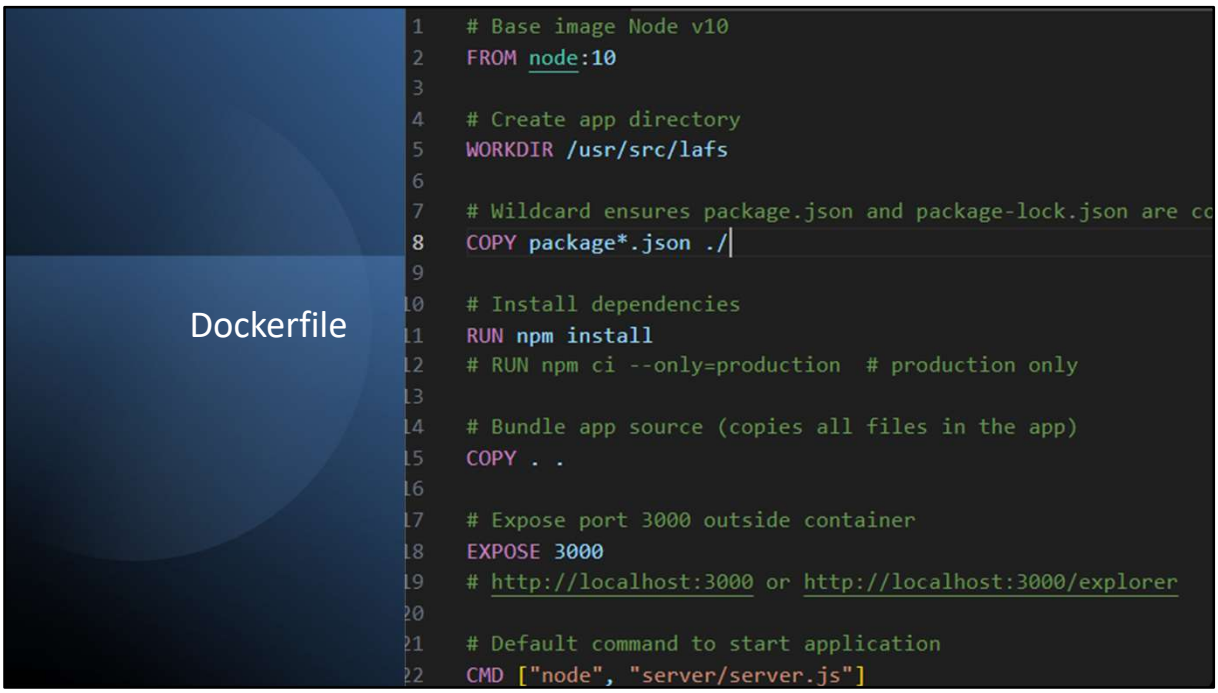
Hello. I'm Matthew Pool, and this presentation covers cloud migration and development. I hope you enjoy.

Overview

To the cloud

In this presentation, I'm going to go over some of the intricacies of migrating an app to the cloud and developing the cloud infrastructure for which it runs upon. This just means that instead of code located on local machines with servers that are manually managed, we can just upload the code to the cloud and have these things managed for us. This allows seamless scalability, security, performance, and other benefits.

Docker automates software deployment by providing containerization and allowing for a microservices architecture, which means we can develop and deploy components independently. This works by running apps in containers [SPACE] that allow a consistent environment to be run across different platforms, so no need to worry about dependencies and related issues.
[SPACE] The command seen at the bottom runs a MongoDB container using Docker, and maps the container's ports to the host machine's ports, so that the MongoDB service inside the container is accessible from the host machine.
For example, the original application may use a NoSQL database like [SPACE] MongoDB. This database can be containerized and converted to a cloud database like Amazon's DynamoDB.



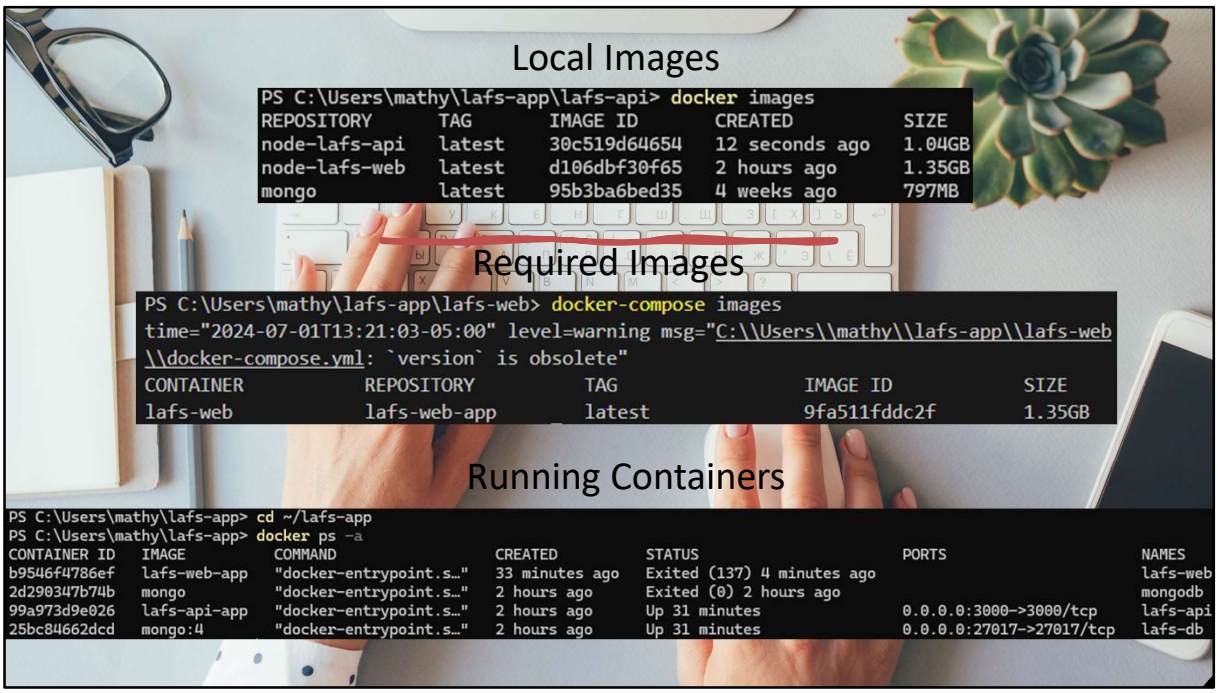
Docker containerization uses a Dockerfile for each container – typically one for the backend and one for the database. The frontend can also be containerized, if you're developing a web app. Here you can see the Dockerfile code for a backend API container. This file specifies the base image to use, the build directory, the port to expose, commands to execute, and files to copy for building the container.

```yaml
# /lafs-web/docker-compose.yml

# NOTE: Indentation matters in YAML
version: '3.7'
services:
  # Angular frontend application
  app:
    container_name: lafs-web
    restart: always
    build: .
    ports:
      - '4200:4200'
    command: >
      bash -c "npm install && ng serve --host 0.0.0.0 --port 42
# Attach the external network to these containers
networks:
  default:
    external:
      name: lafs-net
```

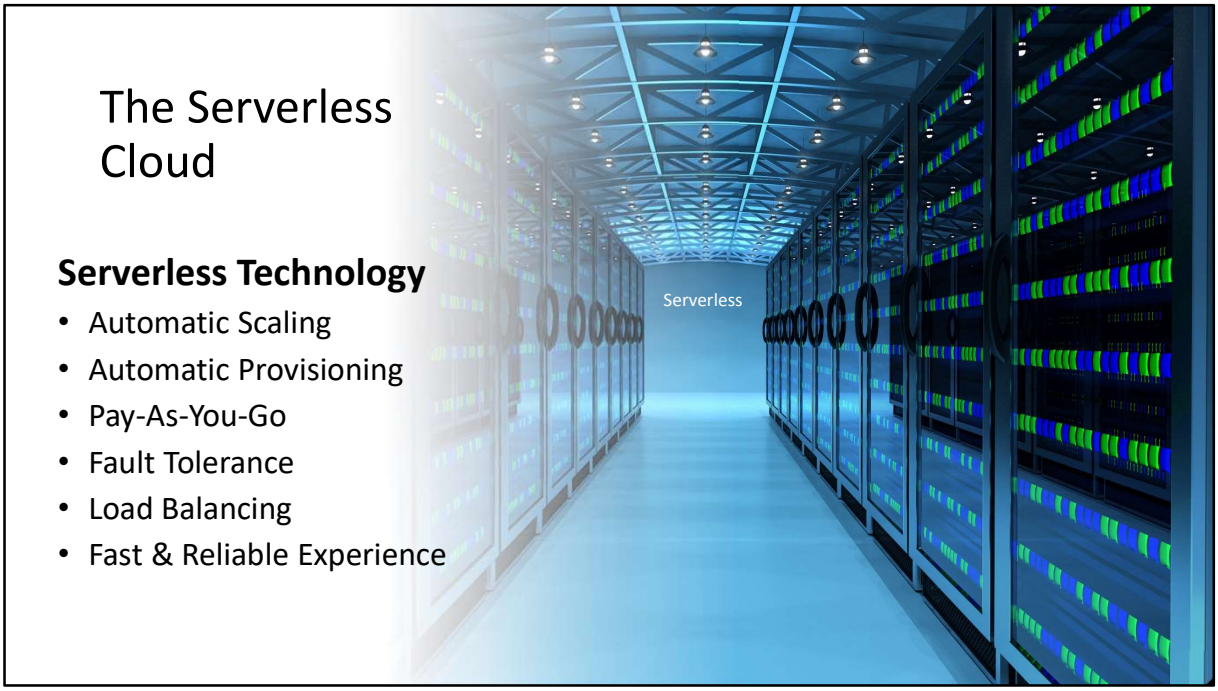
Orchestration

- Kubernetes & Docker Compose
- Manage Multiple Containers
- docker-compose.yml

Kubernetes and Docker Compose can help automate and manage deployment and orchestration of multiple containers. Here at the top right of the screen, you can see an example of a docker-compose.yml file. This file defines the Docker Compose version, the services, the port mapping, the command to install dependencies and start the Angular server, the network, and other attributes. In addition to these attributes, the backend docker-compose file specifies the database as well.



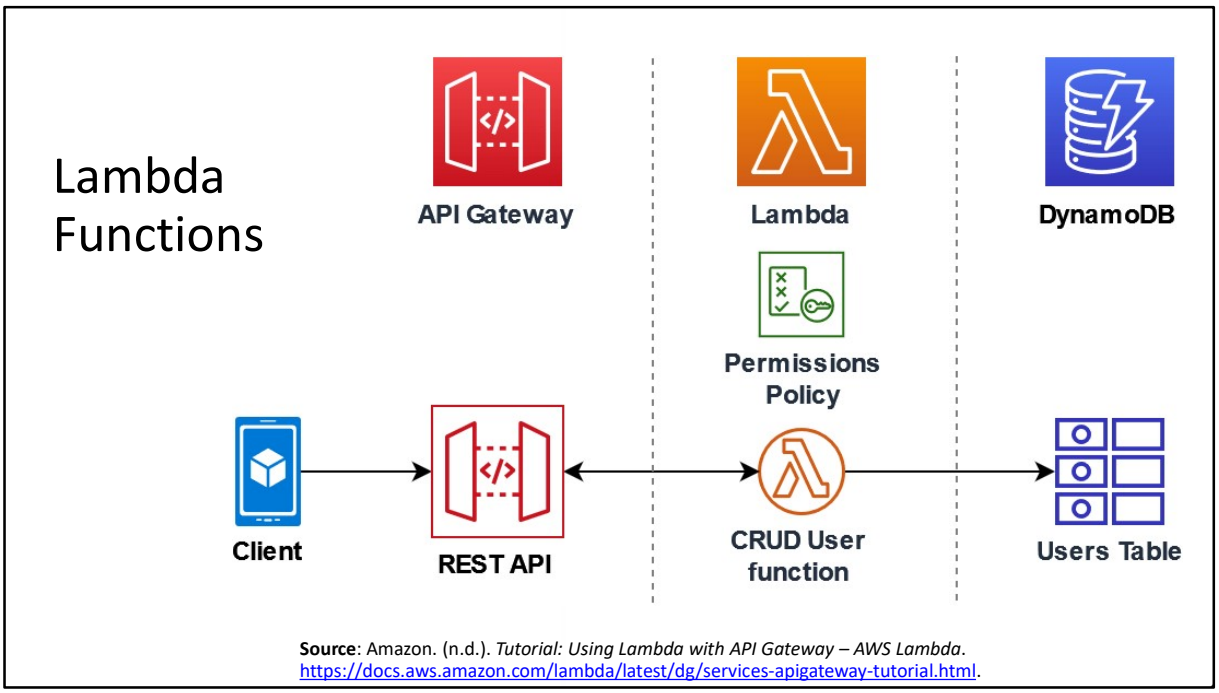
To see the Docker images stored on your local machine, [SPACE] you just enter the command ‘docker images’ and details are spit out. To see the images required by the services in the docker-compose.yml file, [SPACE] you run the command ‘docker-compose images’ as seen here. It gives you the container name, repository, ID, and other information. Details about currently running containers [SPACE] can be seen by running the ‘docker ps’ command, as seen here. There’s lots of juicy data for you there.



So what do we mean by “serverless”? Serverless computing is a cloud computing model in which scaling and provisioning is performed automatically by the cloud provider. Serverless computing provides a pay-as-you-go model, which aids in scalability and is especially helpful for startups, since you only pay for what you use. Serverless providers also offer high availability and fault tolerance, as well as load balancing, to ensure a fast and reliable experience for users. Some examples of serverless platforms include AWS Lambda, Google Cloud Functions, and Azure Functions.

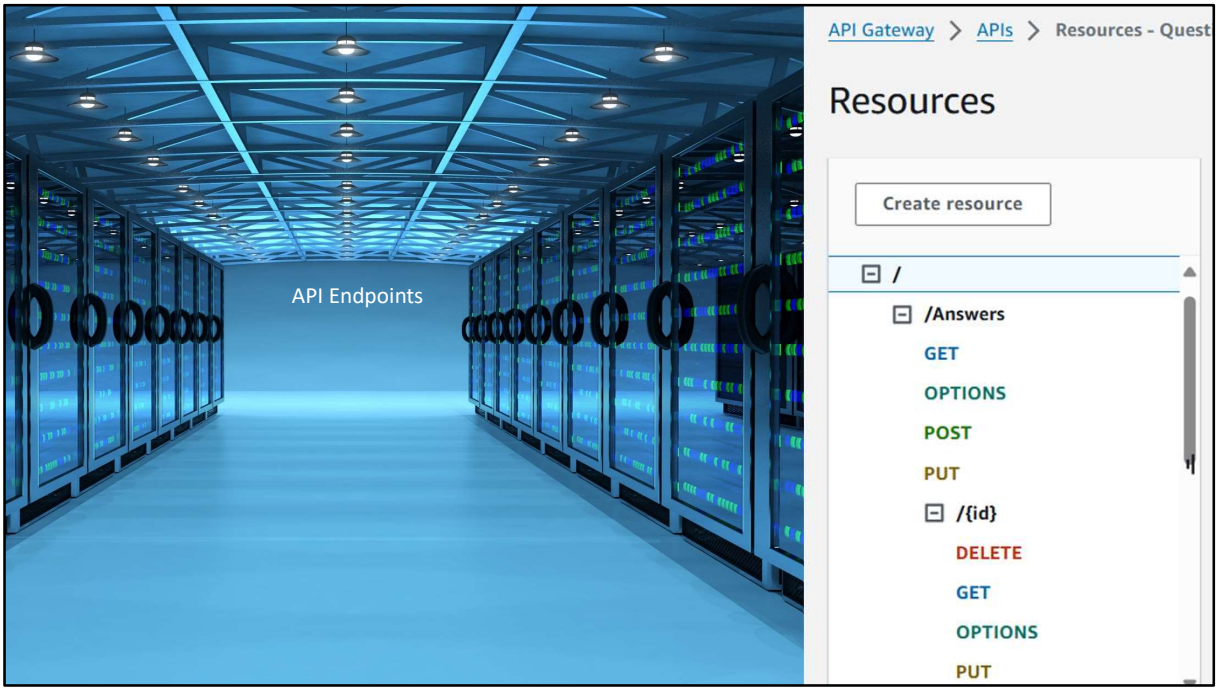
Amazon S3, or Simple Storage Service, is an object storage by Amazon Web Services (AWS) and provides scalable, high-speed, cloud storage using buckets. S3 also provides a minimum 99.99% [SPACE] availability and [SPACE] durability, as well as [SPACE] data encryption for data at rest and in transit. So, for example, the containerized frontend of an application can be uploaded to an S3 bucket and served to customers via a URL.
Now, on the other hand [SPACE] local storage is limited by its physical hardware and must be purchased upfront. Local storage must also be self-managed, including encrypting sensitive data. S3 provides a much better [SPACE] solution to traditional local storage, as it resolves these limitations, provided you have an internet connection.

A serverless API uses a pay-as-you-go model that offers automatic provisioning and scaling, as well as high availability and fault tolerance. Developers don't have to worry about [SPACE] server management, leading to faster development and deployment and happier [SPACE] software developers.
AWS Lambda used in conjunction with [SPACE] API Gateway provides a platform for creating serverless APIs. API Gateway routes incoming HTTP requests to the appropriate backend service or Lambda function based on the defined API endpoints and methods. These Lambda functions are triggered by these requests, as well as events like DynamoDB updates or S3 uploads, for example. The Lambda function processes incoming events, performs the core application logic, and returns a response to the client via API Gateway.



Source: Amazon. (n.d.). *Tutorial: Using Lambda with API Gateway – AWS Lambda*. https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway-tutorial.html.

Looking at this image from Amazon, we can see that a web browser (the client) makes a request to API Gateway's REST API, which passes the request to AWS Lambda. Lambda follows a permissions policy that defines the allowable CRUD functions (CREATE, READ, UPDATE, and DELETE). Lambda executes the appropriate code, communicating with the corresponding DynamoDB table to perform the requested operation. A response is then passed back down the pipeline and returned to the client.



The image on the right-hand side shows a few API endpoints manually defined in API Gateway, with each endpoint using an HTTP method: GET, POST, PUT, DELETE, or OPTIONS. These methods are used to access the resource path for the Answers table.

Lambda Function

index.mjs | Environment Var | Execution results

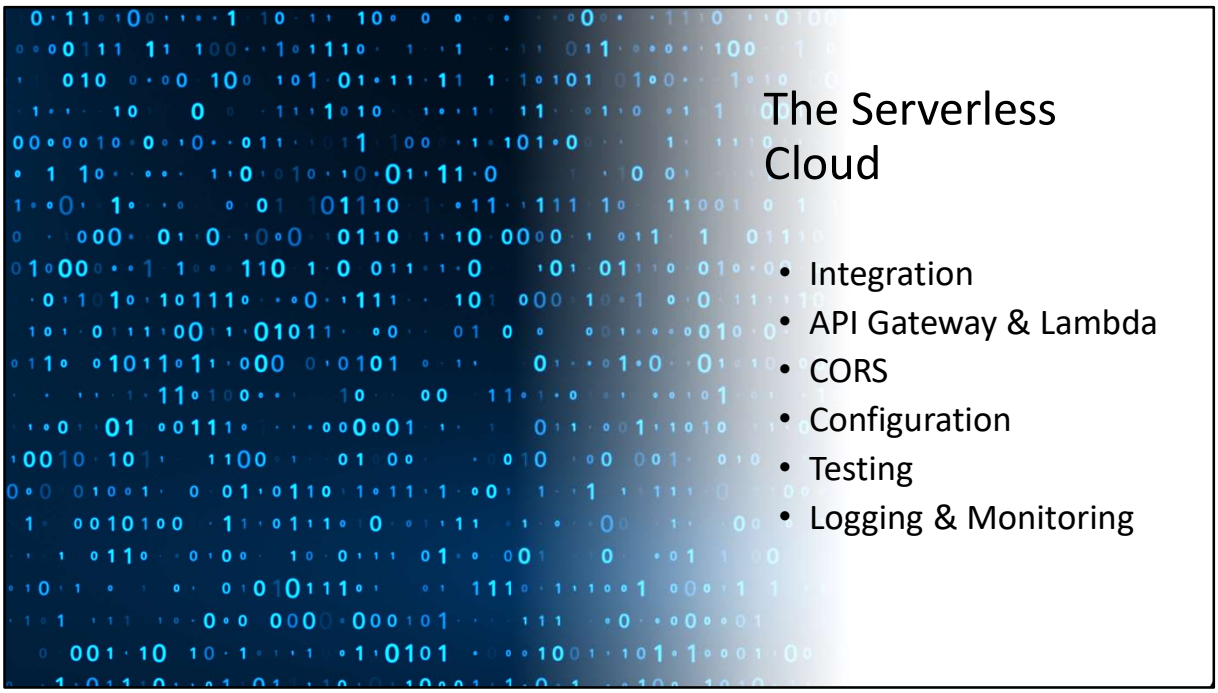
```
export const handler = async (event) => {

  const response = {
    // HTTP OK Success
    statusCode: 200,
    // Create some text to send back to the client
    body: JSON.stringify(event),

  };

  // log the event to the logger
  console.log("Event: \n" + JSON.stringify(event));
  // log the response to the logger
  console.log("Response: \n" + JSON.stringify(response));

  return response;
};

```

Test

Event JSON

```
{
  "resource": "/Answers",
  "httpMethod": "GET",
  "queryStringParameters": {},
  "multiValueQueryStringParameters": {}
}
```

Each Lambda function consists of an index.mjs file defining the response headers, the asynchronous handler method, and other code. AWS Lambda also allows creating tests [SPACE] to mimic requests. This specific test performs a GET request from the /Answers resource.



So how do we integrate the frontend with the backend? Well, API Gateway can be configured to expose the Lambda functions as HTTP endpoints and expose its own endpoints and methods. CORS (cross origin resource sharing) is configured in API Gateway to allow the frontend to interact with the backend. Configuration files can then be used to manage different environments (like development and production).
Then, integration of the frontend and backend can be tested to ensure proper communication and operation. Logging and monitoring tools can be used to debug and track API usage and analytics.

The Serverless Cloud

Database

- MongoDB
- DynamoDB
- Permissions Policy

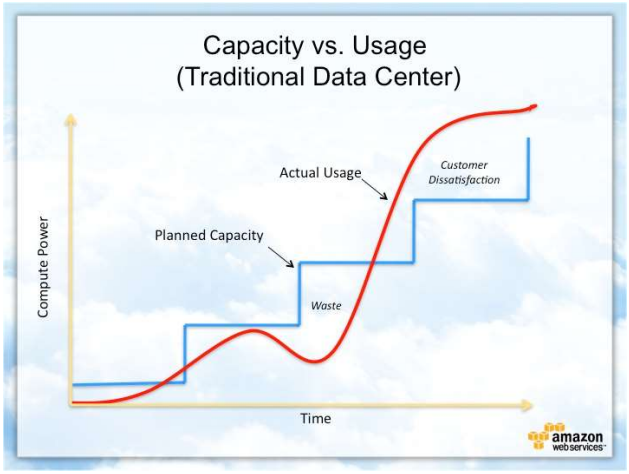
```
{
    "Version": "2012-10-17",
    "Statement": [{
            "Effect": "Allow",
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:GetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchWriteItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem"
            ],
            "Resource": "arn:aws:dynamodb:eu-west-1
        },
```

Source: Amazon. (n.d.). *AWS Lambda: Allows a Lambda function to access an Amazon DynamoDB table*. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_examples_lambda-access-dynamodb.html.

Earlier I mentioned [SPACE] MongoDB. MongoDB is a document-based NoSQL database that uses documents with varying schemas, or structures. MongoDB is commonly migrated to a cloud database like [SPACE] DynamoDB, which acts as a key-value store and optionally a document-based database, as well.
[SPACE] [SPACE] Here we can see a typical permissions policy from [SPACE] Amazon that gives access to DynamoDB methods. You can see various allowed actions here.

Cloud-Based Development Principles

- Elasticity
- Cloud Advantage
- Waste
- Customer Dissatisfaction
- Cloud Computing Solution
- Zero Waste

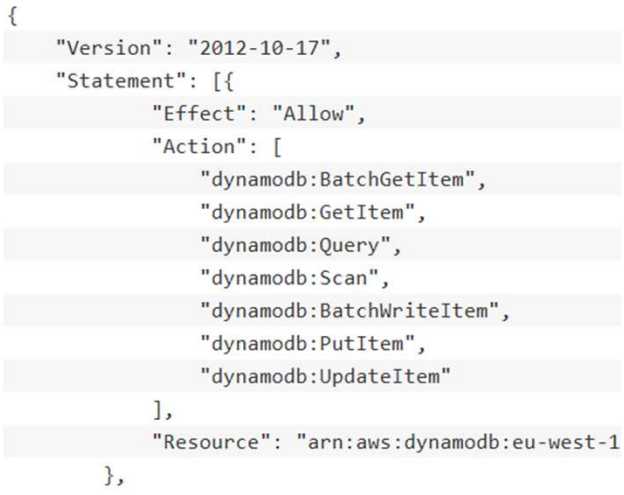


Elasticity refers to the ability to dynamically adjust resources in response to varying demand. The advantage for the [SPACE] cloud model is that there are no upfront costs or manual management needed as your hardware scales.
[SPACE] This image of a traditional data center shows planned capacity (the blue line) has "jumps" indicating more hardware being purchased. The actual resource usage (the red line) shows that much of the resources go unused, leading to waste [SPACE]. And sometimes, there are not enough resources to go around, leading to [SPACE] customer dissatisfaction and other issues. [SPACE] Cloud computing offers the best solution and only provides the resources you need at any given moment, whether traffic goes up or down. In other words, [SPACE] zero waste.

Securing Your Cloud App

Security

- IAM
- Policies
- Roles
- Permissions



```
{
    "Version": "2012-10-17",
    "Statement": [{
            "Effect": "Allow",
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:GetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchWriteItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem"
            ],
            "Resource": "arn:aws:dynamodb:eu-west-1
        },
```

To prevent unauthorized access, [SPACE] Identity and Access Management, or IAM, can be implemented using users, groups, and permissions. [SPACE] Policies can be created and attached to various [SPACE] roles you create. These policies are JSON documents that specify [SPACE] permissions, which define the actions allowed for specific resources. [SPACE] Again, looking at this image of a JSON policy, we can see the different DynamoDB actions that are allowed for the resource listed near the bottom here.
IAM roles can be assigned to API Gateway to allow invoking specific Lambda functions. This, along with the Principle of Least Privilege and HTTPS encryption, can be implemented to secure your connection. Roles can be assigned to Lambda functions to grant necessary permissions to access the database, and encryption can be used for data at rest and in transit. Public access to S3 storage access can be completely blocked as well.

CONCLUSION

To wrap up:

- Migrating an app to the cloud provides scalability, reliability, security, automatic provisioning, and a cost-efficient pay-as-you-go model.
- Docker helps by using containerization to split your app into separate microservices that can be independently developed and deployed; although, API endpoints and IAM roles and policies must be manually configured.
- API Gateway processes incoming HTTP client requests and passes them to AWS Lambda, which follows a permissions policy to allow or deny CRUD functions for DynamoDB, which processes the CRUD function and sends a response back down the pipeline to the client.

I hope this helps. Thank you for your time.

To wrap up:

- [SPACE] Migrating an app to the cloud provides scalability, reliability, security, automatic provisioning, and a cost-efficient pay-as-you-go model.
- [SPACE] Docker helps by using containerization to split your app into separate microservices that can be independently developed and deployed; although API endpoints and IAM roles and policies must be manually configured.
- [SPACE] And API Gateway processes incoming HTTP client requests and passes them to AWS Lambda, which follows a permissions policy to allow or deny CRUD functions for DynamoDB, which processes the CRUD function and sends a response back down the pipeline to the client.

[SPACE] I hope this helps. Thank you for your time.

CONCLUSION

Hold for applause… ☺

Hold for applause…