Eastern Washington University
EWU Digital Commons

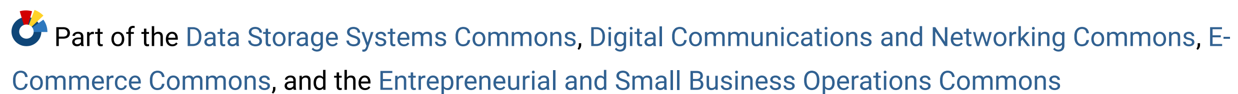
EWU Masters Thesis Collection | Student Research and Creative Works

Fall 2022

Towards Cloud-Based cost-effective serverless information system

Isaac C. Angle

Follow this and additional works at: https://dc.ewu.edu/theses


Part of the Data Storage Systems Commons, Digital Communications and Networking Commons, E-Commerce Commons, and the Entrepreneurial and Small Business Operations Commons

TOWARDS CLOUD-BASED COST-EFFECTIVE SERVERLESS INFORMATION SYSTEM

A Thesis

Presented To

Eastern Washington University

Cheney, Washington

In Partial Fulfillment of the Requirements

for the Degree

Master of Science in Computer Science

By

Isaac C. Angle

Fall 2022

Thesis of Isaac Angle Approved By

Yun Tony Tian — Date 12/08/2022

Name of Chair, Graduate Study Committee

Bojian Xu — Date 12/08/2022

Name of Member, Graduate Study Committee

Kurt Otto Buchholz — Date 12/06/2022

Name of Member, Graduate Study Committee

Abstract

TOWARDS CLOUD-BASED COST-EFFECTIVE SERVERLESS INFORMATION SYSTEM

By

Isaac C. Angle

Fall 2022

E-commerce information systems are becoming increasingly popular for businesses to adopt. In this work, we propose a serverless information system that will reduce costs for small businesses trying to create an e-commerce website. The proposed serverless system is built entirely in Amazon Web Services (AWS). The proposed serverless system allows businesses to pay for the use of cloud resources on a per-order granularity. This model reduces the cost of the information system when compared to a traditional cloud-based system. As e-commerce websites become more vital for small businesses, a cost effective serverless approach is promising.

Acknowledgement

A special thank you to Yun Tian. His advice and knowledge of cloud systems and AWS were instrumental to the foundation of this project.

Contents

1 Introduction

1.1 Traditional 3-tiered Information System . . . .

1.2 Introduction to Cloud Computing . . . .

1.3 The Motivation for This Work 2

1.4 The Problem We Addressed 3

1.5 Organization of This Paper 4

2 Related Work **4**

3 Methodology **7**

3.1 Overview of Services Used 7

3.1.1 Cognito 8

3.1.2 API Gateway 8

3.1.3 Lambda 8

3.1.4 DynamoDB 9

3.1.5 CloudWatch 9

3.1.6 Athena 9

3.1.7 S3 10

3.1.8 CloudFront 10

3.2 Our System Design 10

3.2.1 Why We Used This Design 12

3.2.2 The Advantages and Limitations of the Design 12

3.3 Implementation 13

3.3.1 Login and Registration 15

3.3.2 Retrieving and Displaying Product Information 16

3.3.3 Updating the Cart and Checkout 19

4 Performance and Cost Analysis **20**

4.1 Cost Comparison 20

4.1.1 Cost of Serverless Design 20
4.1.2 Cost of Traditional 3-Tiered Design with EC2 Servers 22
4.1.3 Cost Comparison Between the Serverless and 3-Tiered Architecture 23
4.2 Athena Data Analytics 24
4.3 Performance Comparison 26
4.3.1 Execution Time of Traditional 3-Tiered System 27
4.3.2 Execution Time of Serverless System 27

5 Conclusion **29**

6 Future Work **30**

References **32**

Vita **36**

1 Introduction

With the surge of users buying products from the internet, e-commerce websites have increased in popularity. To keep up with this trend, most companies have developed a website to cater to these online shoppers. E-commerce websites are not limited to a specific type of product and are a necessary part of any growing business in today's world [1]. These websites act as the portal to a back-end information system. Normally, such information systems will provide the following features: including sign-up of users, shopping cart, authenticating and authorizing users, processing the customers' orders, and generating sales reports.

1.1 Traditional 3-tiered Information System

Traditionally, the 3-tiered architecture is widely adopted when designing and implementing an e-commerce back-end information system. The 3-tiered architecture in an information system consists of a front-end web page, an application server, and a database in the back-end [2] [3]. The web page is normally created using a mix of front-end languages such as HTML, JavaScript, and CSS. The application server hosts the main logic for the design and can be stored on a physical server or one located in the cloud. For smaller businesses, using a cloud-based server as the application server can be the more cost-effective approach.

1.2 Introduction to Cloud Computing

The cloud provides on-demand services (IT resources) accessible to the companies. Cloud resources are geographically located outside the company's on-premise (on-site and local) data center. The hardware and software contained within a cloud provider's data center that enable the provision of the cloud services, and the cloud services themselves, provided to the general public over the Internet, is called the cloud [4].

Cloud services are available in a pay-as-you-go manner. Users are charged financially based on the amount of resources used and the duration of time used by the service [5]. Cloud users may request for IT resources any time when they are needed, and release the resources after they completed their computation tasks [6].

The key advantage to using cloud services over on-premise IT resources is that the demand for the service doesn't need to be known in advance. For example, a server will need to be provisioned

to meet the highest demand during a month and will be underutilized during other times of the month. Cloud services mitigate this issue by dynamically adding more servers for processing the high workload and removing idle servers when fewer requests are received.

Other advantages of using the cloud include the following features. The IT administrators workload will be reduced as well. Physical servers need to be maintained and upgraded over time. Cloud-based services are maintained by the cloud service, which alleviates the time and energy spent on physical servers.

In this work, we chose to use Amazon Cloud Services (AWS) in designing and implementing our system due to its high popularity [7]. AWS services in the cloud enables the design and implementation of the entire 3-tiered information system. The front-end web portal and the application server can both be created and stored in an EC2 machine. EC2 machines are cloud-based servers provided by AWS [8]. The database can be hosted in the cloud as well, using Amazon's Relational Database Services (RDS). The relational databases in AWS can be connected to the EC2 server.

The EC2 server has a public IP address that can be reached by the end customers. Then, Domain Name Service (DNS) translates or associates the public IP into a readable website domain name [9]. AWS offers a DNS service called Route 53 which will allow customers to use the registered domain name to reach/access the website. With these services, a basic functional e-commerce website can be created [10] [11].

1.3 The Motivation for This Work

There exists at least one disadvantage in the traditional 3-tiered cloud-based information system. In the 3-tiered information system, the EC2 web server that hosts the business logic will have to run constantly, without stopping and without interruptions, when serving customer's requests. But for small businesses, their information system may only have a few hundred orders per day, such as a restaurant, during two or three busy hours in the day. For the rest of the time, the EC2 server will be idle. For every idle hour, the EC2 server costs the company money without generating any financial gain. To mitigate this issue, we propose the adoption of a serverless architecture to create the website instead of using the traditional 3-tiered architecture. The serverless design allows the businesses to pay for the use of the cloud resources on a per-order granularity, which will greatly reduce the cost on information technology (IT) for companies.

1.4 The Problem We Addressed

In this work, we designed and implemented a cost-effective information system, which is based on the following AWS services: API Gateway, Lambda function, and DynamoDB [12] [13]. The system is serverless and hosted entirely within the AWS cloud. The advantage of this design is that it does not use a constantly running server. AWS Lambda is used in the place of a server. Lambda is an event-driven service that can be used to run code for any application without the need for provisioning a server [14]. An instance of Lambda is called a Lambda function. Lambda functions can be triggered by other AWS services, the most common being API Gateway.

API Gateway is a service that makes it easy for developers to create and manage APIs. API Gateway acts as the entry way to other AWS services [15]. For the serverless information system, we used API Gateway as a door to the Lambda functions. The Lambda functions then access DynamoDB, which is a cloud-based NoSQL database system. NoSQL databases follow a key-value model where every column in the database corresponds to a unique primary key [16]. The primary key is the only required column; any number of additional columns can correspond to the primary key. Because of this model, DynamoDB can contain a large amount of data and is less expensive than many other databases [16]

Our proposed serverless system is architectured in the following fashion. The front end website is stored in an S3 bucket and made publicly available with CloudFront. S3 buckets are large cloud storage devices and CloudFront is Amazon's Content Delivery Network (CDN). The application server is created using the API Gateway and Lambda function combo. Together they do what an application server normally does; receive requests from the front end and access the database. Using the serverless model can be more complicated than the traditional 3-tiered approach, but the serverless model is more cost-effective. The entire design and implementation will be presented in Section 3.

The advantage of using the serverless model is API Gateway and Lambda functions only cost money when they are executed. The API gateway will only run when a request is made from the front end. If there are no orders being made, then the API Gateway and Lambda functions don't cost anything. If a company only has a few hundred people shopping per month, then the company will not lose any money during the idle hours. The S3 bucket, CloudFront, and DynamoDB have

a small monthly cost, but with our design the cost was usually less than one dollar per month for a small business [17].

To discover how cost-effective the serverless design is in comparison with the traditional design, several tests were conducted. The tests are recorded in section 3 of this paper. In our experiments, the serverless architecture was found to be more cost-effective than the EC2 machine as long as the Lambda functions didn't receive more than 11 million requests per month. If the number of requests is higher than 11 million, the EC2 machine will cost less and perform better in all metrics. The EC2 machine was also consistently faster than the serverless architecture. A database request will take 5 milliseconds on the server and up to 100 milliseconds for an already warm Lambda function. The serverless architecture will provide acceptable loading times and be much less expensive than an EC2 machine on average.

1.5 Organization of This Paper

The outline for this paper is as follows. In section 2 a review of previous works in the area of serverless design will be presented. Section 3 will explain the services that were used to create the information system. Once the services are explained, the architecture of the serverless information system will be explained. In section 4 a comparison between an EC2 server and the serverless website will be conducted. The comparison will include the cost of each service, how easy it is to get reports from the NoSQL database, and a comparison on speed. Section 5 will give an overview of the results from Section 4. Section 6 will discuss future work that can be completed as well as a few closing observations.

2 Related Work

In this section we provide an overview of previous works in serverless computing. Cloud Computing is the use of any cloud-based resources that use the pay-as-you-go model for pricing. It is becoming increasingly popular, and is used in many use-cases. These include event-triggered computing, live video broadcasting, Internet of Things (IoT) data processing, and shared delivery systems. IoT is the network of physical objects that contain sensors and software [18]. The continued implementation of cloud computing will likely lead to simpler and more effective resource management. It is to be noted that this analysis is based off a small sample size of cloud-based

applications [19].

Serverless computing allows users to create functions that follow a different pay structure. The developers are only charged for the time that the function runs, the amount of memory used, and the number of back-to-back executions of multiple functions. To help reduce this cost, fusing of functions, splitting functions across more resources, and the memory for each function was reviewed. Fusing functions was shown to reduce an image processing application cost by almost 40% of the original cost. Placing the functions in different locations reduced cost by over 50% of the original cost. Having the correct amount of memory can reduce cost and latency as well [20].

AWS cloud computing has the advantage of a pay-as-you-go payment model. However, the serverless architecture can cause more latency. To test this latency, Lambda functions were triggered by API Gateways and other Lambda functions. The latencies that were found were small on their own but could add up when combined together as a part of a function call. If an application needs short response times, then these latencies should be addressed. To test the latency in a production environment, a React app was created and connected to an API Gateway, which was connected to a Lambda function and DynamoDB. When many users were tested on the site at the same time, some of them had unresponsive applications. Because of this, developers of applications in AWS should know the underlying architecture that is needed to create their application in order to reduce latency [21].

Serverless computing has potential to be used as a highly scalable and available architecture. However, there are some flaws and gaps that need to be addressed when viewing this architecture. Serverless computing has limited lifetimes because Lambda functions only run for a short period of time. Lambda functions cannot talk to each other except through a slow intermediary service. When training a neural network through this architecture, it was 21 times slower and 7 times more expensive than running the same network on an EC2 machine [22].

Function-as-a-Service (FaaS) is useful for building highly available and scalable applications. Most FaaS services abstract the underlining infrastructure to increase ease of use. Serverless Application Analytics Framework (SAAF) was created to allow the developers of these apps to view information regarding the underlying architecture of their applications. The service can be used with many different programming languages as well as with multiple different cloud services. With this tool in hand, analysts can quickly and easily create experiments and view the results [23].

Serverless computing for a single cloud platform is easy to deploy and allows the user to use the services without worrying about the underlying architecture. Using this paradigm in a multi-cloud environment is a much harder endeavor. To combat this issue, a model for deploying these applications was made. The model is called Topology and Orchestration Specification for Cloud Applications (TOSCA). The model was shown to work for multi-cloud-based applications. If the application follows the TOSCA guidelines, it can automatically execute on multiple cloud services [24].

Cloud computing architecture needs to be secured as much as traditional hardware. The responsibility to secure AWS services is up to Amazon and the responsibility to secure applications created in the cloud and how they connect to the cloud is up to the customers using AWS. There are six proposed design patterns that can be used to secure a cloud application; periodic invocation pattern, event-driven pattern, data transformation pattern, data streaming pattern, state machine pattern, and bundled pattern. Using one of these patterns allows for easy creation of secure applications in the cloud [25].

Serverless architecture is limited by execution time and total storage space. This can limit the effectiveness of deploying certain kinds of applications in the cloud. Because of these restraints, workflow scheduling has strict limitations when deployed in the cloud. To combat these issues a new cloud-based scheduler was created. The scheduler is not constrained by the lack of stateful Lambda functions and allows for easier debugging and testing. The application has not been added to the benchmark of other systems yet, but it would be a good performance comparison for other workflow schedulers that are created in the cloud [26].

Serverless architecture is an emerging technology which provides a solution for deep learning models. The serverless model works well for deep learning due to its pay-as-you-go model. The infrastructure can also be increased or decreased based on the needs of the algorithm. Deep Reader is a deep learning algorithm that takes advantage of the serverless model by using Lambda functions to perform the needed calculations. When compared to a virtual machine the Lambda functions had both a lower cost and a higher performance when compared to the virtual machines [27].

Sequential and MapReduce models require an ”always on” infrastructure and many machines to run the algorithms in parallel. Because the serverless architecture allows for infinite scaling, it is the ideal solution for these models. In order to utilize the serverless model for a MapReduce problem,

the following AWS services were used: S3, SQS, Lambda, DynamoDB, and API Gateway. By creating the application in this way, there were several benefits. The application was highly available and reduced costs. The system could be scaled to as large as needed for any given problem [28].

DoubleML-Serverless is a proposed double machine learning application that can be created in the AWS cloud. The double machine learning paradigm can be used on many models. Creating the application in the cloud will allow for the architecture to grow and be deployed with ease. The cost for the architecture can be increased or decreased based on the needs of the model [29].

An application was created and deployed in AWS for big data analytics. The system used Amazon S3, Glue, Athena, QuickSight, Kinesis, SageMaker, and Step Functions. Included in the application was a Data Lake where many different types of data can be stored. The Data Lake can grow due to the AWS architecture and a pay-as-you-go pricing scheme, allowing for almost infinite scaling. Analytics can be performed with the help of AWS Glue, allowing for a large amount of data to be analyzed in many different ways [30].

OpenLambda is a proposed Lambda function-based service that is an open-source version of AWS Lambda. Lambda is highly scalable and does not require complicated configuration. Lambda uses interpreted languages, has package support, allows for a state-like configuration with cookies and sessions, and connects with databases like DynamoDB for data access. OpenLambda follows this same Lambda function format and will include LambdaBench which will allow the user to view benchmarks of their infrastructure [31].

SIREN is a distributed machine learning application that was created in the AWS cloud using Lambda functions. The Lambda functions are used to split the load for training the system. To reduce costs and increase the quality of the training data, there is also a scheduler that balances the number of Lambda functions being used. The training with this architecture has resulted in a 44.3% speed increase, for the same price, when compared to using EC2 machines [32].

3 Methodology

3.1 Overview of Services Used

In this section, we present the AWS cloud services that we utilized to implement the proposed serverless information system.

3.1.1 Cognito

Cognito is a secure, customer identity service that scales with a large number of users. users can sign-up and sign-in with their credentials using Cognito. Cognito scales to millions of users and integrates with the front end and back end development [33]. Cognito can integrate with many of the other AWS services, including API Gateway.

3.1.2 API Gateway

The API Gateway acts as a mediator between the front end and the back end of an application. API Gateway allows requests to be sent to other AWS services in the same way an API would. The gateway can handle requests from other AWS services or any third-party software that can send HTTP requests. These requests are usually sent from applications that are trying to access data stored in the back end of the application.

The API Gateway can be created as an HTTP API, WebSocket API, or a REST API. The WebSocket API uses a persistent connection between the front end and the back end. This works well for chat applications that need a continuous connection between the front end and the back end to display messages quickly. The HTTP API is a REST API with fewer features than the traditional REST API that AWS offers. HTTP is the more cost-effective service when compared to the REST API, but it has fewer features [15]. The REST API can be connected to a Cognito user pool which will restrict access to only authenticated users. The access token is retrieved from Cognito and must be sent with the request to gain access to the restricted sections of the back end.

3.1.3 Lambda

AWS Lambda acts as the application server for serverless applications. Lambda functions are stateless, event-driven, blocks of code. Lambda supports multiple major programming languages like Python, Java, and JavaScript. Lambda functions are triggered by events from other AWS services. API Gateway is a popular AWS service to use with Lambda functions. When a request is sent to the API Gateway, the API triggers the Lambda function to initialize and sends it the request. The Lambda function will receive the request, which can contain values in the body and the header. These values can be parsed from the request with Lambda-specific functions. Once the function has run, it will send a response back to the API Gateway [14].

Lambda Functions can filter through requests to ensure only valid requests are using the Lambda function. The function can check to make sure that the requests that are received are in the correct format. If the request is not in the correct format, then the function can reject it and return an error message. Because of this, Lambda functions can keep malicious requests from affecting other AWS services. Lambda Functions are also stateless, which means they don't remember any past requests that have been received. Previous invocations cannot be used to cause a vulnerability in the current invocation of the function.

3.1.4 DynamoDB

DynamoDB is Amazon's NoSQL database service. A NoSQL database differs from a relational database in several ways. NoSQL databases follow a key-value model where every value in the database corresponds to a key. Because of this, the database can store values with a varying number of columns. The values must correspond to the primary key, but the number of columns that correspond to the key may vary. With this design, the database can store data of different data types or sizes in the same table. The advantages to this design are quick read and write speeds, easy to scale, and lower cost than a relational database. The downside to using a NoSQL database is it doesn't use SQL, which is the standard choice for database languages. NoSQL databases don't support complicated joins between tables either; a single database table is commonly encouraged for most applications [16].

3.1.5 CloudWatch

CloudWatch collects and visualizes logs and data from other AWS services. Alarms and automated responses can be created to activate when a threshold is met. CloudWatch can be used to troubleshoot AWS services by providing insights within the logs for the service. The detailed logs CloudWatch keeps for each service provide developers with useful insight into their information systems [34].

3.1.6 Athena

Athena performs SQL queries on data stored in S3 Buckets. Athena can use data from other AWS services, but it must first be transferred to an S3 bucket. Once the data is stored in the bucket, Athena can then perform powerful SQL queries. An important use for Athena is performing com-

plicated queries on DynamoDB tables. Because DynamoDB is a NoSQL database, complicated joins cannot be performed on the tables. If the data from the table is transferred to an S3 bucket, Athena can then be used to mitigate this issue.

3.1.7 S3

S3 provides a service for storing large amounts of data in objects called buckets. These buckets can store file structures and even full sized apps. S3 buckets are easily connected to other AWS services, providing an exceptional amount of versatility. The buckets can be used with Athena to provide insights into big data sources. The buckets can also be made public, allowing web pages to be served to users through the link to the bucket. If the link to the bucket is connected to a domain name, a website address can be created.

3.1.8 CloudFront

CloudFront is Amazon's Content Delivery Network (CDN) service. The service delivers content from other AWS services to the public. The AWS services that can be shared are S3 and API Gateways. If the S3 bucket is made public, CloudFront can share the public link. The CDN can be used to serve web pages to clients as long as it is set up with a registered domain. The domain can be created through AWS Route 53 or with a third-party DNS software.

3.2 Our System Design

Figure 3.1 presents the design of the proposed system. The front end of the information system is a website created entirely with React[35]. The website provides users the functionality to view the available products and add them to their cart. Users can also login to their account or create a new account. Users can add products to their cart as a guest, but only authenticated users can checkout. All the products listed on the website are stored in a database in the back end. The front end retrieves information from the back end by sending an HTTP request to one of the API Gateways.

HTTP stands for Hypertext Transfer Protocol and is used to access the API's stored in the back end [36]. There are three different types of HTTP requests that are commonly used for data retrieval from a database. An HTTP request that only retrieves values from the database is called a GET request. POST requests create new values in the database. PUT requests update products in the database that have already been created. The POST and PUT requests have new values stored in the

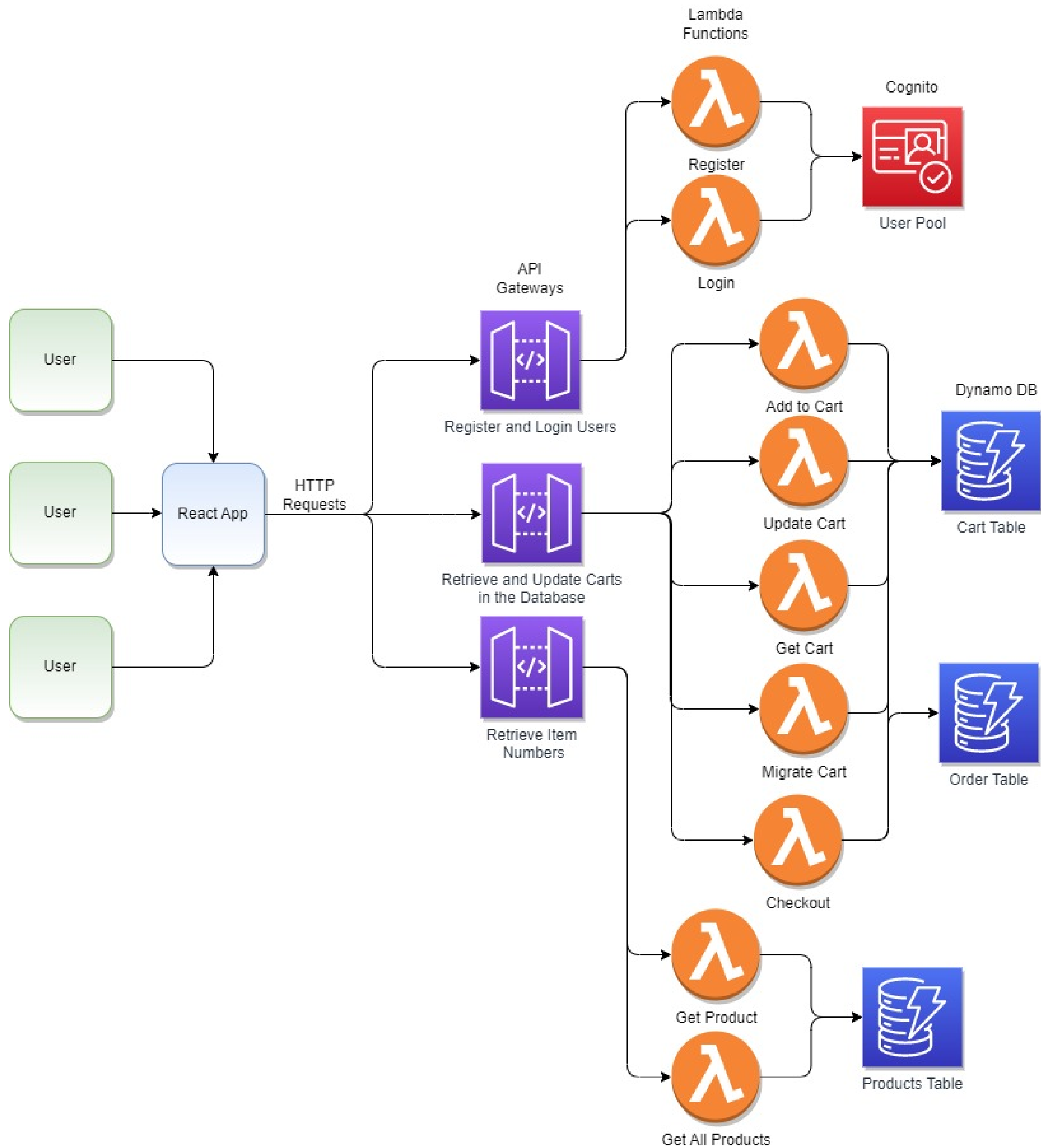


Figure 3.1: Information System Architecture

request. The GET request only retrieves data from the database. Each of these requests are used to access the API Gateways in the back end.

The back end of the system consists of three API Gateways. Each API has a unique function that it fulfills. One of the APIs is used to access the Login and Register Lambda functions. The Register function is used to create a new user and the Login function allows an existing user to login to their account. Both of these functions use the same Cognito user pool to store and access the user

credentials.

The next API Gateway gives access to the Add to Cart, Update Cart, Get Cart, Migrate Cart, and Checkout Lambda functions. The Add to Cart function updates the user's cart by adding new products to the database table. The Update Cart function updates the quantity of a product already stored in the user's cart. The Get Cart function returns all the products stored in the user's cart. The Migrate Cart function updates the cart's guest id to an authorized user id when a client signs in. This function is responsible for products in guest carts moving to the authenticated user's cart. The Checkout function retrieves the user's cart from the Cart table, saves a copy, and then deletes it from the Cart table. The function then stores the cart in the Order table as a completed order.

The final API Gateway is used to access the Products table using the Get Product and Get All Products Lambda functions. The Get Product function returns a single product from the database. The Get All Products function scans the entire Products table and returns all the products contained inside the database. This function is used to populate the front end with pictures and information for each product.

3.2.1 Why We Used This Design

We used this design for our information system because the serverless architecture provides a scalable back end. DynamoDB is a serverless database capable of storing large amounts of data. Lambda is a serverless service that scales automatically and can access a DynamoDB table. Instead of using one Lambda function in the back end, multiple Lambda functions were used for simplicity. In our design, each Lambda function completes only one task. The front end cannot send a request to a Lambda function without an API Gateway. The API Gateway is responsible for receiving the requests from the front end and discerning which Lambda function to send them to. There are three API Gateways in the design to separate the design into different parts. The design is split into user authentication, updating the cart, and retrieving products from the database.

3.2.2 The Advantages and Limitations of the Design

There are two advantages when using our design instead of the traditional 3-tiered architecture. The first advantage is that the back end doesn't use a server. Servers will run constantly, even during hours of the day when no clients visit the website. When the server is running during these slower hours of the day, the company is paying for the server without making any profit. With our serverless

design, the company will only be charged when clients use the website. The second advantage is our design automatically scales with an increase in requests. Because every service used in our design is serverless, the back end will scale automatically with an increase of user requests from the front end. The design is more cost-effective and easier to scale than the traditional design.

DynamoDB does not create detailed reports because it cannot perform complex joins. A join is used to combine the data from two different tables using a shared column. The Order table does not store the product information in the database table. Instead, a product id is stored in the database which can be linked to the Products table. A relational database can perform a join on these two tables; however, DynamoDB is a NoSQL database. NoSQL databases do not support complex join operations. When the two tables are joined, useful reports can be created. For example, a report representing the popularity of a product or a report representing the biggest spenders. To mitigate this limitation, we propose the use of AWS Athena in Section 4.2.

3.3 Implementation

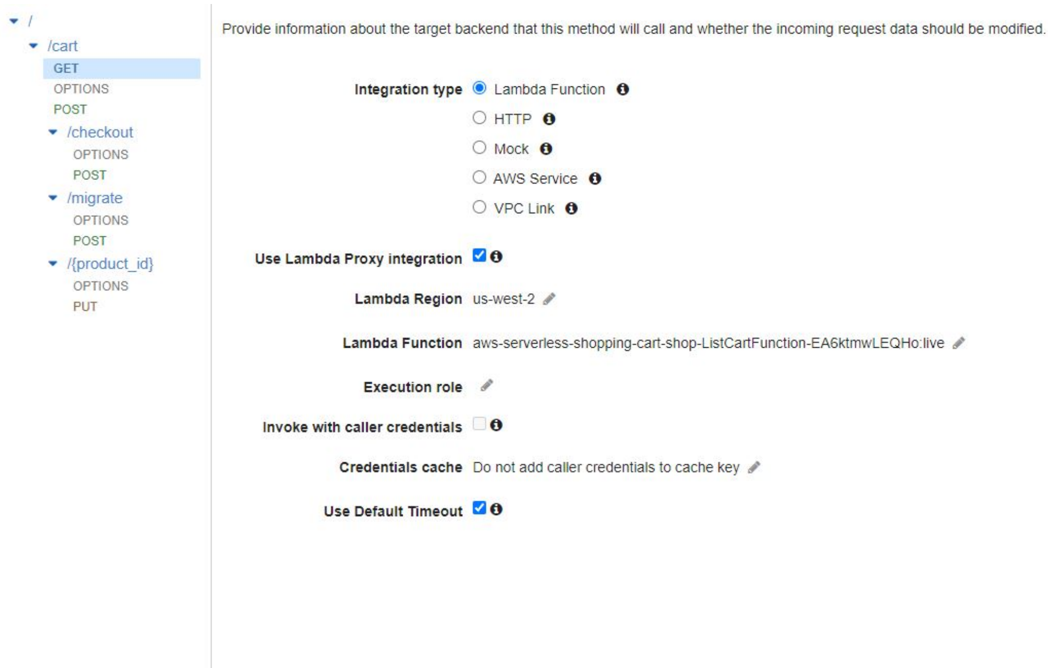


Figure 3.2: API Gateway Resources

The proposed architecture uses HTTP requests to access the databases in the back end. Every HTTP request consists of a routing URL, some header information, and a body. The routing URL acts like a path that leads to an API Gateway. A routing URL for the API Gateway presented

in Figure 3.2 is ”https://3fyby70779.execute-api.us-west-2.amazonaws.com/Prod/cart”. From the beginning of the URL until the first single forward slash is the path to the API Gateway. The next section of the URL contains the version of the API Gateway. In this case Prod, which is short for production. the version name can be set to any value, we used Prod because the version was used for production. The final part identifies which resource the API Gateway should use, in this case cart. The cart resource is shown in Figure 3.2.

Resources can contain GET, POST, and PUT requests. The resources are shown in the left side panel of Figure 3.2. Each of the different types of HTTP requests will send the request to a different Lambda function. In Figure 3.2 a GET request path within the cart resource is chosen. This resource is integrated with a Lambda function and it is a proxy integration. When the proxy integration box is checked, the API Gateway will send the request to the Lambda function without intervention. The integrations act as triggers for the Lambda functions, causing the function to run when a request is made to this integration.

The Checkout resource can be accessed with this routing URL: ”https://3fyby70779.execute-api.us-west-2.amazonaws.com/Prod/cart/checkout”. The difference between this routing URL and the cart URL is the extra checkout on the end. To access the Checkout Lambda function, a POST request must be sent. Every resource in the API Gateway’s follows the same format, the only exceptions are the resources that require a product id.

The routing URL for the product_id resource requires a product id to be sent inside the routing URL, for example, ”https://3fyby70779.execute-api.us-west-2.amazonaws.com/Prod/cart/productId”. The real routing URL will include a valid product id in place of the ”productId” value. The product id could also be sent in the body, but with this design the body can contain the same values as other requests. The only part of the request that changes is the product id in the routing URL.

```
await fetch('https://3fyby70779.execute-api.us-west-2.amazonaws.com/Prod/cart', {
  headers: {'Content-Type': 'application/json', 'Authorization': token},
  method: 'POST',
  body: JSON.stringify({productId, quantity}),
  credentials: 'include',
})
```

Figure 3.3: HTTP Request Front End Example

Figure 3.3 is an example of the format the front end uses to send an HTTP request to the Add to Cart function. The HTTP requests have headers which contain information about which user is trying to access the Lambda function and in what format the data should be sent back. The ”Content-Type” is the format used for the body of the request. The ”Authorization” header is the access token sent with authorized users; if the user is a guest, this value will be blank. The method is the type of HTTP request, in this case it is a POST request.

The body contains the new values that will be added to the database. The values being added to the database in Figure 3.3 are the product id and the quantity. Both of these values are stored in JSON format. JSON is a string with a specified data format [37]. the credentials are set to ”Include”, this means that cookies are enabled for this HTTP request. Cookies are explained in more detail in Section 3.3.1.

In our implementation of the information system, the body of a POST or PUT request consists of a product id number and a quantity to be added or removed from the cart. The GET requests have an empty body because they are only used to retrieve data from the database. Each HTTP request sent by the front end uses Cross Origin Resource Sharing (CORS) to access the back end.

CORS permits access to restricted resources from outside to reach the back end [38]. API Gateway can be integrated with CORS to limit the origin of outside traffic that can access the API. If a request is sent to an API Gateway which has CORS enabled, it will be immediately rejected unless the request was sent from the URL that was specified. To gain access to the back end of the information system, an origin must be specified in the HTTP request. This origin must match the origin that was specified when CORS was integrated with the API. With CORS enabled, the API Gateway is more secure because HTTP requests that are sent from outside of the specified origin will be rejected.

3.3.1 Login and Registration

Figure 3.4 presents the design for the login and registration used for the application. To create a new user within the system, a customer must enter a username and password. The credentials will be sent to the back end through an HTTP request. The API Gateway will route the request to the Register function. The Lambda function will parse the username and password from the body of the HTTP request and send them to the Cognito User Pool for validation. If the username is not

in use and the password is long enough, Cognito will return an access token. The access token is a long string of characters used to identify the user when they access the restricted sections of the back end. The Lambda function will return the access token in the response and the front end will store it in local memory so it can be used to access the Migrate Cart, and Checkout function.

The access token is used to access the Migrate Cart and Checkout functions because they are restricted. The access token is a unique string of random characters used to identify an authenticated user. The access token is sent along with the requests and acts as a key that unlocks access to restricted API Gateway resources. The API Gateway resources that are restricted are connected to the Cognito user pool. These resources know when a valid user is accessing the Lambda function because they compare the access token that was sent with the access token given by Cognito.

Every Lambda function can be accessed without creating an account except for the Migrate Cart, and Checkout function. The customer must create an account before paying for the products in their cart. With this design, guest users can access the website and add products to their cart. Each of the guest users is identified by a unique cookie which represents them. A cookie is a piece of information that is saved in the browser [39]. Web browsers are stateless, which means they don't remember any previous messages that are sent. Cookies are used to provide a way for the browser to recognize the user across many different messages [39]. The unique cookie is sent with every HTTP request to identify each individual guest user. An unfinished guest cart in the database will only stay in the database for a week. Once a week has passed, the cart will be removed from the database. Authenticated users have 30 days until their cart expires. This design allows the website to reach a larger customer base while still retrieving the necessary information during checkout.

3.3.2 Retrieving and Displaying Product Information

The Get Product function is used to retrieve a product from the database. The Python code for the Get Product Lambda Function is provided in Figure 3.5. The Python library ”Boto3” is used to retrieve values from the database. With this library a query can be made to the database to retrieve products. The Get Product function only retrieves a single product. The function is primarily used by the Add to Cart and Update Cart functions. Each functions sends a request to the Get Product function and retrieves the desired product id when adding a product to the cart. The ”os” library is used to retrieve environment variables that are stored within the Lambda function.

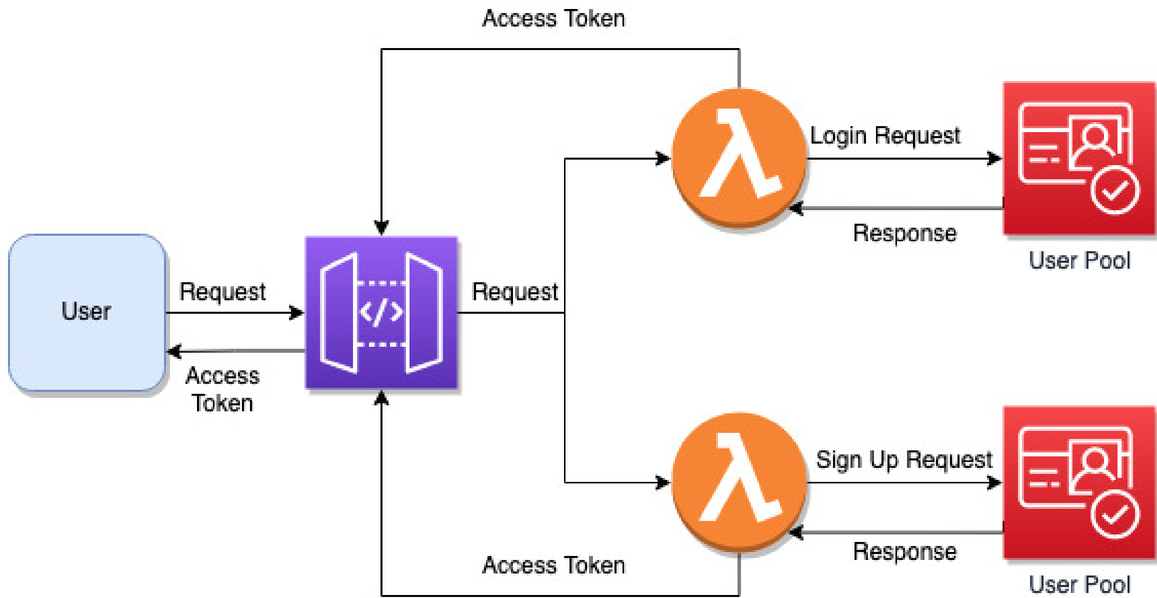


Figure 3.4: Login and Registration Model

The environment variables that we used for this function are the allowed origin and the name of the database table. The allowed origin is the CORS path that must be included in any requests that are sent to this Lambda function. The allowed origin is one of three different headers that are returned in the response to the request.

The headers for the response are created as a dictionary with key-value pairs. The "Access-Control-Allow-Headers" limits the type of header that can be used in the request. For this function, only the Content-Type header can be requested. The content type used in the response is in JSON format. An example of a product from the database stored in JSON format is given in Figure 3.6. The "Access-Control-Allow-Methods" is the different types of HTTP requests that this function will accept. GET requests are the only type of requests that are sent from the front end, the OPTION request is only used for gathering information. After creating the response headers, the function initializes the database. After initializing the database and identifying the correct table, the Lambda function retrieves the product id from the path parameters. The parameters are stored in the routing URL of the request. Once the product id is retrieved, a query is made to the database that includes the id as the primary key. Once the product has been retrieved, the function sends back a positive status code, in this case 200, along with the headers that were created earlier and the product in JSON format.

```python
import json
import os
import boto3

from boto3.dynamodb.conditions import Key

HEADERS = {
    "Access-Control-Allow-Origin": os.environ.get("ALLOWED_ORIGIN"),
    "Access-Control-Allow-Headers": "Content-Type",
    "Access-Control-Allow-Methods": "OPTIONS,GET",
}

dynamodb = boto3.resource("dynamodb")
table = dynamodb.Table(os.environ["PRODUCT_TABLE_NAME"])


def lambda_handler(event, context):
    """
    Return the product.
    """

    path_params = event["pathParameters"]
    product_id = path_params.get("product_id")

    product = table.query(KeyConditionExpression=Key('productId').eq(product_id))

    return {
        "statusCode": 200,
        "headers": HEADERS,
        "body": json.dumps(product),
    }
```

Figure 3.5: Get Product Lambda Function


```json
{
    "category":"Burger",
    "description":"Fresh cooked burger with cheddar cheese and tomato, served on our traditional bun",
    "picture":"https://live.staticflickr.com/65535/52003329533_b9bcd159ca_k.jpg",
    "price":"14",
    "name":"Cheese Burger",
    "productId":"4c1ggdaa-223a-4ea8-ab32-58c213604e3c"

}
```


Figure 3.6: JSON Object

The Get All Products Lambda function follows the same format as the Get Products function. The Lambda functions have the same headers, libraries and database table. The Get All Products function performs a scan on the database instead of a single query. The scan returns all of the products in the database. Because the function is not querying a single product, there are no path parameters that need to be parsed from the routing URL. Once the products are retrieved from the database, the response is created using the 200 status code, the headers, and a list of all products in

JSON format.

3.3.3 Updating the Cart and Checkout

A separate API Gateway is used to access the lambda functions that add, retrieve, and update products in the database. When a user adds a product to the cart, on the front end web page, a request is sent to the API Gateway and the API routes the request to the Add to Cart function. The product id and a quantity of 1 are sent within the body of the request. The request will also include a cookie if the user is a guest or a user id if they have created an account. The Lambda function will parse the user id, product id, and quantity from the request. Once the relevant information has been extracted from the request, the Lambda function will store the values in the database table.

If a user is viewing their cart and they decide to update the quantity of a product in their cart, a request to the Update cart function is made. The Add to Cart function can perform the same action as the Update Cart function, but updating the cart is more efficient. The Update Cart function updates the values in the database table by increasing or decreasing the value of the quantity variable. To achieve this, the function copies the product and adds the quantity value from the request to the copied product. The new quantity value is then stored in the database along with the copied product. The Update Cart and Add to Cart functions allow users to change the values in their cart and the Get Cart function allows users to view the products in their cart.

Figure 3.7: Cookie Example

The Get Cart function is used to retrieve a user's cart from the Cart table. This function is called when the front end web page is refreshed, a user adds a product to their cart, or a user updates a product in their cart. The requests sent to the Lambda function contain the user id or cookie to identify which user is requesting access to their cart. An example cookie is shown in Figure 3.7. Once the Lambda function has retrieved the user's cart from the database, it returns the cart as a

list of products. The products in the list contain the information for each product as well as the quantity and the user id. Once the user has viewed their products and wants to pay for their cart, the Checkout Lambda function is used.

The Migrate Cart function migrates a guest cart to an authenticated user's account when that user signs in. When a guest user wants to checkout, they must first create an account or login to an existing account. Once the user has logged into their account, any products in their guest cart will also be in their user cart. The function transforms the guest cart to a user's cart by replacing the cookie with a user id. The user id is retrieved from Cognito when the user signs in, is stored in local memory, and then sent with the request to migrate the cart.

The Checkout Lambda function has access to two database tables. The function retrieves the cart from the Cart table and stores the finished order in the Order table. Once the cart is retrieved, the function saves a copy of the cart and deletes it from the Cart table. The function then parses the quantity and product id values from the cart as a list. This list is then stored inside the Order table with the user id. The specific information for each product is not stored in the Order table to reduce the size and cost of the database; instead product ids are stored in the Products database table.

4 Performance and Cost Analysis

4.1 Cost Comparison

First, we compared the cost of the serverless architecture with the traditional 3-tiered information system. The cost of the Lambda functions, Cognito, API Gateway, DynamoDB and the S3 bucket will be investigated in the tests.

4.1.1 Cost of Serverless Design

We inserted 1,000 randomly generated orders into the Order DynamoDB database table for the comparison. There are two other database tables that were used in the calculation: the Products table, which contains the product information, and the Cart table, which is used to store carts. All three of the databases will be included in the cost analysis.

AWS rounds up to the nearest gigabyte for storage costs and every database used for the calculation held less than one gigabyte of data. The cost for storage will be calculated as the cost for one gigabyte of data stored [17]. Each gigabyte stored in DynamoDB costs 25 cents and three databases

will cost 75 cents per month. It is worth mentioning that the storage cost is only $\frac{.75}{11.29} = 6.6\%$ of the total monthly cost. Instead, the dominate portion of the cost was incurred by the Lambda function execution with a percent of 38.7%, as shown in Figure 4.1. The number of read from, and writes to the databases was set to one million per month to allow for growth within the same price range. The 1 million writes per month costs $1.25 per database and the one million writes per month costs 13 cents per database. Which calculates to $4.89 per month for the database access cost [17].

DynamoDB	Cognito	Lambda	API Gateway	S3 Bucket
Storage Cost: $0.25 per table	first 50,000 users are free	1 million requests: $2.88	Cost for 1 million requests: $3.50	Cost for 1 GB of Storage: $0.02
Write cost: $1.25 per table		512MB storage: $0.0		
Read Cost: $0.13 per table				
Total: $0.25*3 + $1.25*3 + $0.13 * 3 = $4.89	Total: $0.0	Total: $2.88	Total: $3.50	Total: $0.02
Total: $4.89 + $2.88+ $3.50 + $0.02 = $11.29				

Figure 4.1: Cost Calculation for the Serverless Architecture

The cost of Cognito was not shown in Figure 4.1, but we examined the cost. Cognito is free for the first 50,000 users, so it was not included in the price comparison because we assume the number of users will be under 50,000. The S3 buckets cost 2 cents per gigabyte and the current application is much smaller than one gigabyte [17]. AWS rounds up to one gigabyte for the S3 buckets, so two cents was used for the comparison.

The API Gateway costs $3.50 for every 1 million requests [17]. Using multiple API Gateways for the back end does not increase the cost. The cost is calculated by the amount of requests made each month between all of the API Gateways. In the cost comparison, the amount of requests made will be increased in one million increments. The two services that will be increased in one million increments are the API Gateway and Lambda.

The cost for Lambda functions is calculated by the number of requests received as well as the time it takes for the Lambda function to process each request. The first request a Lambda function receives requires more time to process than subsequent requests. This is called a cold start. A cold start incurs larger latency because the service is initializing the cloud hardware and software. All subsequent requests the Lambda function receives are called warm starts and will be processed in a much smaller amount of time. Warm starts are much faster than cold starts because the underlying hardware and software has been initialized and cache memory has been warmed up. The Lambda function will shut down again after a short time frame and the next request after the Lambda function

has shut down will be a cold start. To average the runtime for the Lambda functions, we used one cold start and two warm starts.

We used the average of one cold start and two warm start run times because it takes into account users performing more than one action in quick succession. The average time used in the cost comparison was $(829 + 63 + 71)/3 = 321$ milliseconds. We used the Add to Cart function to retrieve the run time values because it is the function that requires the longest time to run. The cost for each one million requests to the Lambda function is $2.88. Figure 4.1 contains total cost for one million requests.

4.1.2 Cost of Traditional 3-Tiered Design with EC2 Servers

Description	EC2 On-Demand	EC2 Reservered	RDS On-Demand	RDS Reserved
CPUs	2	2	2	2
Memory	4 GB	4 GB	2 GB	2 GB
Cost per Month	$30.37	$17.75	$50.17	$42.35
Total Cost for On-Demand Each Month:	$30.37 + $50.17 = $80.14			
Total Cost for Reserved Each Month:	$17.75 + $42.35 = $60.10			

Figure 4.2: EC2 Total Cost Chart

We used the t3.medium EC2 server for the cost comparison because it is a server capable of storing an application server and is less expensive than the other options [40]. The t3.medium server has four gigabytes of memory and two virtual CPUs. The server is considered general purpose and can be used for small web servers. The t3 EC2 servers also have burstable performance, allowing the server to handle a larger amount of requests for a short period of time [40]. The t3.medium EC2 server is a low priced server in AWS.

For the on-demand pricing model, the t3.medium EC2 server costs $30.37. When the on-demand model is used, the cost for the server must be payed monthly. The EC2 server can be shut down at any time to remove this cost. AWS also offers EC2 Reserved Instances which reduce the cost of an already running server. When this model is used for the EC2 server, the cost for a year of using the service must be payed up front. If the EC2 server is removed from the account, the cost for the year will still be charged [41]. If the reserved model is used, then a heavy discount will be added to the EC2 server on-demand cost. Both of these models, with the cost of a database, are shown in Figure 4.2.

The Relational Database Service (RDS) was used instead of a DynamoDB database table. RDS

can be connected with an EC2 server, a small RDS costs $50.17 per month [17]. The cost can be reduced by using a Reserved Instance and paying for the cost of the service upfront. If the cost for a year is payed upfront, the cost for the RDS per month is reduced to $42.35 [17]. When the RDS is included, The total cost, per month, for the on-demand EC2 server is $80.14. If the reserved model is used and the cost is payed in advance for both services, the total cost is $60.10.

4.1.3 Cost Comparison Between the Serverless and 3-Tiered Architecture

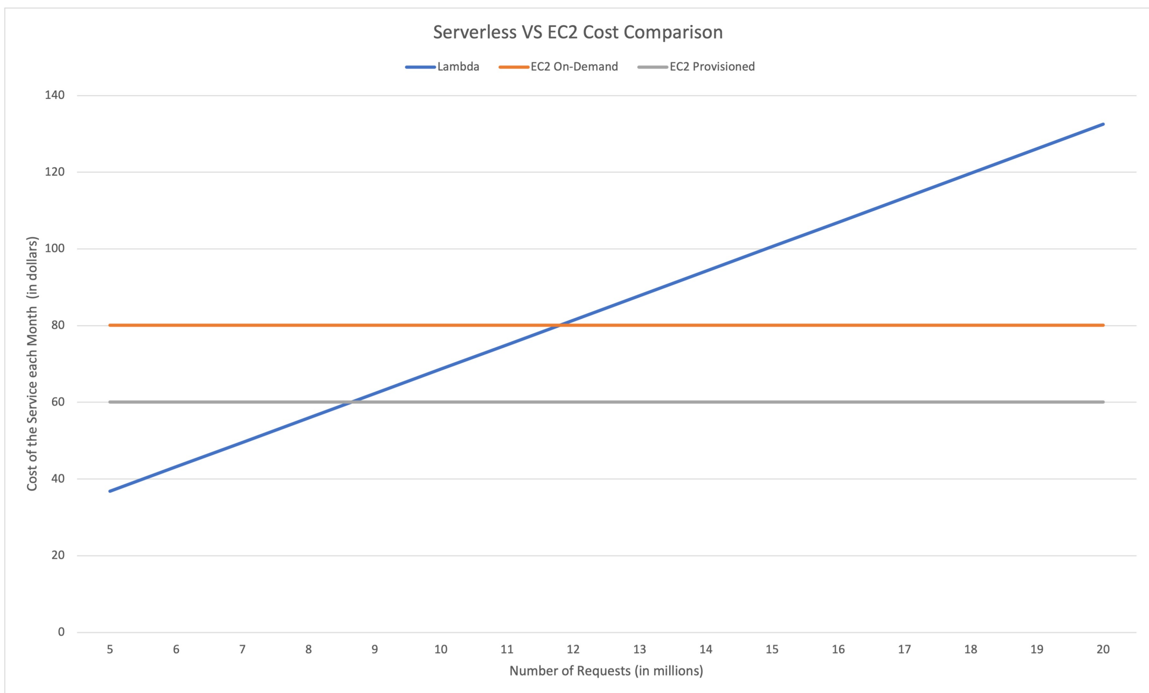


Figure 4.3: Cost Analysis of Lambda Function vs EC2 Machine

Figure 4.3 represents the cost comparison we performed with the two architectures. The chart increases by one million request increments and it includes the base price for all services listed above as well as an increase in the price for Lambda and the API Gateway. The Lambda and API Gateway cost increase for one million request increments. Based on Figure 4.3, when the back end receives between 11 and 12 million requests per month, the on-demand EC2 machine becomes the more cost-effective approach. If the provisioned cost model is being used, the EC2 machine becomes more cost-effective between 9 and 10 million requests. If a company has more than 9 million incoming requests per month, then using a provisioned EC2 server is a cost-effective approach.

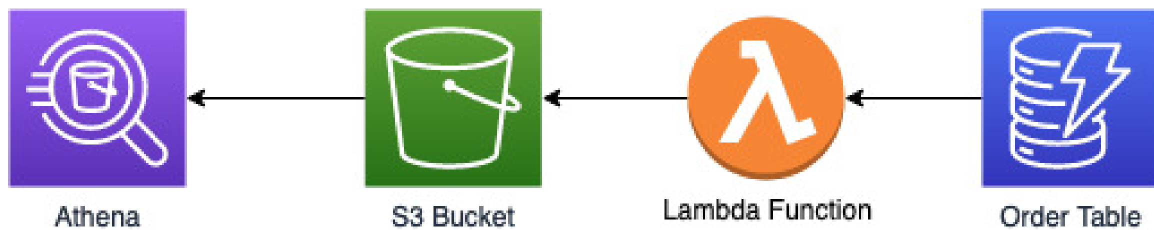


Figure 4.4: Athena Query Flow

4.2 Athena Data Analytics

Because DynamoDB cannot perform complex joins between tables, we used AWS Athena to perform data analytics. The 1,000 orders stored in the Order table were used for the data analytics in this section. Each order stored in the database table was created with three random products and a random quantity between one and ten. The data was moved to Athena using Athena's built in DynamoDB connection option so more complicated queries could be run. We used Athena to perform a join on the products database and the order database.

Athena provides several methods to retrieve data from a DynamoDB table. To use one of these solutions, a data source must be created. Data sources are used as a connection between Athena and the data source. When a data source is first created, the type of data source must be selected. For the purposes of this test, we chose DynamoDB as the data source. Once the data source is chosen, Athena must be connected to a Lambda function. The Lambda function will retrieve the data from DynamoDB and store it in an S3 bucket. Athena will then perform queries on the data in the S3 bucket.

Figure 4.5 represents an example query that returns the revenue gained from each product on the menu. The SQL query flattens the list of products in each order so they are in one column. This is done with the cross join unnest command. Once completed, the sum of each price is computed by multiplying the corresponding quantity with the price and summing up the result. The price is retrieved from the products database by joining the two tables with the product id's. The two columns that are returned are the name of the product and it's total revenue. The result is then grouped by the name to create an easy to read format.

The query requires a little over seven seconds to run. Any subsequent runs for the same query require five seconds to finish. The amount of data scanned for the query is 183 kilobytes. Athena rounds up all queries to ten megabytes when calculating the cost for the month. Because the data

```
SELECT menu.name AS "name",
    SUM(
        allCartItems [ 1 ].quantity * CAST(menu.price AS INT)
    ) AS "profit"
FROM "restaurantorders" AS "orders"
    CROSS JOIN UNNEST(orders.cartItems) AS t(allCartItems)
    INNER JOIN "menuitems" AS "menu" ON allCartItems [ 1 ].productid = menu
        .productid
GROUP BY menu.name
```

Figure 4.5: Athena Profit Query

scanned is much smaller than ten megabytes, there is room for many queries to be made without increasing the cost for the service [17].

Once the query is finished, the columns are displayed below the editor and can be exported. We exported the results to an Excel file to represent the analytical value that Athena can provide. Figure 4.6 represents the query results as a bar chart. The X-axis is the name of the product and the Y-axis is the revenue in dollars. Each product was chosen at random so the chart does not represent any relationship between the products and the revenue. We created this chart to represent the capabilities of Athena.

A query which displays the total revenue gained from each client was also created to represent the capabilities of Athena. The revenue was sorted in descending order to display the high spenders. Figure 4.7 displays the code used for this query. The columns returned in the results are the user id and the revenue. The values are grouped by user id, and the revenue is sorted in descending order. The query requires nine seconds to run. Any subsequent queries require five seconds to complete. The data returned is 222KB, which is lower than ten megabytes.

AWS Athena is a powerful tool that can perform useful queries. With a more specific business model, Athena will be a useful tool to create queries and perform data analytics on the database tables. For example, Athena can be used to show the profit made each month as well as get insights into the popularity of each product. Athena or a third-party software is necessary to query the database and perform data analytics when DynamoDB is used for data storage.

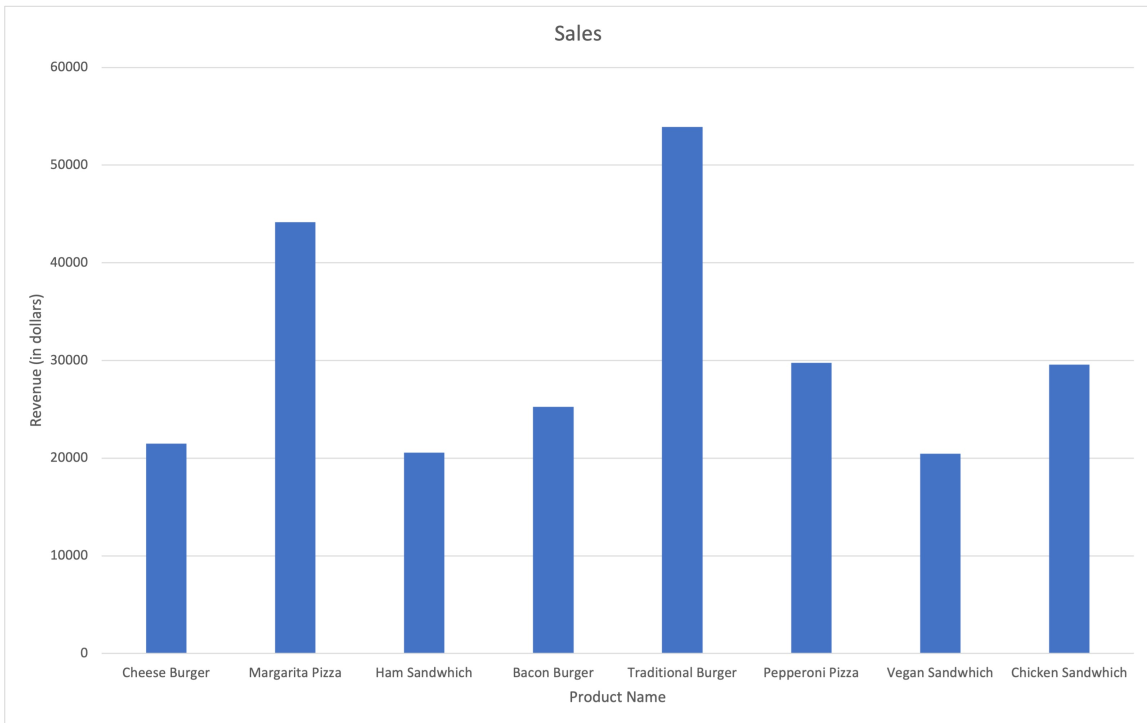


Figure 4.6: Sales Report for Each Product

```
1   SELECT orders.userid AS "userid", SUM(allCartItems[1].quantity * CAST(menu.price AS
        INT)) AS "profit"
2   FROM "restaurantorders" AS "orders"
3   CROSS JOIN UNNEST(orders.cartItems) AS t(allCartItems)
4   INNER JOIN "menuitems" AS "menu" ON allCartItems[1].productid = menu.productid
5   GROUP BY orders.userid
6   ORDER BY profit DESC;
7
8
9
10
11
12
```

Figure 4.7: Athena Users Revenue Query

4.3 Performance Comparison

In this section we compare the execution time cost of the serverless architecture and the 3-tiered architecture. First, we will calculate the performance of the EC2 machine. The following section will calculate the performance of the serverless architecture.

4.3.1 Execution Time of Traditional 3-Tiered System

We created an EC2 server with an Apache web server and an RDS to test the execution time. The web server used PHP for the back end and HTML for the front end. The RDS was accessed through MySQL commands. The latest version of PHP, Apache, HTML, and MySQL were used. The specific versions are displayed in Figure 4.8. The front end and the back end of the system is stored in the EC2 machine and the database is stored in the RDS.

The front end of the web server retrieves a user id and a description from the user. Once the values are retrieved from the user, a call to the database stored in the RDS is made. The values are stored in the database and then retrieved and displayed on the web page. The server was setup to represent a product being added to a user's cart. The database calls were much faster than the serverless architecture.

Software Name	Software Version
PHP	PHP 8.0
Apache	Apache 2.4.54
HTML	HTML5
MySQL	MySQL 8.0

Figure 4.8: Web Server Software Version Chart

After ten runs, the longest database call for the EC2 machine was ten milliseconds. The time was calculated by retrieving the current time before the database call as well as the time after the database call was completed. The time after the database call was then subtracted from the initial time to calculate the total time needed to perform the call.

4.3.2 Execution Time of Serverless System

To calculate the efficiency of the serverless system, we tested the time it takes the Lambda functions to run. The Lambda functions are the most time consuming service in the back end. We tested the efficiency of the Lambda functions by separating them into cold start times and warm start times. For the cold starts we ran each Lambda function five times and recorded the maximum value.

Figure 4.9 represents each functions longest cold start time. The functions with the longest run-time are the login, register and add to cart functions respectively. These functions require the most

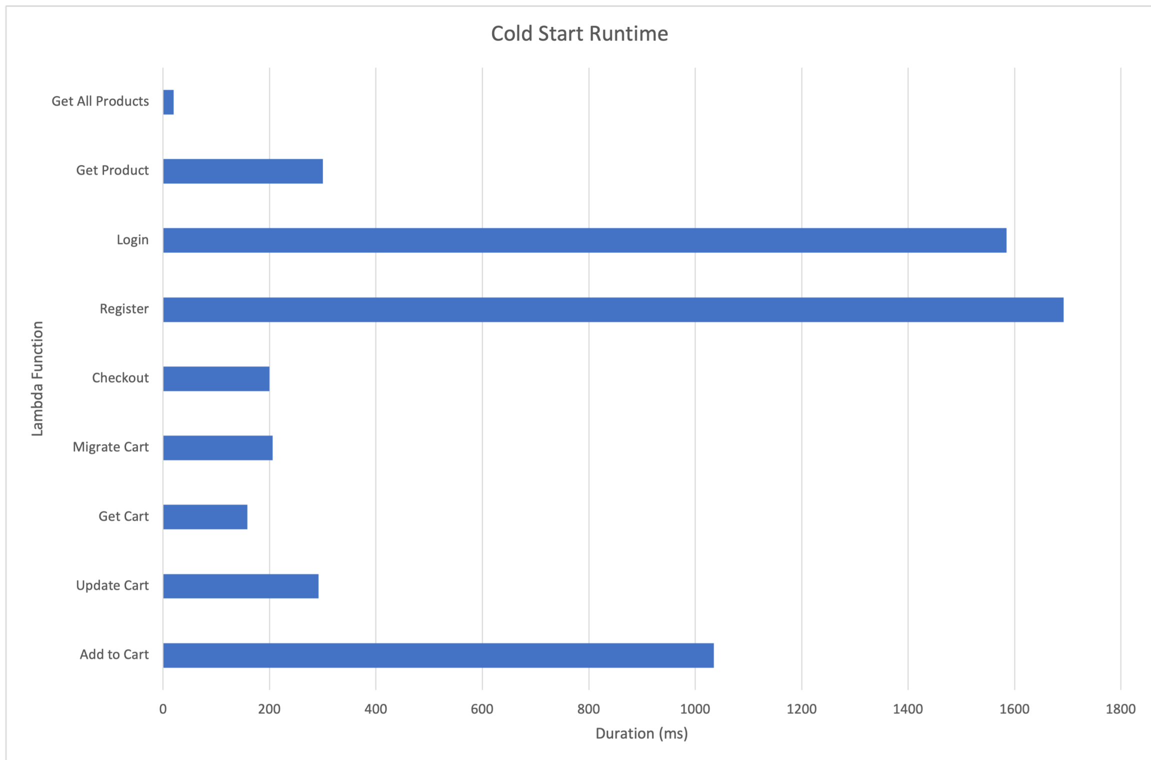


Figure 4.9: Cold-Start Execution Time

time to run because they perform more computationally intense tasks than the other functions. The cold start time for the login and register functions is longer than one second. These two functions need to send a request to Cognito which requires more time. The Add to Cart function is the most used function in the system and requires one second to run a cold start.

To reduce the time the Add to Cart function requires to run, we created a dummy request to warm the function. The dummy request is sent whenever the page is reloaded. Having a dummy request will increase the cost of the function by a small margin, but the decrease in runtime is significant. The runtime for the Add to Cart function is less than 100 milliseconds for a warm start as seen in Figure 4.10.

The increase in efficiency is significant for the other Lambda functions. Figure 4.10 represents the function's average runtime after a cold start. We ran each of the functions in Figure 4.10 ten times after the initial cold start. The average of the ten runs was used for the graph. The Login, Register, Migrate Cart, and Checkout functions were not included because most users will only run them once.

The Add to Cart, Update Cart and Get Product Lambda functions have the largest increase in

efficiency once the function is warmed up. The Add to Cart function's runtime decreases from over one second to less than 100 milliseconds on average. The Update Cart and Get Product Lambda functions runtime decreases from 250 milliseconds to less than 90 milliseconds. With the dummy request reducing the runtime for the Add to Cart function, all of the functions that are used for the cart require less than 400 milliseconds to run.

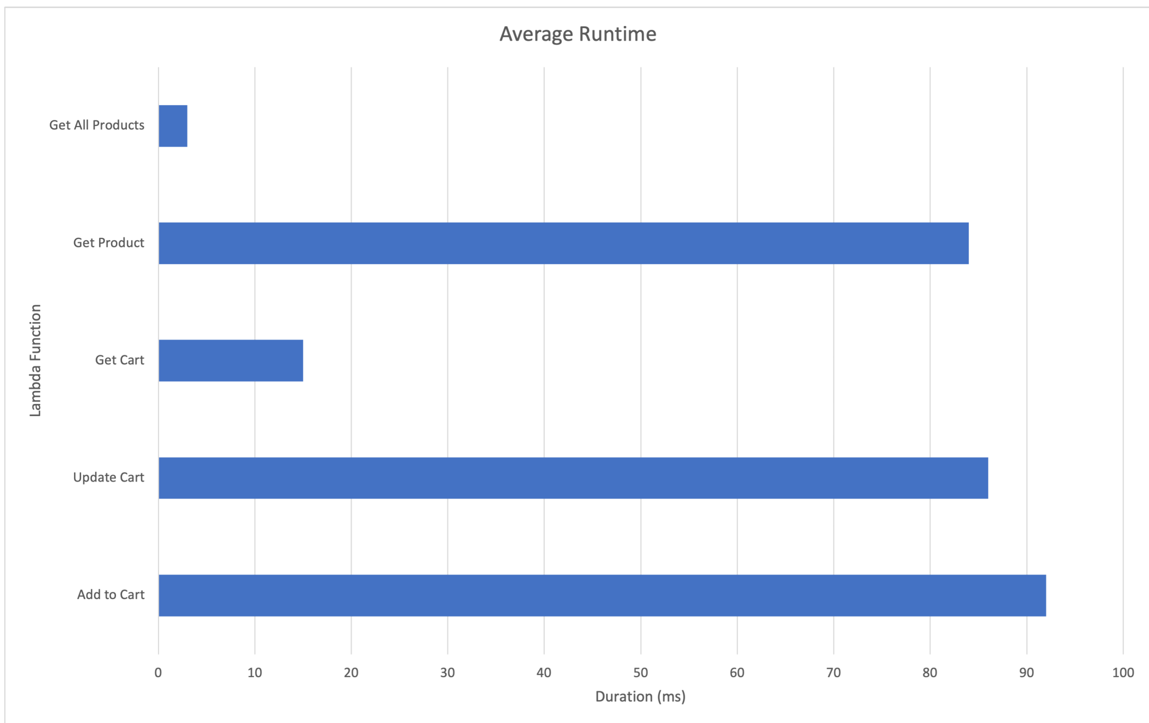


Figure 4.10: Average Warm Start Execution Time

5 Conclusion

Our serverless architecture is a good choice for smaller businesses trying to reduce financial cost for their information system. The cost for the serverless architecture uses a pay-as-you-go model. During the hours of the day when traffic is low, the infrastructure will cost less than when the traffic is high. The benefit for smaller businesses is the website is available to customers without paying for the idle time.

Our serverless design will cost less than the on-demand 3-tiered traditional design as long as the number of requests is less than 11 million per month. If a reserved instance is used for both the EC2 machine and the RDS, then our proposed system is more cost-effective while the number of requests, per month, is below 8 million. A disadvantage of our design is that DynamoDB cannot

produce advanced reporting for the system.

DynamoDB does not allow for complicated joins, unlike the traditional relational database. Because DynamoDB does not support joins, creating complex reports is not possible. To solve this problem, we used AWS Athena to perform the complicated joins and create complex reports. With Athena, multiple databases can be exported into S3 buckets and then complicated joins can be performed. This mitigates the disadvantage of a NoSQL database, but will cost extra money for each report generated. However, complex report generation is performed much less frequently in practice.

Our serverless design is more cost-effective than the traditional 3-tiered design, but it is not more efficient. The EC2 machines in the traditional design more efficient than the Lambda functions in the serverless design. The Lambda functions need to be initialized, receive the request, process the request, and then send a response back to the front end. Performing an initial dummy request to remove the cold start time is feasible. Performing the dummy start will increase the Lambda function's efficiency, but the Lambda function will still process requests slower than the EC2 machine. For most information systems, a delay of 100 milliseconds will not greatly undermine the user experience. However, if the application must provide feedback to the user in under 100 milliseconds, then an EC2 machine is the better choice.

6 Future Work

To increase the usability of the serverless application we built, we will share an easy to build version of the product publicly on the internet. The application is currently stored in GitHub and can be accessed by the public. However, the GitHub repository only contains the code for the functions and the front end. If other developers want to use the code, they need to create their own functions in AWS and import the code. To simplify this process, the code can be used in conjunction with CloudFormation to allow anyone to build the entire project using a CloudFormation script. The CloudFormation script contains instructions that will deploy the entire information system in a more convenient way.

The other improvement that could be made on this project will be to increase the amount of reporting tools. Athena has the capability generate reports which could be useful to an e-commerce business. For example, reports on customer loyalty could be used to reward repeat customers with

discounts or promotions. Athena could be used to create a report showing what time of the day users place their orders. In the future, we will add these functionalities into the proposed information system.

References

[1] W. Hong, J. Y. Thong, and K. Y. Tam, "Designing product listing pages on e-commerce websites: an examination of presentation mode and information format," *International Journal of Human-Computer Studies*, vol. 61, no. 4, pp. 481–503, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S107158190400014X

[2] K. Abhinav, M. Vishal, and N. Erich, "Controlling the performance of 3-tiered web sites: Modeling, design and implementation," Jun 2004. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1005686.1005744?casa_token=YchoGA1K4PMAAAAA:3f7iDd-bETKLiUPD0AkVFDlKlabFiRfX7PDTUtkbympEw_9eUXDFSP-ELGdW91WNV9l_WCgAobBGMw

[3] X. Liu, J. Heo, and L. Sha, "Modeling 3-tiered web applications," in *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2005, pp. 307–310.

[4] R. L. Grossman, "The case for cloud computing," *IT Professional*, vol. 11, no. 2, pp. 23–27, 2009.

[5] S. Ibrahim, B. He, and H. Jin, "Towards pay-as-you-consume cloud computing," in *2011 IEEE International Conference on Services Computing*, 2011, pp. 370–377.

[6] E. Brynjolfsson, P. Hofmann, S. L. i. P. Alto, and J. Jordan, "Cloud computing and electricity: Beyond the utility model: Communications of the acm: Vol 53, no 5," May 2010. [Online]. Available: https://dl.acm.org/doi/fullHtml/10.1145/1735223.1735234?casa_token=VIX72dJv_5IAAAAA\%3Aw5k4DZ5L-rKTj_IsBSoi5ovzjUcY-QGMKRUNAhnYWrhR4D-yQDvHrRkz0YkFFlSyZQqYt9Nuq4bi

[7] I. Bermudez, S. Traverso, M. Mellia, and M. Munafò, "Exploring the cloud from passive measurements: The amazon aws case," in *2013 Proceedings IEEE INFOCOM*, 2013, pp. 230–234.

[8] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center," in *2010 Proceedings IEEE INFOCOM*, 2010, pp. 1–9.

[9] [Online]. Available: https://aws.amazon.com/route53/what-is-dns/

[10] J. Nadon, *Database Services in AWS*. Berkeley, CA: Apress, 2017, pp. 127–151. [Online]. Available: https://doi.org/10.1007/978-1-4842-2589-9_10

[11] L. N. G., "Database migration on premises to aws rds," *EAI Endorsed Transactions on Cloud Systems*, vol. 3, no. e11, 4 2018.

[12] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[13] K. Kritikos and P. Skrzypek, "A review of serverless frameworks," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 161–168.

[14] [Online]. Available: https://aws.amazon.com/lambda/

[15] [Online]. Available: https://aws.amazon.com/api-gateway/

[16] J. Han, H. E, G. Le, and J. Du, "Survey on nosql database," in *2011 6th International Conference on Pervasive Computing and Applications*, 2011, pp. 363–366.

[17] [Online]. Available: https://calculator.aws/#/

[18] [Online]. Available: https://www.oracle.com/internet-of-things/what-is-iot/

[19] R. A. P. Rajan, "Serverless architecture - a revolution in cloud computing," in *2018 Tenth International Conference on Advanced Computing (ICoAC)*, 2018, pp. 88–93.

[20] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 300–312.

[21] D. Bardsley, L. Ryan, and J. Howard, "Serverless performance and optimization strategies," in *2018 IEEE International Conference on Smart Cloud (SmartCloud)*, 2018, pp. 19–26.

[22] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," 2018. [Online]. Available: https://arxiv.org/abs/1812.03651

[23] R. Cordingly, H. Yu, V. Hoang, Z. Sadeghi, D. Foster, D. Perez, R. Hatchett, and W. Lloyd, "The serverless application analytics framework: Enabling design trade-off evaluation for serverless software," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, ser. WoSC'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 67–72. [Online]. Available: https://doi.org/10.1145/3429880.3430103

[24] M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, and V. Yussupov, "Modeling and automated deployment of serverless applications using tosca," in *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*, 2018, pp. 73–80.

[25] S. Hong, A. Srivastava, W. Shambrook, and T. Dumitras, "Go serverless: Securing cloud via serverless design patterns," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA: USENIX Association, Jul. 2018. [Online]. Available: https://www.usenix.org/conference/hotcloud18/presentation/hong

[26] A. Mujezinovix0107; and V. Ljubovix0107;, "Serverless architecture for workflow scheduling with unconstrained execution environment," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 242–246.

[27] D. Chahal, R. Ojha, M. Ramesh, and R. Singhal, "Migrating large deep learning models to serverless architecture," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2020, pp. 111–116.

[28] A. Parres-Peredo, I. Piza-Davila, and F. Cervantes, "Building and evaluating user network profiles for cybersecurity using serverless architecture," in *2019 42nd International Conference on Telecommunications and Signal Processing (TSP)*, 2019, pp. 164–167.

[29] M. S. Kurz, "Distributed double machine learning with a serverless architecture," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, ser.

ICPE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 27–33. [Online]. Available: https://doi.org/10.1145/3447545.3451181

[30] M. M. Rahman and M. Hasibul Hasan, "Serverless architecture for big data analytics," in *2019 Global Conference for Advancement in Technology (GCAT)*, 2019, pp. 1–5.

[31] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with OpenLambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, Jun. 2016. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[32] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1288–1296.

[33] [Online]. Available: https://aws.amazon.com/cognito/

[34] [Online]. Available: https://aws.amazon.com/cloudwatch/

[35] [Online]. Available: https://reactjs.org/

[36] [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP

[37] [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

[38] M. Contributers, "Cross-origin resource sharing (cors) - http: Mdn," 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[39] M. Labs, "What are computer cookies?" Sep 2021. [Online]. Available: https://www.malwarebytes.com/blog/news/2021/09/what-are-computer-cookies

[40] [Online]. Available: https://aws.amazon.com/ec2/instance-types/t3/

[41] [Online]. Available: https://aws.amazon.com/ec2/pricing/reserved-instances/

Vita

Author: Isaac C. Angle

Place of Birth: Knoxville, Tennessee

Undergraduate Schools Attended: Spokane Community College,
Whitworth University

Degrees Awarded: Bachelor of Science, 2020, Whitworth University

Honors and Awards: Graduate Assistantship, Computer Science Department, 2020-2022, Eastern Washington University