Eric Boilard
Southern New Hampshire University
April 17th, 2022

Project Two Conference Presentation: Cloud Development

CS 470 Project Two Conference Presentation: Cloud Development

Eric Boilard

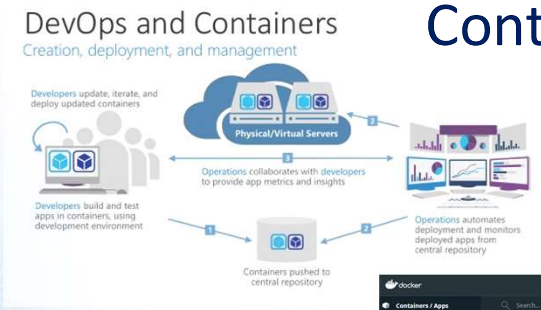
04/2022

Video Link: https://youtu.be/mq4jSE-M8WA

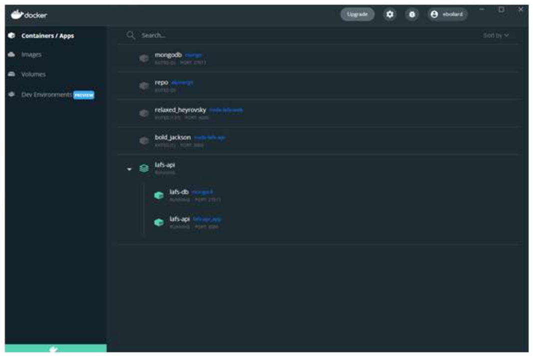
Overview

Hi Everybody,

My name is Eric Boilard, and I am a student here at Southern New Hampshire University and a former employee of the company AMS Osram where I spent most of my working years in various positions. I have only been programming for about 4 years now and I am excited to bring my education to a close with something as fun as taking a full stack application and pushing it to the cloud through Amazons AWS. This presentation is going to cover exactly that and all of the fun details in between that make AWS exciting to use. Join me on our cloud migration journey where we start with a short discussion on migration models.


DevOps and Containers
Creation, deployment, and management
Developers update, iterate, and deploy updated containers
Physical/Virtual Servers
Operations collaborates with developers to provide app metrics and insights
Developers build and test apps in containers, using development environment
Operations automates deployment and monitors deployed apps from central repository
Containers pushed to central repository
1
2
2
3

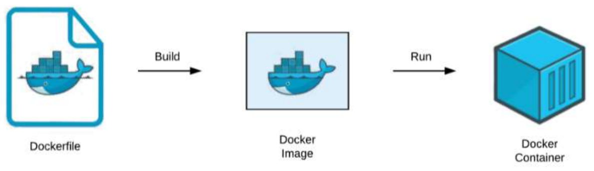
Containerization

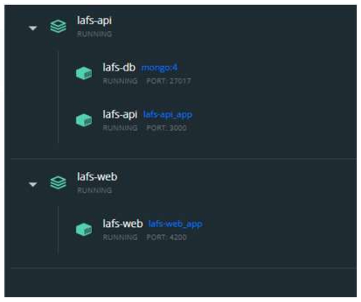


There are multiple different models used to migrate a full-stack application to the cloud. Some models are simply a pick and place system where you can move the entire application to the cloud seamlessly. Other models involve some level of refactoring whether it be a very small amount just to re-platform the project or an aggressive amount which involves refactoring the entire project to work in the cloud. My experience within this course project was that we had to do a very small number of changes to the application itself to get it ready to migrate fully to the cloud and most of the work involved was in the setup of our cloud environment to be able to work with our pre-existing application.
Alright…
Now that we know the types of models available and the type that we used, lets get into containerization. Containerization is when you take all of the code and all of the necessary files that come with it like libraries, frameworks, and dependencies and package it up into little containers where they are isolated from the outside environment. In the same way that a physical shipping container can move from one port to another with its contents intact, the same occurs with our containers. You can move these containers across environments and since they are their own fully functional environments, they work without needing to change any information. So, what do we need to containerize our application? Well, we need a containerization software and here we use Docker, and then we need two files, a dockerfile and a YAML file. Our dockerfile controls our runtime

environment and the installation of packages while our YAML file controls the hardware and network security requirements. Once we have these elements in place, we simply use docker in the command prompt to build our containers and run them with short commands thanks to docker compose.

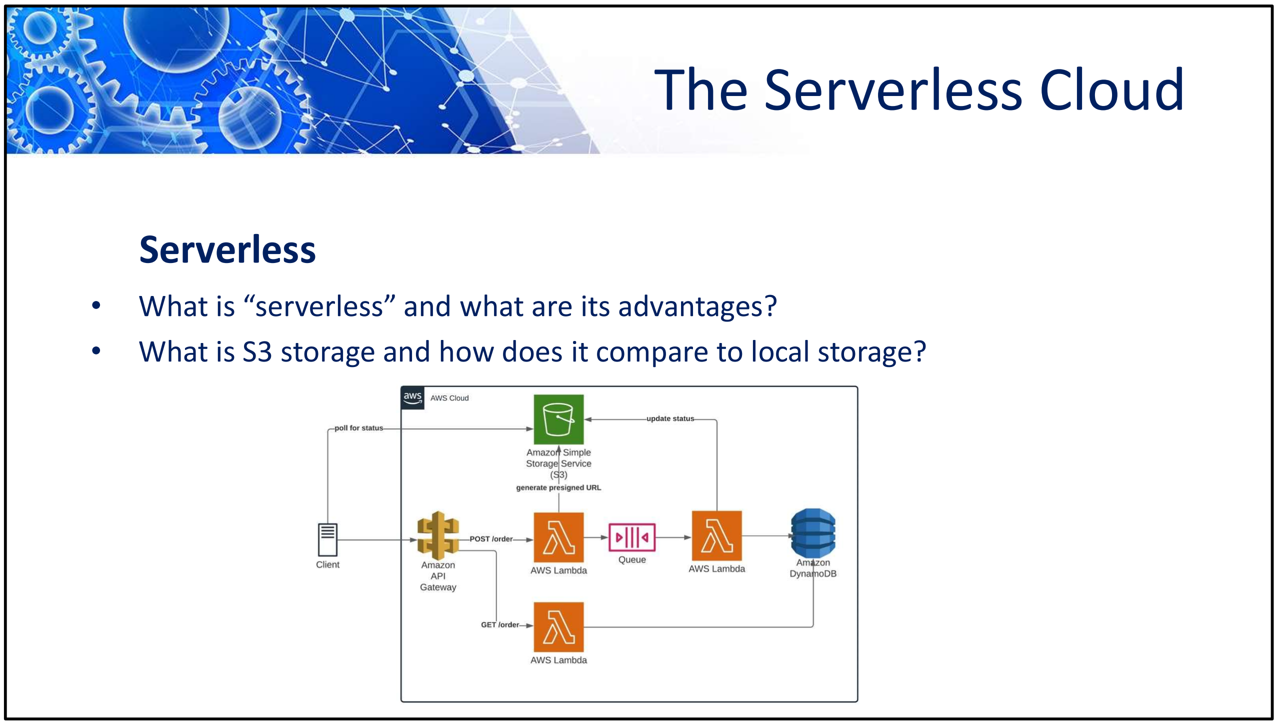
Orchestration





```
lafs-api
RUNNING
    lafs-db mongo:4
    RUNNING   PORT: 27017
    lafs-api lafs-api_app
    RUNNING   PORT: 3000
lafs-web
RUNNING
    lafs-web lafs-web_app
    RUNNING   PORT: 4200
```

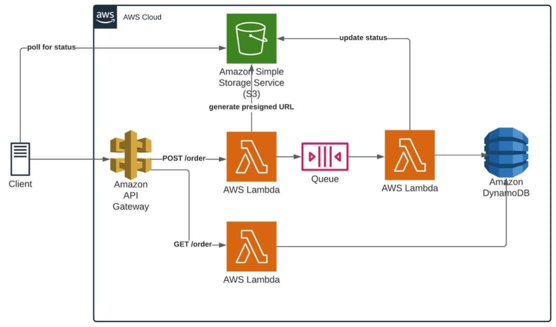
Okay so containers are pretty great and work for many applications but theres a small issue. These containers need to be managed and when the number of containers starts to grow so does the task of managing each one. This is where orchestration comes in. Orchestration takes the busy work away from the developers and automates the process. This is why we use Docker compose, it does the heavy lifting and helps us run multiple containers and environments at once with one command and one system. Not only can we start everything up and shut it all down with very little instruction, but the entire management of resource allocation, scaling, load balancing, provisions, and configuration is done for us. This makes the running of our containers very efficient and less of a burden as we focus on other aspects of our project.



The Serverless Cloud

Serverless

- What is “serverless” and what are its advantages?
- What is S3 storage and how does it compare to local storage?

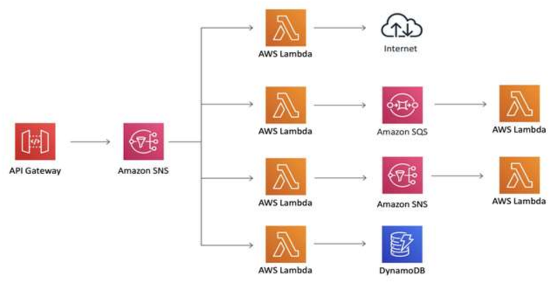


Now that we have packaged up our application nicely and made use of Docker compose, we need to start thinking about what kind of cloud development practice we are going to use. For this project we are going to use the serverless cloud model which means exactly that, no servers. Well kind of, what serverless really means is that we don’t have to worry about servers, that’s for our cloud service provider to worry about. For example, for this project we use Amazons AWS which is a web/cloud service. When we use this service, we don’t have to worry about managing servers regarding security, load balancing, resource management etc. Since we don’t have to worry about the management of our servers we really just need to worry about application and getting it running. I like to think of this like a rent-a-car service because you don’t need to maintain the car yourself and you only have to pay for what you use while you focus on the journey. Unlike a rent-a-car, given how small a project is you might not have to pay for cloud services like AWS. In fact, there are many free options out there that only start charging you once you reach a certain storage size or traffic flow.

Speaking of storage, how exactly does that work with AWS? AWS has something called the simple storage system, also known as S3. This is an object storage service that offers data availability, performance, security, and scalability that you can access at any time and anywhere. These objects get stored in buckets which can be configured in many ways and can accept any data type. In this project we take some of our necessary web application files and store them in a bucket with each file being its own object. Just like we mentioned before, this is free, specifically, up to 5gb of data, which makes this very possible to use for many small projects. For reference, this projects bucket was only 1.9MB in size. The only differences between using S3 and local storage is that local storage will have less latency depending on the hardware being used and local storage is also more secure.

The Serverless Cloud

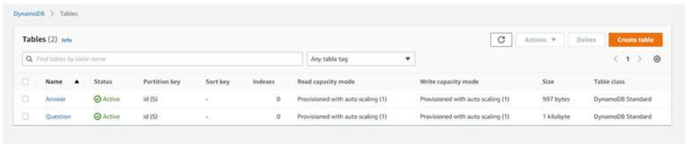
API & Lambda



Okay, now that we have storage covered, what do we do with our API and what is a Lambda? Well, we need to build our API in the cloud, and this brings along a few really great advantages including added security and some easy to monitor metrics. Our API is what's called an API Gateway, and this acts as an intermediary between our web application where the users interact and our Lambda functions which GET, POST, PUT, and DELETE information. Lambdas are functions that we build to complete certain tasks. For example, a lambda called DeleteRecord is going to be the function responsible for deleting a single record. Each Lambda function is built with an index.js file which gives the function instructions on what to do. Once we have all of our Lambdas created with instruction on what to do when they are called, we need to make sure we have permission to do these things. We do this by creating some policies with defined permissions which we will cover later in this presentation. Next, we build our API which is composed of all the types of requests it can receive from the users. If a user makes a GET request for a question the API Gateway will make a request to the correct Lambda function and the Lambda function will then send a response back. To summarize the steps needed to integrate the front-end and back-end, we need to set up our lambda functions, define roles/permissions, create our API Gateway, and grant access to our API to connect to our Lambda functions.

The Serverless Cloud

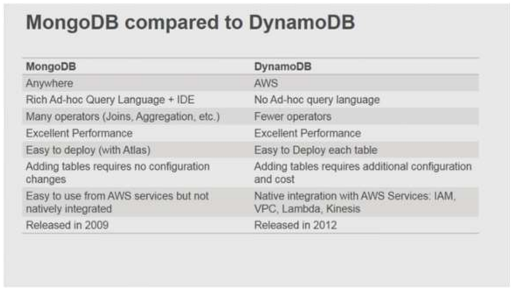
Database



DynamoDB > Tables

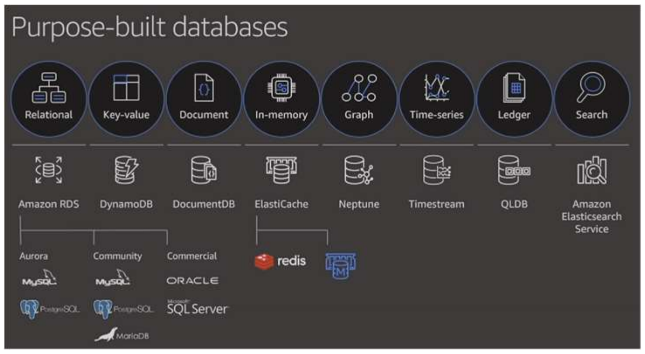
Tables (2) Info

	Name	Status	Partition key	Sort key	Indexes	Read capacity mode	Write capacity mode	Size	Table class
☐	Answer	✓ Active	id (S)	-	0	Provisioned with auto scaling (1)	Provisioned with auto scaling (1)	997 bytes	DynamoDB Standard
☐	Question	✓ Active	id (S)	-	0	Provisioned with auto scaling (1)	Provisioned with auto scaling (1)	1 kilobyte	DynamoDB Standard



MongoDB compared to DynamoDB

MongoDB	DynamoDB
Anywhere	AWS
Rich Ad-hoc Query Language + IDE	No Ad-hoc query language
Many operators (Joins, Aggregation, etc.)	Fewer operators
Excellent Performance	Excellent Performance
Easy to deploy (with Atlas)	Easy to Deploy each table
Adding tables requires no configuration changes	Adding tables requires additional configuration and cost
Easy to use from AWS services but not natively integrated	Native integration with AWS Services: IAM, VPC, Lambda, Kinesis
Released in 2009	Released in 2012



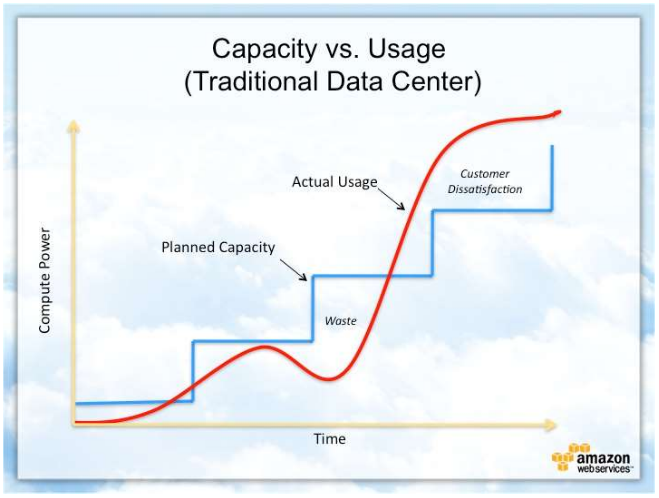
Before we started integrating our project to the cloud, we used MongoDB as our database. Now that we are using the cloud, we are using an AWS specific database called DynamoDB. As we can see above in the graphic, AWS has many purpose-built databases for different occasions but today we focus on DynamoDB. MongoDB and DynamoDB are very similar with only a few differences:

MongoDB can be used anywhere while DynamoDB is platform specific to AWS.
DynamoDB has fewer operations than MongoDB.
Adding tables in DynamoDB requires additional configuration while MongoDB does not.
And DynamoDB only allows for Key-value tables where MongoDB has more flexibility.

In this project we performed the major CRUD queries including create, read, update, and delete. In order to make this happen we had to grant permissions and integrate policies using AWS IAM.

Cloud-Based Development Principles

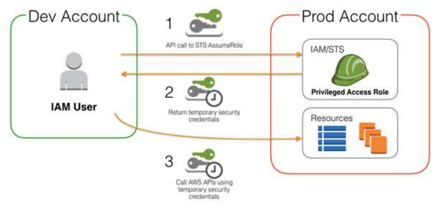
- Elasticity
- Pay-for-use model



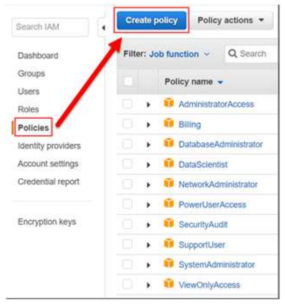
There are some principles of cloud-based development that are important to understand moving forward. The first is Elasticity which refers to how the cloud can grow or shrink in capacity and resources to fit your needs. This is important because it takes away the need to manage these items yourself as the developer and creates a more fluid experience. The next principle is the Pay-for-use model which many larger cloud service providers offer. This refers to the rent-a-car analogy I made earlier where you only pay for what resources you are actually using. In most cases when building smaller projects, you want need to pay anything as there is typically a free threshold. The pay-for-use model is important because it gives even more incentive to use cloud-based development because in the long run you will save money on resources you might have otherwise spent needlessly. As the graph above shows, there can be a considerable amount of waste during project development but with cloud-based development the compute power is kept in line with usage to avoid unnecessary cost.

Securing Your Cloud App

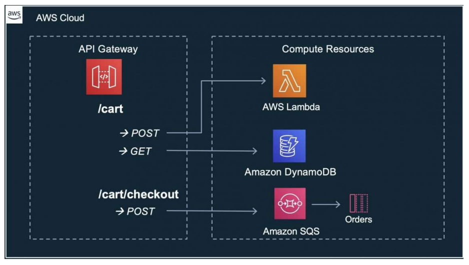
Access


Dev Account
IAM User
1
API call to STS:AssumeRole
2
Return temporary security credentials
3
Call AWS APIs using temporary security credentials
Prod Account
IAM/STS
Privileged Access Role
Resources

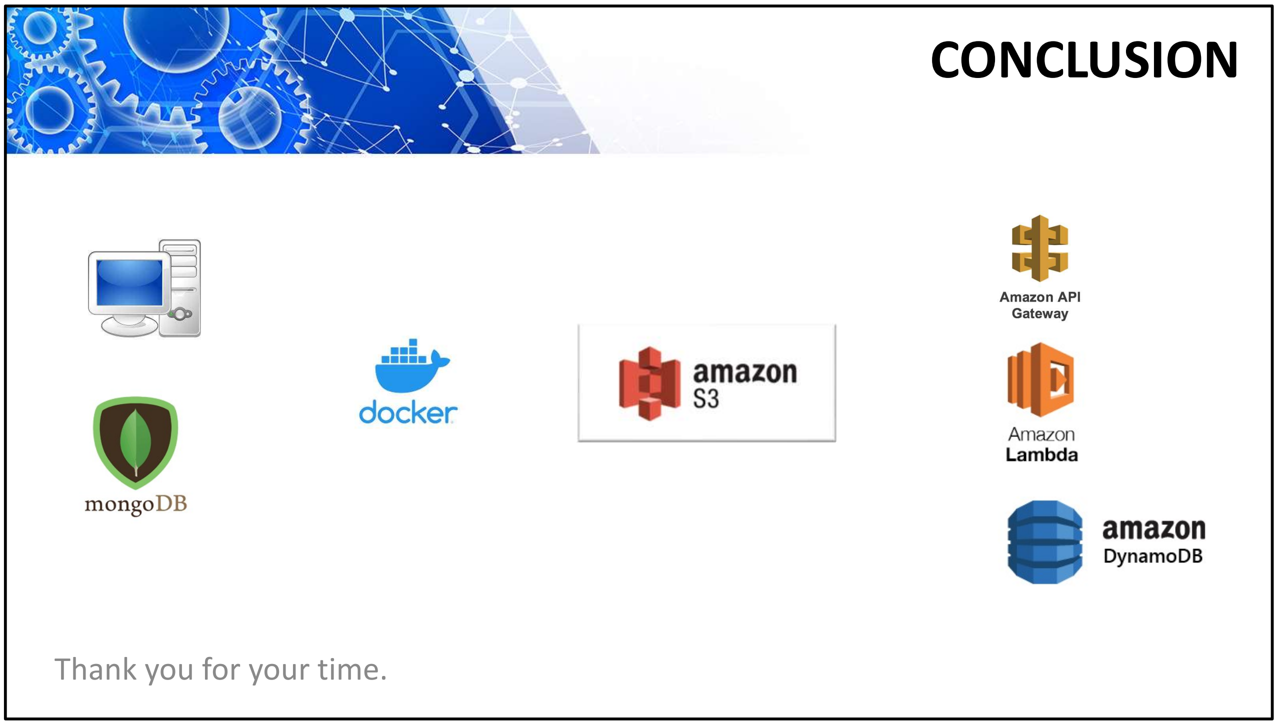
Policies



API Security



Preventing unauthorized access to our application and data is very important. Thankfully, AWS has some built in functionality called IAM which stands for identity and access management. This is where we create policies, attach policies to functions, and assign roles within our application. We can prevent unauthorized access by only giving the access that is needed to complete certain tasks. This is a minimum access approach which ensures that we are only allowing what is necessary and can help prevent unwanted access through our application. A role is an identity you can create that has specific permissions with credentials and a policy is an object in which we define permissions. We created a policy called Lambda Access to question-and-answer table which contains a json file that allows for certain actions within our database. We then assign this policy to multiple roles which gives these roles the access they need to the database. We secure the connection between Lambda and Gateway by granting invocation permissions and headers. We secure our Lambdas by granting least privilege and we keep our buckets secure by keeping them private or specifying the access that is allowed by certain individuals. You can also make them public like in our project but then creating a policy that allows for read and get only. This would be a different json policy, specifically a bucket policy.



CONCLUSION

Thank you for your time.

To briefly summarize this project and the migration to the cloud we will divide the topics into three points.

We first took a local application that used MongoDB and we learned how to containerize our application using docker and docker compose. We then deployed our application to an S3 storage bucket which contained 14 objects involving our web application. Lastly, we configured our DynamoDB database, our Lambda Functions, and our API Gateway to all communicate together so that a user's request was met with the correct response, all while ensuring each component was secure by utilizing IAM roles, policies, and permissions. In all, we were able to migrate our application to the cloud to optimize performance, resource utilization, and efficiency while maintaining our applications functionality and security.

RESOURCES

https://yt3.ggpht.com/5aY7Le5ad0fworhiDpgdjRn9h4fplTOWB8Yoodac0GVeSBawB2Hsmedg5XG66RiZR8xqcoGtneE=s900-c-k-c0x00ffffff-no-rj

https://nercomp.org/wp-content/uploads/2018/04/SNHU-Abbreviated-Blue-Logo-2017.png

https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcT80l73TdYIEWsMWflGX19pkqw5VbE61MSS5g&usqp=CAU

https://blog.aspiresys.com/wp-content/uploads/2018/08/Process-flow-by-Microsoft.jpg

https://miro.medium.com/max/925/1*XYhEwqz78qHuSsZvUt5R6Q.png

https://d2908q01vomqb2.cloudfront.net/fc074d501302eb2b93e2554793fcaf50b3bf7291/2020/07/14/Common-use-cases-with-different-data-flows-through-AWS-Serverless-services-Liberty-1-1024x576.png

https://d2908q01vomqb2.cloudfront.net/fc074d501302eb2b93e2554793fcaf50b3bf7291/2020/07/14/Common-use-cases-with-different-data-flows-through-AWS-Serverless-services-Liberty-1-1024x576.png

https://imgopt.infoq.com/fit-in/1200x2400/filters:quality(80)/filters:no_upscale()/articles/serverless-amazon-s3/en/resources/1lambda-function-that-receives-the-POST-request-will-generate-the-presigned-URL-1635434270139.jpg

https://res.cloudinary.com/dbzzslryr/image/upload/v1611072337/aws_databases/aws_databases.png

https://d2908q01vomqb2.cloudfront.net/22d200f8670dbdb3e253a90eee5098477c95c23d/2017/11/12/JC1_Createpolicy_1117.png

https://d2908q01vomqb2.cloudfront.net/22d200f8670dbdb3e253a90eee5098477c95c23d/2016/09/24/BrianWagner_stsAssumeRole.png

https://d2908q01vomqb2.cloudfront.net/fc074d501302eb2b93e2554793fcaf50b3bf7291/2019/07/29/Arch-comparison-1-1141x630.jpg

https://www.prolim.com/wp-content/uploads/2019/09/amazon-api-gatewat-1.jpg

https://miro.medium.com/max/360/1*qg4GEY91S1IHZ1nXfPCVSg.png

https://marvel-b1-cdn.bc0a.com/f00000000152152/www.zend.com/sites/default/files/image/2019-09/logo-docker.jpg

https://infinapps.com/wp-content/uploads/2018/10/mongodb-logo.png

http://coderdiaries.com/wp-content/uploads/2019/09/amazon-dynamodb-logo.png