# **Serverless and Microservices for AWS**

### **Why serverless? Why microservices?**

Selecting transcript lines in this section will navigate to timestamp in the video

- As builders, we want to spend our time focused on the things we do best, building and solving customer problems. Serverless helps us on that journey by letting us focus on the things we do best, solving our customer's problems without having to fret over how to keep managing and scaling our infrastructure. In this course, I'll take you through how serverless and microservices can help by using some real-world examples from a Transit Insight dashboard. Want to know how to build and see it in action? This is a great place to start for more about how Lambdas and Step Functions really make a difference. I'm Aileen Gemma Smith, CEO of Vizalytics Technology and an AWS Community Hero. Our team uses serverless in our solutions for clients in transit. I'm excited to share how serverless has helped to make the work we do with graph technology even more powerful. Join me on my LinkedIn Learning Course on serverless with AWS.

### **What you should know**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] This course assumes you have a working knowledge of modern programming languages such as Python, Java, Node.js, or C#. The examples in this course will be in Python and JavaScript. I'll be covering a wide range of topics and services focused on the AWS cloud ecosystem. You should have a basic familiarity with cloud concepts, but specific AWS experience is not required. The course examples are focused on collecting and processing realtime data. It will be helpful if you have at least some experience building real-world, data-oriented applications.

### **Serverless and microservices in action**

Selecting transcript lines in this section will navigate to timestamp in the video

- We'll look at the Serverless landscape available on AWS. Serverless applications use managed services. There's no need to provision servers, manage server hardware, or administer servers. AWS has Serverless components that support a full range of core capabilities needed to develop your sophisticated Serverless applications. Let's dive into Serverless. Serverless is a fully managed approach to building, deploying, and managing your technology infrastructure. Serverless gives you compute without the need for owning, maintaining, or even provisioning servers. Serverless gives you sophisticated networking and enterprise-level integration tools that automatically scale to your workloads. Serverless gives you durable storage without the need to manage your own backups. Serverless gives you managed databases that can grow to meet your needs without having to plan for server and storage updates. Serverless can help you focus on your business by removing the need to worry about building a data center team and will scale as your business scales. Because traditional data center operations are fully managed behind the scenes, your company's operational complexity is reduced. Serverless can also help you reduce cost. A major cost-saving benefit of Serverless is paying only for the resources that are actually used. Serverless also reduces cost and complexity through removing processes and tasks that previously required in-house talent that may not have been core to your business. Backups, version management and patching are automatic, and configurable, removing the need for system administration specialists on your team. Serverless on AWS can help you to streamline your overall operation. Common tasks such as security, error handling, alerting and logging can be implemented and managed using standard components and technologies, such as IAM and CloudWatch. With Serverless technologies, you pay only for what you use. This includes compute, storage, network transfers, API calls. For example, AWS Lambda provides on demand compute. You only have to pay for the executions that actually occur, and they're billed to the millisecond. In many cases, this is orders of magnitude cheaper than the cost of having a server running 24/7. Other AWS technologies, such as Amazon Simple Storage Service, or S3, can be used to serve websites with high-performance and minimal cost compared to even the smallest server running 24/7. Autoscaling, where resources expand and contract based on real time usage, is far more cost-effective than paying for standby servers to manage occasional usage spikes. Let's review the Serverless landscape on AWS.

### **Serverless landscape on AWS**

Selecting transcript lines in this section will navigate to timestamp in the video

- We'll look at the serverless landscape available on AWS. Serverless applications use manage services, there's no need to provision servers, manage server hardware, or administer servers. AWS has serverless components that support a full range of core capabilities needed to develop your sophisticated serverless applications. These components include compute, storage, data stores, API proxies, application integration and orchestration, analytics, and even developer tools. Let's dig into the possibilities. Storage is a logical first step in serverless application building. Amazon's simple storage service, S3, is the most well known. It's a secure, durable, and seemingly infinitely scalable object store. It supports multiple tiers of durability, and you can trade off durability and retrieval time against storage costs. This tiering can be fully automated with sophisticated, yet easy to configure rules. There are currently seven categories of fully managed databases. Relational, key-value, in-memory, document, graph, time series, and ledger. We'll do a deep dive of each of these in a future video. For now, we'll give an overview. Relational databases include Amazon RDS, fully managed relational databases including MySQL, PostgreSQL, MariaDB, Oracle, and SQL Server. These are managed versions of the releases that you might've previously installed and managed on your in house hardware. Amazon Aurora takes it one step further. It provides a relational database fully compatible with MySQL or PostgreSQL, that is automatically scalable, distributed by default, and faster than their open source peers. Amazon Aurora Serverless takes this one step further by making the database fully on demand, starting up, shutting down, and scaling based on application needs. This is ideal for infrequently used applications. Finally, Amazon Redshift can provide a fully serverless data warehouse solution. Amazon DynamoDB is a key value store with single digit, millisecond response time, ideal for high-traffic web applications, E-commerce, and gaming. Its speed and ease of use make it useful as a simple cache for certain applications. In-memory databases are typically used for session management and caching. AWS ElastiCache is a fully managed clustered solution compatible with either Redis or Memcached, the two most widely used opensource in-memory databases. Amazon Document DB is a recently announced document database, compatible at the API level with MongoDB. Amazon Neptune is a graph database that can be used with either a property graph model supporting Gremlin, and ideal for classic applications like fraud detection and recommendation engines, or as a semantic triple store that supports SPARQL 1.1 and can be used to build knowledge graphs. Another recent announcement is Amazon Time Stream, a time series database, ideal for IOT applications and telemetry. It can handle trillions, with a T, trillions of events per day. Finally, Amazon Quantum Ledger DB, provides an immutable and verifiable transaction log without the need to manage a block chain network. Building an application requires much more than object storage and database technology. The logic of your application requires compute resources. Historically, this required you to manage servers. Serverless compute removes the servers from the equation and lets you concentrate on the logic. AWS Lambda is the basic building block for serverless compute. The code runs on demand and can be triggered by events, such as a user action, or periodically as needed. AWS Lambda logic can be programmed in virtually any language. It natively supports Python, Java, NodeJS, Go, Ruby, PowerShell, and C#. There's a Runtime API which allows you to use other languages. If AWS Lambda limitations, such as the 15 minute execution time limit, or memory constraints are an issue with your architecture, AWS Fargate offers a fully managed infrastructure for container based applications, again, without any need to provision servers. API management is supported by Amazon API Gateway, allowing front end systems to easily call serverless compute assets. Serverless compute implies a different, more functional approach to application architectures. Most commonly, functionality is developed as microservices. Communication among these microservices requires various approaches to application integration and orchestration. Where decoupling is needed between microservices, often to support fall tolerance and scaling, message passing and queuing are needed. Amazon SQS is a fully managed and scalable queuing system that includes general and FIFO queues to pass information from one service to another. Where a published subscribe paradigm is more appropriate, Amazon's Simple Notification service, ANS, is available. Where coordination among Lambda based services and functions is called for, AWS provides step functions to orchestrate sophisticated interactions by defining state machine style functions. Analytics for streaming data is provided by Amazon Kinesis. We'll touch on this later in the course. And finally, AWS has a number of tools available to help developers in the development process. There are IDEs, continuous integration and deployment tools, SDKs, and monitoring and logging tools. For this course however, we'll mainly look at the AWS serverless application model and the AWS serverless application repository to look at how that can facilitate your serverless journey.

### **How to choose a serverless technology**

Selecting transcript lines in this section will navigate to timestamp in the video

- Last time, we talked about the serverless landscape available with AWS. This time, we'll look at how to decide which components are right for the problem you're trying to solve. We'll focus on selecting a compute option, data storage, and touch briefly on integration and orchestration. I'll show a lot more implementation details in future videos. For now, I want to focus on the why of choosing technologies rather than the how of building. A few examples scenarios will be helpful to think through this section of technologies for serverless application. Here we're just looking at the technology section. We'll dive deep into real-world examples in future videos. The first common application we'll look at is a single-page web delivered application. Let's consider a simple transit app, Where's My Train. In its simplest form, we'll need some HTML and some JavaScript so we can call out to the transit agency's APIs. In the AWS serverless ecosystem, this can be built and deployed entirely on S3, the Simple Storage Service. No other components are needed. Compute, load balancing, and scale are taken care of automatically. Of course, this is a simple application scenario. In the real world, transit agencies impose API call limits. Users and app developers want accounts where preferences can be stored, and improvements to arrival time predictions are needed to deal with transit disruption. The list goes on. So let's make this a bit more useful and a little bit more complicated. The transit agency publishes alerts about service disruptions. These alerts are updated once a minute, and they contain all the service delays and disruptions currently active across the whole city. Here's an example of the JSON the service returns. Things to note are that the data returned is in an array of all the current delays. For each disruption, the service returns the agency, the route, the amount of delay in seconds, and the stations or stops between which the delay is occurring. The reason for the delay is also provided. Calling the service disruption API directly from JavaScript on the web browser is going to give the user lots of data that may not be relevant to the user at all. For example, they may not care about the buses or the ferries if they're waiting for a train. In order to get to the next level of personalization and functionality, we need some backend services. So the first thing we need to do is call the service disruption API once every minute. This requires some compute. The options are running an EC2 server using containers via Fargate or using AWS Lambda. So how do we decide? We need this compute power once every minute. In this simple scenario, our application backend will make the API call to get the service disruption content, parse through the data, and store it indexed by train, service, and station. In the AWS ecosystem, these kinds of things are almost always appropriate for AWS Lambda. The decision is going to depend on how much time is needed for the processing. I found that the best approach is to use AWS Lambda until you're proven otherwise. In this case, the API call takes a few hundred milliseconds, and the parsing and storage, another few hundred milliseconds. This means you only need that compute power for one second every 60 seconds. This is ideal for Lambda. When would you use Fargate instead? When Lambda limits are exceeded. Those limits are time. Lambdas top out at 15 minutes of execution time. RAM, Lambdas can only provide up to three gigs of RAM or compute power, which is managed behind the scenes and scales based on your allocated RAM. Fargate will give you control over memory and compute. Now we've decided that we'll use AWS Lambda to capture and parse the service disruption content. How should we store that data? The decision involves how the data will be used and how it will be served up to the user. And of course, the number of users that will be accessing the data. In this simple example, let's assume that the user needs to understand delays that will affect them right now as they wait for a train to take them from their origin station to their destination. As we described in the previous video, there are seven categories of fully managed databases available. Relational, key-value, in-memory, document, graph, time series, and ledger. Which one is right for this scenario? Well, it's not time series or ledger, so those aren't really relevant here. And any of the others can probably work, but there are cost and efficiency factors that will drive that decision. If our application is really straightforward, we'll tell the user of disruptions involving either the origin or the destination station. Then a key-value approach, like DynamoDB can be easy to use and cost efficient. If the logic is more sophisticated, a relational or graph database might be more appropriate. But this comes at the cost of having a managed database always available. Defining a DynamoDB table is much lighter weight. Now that we've decided on DynamoDB for our key-value store, let's briefly think about managing the data flow from the transit agency's API to our DynamoDB table. We need to call the transit agency API once per minute. Scheduling a Lambda to run at that rate is easy. Trigger with a CloudWatch Event where the rate is one minute. There, is still a decision you might want to make. Once you call the API, should you parse the data directly in that Lambda and put it into DynamoDB? I'll dive into this kind of decision-making more deeply in the building serverless components chapter. But for now, let's just look at a simple alternative. It's often a good idea to split up the API call from the processing of the data received. It gives you flexibility for future changes, testing, data analysis, and recovering from errors when new requirements arise. What I found to be very useful is to build that Lambda to store the raw returned JSON data to an S3 bucket. Within the bucket, we have a key prefix providing the standard folder-like hierarchy for storing the data by data type, agency, date-time partition, and finally, a file name. This allows you to manually find things quickly for debugging and in the future supports using the S3 structure for in-place analytics over your data. Then we'll build a second Lambda function triggered off the S3 Put Object Event that actually parses the data and enters it into the data store. This approach decouples the data collection from the parsing logic which gives you flexibility and increased robustness. For example, if the database storage logic fails, the data is safe and available for reprocessing. Based on application needs, you can set a time to live on the raw JSON data and let S3 delete the data after a day, or a month or a year. Or you can make use of S3's tiered storage levels to trade access time for vastly less expensive storage costs.

### **What is a microservice?**

Selecting transcript lines in this section will navigate to timestamp in the video

- In this chapter we'll talk about microservices. And in particular, serverless microservices and how they differ from other available architectures. The typical architectural approaches to application building follow a continuum from monolithic architecture to service-oriented architecture, SOA, to microservices, and finally, serverless microservices. Monolithic architectures are the traditional approach. They are built as a single code base and may run on a single hardware platform. Changes to monolithic applications may require building and deploying the entire application, even for small or specific changes or fixes. There are some positives about monoliths, though. Development can be simpler. Cross cutting concerns like security, logging and data transfer may be easier to handle. Actual performance may be higher as well. The trade-offs can be significant, though. Over time, the code base can become complex. Refactoring to address this may be a huge undertaking, as it likely touches all parts of the code base. It's also difficult to move to new technologies. Perhaps the biggest downside is the lack of agility, particularly when multiple teams are working on the same app. Service-oriented architectures move from monolithic approaches by creating distributed application components that communicate via messaging middleware technology. Services in SOA can be entire subsystems. Microservices are characterized by services that are generally single-purpose. Communications among microservices typically use REST, or messaging queuing approaches. Serverless microservices architectures are microservices deployed in a cloud environment using serverless technologies for compute. Communications among the microservices are the same as server-based microservice deployments, REST, events or messages.

### **When should you use microservices?**

Selecting transcript lines in this section will navigate to timestamp in the video

- Earlier, I gave an overview of application architectures: monolithic, service-oriented, and microservices. In this video, I'll discuss the scenarios that are ideal for a microservices approach. I'll discuss the scenarios that are ideal for a microservices approach. Microservices are meant to do one thing really well. They should be stateless and scalable. And composable, allowing you to build complex functionality from reusable building blocks, using technology agnostic communications paths. When thinking of the ideal use cases for microservices architecture, for all or parts of an application, it's helpful to think of the ways that microservices can be triggered or accessed. There are two primary options for calling microservices: APIs and events. APIs are straightforward. A call is made to a microservice, usually over HTTP, with some appropriate parameters, and the microservice performs the required task and returns the appropriate values to the caller. This approach is ideal when a request/response paradigm makes sense. APIs can be used in a program to program sense in the backend systems, or could be publicly facing to be called by various client programs, such as a web browser, in the classic JavaScript AJAX pattern. Events bring a much more generalized, asynchronous nature to microservices architectures. Events can be thought of as messages. In some cases, those messages may be as simple as a time tick. Run this service once every minute. In others, the messages may have content that is important for the microservice to perform its calculations. Let's talk about use cases that are ideal for a microservices architecture. Are there functions that are suited to a request/response approach? Let's think of some examples. Microservices that need to be called from external clients, backend services in n-tier architectures, requests where the results are needed synchronously. Event-based microservices are often the most versatile. They are ideal for building data pipelines, but can be used for transactions, and in many real-time scenarios as well. In a data pipeline, services may start with a time tick. Call an external API once every 15 seconds to receive some data. The initial microservice serializes that data into a queue. Other microservices listen on that queue and process the received messages. When that processing is complete, a new message can be sent to trigger the next stage of processing. At each stage of processing, there is an opportunity for scale. Here is an example of a real-world data pipeline for transit-related data. For now, I'll give you a brief overview. The pipeline is gathering data from multiple sources in various formats, transforming the data, and ultimately adding it to a knowledge graph. From left to right, you can see several groups of functionality. First is a scheduler. Data from different sources is collected at different rates, from several seconds to several hours. The first compute components are called data collectors. They are tasked by the scheduler via messages, gather the requested data, store it in a serialized raw format in S3. The data is acted upon by the second group of compute components, data converters. These convert the raw serialized data into a common serialization with a defined semantics, and again, serialize that data to S3. Finally, the third group of components process the common format data and then insert it into the knowledge graph. Microservices are good, but not for every single use case. Keep in mind too, when microservices might not be the right approach. For an MVP. Microservices architectures can be complex. It's often best to build your MVP as a monolith. If your application is quite simple. Microservices could add complexity where it's not really needed, or your team is not experienced enough. Microservices architectures can be complicated. Or if the application isn't complex enough that you're running into issues with a monolithic architecture.

### **Serverless architectures on AWS**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Let's look at a complex, real-world application, and how to approach it with a server-less, microservices solution. Throughout this chapter, I'll discuss a transportation inside dashboard that might be used behind the scenes of a public transit agency in a metropolitan area. In this case, I'll talk about a rail system. But it could also be buses, or ferries, or trams. In the real world, timetables can be published at random times throughout the day and subsequent publications take precedence over previous versions. The system will keep track of where all the trains are located and whether or not they're on time. It will show punctuality metrics in real time and will characterize delays when they occur, so that train controllers can understand if a delay pattern will likely impact other trips on the network. Clearly, there are a lot of moving pieces. Not only on the rails, but also in the system I'll illustrate. Before we begin, we have to look at what data is available to approach the problem. On any given day, there are timetables for each of the various train lines. The transit infrastructure provides the identification and the location of each train once it beings its trip. There is semi-static, geospatial data for the tracks. Calculating punctuality metrics for a particular train requires the relevant timetable and the actual times that the train has arrived at and departed from each station on the route. What are the top level components of the system? There are two primary data pipelines. One for timetables and a second for vehicle position data. First, let's understand the steps that we need to follow to process this data. Call the timetable service if there are new timetables, then download them. Process each timetable package and insert it into our database. The next data pipeline will process vehicle position reports. This data is updated in near real-time for each one of the trains in operation. So, there may be hundreds of trains in operation on the network at any given time. There are few complications that arise that we'll discuss a little later. The steps we'll follow to process the data are: call the vehicle position report service run by the transit agency and serialize the data to JSON for further processing, process the JSON data and insert it into the database. We'll need to determine when each train reaches a station stop. Calculate the metrics. Can we characterize a delay? Cache the results where we can. How we build the delay characterization model in this model, remember, the type of delay matters, so we'll want to know what kind it is.

### **Hybrid architectures**

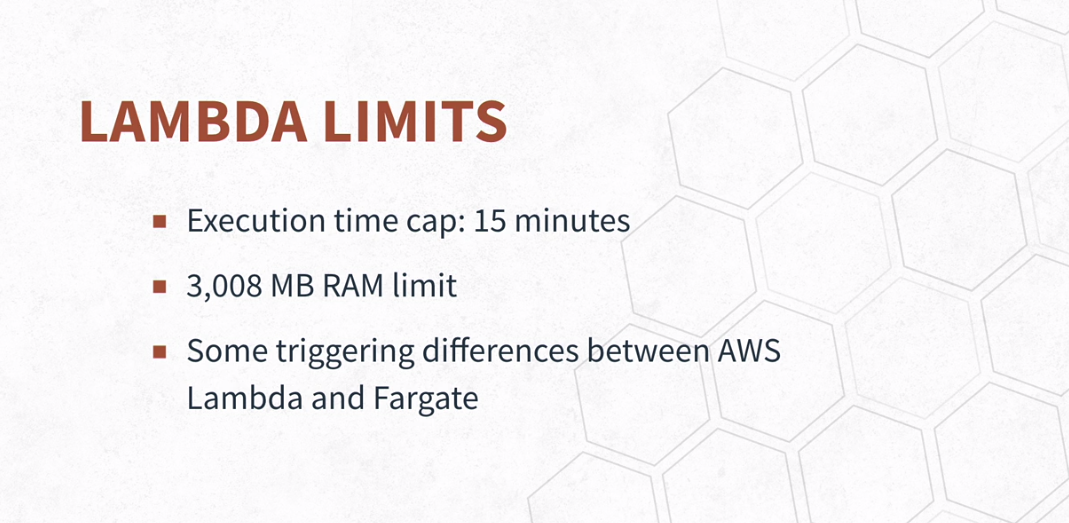
Selecting transcript lines in this section will navigate to timestamp in the video

- Some systems may not be suitable for a completely serverless approach. This video will help you understand the AWS services that can be leveraged when a hybrid architecture is required. Real-world, complex systems cannot always be put in a completely serverless architecture. In the AWS cloud environment you may encounter situations where the limits around lambda functions make it necessary to build certain components differently. AWS lambda imposes a few limits. The first is execution time. Until recently, execution time was capped at five minutes. Today, it is 15 minutes. There is 3,008 megs limit on RAM, processing cores are managed behind the scenes, and scale up based on the RAM selection, but are otherwise, something of a black box. There are also some limitations around triggering of Lambdas via events or messages. The fall back from Lambda for serverless compute is to move to a containerized architecture for that service. Amazon Fargate is a serverless manager for containers and can keep you from having to spin up and manage your own EC2 instances. There's a lot more flexibility with containers, but they're much more heavy weight. A good rule of thumb, start with AWS Lambda until proven otherwise, then only if necessary, move to Fargate. To illustrate this issue, we'll look at our transit application, specifically, at timetable processing. You'll remember from the previous video that the timetables are available via an API call to the transit agency service. It turns out there are actually 15 different timetables published by various subagencies. Due to the nature of the timetables, they have to be processed in order for each agency. When they're published once every day or so, it isn't a significant architectural problem. But, when they're published in quick succession, say three updates in 10 minutes, then problems can arise, particularly since some timetables may take 20 minutes to process. This problem poses a few challenges for serverless. First, we can run up against the 15 minute Lambda execution limit. Second, we need to process the timetables for each agency in order. Processing publication sequence is an interesting problem in an event driven scenario. Let's put aside the 15 minute execution limit for a moment. If we trigger the processing of the timetable based on the S3 put object event, a typical approach, we have no control over the processing order. While the event is being processed, an update could be written and processed by a second invocation of that processing function, which may complete ahead of the first. The same goes for placing the event in a standard SQS queue. We'll do a deep dive on SQS and other messaging options later. Multiple Lambdas can be invoked to read from the queue, so again, we can end up processing these updates in parallel, rather than in sequence. What we need is a way to enforce FIFO, first in, first out in the queue. But for efficiency, we want it to be FIFO for any given agency. We want to have different agencies processed in parallel. FIFO queues guarantee delivery order. They also support message groups. What we can do then is put our event in a FIFO queue and set up message groups for each publishing agency. That way, we can have a single processor for each message group agency listening to that FIFO queue. This provides the functionality that we're after. For obvious, or maybe not so obvious reasons, you can't trigger a Lambda function from a FIFO queue. This would make the delivery order contract impossible to enforce. In our case, using message groups to segregate agency processing would also be a problem for a Lambda based implementation. So, for this situation, we need a long running process that can listen to the FIFO queue and spawn tasks to process each agency's timetable stream. Here, we send the message and note the message group ID is set to agency ID. Here, we have our long running receive loop that waits on data from the FIFO queue. Note, that because message group ID is set to agency ID, by the sender, previous code we just looked at, no message with that agency ID will be delivered to any receiving process until the previous message processing is complete. Also note, that the process call runs in a new thread and the queue cleanup is handled there. And this is just one example of a situation where serverless may not be the best possible choice for a particular service.

### **Serverless compute**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In the AWS serverless ecosystem, Lambdas are the primary computational unit of a serverless application. The fallback from Lambda for serverless compute is to move to a containerized architecture for that service. Amazon Fargate is a serverless manager for containers, and can keep you from having to spin up and manage your own EC2 instances. In addition to the ability to build long-running processes, there is also more flexibility with containers versus Lambdas. The downside is that they're more heavyweight. A good rule of thumb when architecting a serverless application is to start building using AWS Lambda until you run into limitations. Then, only if necessary, move to Fargate. Keep in mind, if you're already using containers in Docker, Fargate will be familiar. AWS Lambda imposes a few limits. First is execution time. It's capped at 15 minutes. There's a 3,000 meg limit on RAM. Processing cores are managed behind the scenes and scale up based on RAM selection, but otherwise are something of a black box. There are also some limitations around triggering of Lambdas via events or messages. AWS Lambda supports a number of programming languages natively: Java, Node.js, Python, C#, PowerShell, Go, and Ruby. The examples I'll show are in Python. Let's dive in to some examples that illustrate how to build up components of our transit application. We'll focus on the thinking behind the architecture as well as the code. We talked about data pipelines that we need to build. Now, let's look at vehicle position report processing. If you haven't seen one, this is a sequence diagram. And they're useful to illustrate interactions among components in a software system. Communications among components follow the arrows left and right. Time is on a vertical axis and increases from top to bottom. I'll start with a Lambda function, labeled raw capture. Position reports are obtained via the transit agencies, GTFS, Real Time Vehicle Position Service, and API call that returns data using protocol buffers, a language-neutral and platform-neutral mechanism for serializing structured data. It's a good practice to store data received from services such as GTFSR in as close to raw format as possible. S3 is an ideal repository to use for this purpose. Keeping the raw data around allows you to rerun subsequent processing stages and will save you many headaches down the line. It also means that the Lambda, at the head of your data pipeline, can be very simple. The function has three basic steps. First, make the API call and retrieve the protobuf data response, two, we're going to convert the protobuf data to JSON, and three, we're going to upload that JSON file to S3. One thing to think about when you create your raw data repository on S3, is data partitioning. You could throw every object directly into the S3 bucket, but you'll find that quickly becomes unwieldy as you build up hundreds, to thousands, to millions of objects. For this type of data, time-based partitioning approach is ideal. In this case, due to the volume of data, we partition each feed's data by year, month, day, and hour. And that, makes it very easy to navigate on the console, or via the CLI. But perhaps, and even more important, it sets things up so you can easily repurpose the data as a data link, and then analyze it using Amazon Athena or Glue. Now that we have the data in S3, the next step is get it into our database. So for that, we'll create a second Lambda function. It will be triggered by the PutObject Event in S3 for each response processed by the previous Lambda. You can see, in the Event response, that you get the bucket, and the object key for the JSON object. Note, the key comes across as URL uncoded, so you have to unquote it. Now, we'll load that data from S3 into the Lambda function. This can be done with a single line of Python code. Finally, we'll write the data into our database. The details of the conversion for insertion into the database are a little bit beyond the scope of this course, however, there are several things to keep in mind with databases and AWS Lambda. First, be careful of Lambda execution explosion. When you have high frequency event-driven triggers, you can end up with hundreds of parallel executions, each connecting and writing to your database. This can quickly cause problems, so set your Lambda concurrency limits to a reasonable number, maybe 10 or 20, as is appropriate to your use case. Aside from that, remember that AWS may keep a Lambda environment warm for some number of minutes. Although there's no mechanism for real connection pooling at this time, there are some techniques that can be used. One approach to taking advantage of a warm database connection in a Python Lambda environment is this: create a database container class, then instantiate this class outside the Lambda handler. When you make the call to obtain a database connection first try to execute a simple select on the instantiated class. If that works, you can reuse the connection. If it throws an exception, you'll need to create a new connection. Things to remember about serverless data pipelines, save raw data as close to the origin as possible. If the stages of the pipeline are complex, save the intermediate transformations as well. It can be a life saver when things, inevitably, go awry.



### **AWS connecting the dots, part I**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Building a serverless application from a number of single-purpose stateless microservices means there's a lot of communications going on. This all needs to be setup and connected in the right way to keep things going smoothly. All this communication seems complex and it can be but it also brings some major benefits and one of the main ones is decoupling. To get this benefit, you'll often want to design your components to operate as a pipeline. There are few things to think about. Communication between components can be more or less decoupled. The degree of the coupling can have significant implications on the overall architecture of your application, its performance, and its scalability. There are several common patterns. Direct calls, either synchronous or asynchronous create fairly tight coupling. Producer-consumer approaches can be much more decoupled with behaviors that differ based on how the producer and consumer communicate. Pipeline architecture supports situations where the producer doesn't need results back from the consumer. Instead, they pass the results along to the next stage for processing. Producer-consumer and pipeline architectures can use the same types of messaging, cues, messages, and streams. Note, when we have a situation where we need tighter coupling between services, there are some good solutions available. Serverless microservices should be designed to do one thing well, venues, events, and messages to access other services. When you architect this, you'll need to think about what needs to run sequentially and what can run in parallel, then you can choose an integration approach. The first thing to think about is what is the intermediate step between processing stages. Do we want to or need to save a new object or are we augmenting that data in place? Where data can be serialized to S3, it's often useful to save some intermediate versions as processing occurs 'cause of dual benefit, you can restart processing at a checkpoint based on that intermediate form of the data and you can use S3 events to trigger the next stage of processing. If the data is heading straight into a database, we may be augmenting data in place. In a monolithic app, you might leverage processor cores using a producer-consumer model with threads. In a simple case, consider that you have to run a number of different machine learning models against each trip when you get a new position report. Each request takes several minutes to return an answer. If you run them sequentially as is shown in the first block of the pseudocode, the elapsed time will be the sum of each call. In a producer-consumer multi-threaded model, they can all run in parallel as shown in the second block of pseudocode. A new process thread is spawned to execute the model run. They all run in parallel and the join waits until all results are complete before allowing execution to proceed. Here your elapsed time will be closer to the amount of time that the slowest model run took. That's a lot more time efficient and a common pattern in monolithic applications but building complex multi-threaded components in a serverless world may cause you to run into limits. For instance, your lambda the execution limits. The code is also a lot more complex. Producer-consumer models in a serverless environment will separate the producer and consumer processes into independent services communicating using cues, messages, or streams. Pipelines run in a more linear fashion. A pipeline stage performs its work, passes the results forward. The components of the stage don't need to be aware of the operation of upstream or downstream processing steps. They're only concerned with the data formats they act upon and the data formats they produce. The receipt of input data and the delivery of output data will impact the performance and scalability of the pipeline architecture. There are three principal AWS services to choose from to message among services, SQS for queuing, SNS for messaging, and Kinesis for streams. Here's what to think about in choosing. SQS, Simple Queue Service is a queue. It is intended to buffer messages from a producer that will later be processed by a consumer. The important things to understand about SQS and queues in general is that there is a single consumer per message. If there's a lambda function, listening for messages on the queue, messages can be taken off the queue by multiple instances of the same lambda function up to your specified concurrency limit. If that intermediate stage needs to be acted on by multiple consumers, each performing a different task, you need to use multiple queues, and that's probably not the right approach. SQS also provides FIFO queues. These cues guarantee delivery of messages within defined message groups. SNS, Simple Notification Service is a publish-subscribe model. This is ideal for sending messages that need to have multiple consumers. There can be up to 10 million subscribers on an SNS topic. This make sense for situations I described earlier, initiating calls against an arbitrary number of machine learning models. HML service service subscribes to the topic and can independently execute when the message is received. One thing to keep in mind with SNS is that you have to design around failure models. If a consumer fails, the message has been consumed and you could lose data. SQS doesn't have this problem. If the consumer doesn't successfully complete the processing of the message, it goes back into the queue and can be picked up and processed again. Finally, there is Kinesis. Kinesis provides some of the best pars of queues and publish-subscribe at the expense of a bit more requirement for the user to manage setup, scale, and operation. Kinesis is a stream-based service. Similar to SQS, the messages are persistent in the stream so data loss doesn't occur if there are problems on the consumer side. Similar to SNS, Kinesis can have multiple consumers for each message. Message size is also much larger than the 250k limits in SQS and SNS. It can have multiple shards within a stream and consumers can be assigned to specific shards. Finally, Kinesis supports the concept of replay. Time can be set back to replay messages in the stream.

### **AWS connecting the dots, part 2**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Connecting multiple serverless microservices to build an application requires communication and coordination among the components. Using the correct architectural patterns will provide opportunities for a efficient operation and scale. In part one, we talked about the services available to connect our serverless components. In part two, we'll give concrete examples of using SQS and SNS from our public transit insight dashboard application. We'll start wit SQS, Simple Queue Service. We use queues in many places where messages will be consumed by a single consumer. Here, we're processing the vehicle position reports and we will send a list of associated trip IDs using a queue. We're using a standard queue in this case. Two queue attributes to be aware of: Message retention period. This value determines how long an unprocessed message will remain in the queue available for receipt. And dead letter queue. Setting up a dead letter queue enables you to capture any messages that fall out due to errors and expiring message retention. You can then set up error handling to deal with these. Our Lambda Functions are written in Python, so we use Boto 3 to interact with AWS. It's automatically available to Python Lambdas. Setting up to write to a queue is straight forward. First, we get the queue URL from the queue name. The message body is a JSON string, so we encode that from the raw event message. Finally, we send the message. Configuring a consumer Lambda is easy as well, just select the appropriate SQS queue from the dropdown. Batch size allows a single Lambda to receive more than one message at a time. In this case, the event passed to a Lambda handler will contain an array of up to the configured batch size. If there are fewer messages available in the queue, setting the batch size will not cause the system to delay the triggering event until many messages are available. Using a FIFO queue is similar to a standard queue once you've created it. Note that while you can write to a FIFO queue from a Lambda Function, you can't trigger your Lambdas from the FIFO queue messages. Workarounds for guaranteeing ordering of messages using Kinesis are available but beyond the scope of this course. The writing process to a FIFO queue is essentially the same as writing to a standard queue. First, we get the queue URL from the queue name. The message body is a JSON string so we encode from the raw event message. Finally, we send the message. The one difference is the message group ID, which allows guaranteed ordering of subsets of messages within the same queue. Reading from a FIFO queue requires a listener loop. Your code will sit in a loop, listening for messages using the SQS receive-message call. In that call, you specify the max number of messages and the wait time to listen. If the wait time expires without a message or any error occurs, the while true loop will loop again and the wait will start again. When a message is received, response is not none and the message is found in the response object. SNS is a good choice when you need a publish-subscribe model where there can be multiple consumers of a single message. We use SNS to send out messages when a trip is complete. Multiple consumers use this information to update metrics for display on the dashboard to analyze delay characteristics, to send out alerts and to create different training data sets for ML models. Creating an SNS topic is easy on the SNS console. The only required setting is to give the topic a name. As is publishing a message to the SNS topic, again, since we're in Python, we'll use Boto 3 to access AWS resources. In this case, the message is a simple string representing the trip ID. You could also use a JSON string. The queue URL is created when you create the topic and publishing the message is as easy as calling SQS send-message. Configuring your Lambda Function to be triggered from SNS messages can be done on the console. Just select the appropriate SNS topic from the pull down. And the contents of the message come through as part of the triggering event. Because the message is a simple string, the trip ID, we can use it directly.

### **The importance of APIs**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] When microservices need to be directly called rather than triggered by an event or a message, an API is required. Typically, APIs will be called using HTTP. We usually think of APIs as public facing, called from the UX client of an application to get data from the back end. In the AWS serverless ecosystem, this is well supported and easy to use. Completely serverless web applications can be built using S3, with calls back to Lambda functions for dynamic data and interaction. Lambda functions seamlessly integrate with Amazon API Gateway to allow them to be called from your JavaScript based UX. Let's look at an example from our public transit insight dashboard. The dashboard shows a transit map on the left, highlighting track segments where delays are causing problems. On the right, live punctuality metrics are displayed. Below the map, delay insights are displayed. The data that drives this display is gathered, analyzed, and cached on the back end. The front end is served from Amazon S3 as a static website. JavaScript code on the page obtains the dynamic feed via Amazon API Gateway. The usage profile for the dashboard is read-heavy. There are many clients, but they're all looking at the same screens. Each of the clients gets the dynamic data via API call. That means there might be hundreds of calls to the API every minute. Generating this data on demand will place a processing burden on the back end systems. This is an ideal scenario for caching. A quick aside. Depending on the performance requirements and projected load, you can use S3 as a cache very effectively. Just serialize the results as JSON onto S3. Then, the API calls just need to return the cached JSON. Other caching options, such as ElastiCache, are a bit faster. But if your use case and performance requirements support it, using S3 as a cache in this way can save you a lot of money versus maintaining a cache cluster. It's really up to you. With a Lambda function responding through API gateway, query string parameters from the JavaScript web client are passed through to the Lambda handler in the event. We use the query string parameters to build the S3 key to locate the correct cache file. Finally, we retrieve the JSON from S3 and return it in a special response payload that APA Gateway proxies back to the calling client. You can see here on the API Gateway Config screen there's almost no setup required on API Gateway. Select Use Lambda Proxy Integration and everything takes care of itself. API Gateway's Lambda proxy will pass request payload and/or parameters into your Lambda via the event object and the response JSON data directly back through to your JavaScript client where it can be handled in your success function. You can also set up Lambda functions to be callable via API Gateway from another Lambda function. This allows some flexibility versus making a direct Lambda invocation from one service to another. In most cases, you won't want to invoke one Lambda from another. This kind of dependency can lead to unintended side effects and difficult de-bugging. Step functions may be a better approach when there's a need to coordinate multiple Lambdas in a complex way. I'll talk about that next.

### **Serverless machine learning**

Selecting transcript lines in this section will navigate to timestamp in the video

- We've looked at event driven and message based coordination of the serverless components that make up a serverless application. In some cases, more complex coordination of Lambda functions is required. AWS step functions are designed to address this need. In our public transit inside dashboard, we have a feature that characterizes delays. Are they train related or track related, and are they likely to create knock on delays or be repeated by other trains on subsequent trips. Let's focus on the orchestration of the training process. Model training requires a training data set to be available and the training itself takes time, may even be measured in hours. Retraining the model is required periodically. We're generating data constantly and patterns change in transit over time. So, for example, let's assume we retrain the model once a day. There are a few steps that are needed. First, is there new training data available? If so, let's start a new training run. Once the training is done, we'll need to deploy the new model so it can be used. After the model is deployed, there will be some housekeeping that needs to be done. Step functions can be used to orchestrate the execution of these Lambdas. Step functions are essentially state machines used to control Lambda functions. The return values of each Lambda function can drive a state transition in the state machine. When the step function is fired off, in this case, using a cloud watch cron event, every night at midnight, it invokes the first Lambda function. A function returns true or false depending on whether there is new training data available. Based on this, the state machine either exits, if there's no data available, or invokes the start training Lambda, if there's data. The start training Lambda kicks off a long running training task. It returns a training ID for the process that has just been started. After the training run has been initiated, the step function periodically checks the training progress. This can be checked once a minute. When the training is complete, the loop will end, and the step function will invoke a Lambda function that will deploy the new model so it can be used live. Finally, when the deploy function has succeeded, a final Lambda is invoked to do some housekeeping such as cleaning up intermediate files.

### **Orchestration**

Selecting transcript lines in this section will navigate to timestamp in the video

- Amazon SageMaker is machine learning as a service. Building, training and deploying models is a complex topic. It's assumed for this lesson that we've already done a lot of exploration and testing to arrive at a data model that works for our problem. For our public transit insight dashboard, we're now going to analyze delays. We want to understand whether a delayed trip is experiencing a compounding delay, which points to a rolling stock problem and may lead to knock-on or follow-on delays, or a spot delay, which implies that future trips in the area may be impacted, or a transient delay that isn't a network-level concern. For this problem, we need a classifier. Amazon SageMaker provides two built-in algorithms for classification: linear learner and XGBoost. You can also bring your own algorithm and use it with SageMaker. We'll be using XGBoost. Our focus today is on the mechanics of using SageMaker in a serverless app. The data science behind it all is beyond the scope of this course. Other parts of our serverless application will be creating training vectors. These training vectors will be used to retrain the model periodically. In order to use them, we need to create a manifest file. The manifest is a JSON file with a list of each S3 key to use for the training. We'll use a rolling window of the last two months of data. This task occurs in the check data step described previously. The next step is to kick off the training run. This is a fairly rote exercise. The models themselves run in containers and we use static references supplied by AWS on their documentation page. We need to give the training job a name, set a few hyper parameters, provide a URI to the manifest file, a path to S3 for output data, training instance sizing, and a stopping condition with a max runtime in seconds. We'll return the training job name as well as some other data. The training job name will be needed to check the status of the training job. Training is complete when it comes back as completed. In this case, the response will also include an S3 output path that can be used to create the model data URL that will be needed for deployment. Keep in mind, if something goes wrong, status could come back as failed, and the appropriate action should be taken. Otherwise, training is still underway and you should check again at the next status check interval. Deploying the model takes just a few steps. Creating the model from the model data URL, creating the necessary configuration for the endpoints, and then either creating the endpoint or updating it if it was created earlier. Although the steps are simple, deploying the model can take some time. Another status check process will have to periodically check the deployment to determine if it has completed. In this case, the status will be in service when everything has successfully deployed, failed if there was an error, or rolling back if there was a failure, and a rollback to a previous version is possible. Now that we've deployed the model, we can access the endpoint to classify a delayed train. What we're looking at here is rolling alerts, and we can understand what track segments this is happening at. What you see here is for this train line, four potential trips have been impacted with delays of a total of 14 minutes. The next rolling alert lets us know nine potential trips have been impacted with a total of 24 minutes of delays.

### **All the serverless pieces in action**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Now that we've talked about many of the resources and techniques for building serverless applications on AWS, let's bring it all together to show how we deal with real-time position data to create the training data for our delay characterization model. This sequence diagram shows the full process from obtaining raw vehicle position reports to storing new training new data. The first task is to get the vehicle position reports. We need to request these reports at least once every 15 seconds. Here we immediately run up against an AWS limit. CloudWatch scheduling events have a minimum rate of one minute. Spinning up an EC2 server for this would work, but it'd be quite expensive, and of course not serverless. Step functions are a solution that we can use. We'll run the step function once per minute, and it will call a Lambda function that obtains the raw reports four times. The Lambda function calls out to the transit system's vehicle position API and obtains the required data. We serialize that data and store it in S3. The operation is quite fast, and the data is serialized into multiple JSON files based on reporting agency. Storing the raw data in S3 gives you an archive of data that could otherwise be lost. In this case the transit API is realtime only, and it effectively decouples the data gathering from the data processing. The next step is to the process the serialized reports. This Lambda function is triggered by S3 putobject event when the data is written to S3. This step allows for scale. Each putobject triggers an instance of the raw data processing Lambda up to the configured concurrency limit. Since this Lambda is writing to the database, you'll want to be thoughtful of how many concurrent processes you allow to ensure that you don't overwhelm your database on inserts. This Lambda does one more thing. It sends a message with a list of trip IDs that have just been updated with a new vehicle position entry. In this case we'll use SQS so that we can take advantage of its buffering ability and error tolerance. The Lambda function that listens to this queue is responsible for determining which trips have just arrived at a station stop, and if they have is the trip are complete. Station arrivals are inserted into the database. Completed trips then need to be processed further. In this case we'll use an SNS topic to pass the trip ID of completed trips. I chose SNS because I want to support multiple subscribers to the completed trips topic. Different parts of the system will do different things with completed trips. The processing fans out at this point. In this course, we're focused on the processing path that creates training vectors for our delay characterization machine learning model. This Lambda will be responsible for determining the delay profiles, if any, of the trip. In simple terms, it builds a vector of the difference between scheduled and actual arrival/departure at each station, and then the accumulation of those differences over the sequence of the trip. These vectors are stored in tabular or CSV format on S3 and will be used as part of the training set for delay characterization. Now that we've got all the pieces together, let's take another look at that dashboard to see it all in action. Highlighted track segments show problem areas. We also have alerts on our delays.

### **Decomposing the monolith**

Selecting transcript lines in this section will navigate to timestamp in the video

- It's one thing starting a serverless application from scratch. But if you've got an existing monolithic application, then you need to transition. As an example, we'll look at the transition of a monolithic data pipeline process for an open source data-driven insight system. The application pulls in disparate data from a variety of sources. API calls, XML endpoints and email. In all, there's about 20 different data sources. Each updating at different rates and different times. The monolith application was data-driven via a set of configuration files. It was multi-threaded and ran 24/7 on a large EC2 server to receive, process and transform the data into a knowledge graph that drove the customer-facing web application. As you can see, there are a lot of moving parts and the system is quite complex. API calls need to be made at rates from once every minute to once a day. XML endpoints have similar requirements. Email updates are pushed, but in the monolith were managed by pulling as well. Our new serverless architecture will have to address each of these parts. Event driven approaches really leverage the serverless environment on AWS in a great way. So the first thing to think about is how we transform problem into an event driven one. The second thing to think about is making sure that processing data can reuse as much functionality as possible. From a data format standpoint, we've got a few formats to support. CSV, JSON, XML and email to name a few. Finally, we have to deal with data transformation from the common data format. From the perspective of development, we want to maximize code reuse and minimize how many functions we have to maintain. So the first thing we need is a scheduler. We'll store the schedule of calls in a database, so that it can be maintained and updated on a web-based front end. The scheduler will run periodically, and publish tasks on a set of SNS topics that are consumed by a small set of lambdas responsible for setting up the event driven start to the pipeline. There are a few ways that we could set this up, but what we'll have to do is have a topic for each method. API calls, URL endpoint and data format pair. For this example, we're making some simplifying assumptions about how various APIs work in practice. The message we'll send to the appropriate topic will have a URL and category metadata. These topics will each trigger a simple lambda that visits the API or URL, grabs the data and serializes it to an S3 bucket in its source format using common folder hierarchy. From here we can go event driven. We could convert to a common serialization in the first step, but it's always a good idea to save the source format in these kinds of applications, as often the API calls and endpoints are supplying real time data, so there's no ability to go back and get historical if something is lost along the way. Quick aside. I mentioned earlier that some of our data arrives via email push. We can use Amazon SES, simple email service, to receive these emails and save them in standard email text format on S3 where they can be handled by the same put object events as the pulled data. And emails can be converted to JSON as well. Our next stage in the pipeline is to transform data into a common format. For our example we'll convert to JSON. The conversions can be done generically as well. We'll assume again, for simplicity, that there is one conversion lambda per source format. Remember, we're event driven at this point, so these lambdas are triggered whenever a put object event occurs on S3 by the previous pipeline stage. In the final stage of the pipeline, we'll process the JSON and deal with the actual content and semantics, graph builder functions. Depending on the approach to converting the data to a common JSON format, we will be able to have a manageable number of content-parsing lambdas to add the new data to our knowledge graph.

### **The transition period**

Selecting transcript lines in this section will navigate to timestamp in the video

- Earlier, we discussed the re-architecting of a monolithic application into a serverless application. Now, we'll use the same example with a focus on how to transition from monolith to hybrid architecture to full serverless. To review, an application pulls in disparate data from a variety of sources; API calls, XML endpoints, and email. In all, about 20 different data sources, each updating at different rates and different times. Monolith application is data-driven via a set of configuration rules. It is multi-threaded and runs 24/7 on a large EC2 server to receive, process, and transform the data into a knowledge graph that drives the customer-facing web application. The end state architecture has a scheduler function, data gathering, format standarization and finally, lambdas that process data and build the knowledge graph. In this case, we want to make incremental changes to the monolith that allow us to build out the serverless architecture over a transition period, where we would likely start with the scheduler building our serverless app from scratch and the gradual approach we start from the other end. Stage one then is to alter the code in the monolithic app that calls the graph builder components. In its place is a thin layer that serializes the input data to the graph builder and writes it as common format JSON to our S3 hierarchy. That will allow us to build out the serverless graph builder components, triggered off by the S3 PutObject events. After we've ensured that this transition is stable, we can move on to stage two of the transition. Here, we'll move back one more step to the conversion process, from raw input data to the internal common format that was used for the interprocess communications among the processing threads. As in stage one, we'll replace these components with raw format serializers, writing again to our S3 hierarchy. As before, this will allow us to use the S3 PutObject events to trigger our conversion lambda functions. Again, we'll ensure that this phase of the transition is stable, and then we can again move back a step. Stage three will deal with the data-gathering components. Note that depending on the size of your monolithic app, you could do each stage in multiple steps to make sure everything proceeds smoothly. For instance, you might replace just the API calls, deploy, and then begin work on the URL endpoint sources. In this case, we'll alter the monolith by changing the instantiation of data gathering threads to instead send SNS messages to the appropriate topics, as we've described previously. There are a few ways that we could set this up but what we'll do is have a topic for each access method. API call, URL endpoint, and data format pair. For this example, we're making some simplifying assumptions about how various APIs work in practice. The message we'll send to the appropriate topic will have a URL and category metadata. These topics will each trigger a simple lambda that visits the API or URL, grabs the data, and serializes it to our S3 hierarchy. This will be the same files previously created by our stage two raw data serializers. The format standardizers will pick up those events and process data accordingly. At this point, the monolith is serving solely as a task scheduler. We'll replace the scheduler with our lambda-based scheduler. We'll read the schedule of calls from our database, the same one that drove the monolith. The scheduler will run periodically and publish tasks on the set of SNS topics that we defined previously. These are consumed by the data gathering functions that we deployed in the previous stage. At this point, we've completely transitioned to the serverless architecture, and we can turn off our EC2 instance.

### **Deployment strategies**

Selecting transcript lines in this section will navigate to timestamp in the video

- We'll wrap up this course with a chapter on operations. How can you best develop, deploy, and maintain serverless applications? We've been talking extensively about Lambda functions as the core compute unit for serverless applications. During initial development, the ease of use is awesome. Develop, deploy, run. But what happens when your introduction? If you make changes and deploy your Lambda function, it immediately replaces your previous version. There's no opportunity for testing or for rollback in the event of a problem. The solution to this is versioning. AWS Lambda supports versioning and aliases. When you deploy your Lambda function, its version is set to the special version latest. If you don't do anything else, this will be the running version of your function. Publishing a version makes an immutable copy of the latest version and number sit sequentially. Now, you need to think about aliases. A good first pass would be to create dev, QA, and prod aliases. Once you've tested a version, you can assign the prod alias to that version. Subsequent deployments and published versions will not impact the designated prod version. Note, its best practice to publish a new version every time you deploy an updated Lambda function. Eventually, you can automate those deployments and can publish a new version as part of this process. It's important to remember that you need to set the Lambda trigger on the alias. Now, when you publish a new version, test, and are comfortable that you can replace the prod with a new version, you can point the alias to the newly tested version. Well what if you're using API Gateway with your Lambda functions? How does that work with versions and aliases? First, let's look at versioning with API Gateway. API Gateway supports stages. Stages work very similarly to Lambda aliases. The stage you defined becomes part of the URL for the API. When you set up your API Gateway resource to point at a Lambda function, you're pointing it at the Lambda function name that doesn't take into account, any versions or aliases on the Lambda side. If you configure your API Gateway resource like this, you'll call the latest version, which is most likely not what you want. You can add the Lambda alias or version to the end of the definition. Best practice is to define stage variables that match your Lambda aliases, and then use those stage variables as suffixes on your Lambda calls. Finally, this can be made dynamic by using stage variables. Once these are defined, you can define your Lambda call, referencing the variable. Assuming you set up a stage variable called ENV for your environment, that is dev, QA, and prod, you would define your Lambda call like this. Setting up aliases and stages to work together will help keep your deployments going smoothly.

### **Development challenges and solutions**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Like any significant design and development project, serverless applications require thought and planning before you start building. A non-trivial application will be composed of a number or related components. Some of these components will need to be more closely coupled. Some of them more loosely coupled. Some code will be unique to a particular function, while some will be useful across multiple functions. Different development teams may be focused on different parts of the system. All of these factors drive the structure and the process of development to varying degrees. Laying out the application structure up front is the first step. What are the closely-related parts? How do they connect? In the transit insight dashboard, we have four major components. A real-time data pipeline for vehicle positions, a timetable processor, an analytic sub-system, and a Web application. These major components are related, but from a data flow standpoint, can be thought of as essentially decoupled. Each of those top-level components has a number of constituent parts that make up what is referred to as a serverless application. The AWS serverless application model framework bundles up each of these top-level components. You'll have to separate development and code trees associated with each. You want your application naming, an internal structure, to be consistent in your organization, and that will depend a bit on your language of choice. Here is a look at the code hierarchy for the pieces of the vehicle position pipeline. There are three lambda functions that make up the vehicle position pipeline serverless application, plus, two utility packages. Within each serverless application that comprises your overall application, you'll likely have shared code. This should be separated out into modules, again, in a language-appropriate way. You may find that there is also a body of code that is applicable across all of your top-level components. This is where lambda layers really shines. Layers can be managed separately from your application code and provides common libraries that can be directly called, significantly easing code management and deployment processes. They can be created and then added to functions where needed, directly on the console. The serverless application model defines a single stack that deploys together. In this case, the serverless application is vehicle position pipeline. Something to remember is that all functions in the SAM model deploy together. That means that deploying a function that is ready to test, for example, GTFS vehicle positioned to the database in the operational environment while your IDE has partial changes to another function, the GTFS vehicle position capture, in the same serverless application will also deploy the partial changes. In terms of lambda diversions in this example, the default version latest of each of the lambdas in the vehicle position pipeline will be replaced when you deploy. You need to manage this or it will cause problems. You should be sure, even in early development, always explicitly version your functions. Another gotcha to watch out for is SAM configuration. If you setup lambda configurations such as timeouts and RAM from the console, it will be overwritten when you deploy from your development environment. Best practice is always manage these at the source code level and redeploy to publish the changes. Good practice for any development, not just serverless, is to really organize your logging. In Python, the logging package is a great place to start. Get in the habit of using logging instead of print. Logs are available in cloudwatch log groups, but can quickly get out of hand without a plan. Structuring the logging makes it more efficient to search and analyze. One important note from the code snippet. When using logging from lambda, secure information can be inadvertently left in the logs. Setting up a filter on the log records as shown will ensure that your logs don't store that information. A fairly new feature, AWS X-RAY, provides you with sophisticated tools for debugging, tracing, profiling, and understanding service com activity. In this example Python code, tracing code is placed around the put\_object call and will generate detailed profiling data viewable on the X-RAY console. As your organization gets more experience in the development of serverless applications on AWS, you'll quickly see the need for more structured processes. There is much support for development management and operations, including codeCommit, codeBuild, codeDeploy, and codePipeline. Definitely have a look.

### **Operational challenges and solutions**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Once you've got a production version of your serverless application up and running, you'll have to deal with ongoing operations. Errors will occur, debugging will be needed, functionality will need to be changed or added and deployed. Testing is your first line of defense. Testing in your development environment is only half the battle. Once you've deployed your serverless application even before you've changed the LAS to point at your latest published version, be sure to test in the operational environment. No matter how extensive the prior tests, there will always be differences in the operational environment. Areas to pay attention to start with execution role, particularly if you're using Cloud 9, you may be able to do certain things in development that won't be allowed in the operational environment. Setting up a basic test on the console will expose this right away and allow you to adjust the execution role appropriately. Sometimes you'll also run into network issues if you're running your Lambda in a VPC, or should be. Don't forget that when a Lambda runs inside a VPC, it loses access to the internet unless there's a NAT Gateway set up. Let's say you've got an RDS instance running in your VPC with no public internet access and you're gathering data from a public API, like our GTFS data from the transit insight dashboard. Your Lambda won't be able to make those API calls. The next thing to look out for is timeouts. You've set an execution time limit on your Lambda, or if you haven't, you'll find that it is defaulted to 15 seconds. Timeout values have to be carefully chosen. Not all functions will be consistent throughout the day. In our transit insight dashboard, there are logically many fewer train trips in the late night hours and on certain days of the week. Processing will take longer when there are more data points. You may need to set a very high timeout value and then monitor your function over a period of time. A day, or even a week, if there are major loading differences. And only then select the timeout value that only fails when there's a real problem occurring. The Lambda monitoring panel can really help you here. Sizing of other serverless and managed services is important, too. If your RDS database is undersized it may get overwhelmed with too many connections or too many concurrent inserts when the load gets too high. For example, in our transit insight dashboard, we're getting data from as many as 20 different agency feeds every 15 seconds. Those are getting processed and inserted into the database in close to real time. An undersized database may not be able to handle the inserts in less than 15 seconds, leading to a backlog that could snowball out of control. Automatic scaling of your Lambda can also become your enemy here. If you haven't set a concurrent execution limit, you could end up with hundreds or even thousands of concurrent executions. And more quickly than you might expect. The first line of defense against this is to set reasonable concurrent execution limits for your functions. In some cases, this may be as low as one. In our scenario, 20 was a good starting point. Finally, when errors occur, you need to deal with them appropriately. For many message-oriented event-triggered Lambdas, this is where DL queue, dead letter queue, comes into play. With DL queue set up, failed Lambda invocations have their event payload put into the DL queue. Here, you can configure another Lambda function to deal with that error. And of course, you'll want to set cloudwatch alarms as well. Cloudwatch dashboards can show you when and where alarms are active, and you can configure them to publish to SNS topics to activate other error handler Lambdas, or even to send emails to the operations team.

### **Next steps**

Selecting transcript lines in this section will navigate to timestamp in the video

- That brings us to the end of the course. But, we've only scratched the surface of what serverless development on AWS has to offer. This isn't the end and I encourage you to learn more by exploring the many tutorials available on the AWS website. AWS provides extensive free-tier access to many cloud services. So get in there. Go build and experiment. And I also recommend you to check out our many courses on AWS here on LinkedIn Learning. My name is Aileen Gemma Smith and thank you for watching.