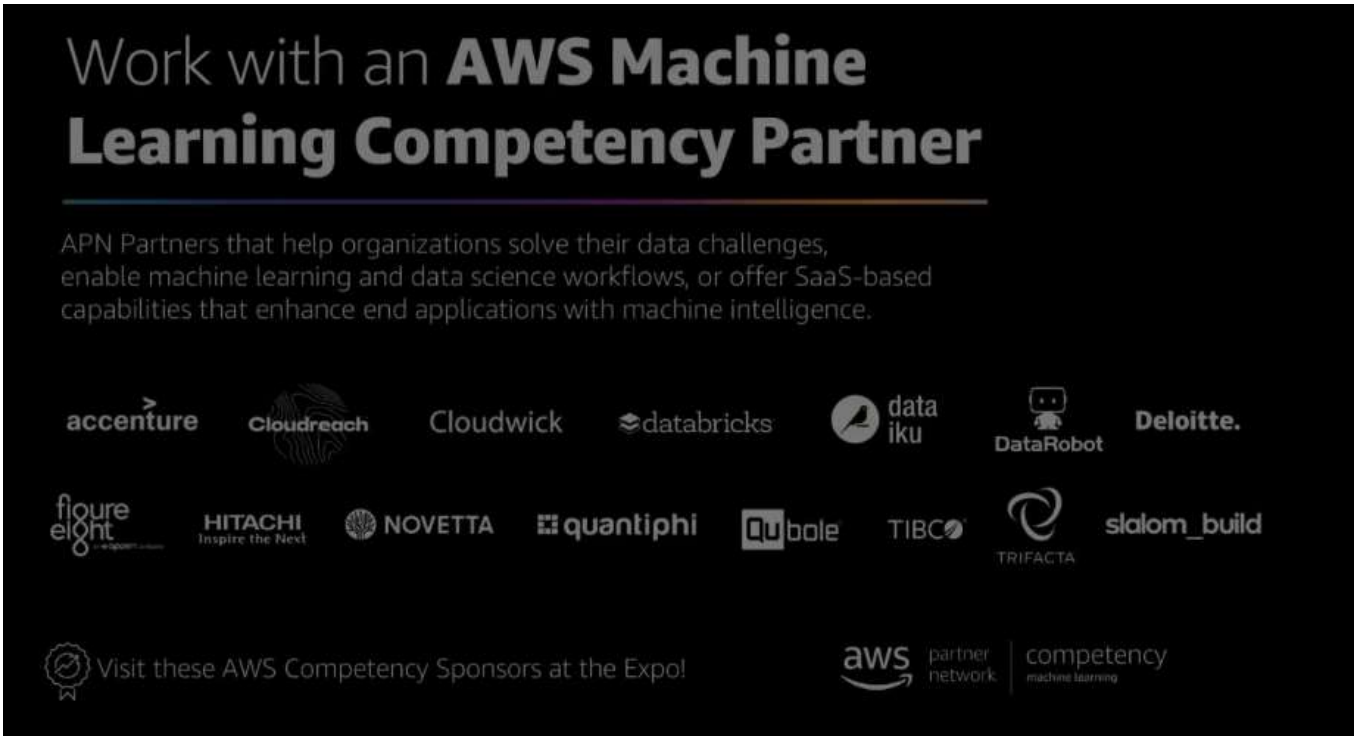
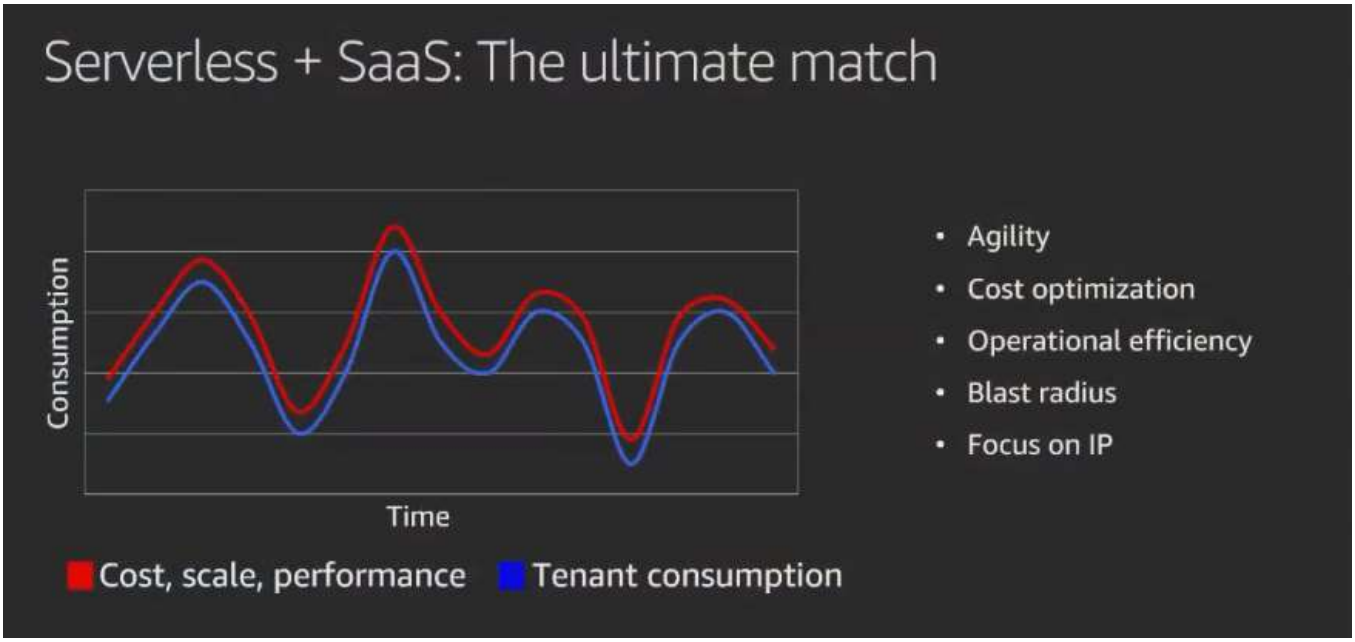
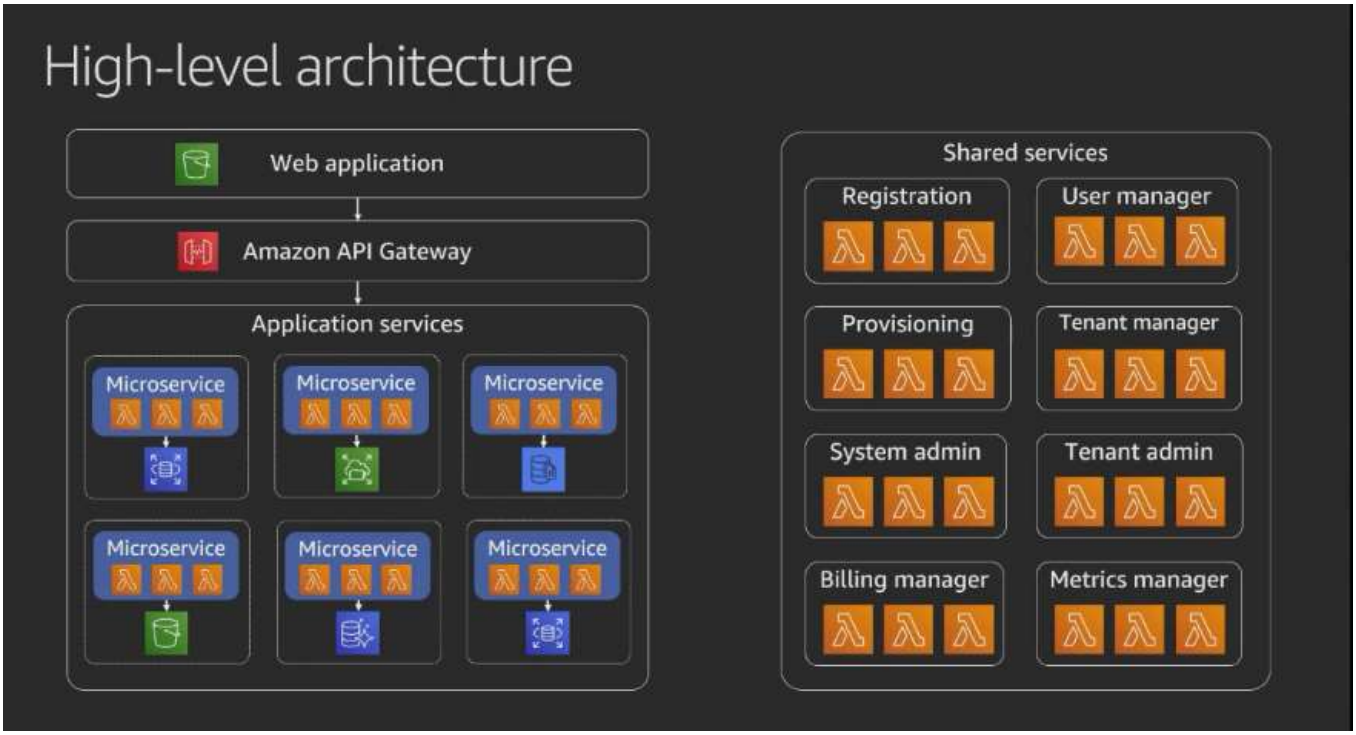
The emergence of serverless infrastructure and services represents a fundamental shift in how developers approach architecting applications. This is especially relevant in the world of SaaS, where systems must efficiently and cost-effectively respond to continually shifting multi-tenant loads and profiles. In this session, we dive into all the moving parts of a serverless SaaS application, exploring the detailed code, design, and architecture strategies that are commonly applied to support the complex scale and agility requirements of multi-tenant serverless SaaS environments. We also look at how serverless influences core multi-tenant strategies, including tenant isolation, service decomposition, management, deployment, and identity.





This is the dream view of a SaaS architect; this matches the overall tenant consumption of our SaaS product. Some stacks are much harder to achieve this consumption model but serverless excels here.

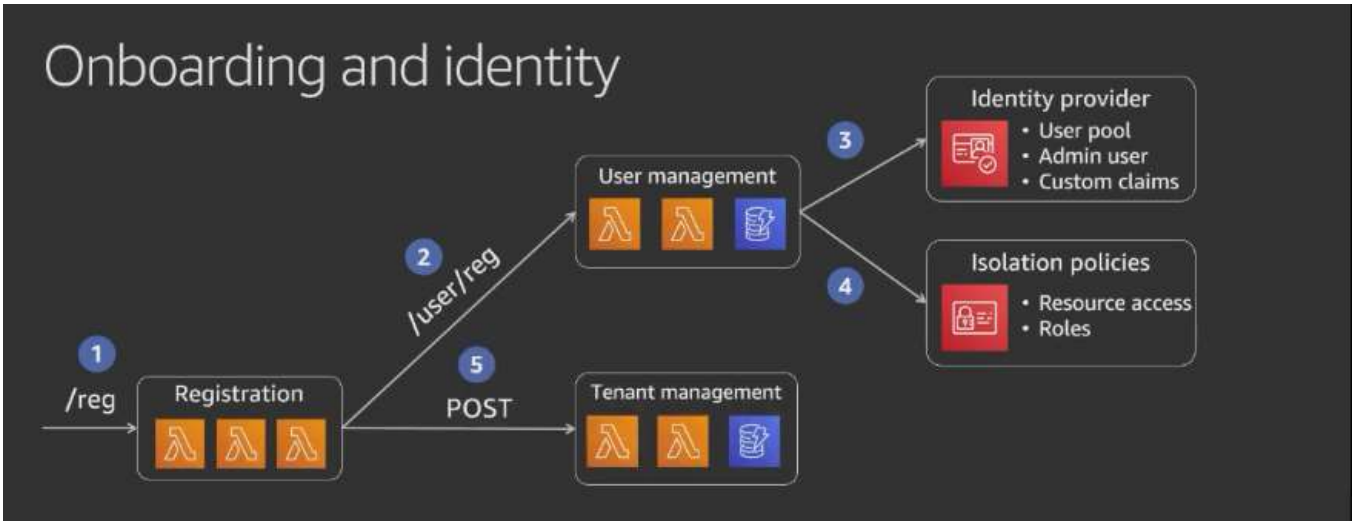


Our microservices are depicted as a collection of Lambda functions that own and manage their data. We also have lots of shared services as a collection of Lambda functions too that enables us to build and support our SaaS experience.

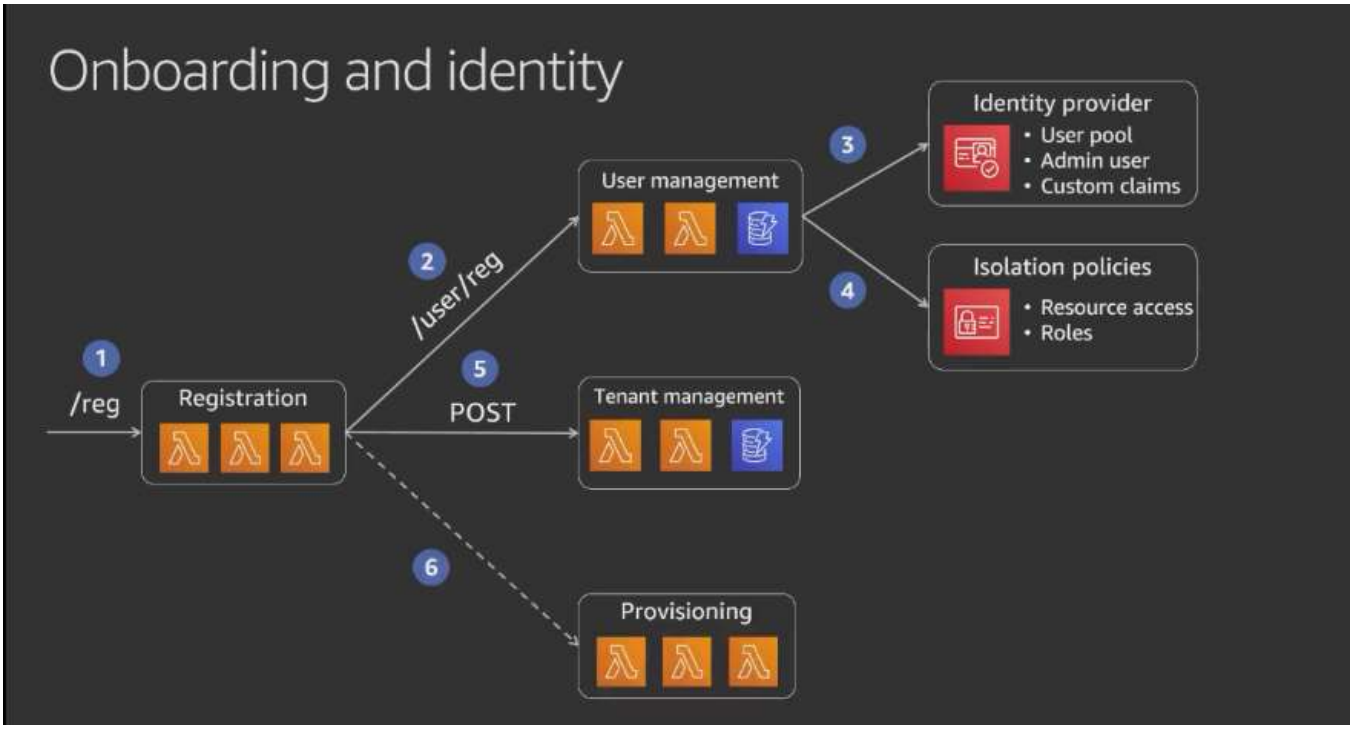
Our starting point is Onboarding and Identity for our SaaS tenants as they join and use our SaaS multi-tenant architecture. We have a Registration service with its lambdas that gets incoming requests, then call the User Management service that does orchestrates the creation of a user and their identity in the system

The User Mgt service goes out to some Identity Provider service to actually provision the identity for the user/tenant being created. This can be Cognito, Okta, etc. Cognito allows you to ***create user pools as set of policies and a grouping construct where a group of users will live in***. We will ***create a user pool for each tenant***, and their custom claims as the sets of attributes that we can put in the token to provide access information about each tenant.

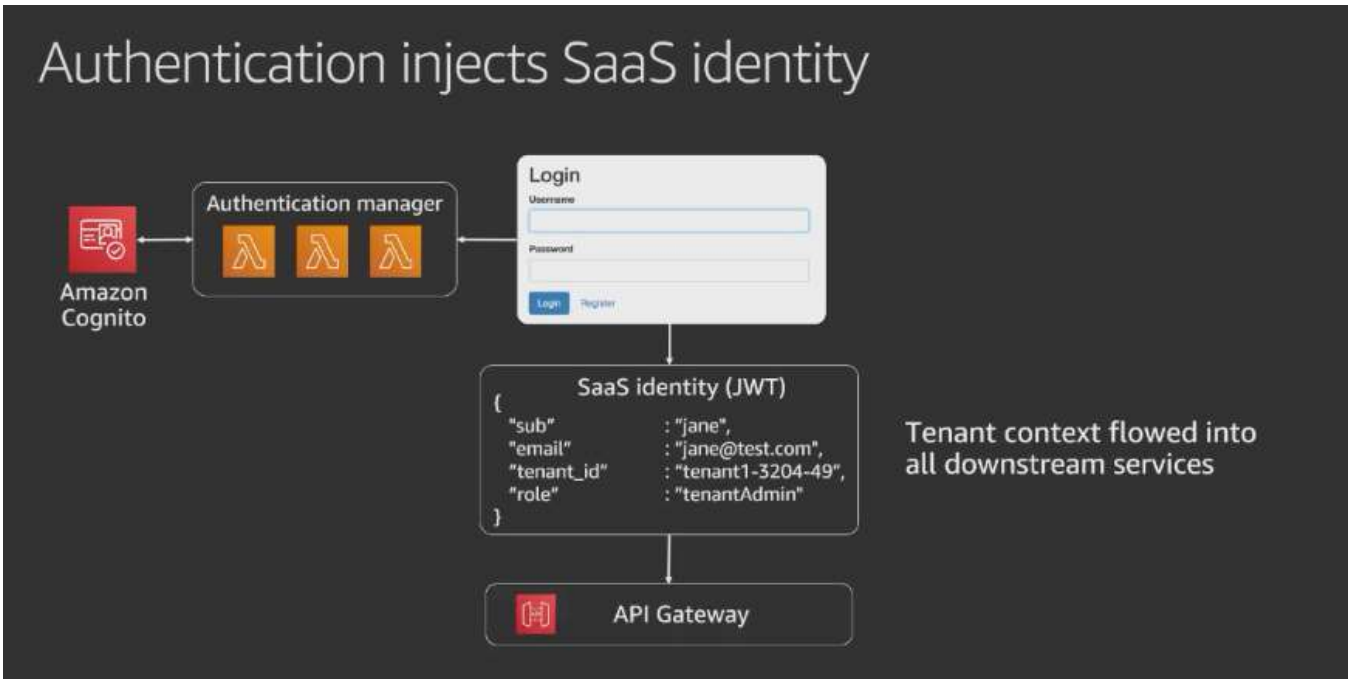
We then provision the policies and roles for the tenant



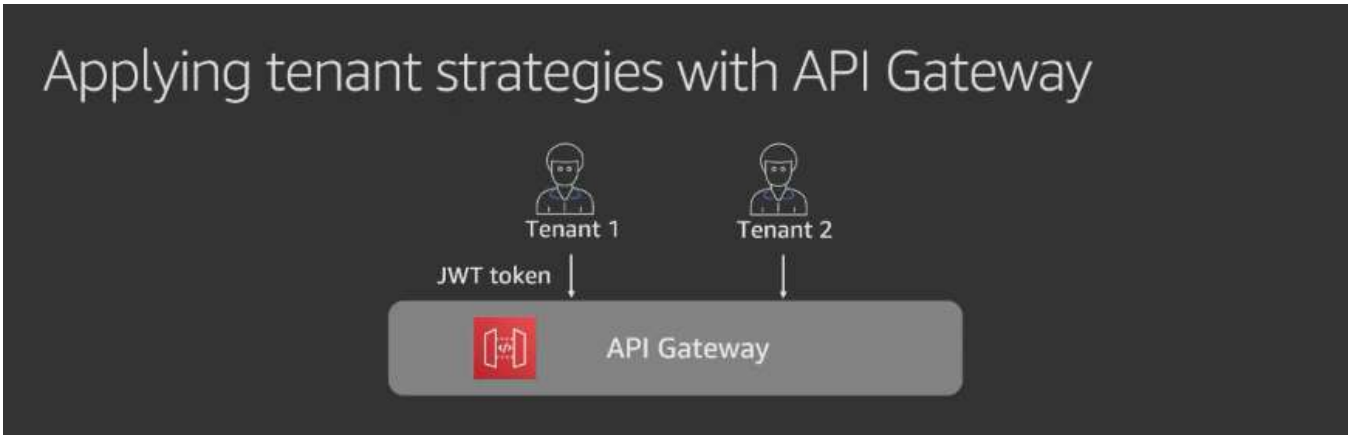
Then we can actually provision the tenant itself. There will be many users to can connect to each tenant.



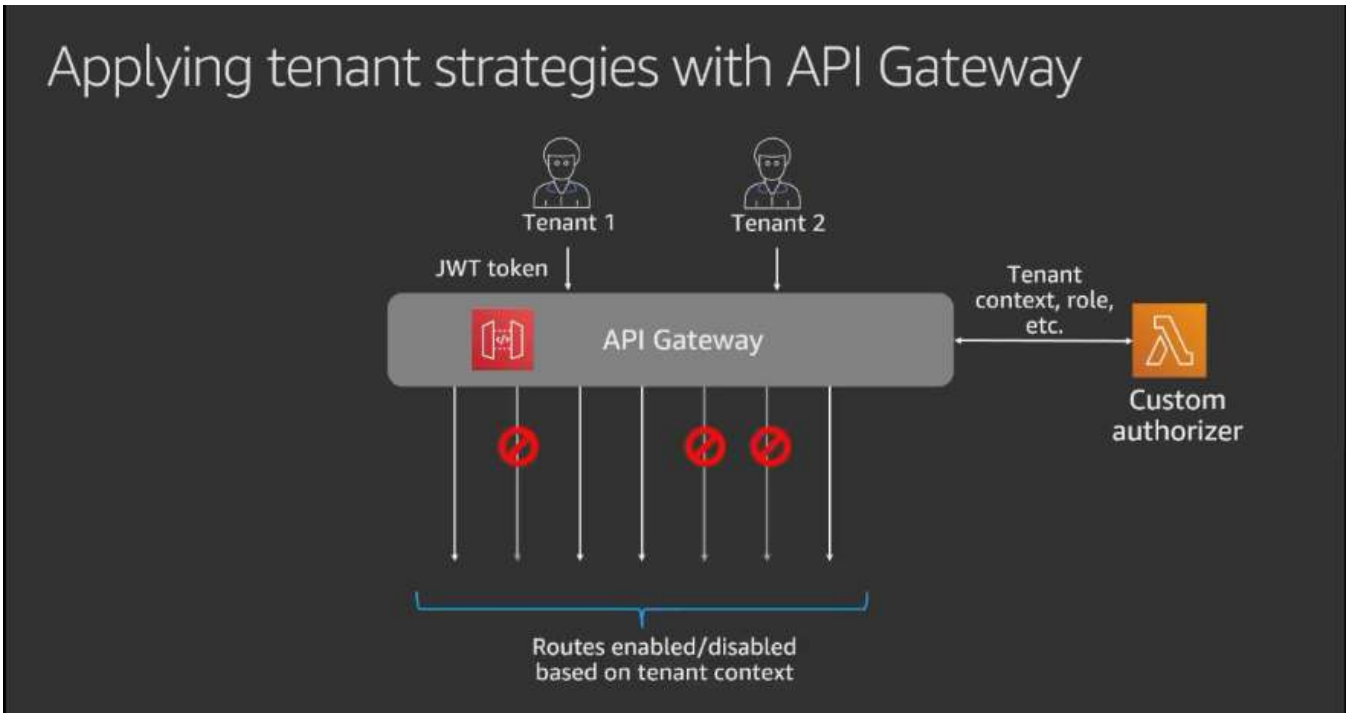
We then may need to go provision resources like (Lambda functions) or infrastructure for this tenant if needed.



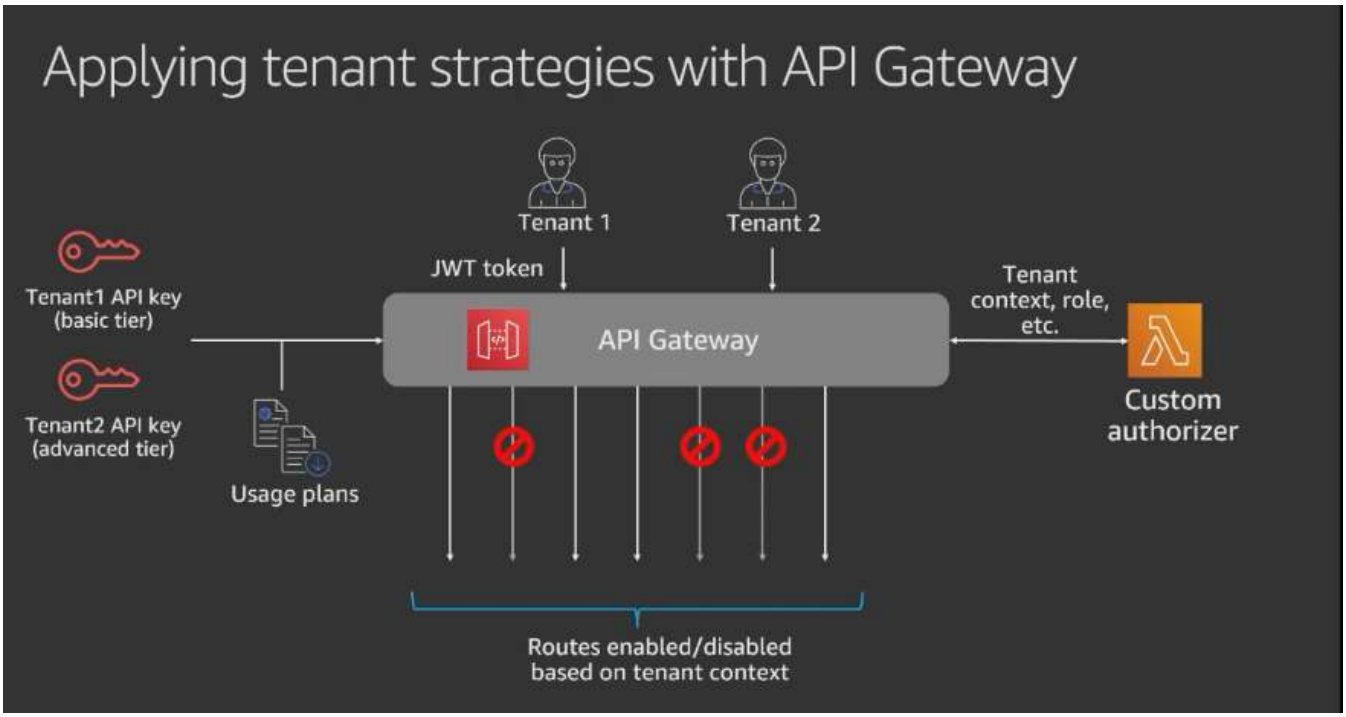
After user provisioning, we need authentication where we go to Cognito and get back a JWT token that is enriched with metadata about the user tenant_id, roles to access the API Gateway and other services. The token is key to determining how you partition data, isolate tenants, etc downstream.



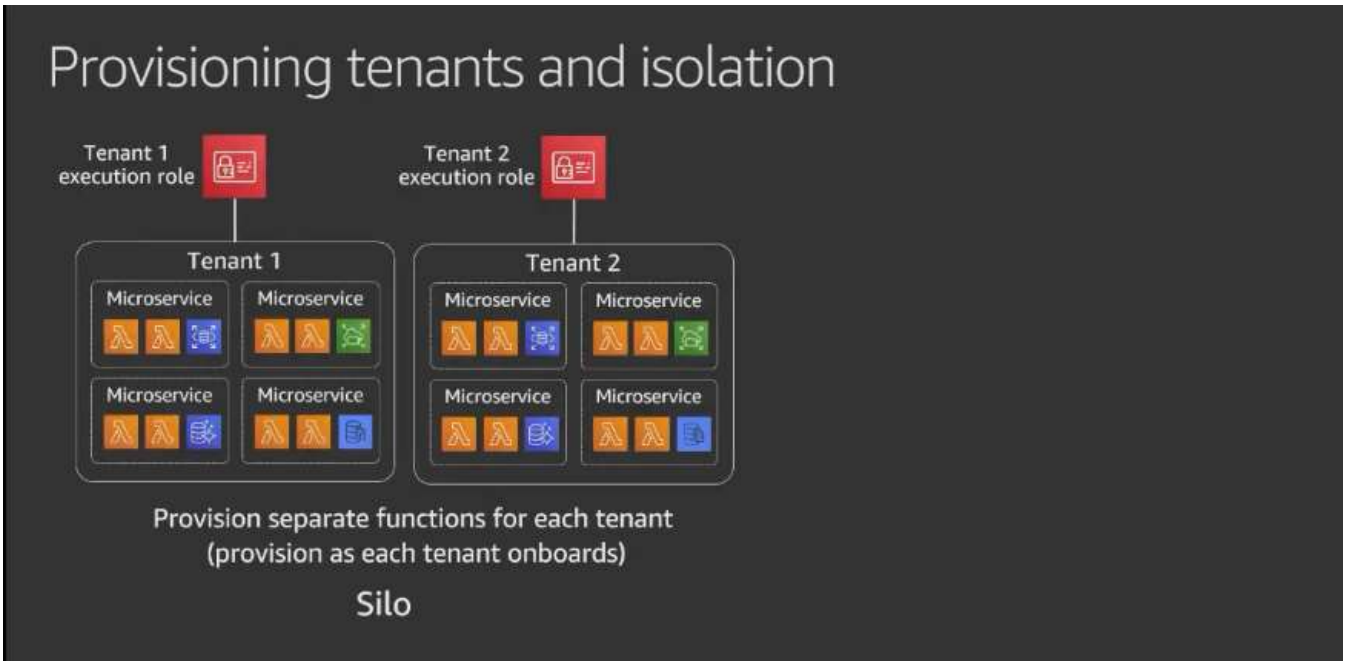
The entry through our API Gateway is our first opportunity to implement our first layer of security for all the other services within our SaaS platform. How can we enrich the security model here using the JWT token that tells us who the user is?



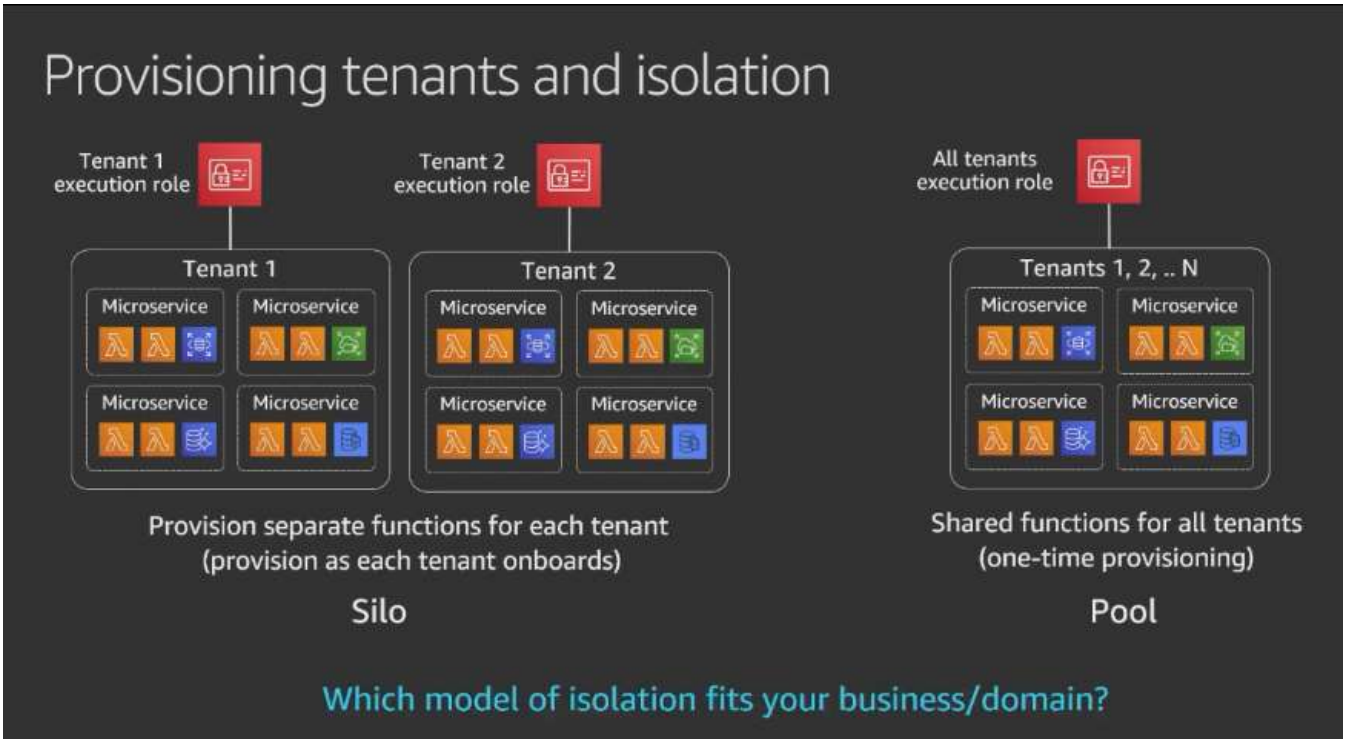
This is a Lambda function that we use to read the JWT token, read the attached user attributes like role and tier. It then configures a policy for that tenant that states the valid user routes and services accessible within the platform as a tenant.



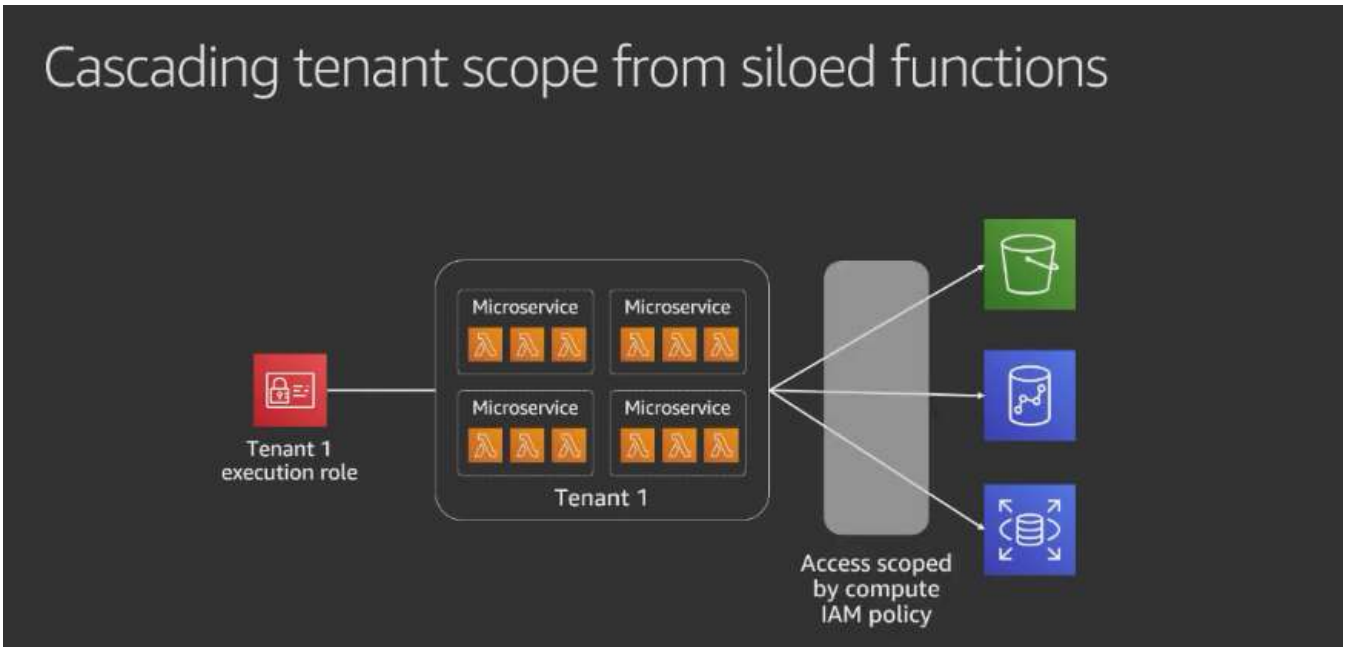
At the API Gateway, we can also use Usage Plans and different API Keys to prevent some tenants from saturating the API Gateway calls, we don’t want the basic tiers to affect or the paid plans.



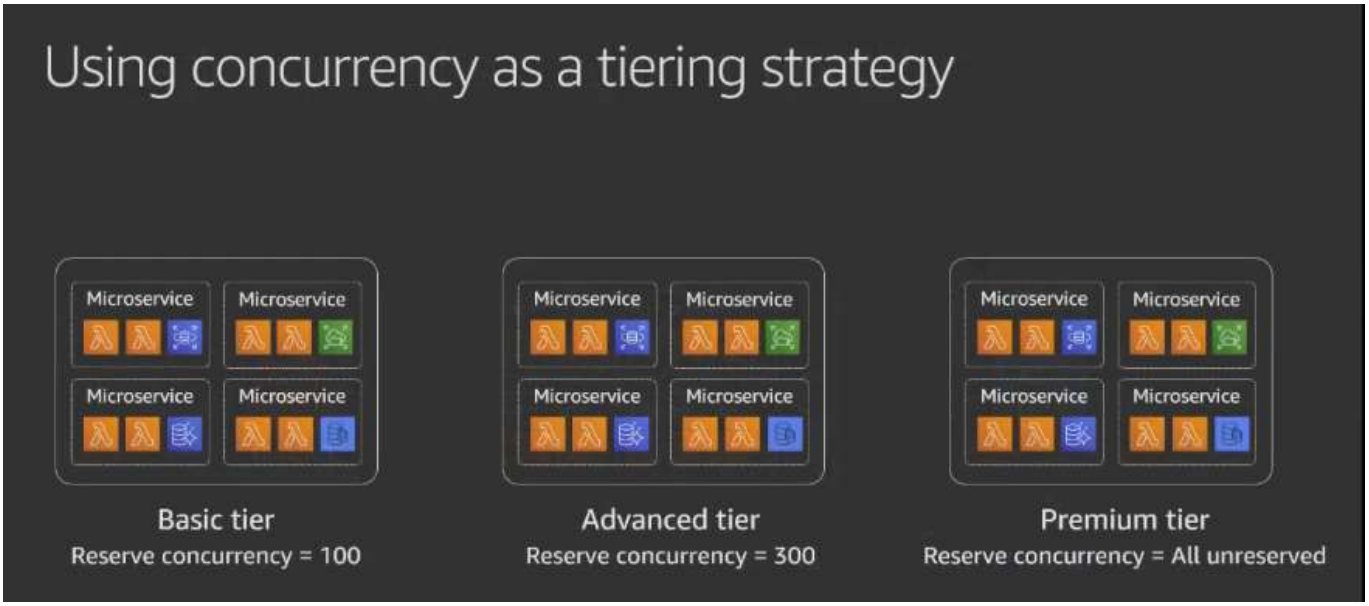
Beyond the API Gateway, we can now look at the different microservices making up our SaaS application. We can implement features like isolation here. We can create siloed resources (like Lambda functions with attached IAM execution roles that access what they can do or call within our system) for some tenant without sharing with other users.



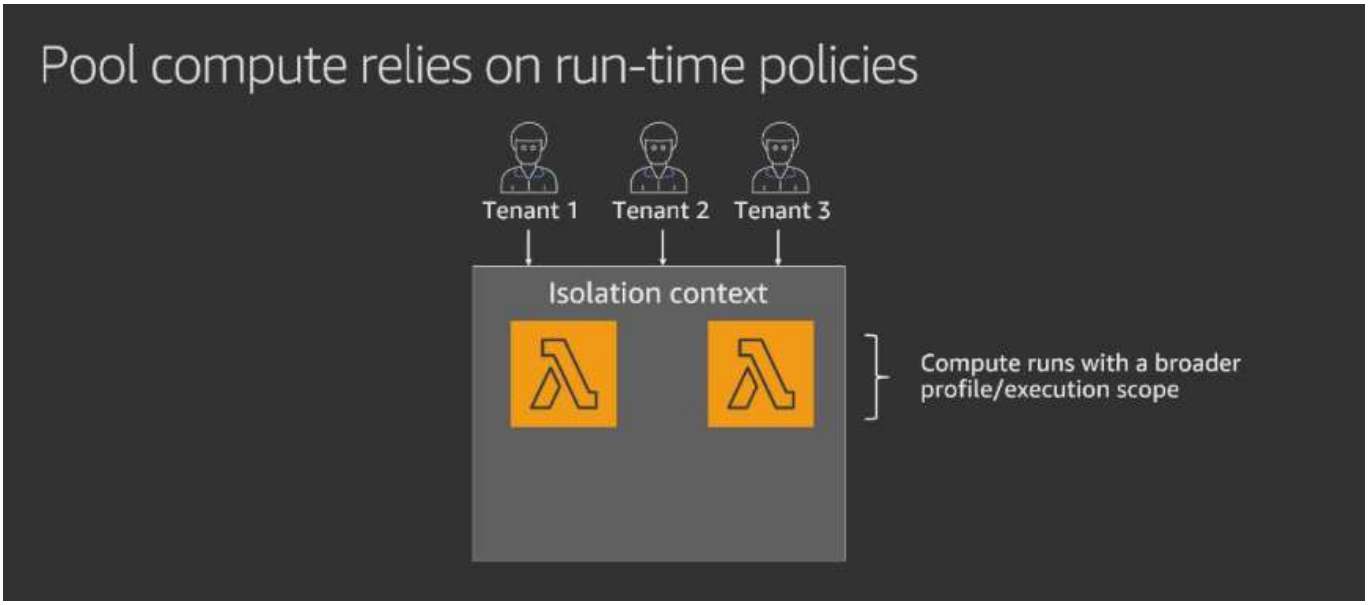
The Pool model is where resources are shared between tenants, we deploy with a role having a much wider execution role that does not guarantee mush isolation between the tenants.



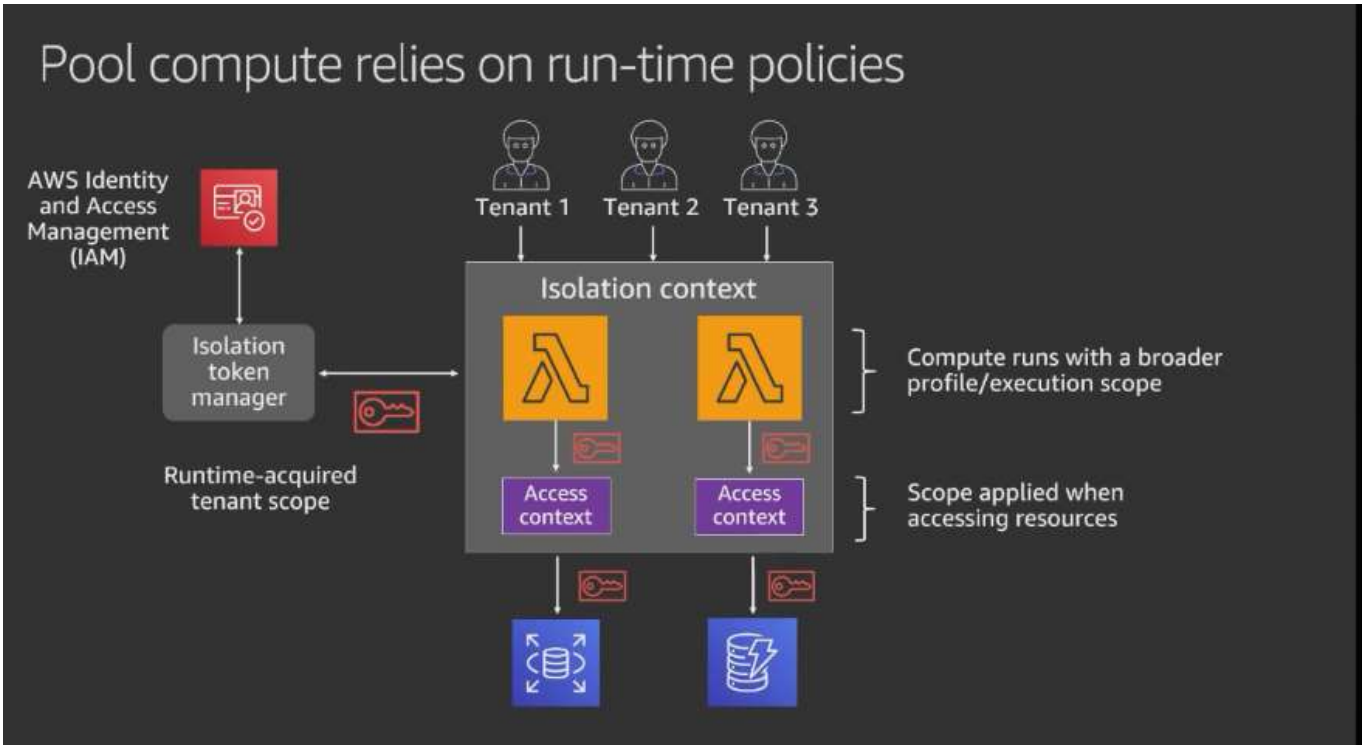
The IAM role attached will not allow this tenant 1 to go outside its scope and maybe touch data of other tenants.



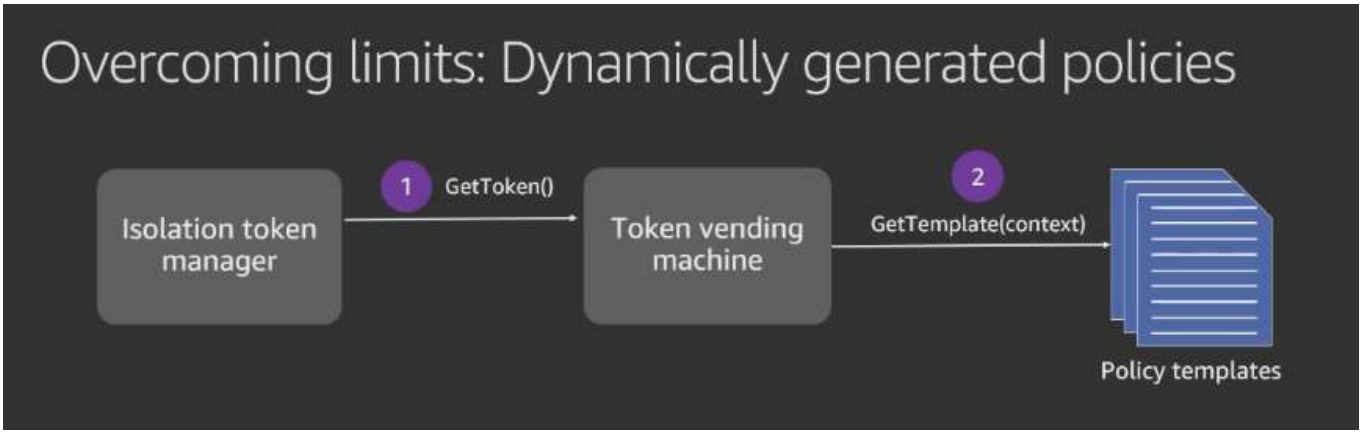
We can also limit the number of concurrent services like Lambdas running in each model, our Premium tier users will not be throttled via concurrency limits but basic tier users will. Note that this approach requires that these tiers are deployed and provisioned independently.



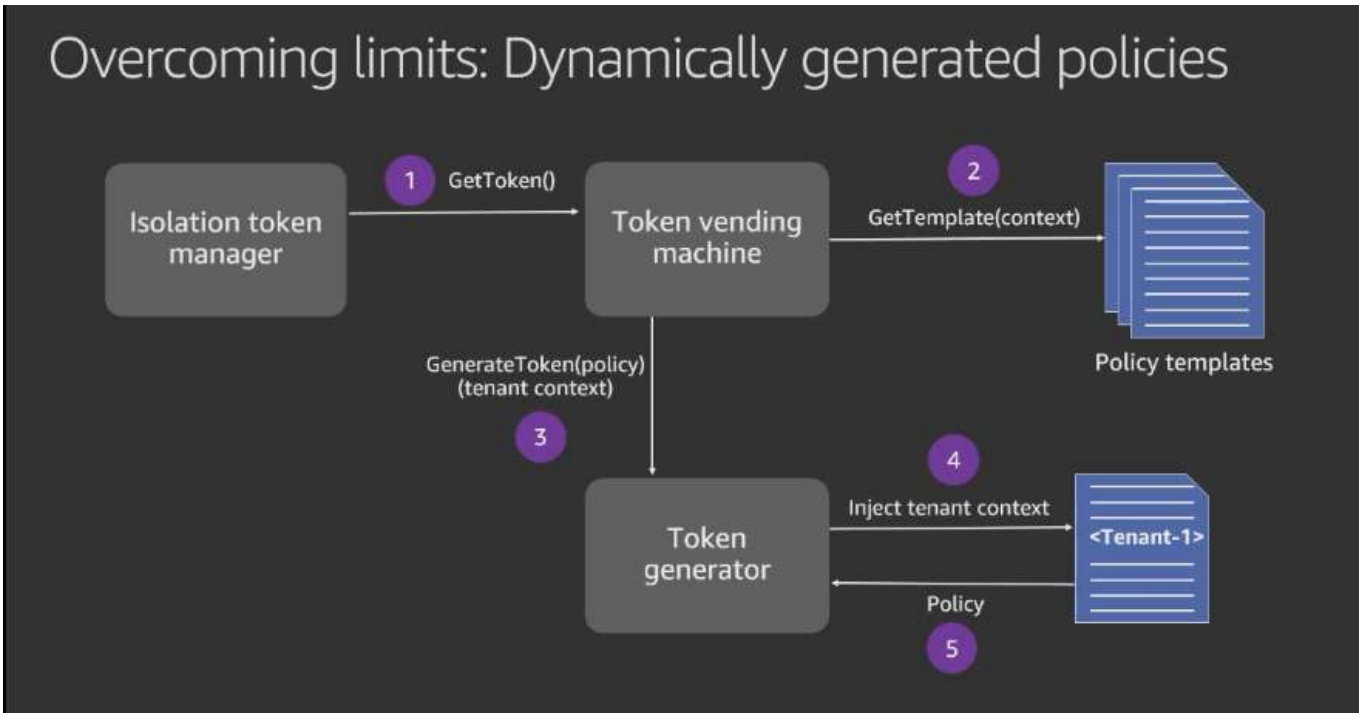
Here we have multiple tenants sharing a pool and our isolation strategy will need to be different. These functions are now shared and executing individual Lambdas.



At runtime, we need to create and use scoped tokens for each active tenant to prevent them from accessing other tenant data when accessing resources with limited access.



Since we might have so many policies and roles created in our SaaS case, we can instead at run-time use dynamically create and generate/define policy and get the credentials in real-time. These created policy templates are not scoped to any particular tenant yet



We now use the Policy Template and the specific tenant context to enable a Token generator create a Policy for each tenant to use. You are now creating and managing the policies yourself.

Creating IAM policy templates

Policy template

```
{
  "Sid": "DynamoDBTenatItemRole",
  "Effect": "Allow",
  "Action": [
      "dynamodb:GetItem",
      "dynamodb:BatchGetItem",
      "dynamodb:Query",
      "dynamodb:DescribeTable"
  ],
  "Resource": [
      "{DynamoTable.ARN}"
  ],
  "Condition": {
      "ForAllValues:StringEquals": {
          "dynamodb:LeadingKeys": [
              "{Tenant.Tenant_id}"
          ]
      }
  }
}
```

Policy instance

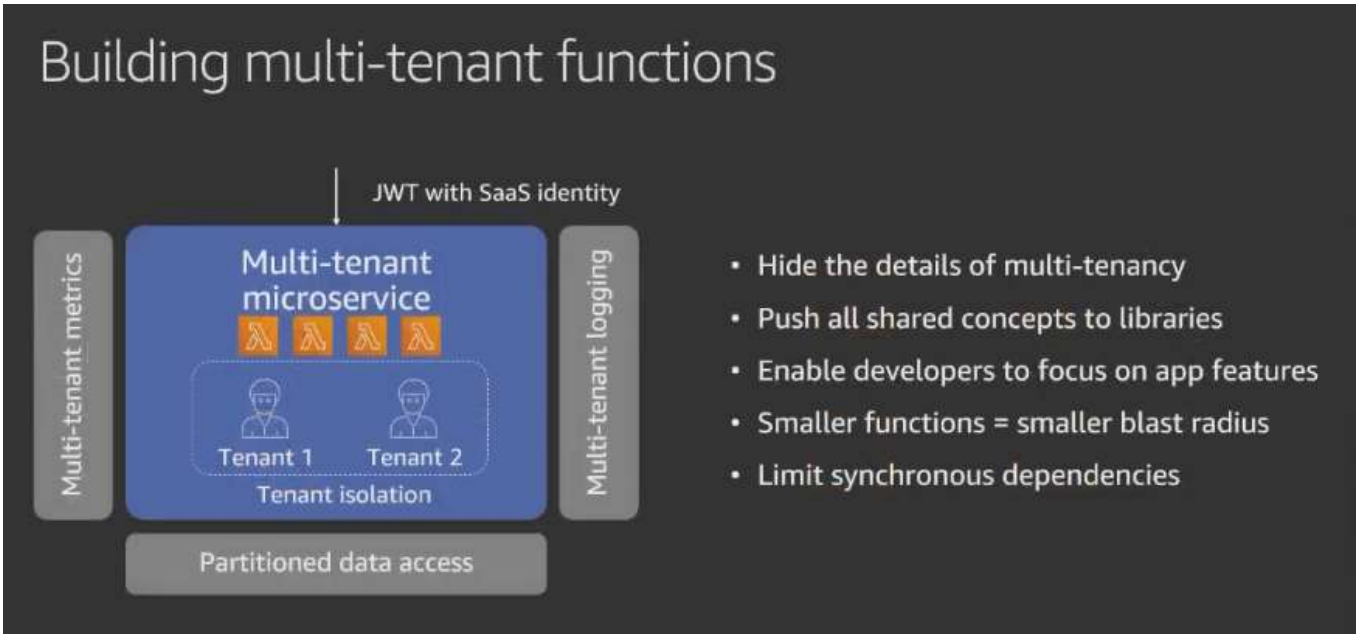
```
{
  "Sid": "DynamoDBTenatItemRole",
  "Effect": "Allow",
  "Action": [
      "dynamodb:GetItem",
      "dynamodb:BatchGetItem",
      "dynamodb:Query",
      "dynamodb:DescribeTable"
  ],
  "Resource": [
      "arn:aws:dynamodb:us-east-1:xxx:table/Order"
  ],
  "Condition": {
      "ForAllValues:StringEquals": {
          "dynamodb:LeadingKeys": [
              "5bd24c40d66c4755819d28ceab9f0826"
          ]
      }
  }
}
```

We inject all the actual context to create a fully formed policy to get a token for as shown below

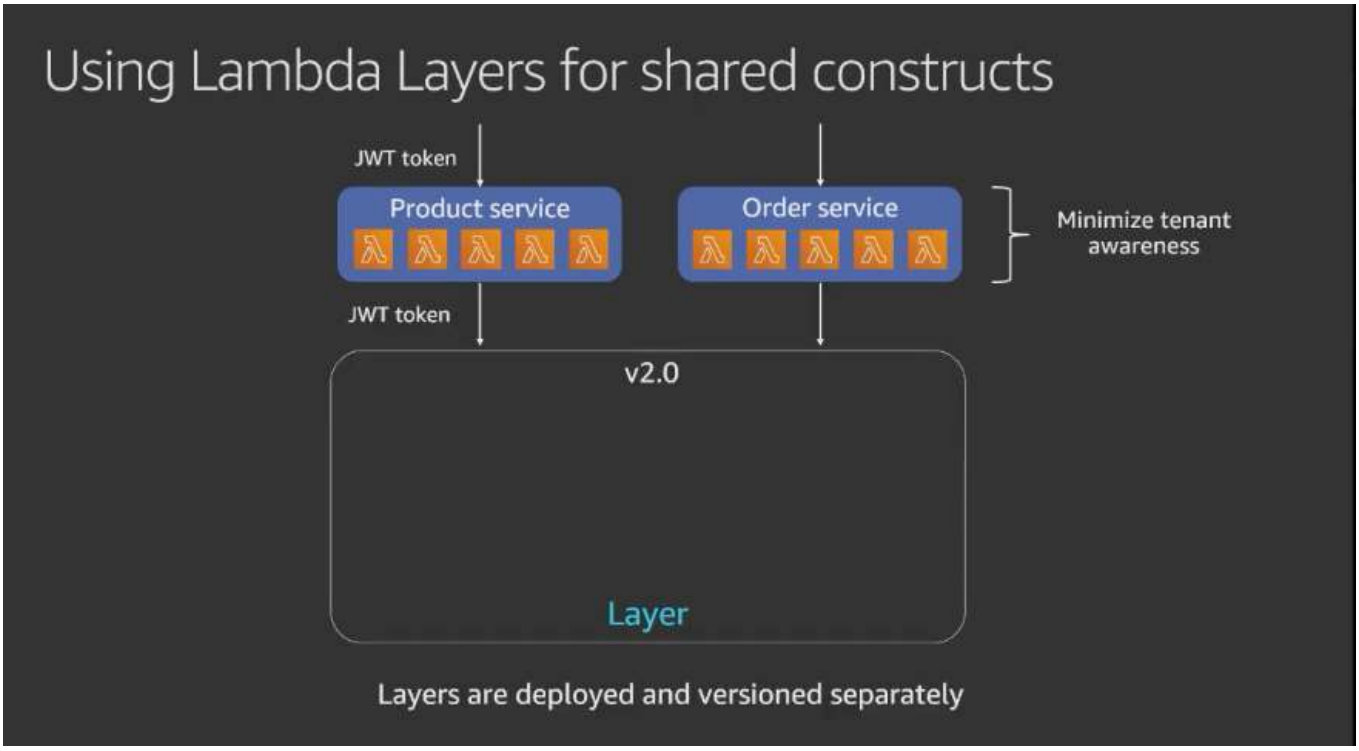
Get credentials for dynamic policy

```
public Credentials getCredentials(Policy policy, string user_name) {
  // create a request with the supplied user name and policy
  var request = new GetFederationTokenRequest()
  {
    Name = user_name,
    Policy = policy.ToJson()
  };

  // construct security token service
  var stsClient = new AmazonSecurityTokenServiceClient();

  // use request to get the federated token
  var response = stsClient.GetFederationToken(request);

  // return the credentials from the response
  return response.GetFederationTokenResult.Credentials;
}
```

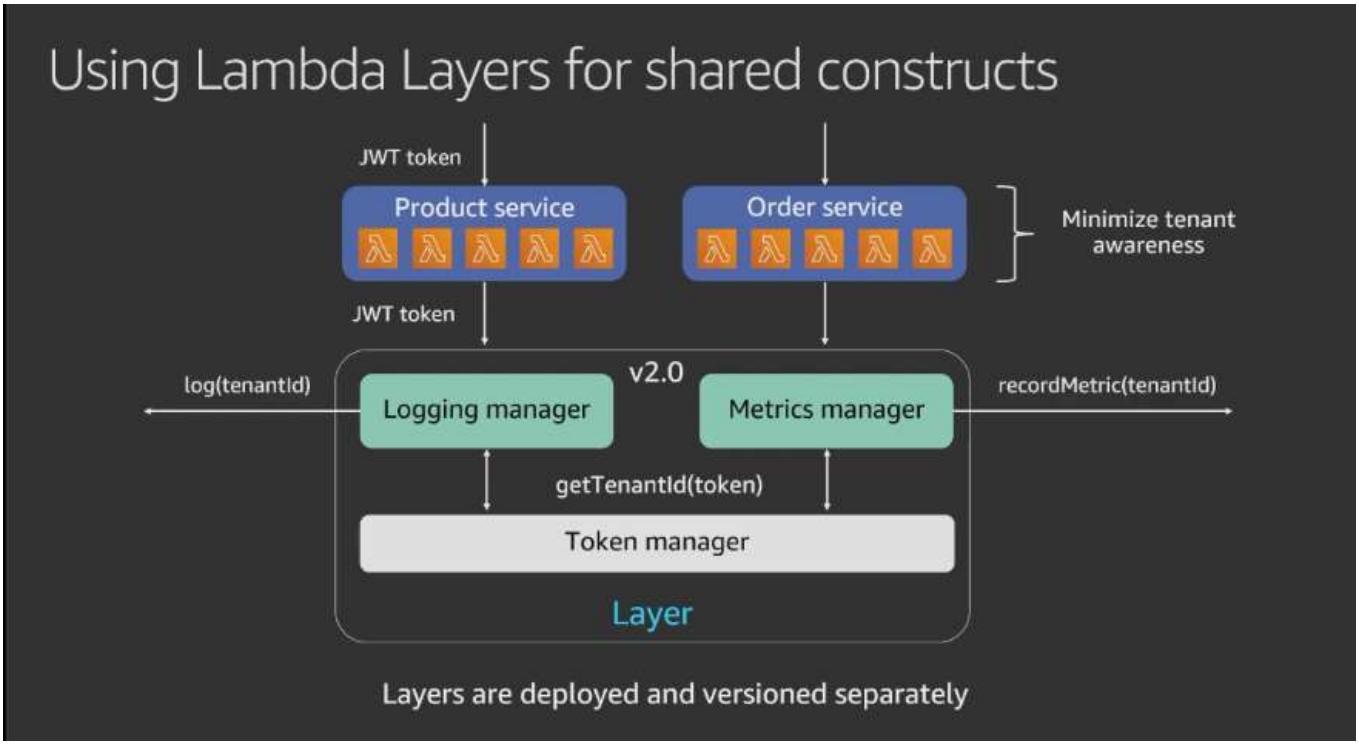
This is how to get a token for a policy using the AWS SDK. The response we get is the JWT token with our credentials to access the resources.



We don’t want developers thinking about multi-tenancy and dealing with tenant context like tenantId. We push multi-tenant concepts out to the edge and out of the developers view. Metrics, partitioned data, logging needs tenant context and needs to be injected into the logs but not by the developers.



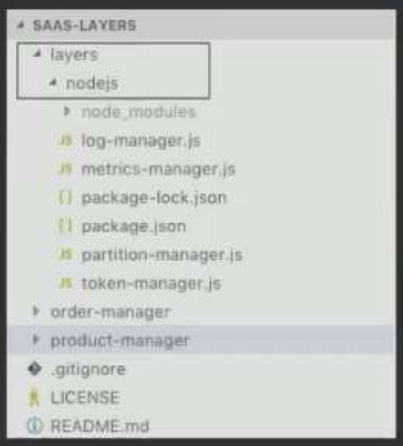
We are using Lambda layers to move out and implement things like logging, metrics, etc. that can be referenced by functions easily and use. The layer will contain the code that has all the multi-tenant concepts



The developer instead calls the Logging Manager or Metrics Manager within their code and pass the token along with the message that needs to be logged. The Logging Manager or Metrics Manager will send the token to the Token Manager to get the details they need like the tenantId and inject it into the log for tenant context. ***This pulls out multi-tenant code from your microservices***.

Introducing layers into your environment

Layers in your IDE



Use language directory conventions to import layer dependencies

```
let AWS = require('aws-sdk');
let logManager = require('/opt/nodejs/log-manager.js');
let tokenManager = require('/opt/nodejs/token-manager.js');
let metricsManager = require('/opt/nodejs/metrics-manager.js');
let dal = require('./order-manager-dal.js');
```

Note that Layers require you to use the '/opt/nodejs' path to access your layers code

Centralizing tenant context management

```
'use strict';

let nJwt = require('njwt');

const signingKey = "af92fb8f-ebbf-4df6-a54f-6355b0e9ea28";

// Get the tenant identifier from the supplied context
module.exports.getTenantId = function(event) {
 var bearerToken = event.headers['Authorization'];
 var tenantId = "";

 try {
  var jwtToken = bearerToken.substring(bearerToken.indexOf(' ') + 1);
  var verifiedJwt = nJwt.verify(jwtToken, signingKey);
  tenantId = verifiedJwt.body.tenantId;
 } catch(e) {
  console.log("Error verifying token: " + e);
 }

 return tenantId;
}
```

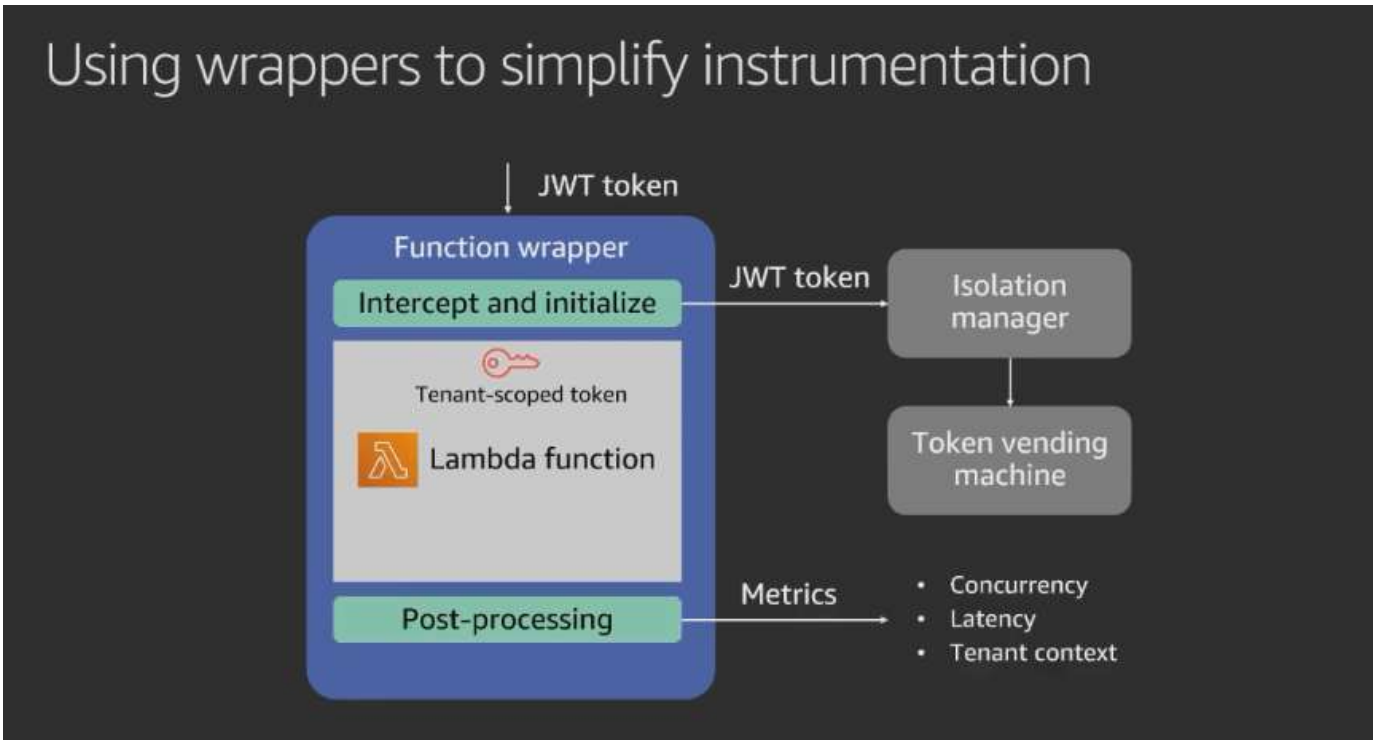
The payoff: A simplified developer experience

```
let AWS = require('aws-sdk');
let logManager = require('/opt/nodejs/log-manager.js');
let tokenManager = require('/opt/nodejs/token-manager.js');
let metricsManager = require('/opt/nodejs/metrics-manager.js');
let dal = require('./order-manager-dal.js');

exports.get = (event, context, callback) => {
    logManager.logWithTenantContext(event, "GetOrder() called. OrderId = " + event.pathParameters.resourceId);

    dal.getOrder(event, function(response) {
        callback(response);
    });
};

exports.put = (event, context, callback) => {
    logManager.logWithTenantContext(event, "PutOrder() called. OrderId = " + event.pathParameters.resourceId);

    dal.updateOrder(event, function(response) {
        callback(response);
    });
};
```

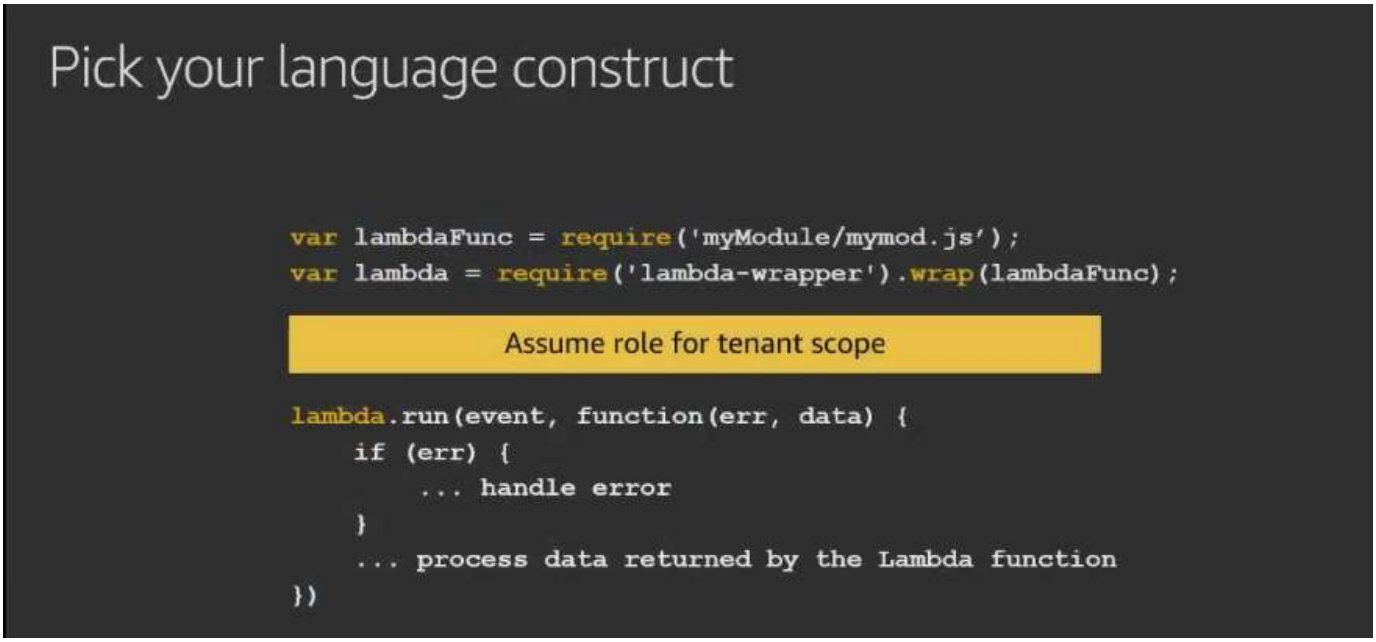
The developer can now focus on what they need to do to add logging by calling a Layer and not refer to any tenant related issue. This code does not know about multi-tenancy!

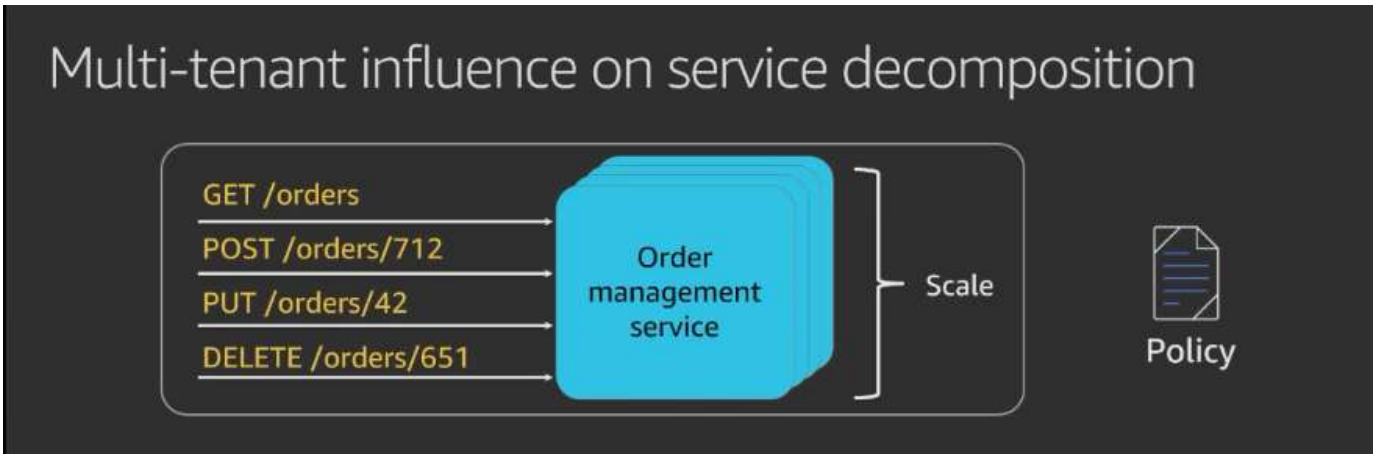
Using wrappers to simplify instrumentation

We can put a wrapper around a Lambda function to do things like preprocess and postprocessing.

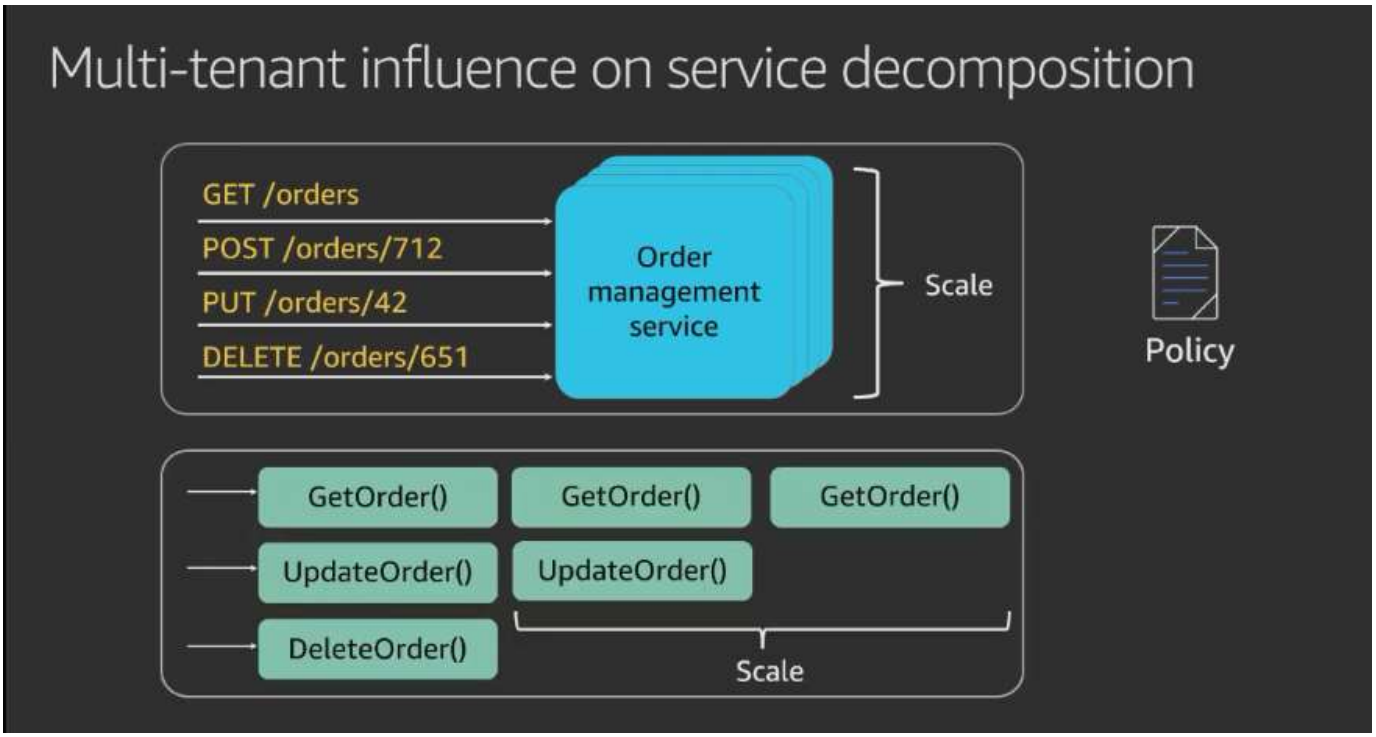


Wrappers can be used for preprocessing before the function code built by the developer is ran. We can also do postprocessing for metrics emitted with tenant context.

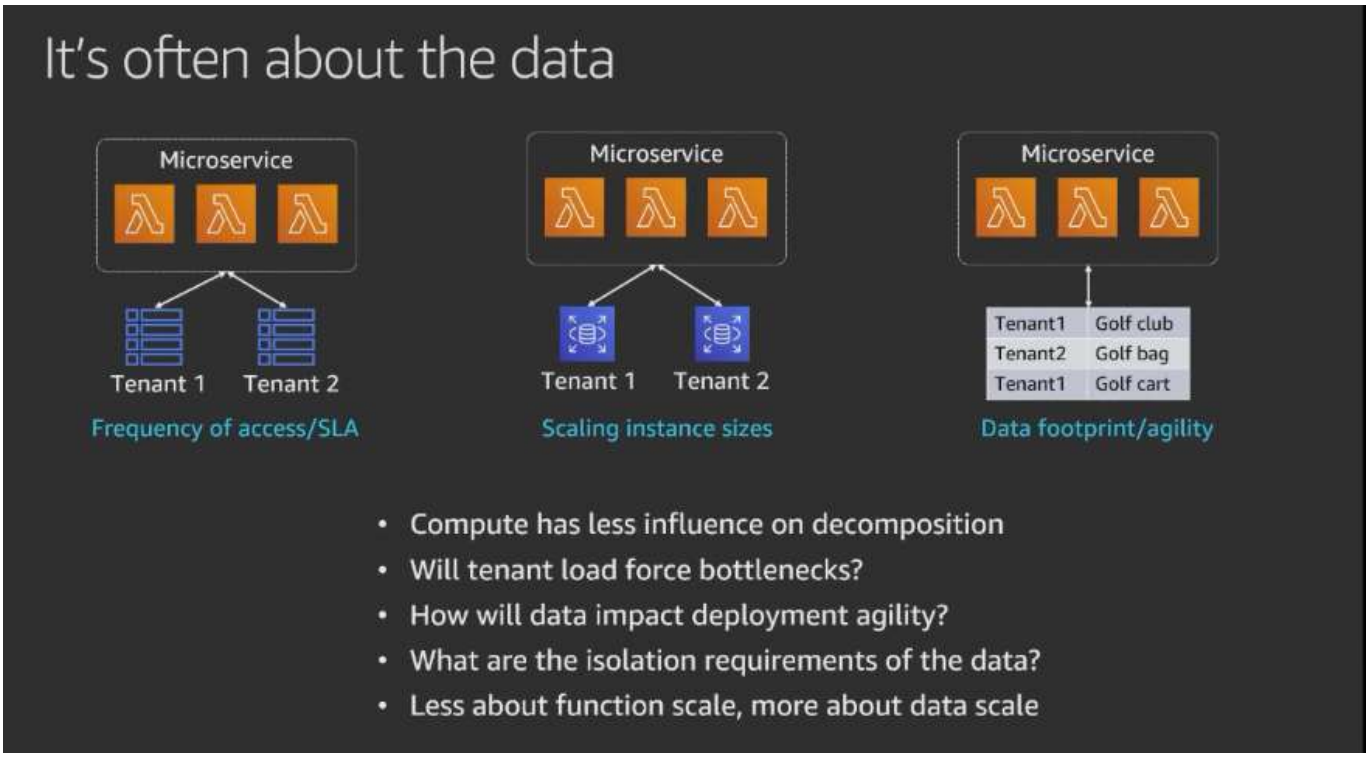




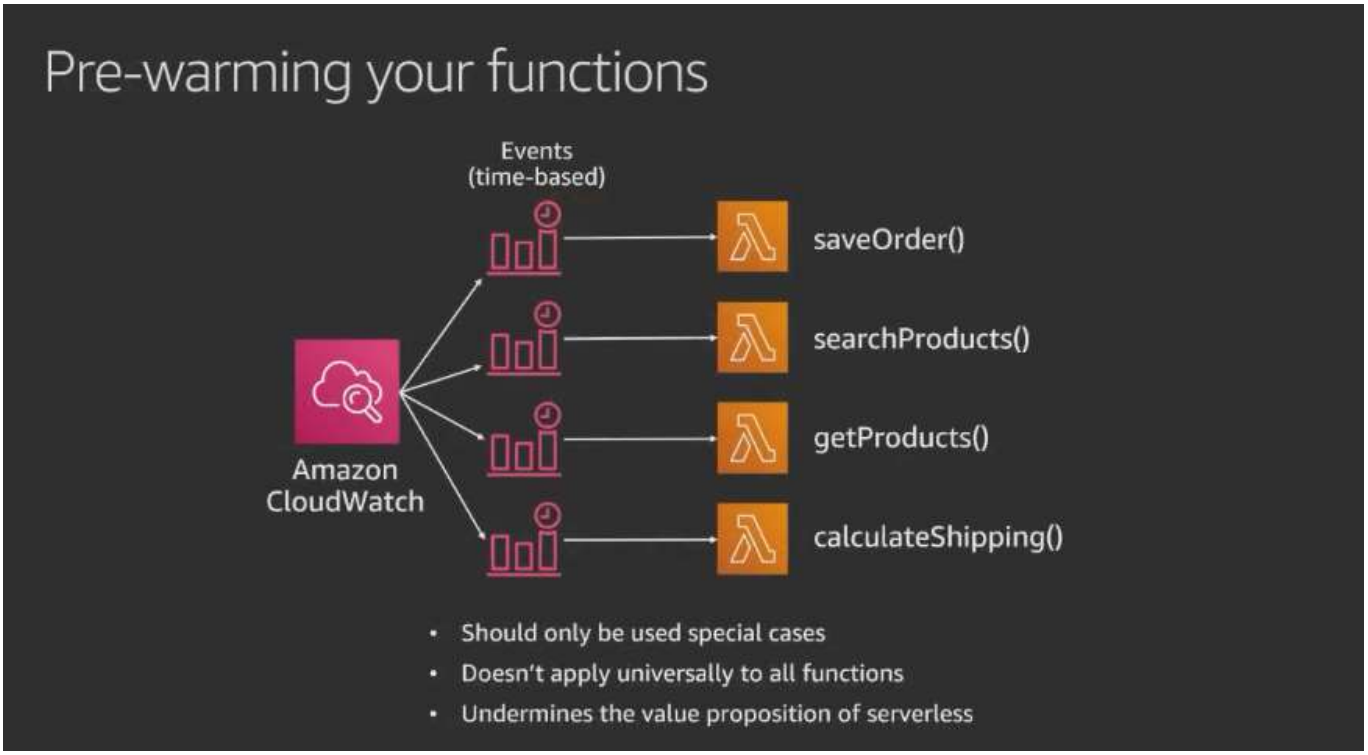
We need to think about our ***unit of scale*** or how to scale microservices above, or functions as below



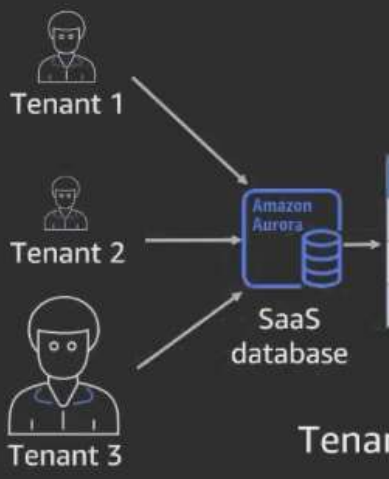
We think about decomposition differently in a Serverless SaaS model below



Decomposition gets different when we talk about the data in the Serverless SaaS model regarding the agility, isolation, performance and compliance data needs.



We still have servers to think about

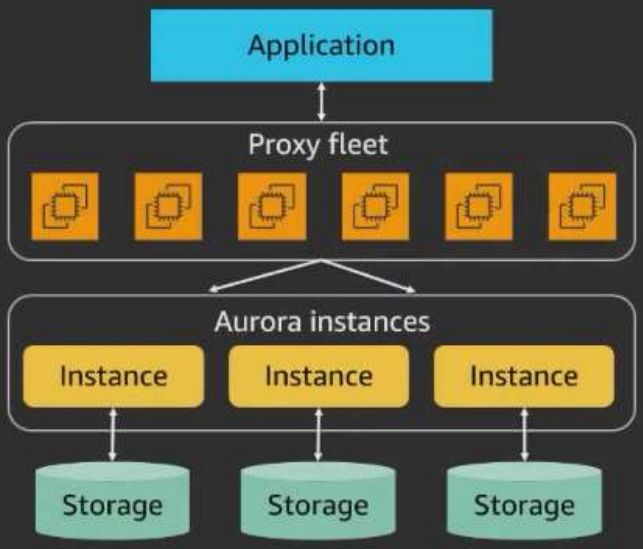


Customer table

TenantID	Name	StartDate
Tenant-1	Jane Doe	01-12-2012
Tenant-4	John Doe	09-01-2015
Tenant-9	Mary Major	11-04-2017

Tenants can rapidly saturate and instance

Extending the value of serverless to storage



Amazon Aurora serverless

- Remove the notion of servers/instances
- All data is kept in highly available storage volume
- Application talks to a MySQL-compatible endpoint
- Fleet of proxy servers manage, queue, and route database traffic

Takeaways

- Serverless enables SaaS agility, resilience, and innovation
- Isolation strategies directly shape deployment footprint
- Consider supporting multiple isolation models (hybrid)
- Factor account limits in your isolation model
- Hide details of multi-tenancy from service developers
- Consider using layers as the home for shared multi-tenant concepts
- Serverless allows us to better align tenant consumption and activity

Related SaaS sessions

aws partner network | saas factory

Breakouts

- ARC210 – Microservices decomposition for SaaS environments
- GPSTEC337 – Architecting multi-tenant PaaS offerings with Amazon EKS

Chalk talks

- ARC413 – SaaS metrics deep dive: A look inside multi-tenant analytics
- ARC305 – Migrating single-tenant applications to multi-tenant SaaS
- GPSTEC306 – Reinventing your technology product strategy with a SaaS delivery model
- API308 – Monolith to serverless SaaS: Migrating to multi-tenant architecture

Related SaaS sessions (cont'd)

aws partner network | saas factory

Workshops

- SVS303 – Monolith to serverless SaaS: A hands-on service decomposition
- ARC308 – Hands-On SaaS: Constructing a multi-tenant solution on AWS

Builder sessions

- ARC405 – Building multi-tenant-aware SaaS microservices

Lightning talks

- DEM142 – SaaS migration strategies
- DEM143 – Multi-region SaaS: Staying agile in a multi-geography model

Learn to architect with AWS Training and Certification

Resources created by the experts at AWS to propel your organization and career forward

Free foundational to advanced digital courses cover AWS services and teach architecting best practices

Classroom offerings, including Architecting on AWS, feature AWS expert instructors and hands-on labs

Validate expertise with the **AWS Certified Solutions Architect - Associate** or **AWS Certification Solutions Architect - Professional** exams

Visit aws.amazon.com/training/path-architecting/

aws training and certification

Thank you!

Tod Golding
todg@amazon.com