AWS Serverless Architecture Solution Design Memo of RHIT Greenhouse IoT System for Microchip

Scott Cai

02/2021

Summary

This memo is used to record and explain the final AWS architecture design of greenhouse IoT system at Rose-Hulman Institute of Technology. Our client, Microchip, would like us to build and deliver a scalable Internet of Thing (IoT) environment on campus, taking advantage of which for their product marketing. This memo is going to mainly discuss about the architecture solution built on Amazon Web Services (AWS), along with some other software frameworks I used to develop the web application. We are going to deep dive into the solutions details, design decisions, as well as some other options that may be helpful.

About the project

The IoT network system collects, organizes and records environmental data in the greenhouse, generating real-time matrix graph to help the client make better data-driven decision. Client can access and monitor the greenhouse remotely across the web application through browsers. With this in-house data collection IoT system, our client can focus on greenhouse environment itself off campus rather than on-site tests and measurement, especially during this special COVID-19 period.

General data flow

Sensor data are transferred to the MQTT client on AWS with topic name "/GreenHouse". After that, they are classified according to their measurement attributes and being inserted into different tables in the AWS Timestream database with certain AWS IoT rules, while abnormal messages will be marked error and send an SNS notification to specified email address. The static web application is designed with Material-UI and built in React.js, hosted in AWS S3 bucket and cached with Amazon CloudFront CDN. When the user clicks on the web page, the web app will fetch the most recent sensor data at a fixed certain time interval from the REST APIs in the AWS API gateway with Ajax calls. Every time the APIs are getting called by GET method, it will then trigger a lambda function to read required data in the database and return back to the API as a JSON response. Then, the latest data point will be drawn on the UI, in which case, users are able to see the real-time sensor data streaming from the onsite greenhouse even though they are at any corner in the world.

GitHub Repository

AWS-IoT-Smart-GreenHouse-Web-UI Git Repo

Design Principles

Developing a software system is more or less like constructing a building. Without a solid foundation, the structural problems can determine the integrity and function of the whole building. When architecting a solution on AWS, there are a few vital aspects and concerns that might influence the system either currently or in the future. Thus, taking our client's feedbacks into consideration, I decided to follow the principles below to build our AWS solutions.

1. High Scalability

One of the key requirements from our client is that the whole project needs to highly scalable, including the ability to increase sensor amount, support further development, and continuously improve supporting processes while delivering business value in the future. It needs to take precautions for the increasing visits and connections from both user side and hardware sensor side, without harming the whole architecture. When necessary, we need to loose the coupling and decouple the infrastructure in our solution.

2. Reliability

Reliability encompasses the ability of a workload to perform its intended function correctly and consistently as expected. The solution is supposed to anticipate failures through its total lifecycle, and automatically recovers from the failure. Besides, it also needs to take operational failures into account. Since any neglect in test procedures could lead to a significant impact on the ability to deliver a reliable system, we shall consider thoroughly about the possible pressures from both user-end and system level to satisfy the requirement for sufficient networking and compute capacity, even in an on-premises environment. In another word, remember to design for failure for a better resiliency.

3. Performance Efficiency

Performance represents how efficient the system to use its computing resources to meet the requirements, and how efficient it can maintain as the demand the technologies evolve in the future. Both time and space efficiency stand for the overall performance of the system solution. To build a high-performance architecture, we need to review our choices from the high-level design to the selection and configuration of resource types, and pick the optimal solution for the particular workloads. In practice, we have to keep making trade-offs in our architecture to improve performance.

4. Cost Optimization

Although the daily AWS maintenance is not included in the total budget of the project, we would like to implement the most cost-efficient solution to run the system and deliver its business value at the lowest price point. It is very important to carefully handle those undifferentiated heavy lifting, which is very likely overprovisioned and spends the most of the money in the cloud services.

*5. Security

Since there is no way for user to control any fan or light in the greenhouse from our web app in the current stage, security tends to be more important in protecting data in database, preventing DDOS attack from application layer with AWS Firewall, deploying security group policies and network access control lists (NACL) in virtual private cloud (VPC) service against malicious or other suspicious visits. Adequate attention should also be given to cross-site scripting (XSS), with which experienced hackers could change the data shown on the public web page easily. I also utilized a lot of IAM roles to limit interactive services access from each other to especially prevent any unexpected attack on databases and add an extra layer of security.

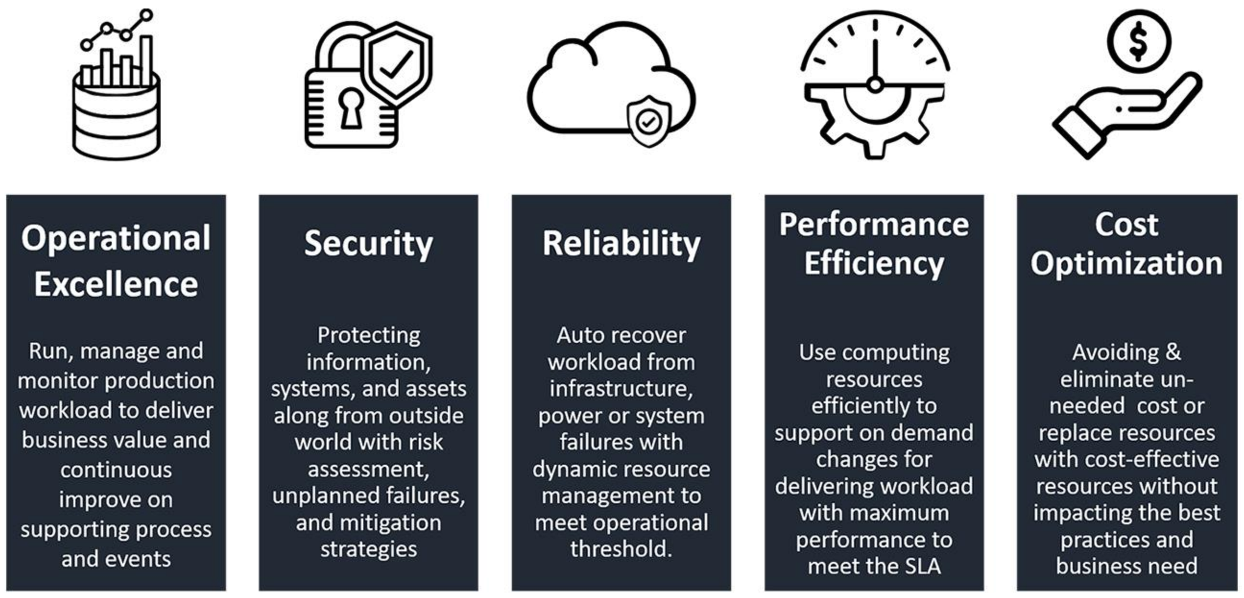

Operational Excellence
Run, manage and monitor production workload to deliver business value and continuous improve on supporting process and events
Security
Protecting information, systems, and assets along from outside world with risk assessment, unplanned failures, and mitigation strategies
Reliability
Auto recover workload from infrastructure, power or system failures with dynamic resource management to meet operational threshold.
Performance Efficiency
Use computing resources efficiently to support on demand changes for delivering workload with maximum performance to meet the SLA
Cost Optimization
Avoiding & eliminate un-needed cost or replace resources with cost-effective resources without impacting the best practices and business need

Figure 1 Five pillars of AWS design principles

Literally, there is a sixth principle that we would like to take advantage of all the available services on AWS and avoid reinventing the wheels.

Design Decisions

1. Backend Architecture

Generally, we would like to design a high available architecture and a fault-tolerant solution for our AWS backend side, while minimizing the costs as much as possible. There are several popular options for the application development – monolithic apps, microservices, and serverless architecture.

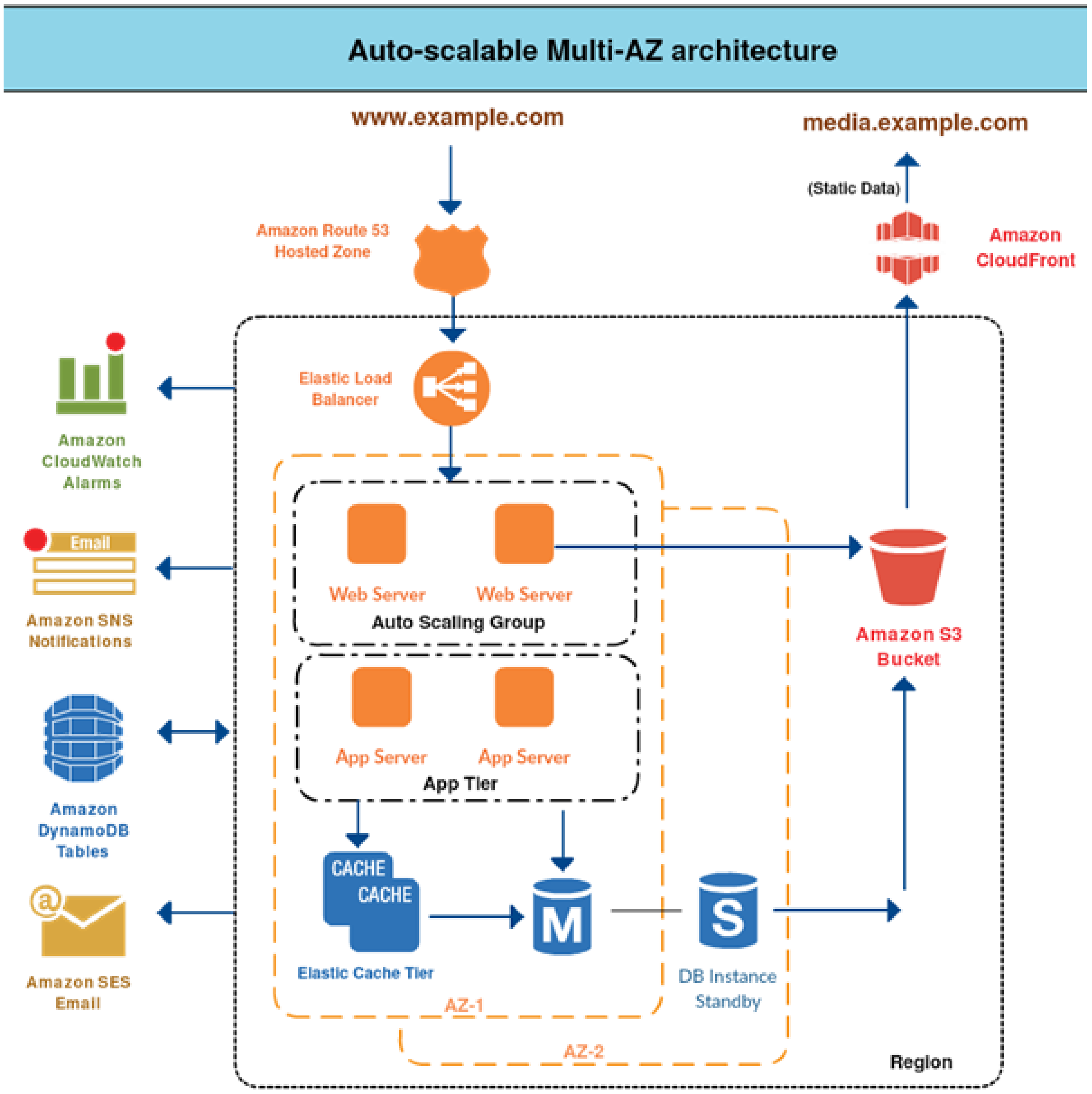


Figure 2 A recommended reliable multi-AZ architecture on AWS with auto-scaling

Usually, a traditional monolithic solution consists of a database, client-side user interface, and server-side application. All the software’s parts are unified and all its functions are managed in one place. On the other hand, traditional monolithic architecture is easy to get massive and cumbersome over time especially when dependencies increase and integrated development environment gets overload. We need to consider very carefully about the server load, write requests, and may even introduce a more robust caching engine to scale the server and increase performance a little bit. It is also very difficult to adopt new technologies as well as limits agility of the whole team.

While serverless architecture is an approach to building and running apps and services without the need for infrastructure management or worrying about provisioning and maintaining the servers. In fact, serverless doesn't really mean "no server." A third-party cloud service like AWS takes full responsibility for these servers instead. In this design, a set of AWS lambda functions are triggered and executed independently based on different events.

When using a serverless architecture, we can better focus on the product itself without worrying about server management or execution environments.

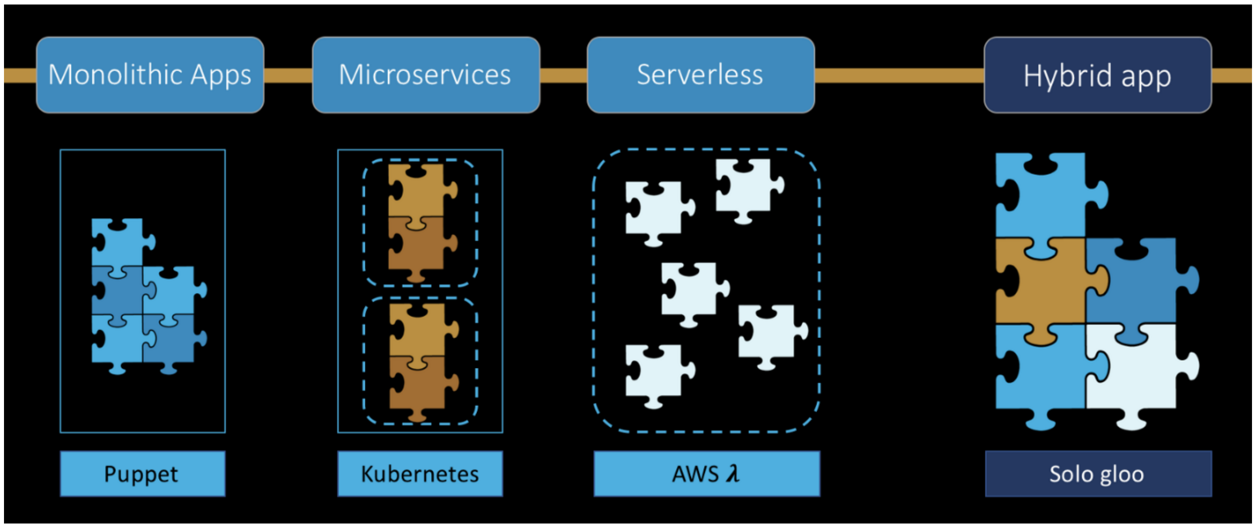


Figure 3 monolithic vs microservices vs serverless architecture

Next, we are going to analyze the pros and cons of building a serverless solution.

1) Pros - Scalability

Serverless app will automatically scale as the load or user base increases without affecting performance. It can handle a huge number of requests, whereas a traditional app will be overwhelmed by a sudden increase in requests.

2) Pros - Pricing

It is also cost-efficient as well, as its pay-per-use approach helps avoid unnecessary app development and deployment expenses. We only need to pay for the server's running periods when some scheduled events give rise to respective actions in the cloud.

3) Pros - Development

With serverless capabilities it becomes quite easy and more comfortable for coders to develop and deploy their applications. We no longer have to think about side activities like infrastructure maintenance or timely data syncing.

4) Pros - Latency

Due to the fact that all the data assets of a company are hosted on cloud-based servers, it allows an organization to allocate its resources on the nearest available for end-users servers. This ensures smooth connectivity and interoperability of data, without any time-critical failures and system downtimes.

5) Cons - Vendor lock in

As a result, changes to business logic are limited and migration from one vendor to another might be challenging.

6) Cons - Runtime constrain

A serverless model isn't suitable for long-term operations. Serverless apps are good for short real-time processes, but if a task takes more than five minutes, a serverless app will need additional FaaS functionality.

7) Cons - Cold starts

If a Lambda function hasn't run for a certain amount of time, it's instance gets removed from cache. The first time it gets triggered or if it hasn't been triggered in a while, it can take a couple of seconds longer than usual to run since it needs to be loaded in again.

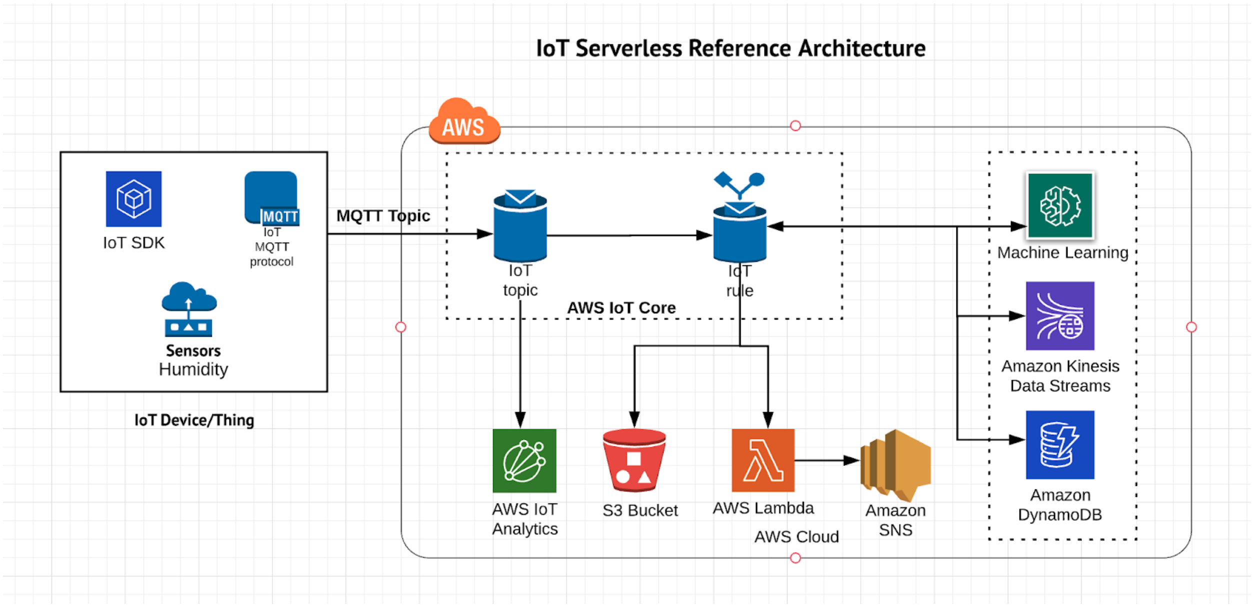


Figure 4 A reference IoT serverless backend architecture

Due to the lightweight programming model and operational ease of a serverless architecture, time to market is greatly reduced for new features, which is a key driver for many businesses. It also means prototypes can be quickly created for IoT solutions using Function as a Service (FaaS) for data processing to demonstrate a new technology to investors or clients. **With this serverless solution, there is no need for us to build or provision an EC2 instance**. It is apparent that serverless backend architecture provides more benefits especially for an IoT system.

2. Database

After solving the problem of backend architecture, we would like to evaluate and choose the right database for the type of data, access model and scalability we require to store the continuous data from the sensors in the greenhouse. The most popular options are either Amazon Relational Database Services (RDS) or DynamoDB.

Since we need to store a number of data from AWS IoT core every second, it sounds that relational databases are the best fits for these high-concurrency transactions. However, when data keeps increasing after several months or years, it will take a long time to scan over all the data rows, which leads to very weak distributed availability due to the poor horizontal scalability. Besides, due to the inflexible data model, it also has difficult schema evolution problem in the later time.

Therefore, the auto-scaling throughput ability shall be able to maximize the performance of DynamoDB. Since DynamoDB is a NoSQL database, we would have more flexible entry options for each different data dataset. It turns out that when the data storage scales up, we have to spend time figuring out a better caching strategy (like Elasticache using Redis, or DynamoDB Accelerator (DAX)) to avoid long-time scanning when looking for an data entry or inserting a new key or attribute into the tremendous data table.

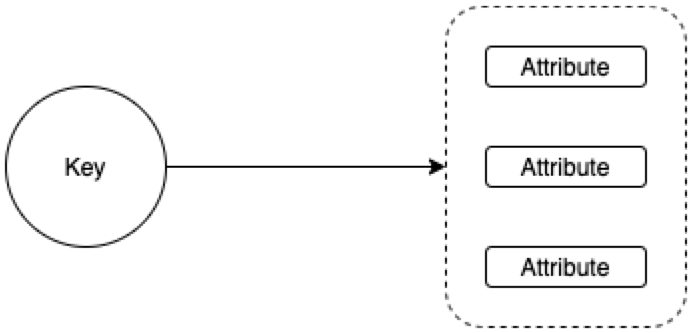


Figure 5 DynamoDB meta data format: one unique key identifies flexible amount of attributes

Timestream database, instead, is designed to store continuous measurements, especially suitable for continuous insertions into the database, for example from a temperature sensor. Each measurement has a name, value, grouping dimension and a timestamp. Due to the design focusing on the batch insertions, writing records into a Timestream database can be as fast as up to 100 records per request. While DynamoDB's batch inserts were sometimes throttled both with provisioned and on-demand capacity. It also has a magnetic memory store layer to backfill those old data if its age exceed the maximum retention time (in this case, we set up to 24 hours), to lower the cost as well as increase its read and write performance.

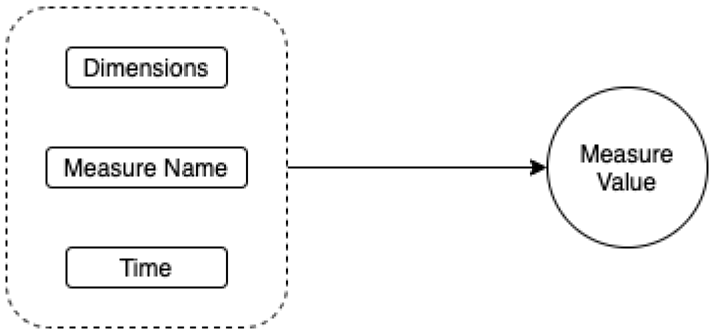


Figure 6 Timestream meta data format

Eventually, I decided to move the timeseries data to Timestream, while keeping a DynamoDB table for precomputing user facing data when necessary. I also enabled the read replica for better read performance and multi-AZ (Availability Zones) for remote disaster tolerance.

3. Front-end Delivery

After determining the server and database, we are going to looking for a solution to host our web application for the front end users. Amazon recommends their AWS Amplify console in the reference, with which we can host static sites with automatic deployment and Amazon Cognito.

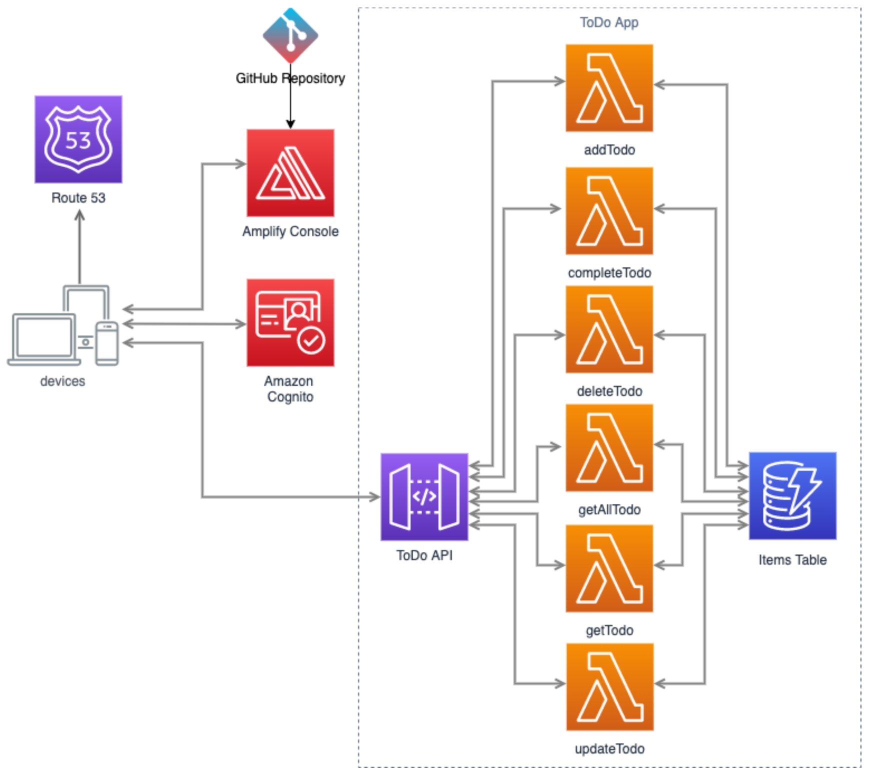


Figure 7 A reference for front-end static content hosting

It looks ideal and perfect for serverless solutions and would save a lot of time building and deploying the web app. However, it is very limited as well. To take advantage of the Amplify Console, we have figure out a way storing all the necessary data in DynamoDB, since this solution only integrates with S3 and DynamoDB. Plus, we would be constrained in amplify-cli while further development would have to be relied on AWS Amplify's updates if it lacks the functionality we need later on. Besides, its client-side libraries are very large, which could lead to problems for client-side browsers.

Thus, we had better take a step back, using S3 and CloudFront as a better solution for scalability. After the front-end React application is built, I uploaded it onto an Simple Storage Services (S3) bucket and deployed it as a static website. I also set up a content delivery network (CDN) service with the help of AWS CloudFront, caching the webpages into every edge locations around the world, helping with the load speed and reducing the latency when our client is travelling around the world for the trade show.

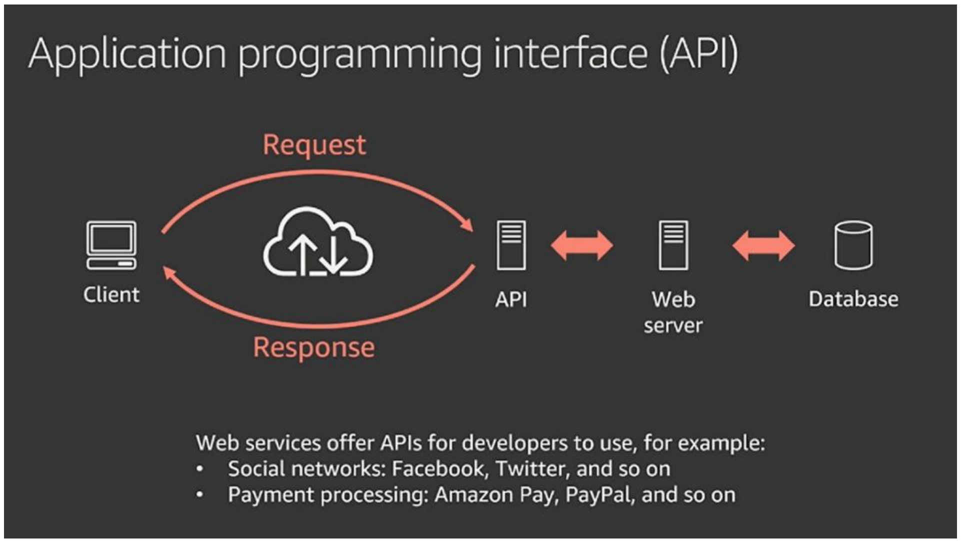
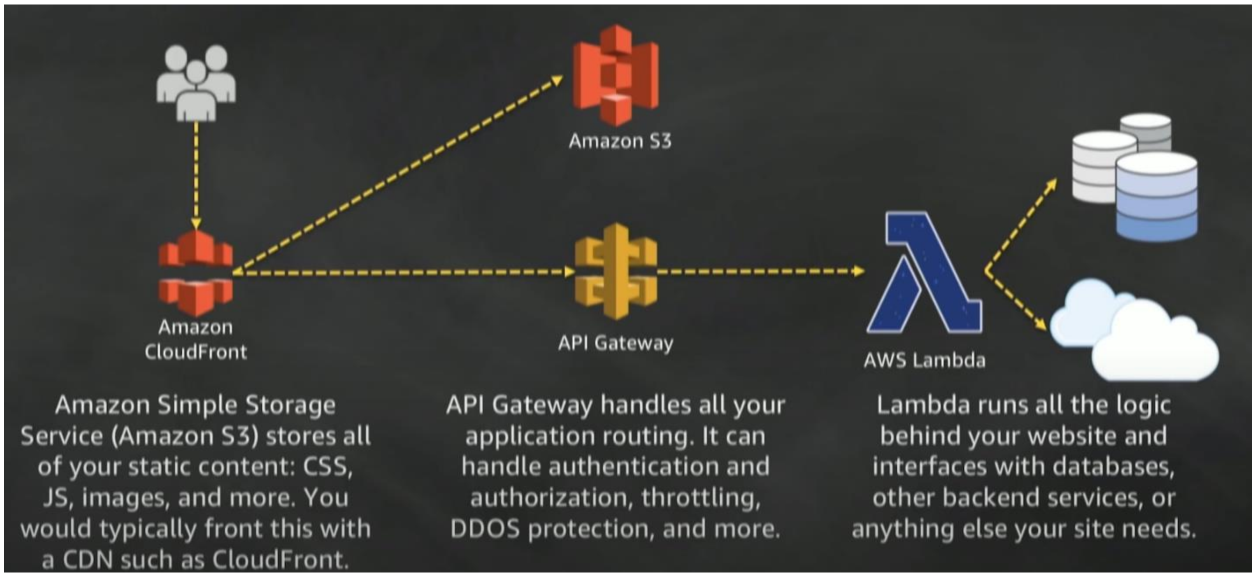

Application programming interface (API)
Request
Client
API
Web server
Database
Response
Web services offer APIs for developers to use, for example:
• Social networks: Facebook, Twitter, and so on
• Payment processing: Amazon Pay, PayPal, and so on

Figure 8 What is API?

With application programming interface (API) set up in API Gateway, the client computers are able to request the required data and fetch the responses found with lambda function in the databases. All we need to is to set up multiple RESTful APIs in the API gateway and ready to scale out to ease the database load. API Gateway handles all the application routing, throttling, authorization in our case, while Lambda functions run all the logic behind the web app and interfaces with databases, SNS, and other backend services. CloudTrail is used to monitor and log the performance of the APIs.



Amazon Simple Storage Service (Amazon S3) stores all of your static content: CSS, JS, images, and more. You would typically front this with a CDN such as CloudFront.

API Gateway handles all your application routing. It can handle authentication and authorization, throttling, DDOS protection, and more.

Lambda runs all the logic behind your website and interfaces with databases, other backend services, or anything else your site needs.

Figure 9 User Interface data access model

Overall, looking back at the design decisions we have made for the past few months in this project, we kept making **tradeoffs** for each part of the architecture. Some might be switching from one database to another; some might be taking advantage of more space for a better time performance; the others might be scaling out when necessary in order to save more costs. All in all, I hope this architecture solution is **scalable** and **reliable** for the future use.

Reference

Serverless Streaming Architectures and Best Practices
Deploy an End-to-End IoT Application
Monolith vs SOA vs Microservices vs Serverless Architecture
Serverless vs. Microservices architecture: what does the future of business computing look
Microservices vs. Serverless
A Beginner's Guide to Scaling to 11 Million+ Users on Amazon's AWS - High Scalability
Architecting for Reliable Scalability | AWS Architecture Blog
How to determine if Amazon DynamoDB is appropriate for your needs, and then plan your migration | Amazon Web Services
Serverless Cost Savings: How Much Could You Save?
ARCHIVED: Performance Efficiency Pillar
Design patterns for high-volume, time-series data in Amazon DynamoDB | Amazon Web Services
Amazon Timestream vs DynamoDB for Timeseries Data

AWS re:Invent

AWS re:Invent 2019: [REPEAT 3] Serverless architectural patterns and best practices (ARC307-R3)
Introduction to AWS Lambda & Serverless Applications
AWS re:Invent 2018: Amazon DynamoDB Deep Dive: Advanced Design Patterns for DynamoDB (DAT401)
AWS re:Invent 2019: [REPEAT 2] I didn't know Amazon API Gateway did that (SVS212-R2)
Getting Started with Amazon Timestream
Building APIs with Amazon API Gateway
Visualizing Data in Amazon Timestream using Grafana

Developer Guide

Amazon Timestream - Developer Guide.
Amazon DynamoDB - Developer Guide
AWS Lambda - Developer Guide
Amazon CloudFront - Developer Guide
Amazon Simple Storage Service - User Guide