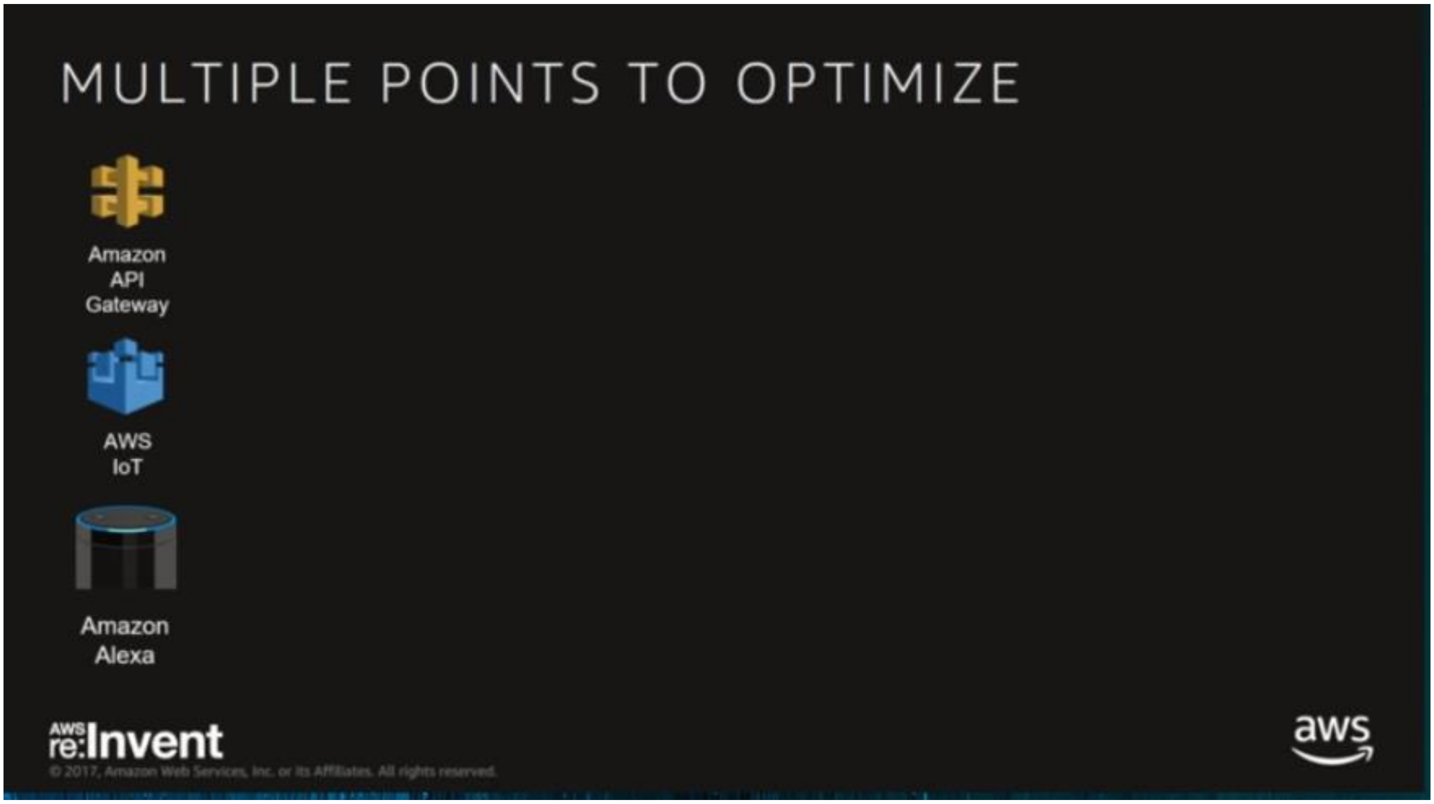
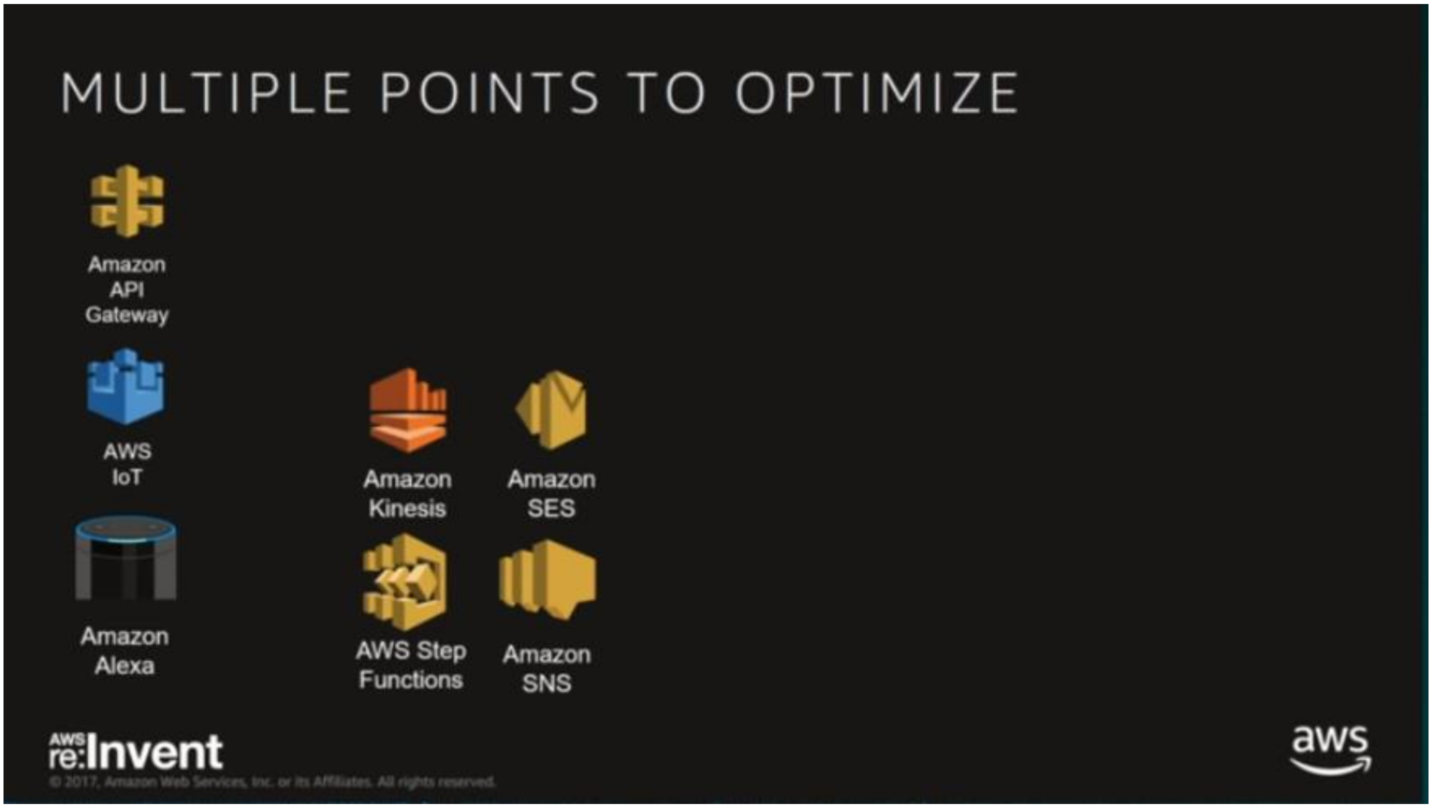
This is an advanced session for Serverless apps using lambdas and API Gateway in production

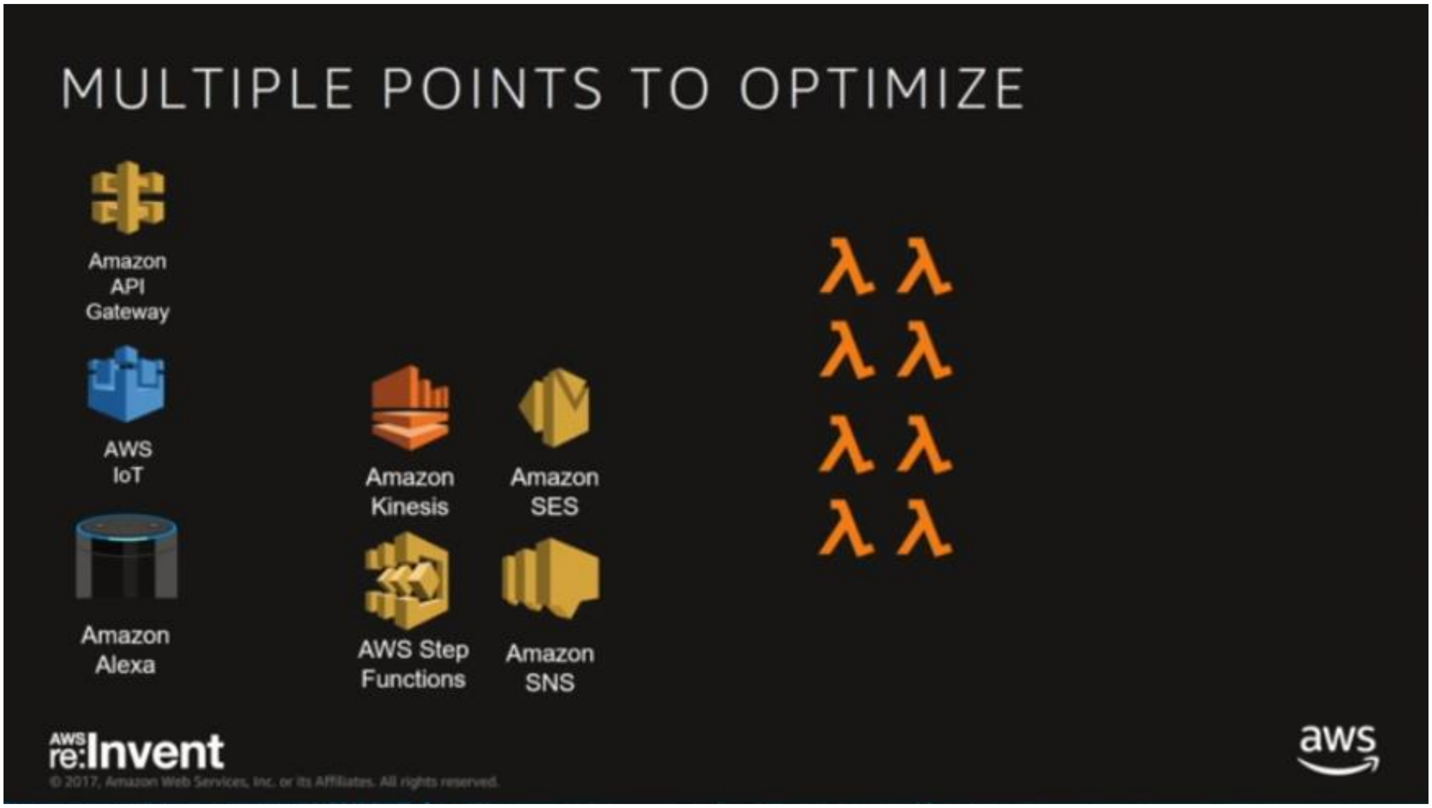
There are a few core principles when building a serverless infrastructure



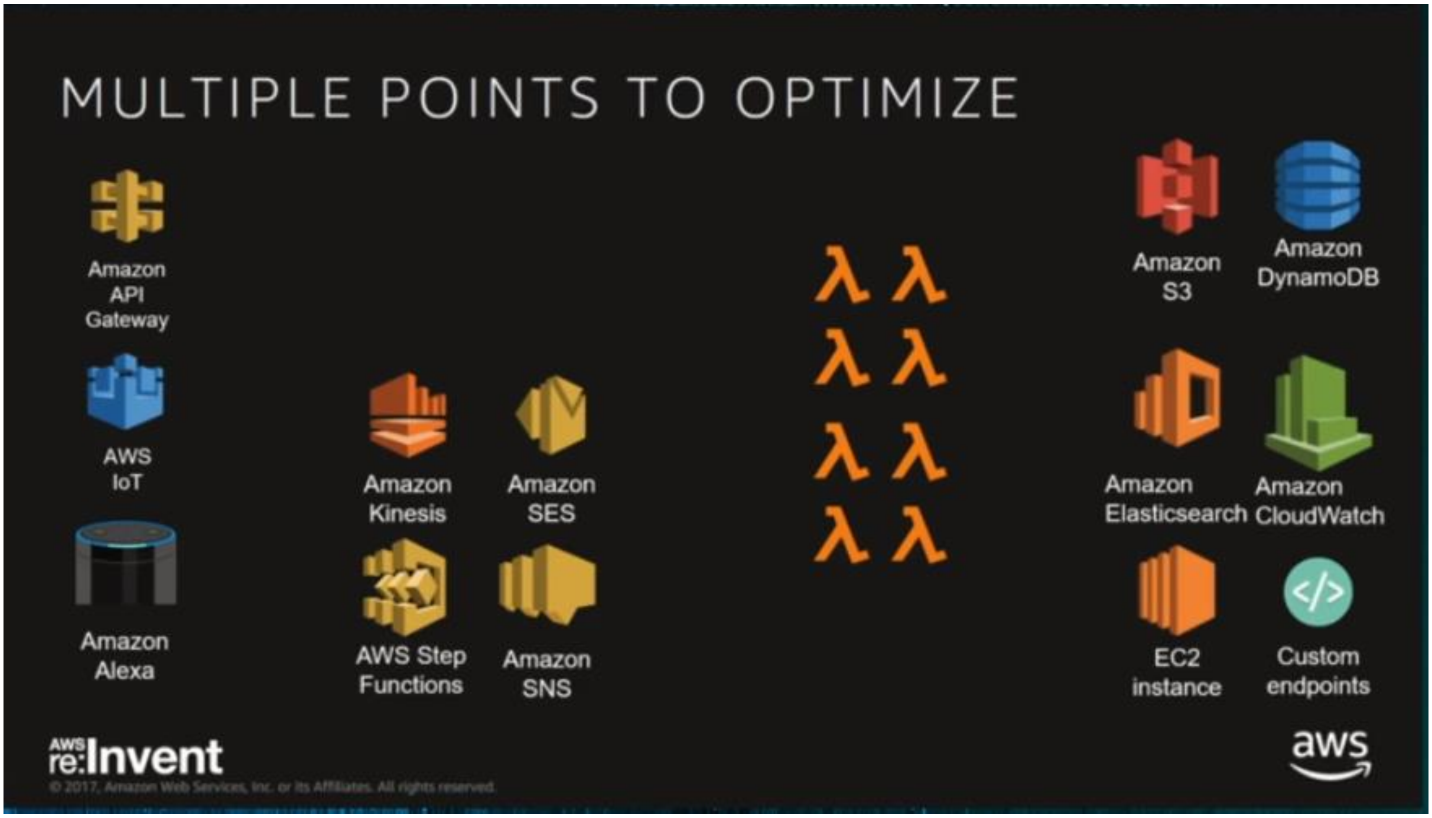
A typical serverless architecture has multiple layers, you have the Access and Authentication layers across your multiple sets of services responsible for deciding the ingress, access and authentication to your backend layers.



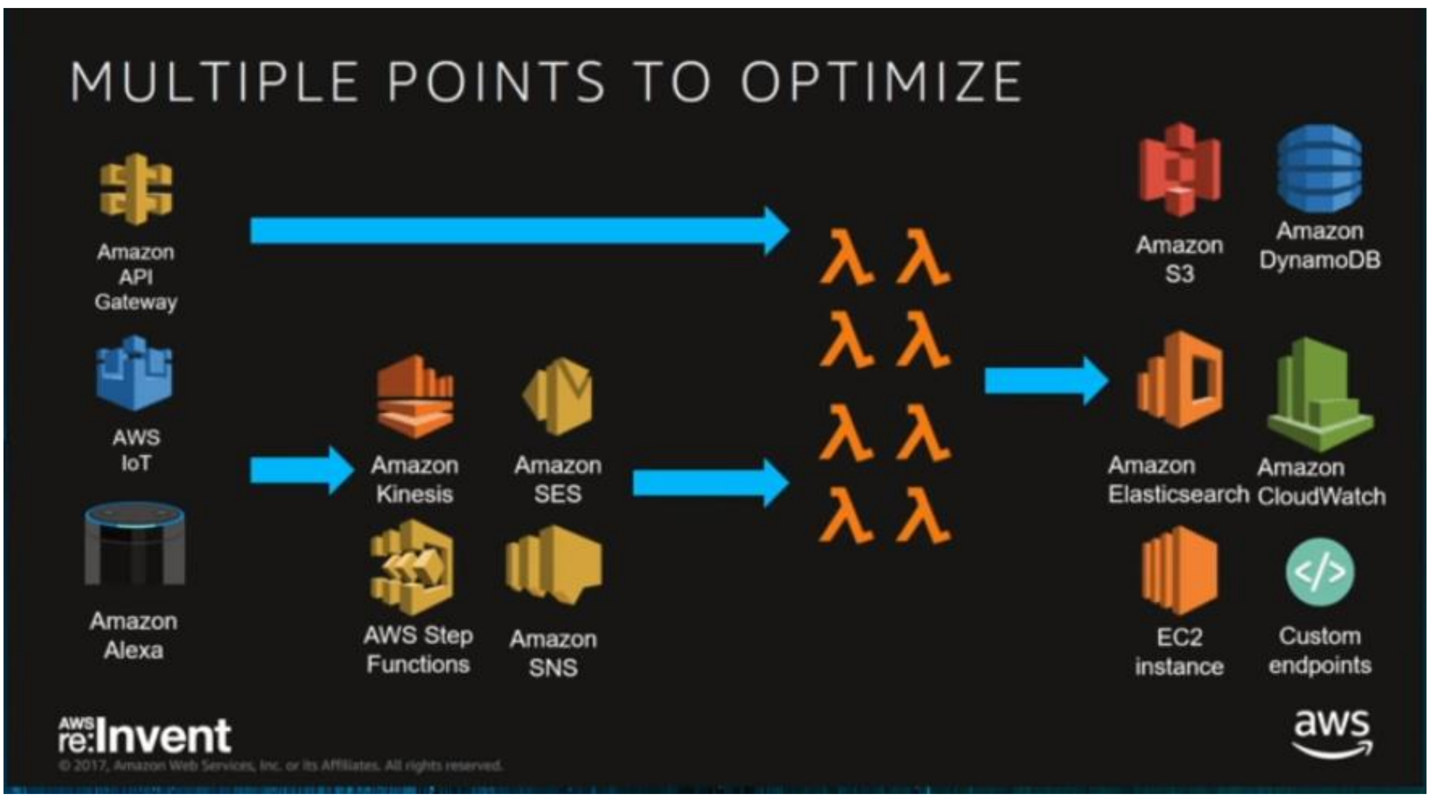
There often is also a messaging and orchestration layer responsible for coordinating calls to your backend with things like buffering, state management, collation of records, batching, etc.



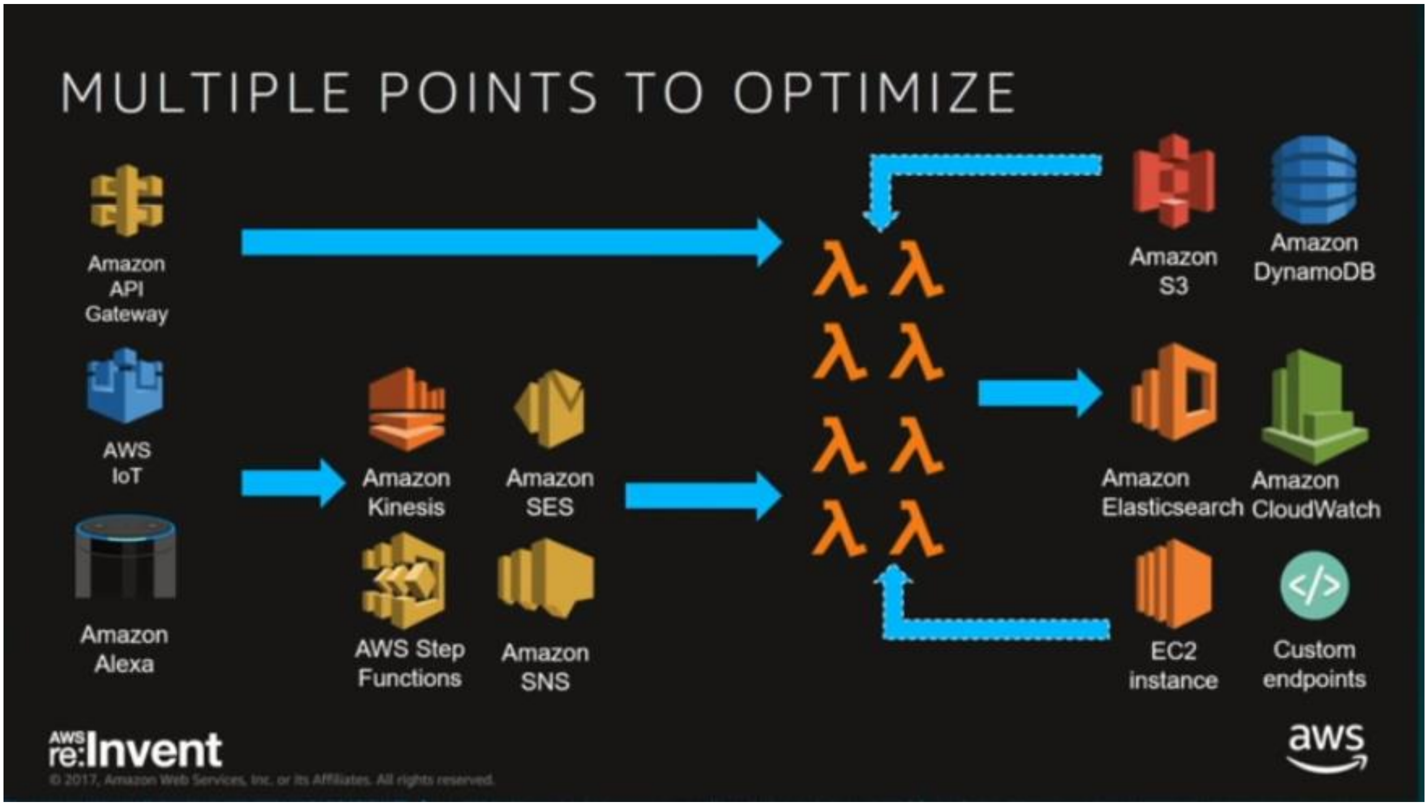
There is also a compute layer where all the business logic resides as lambda functions that gets executed against your backend



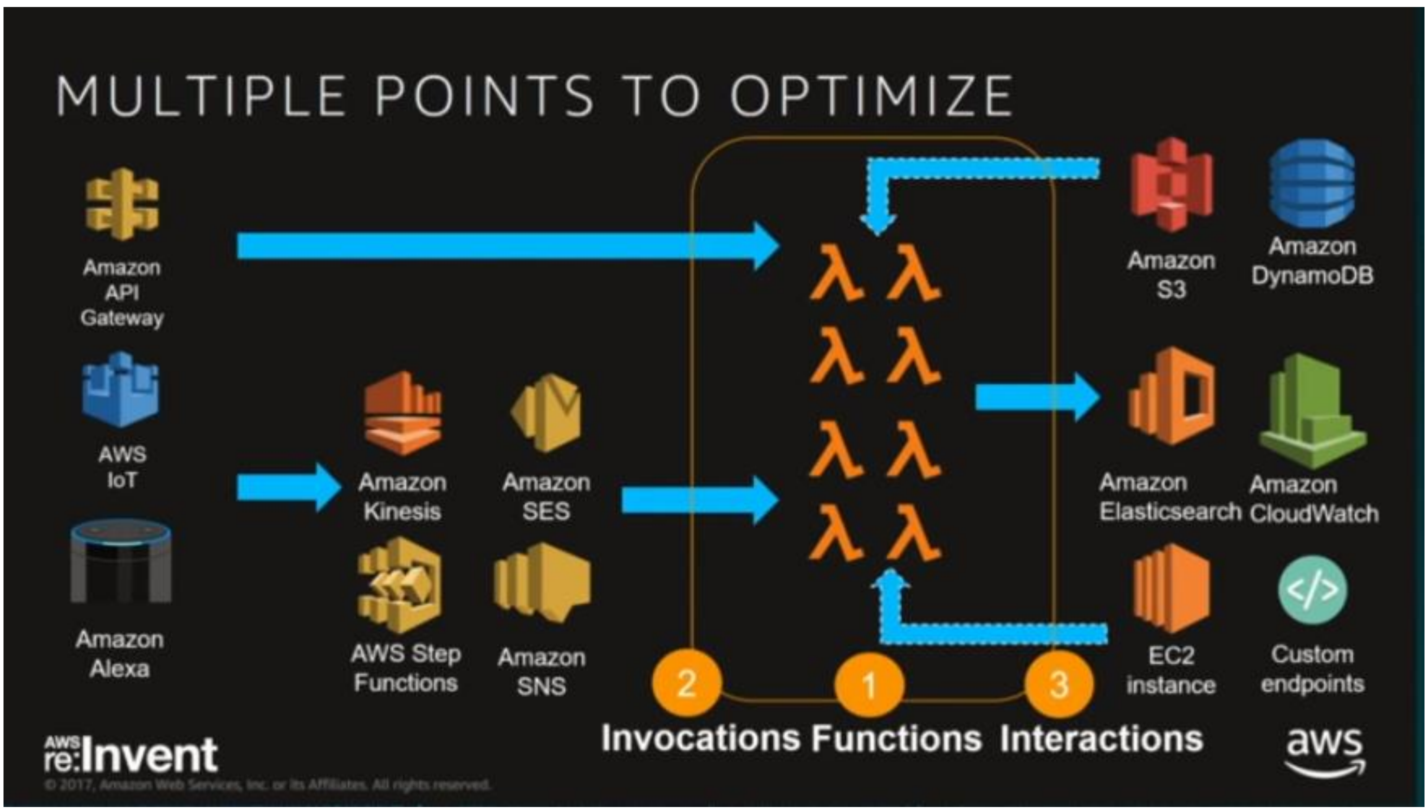
Finally, there is the data layer responsible for state, durable information that your applications need or third-party APIs that you access via available APIs.



These same layers apply whether you are building end user apps which has certain flows

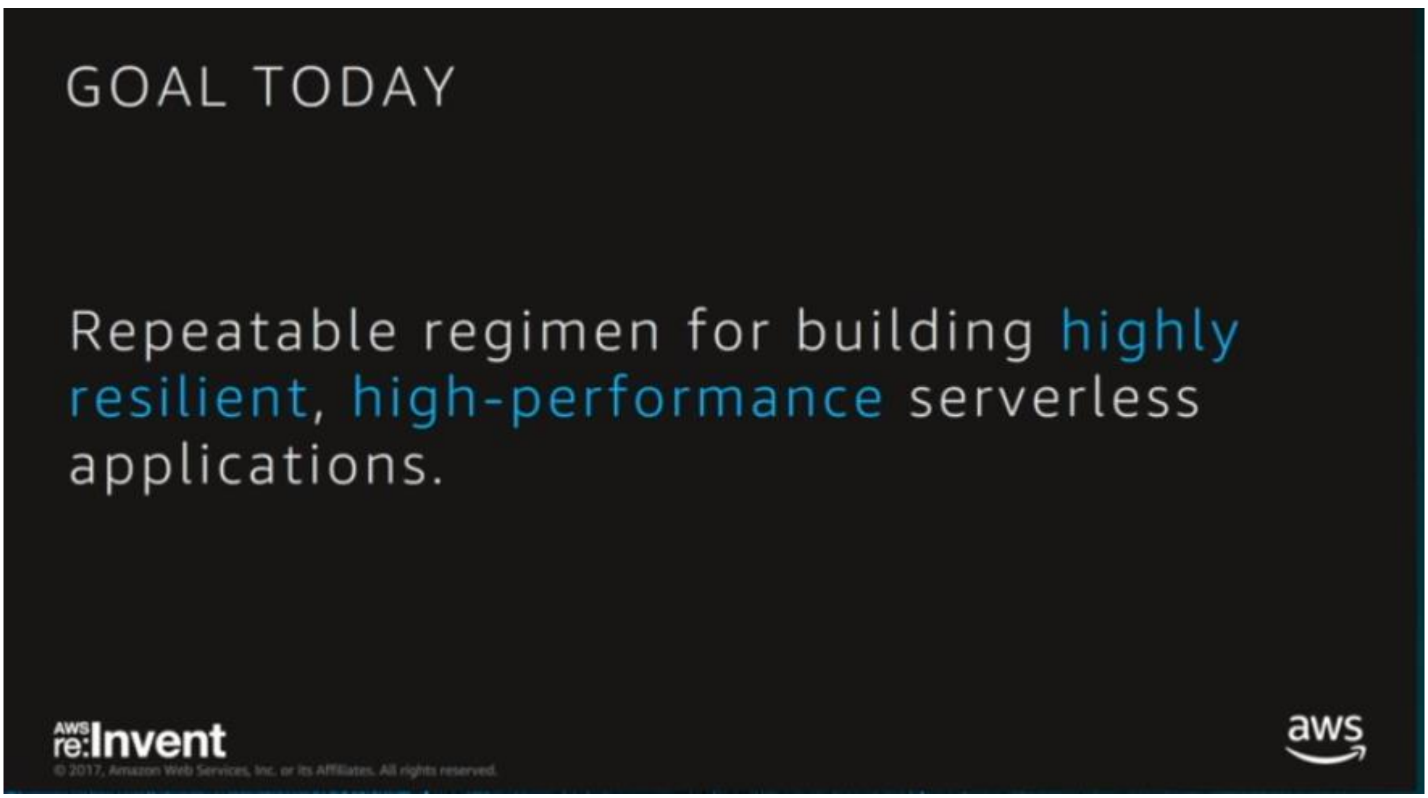


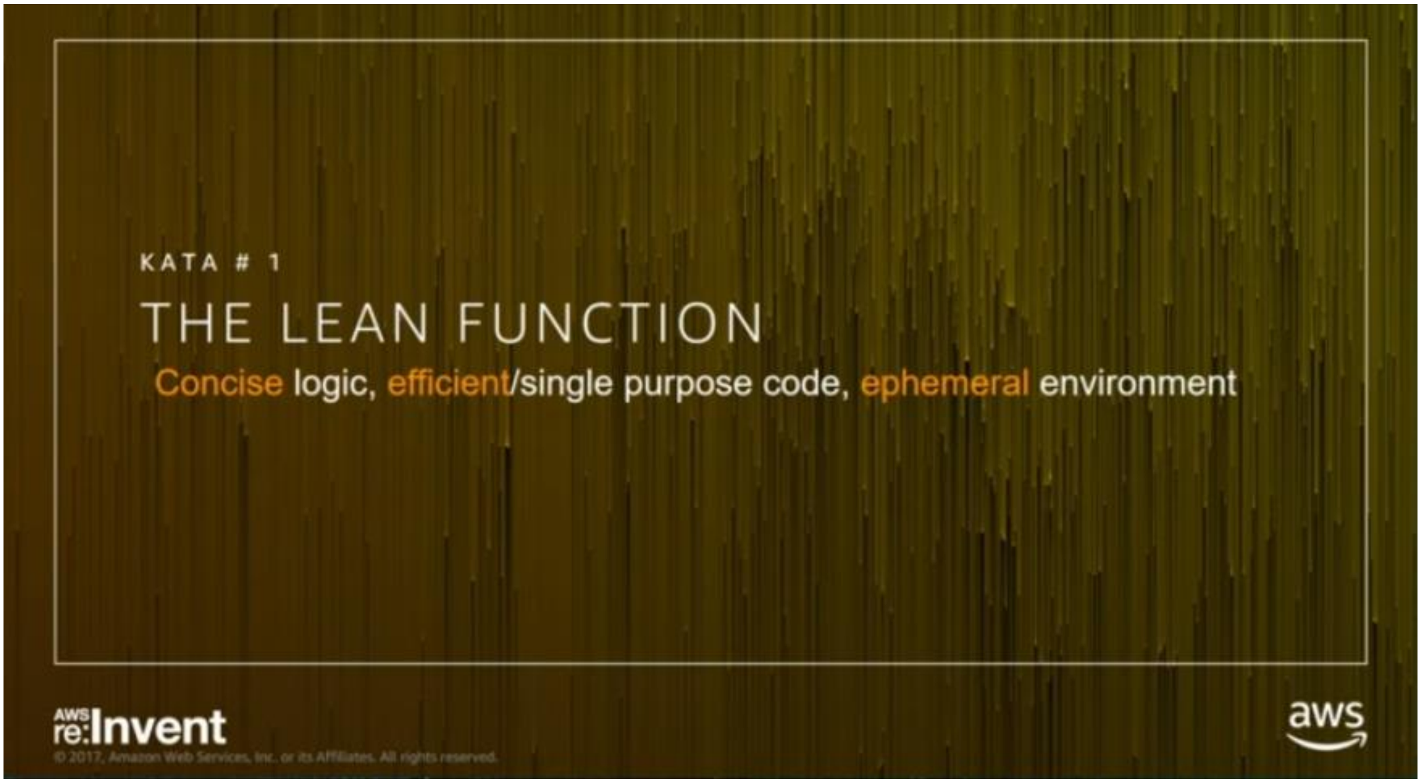
Or when building backend apps where only a couple of the layers are needed



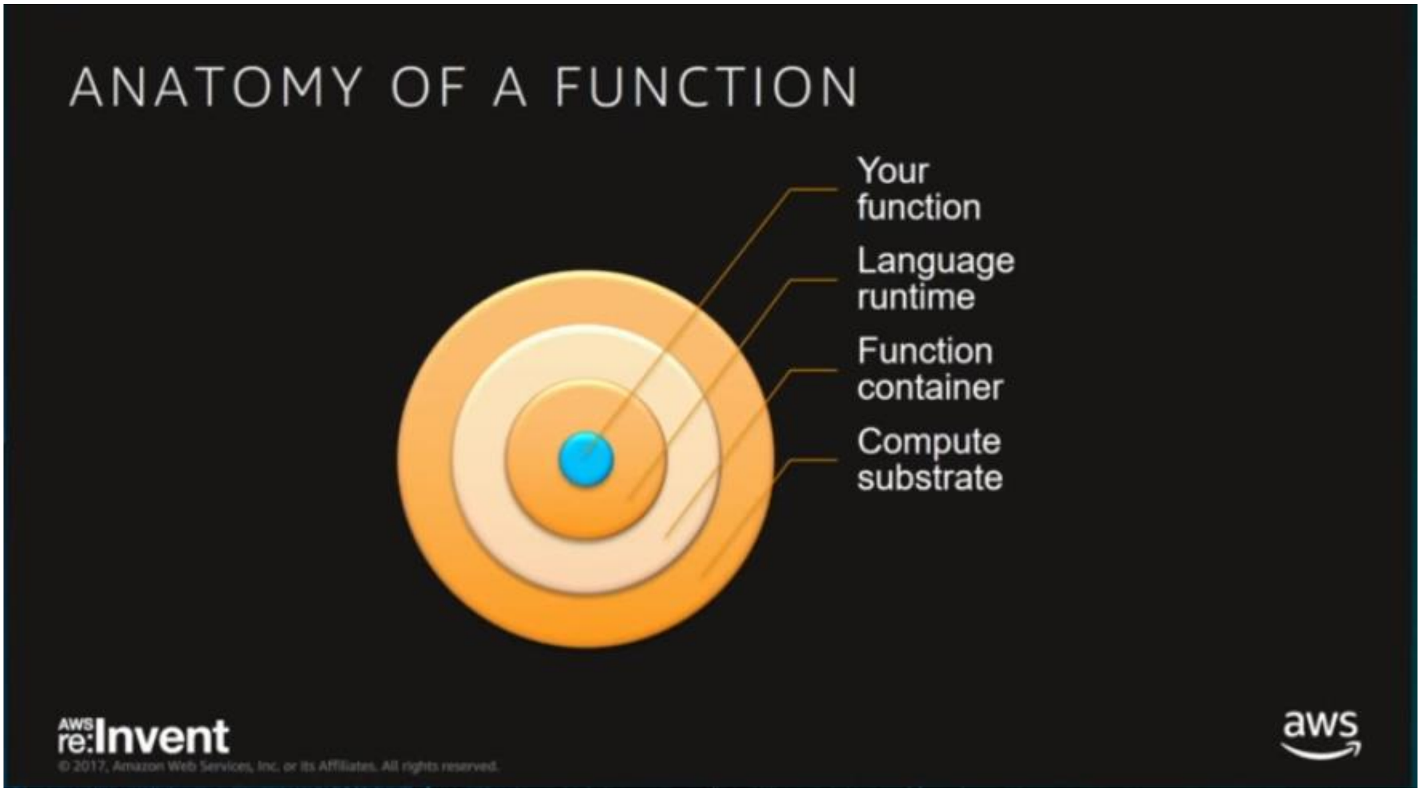
We will be focusing on one specific area for optimization here, looking at your lambda functions, the invocations of these functions, and the interactions these lambda functions have with downstream services.

We have distilled these practices into 4 main katas

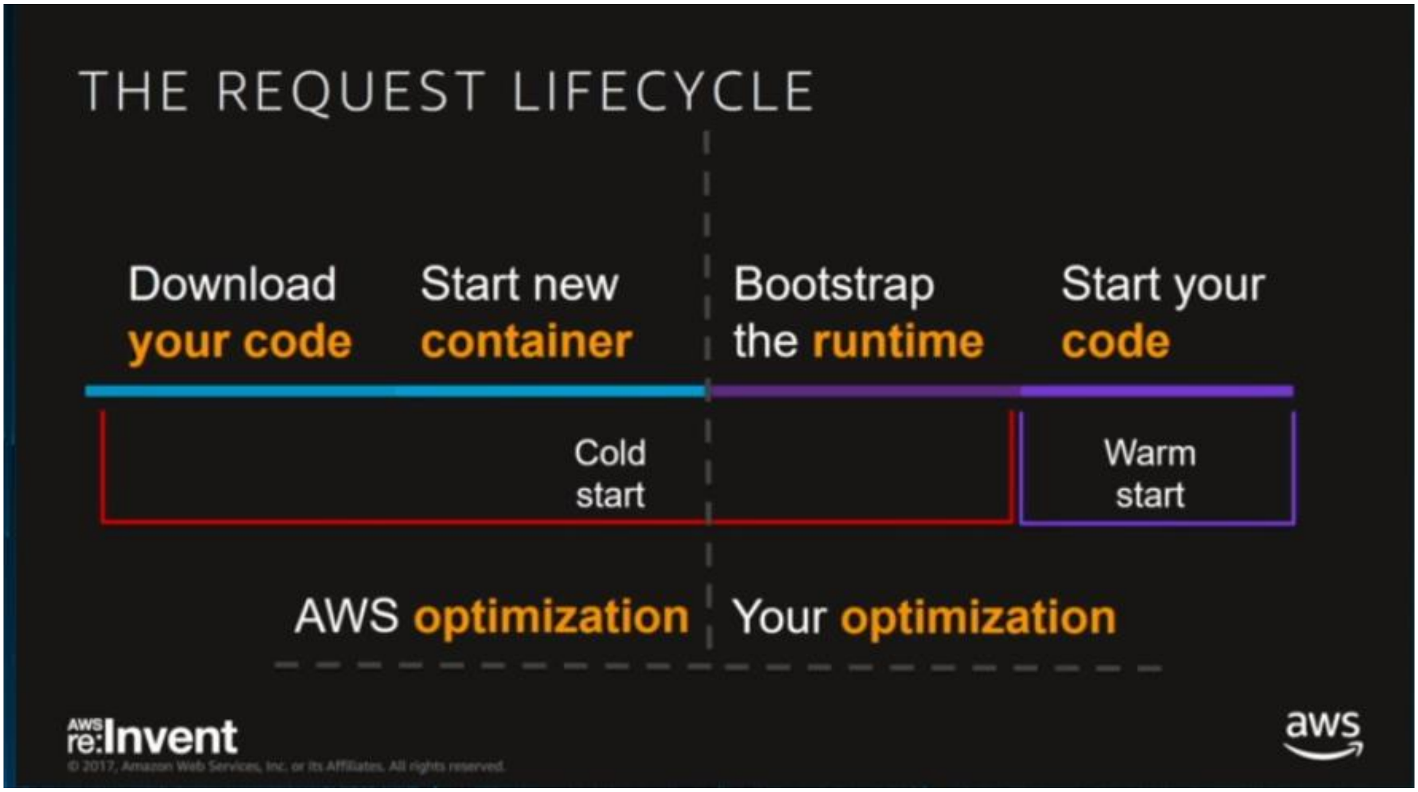




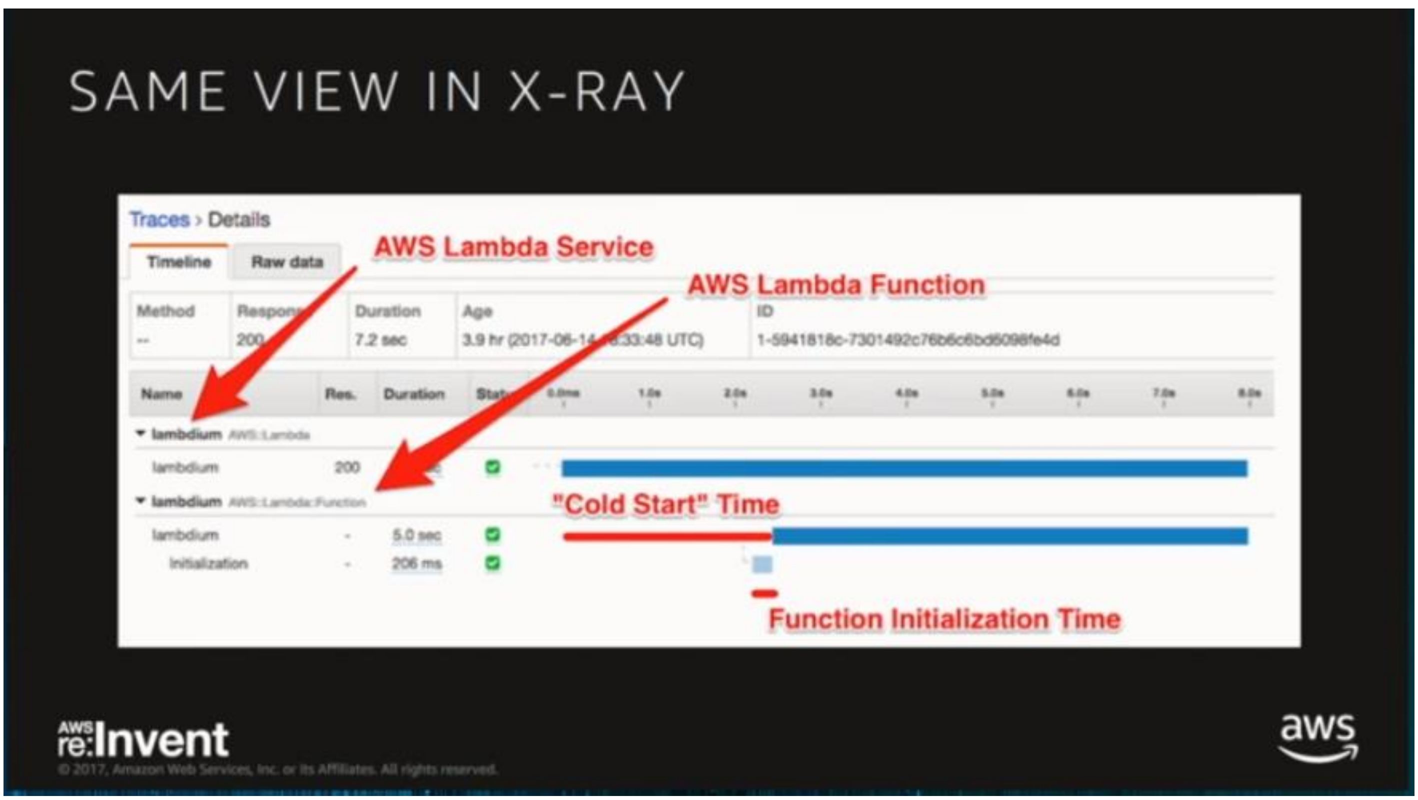
The lambda function itself is the most critical part of your architecture and needs to be optimized, it is the glue for all the other different components to talk to each other, controls your business logic, and it has significant implications on how the overall architecture behaves.



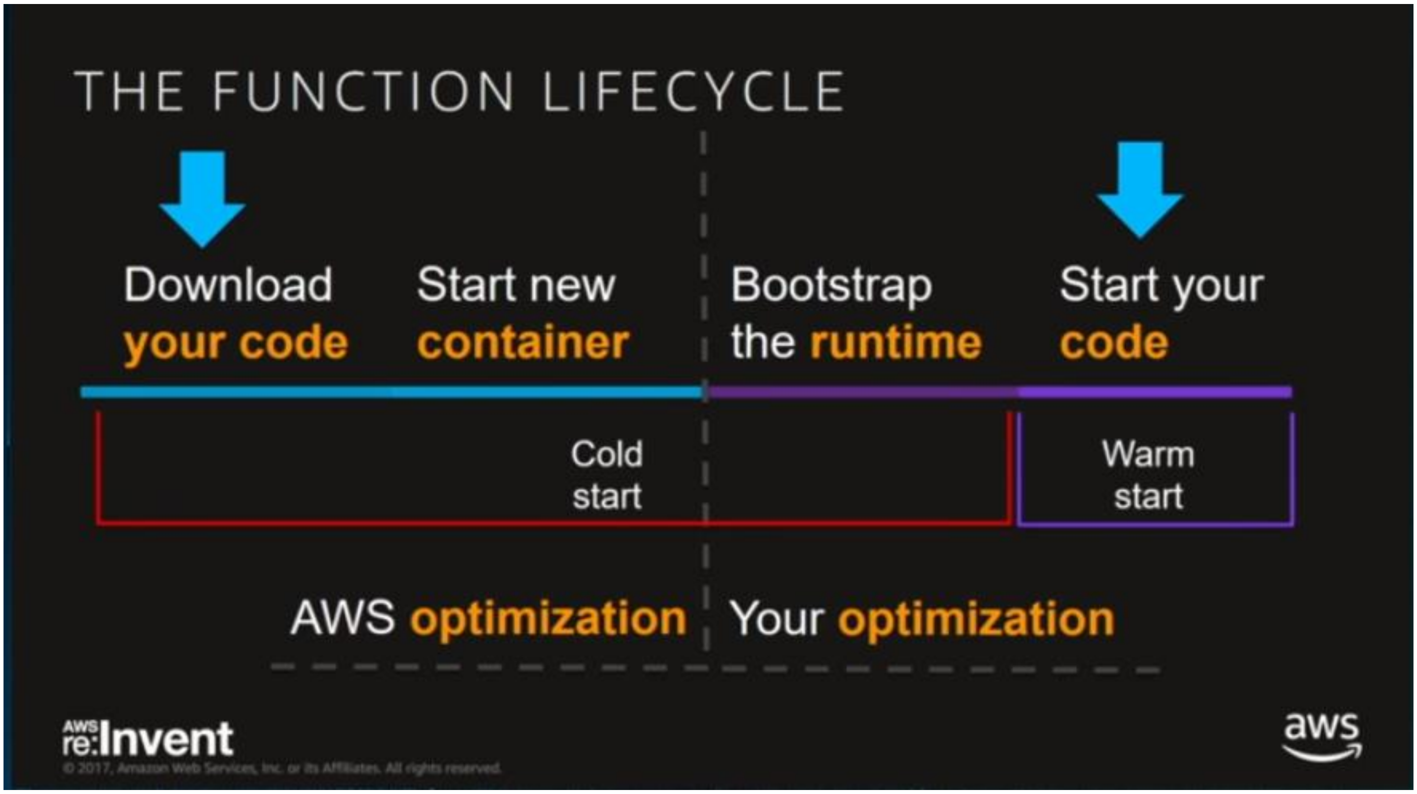
Your function is the code that is executing inside your lambda function’s handler function.

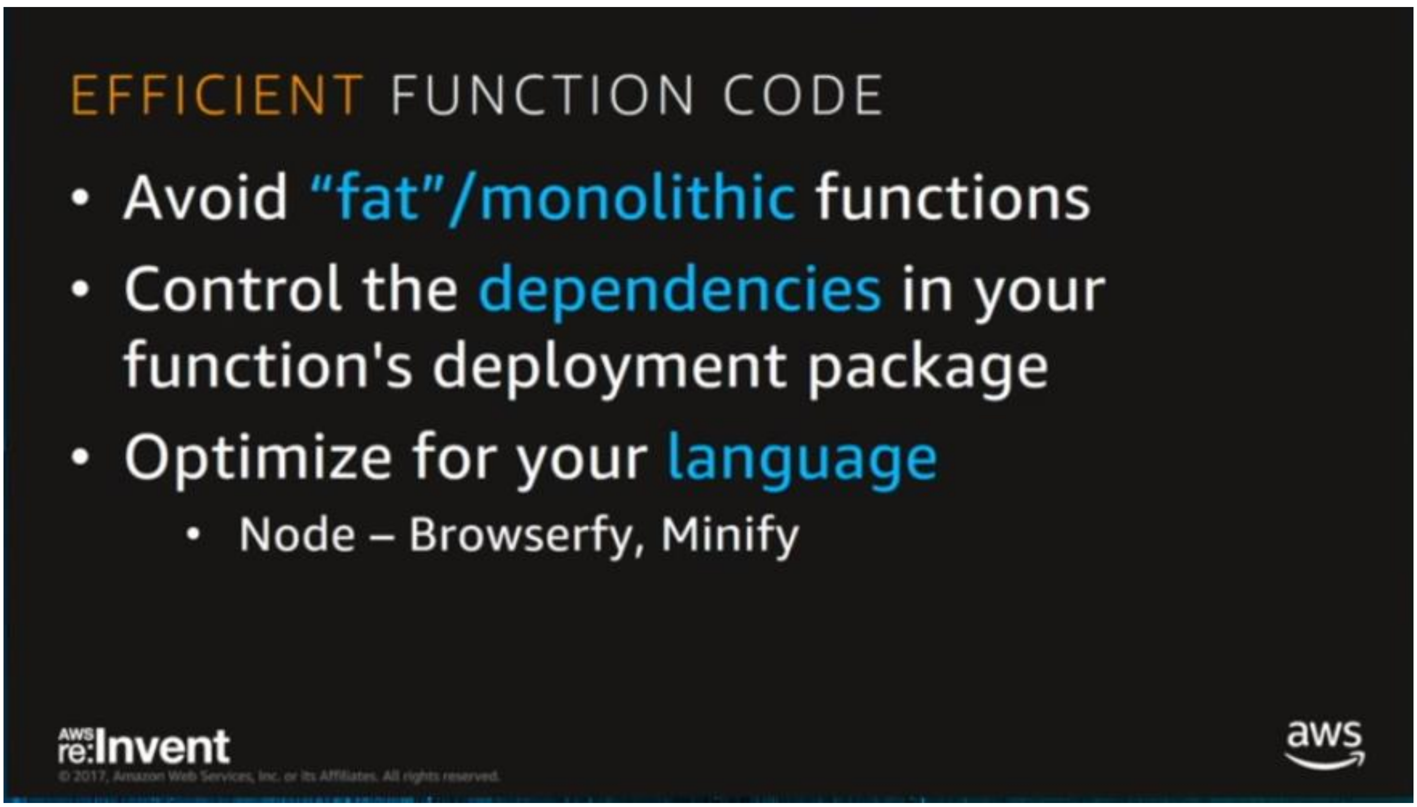


The application you are working on is going to be predominantly in the warm start section and will save you cost when optimized and reduce end to end latency for your calls.

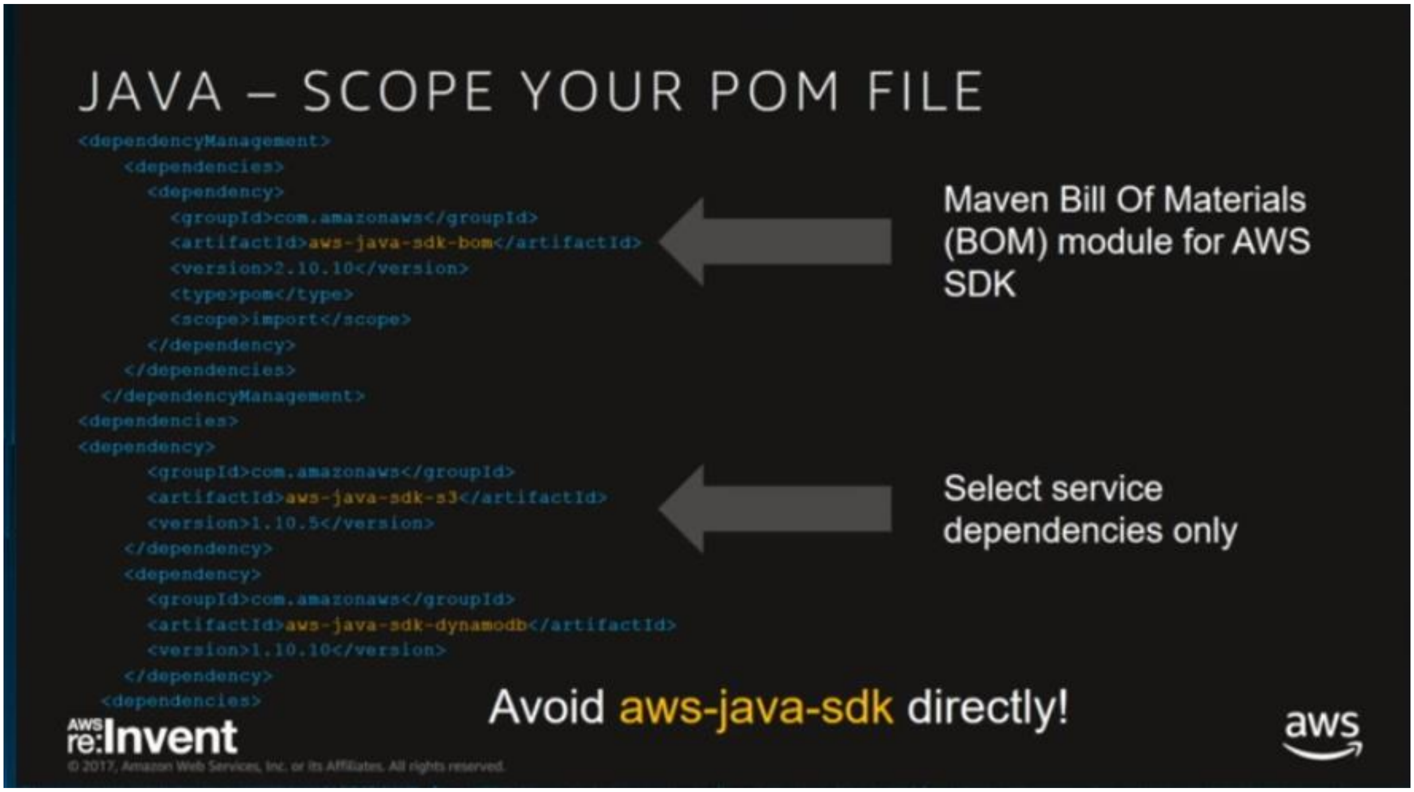


You have visibility into what goes on at runtime

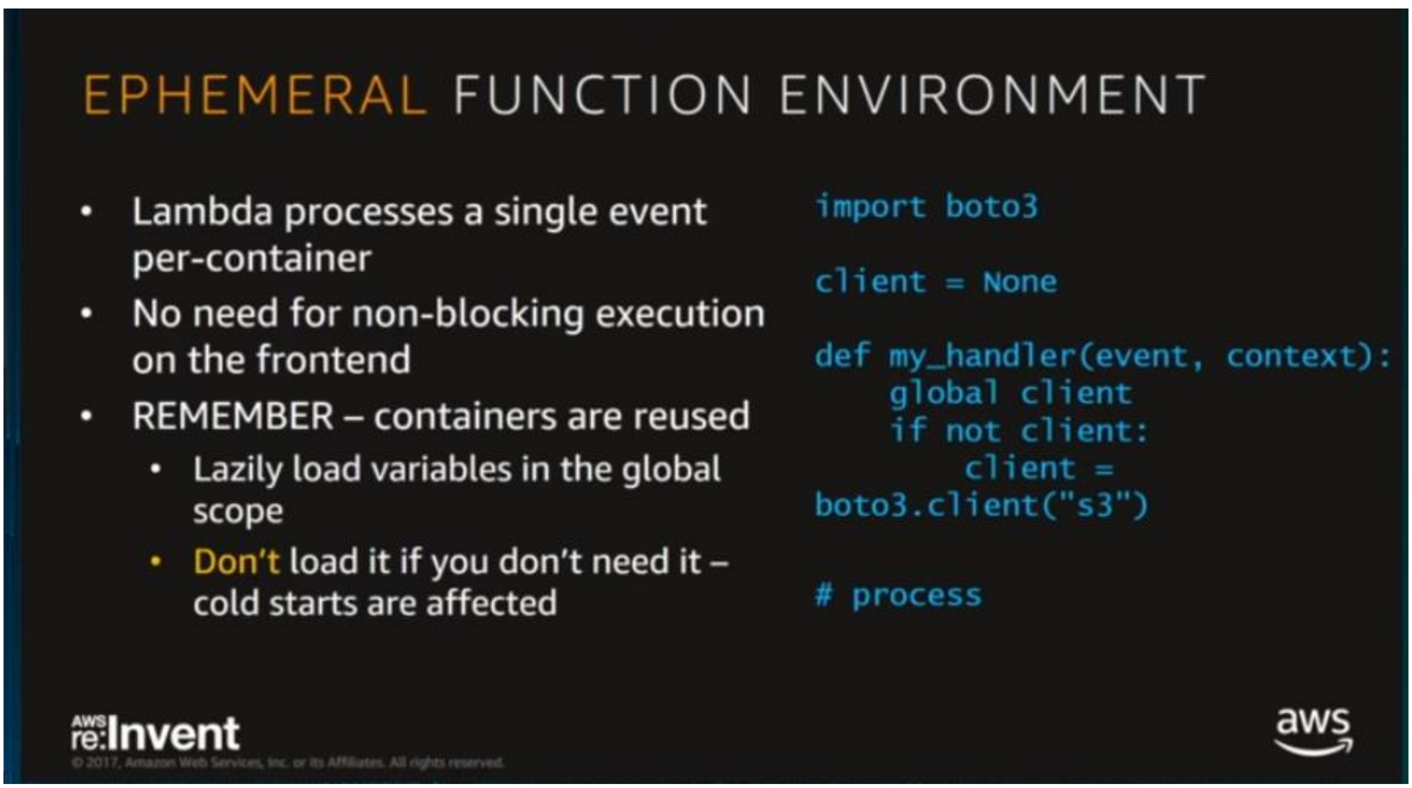


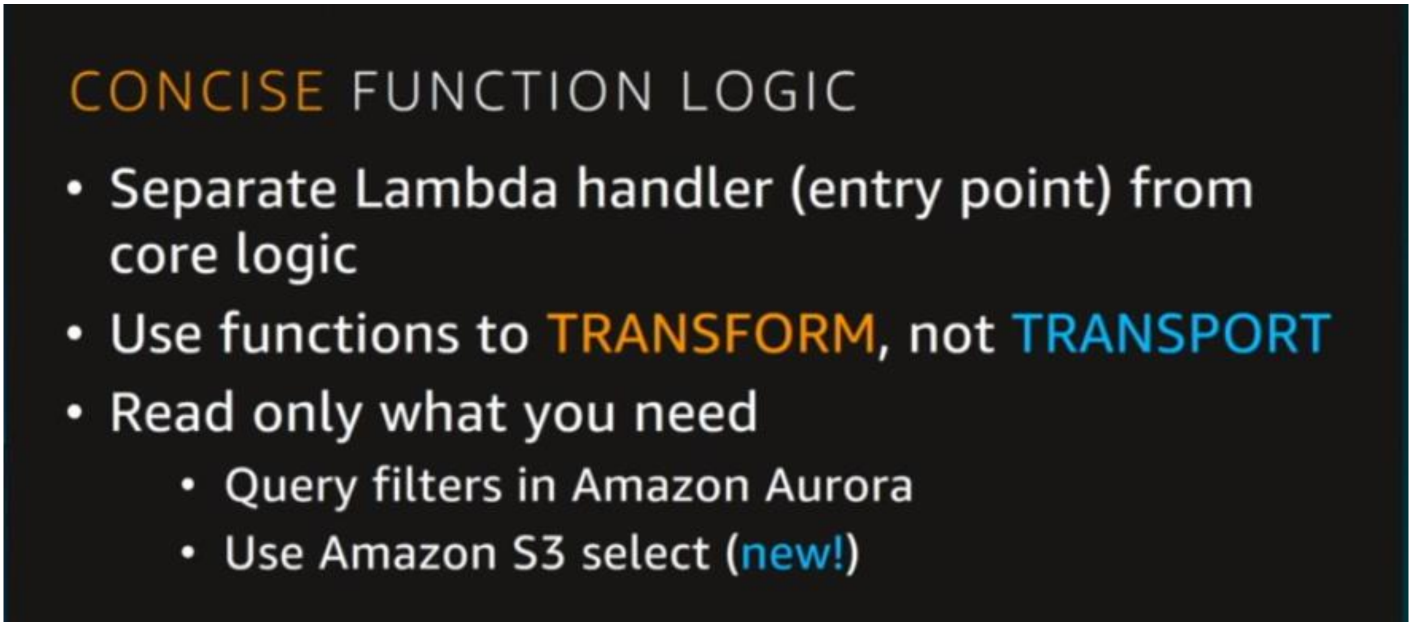


Don't write a bunch of conditional and routing functions, it is always better to write them out as separate and orchestrated individual functions for better performance. You need to compress/shrink the archive that you upload into your lambda function.

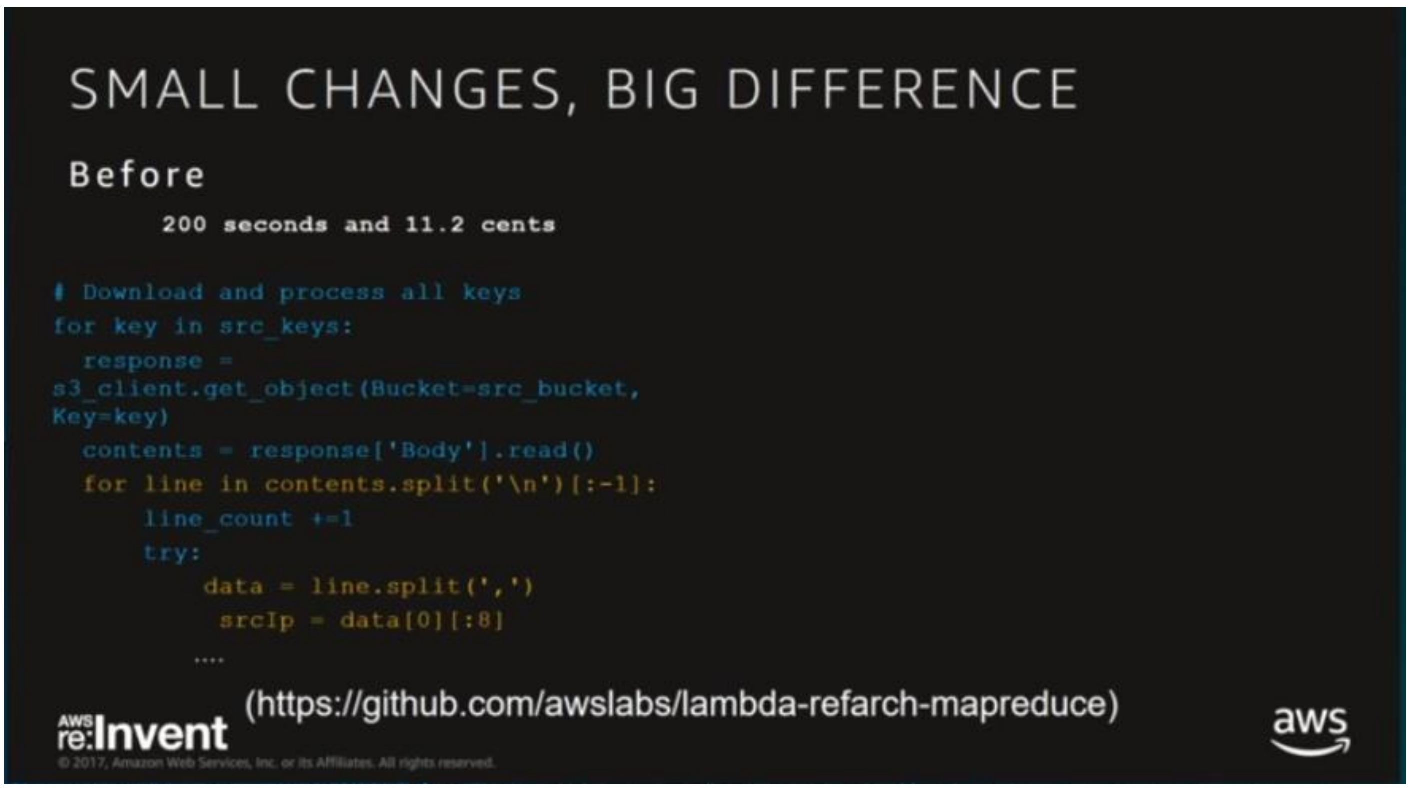


You need to pick the exact dependency that you are needing in your POM.xml file instead of the default of using the entire aws-java-sdk itself. Always strip out unneeded dependencies.

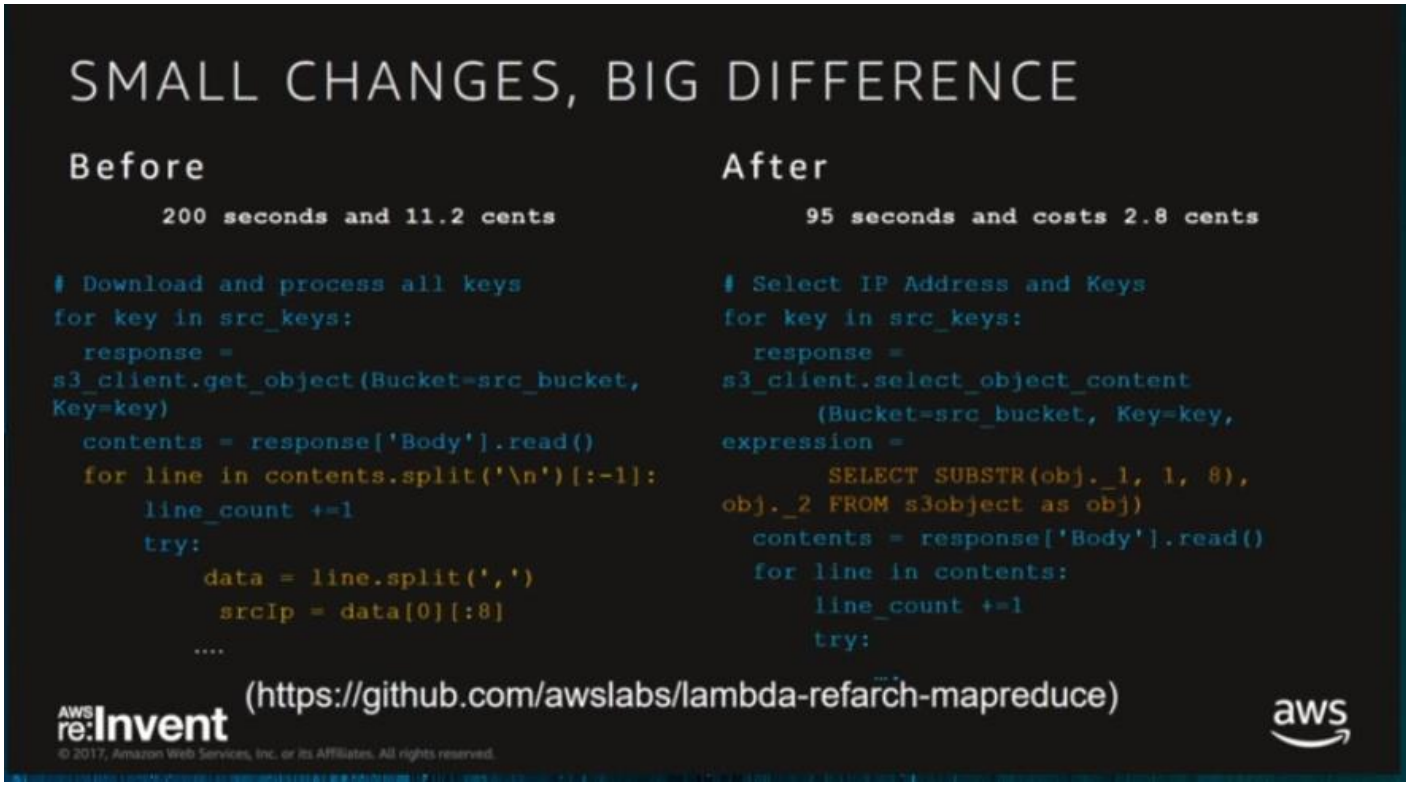




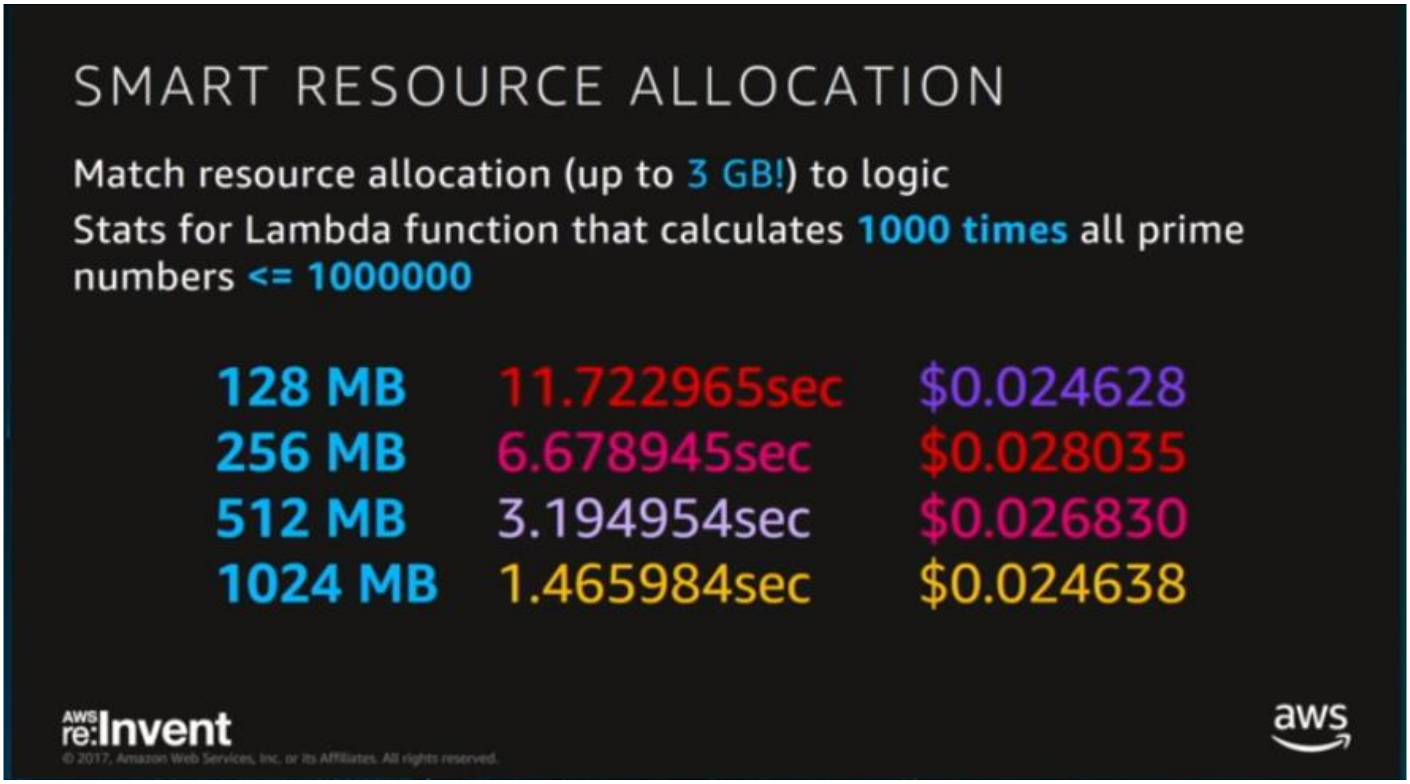
If all your lambda function is doing is I/O then there is probably a better way to do it instead of waiting for data to come in, the more important thing is to make sure your lambda is doing ***TRANSFORM*** things instead of just ***TRANSPORTing*** data back and forth between components.



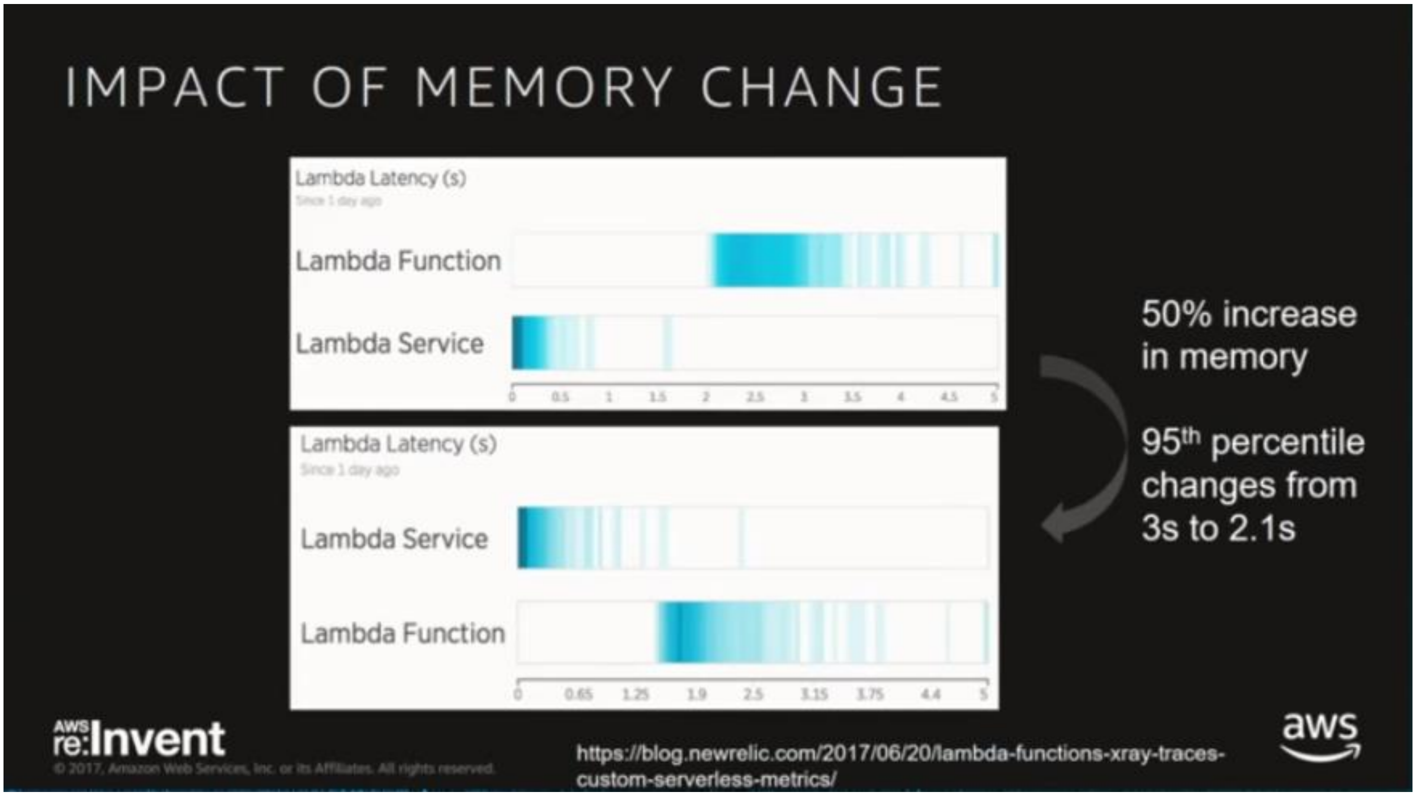
This is part of the AWS Reference Architecture for running MapReduce using lambda, including code samples on GitHub. This particular example is processing about 770 million rows in 200 seconds for 11.2 cents. This specific example is looking at network log data and looking for specific IP addresses and doing an aggregation of IP ranges.



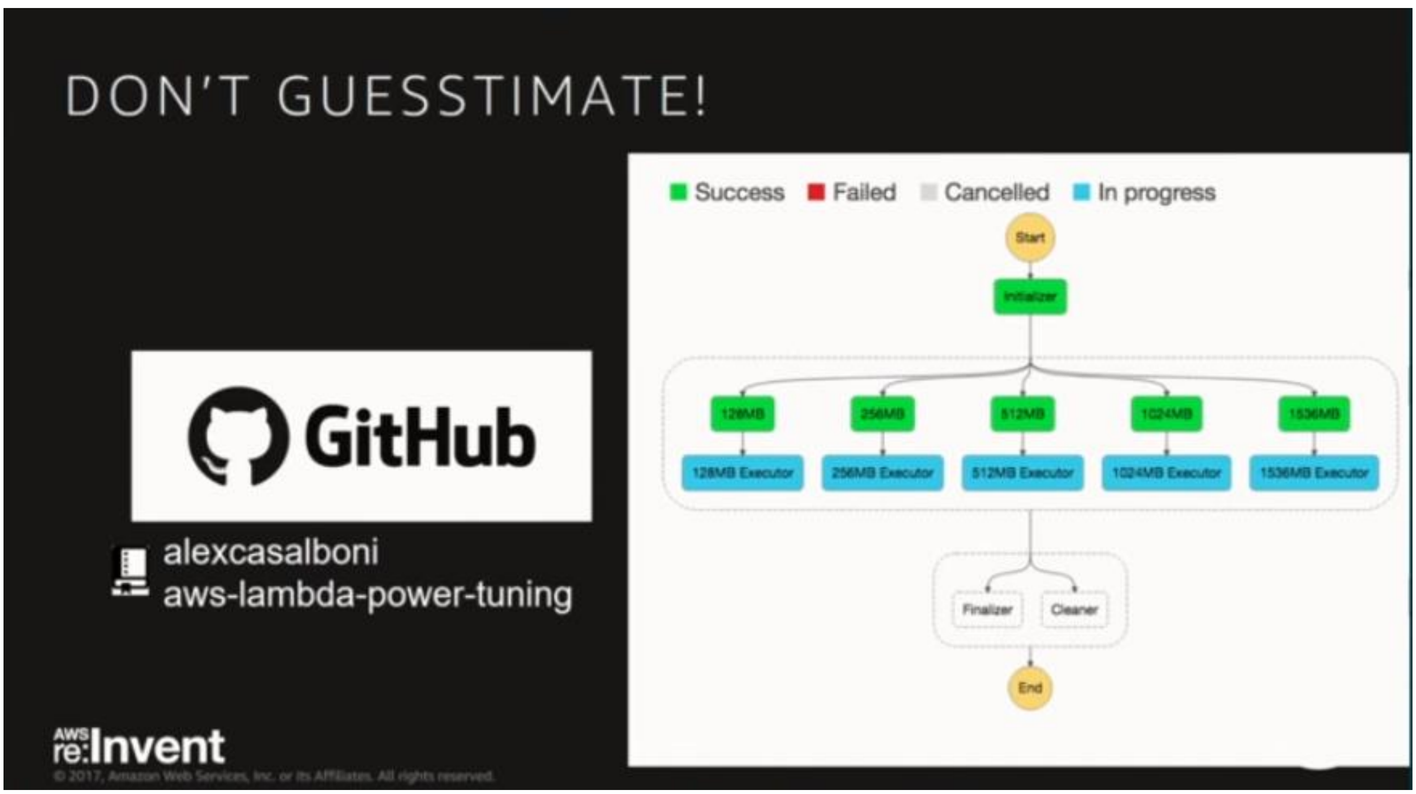
We took the highlighted sections and replaced it with the new ***S3 select function*** which now only looks at the IP from the entire row and then pulls it out. This reduced about 50% of the cost and time to run.



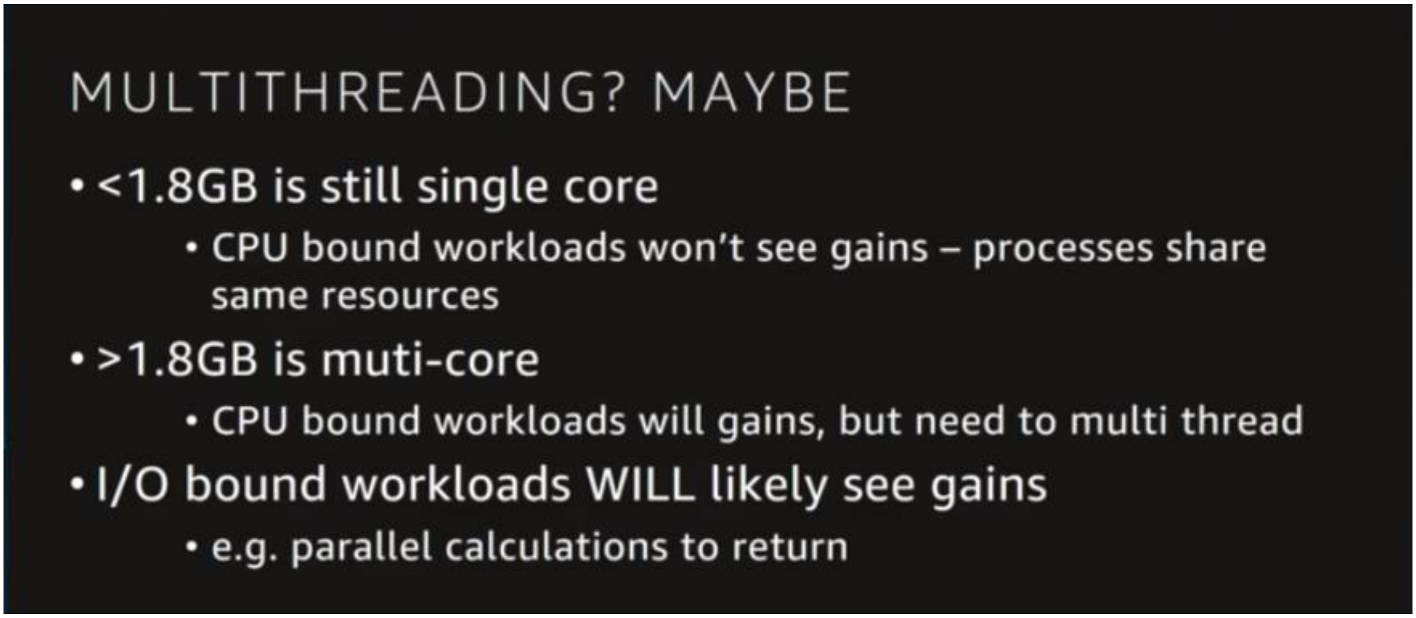
You can trim your logic, your dependencies, your packages, but you also need to make sure that you are tuning your compute logic or compute resources available to match the code that is actually executing. Lambda now allows using up to 3GB of memory use



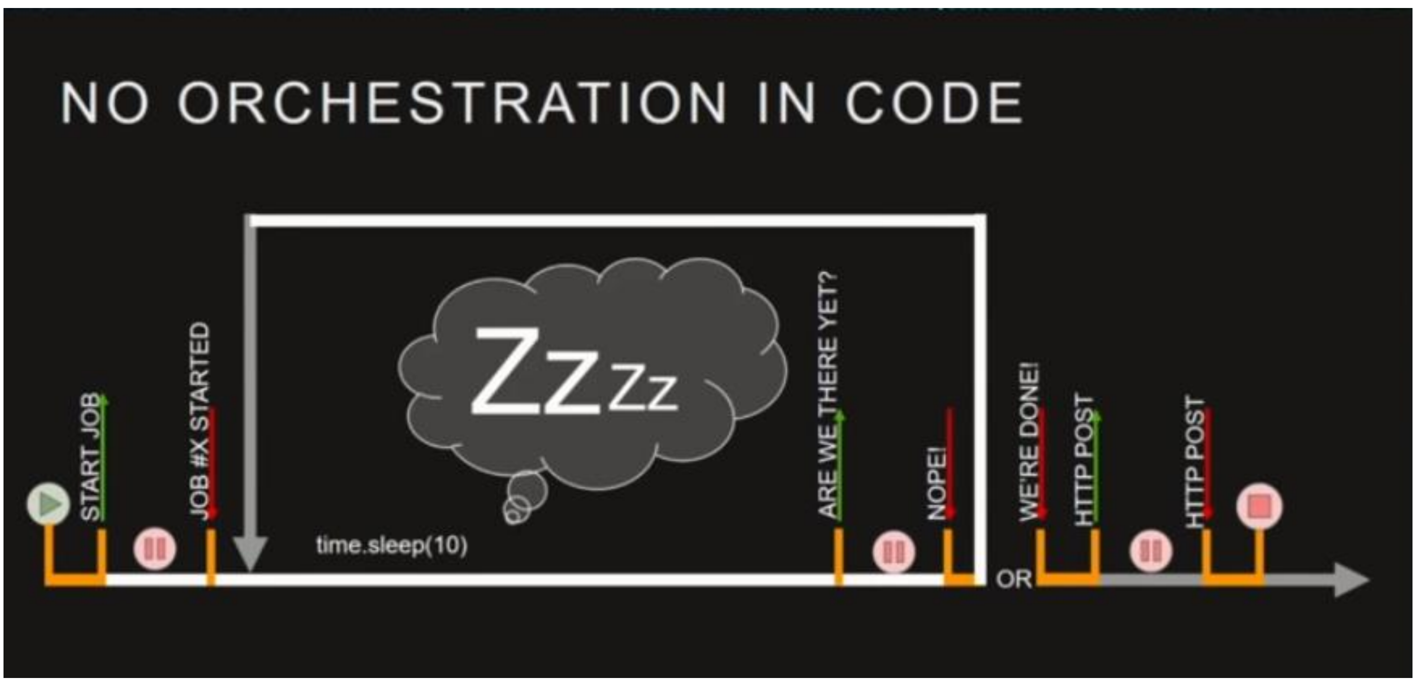
There are many tools for seeing the analysis of what dial settings you need to choose for your compute needs like ***CloudWatch, X-Ray***, etc.

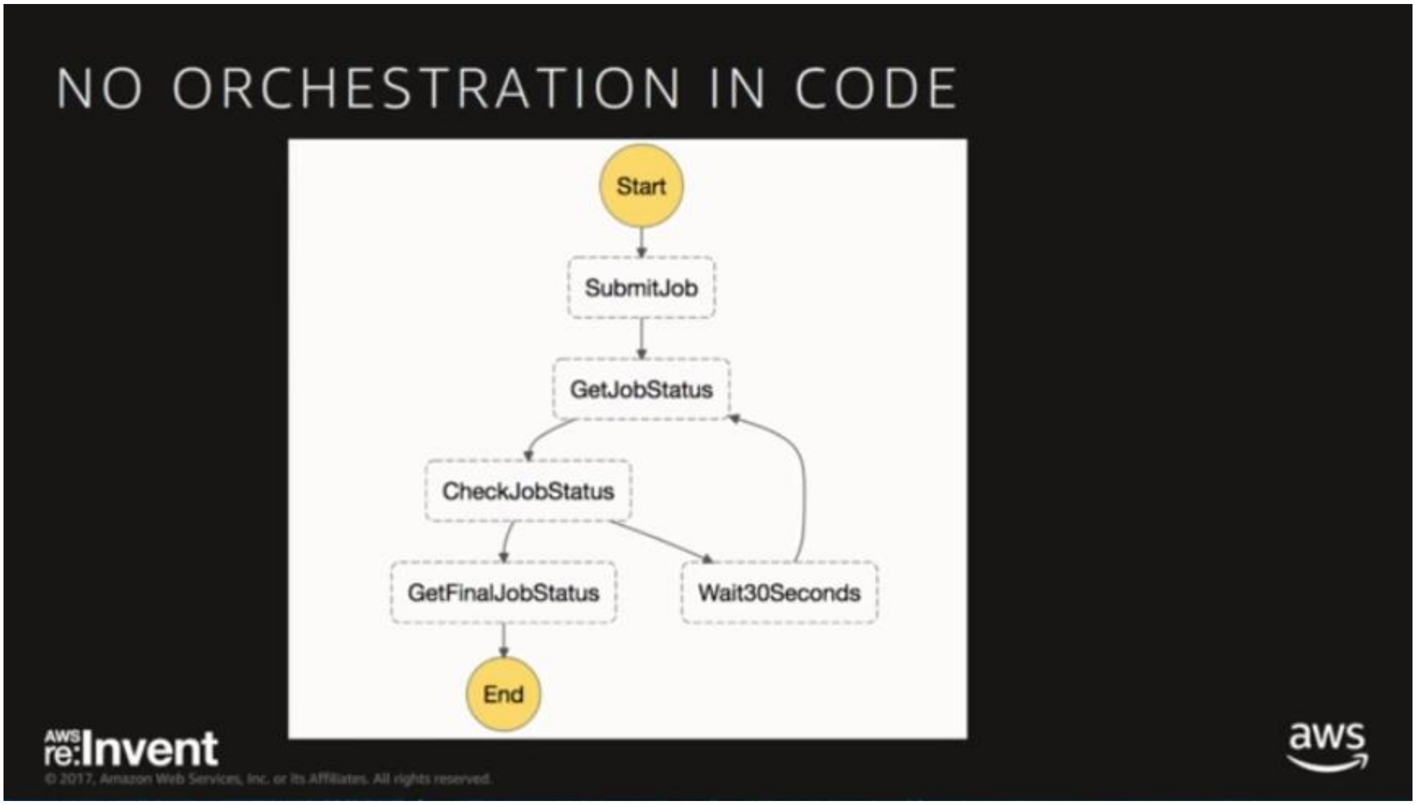


This ***Lambda Power Tuning tool*** is available for free on ***GitHub*** for you to do compute needs analysis, it uses step functions to automatically switch the memory use of your function, run it in parallel for multiple runs, aggregates the output data from ***CloudWatch*** of the execution time and memory settings, and tells you what the appropriate setting should be for that function. This is an important optimization step that you ALWAYS need to do, make this part of your automation and testing regime as you may have changes based on payload, architecture, infrastructure, etc.



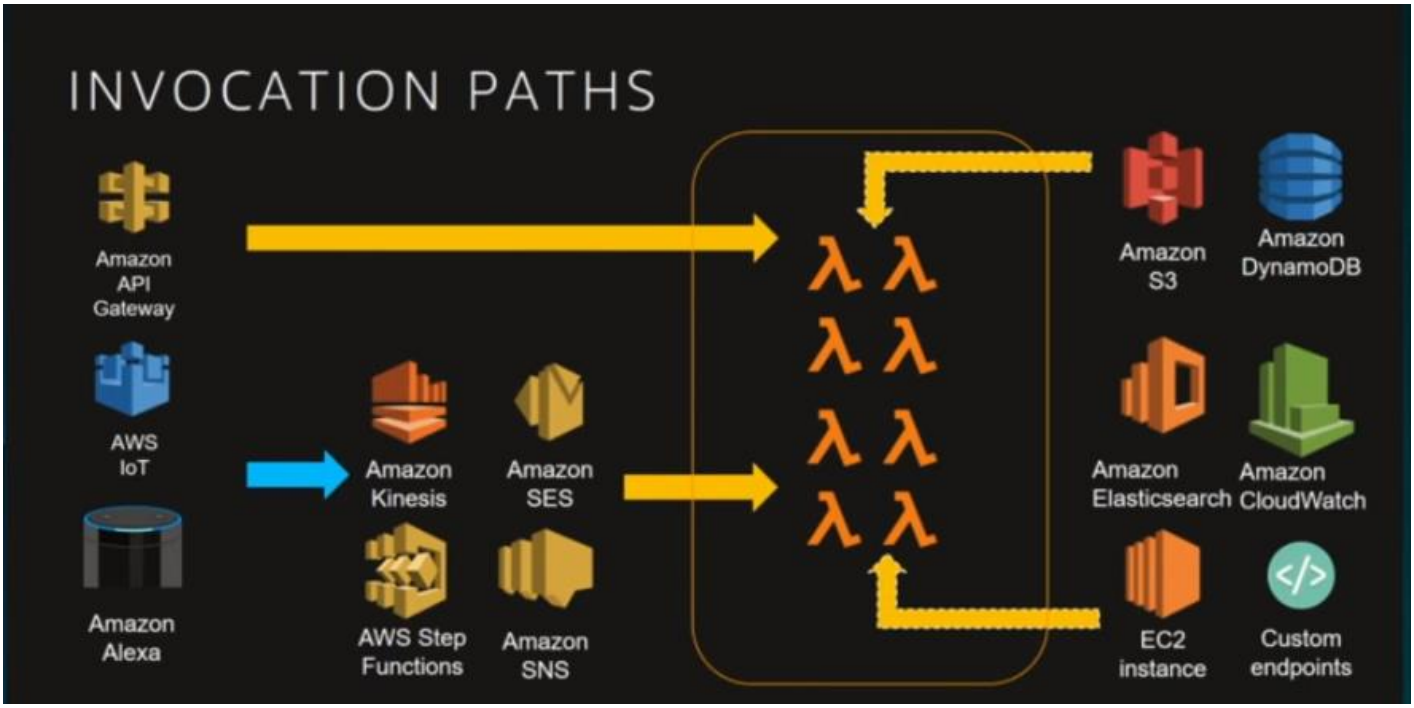
If you have a CPU function, you might be better not worrying about multithreading your lambdas since the function will get access to all the resources allocated to it within the 1 core container. But if you are doing I/O bound workloads, you can do optimizations by running the I/O on a separate thread doing a download and calculation separately, and doing your core computation on another thread. But for applications with lambdas using more than 1.8GB memory, we do have multi-thread functionality available. This is for cases like ML models that lend themselves well to multithreading, you can use lambdas for this to see some performance optimizations.



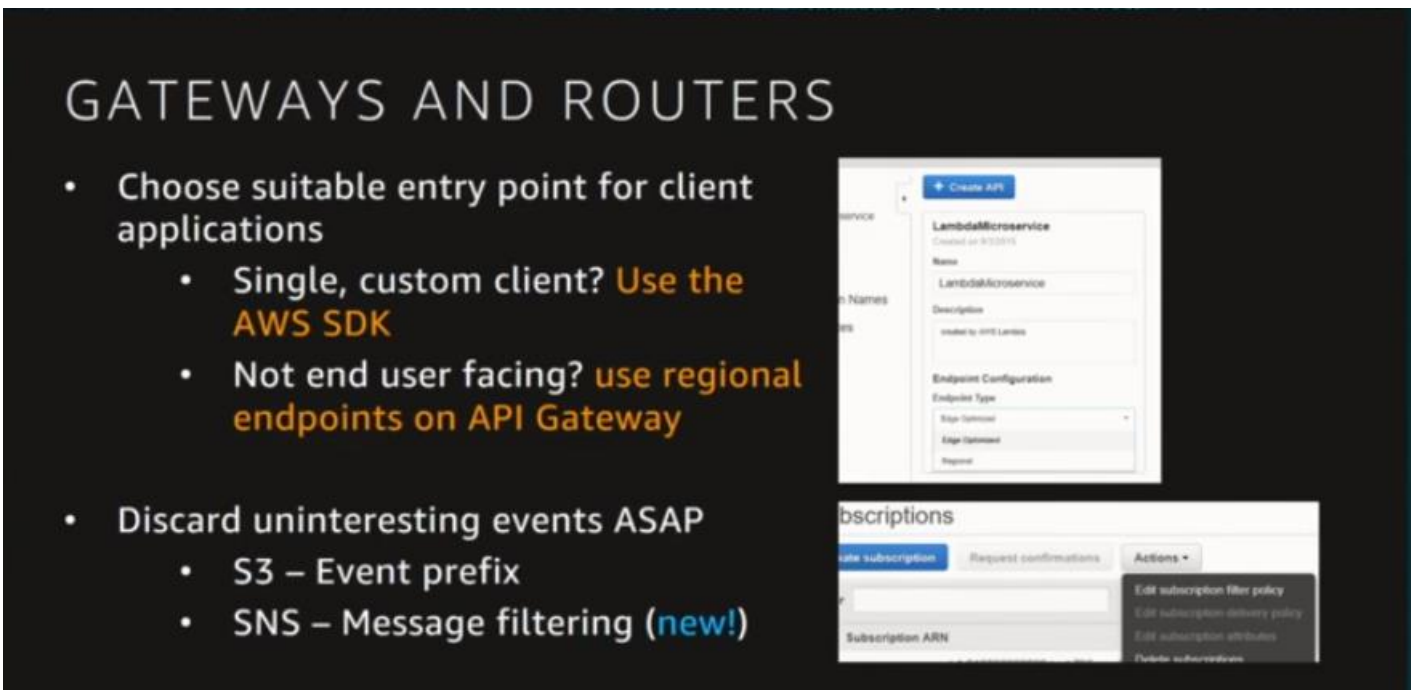


You can just split it up into 2 lambdas, one for preprocessing and then another for the post-processing functions. You can then use a step function or some external mechanism to sequence between the 2 lambdas.

This talks about the actual invocation layer for stuff coming in, how we choose the invocation mechanism makes a big difference.

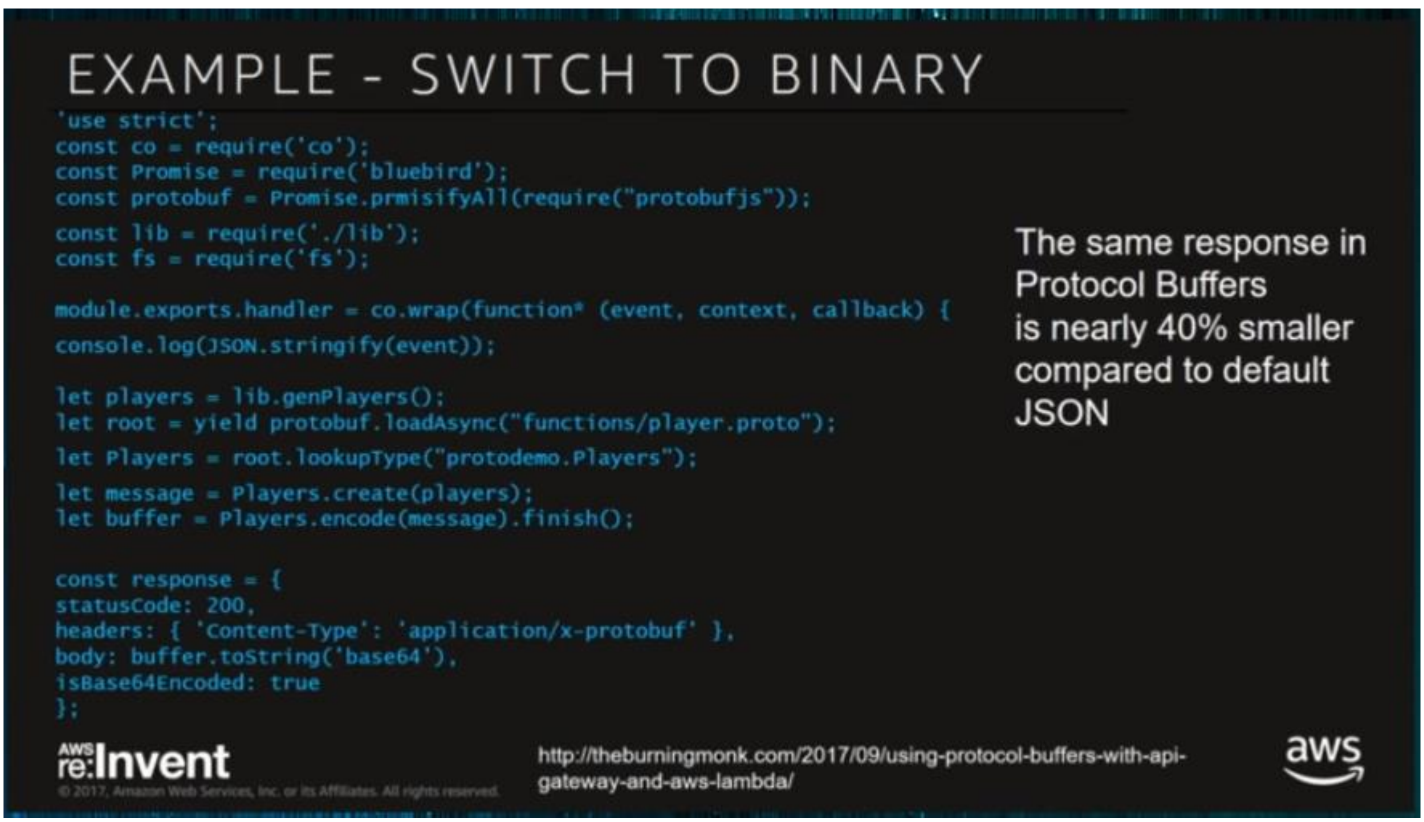


When you are using AWS services to invoke the services, AWS controls the invocation paths and the optimizations in play like building in retries, compacting the payload, efficient mechanisms between the wire.



For use cases where you control the payload like generating data from an IoT device and funneling it in through kinesis, generating payloads that are coming in via the API Gateway. In this case, you need to make choices about how you are going to do the routing and staging invocations to your lambda functions. ***Lambda exposes a direct invoke API that is an HTTP endpoint***, ***you can send a signed request to it and get a response back immediately***. Most services taking data from SNS queues offer a way for you to discard unneeded events data before they get to your lambdas, this will save unneeded processing times and cost. S3 as a way for you to do prefix and suffix filters, this allows you a way to lock down lambda functions to trigger only on updates to certain paths inside your S3 buckets. SNS recently launched a capability for you to do message property based filtering, for example in your SNS payload if you have a property that said ‘this is coming from originator app A’, you can say only process the payload if it is from originator app B.

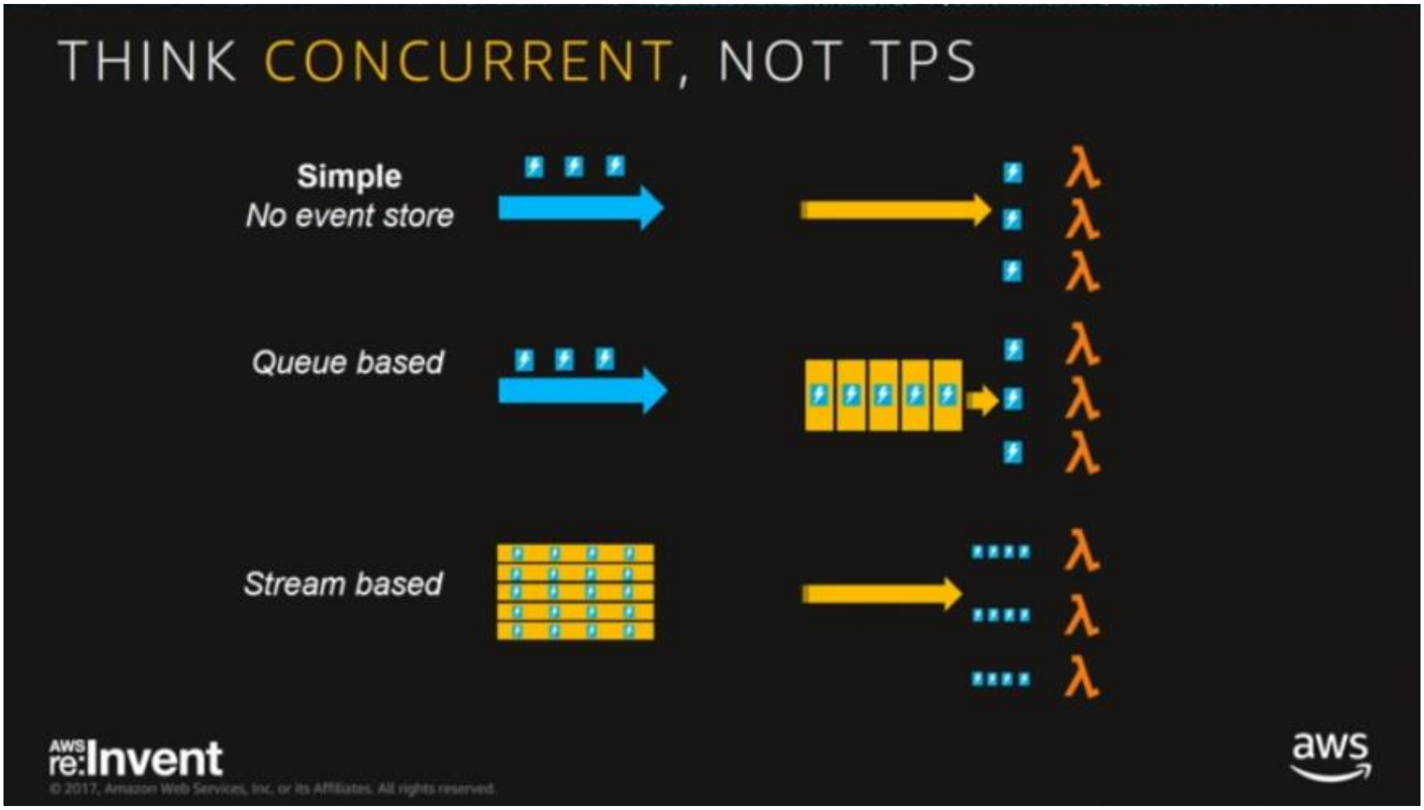
If you use a custom event and you control it, there is a bare minimum that should be in the event itself – this is what is called ***Provenance***. You should be able to say where the event came from, who the identifier is, what timestamp was involved during the event. Everything else is optional, you should avoid too many payloads coming into your function. If you are an image processing system, put the image into an S3 bucket first, then you can have your lambda go and read the image from that bucket if needed.



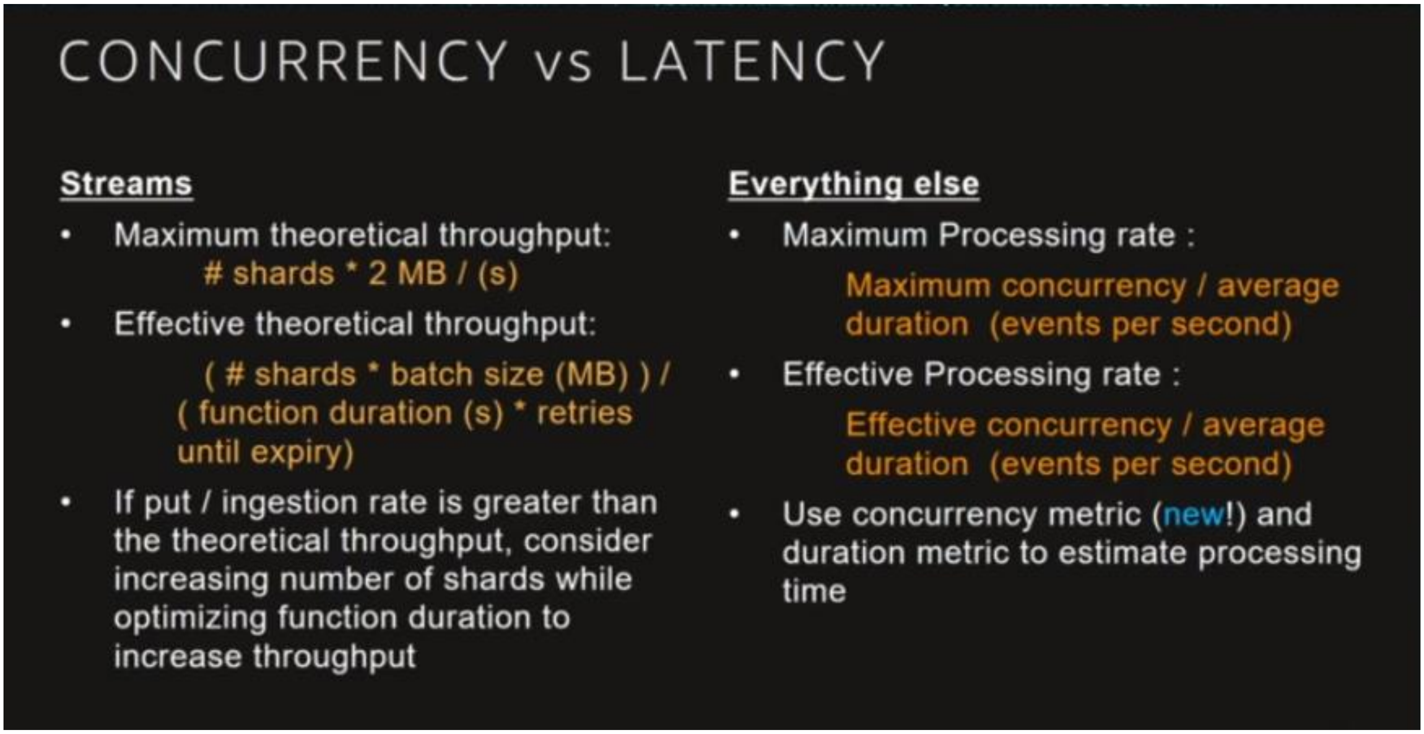
For API Gateway, you can control the protocol that gets passed on into lambda for tuning the responses back, JSON is great but there are other options available. Above is an example of swapping out the JSON responses coming into lambda for processing with Protocol Buffers. This uses the API Gateway capability of supporting binary requests and responses, you can set the response type to binary and change the header to protoBuf and just pass any service that can emit protocol buffers out there to consume the protobuf response.

There is a construct of the Event Store which should or not play a part in your architecture depending on your choice. For many applications, we look at lambda comes in or a request comes in through a router like IoT or API Gateway, turns around and invokes your lambda function. But you may have cases where you need to say ‘these events need to be stable even if the one emitting it goes down’ or ‘this needs to be persisted in some durable fashion if lambda goes down’ or ‘it needs to be retry if lambda fails’. Kinesis tends to be the favorite for many data processing applications where it is used as a bridge between your ingress point and your lambda, you use Kinesis as the streaming mechanism for turning around and processing those applications. SQS is another viable choice to replace Kinesis depending on your use case of queues or streaming based event sources.

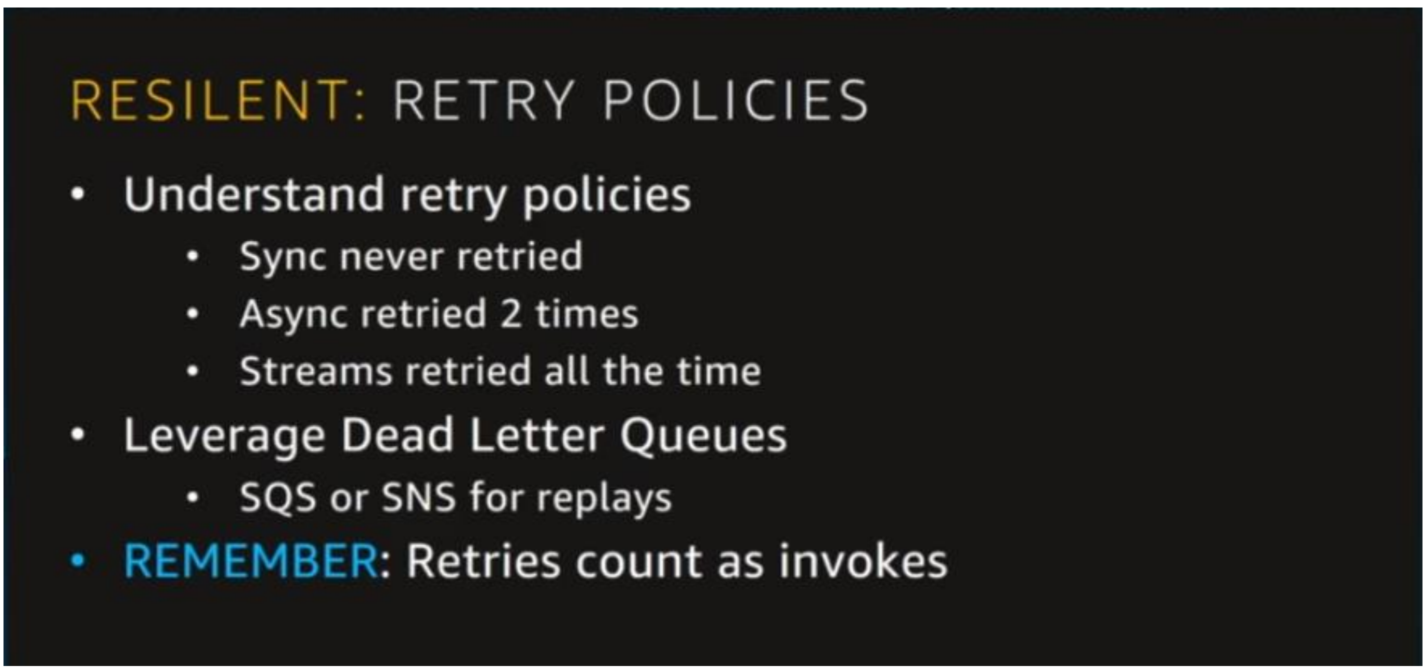
DynamoDB streams is an event store for DynamoDB updates, it gives you a persistent mechanism for seeing what the change log looks like. Lambdas Async API has a queue built in that gives you an event store end to end so that you can do retry policies under the covers.

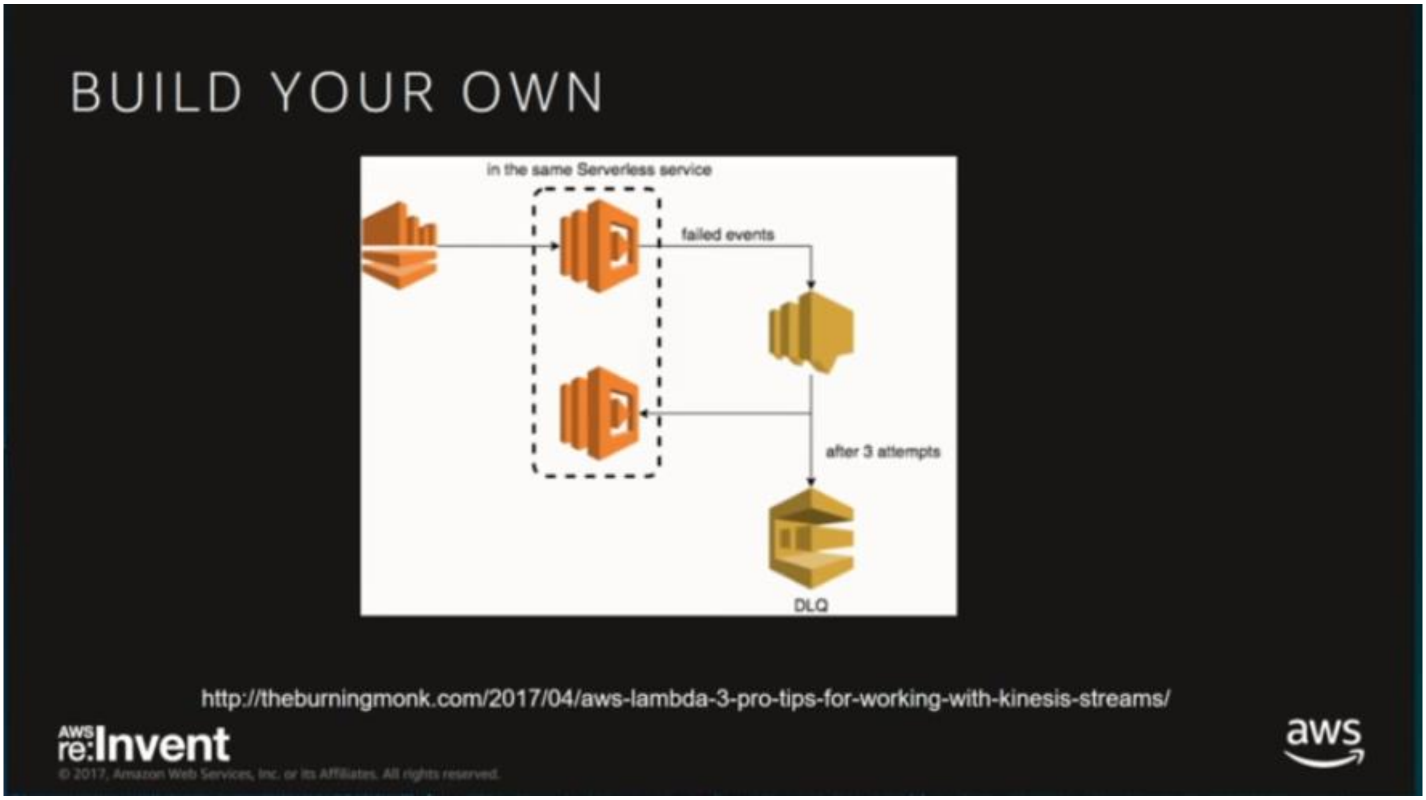


Lambda thinks of its scale in terms of concurrency, it is the unit of the request rate and depends on how you are calling your lambdas. It is a little different if it is stream based for ordering guarantees

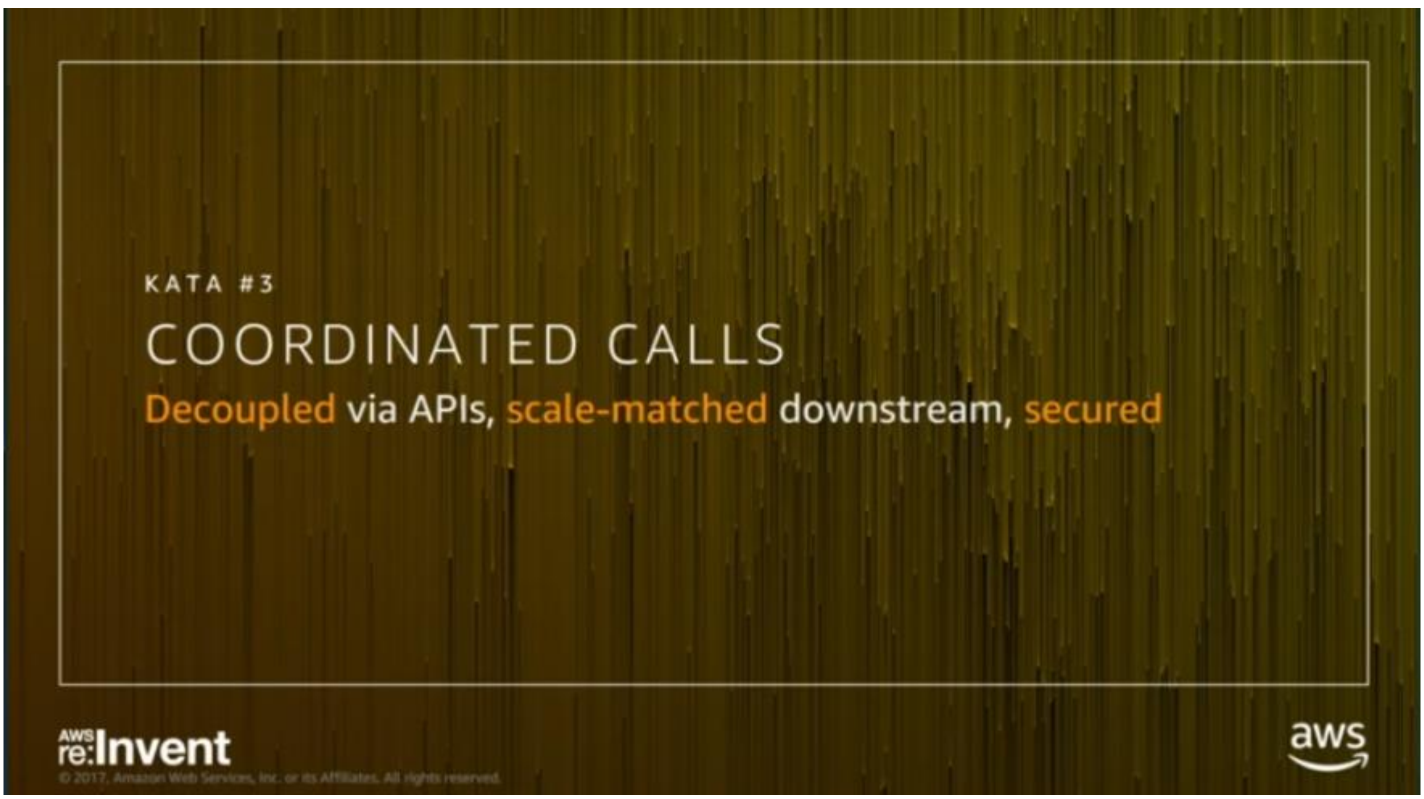


Make sure that you are looking at the new concurrency metric called ***Concurrency invocations*** in your console that tells you what is the effective concurrency available for your account level as well as your individual functions.

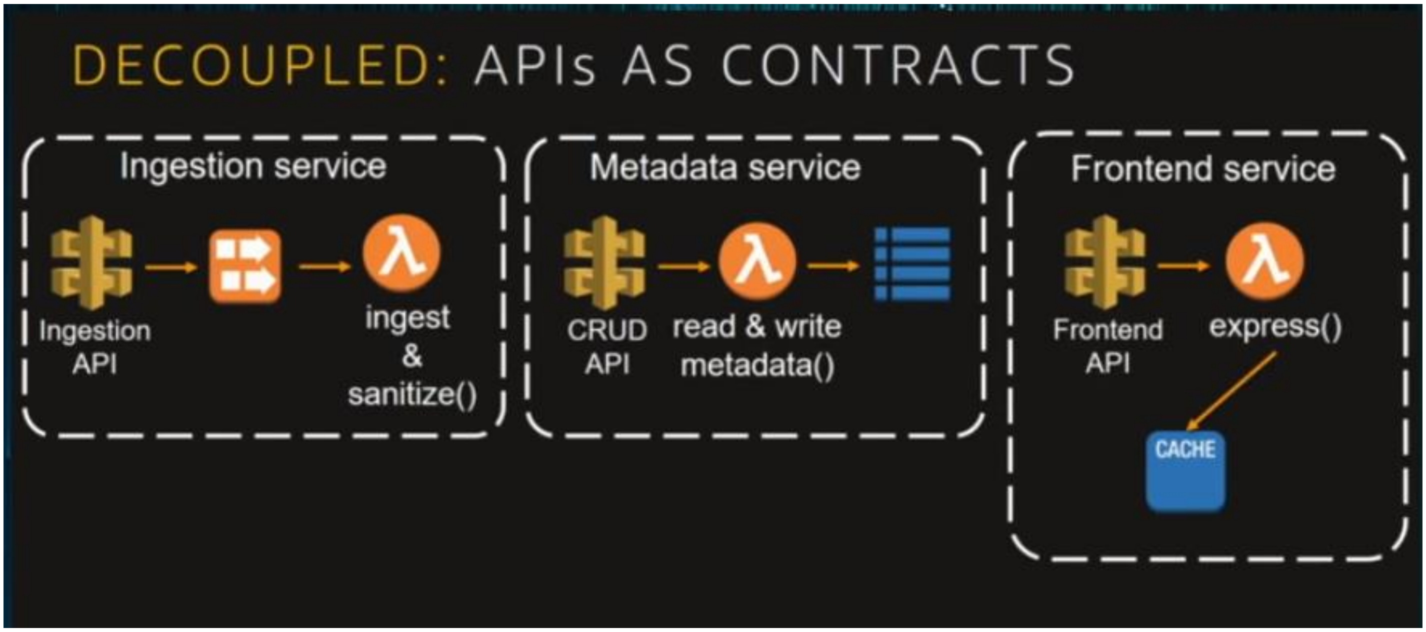




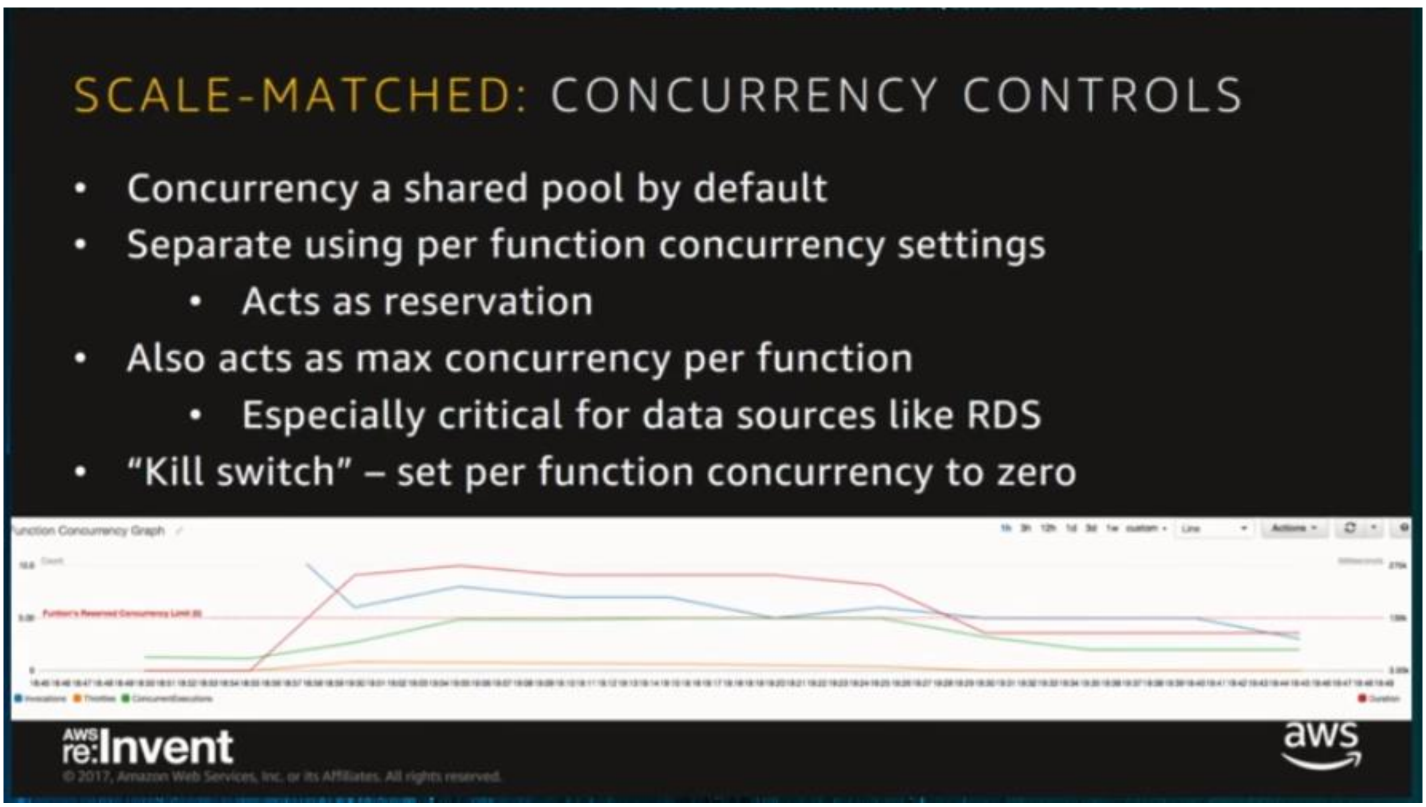
SLS has its own Dead Letter DL queue built in that is the same as the Lambda DL queue, you can route the DL queue of failed attempts from Kinesis to an SNS queue instead of sending an error to Kinesis that will trigger a retry.



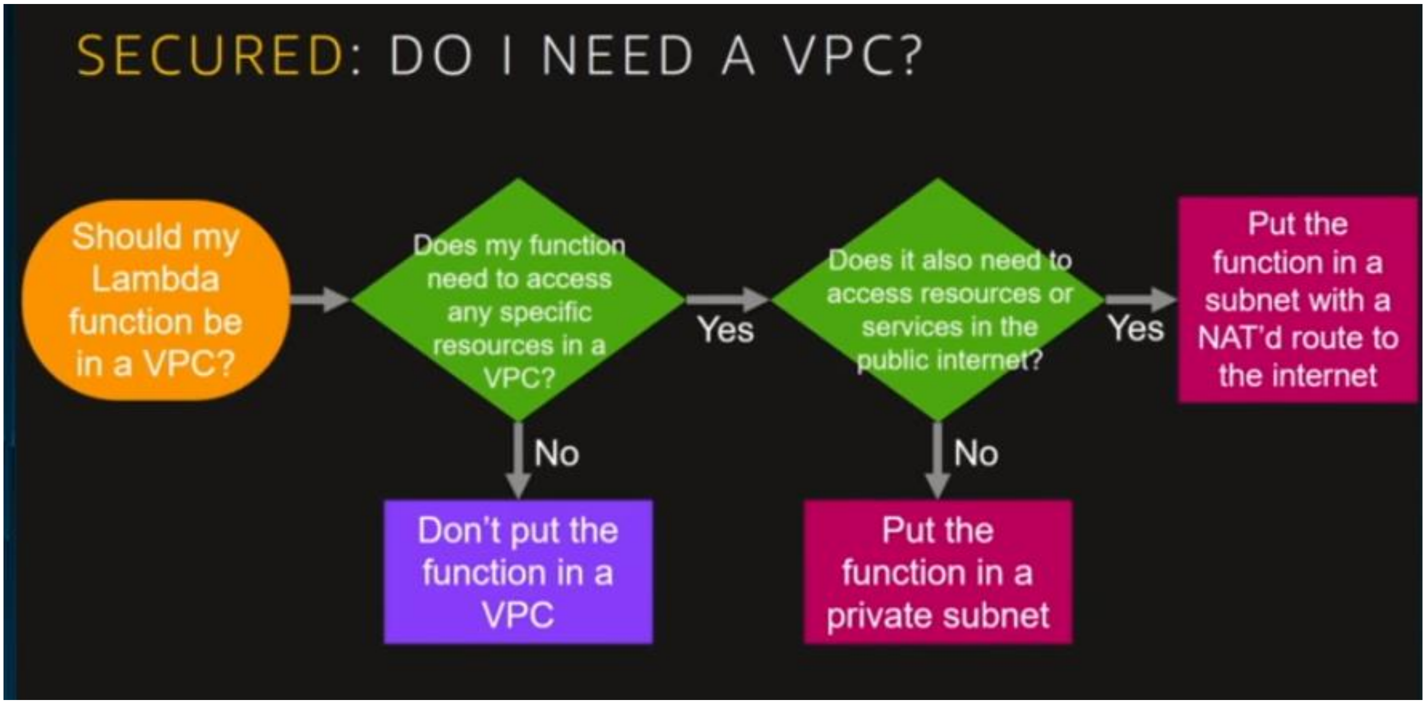
This is about how lambdas talk to downstream services.

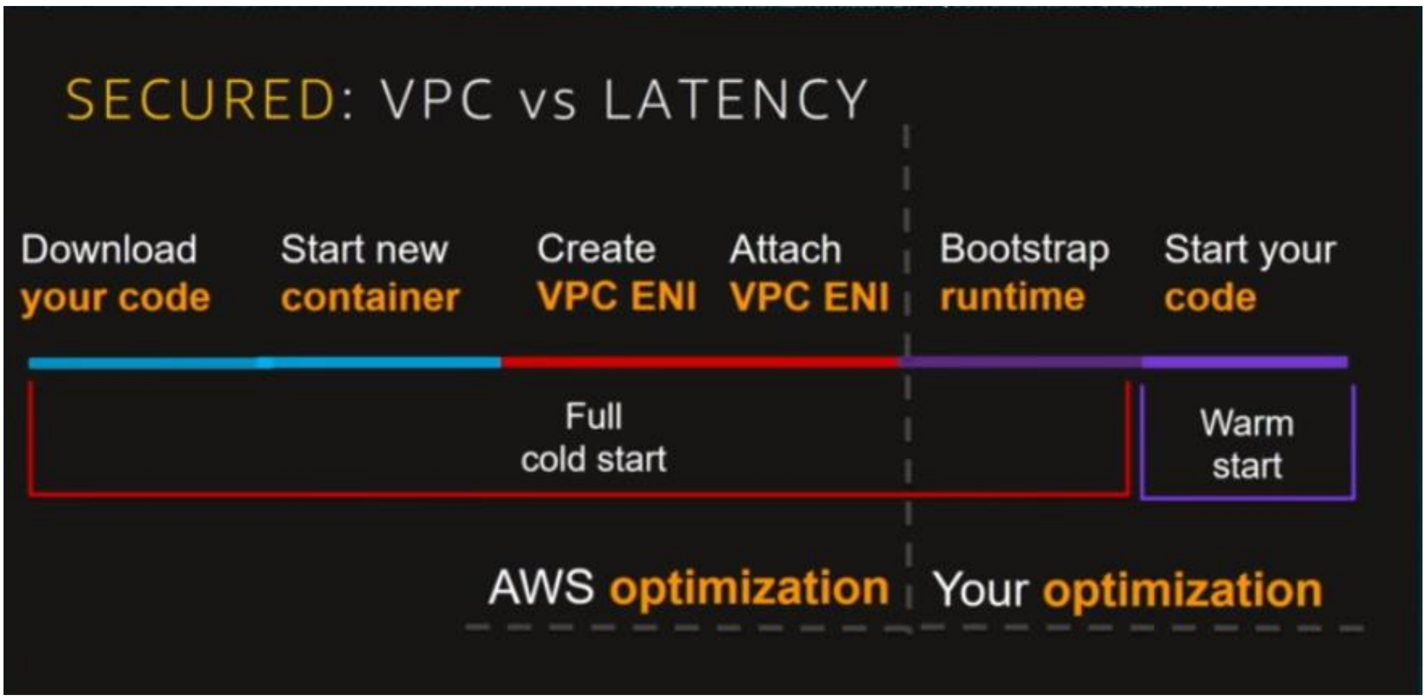


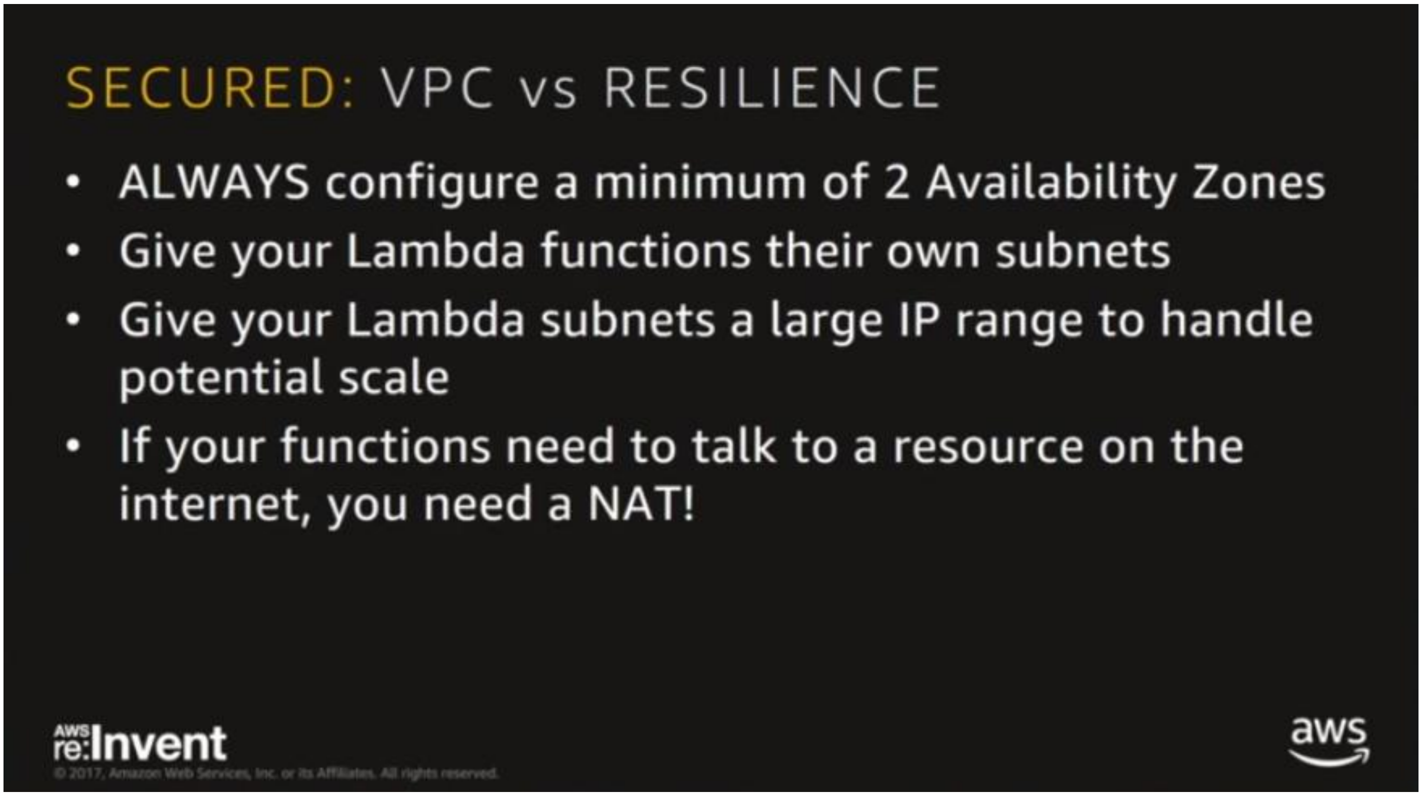
Make sure the way you set up your applications has APIs as Contracts, this is important in the lambda implementation because having APIs between components of your system allows you to have separation of concerns. In this example, we have 3 separate services, an ingestion service that gets processed by a metadata extraction service that is frontend by a frontend service with 3 separate APIs for Ingestion, CRID, and Frontend. ***A DDOS attack on the Ingestion API can be rate limited by the CRUD API to prevent both the Metadata and Frontend services***.



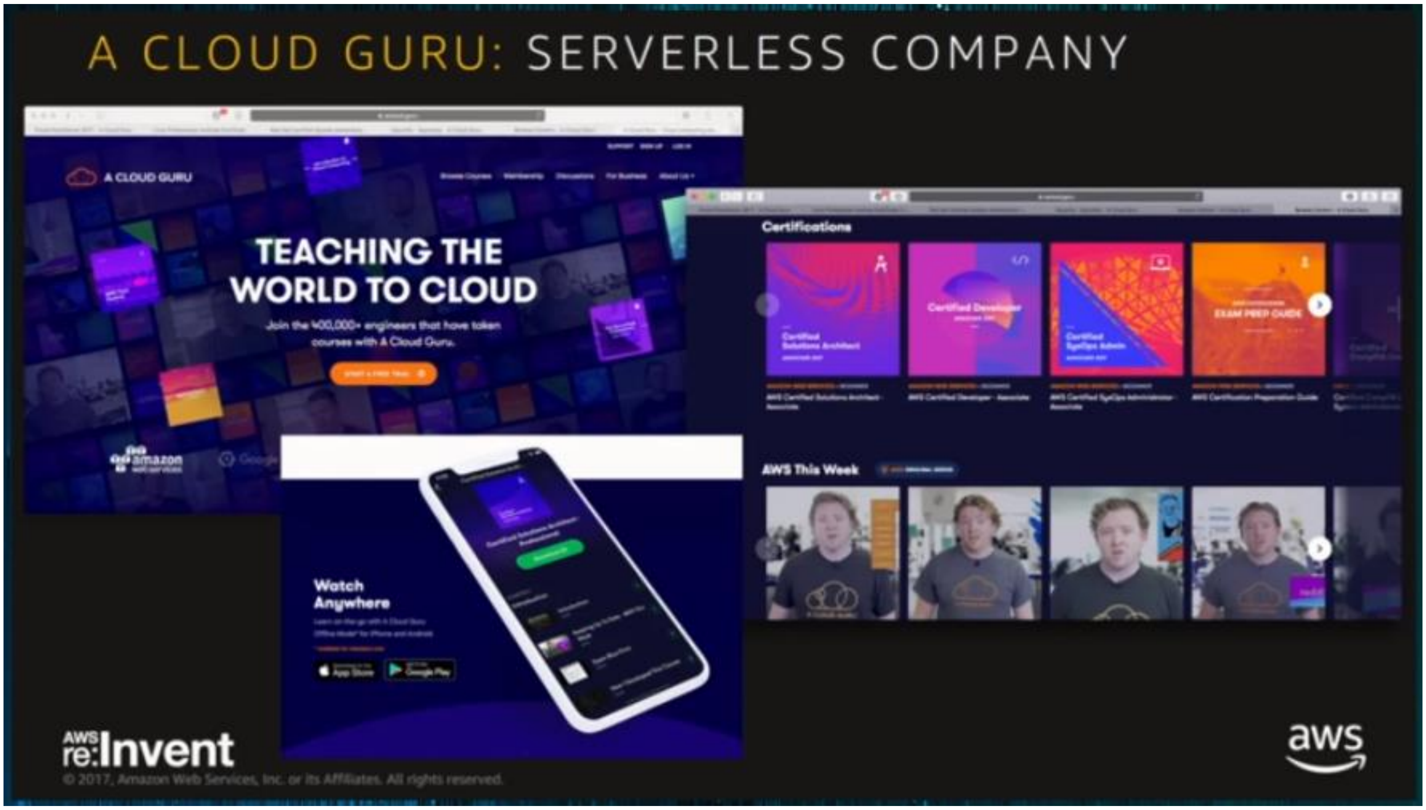
For non-API calls to databases and other service, you need to have Scale Matching for resources that have scale constraints like databases. Make sure that lambdas are calling out to these databases, you are matching the scale limitation for lambda to the end data store. You can now specify the max concurrency for individual lambda functions, this allows you to restrict an individual function from fanning out beyond a specific limit.


SECURED: DO I NEED A VPC?
Should my Lambda function be in a VPC?
Does my function need to access any specific resources in a VPC?
Yes
No
Don't put the function in a VPC
Does it also need to access resources or services in the public internet?
Yes
No
Put the function in a private subnet
Put the function in a subnet with a NAT'd route to the internet

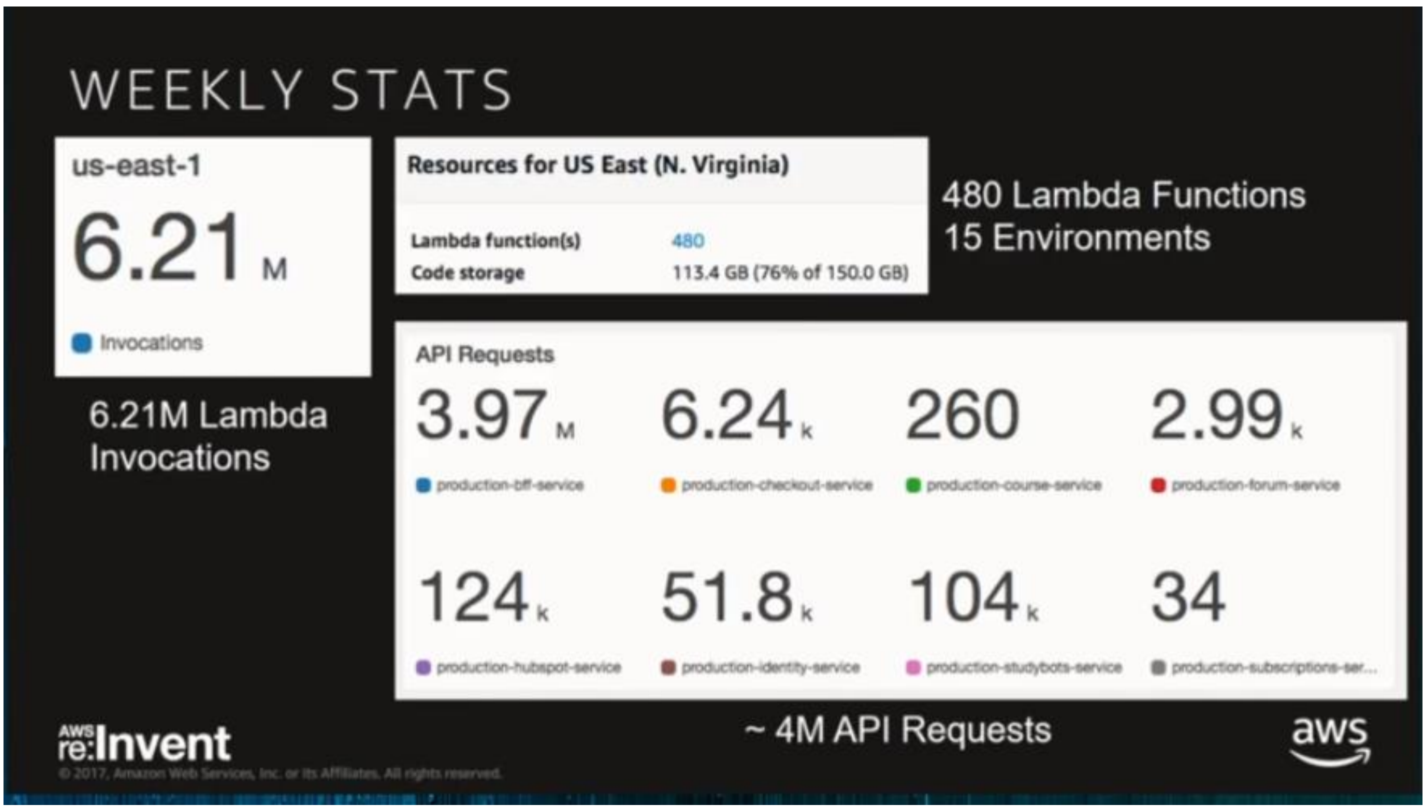

SECURED: VPC vs LATENCY
Download your code
Start new container
Create VPC ENI
Attach VPC ENI
Bootstrap runtime
Start your code
Full cold start
Warm start
AWS optimization
Your optimization



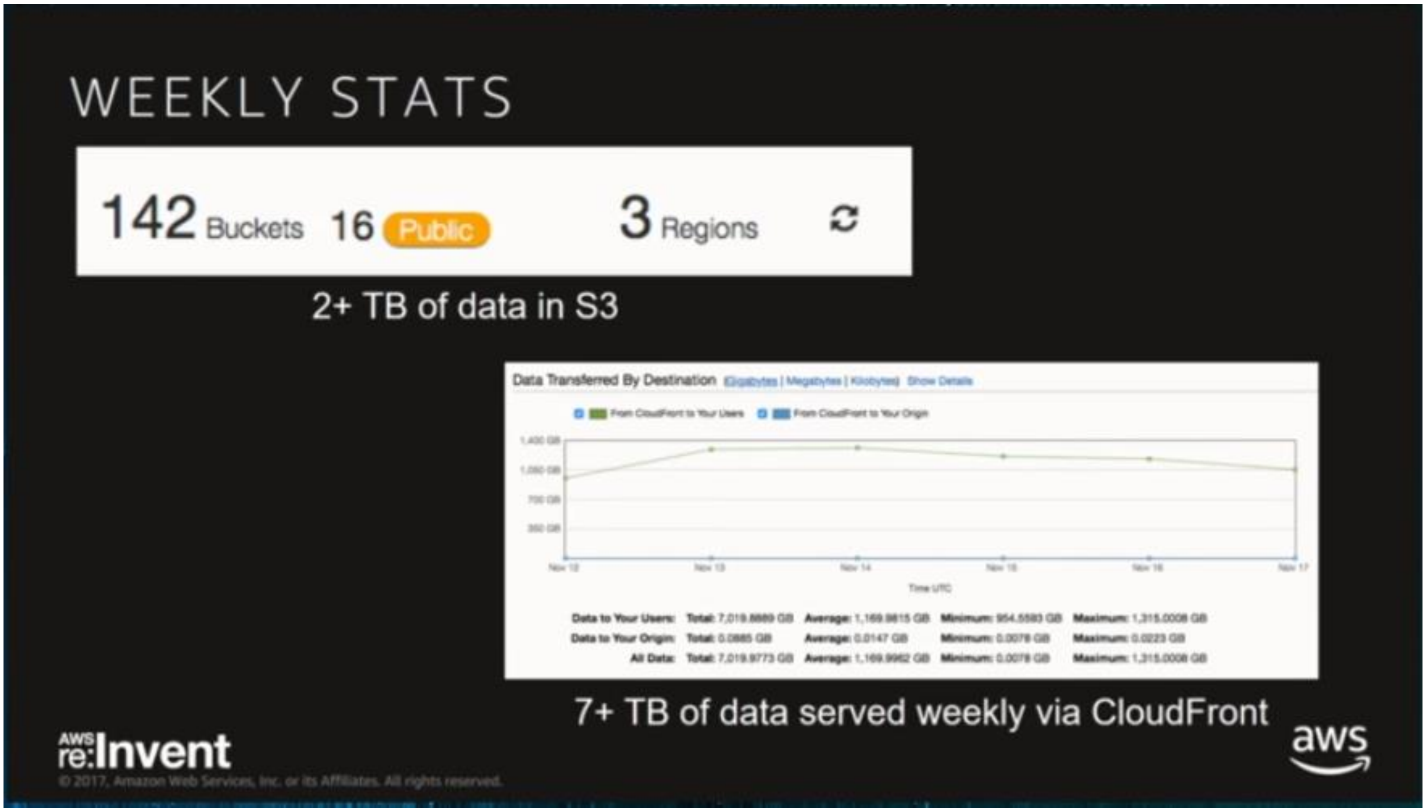
Let us see how to put a lot of these recommendations into practice at A Cloud Guru



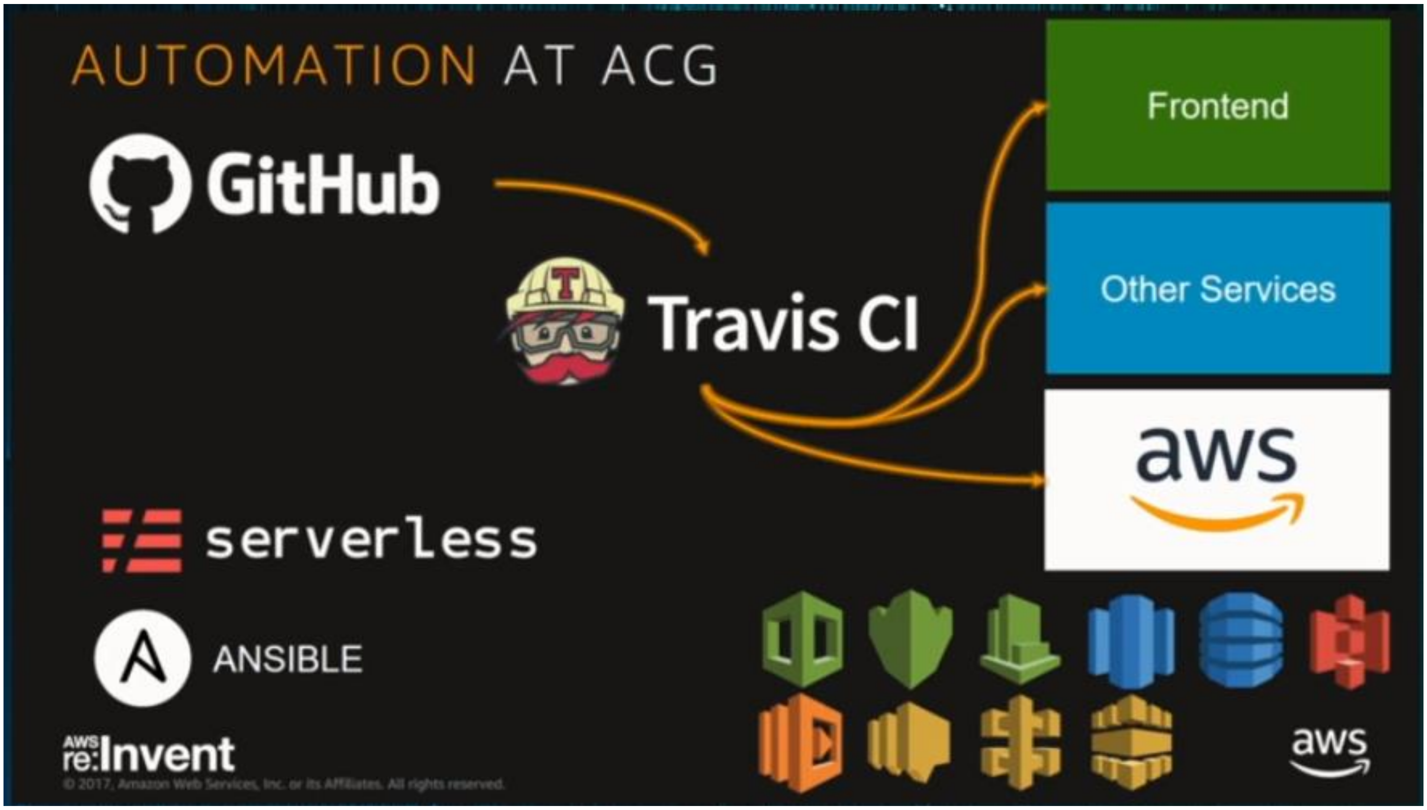
Started 2 years ago and built an entirely Serverless company using lambdas. We also have rich web apps and mobile apps without any servers



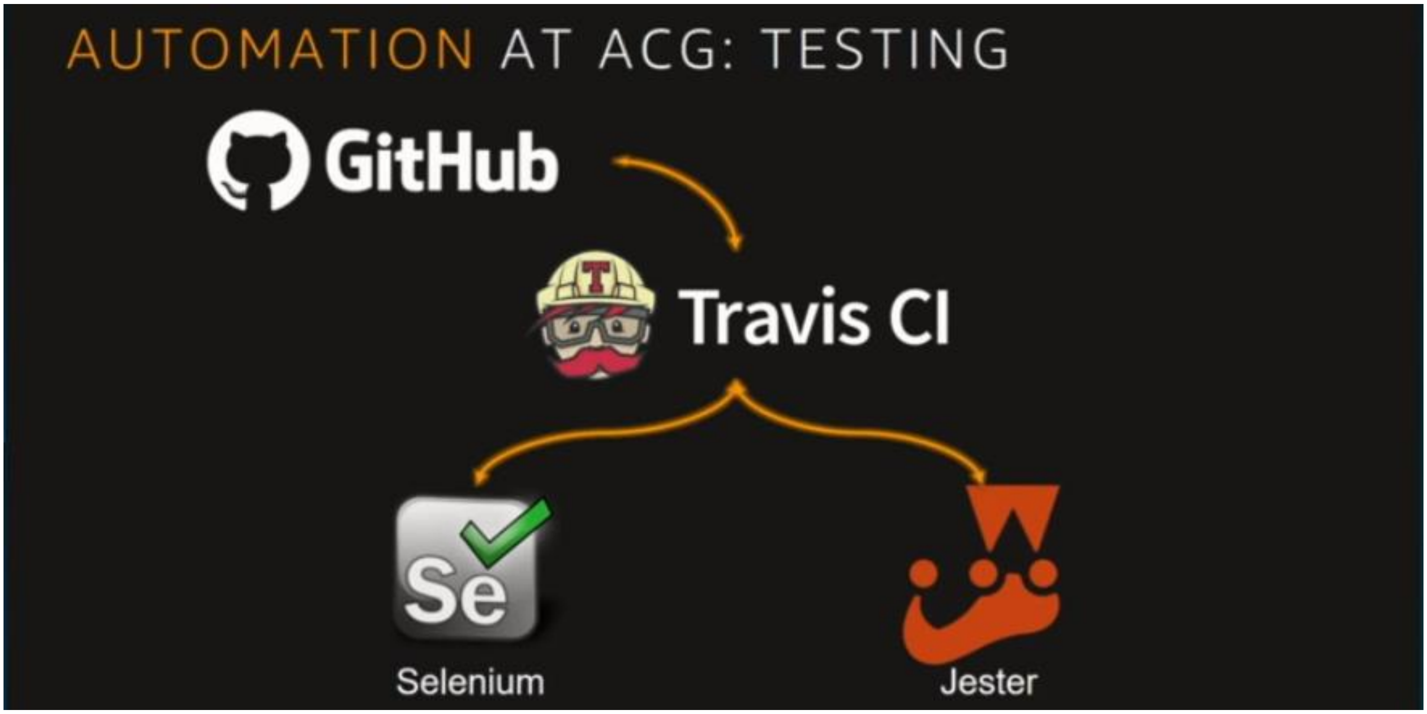
There are ***3.97 million API Requests*** going through the ***production-bff-service*** service which is the GrapQL endpoint. There is a new ***AWS AppSync GraphQL managed service*** that AWS just released



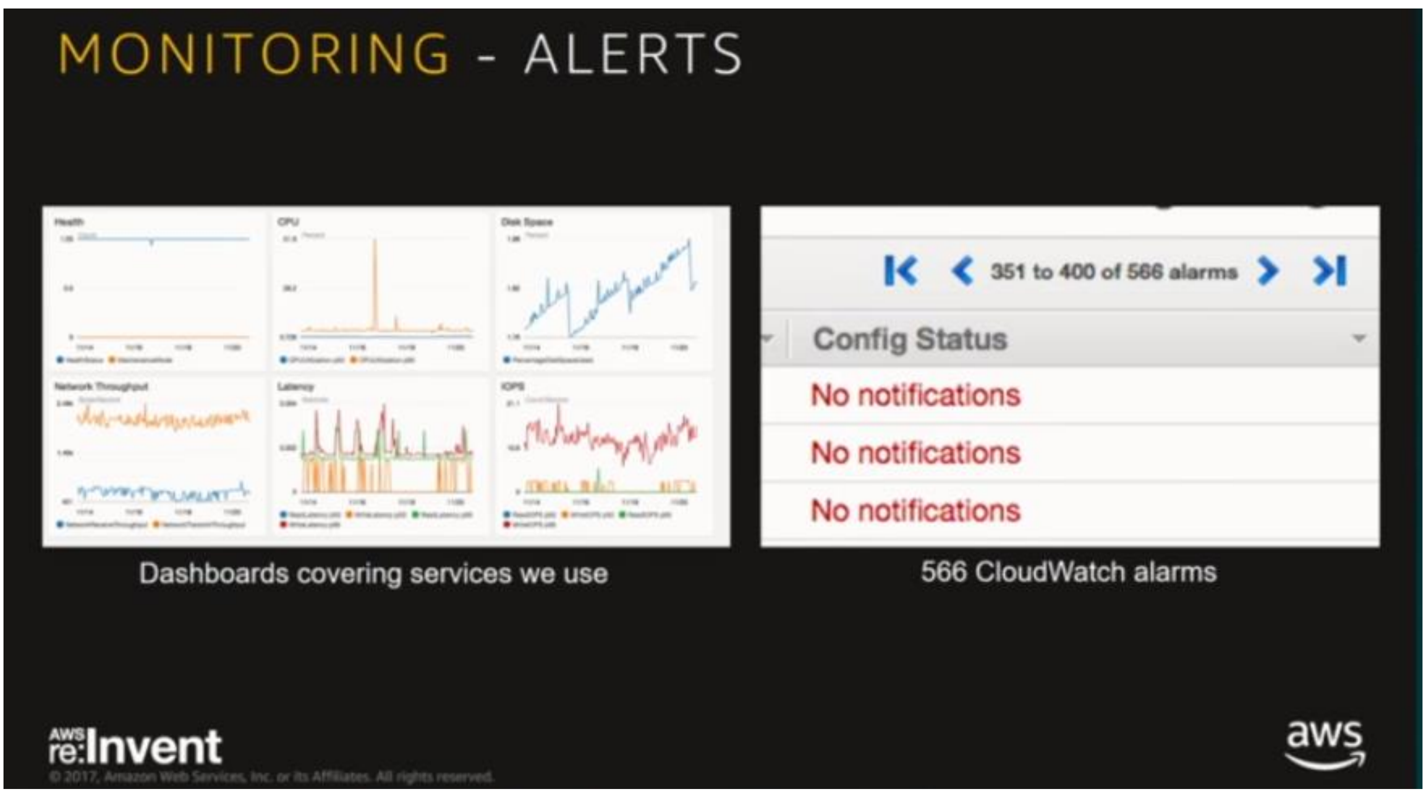
This means we need to optimize the functions in our applications to reduce the overall cost



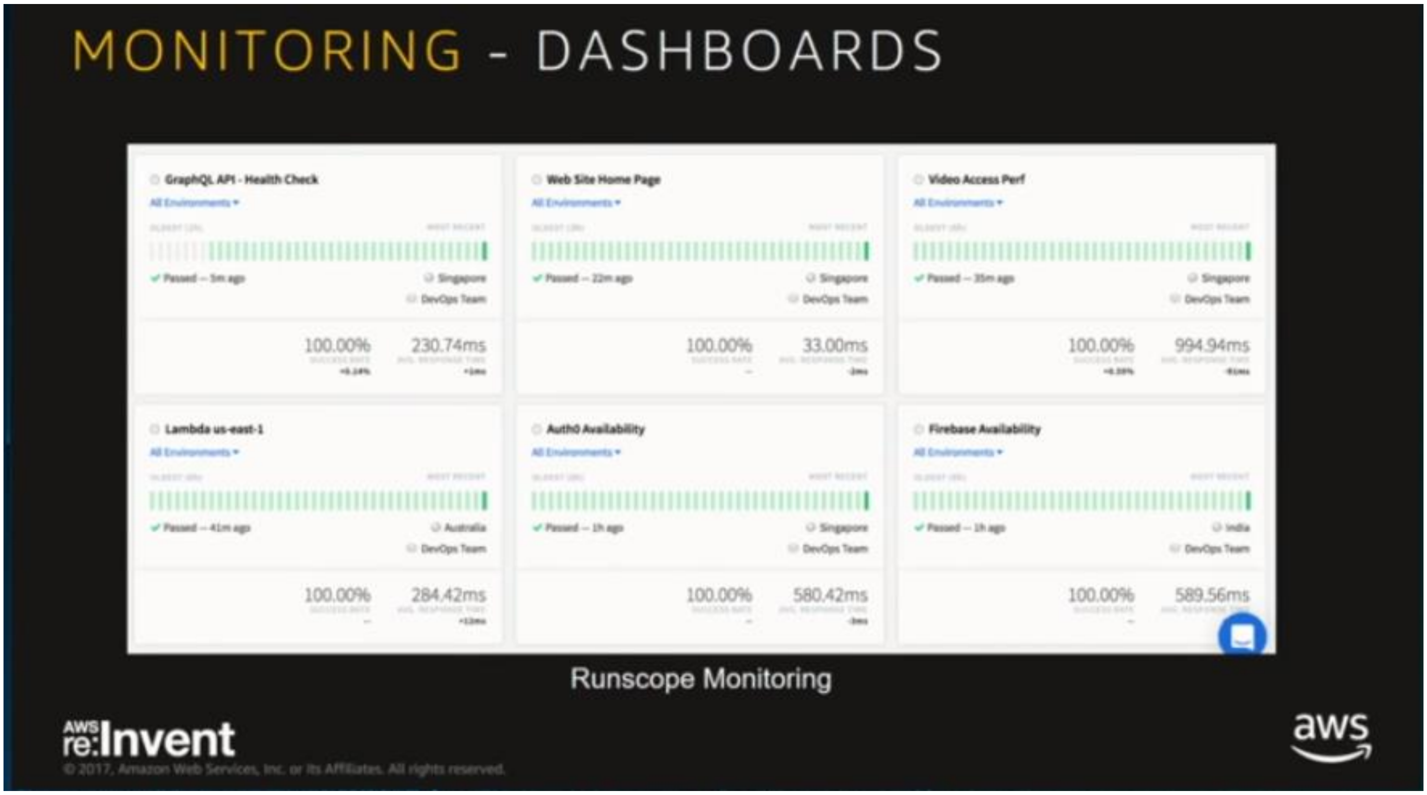
Serverless applications are distributed by nature, so we have several lambda functions, our SPA, various services to updates, we need to make changes to Firebase which is our real-time streaming database. We do this all via automation and testing. We also use the ***Serverless Framework*** for ***orchestration*** and ***deployment***, and use Ansible for our Dev environments so that we can deploy some additional CloudFormation scripts. ***Don't go serverless without automating the provision and deployments of your applications first, you need to setup your deployment pipeline before starting to work on your applications***.



For testing, we use Selenium, Jester and Travis CI automates the testing process for us.

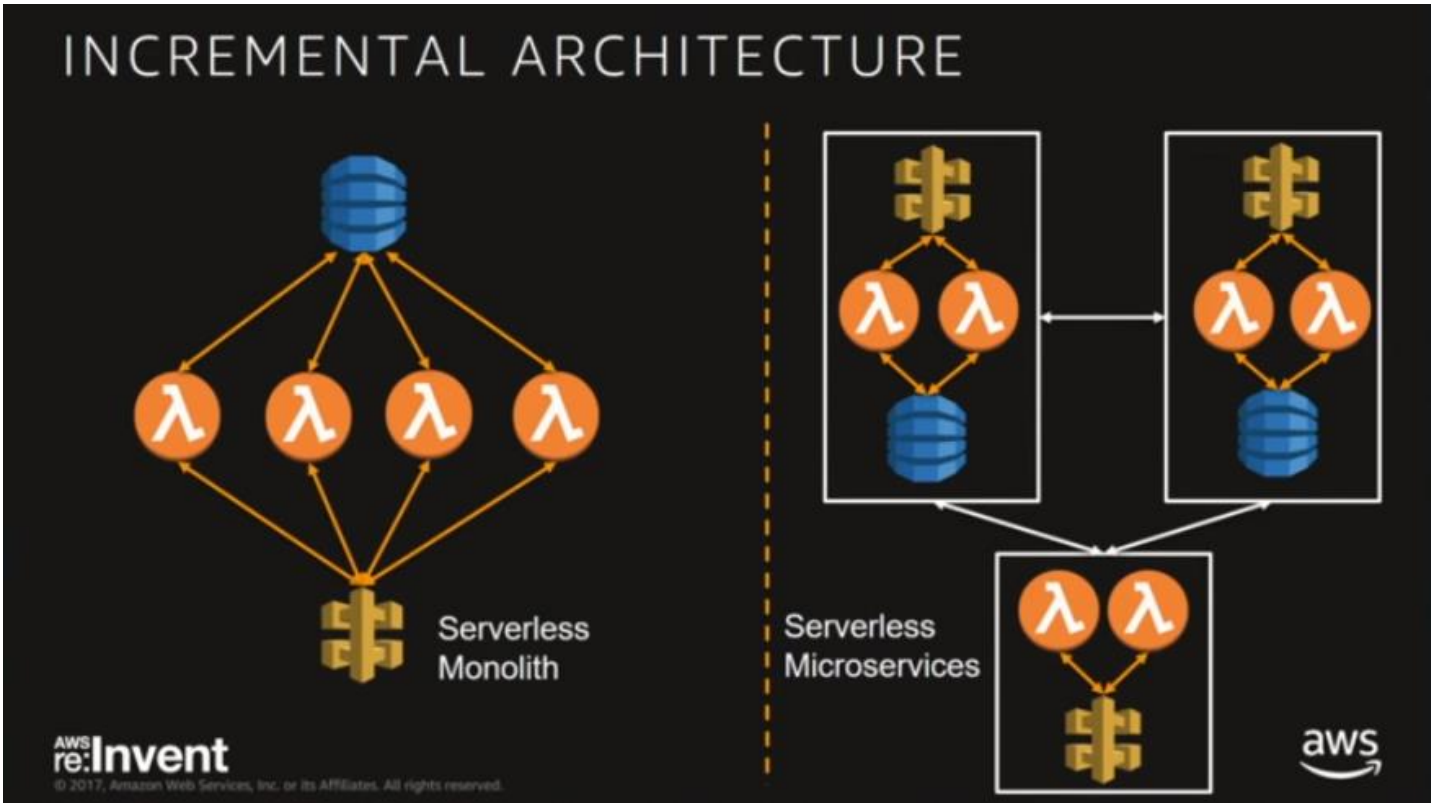


You need to be aware of what is going on in your system, use alerts while also calibrating the signal to noise ratio. You probably should have an alarm for every lambda function you create

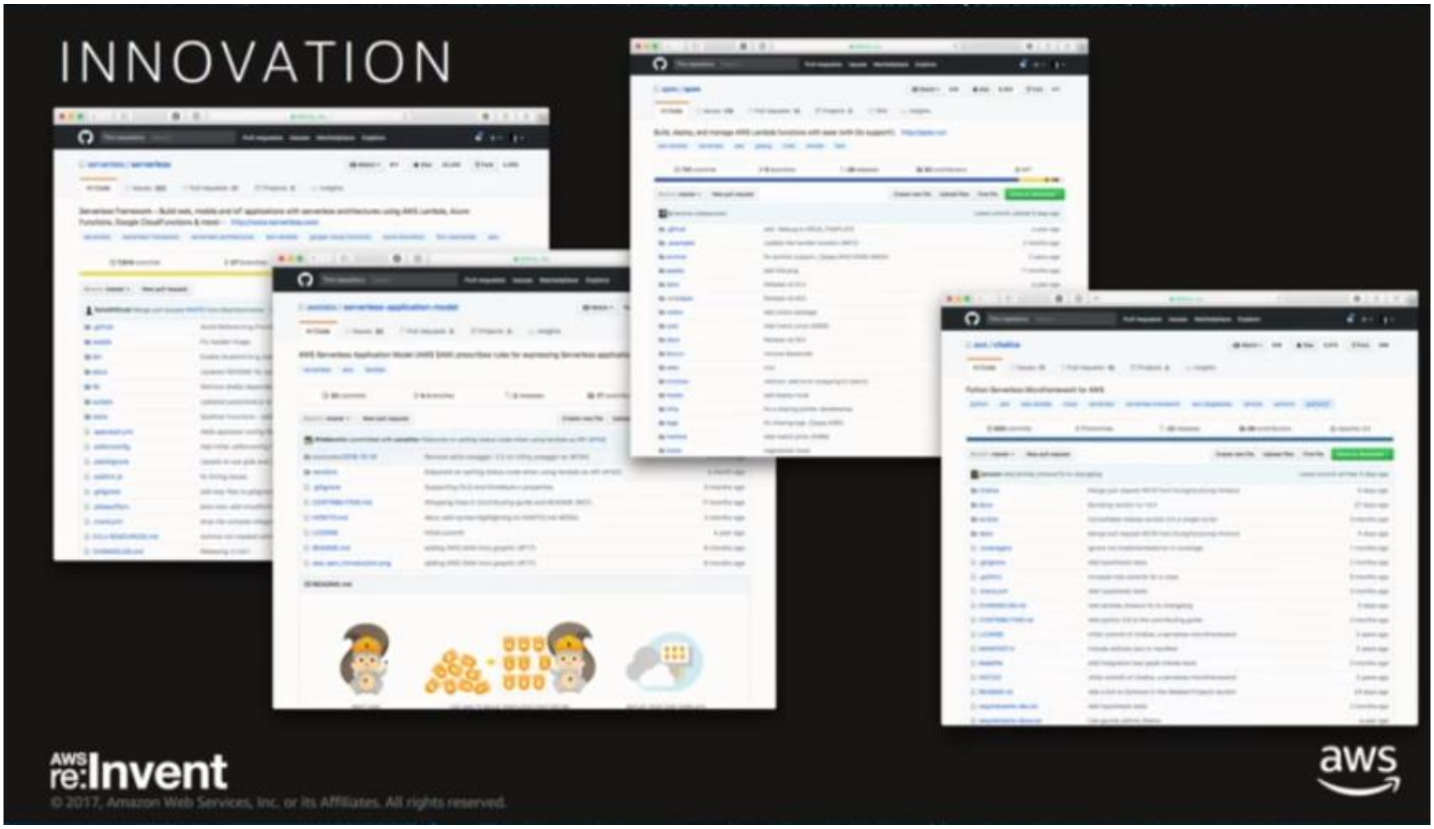


The ***Runscope tool*** helps us monitor the responsiveness and latency within our system. It continuously makes requests to ***CloudFront*** to see whether we can access our files for all our applications like the GraphQL API, Website, Auth) service, Firebase service, etc.

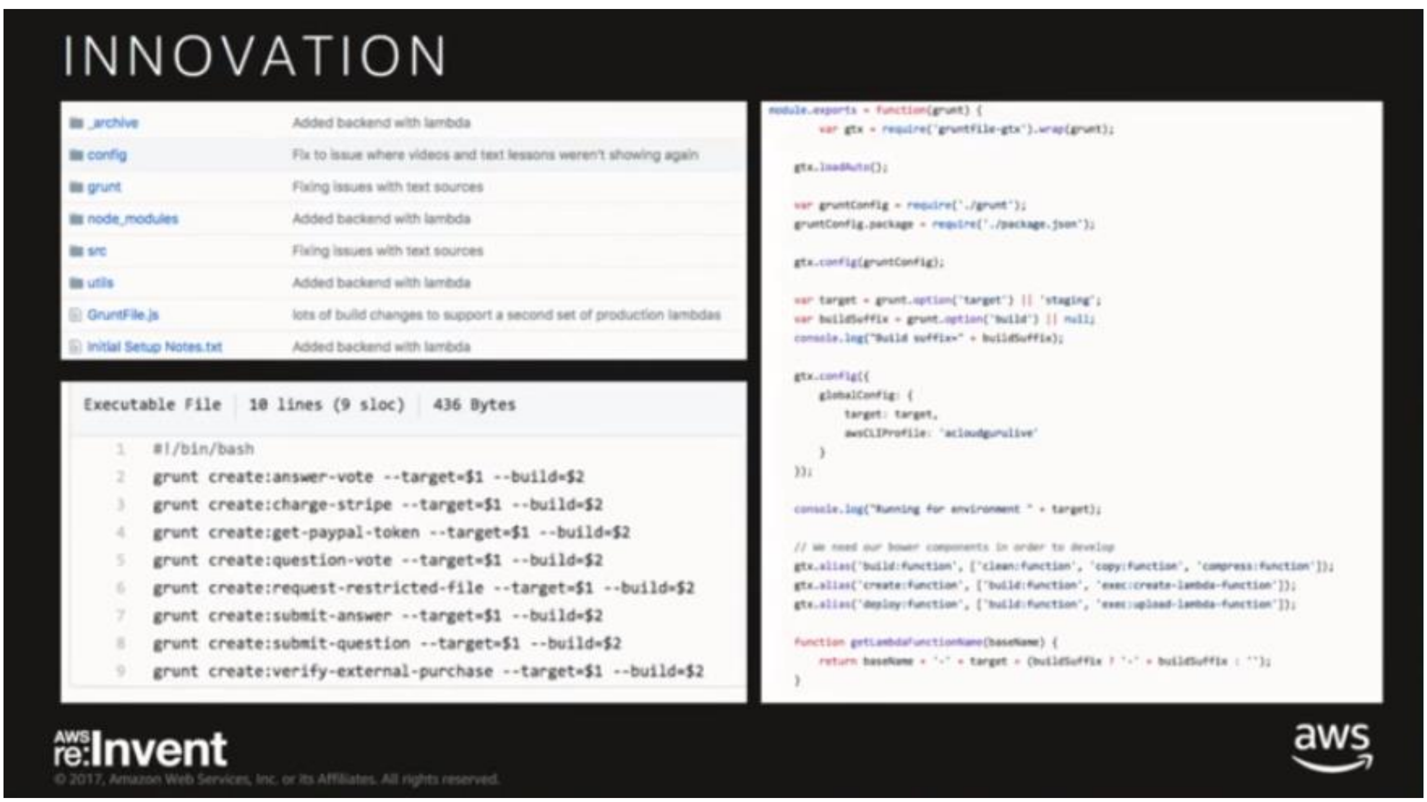
Security in serverless is a joint responsibility between you and AWS. Make sure that your IAM roles, users, and policies are as secured as possible. Have a Role per lambda function. Use things like AWS KMS where possible.



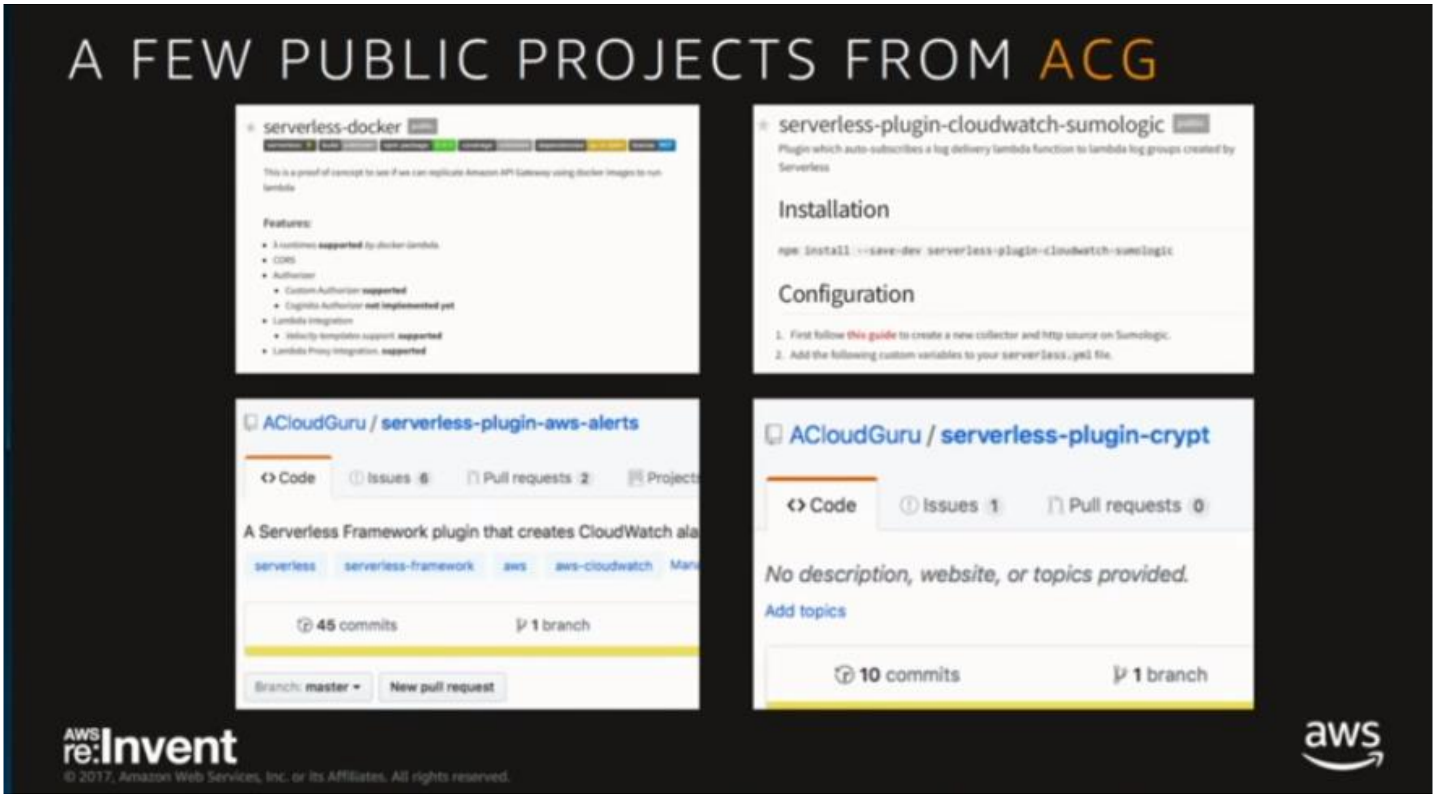
Apply the tips and tips about your architecture incrementally, you can then scale out and transition from a serverless monolith to a ***serverless microservices*** approach just by changing some of the code and changing the ***serverless.yml*** file without provisioning any server at all.



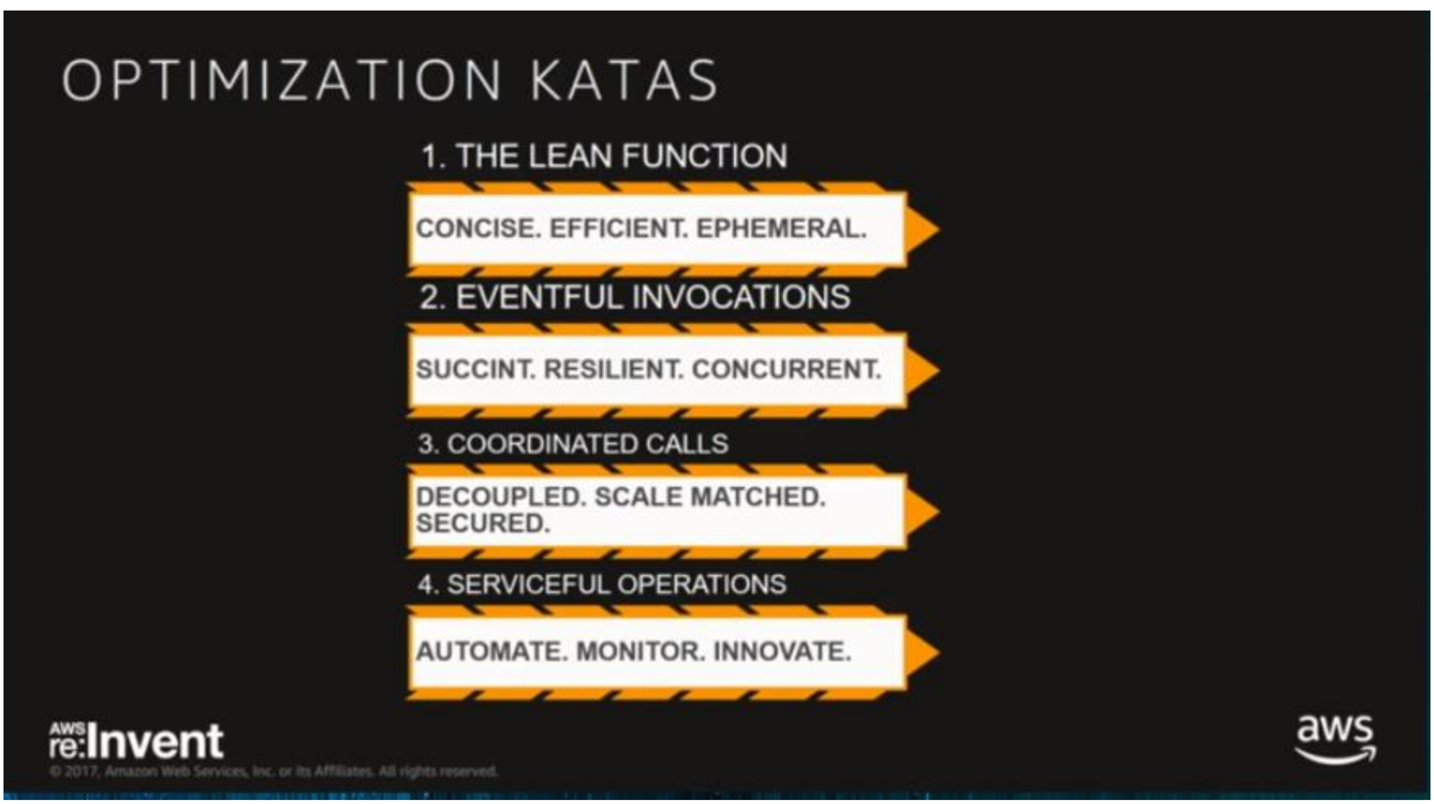
As you evolve, best practices will change



We used to roll out our own deployment scripts using Grunt in the past but this is not needed anymore once we migrated to the Serverless Framework


A FEW PUBLIC PROJECTS FROM ACG
serverless-docker
serverless-plugin-cloudwatch-sumologic
Installation
Configuration
ACloudGuru / serverless-plugin-aws-alerts
<> Code
Issues 6
Pull requests 2
A Serverless Framework plugin that creates CloudWatch ala
serverless
serverless-framework
aws
aws-cloudwatch
45 commits
1 branch
Branch: master
New pull request
ACloudGuru / serverless-plugin-crypt
<> Code
Issues 1
Pull requests 0
No description, website, or topics provided.
Add topics
10 commits
1 branch
AWS re:Invent
© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.
aws


Serverless is a mindset change toward automation, agility, and innovation.


OPTIMIZATION KATAS
1. THE LEAN FUNCTION
CONCISE. EFFICIENT. EPHEMERAL.
2. EVENTFUL INVOCATIONS
SUCCINT. RESILIENT. CONCURRENT.
3. COORDINATED CALLS
DECOUPLED. SCALE MATCHED. SECURED.
4. SERVICEFUL OPERATIONS
AUTOMATE. MONITOR. INNOVATE.
AWS re:Invent
© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.
aws

MORE OPTIMIZATION SESSIONS

SRV303 - Monitoring and Troubleshooting in a Serverless World

SRV322-R2 - Migration to Serverless: Design Patterns and Best Practices

SRV311 - Authoring and Deploying Serverless Applications with AWS SAM

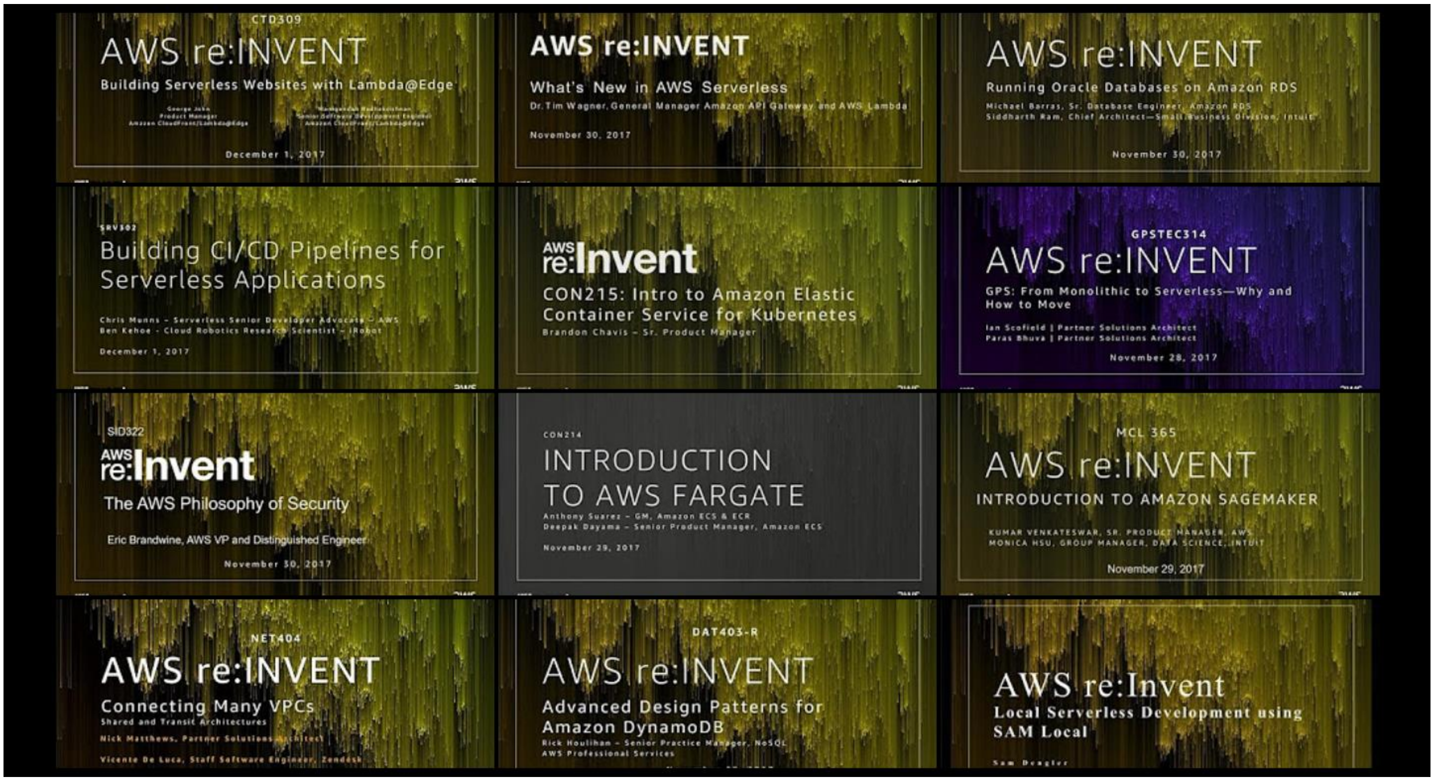
SRV320-R - Best Practices for Using AWS Lambda with RDS/RDBMS Solutions

AWS re:Invent

AWS re:Invent

THANK YOU!

AWS re:Invent


CTD309
AWS re:INVENT
Building Serverless Websites with Lambda@Edge
December 1, 2017
AWS re:INVENT
What's New in AWS Serverless
November 30, 2017
AWS re:INVENT
Running Oracle Databases on Amazon RDS
November 30, 2017
SRV302
Building CI/CD Pipelines for Serverless Applications
December 1, 2017
AWS re:Invent
CON215: Intro to Amazon Elastic Container Service for Kubernetes
Brandon Chavis – Sr. Product Manager
GPSTEC314
AWS re:INVENT
GPS: From Monolithic to Serverless—Why and How to Move
November 28, 2017
SID322
AWS re:Invent
The AWS Philosophy of Security
Eric Brandwine, AWS VP and Distinguished Engineer
November 30, 2017
CON214
INTRODUCTION TO AWS FARGATE
November 29, 2017
MCL 365
AWS re:INVENT
INTRODUCTION TO AMAZON SAGEMAKER
November 29, 2017
NET404
AWS re:INVENT
Connecting Many VPCs
Shared and Transit Architectures
DAT403-R
AWS re:INVENT
Advanced Design Patterns for Amazon DynamoDB
AWS re:Invent
Local Serverless Development using SAM Local