**Project Two Conference Presentation: Cloud Development**

Guilherme Rigaud

04/01/2024

https://youtu.be/Qa4wSpSX-lg

**Narration of Slides**

## Slide 1

Hello and welcome, everyone. My name is Guilherme and today I will discuss how I've transitioned from a monolithic full stack application to a serverless architecture using AWS to improve scalability and operational efficiency.

This presentation will cover the details of migrating to a serverless framework and the strategic benefits I've achieved. Throughout the presentation I will be using some analogies and real-world examples which helped me understand how each technology comes together in Cloud Development. In the next slide I will go over the goals of this presentation more in-depth.

## Slide 2

Our goal today is to simplify the cloud development process and show you the real-world applications and benefits of moving to a serverless architecture.

We'll start by discussing our shift from an old-school full stack application to a modern AWS serverless architecture. I'll explain why we made this change, what steps it involved, and how it's better preparing us for the future as a tech company.

Moving to the cloud can be tricky, and I'll share the major hurdles we encountered, the decisions we had to make, and the solutions we found.

While moving to the cloud, the specific migration models done in this case was “lift and shift” which involves moving application from a physical infrastructure to a cloud infrastructure, and “refactor” which involves modifying the application to better support the cloud environment.

Then, we'll go over the real benefits we've seen from this migration. From big improvements in how efficiently we operate to better customer satisfaction, these benefits have shown us that we made the right move

## Slide 3

Let’s dive into a crucial first step: containerization. Think of containerization as the packing process before a big move. Just like you’d pack items into boxes to make sure they’re easy to transport and arrive safely, we package our applications into containers

The tool that enables this is Docker. It packages an application and its dependencies into a Docker container, which is a standardized unit of software. This is similar to how you might use a travel adapter to ensure your phone charger works in any country

Let me share a bit of my first-hand experience with Docker. Initially, I used Docker to run MongoDB—a database we all know. Instead of installing MongoDB traditionally, which could conflict with other software or depend heavily on the specific setup of my machine, I used Docker. By running MongoDB inside a Docker container, I ensured that my database environment was consistent and portable.

Here's how simple it was: I pulled a pre-configured MongoDB image from Docker's public registry using the command docker pull mongo, then ran it with just one more command. This approach not only saved me installation time but also eliminated any configuration headaches

But what if our applications are more complex? What if they need multiple services to run? In my case my application had a back-end and a front-end service. This is where Docker Compose comes into play. It lets us define and orchestrate multi-container Docker applications using a simple YAML file.

Imagine orchestrating a symphony, where each musician is a container. Docker Compose conducts these musicians so they play beautifully together.

Now that our applications are neatly packed and orchestrated, how do we take them to the next level? This is where serverless architecture enters. In the next slide, we'll see how transitioning these containerized applications to a serverless setup can enhance scalability and reduce operational costs even further

## Slide 4

When we talk about 'serverless', we're referring to a cloud computing execution model where the cloud provider dynamically manages the allocation and provisioning of servers.

A good example I like to use is to Imagine you're creating a show, and you want to delight the audience—your app's users. With serverless, you have an amazing behind-the-scenes crew (the serverless architecture) taking care of the lights, sound, and special effects (databases, storage, and logic)

You, the showrunner, can concentrate on the script and the performers, making the user experience as engaging as possible. And the best part? If more audience members show up, the crew can instantly manage it, no sweat. That's the beauty of serverless; it's ready to handle a cozy small-theater performance or a blockbuster on Broadway night—automatically and without fuss

In essence, serverless helps you build your app like a set of LEGO blocks, each piece doing its job, working together seamlessly. And when it's showtime, everything just works in harmony, giving your users a performance they'll applaud

The benefits are substantial: it's cost-efficient, as you only pay for what you use; it scales with your application's needs; and it allows developers to focus on writing code rather than managing server infrastructure

With the backend managed by serverless configurations, our next focus is on data handling and storage. Moving on, we will explore how AWS S3 fits into the serverless ecosystem, enhancing our application’s ability to store and retrieve data efficiently as demand changes

## Slide 5

**Understanding AWS S3:**

AWS S3, or Simple Storage Service, is like a huge online drive. You can store all kinds of data in it, retrieve it whenever needed, and it’s available all over the internet.

In serverless architectures, S3 is often used to store files, serve multimedia, and manage uploads without maintaining a dedicated server for storage.

**S3 vs. Traditional Storage:**

Accessibility: Unlike traditional storage that might be on a single server or network, S3 data can be accessed from anywhere, anytime. It's truly global.

Scalability: S3 can handle any amount of data, from a single byte to exabytes, without needing prior setup. This contrasts sharply with traditional systems where you might run out of space.

Cost-Effectiveness: Pay as you go—only pay for the storage you use with S3. Traditional storage requires significant upfront investment for infrastructure and ongoing maintenance costs.

Reliability: S3 is designed to deliver 99.999999999% durability over a given year. This means your data is extremely safe and far less likely to be lost compared to traditional hard drives or servers.

## Slide 6

As we extend our serverless architecture, a critical component is the API — the interface between our frontend and the cloud functions. AWS API Gateway simplifies this by managing all HTTP requests and responses. It’s like having a bartender who ensures that every customer order gets the right response from the chefs.

Using serverless APIs offers multiple benefits. First, they're simple and scalable. The API Gateway handles incoming traffic, scales it automatically—up or down—depending on demand. This means you don't have to worry about server maintenance or downtime

They're also cost-effective. Imagine this: you only pay for the API calls you make. No need to spend money on servers that are sitting idle

Lastly, security and compliance are built-in. API Gateway includes mechanisms to protect against unauthorized access and ensures your data handling meets regulatory standards

For instance, in one of our projects, we set up a Lambda function named EchoFunction to respond to HTTP requests via API Gateway. We created this API using a simple setup in the AWS console, selecting methods like GET to handle specific types of requests

When we tested this with a custom input, say a user's name, the Lambda function processed it and immediately returned a personalized greeting. This quick interaction showcases the power of serverless APIs in providing dynamic content to users efficiently.

Now that we have our serverless API efficiently handling requests and integrating with Lambda, the next step is to manage how these functions interact with our database. This brings us to AWS DynamoDB, which we will explore in the next slide. DynamoDB's integration further streamlines our architecture by providing a robust, scalable database solution that complements our serverless setup.

## Slide 7

Let’s explore DynamoDB, AWS's NoSQL database service, designed for modern applications that require consistent, single-digit millisecond latency at any scale. It's like having an ultra-fast, always available filing cabinet that can handle anything from a single document to libraries worth of information without breaking a sweat

DynamoDB is a perfect match for serverless architectures because it automatically scales up or down based on the application's needs. This means you don’t have to manage the database servers or worry about provisioning capacity ahead of time—it's all handled for you, seamlessly

One of the powerful features of DynamoDB is its ability to integrate tightly with AWS Lambda. This integration enables real-time data processing which is crucial for dynamic, responsive applications. For instance, when data in DynamoDB changes—like a new user sign-up or a product update—DynamoDB Streams can trigger Lambda functions automatically. These functions can then execute any necessary logic, such as sending a welcome email or updating related records

Let's walk through a simple example from the tutorial to illustrate this. We started by creating a DynamoDB table named 'Question' to store data about user inquiries. Each item in this table has a unique ID and attributes like the question itself and metadata about votes

Adding data to this table is straightforward. You simply insert items into the table using a JSON format, and DynamoDB handles the rest. For instance, inserting a question might look like the screenshot above in JSON.

## Slide 8

Now let's discuss why and how we transitioned from MongoDB to DynamoDB using real-world explanation and examples.

**MongoDB:** This is a document-based database that excels in handling complex data structures and supports rich queries. It's like having a flexible, dynamic closet where you can throw in various types of clothes without worrying about organizing them perfectly

**DynamoDB:** On the other hand, DynamoDB uses a key-value and document model but is designed for high performance and scalability, focusing on quick access through primary keys. Imagine a meticulously organized set of drawers where each item has its specific place for quick retrieval

A specific difference between MongoDB and DynamoDB is Query transformation.

**MongoDB Queries:** Typically, we use commands like **find()** to fetch data based on complex criteria. These queries are powerful but can become resource-intensive as data grows.

**DynamoDB Queries:** DynamoDB simplifies querying with direct access patterns using **Query** and **Scan**. While this model is more straightforward and efficient at scale, it requires a different approach to data modeling and interaction

I included a practical example of PutCommand . We set up a DynamoDB table for storing user questions, where each question must be quick to access and easy to manage. Right here you can see how a typical insertion script looks

With our data now efficiently migrated and modeled in DynamoDB, the next step involves integrating this data with our serverless backend. In the upcoming slide, we'll explore how serverless APIs like AWS Lambda facilitate seamless interactions between our frontend and this new database structure

Once the data is in DynamoDB, any changes to these items—like a new answer added to a question—can trigger a Lambda function. This function could, for instance, recalculate the total votes or update a leaderboard, and the results are immediately available to all users.

Now that we understand how DynamoDB can serve as a robust backbone for managing data in a serverless environment, let's consider a more complex scenario. In the next slide, we will discuss migrating from a traditional database system, MongoDB, to DynamoDB. We'll look at the differences in data modeling and how to transition smoothly.

## Slide 9

In a serverless architecture, the connection between the frontend, which is what users interact with, and the backend, where all the processing happens, is crucial. Let’s recap how AWS services make this connection seamless and efficient

AWS API Gateway: Think of API Gateway as the front door for all the requests that come to your application. It ensures that every user's request is directed to the right service, handling all the complexities of managing these interactions

AWS Lambda: Lambda is like a personal assistant for your application, ready to perform any task you assign it, whenever it's needed. You don't have to keep it on all the time; it comes into action the moment it’s called upon and then goes back to rest.

Setting Up API Gateway: The first step in integrating our frontend with the backend is to set up API Gateway. Here, we define endpoints—think of them as individual addresses—each linked to different backend functions based on what needs to be done, like retrieving data or processing a payment

Configuring Lambda Functions: Next, we create Lambda functions. Each function is tailored to perform a specific backend task efficiently. For example, one Lambda function could handle user logins, while another manages user queries

Ensuring Seamless Communication: Finally, we use API Gateway to connect these Lambda functions to the frontend. This setup not only secures the communication but also ensures that data flows smoothly back and forth as needed, without any delays that are typical with traditional server setups

Let’s consider a practical scenario. We set up an API called ‘Questions & Answers’ using API Gateway. We then linked various HTTP methods like GET and POST to different Lambda functions designed to handle specific tasks such as fetching questions or posting answers

For instance, when a user submits a question through our frontend, API Gateway receives this POST request and triggers a Lambda function called UpsertQuestion. This function processes the input and updates our database in real-time, and the user sees confirmation instantly on their screen

With our frontend and backend perfectly in sync through serverless APIs, our next focus is on ensuring that this setup is secure. In the next slide, we’ll explore how AWS Identity and Access Management (IAM) plays a pivotal role in fortifying our serverless architecture, protecting it from unauthorized access and potential threats

## Slide 10

As we move deeper into serverless architectures, securing our applications becomes paramount. AWS Identity and Access Management, or IAM, is the toolkit we use to ensure that our resources are accessed securely and appropriately

IAM allows us to meticulously manage who can access what in our applications. It's like giving a key card to each member of your team, where each card opens only specific doors. For instance, you can set which users or services can interact with your APIs and which can execute your Lambda functions

AWS API Gateway plays a critical role here, acting as a robust gatekeeper between our front end and Lambda functions. It ensures that all connections are secure, managing traffic to only allow authorized requests through.

Consider our 'Questions & Answers' API: API Gateway manages incoming requests, directing them to the correct Lambda function, like GetSingleRecord for retrieving data or UpsertQuestion for updating our database. Each of these functions is triggered securely, with IAM ensuring that only legitimate requests processed by API Gateway can activate them.

Now that we've secured our serverless infrastructure, let's look at how these technologies not only protect but also enhance our applications through principles like elasticity and pay-for-use. The next slide will delve into these fundamental principles of cloud development, which help optimize costs and scalability

## Slide 11

Let's recap how we've integrated key technologies to create a seamless, efficient serverless architecture that scales and secures dynamically.

A user logs into our application, which is served statically from AWS S3. The user interface they interact with is rendered in their browser, featuring responsive design loaded directly from S3 buckets.

The user types their question into the UI and clicks 'Submit'. This action triggers a POST request to AWS API Gateway, effectively 'knocking on the front door' of our backend services.

API Gateway receives the POST request and checks the configured routes. It identifies that this request should trigger the UpsertQuestion Lambda function, which is set up to handle new questions and updates to existing ones

IAM roles associated with the API Gateway ensure that only authenticated users can submit questions. The request includes the user's authentication token, which API Gateway uses to verify permissions before proceeding.

The UpsertQuestion Lambda function is invoked. It takes the data from the request, which includes the question text and user metadata, and processes it. This might involve sanitizing the input, validating the data, and then storing the question in a DynamoDB table designed to hold questions

Once the question is stored, Lambda prepares a response. This can include a success message and any additional data needed by the frontend, such as the question ID. Lambda sends this response back through API Gateway to the user's browser

All of this is possible due to the steps we've taken to transition our traditional full-stack application to the cloud, in the next final slide we will look at an overview of what was done.

## Slide 12

Now let's discuss everything done to achieve a serverless application, first we began with

Docker for Consistency and Portability:

Utilized Docker to containerize MongoDB, ensuring consistent environments across all platforms and simplifying deployment processes.

Commands like docker pull mongo and docker run enabled easy scalability and movement across systems without reconfiguration.

Docker Compose for Enhanced Orchestration:

Deployed Docker Compose to manage multi-container setups, connecting backend components like Node.js and MongoDB into a unified network.

Established internal communications without exposing ports externally, enhancing security.

AWS S3 for Robust Storage:

Hosted static assets (HTML, CSS, JavaScript) on AWS S3, ensuring efficient delivery and scalability.

Configured buckets for web hosting, demonstrating S3's capability to serve content directly to browsers.

API Gateway for Dynamic Interactions:

Used API Gateway to manage and route HTTP requests to Lambda, acting as a secure entry point that enhances scalability and responsiveness.

Integrated seamlessly with Lambda to trigger backend processes based on user requests, ensuring efficient real-time interactions.

AWS Lambda for Serverless Processing:

Implemented Lambda for executing backend logic in response to API triggers, directly interacting with DynamoDB for data operations.

Benefited from a fully managed, event-driven architecture that offloads server management, allowing focus on core application logic.

IAM for Comprehensive Security:

Configured IAM roles to strictly manage access permissions across AWS services.

By integrating these technologies—Docker for containerization, S3 for storage, API Gateway for routing, Lambda for processing, and IAM for security—creates a robust, scalable, and secure serverless application.