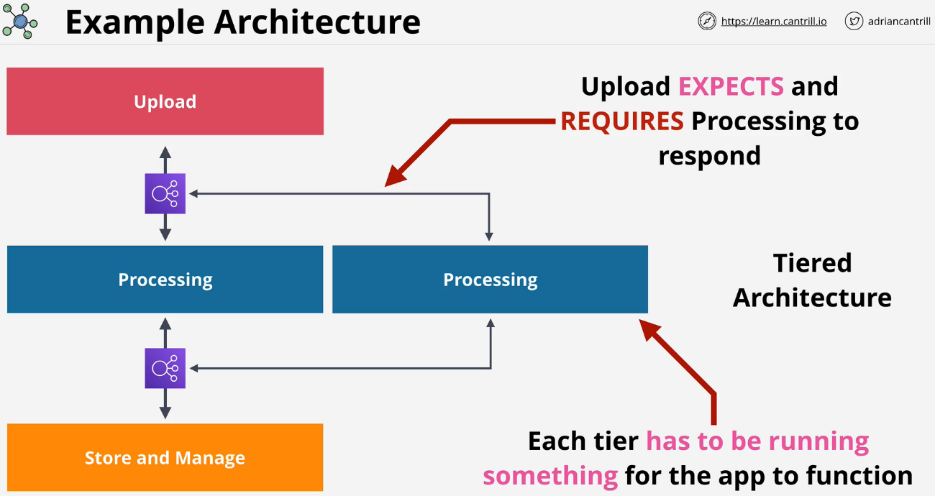
**Architecture Evolution**

**Monolithic**

* **Fails together**.
  + One error will bring the whole system down.
* **Scales together**.
  + Everything expects to be running on the same compute hardware
* **Bills together**.
  + All components are always running and always incurring charges.

This is the least cost effective way to architect systems.

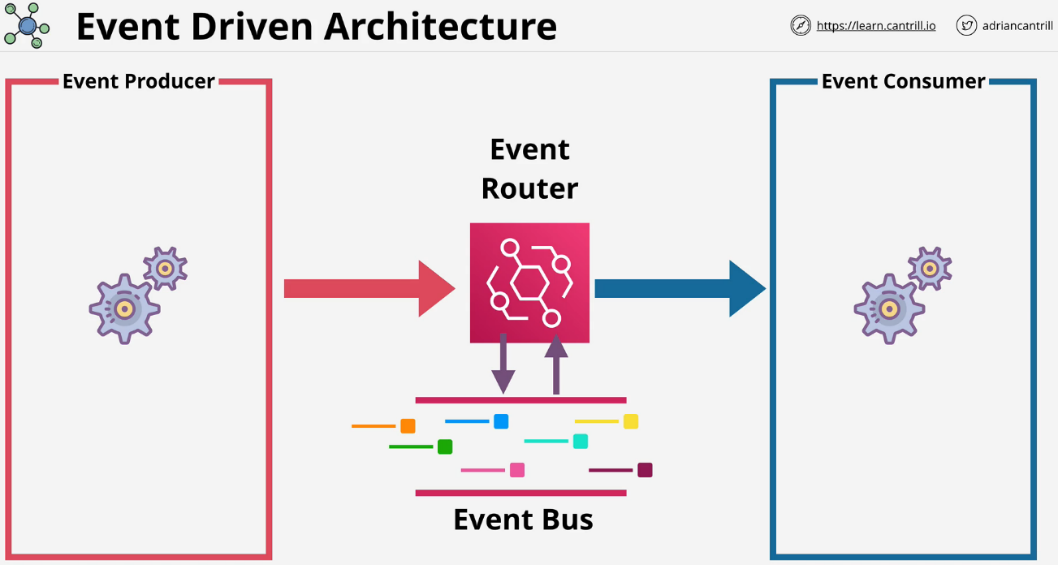
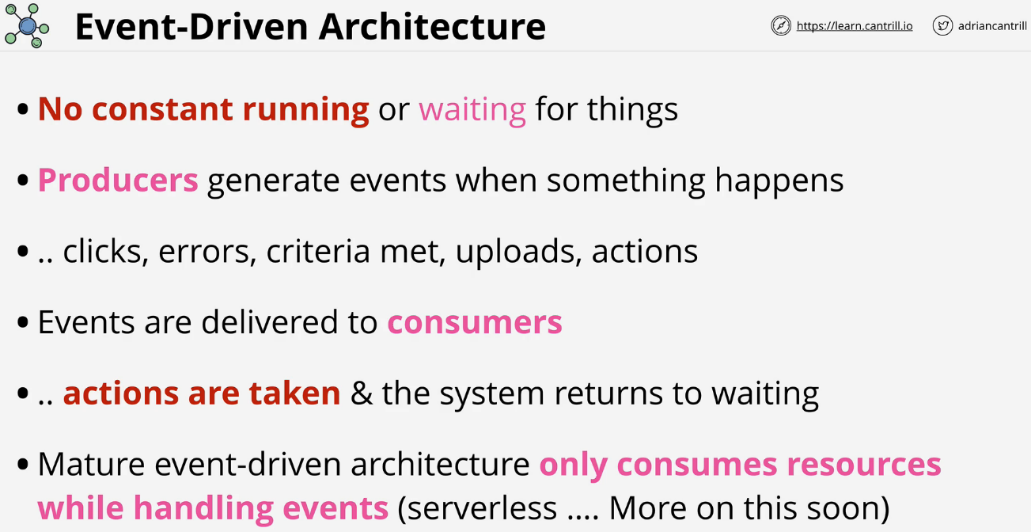
**Tiered**

* Different components can be on the same server or different servers.
* Components are coupled together because the endpoints connect together.
* Can adjust the size of the server that is running each application tier.
* Utilizes load balancers in between tiers to add capacity.
* Tiers are still **tightly coupled**.
  + Tiers expect a response from each other. If one tier fails, subsequent tiers will also fail because they will not receive the proper response.
  + Back loads in one tier will impact the other tiers and customer experience.
* Tiers must be **operational** and send responses **even if they are not processing anything** of value otherwise the system fails.

**Evolving with Queues**

* Data no longer moves between tiers to be processed and instead uses a queue.
  + Often are **FIFO** (first in, first out)
* Data **moves into a S3 bucket.**
* Detailed information is put into the next slot in the queue.
  + Tiers no longer expect an answer.
* Upload tier sends an async message.
  + The upload tier can add more messages to the queue.
* The queue will have an autoscaling group to increase processing capacity.
* The autoscaling group will only bring up servers as they are needed.
* The queue has the location of the S3 bucket and passes this onto the processing tier.

**Event Driven Architecture**

* **Event producers**
  + Interact with customers or systems monitoring components.
  + Produce events in reaction to something.
  + Clicks, events, errors, actions
* **Event consumers**
  + Pieces of software waiting for events to occur.
  + Actions are taken and the system returns to waiting
* Services can be producers and consumers at once.
* Resources are not waiting around to be used.
* **Event router** is needed for event driven architecture that also **manages an event bus.**
* Only **consumes resources while handling events.**

**AWS Lambda**

* Function-as-a-service (FaaS)
  + Service accepts functions.
* Event driven invocation (execution) based on an event occurring.
* **Lambda function** is piece of code in one language.
* Lambda functions use a **runtime** (e.g. Python 3.6)
* Runs in a **runtime environment**.
  + Virtual environment that is ready to go to run code in that language.
  + You are billed only for the duration a function runs.
  + There is no charge for having lambda functions waiting and ready to go.

**Lambda Architecture**

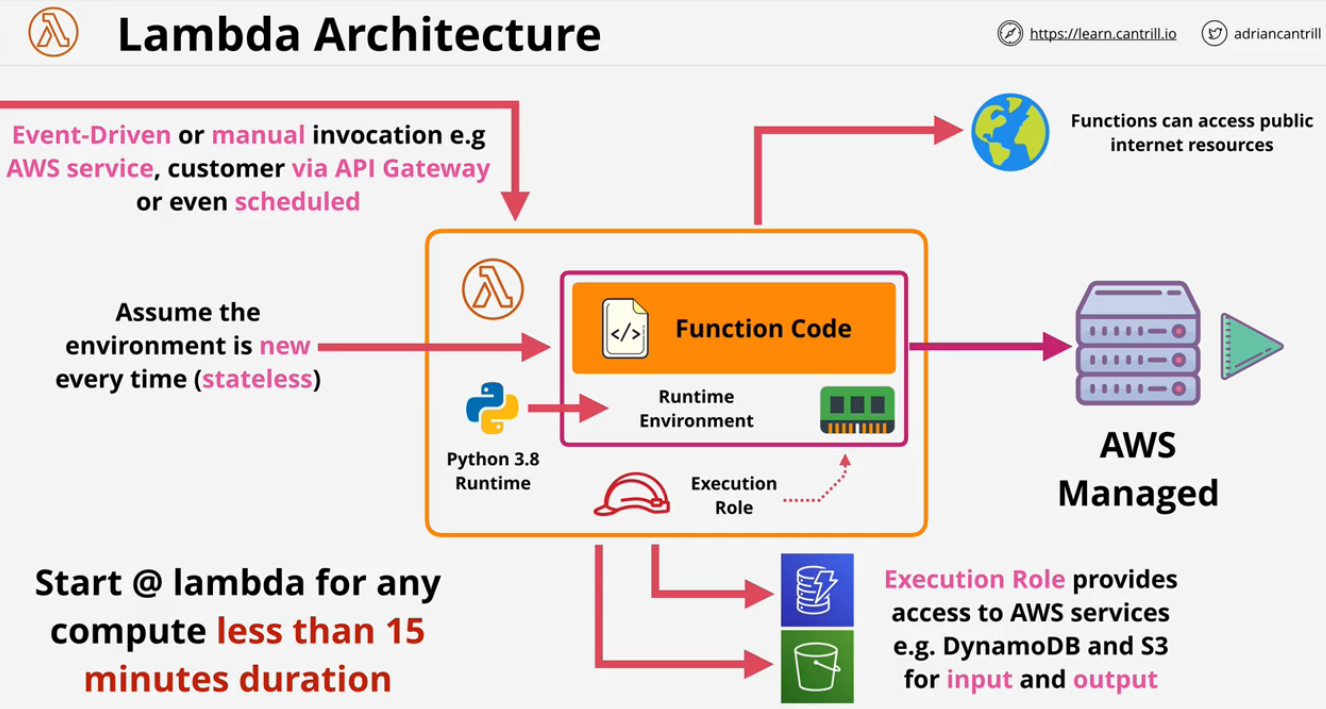
Best practice is to make it very small and very specialized. Lambda function code, when executed is known as being **invoked**. When invoked, it runs inside a runtime environment that matches the language the script is written in. The runtime environment is allocated a certain amount of memory and an appropriate amount of CPU. The more memory you allocate, the more CPU it gets, and the more the function costs to invoke per second.

Lambda functions can be given an IAM role or **execution role**. The execution role is passed into the runtime environment. Whenever that function executes, the code inside has access to whatever permissions the role's permission policy provides.

Lambda can be invoked in an **event-driven** or **manual** way. Each time you invoke a lambda function, the environment provided is new. **Never store anything inside the runtime environment, it is ephemeral**.

Lambda functions by default are public services and can access any websites. By default they cannot access private VPC resources, but can be configured to do so if needed. Once configured, they can only access resources within a VPC. **Unless you have configured your VPC to have all of the configuration needed to have public internet access or access to the AWS public space endpoints, then the Lambda will not have access.**

The **Lambda runtime is stateless**, so you should always use AWS services for input and output. Something like DynamoDB or S3. If a Lambda is invoked by an event, it gets details of the event given to it at startup.

Lambda functions can run up to **15 minutes**. That is the max limit.

**Key Considerations**

* Currently 15 min execution limit.
* Assume each execution gets a new runtime environment.
* Use the execution role which is assumed when needed.
* Always load data from other services from public APIs or S3.
* Store data to other services (e.g. S3).
* 1M free requests and 400,000 GB-seconds of compute per month.

**CloudWatch Events and EventBridge**

**Delivers near real time stream of system events that describe changes in AWS products and services**. EventBridge will replace CW Events.

**EventBridge can also handle events from third parties.** Both share the same underlying architecture. AWS is now encouraging a migration to EB.

**CloudWatch Events Key Concepts**

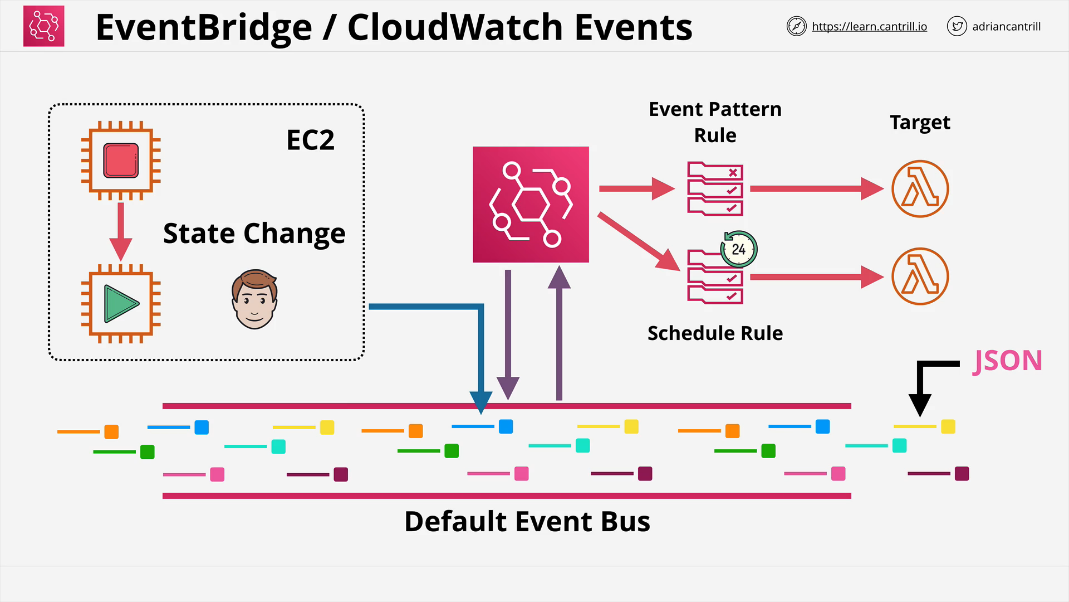
They can observe if X happens at Y time(s), do Z.

* X is a supported service which is a producer of an event.
* Y can be a certain time or time period.
* Z is a supported target service to deliver the event to.

EventBridge is basically CloudWatch Events V2 that uses the same underlying APIs and has the same architecture, but with additional features. Things created in one can be visible in the other for now.

Both systems have a **default Event bus** for a single AWS account. A bus is a stream of events which occur for any supported service inside an AWS account. In CW Events, there is only one bus (implicit), this is not exposed. **EventBridge can have additional event buses for your applications or third party applications and services.** These can be interacted with in the same way as the default bus.

In both services, you **create rules** and these rules **pattern match events** which occur on the **buses** and when they see an event which matches, they **deliver that event to a target**. Alternatively you can have **schedule based rules** which match a certain date and time or ranges of dates and times.

Rules match incoming events or schedules. The rule matches an event and routes that event to one or more targets as you define on that rule.

Architecturally at the heart of event bridge is the **default account event bus**. This is a stream of **events generated by supported services within the AWS account.** Rules are created and these are linked to a specific event bus or the default event bus. Once the rule completes pattern matching, the rule is executed and moves that event that it matched through to one or more targets. The events themselves are JSON structures and the data can be used by the targets.

**The Patterns inside Event Bridge can be defined as:**

**Event pattern:** Choosing Event Pattern makes this a rule that is triggered when an event matching the pattern occurs. You can **choose a pre-defined pattern that matches a certain type of event from a certain service, or you can create a custom pattern**. Even if you choose a pre-defined pattern, you can then edit the pattern to customize it.

**Schedule:** Choosing Schedule makes this a rule that is triggered regularly on the schedule you specify. You can use a **simple fixed-rate schedule that triggers every certain number of minutes, hour, or weeks, or you can use a cron expression to create a more fine-grained schedule**, such as “the first Monday of each month at 8am.”

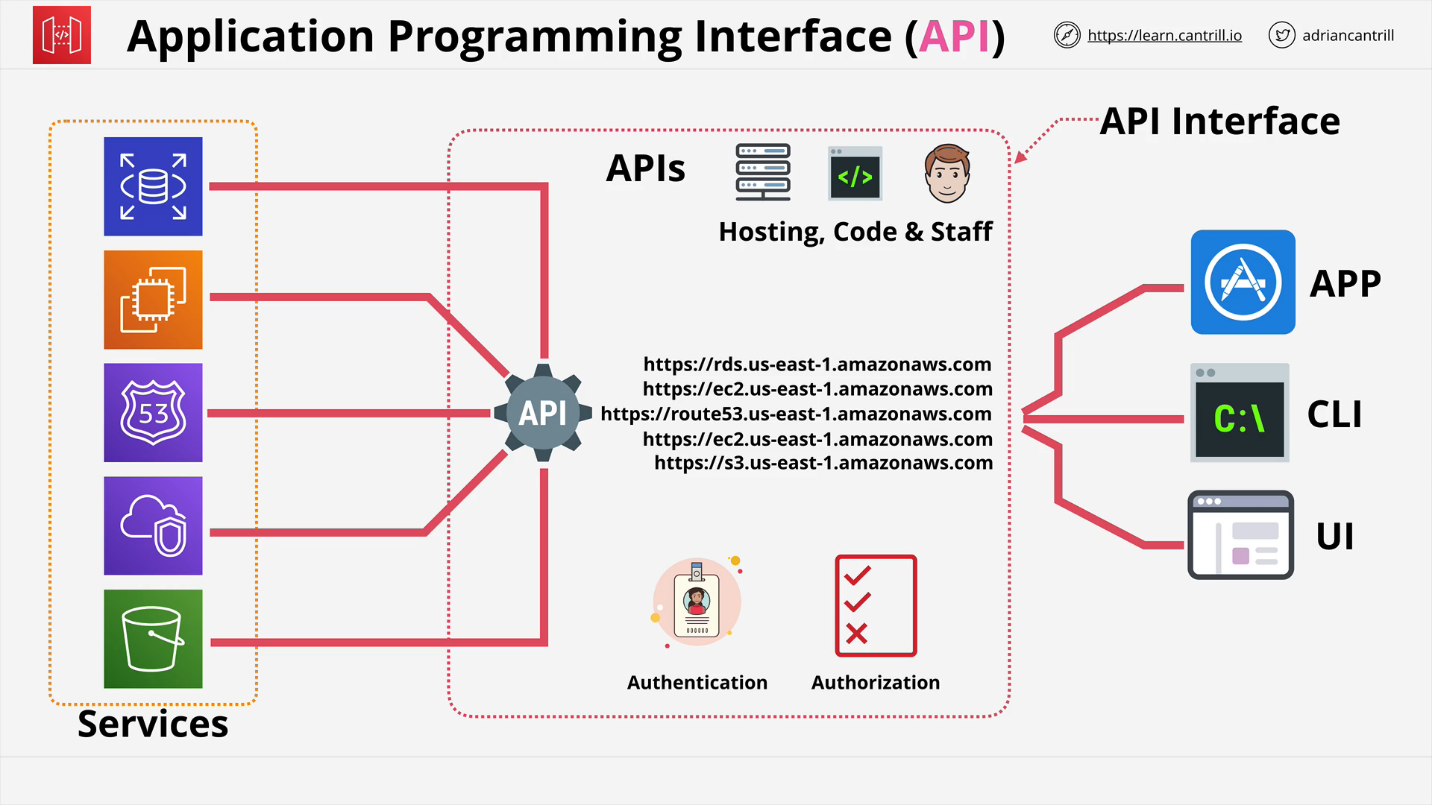
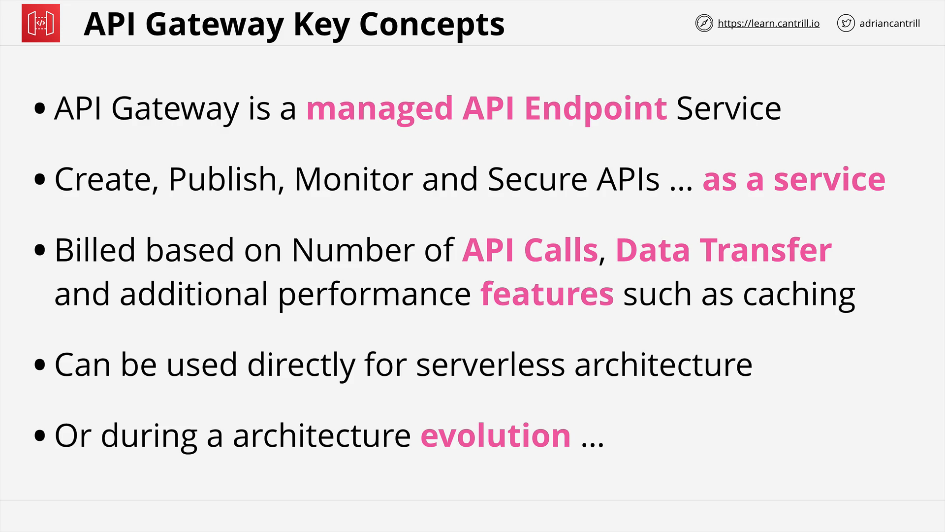
Schedules **are not supported on custom or partner event bus.**

**Cron Expression:**

**“55 16 ? \* \* \*”**

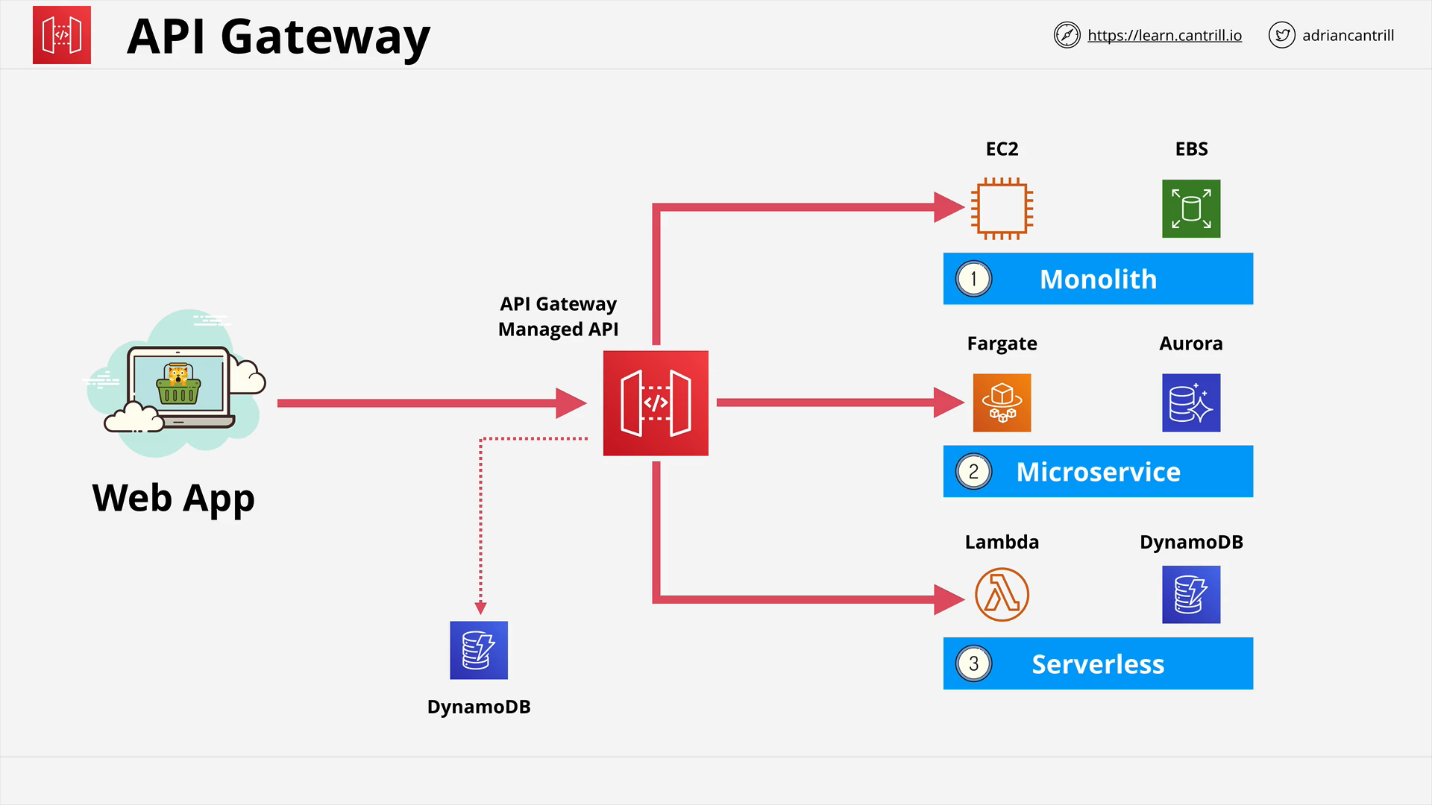
* “51” -> Minutes
* “18” -> Hours
* “?” -> Days of the Month
* “\*” -> Month
* “\*“-> Days of Week
* “\*” -> Year

**Application Programming Interface (API) Gateway**

* A way **applications or services can communicate with each other**.
* API gateway is an AWS managed service.
  + Provides managed AWS endpoints.
  + Can also **perform authentication to prove you are who you claim.**
  + You can create an API and present it to your customers for use.
* Allows **you to create, publish, monitor, and secure APIs.**
* **Billed** based on:
  + number of API calls
  + amount of data transferred
  + additional performance features such as caching
* **Serve as an entry point for serverless architecture.**
* If you have on premises legacy services that use APIs, this can be integrated.

Great during an architecture evolution because the endpoints don't change.

1. Create a **managed API and point at the existing monolithic application.**
2. Using **API gateway allows the business to evolve along the way slowly**. This might **move some of the data to fargate and aurora architecture**.
3. Move to a **full serverless architecture with DynamoDB**.



**Serverless**

This is not one single thing, you manage few if any servers. This **aims to remove overhead and risk as much as possible**. Applications are a collection of small and specialized functions that do one thing really well and then stop.

These functions are **stateless and run in ephemeral environments**. Every time they run, they obtain the data that they need, they do something and then optionally, they **store the result persistently somehow or deliver the output to something else.**

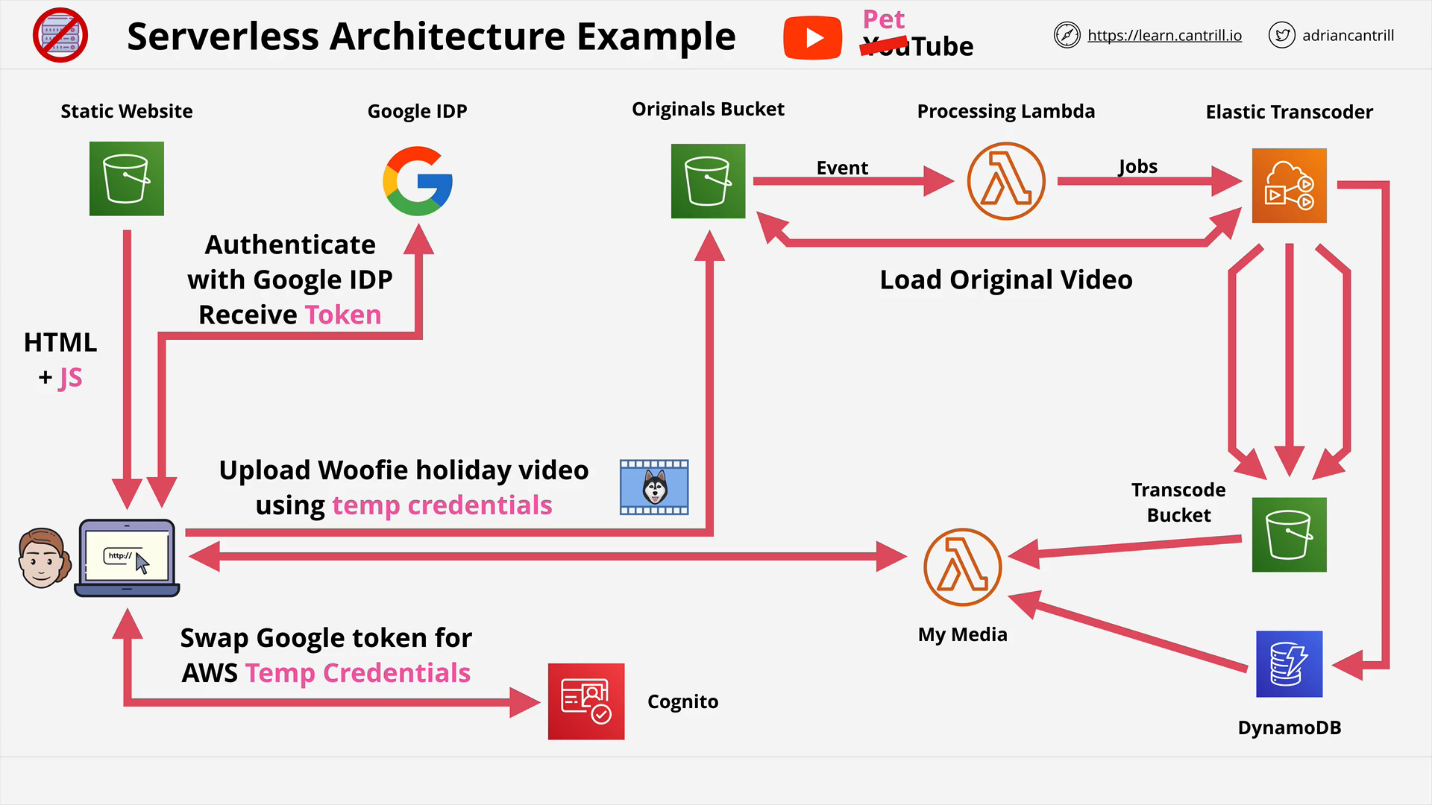
Generally, everything is event driven. Nothing is running until it's required. While **not being used, there should be little to no cost.**

Should use **managed services when possible.**

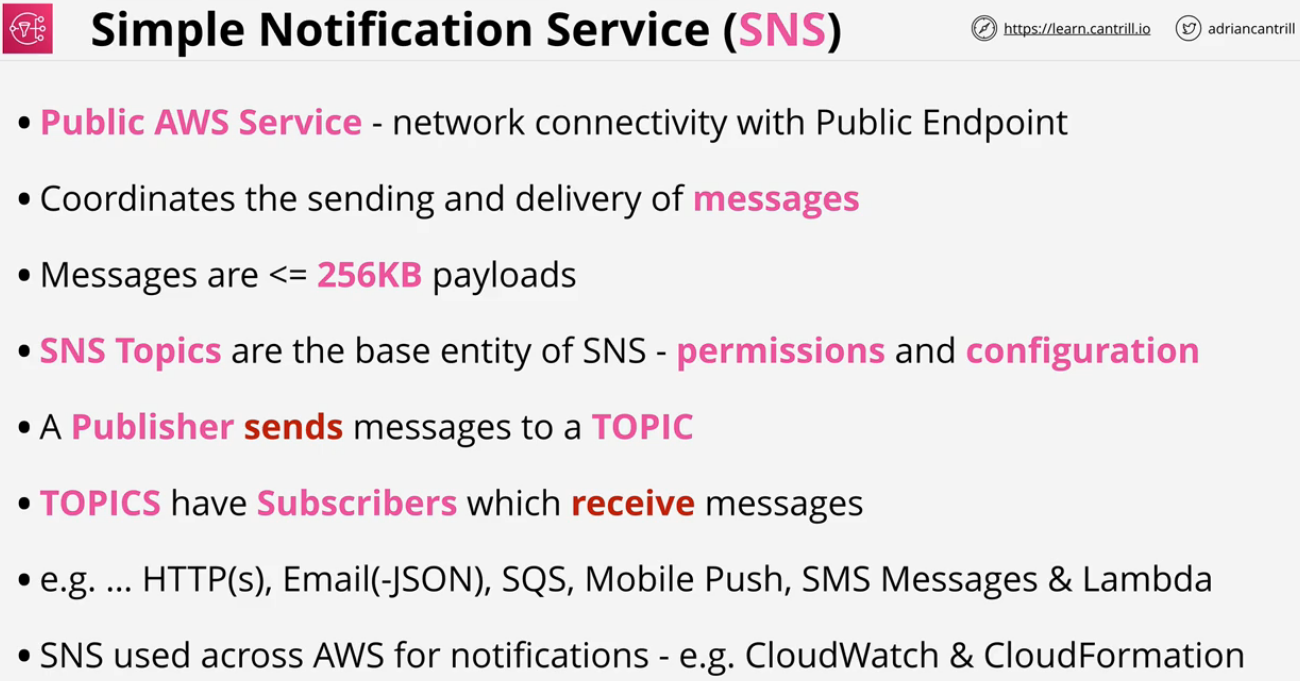
Aim is to **consume as a service whatever you can, code as little as possible, and use function as a service for any general purpose compute needs, and then use all of those building blocks together to create your application**.

**Example of Serverless**

A user wants to upload videos to a website for transcoding.

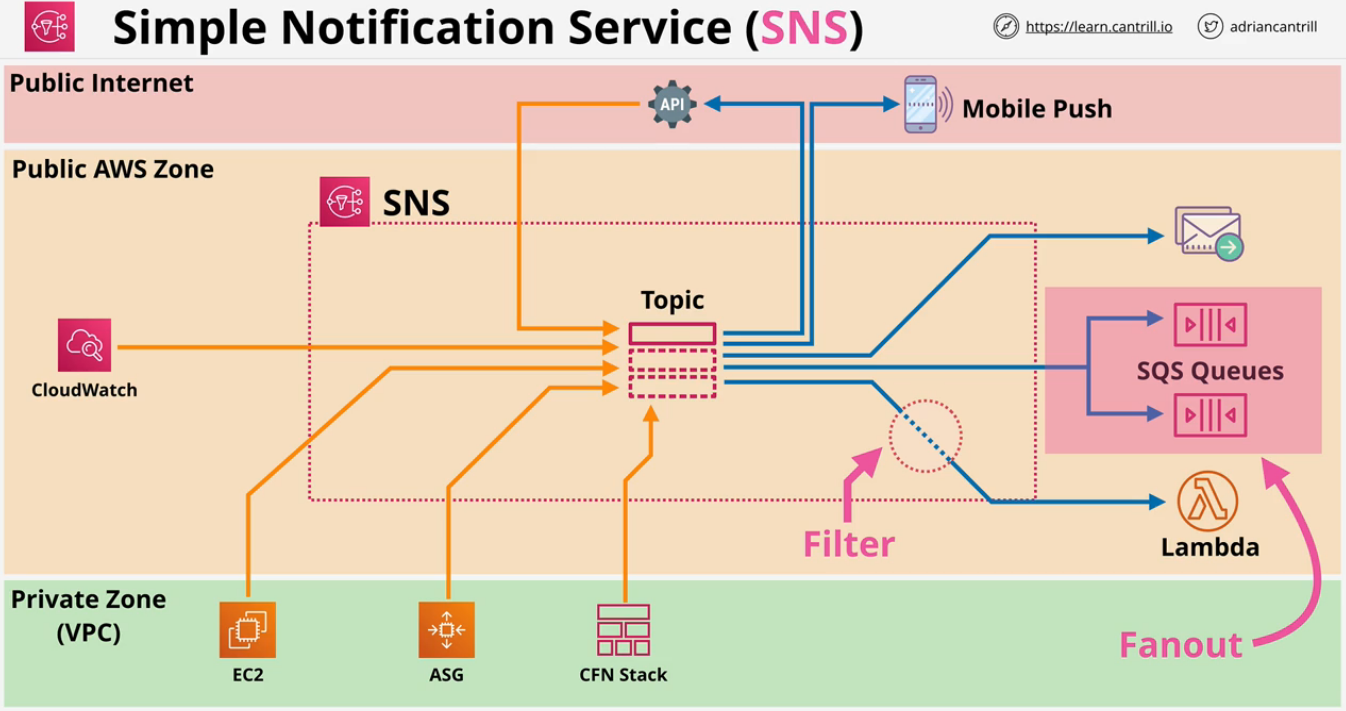
1. User browses to a static website that is running the uploader. The JS runs directly from the web browser.
2. Third party auth provider, google in this case, authenticates via **token**.
3. AWS cannot use tokens provided by third parties. **Cognito** is called to swap the third party token for AWS credentials.
4. Service uses these temporary credentials to upload a video to S3 bucket.
5. Bucket will generate an event once it has completed the upload.
6. A lambda triggers to transcode the video as needed. The transcoder will get the original S3 bucket video location and will use this for its workload.
7. Output will be added to a new transcode bucket and will put an entry into DynamoDB.
8. User can interact with another Lambda to pull the media from the transcode bucket using the DynamoDB entry.

**Simple Notification Service (SNS)**

* HA, Durable, **PUB/SUB** messaging service.
* **Public AWS service** meaning to access it, you need network connectivity with the Public AWS endpoints.
* Coordinates sending and delivering of messages up to **256KB** in size.
  + Messages are not designed for large binary files.
* **SNS topics** are the base entity of SNS.
  + Permissions are controlled and configuration for SNS is defined.
* **Publisher** sends messages to a topic.
  + Topics have subscribers which receive messages.
* **Subscribers** receive all of the messages sent to the Topic.
  + Subscribers can be HTTP and HTTPS endpoints, emails, or SQS queues.
  + Filters can be applied to limit messages sent to subscribers.
* **Fanout allows for a single SNS topic with multiple SQS queues as subscribers.**
  + Can create **multiple related workflows.**
  + Allows multiple SQS queues to process the workload in slightly different ways.

**SNS Offers:**

* Delivery Status including HTTP, Lambda, SQS
* Delivery retries which ensure reliable delivery
* HA and **Scalability** (**Regional**)
* **SSE** (server side encryption)
* **Topics can be used cross-account via Topic Policy(Resource Policy)**



**AWS Step Functions**

There are many problems with lambdas limitations that can be solved with a state machine. **A state machine is a workflow**. It has a **start point, end point** and in between there are **states**. **States** are things inside a **State Machine which can do things.** States can do things, and **take in data, modify data, and output data**.

State machine is designed to perform an **activity or workflow with lots of individual components and maintain the idea of data between those states.**

Maximum duration for a state machine execution is 1 year.

Two types of workflow

* **Standard**
  + Default
  + 1 year workflow
* **Express**
  + Designed for IOT or other high transaction uses
  + 5 minute workflow
  + Provides better processing guarantees

Started via **API Gateway, IOT Rules, EventBridge, Lambda**. Generally used for back end processing.

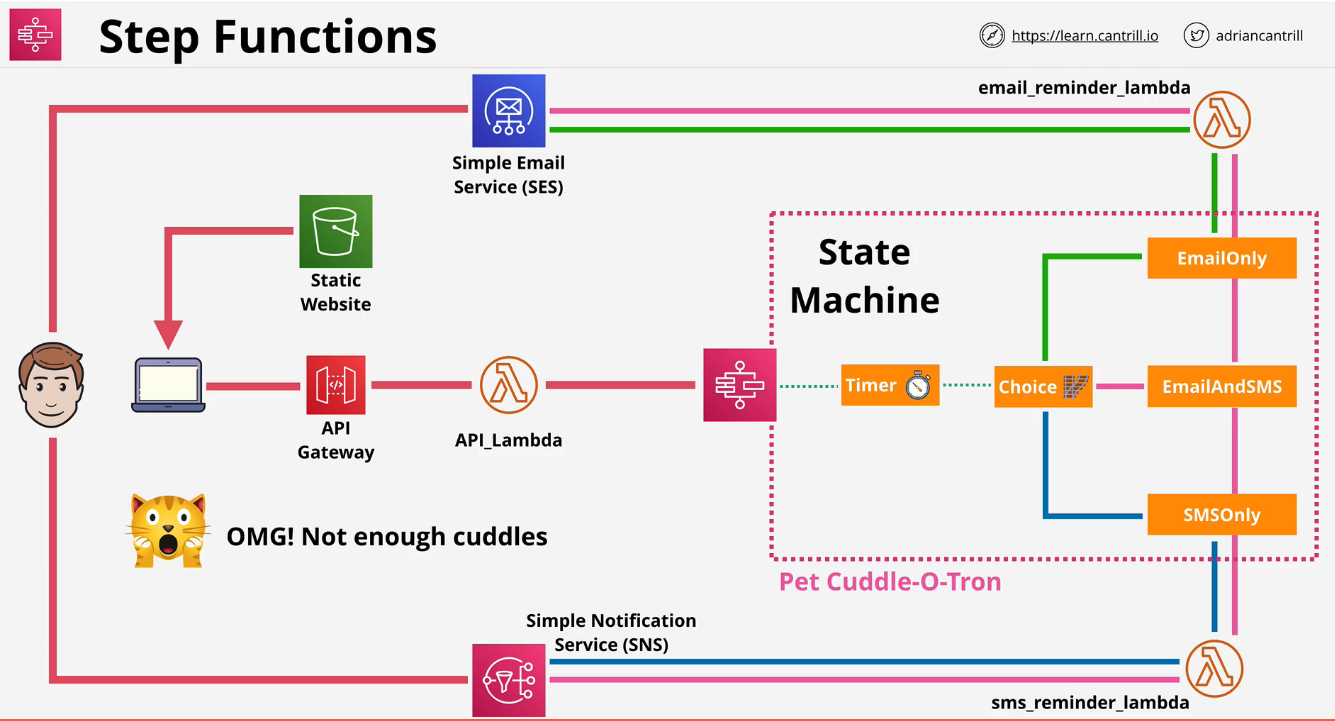
With State machines you can **use a template to create and export State Machines once they're configured to your liking, it's called Amazon States Language or ASL.** It's based on JSON.

**State machines** are **provided permission to interact with other AWS services via IAM roles**.

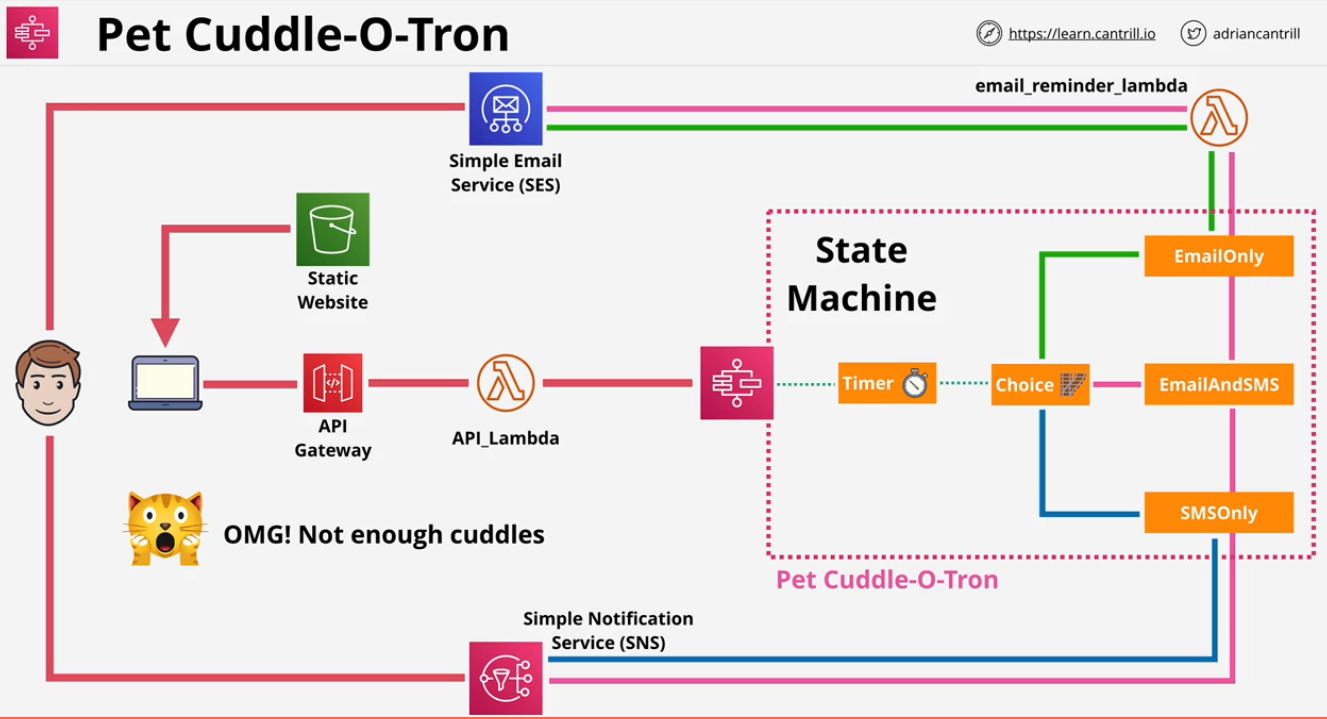
**Step Function States**

States are the things inside a workflow, the things which occur. These states are available.

* Succeed and Fail
  + the process will succeed or fail.
* Wait
  + will wait for a certain period of time
  + will wait until specific date and time
* Choice
  + different path is determined based on an import
* Parallel
  + will create parallel branches based on a choice
* Map
  + accepts a list of things
  + for each item in that list, performs an action or set of actions based on that particular item.
* Task
  + represents a single unit of work performed by the State Machine.
  + it allows the state machine to actually do things.
  + can be integrated with many different services such as **lambda, AWS batch, dynamoDB, ECS, SNS, SQS, Glue, SageMaker, EMR,** and lots of others.



Cuddletron Architecture:



**Simple Queue Service (SQS)**

**Public service** that provides **fully managed highly available message queues.**

* Replication happens within a region by default.
  + Messages are guaranteed in the order they were received
  + **Provides FIFO with best effort, but no guarantee**
* Messages up to **256KB in size.**
  + Should link to larger sets of data if needed.
* Polling is checking for any messages on the queue.
* **Visibility timeout**
  + The amount of time a client has to process a message in some way
  + When a client polls and receives messages, they aren't deleted from the queue and are **hidden for the length of this timeout.**
  + This is the amount of time that a client can wait to work on the messages.
  + If the client does not delete the message by the end, it will reappear in the queue.
* **Dead-letter queue**
  + if a **message is received multiple times but is unable to be finished, this puts it into a different workload to try and fix the corruption.**
* ASG can scale and lambdas can be invoked based on queue length.
* **Standard queue**
  + multi-lane HW
  + **guarantee the order and at least once delivery.**
* **FIFO queue**
  + single lane road with no way to overtake
  + **guarantee the order and at exactly once delivery**
  + **3,000 messages p/s with batching or up to 300 messages p/s without**

**Billing**

Billed on **requests** not messages. A request is a single request to SQS. One request can return **0 - 10 messages up to 64KB data in total**. Since requests can return 0 messages, **frequently polling a SQS Queue, makes it less effective.**

Two ways to poll

* **short (immediate)** : uses **1 request and can return 0 or more messages**. If the queue is empty, it will return 0 and try again. This hurts queues that stay short
* **long (waitTimeSeconds) :** it **will wait for up to 20 seconds for messages to arrive on the queue**. It will sit and wait if none currently exist. [Recommended]

**Messages can live on SQS Queue for up to 15 days.** They **offer KMS encryption at rest.** Server side encryption. Data is encrypted in transit with SQS and any clients.

Access to a queue is based on identity policies or a queue policy. **Queue policies only can allow access from an outside account. This is a resource policy.**

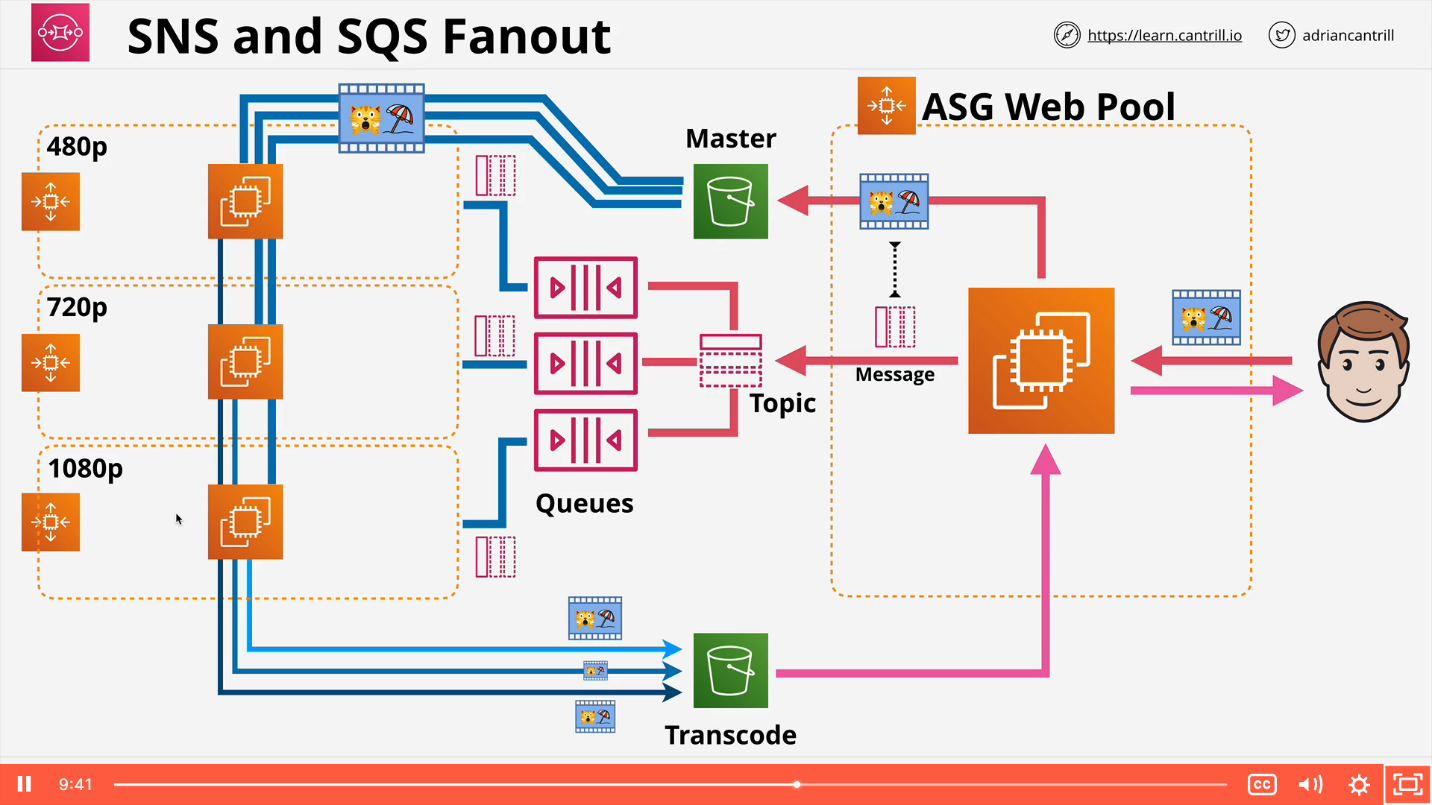
**SNS & SQS Fanout Architecture:**

Suppose in a scenario where a video is uploaded to S3 and we want 3 different versions of the video we may use the SNS & SQS Fanout architecture. So for workloads like this where one job is logged and multiple different outputs are needed this architecture can be used. So once a master video is uploaded to Master S3 bucket a message is sent but instead of going to the SQS Queue the message is added onto a SNS topic. Now this topic has multiple subscribers, for each different resolution of video required there is one SQS queue configured as a subscriber to that topic.

So each size has its own queue and its own Auto Scaling group which scales based on the length of that individual queue. And this means that if different workload types needs different sizes or capabilities of instances, they can independently scale.

In the exam there may be question about an object being uploaded to an S3 bucket, and a requirement for spawning multiple jobs. S3 buckets are capable of generating an event when an object is uploaded to that bucket but it can generate only one event. So in order to take that one event create multiple events that can be used independently, you can use this fan out architecture.

So we take one SNS topic with multiple subscribers, generally multiple SQS Queues, and then message will be added into each of those queues allowing for multiple jobs to be started per object upload.

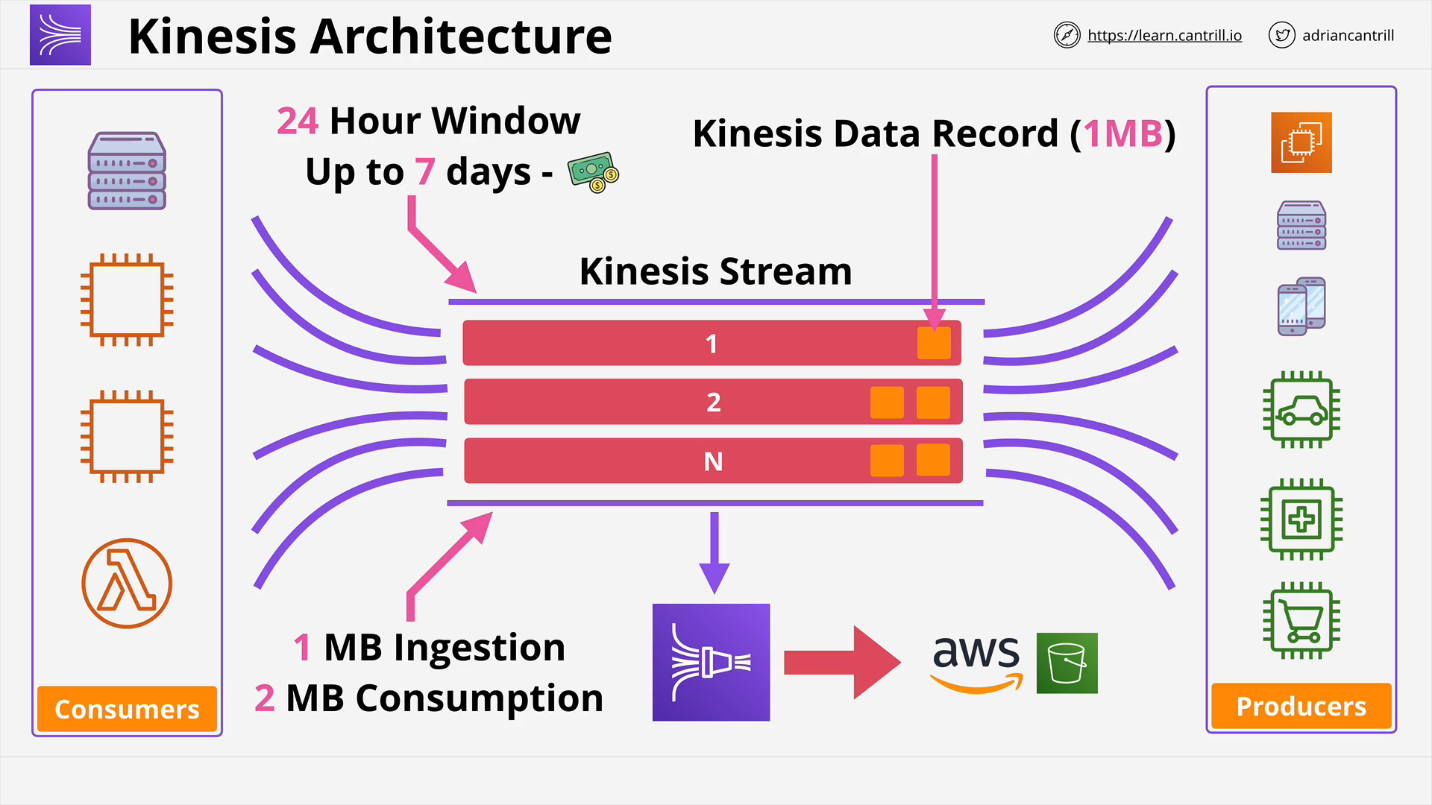


**Kinesis**

* Scalable streaming service. It is designed to inject data from lots of devices or lots of applications.
* Many producers send data into a **Kinesis Stream.**
* The stream can scale from low to near infinite data rates.
* Highly available **public service by design.**
* Streams store a **24-hour moving window of data.**
  + **Can be increased to 7 days.**
  + **Data 24 hours + 1s is replaced by new data entering the stream.**
* Kinesis **includes the storage costs within it for the amount of data that can be ingested during a 24 hour period.** However much you ingest during 24 hours, that's included.
* Multiple consumers can access data from that moving window.
  + **One might look at data points once per hour**
  + **Another looks at data 1 per minute.**
* Kinesis stream starts with **1 shard and expands as needed.**
  + Each shard can have **1MB/s for ingestion** and **2MB/s consumption.**

**Kinesis data records (1MB)** are stored across shards and are the blocks of data for a stream.

**Kinesis Data Firehose** connects to a Kinesis stream. It can move the data from a stream onto S3 or another service.



**SQS vs Kinesis**

**Kinesis**

* Large throughput or large numbers of devices
* Huge scale ingestion with multiple consumers
* Rolling window for multiple consumers
* Designed for data ingestion, analytics, monitoring, app clicks
* Each consumer reading data at different rates
* Mobile click stream

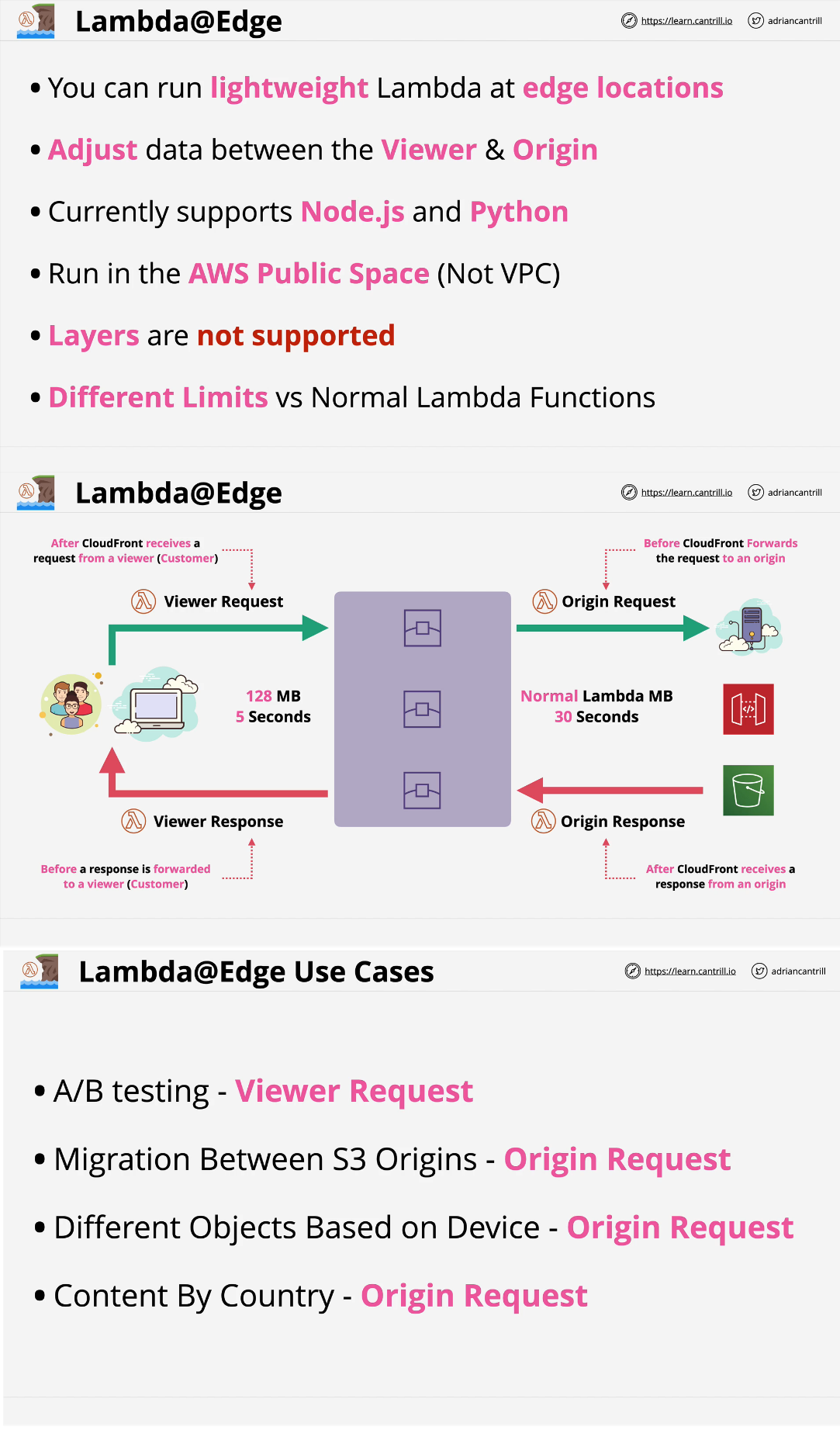
**SQS**

* 1 thing or 1 group of consumers sending messages to the queue
* One consumption group from that tier
* Allow for async communications
* Once the message is processed, it is deleted
* Decoupling application states
* No persistence.

**During Exam:**

Ingestion of Data, Large throughput, large number of devices, Streams, Consuming data at different rates ->**Kinesis**

Web tier Worker pool, decoupling, asynchronous communication -> **SQS**

**Lambda Edge**