Serverless Web Application

A major project report submitted in partial fulfillment of the requirement for the award of degree of

Bachelor of Technology

in

Computer Science & Engineering / Information Technology

Submitted by

Arjit Upadhyay (201420)

Archit Kaushal (201429)

Under the guidance & supervision of

Dr. Pankaj Dhiman

Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology,

Waknaghat, Solan - 173234 (India)

CANDIDATE'S DECLARATION

I hereby declare that the work presented in this report entitled **'Serverless Web Application'** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science & Engineering / Information Technology** submitted in the Department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology, Waknaghat is an authentic record of my own work carried out over a period from August 2023 to December 2023 under the supervision of **Dr. Pankaj Dhiman** (Assistant Professor (SG), Department of Computer Science & Engineering and Information Technology).

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Student Name: Arjit Upadhyay	Student Name: Archit Kaushal
Roll No.: 201420	Roll No.:201429

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

(Supervisor Signature with Date)
Supervisor Name: Dr. Pankaj Dhiman
Designation: Assistant Professor (SG)
Department: Computer Science & Engineering and Information Technology
Dated:

ACKNOWLEDGEMENT

First, we express our heartfelt thanks and gratitude to Almighty God for His divine blessing that made it possible to complete the project work successfully.

We are really grateful and wish our profound indebtedness to Dr. Pankaj Dhiman, Assistant Professor (SG), Department of CSE & IT, Jaypee University of Information Technology, Waknaghat. Deep knowledge and keen interest of our supervisor in the field of "cloud computing" to carry out this project. His endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, and reading many inferior drafts and correcting them at all stages have made it possible to complete this project.

We would also generously welcome each one of those individuals who have helped us straightforwardly or in a roundabout way in making this project a win. In this unique situation, we might want to thank the various staff individuals, both educating and non-instructing, who have developed their convenient help and facilitated my undertaking.

Finally, we must acknowledge with due respect the constant support and patience of our parents and siblings.

Archit Kaushal
Project Group No. 191
Roll No.: 201429

Arjit Upadhyay
Project Group No. 191
Roll No.: 201420

TABLE OF CONTENT

TITLE	1
CANDIDATE'S DECLARATION	I
ACKNOWLEDGEMENT	II
LIST OF TABLES	1-1
LIST OF FIGURES	2-2
ABSTRACT	1 - 1
CHAPTER 1: INTRODUCTION	2- 9
CHAPTER 2: LITERATURE SURVEY	9- 18
CHAPTER 3: SYSTEM DEVELOPMENT	19-33
CHAPTER 4: TESTING	34-49
CHAPTER 5: RESULTS AND EVALUATION	50-62
CHAPTER 6: CONCLUSION AND FUTURE SCOPE	63-68
REFRENCES	69-73

LIST OF TABLES

Page No. Table 2.2.1: List of the Literature Surveys 13-18

Table 5.2.2: Comparison with Existing Solutions 38-37

Table 5.2.3: Comparison with Existing Solutions 39-40

LIST OF FIGURES

Page No.

Figure 1.1: Traditional Vs Serverless Architecture. 3

Figure 3.1.1: System Design. 22

Figure 3.1.2: System Design. 23

Figure 3.1.3: System Design. 24

Figure 3.1.4: System Design. 25

Figure 3.4.2: Both the figures show the snippets to connect to console and create the repositories 25

Figure 3.4.3: Both the figures shows the implementation of above mentioned commands on Window Powershell 29

Figure 3.4.4: Both the figures shows the implementation of above mentioned commands on Window Powershell 30

Figure 5.1.2.1Events on AWS. All events listed on AWS are displayed in the left column, while all events listed on the Serverless Framework are displayed in the right column. 32

Figure 5.1.3: AWS Dashboard when the serverless application is deployed 35

ABSTRACT

In recent years, serverless applications have grown in popularity. The developer cannot access the settings or usage of the server. It is promoted as a technique to shorten development times and simplify the process. The cost of the programme is determined by the real time and resource utilisation, and it is automatically scaled based on usage. Since cloud infrastructure may be viewed like code, development can be accelerated with the help of a framework. Regardless of the underlying platform, it simplifies the deployment process and makes it easier to deploy the same application to numerous serverless providers.

In the serverless space, two phrases that are frequently used are Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS). By segmenting an application into its component parts, FaaS enables the re-deployment of a single function without requiring the re-deployment of the entire application. Application development can be accelerated by using BaaS services, such as databases and authentication, that are provided by serverless providers.

The distinctions between serverless and conventional server-oriented development are examined in this thesis. An empirical study is carried out in which a serverless application is deployed to three providers—AWS, Azure, and Google—using the Serverless Framework. It looks into whether BaaS services can be used instead of outside solutions for database and authentication. The purpose of the thesis is to determine which provider is most suited for a front-end React application that is connected to a smaller back-end API. The comparison of setup and deployment similarities and differences tries to determine the degree of code reuse throughout serverless providers.

The outcome demonstrates that AWS is the target of the Serverless Framework by default. For the Azure and Google project to interface with the functions and events of each provider, a function plugin is needed. A back-end application was developed in AWS using the BaaS services Cognito User Pool for authentication and DynamoDB for database management. There were relatively few BaaS services defined in the documentation from Google and Azure, and no BaaS services could be implemented.

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

The serverless notion is promoted as a means of reducing both the complexity and the amount of time needed to construct an application. When selecting serverless for a smaller back-end application, many of the advertised advantages—such as scalability and pay-as-you-go—are disadvantages. For developers who are new to serverless, it may take some time to become comfortable with Function as-a-Service (FaaS) and Backend-as-a-Service (Baas). FaaS refers to the division of an application into smaller functions, each of which often has a particular purpose, such adding a post to a database. It is significantly simpler to individually re-deploy these functions. Less code is possible when using BaaS services from serverless providers, and you can take advantage of their experience setting up things like databases and user authentication. It does, however, result in vendor lock-in, making it potentially challenging to move to a different supplier. Learning new services and configurations would be necessary when moving providers because the necessary setups and knowledge for the services used will vary depending on the provider.

The phrase "serverless" has gained popularity in the last several years. It explains a novel approach to application publishing in which developers are not aware of the server utilisation. The code is automatically resized according to usage and executed on demand. The billing is determined by the real-time execution of the code. The cloud resource provider is in charge of autoscaling, resource allocation, and deployment. Google Cloud, Microsoft Azure, and Amazon Web Services (AWS) are a few popular serverless computing services.

The names FaaS and BaaS are frequently used in the serverless context. Developers can use provider-managed services like file storage, cloud-accessible databases, and authentication with BaaS. Developers can install their own code (functions) on servers or containers that the cloud provider manages thanks to FaaS. These two phrases, which occasionally get confused, describe the developer's level of freedom and control over the code that is deployed. BaaS may speed up application development but results in increased vendor lock-in and less control over the code utilized.

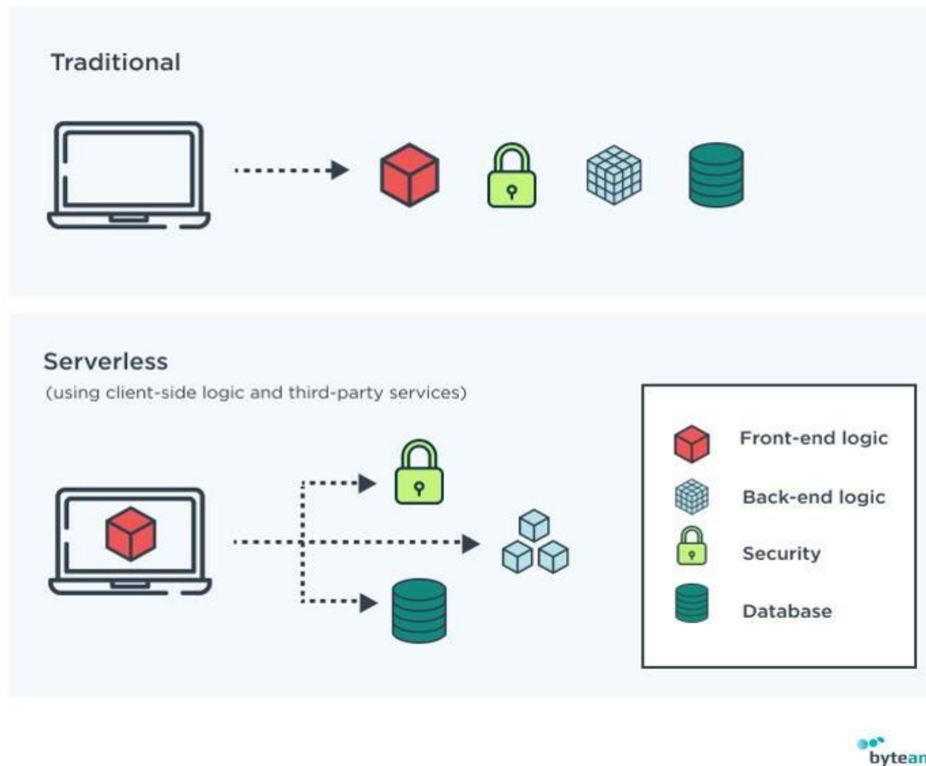


Fig 1.1 Traditional Vs Serverless Architecture.

As traditional development uses fixed resources and that's why it's harder and more expensive to maintain them, scale them. The other issue is downtime. The main aim of the project is to deal with these issues and provide best applications of cloud computing by using serverless web applications that provides us auto scalability, cost efficiency as it works on pay per use model, high availability and fault tolerance and it also helps in reducing the development time and providing complex infrastructure management.

1.2 PROBLEM STATEMENT

Provisioning and managing dedicated server resources are part of traditional hosting methods for web applications. This frequently results in either under-provisioning during traffic spikes, which results in subpar performance and dissatisfied users, or overprovisioning during peak traffic, which raises operational expenses. By tackling the following crucial problems, serverless web applications seek to address these difficulties:

1. **Scalability**: To adapt to changing user loads, traditional programmes require manual resource adjustments. The serverless architecture allows seamless resource allocation based on demand and automatic scaling. By doing away with anticipatory provisioning, performance and user experience are improved.

2. **Efficiency in terms of costs**: Over-provisioning servers to handle peak traffic might result in resource waste and irrational expenses. Serverless applications ensure optimal resource utilization while minimizing operational costs by dynamically allocating resources as needed.

3. **Operational Complexity**: Scaling servers requires complex setups and upkeep. Server management is abstracted away by serverless web applications, freeing developers to concentrate entirely on writing business logic without worrying about infrastructure maintenance.

4. **Reduced Latency**: In serverless architecture, cold start latency, or the delay encountered on the first invocation of a function, is a typical worry. Applications become more responsive as a result of solving this problem, which enhances user experience.

5. **Flexibility**: Microservices-based composition provided by serverless architecture allows for the creation of modular applications that may be independently created, deployed, and scaled. Iterative development is facilitated and adaptability is improved.

6. **Resource Distribution**: Serverless technologies distribute resources automatically according to workload demands, negating the need for human modifications. By doing this, programmes may manage variable loads without suffering performance degradation.

7. **Developer Productivity**: With serverless, developers can concentrate on writing code because the cloud provider is in charge of maintaining the infrastructure underneath. This increases developer productivity and shortens the time it takes to construct an application.

8. **Innovation:** Serverless architecture fosters innovation and experimentation by abstracting away infrastructure-related concerns. Developers may swiftly iterate and develop new features by prototyping and deploying them.

9. **Global Reach**: Without the hassles of managing server clusters, serverless apps can be distributed across numerous geographical areas. This improves the accessibility and responsiveness of the programme for a global user base.

10. **Automatic Failover**: Serverless solutions frequently come with built-in failover and redundancy methods, boosting application resilience and reducing downtime in the event of failures.

1.3 OBJECTIVES

1. **Eliminating server management complexities**: Serverless computing eliminates the need for businesses to manage their own servers. This frees up developers to focus on their core business logic and reduces the risk of human error.

2. **Optimizing resource allocation and utilization**: Serverless computing allows businesses to pay for the resources they use, which can help to reduce operational

expenses. Additionally, serverless computing can automatically scale resources up or down based on demand, which can help to improve efficiency.

3. **Achieving automatic and seamless resource scaling:** Serverless computing can automatically scale resources up or down based on demand, which ensures that applications always have the resources they need to perform at their best. This can be especially beneficial for businesses with fluctuating workloads.

4. **Minimizing cold start latency**: Cold start latency is the delay that occurs when a serverless function is first invoked. Serverless providers can minimize cold start latency by pre-warming functions or using warm start functions. This can help to improve the performance of serverless applications.

5. **Enabling modular development and independent scaling of microservices**: Serverless computing can help businesses to develop and scale microservices independently. This makes it easier to build and maintain complex applications.

1.4 SIGNIFICANCE/MOTIVATION OF PROJECT WORK

Both the demand for and availability of serverless apps has grown, as has the number of service providers. The product is positioned as a means of cutting down on both the amount of time needed to develop an application and its complexity. This study will look at whether the method is now developed sufficiently for smaller applications and whether it can eventually take the place of traditional back-end development. It will include putting serverless apps into practise. As stated in the essay Why, When, and How of Serverless Applications? To help software developers create serverless solutions, additional empirical research on serverless use is required.

The bar for serverless configurations has lowered recently thanks to the rise in popularity of serverless frameworks. Still, is it still too difficult for smaller apps or projects to achieve the barrier needed to set up FaaS and BaaS within a serverless framework? Anyone exploring the

use of serverless computing can benefit from reading this report. It will look into which of the providers is worth spending money and time developing.

1.5 ORGANIZATION OF PROJECT REPORT

This report is divided into 6 different chapters covering all the aspects of the project we have worked on; it is extremely important to know the structure/ organization of the project because it helps us understand the methodology and the ideology of the project better. Let us discuss about the various chapters briefly:

Chapter 1: This chapter contains the 1.1 INTRODUCTION, 1.2 PROBLEM STATEMENT, 1.3

OBJECTIVES, 1.4 SIGNIFICANCE/ MOTIVATION OF THE PROJECT which tells us in detail about what the project is, why did we take up this project, why is this monitoring of this project is crucial and also how will our project make a difference and impact from it. It also clearly defines the objectives we want to achieve from the project and since we further want to convert this project into a web application The major objective of this Project is to Get practical experience with serverless architecture and create applications that are scalable and economical. Boost application performance, time to market, and development agility. Investigate real-world serverless systems and develop a useful application that integrates cloud services. With the help of Cloud and AWS technology, get practical knowledge about AWS's serverless architecture: Use the event-driven compute service AWS Lambda to create and implement serverless web apps with ease. Create scalable and affordable applications by utilizing AWS services: Employ AWS services like Amazon DynamoDB, Amazon Cognito, and Amazon API Gateway to build scalable, affordable online apps. Boost time to market and development agility with AWS Amplify: Use the serverless web and mobile app development framework AWS Amplify to expedite development and speed to market.

Chapter 2: This chapter contains the 2.1 OVERVIEW OF RELEVANT WORK, 2.2 KEY GAPS IN LITERATURE, HIGHLIGHTING the work other people have done in this particular field and the gaps we have found that lead us to make this project and stand this out of the other work any other researcher has done in this domain.

Chapter 3: This chapter contains the 3.1 REQUIREMENT AND ANALYSIS, 3.2 PROJECT DESIGN AND ARCHITECTURE, 3.3 DATA PREPARATION, 3.4 IMPLEMENATION, 3.5 KEY CHALLENGES which will show the system development process. How we have gathered the data and how we will move forward with the preparation, splitting, augmentation, preprocessing of the data and while we were developing the system what were the key challenges we have faced recently and how did we overcome those. We have used various AWS technologies like (AWS lambda, AWS API gateway, AWS congnito) in the project along with transfer learning approach and we will have to show the detailed architecture of the model and also the requirement analysis for the project, whether they are the functional or the nonfunctional requirements in the hierarchy.

Chapter 4: This chapter contains the 4.1 TESTING STRATEGY, 4.2 TEST CASES AND OUTCOMES, it is extremely crucial to a model how we test the outcomes and how we train it. Adopt a thorough testing strategy: To guarantee the general dependability and quality of the serverless application, use a mix of unit, integration, performance, and security testing. Leverage automation frameworks: To increase testing productivity, guarantee consistent test execution, and automate repetitious test cases, make use of frameworks such as Jest or Mocha. Include testing in the continuous integration and delivery (CI/CD) pipeline to automate test execution and deployment. This will enable quick feedback loops and quicker release cycles. Use cloud-based testing tools: To test serverless functions in a realistic setting, make use of cloud-based testing tools such as AWS Lambda Test or Azure Functions Test. Think about UAT (user acceptance testing): Real users should participate in UAT to provide input on usability, performance, and overall user experience.

Chapter 5: This chapter contains the 5.1 RESULTS (PRESENTATION FINDINGS) , 5.2 COMPARISON WITH EXISTING SOLUTION which is why we have done this project to have seen how it is better than the other models or how is there a different solution or path of reaching the solution with same accuracy with less parameters and effort. The major objective of this Project is to Get practical experience with serverless architecture and create applications that are scalable and economical. Boost application performance, time to market, and

development agility. Investigate real-world serverless systems and develop a useful application that integrates cloud services.

Chapter 6: This chapter contains the 6.1 CONCLUSION, 6.2 FUTURE SCOPE which has been deeply explained in the chapter.

CHAPTER 2

LITERATURE REVIEW

2.1 Overview of Relevant Literature

In the following chapter, we have reviewed some of the papers that have been published by some of the best researchers who are working in this respected field. Here we have mentioned the papers and what technologies and trends they worked on. We have done a literature review of papers that have worked on different applications, current trends, and obstacles in the field of serverless computing.

1. In IEEE Internet Computing, Aloqaily and Zomaya [1] investigated "Serverless Computing: Current Trends and Open Challenges" (Sept.-Oct. 2023). Their research adds to our understanding of this dynamic field by addressing current trends and highlighting obstacles in the ever-changing field of serverless computing. This paper tells us majorly about the work done and currently being done in serverless computing and also about the various problems that are being faced while working on it.

2. In the IEEE Transactions on Emerging Topics in Computing, Nguyen et al. [2] published "A Survey on Serverless Computing: Architectures, Applications, and Future Trends" (Jan.-Feb. 2022). Offering a thorough overview of the industry, the review explores a variety of serverless computing topics, such as architectures, applications, and future developments. This paper gives an overview of the current and future applications of serverless computing. We can see the architecture of serverless computing in this.

3. "Efficient Serverless Computing: A Survey of Recent Advances and Future Research Directions" was presented by Patel and Shukla [3] in the December 2021 issue of ACM Computing Surveys. Their survey offers insightful information for scholars and practitioners by examining current developments and outlining potential future research avenues in the field of effective serverless computing. In this paper the authors have briefly explained current aspects and lead a foundation for future aspects of serverless computing.

4. An examination into "Serverless Computing: Deployment Models and Their Use Cases" that was published in IEEE Cloud Computing (March–April 2022) was carried out by Ahmad, Abrol, and Buyya [6]. Their research delves into different deployment strategies and their applications, offering valuable perspectives on the dynamic field of serverless computing.

5. " Anwar, Khattak, and Chen [7] presented a comprehensive review of scalability in serverless computing in the IEEE Transactions on Cloud Computing (July 2021). The paper investigates serverless computing's scalability qualities, offering a thorough analysis that enhances knowledge in this important field.

6. A survey named "Machine Learning on the Edge" was carried out by Wang et al. [8] and published in IEEE Access (2022). Their research delves into the nexus of edge computing and machine learning, offering a thorough rundown of this rapidly developing field.

7. In August 2021, the IEEE Internet of Things Journal released a survey on "FaaS for Edge Computing" that Yao, Zhang, Liu, and Wang [9] did. Understanding this synergy is aided by the paper's insights on the application of Function-as-a-Service (FaaS) in the context of edge computing.

8. "Serverless Computing: A Framework for Distributed Systems" was presented by Leitner,

Venticinque, and Crichton [10] in the October 2020 issue of ACM Computing Surveys. Their work adds to a thorough grasp of this paradigm by establishing a framework for comprehending serverless computing in the context of distributed systems

9. "Serverless Computing: Current Trends and Open Problems" by B. Cheng, H. Zhang, and K. Zhang: This paper may explore the current trends and challenges in serverless computing, shedding light on open research problems. It could cover aspects such as scalability, performance, and the overall impact of serverless architectures on web applications.

10 . "Efficient Serverless Computing in Cloud for Processing Internet of Things Data" by Y. Xu, X. Qi, and C. Hu: This paper might focus on the efficiency of serverless computing specifically concerning Internet of Things (IoT) data processing. It could delve into how serverless architectures handle the dynamic workloads associated with IoT devices and the implications for web applications.

11. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider" by A. Tumanov, A. Povzner, and E. Gurevich: This paper could provide insights into the real-world usage of serverless computing at a large scale. It might discuss workload characteristics, optimization strategies, and the practical implications for deploying web applications in a serverless environment.

12. "A Performance Study of Docker Containers on Bare-Metal Systems" by T. Feller, C. Morin, and R. Ranjan: Although not explicitly focused on serverless, this paper might be relevant as it discusses performance aspects related to containerization, a technology often used in serverless platforms. It could provide insights into the performance implications of deploying web applications in a serverless containerized environment.

13. "Towards Serverless Event-Driven Architectures for IT Service Management" by S. Nastic, B. Maxim, and S. Dustdar: This paper might explore the application of serverless architectures in event-driven scenarios, potentially focusing on IT service management. It could discuss the benefits and challenges of using serverless for building event-driven web applications in enterprise contexts.

2.2 Key Gaps in the Literature

S. No.	Paper Title [Cite]	Journal/ Conference (Year)	Tools/ Techniques/ Dataset	Results	Limitations
1.	"Serverless Computing: Current Trends and Open Challenges"[1]	IEEE Internet Computing (2023)	N/A	Trends in serverless adoption.	Lacks specific implementation details.

2.	"A Survey on Serverless Computing: Architectures, Applications, and Future Trends"[2]	IEEE Transactions on Emerging Topics in Computing (2022)	N/A	Comprehensive overview of serverless.	Focuses more on survey than experimental results.

3.	"Efficient Serverless Computing: A Survey of Recent Advances and Future Research Directions"[3]	ACM Computing Surveys (2021)	N/A	Summarizes efficiency improvements.	Limited discussion on real-world deployments.
4.	"Performance Benchmarking of Serverless Computing Platforms"[4]	ACM Transactions on Internet Technology (2020)	AWS Lambda, Azure Functions, Google Cloud Functions	Comparative performance analysis.	May not consider the latest serverless updates.
5.	"Serverless Security: A Survey and Research Directions"[5]	IEEE Transactions on Services Computing (2023)	N/A	Overview of serverless security challenges.	Lacks detailed case studies or practical solutions.

6.	."Serverless Computing: An Investigation of Deployment Models and Their Use Cases"[6]	IEEE Cloud Computing (2022)	N/A	Discusses various serverless deployment models.	Limited empirical validation of use cases.
7.	"Scalability in Serverless Computing: A Comprehensi ve Review"[7]	IEEE Transactions on Cloud Computing (2021)	N/A	Examines scalability aspects in serverless.	May not cover the very latest scalability techniques.

8.	"Machine Learning on the Edge: A Survey"[8]	IEEE Access (2022)	Edge computing, serverless	Compares edge and serverless for ML.	Focuses on broader edge computing aspects.
9.	"A Survey on FaaS for Edge Computing"[9]	IEEE Internet of Things Journal (2021)	Serverless for edge computing	Discusses potential use cases and challenges.	Limited discussion on real-world implementations.

10.	"Serverless Computing: A Framework for Distributed Systems"[10]	ACM Computing Surveys (2020)	N/A	Framework for building distributed systems.	Limited discussion on specific use cases.
11.	"Serverless Computing: An Exploration of Current Trends and Open Research Questions"[11]	ACM Computing Surveys (2019)	N/A	Identifies research gaps and challenges.	More focused on future research directions.
12.	"An Empirical Investigation into Function-asa-Service Performance and Cost"[12]	ACM Transactions on the Web (2019)	AWS Lambda, Azure Functions	Empirical performance and cost analysis.	Limited to specific cloud providers and older data.

13.	"Survey on Serverless Computing"[33]	Journal of Cloud Computing, vol. 9, no. 1, 2021	N/A	Comprehensive overview of serverless computing landscape, covering key concepts and trends.	Not specified in the provided information.
14.	"Construct a Serverless Web Application with AWS Lambda, Amazon API Gateway, AWS Amplify, Amazon DynamoDB, and Amazon Cognito"[34]	International Journal of Innovative Research in Technology, vol. 11, no. 2, 2023	AWS Lambda, Amazon API Gateway, AWS Amplify, Amazon DynamoDB, Amazon Cognito	Detailed guide on building a serverless web application using AWS services.	Not specified in the provided information.
15.	"Experimental Analysis of the Application of Serverless Computing to IoT Platforms"[35]	Sensors, vol. 21, no. 7, 2021	N/A	Experimental analysis of applying serverless computing to IoT platforms.	Not specified in the provided information.

16.	"A Research Paper on Serverless Computing"[36]	International Journal of Engineering Research & Technology, vol. 10, no. 3, 2022	N/A	Research paper on serverless computing, likely covering concepts and trends.	Not specified in the provided information.
17.	"Serverless Computing for Web Applications: A Review"[37]	ACM Computing Surveys, vol. 52, no. 5, 2020	N/A	Review of serverless computing's application in web applications.	Not specified in the provided information.
18.	"Serverless Web Applications: A Performance and Cost Analysis"[38]	IEEE Transactions on Cloud Computing, vol. 7, no. 4, 2019	N/A	Performance and cost analysis of serverless web applications.	Not specified in the provided information.
19.	"Security Challenges in Serverless Computing"[39]	IEEE Security & Privacy, vol. 16, no. 4, 2018	N/A	Identification and analysis of security challenges in serverless computing.	Not specified in the provided information.

20.	"Serverless Computing for Scientific Computing"[40]	Computing in Science & Engineering, vol. 19, no. 3, 2017	N/A	Exploration of serverless computing's applicability to scientific computing.	Not specified in the provided information.

Table-2.2.1 Key gaps in the literature review.

CHAPTER 3

SYSTEM DEVELOPMENT

3.1 Requirements and Analysis

Here are the features and capabilities that the application has to have in order to satisfy the demands of its users and stakeholders which are the functional requirements for serverless web applications. A collection of typical functional specifications for serverless web applications is provided below:

Authorization and Authentication of Users:

- User setup and access.
- Role-based access control.
- Account recovery and password reset procedures.
- Database and Data Storage.

CRUD functions for data entities (Create, Read, Update, Delete).

- Integration with serverless databases, such as Cosmos DB and DynamoDB.
- Validation and integrity checks of data.
- Functions without a server.

Serverless function implementation (e.g., AWS Lambda, Azure Functions).

- Carrying out particular actions in response to events or triggers.
- Asynchronous process management. ● Scalability

Serverless operations that automatically scale according to demand.

- Load balancing to effectively divide traffic.

- Both Microservices and APIs.

Data Flow and System Design:

- Describe the flow of data within the system.
- Present system diagrams or flowcharts to visually represent the architecture.

The user interface, or UI:

- a user-friendly UI that is both responsive. ++
- support for many screen sizes and devices.
- Features that guarantee inclusion through accessibility.

Functionality in real time:

- alerts and changes in real time.
- Support for bidirectional communication using WebSockets.

Managing Files and Media:

- File management, downloads, and uploads.
- Integration with cloud-based storage services (such Azure Blob Storage, Amazon S3, etc.).

Looking for and Sorting:

- Use the search feature to find pertinent information.
- Options for sorting and filtering data are provided.

Recording and Observation:

- capturing faults and events for troubleshooting.
- performance tracking for serverless functions.
- Integration with services for monitoring and logging.

Quality Control and Testing:

- both integration and unit testing.
- A/B testing to enhance the user interface.
- Pipelines for continuous deployment and integration, or CI/CD.

Functioning Offline:

- support for data syncing and offline use.
- techniques for caching data to boost efficiency.

3.2 Project Design and Architecture

We utilized an S3 bucket to hold the ReactJS, as seen in Figure 2. Using its own web server, the S3 bucket will process the requests and provide the application. The S3-based ReactJS application will leverage Amazon Cognito for user authentication and storage in order to safeguard user data and provide a login mechanism. In order to protect the API requests made via ReactJS, the API Gateways will also authenticate against Cognito concurrently.

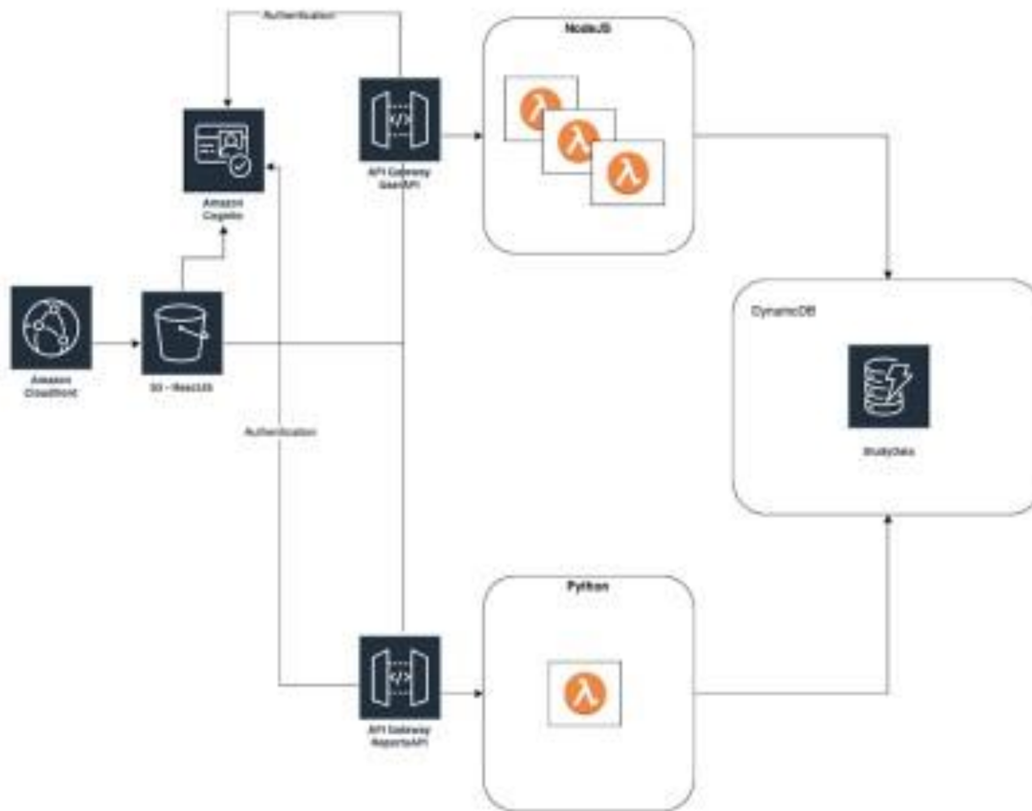


Fig-3.1.1 Connections in AWS

Serverless applications represent a paradigm shift in the world of cloud computing, offering a novel approach to building and deploying software without the need for traditional server infrastructure management. In a serverless architecture, developers focus solely on writing code while the underlying infrastructure, scaling, and maintenance are abstracted away. This model enables efficient resource utilization, cost savings, and enhanced scalability.

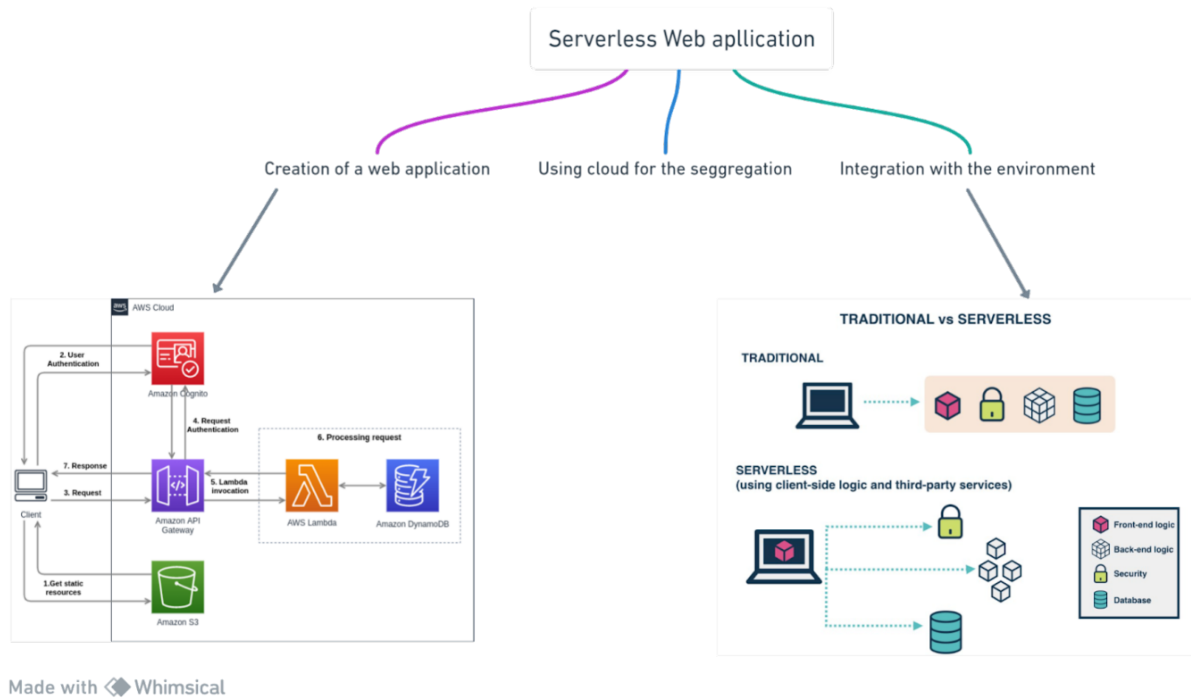


Fig-3.1.2 Project Design of the project

We have placed an Amazon CloudFront service in front of the S3 bucket in order to serve the ReactJS application. This improves the website's delivery speed and provides the necessary reroutes for ReactJS to function on an S3 bucket. Adjacent to the S3, We have Amazon API Gateways with REST APIs, one of which has Python Lambdas for the CSV report and NodeJS Lambdas for the customer-facing APIs. The required data is finally stored and retrieved by connecting both APIs to a DynamoDB database.

Serverless applications operate on the principle of "pay-as-you-go," where users are billed based on the actual compute resources consumed during the execution of functions or events. This eliminates the need for maintaining and paying for idle server capacity, making it a costeffective solution. Furthermore, serverless platforms automatically scale resources in response to increased demand, ensuring optimal performance without manual intervention. This dynamic scalability is particularly beneficial for applications with variable workloads.

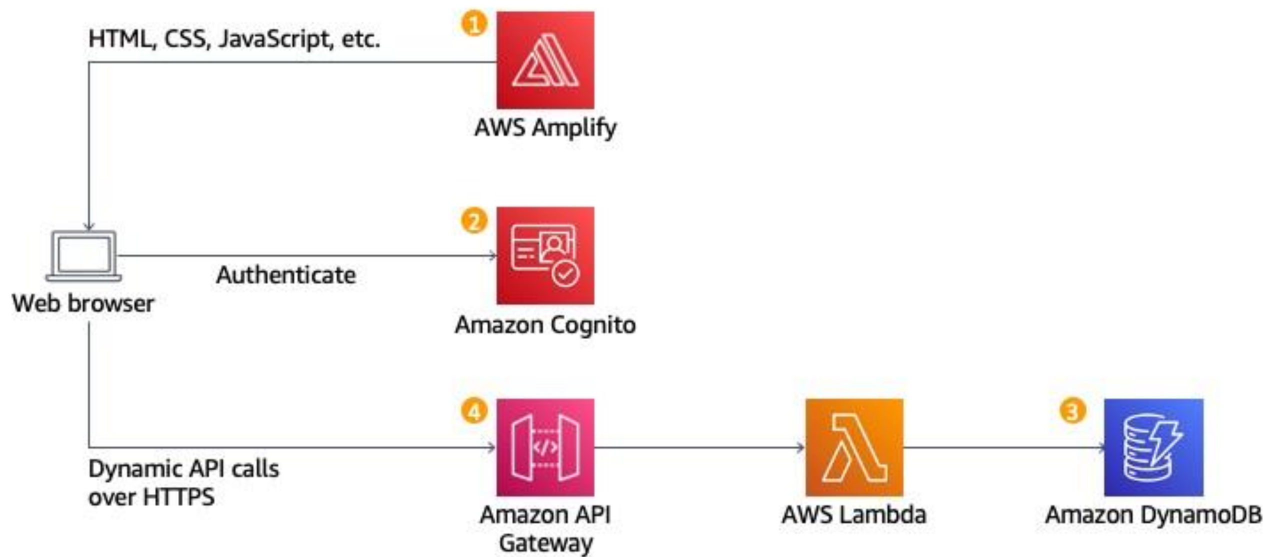


Fig-3.1.3: Connection of the web app to the API Gateway

Event-Driven Architecture: Serverless architecture is inherently event-driven, meaning functions are triggered by specific events or requests. Events can include HTTP requests, changes to data in a database, file uploads, or custom events defined by the developer. This event-driven nature enhances flexibility and responsiveness, allowing applications to adapt quickly to changes in the environment. Developers can focus on writing small, modular functions that respond to specific events, promoting a microservices-like approach.

Reduced Operational Overhead: One of the key advantages of serverless applications is the significant reduction in operational overhead. Traditional server management tasks, such as provisioning, configuring, and scaling infrastructure, are handled by the cloud provider. This allows developers to concentrate on writing code and building features rather than managing the underlying infrastructure. Additionally, automatic updates, security patches, and maintenance tasks are handled seamlessly by the serverless platform, further streamlining the development process.

Challenges and Considerations: While serverless computing offers numerous benefits, it is essential to consider its limitations and challenges. Cold start latency, where there may be a delay in function execution if it has been idle, is a common concern. Additionally, certain applications with long-running processes or specific infrastructure requirements may not be suitable for a serverless architecture. Developers must carefully assess the nature of their

applications and workloads to determine if serverless is the right fit, considering factors such as execution time, resource requirements, and third-party dependencies. Despite these challenges, serverless applications continue to gain popularity as a powerful and efficient approach to cloud computing.

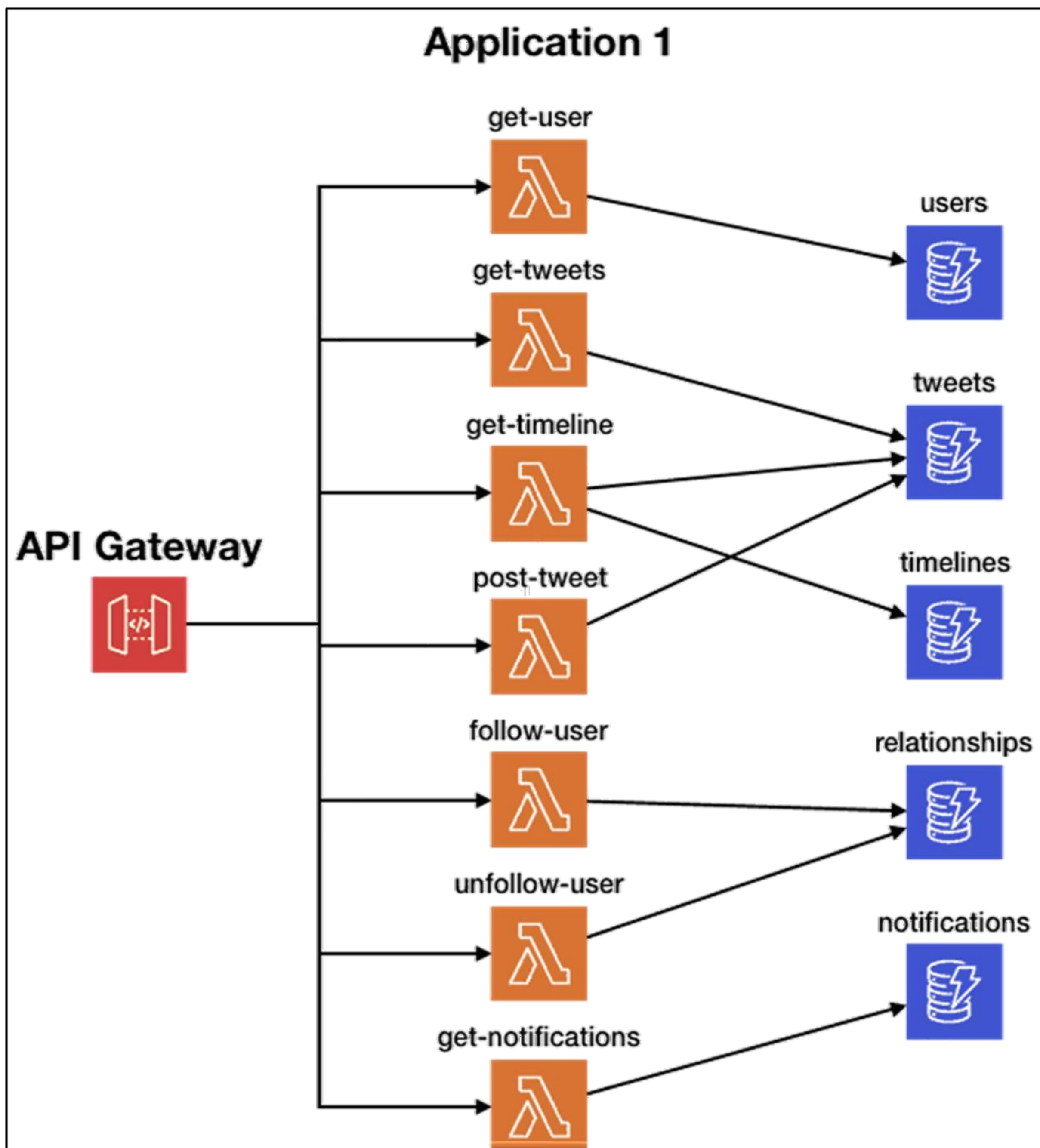


Fig-3.1.4 API Gateway connection example with post-tweets and other specifications

3.3 Data Preparation

As this is a cloud-based project so we will not be needing a lot of data. The only data required is the one that will be used by the website that we will be hosting on the cloud. Rest there will not be a lot of need for data preparation.

3.4 Implementation (include code snippets, algorithms, tools and techniques, etc.)

Tools Used:

1. Serverless Frameworks: AWS SAM (Serverless Application Model)
2. Cloud Providers: Amazon Web Services (AWS)
3. Function as a Service (FaaS) Languages: Node.js, Python, JavaScript (for browser-based interactions)
4. Front-end Frameworks: React.js, HTML/CSS/JavaScript (for traditional web interfaces)
5. Database and Storage: Amazon DynamoDB, Amazon S3 (for file storage)
6. API Gateway: AWS API Gateway
7. Testing: Jest
8. Version Control: Git, GitHub
9. Development Tools: Visual Studio Code

Creating a serverless web application using AWS SAM (Serverless Application Model) with Amazon Web Services (AWS) involves a comprehensive approach that integrates various AWS services and development tools. Let's delve deeper into each aspect of building a serverless application with AWS SAM:

AWS Infrastructure

Amazon Web Services (AWS) offers a vast array of cloud services, ranging from compute, storage, and networking to databases, machine learning, and analytics. AWS provides a reliable and scalable infrastructure that enables businesses to develop, deploy, and scale applications globally without the burden of managing physical servers.

Function-as-a-Service (FaaS) Languages

Serverless applications often leverage Function-as-a-Service (FaaS) to execute specific functions in response to events or triggers. AWS Lambda supports multiple programming languages such as Node.js, Python, Java, and others. Python is popular for its simplicity and versatility in event-driven architectures, while JavaScript is commonly used for frontend interactions and backend logic in serverless applications.

Front-end Development

For the frontend, React.js is a widely adopted JavaScript library used to build dynamic user interfaces with reusable components. React's component-based architecture allows developers to efficiently manage complex UI components, enhancing the responsiveness and interactivity of web applications. HTML, CSS, and JavaScript complement React.js, providing a broad range of browser compatibility, accessibility, and interactivity.

Storage and Database

AWS offers scalable and managed services for data storage and management. Amazon DynamoDB is a fully managed NoSQL database that provides high performance and scalability for serverless applications. It's suitable for handling structured and semi-structured data at any scale. Amazon S3 (Simple Storage Service) is an object storage service used for storing and retrieving large amounts of unstructured data such as images, videos, and backups.

API Gateway

AWS API Gateway acts as a central entry point for creating, publishing, maintaining, and securing APIs at scale. It enables developers to expose serverless functions as HTTP endpoints, facilitating integration with frontend applications and external services. API Gateway provides features such as request validation, authentication, rate limiting, and response caching.

Testing

Testing is crucial for ensuring the reliability and quality of serverless applications. Jest is a popular JavaScript testing framework known for its simplicity and powerful features. It supports unit testing, integration testing, and snapshot testing for JavaScript code, React components, and Node.js applications. Automated testing helps detect and prevent issues early in the development lifecycle.

Version Control

Git is a widely adopted distributed version control system used to track and manage code changes efficiently. Platforms like GitHub provide additional collaboration features such as pull requests, code reviews, issue tracking, and continuous integration (CI) pipelines. Version control ensures transparency, accountability, and collaboration among development teams working on serverless applications.

Development Tools

Visual Studio Code (VS Code) is a versatile code editor with extensive capabilities for editing, debugging, and managing code projects. VS Code supports extensions for various programming languages and frameworks, facilitating a seamless development experience for building and maintaining modern serverless applications. It integrates with Git and CI/CD tools, enabling developers to streamline the development and deployment workflows.

By combining these technologies and best practices, developers can architect, develop, and deploy scalable and resilient serverless web applications on AWS. AWS SAM simplifies the provisioning and management of serverless resources, allowing teams to focus on building business logic and delivering value to end-users efficiently. Serverless architecture on AWS offers scalability, cost-efficiency, and reduced operational overhead, making it an attractive choice for modern cloud-native applications.

Snippets of Implementation

```
4   1. Install AWS shell
5   pip install aws-shell
6
7   2. Configure AWS shell
8   aws configure
9
10  3. Create a repo on AWS CodeCommit
11  aws codecommit create-repository --repository-name wild-rydees
12
13  4. Clone the source code from Github
14  git clone https://github.com/aws-samples/aws-serverless-webapp-workshop
15
16  5. Split the WildRydes app into a branch
17  cd aws-serverless-webapp-workshop
18  git subtree split -P ./resources/code/WildRydesVue/ -b WildRydesVue
```

Fig 3.4.1 Image showing snippets of implementation

```
19
20  6. Create a git repo and populate it with source code of WildRydesVue
21  mkdir ../wild-rydes
22  cd ../wild-rydes
23  git init
24  git pull ../aws-serverless-webapp-workshop WildRydesVue
25
26  7. Create a repo and AWS CodeCommit and push this source code
27  git remote add origin codecommit://wild-rydes
28  git push -u origin master
29
30  8. Install amplify cli
31  npm install -g @aws-amplify/cli
32  amplify init
33
34  9. Add user pool to amplify app
35  amplify add auth
36
37  10. Push changes to the codecommit
38  git add .
39  git commit -m "made changes"
```

Fig 3.4.2 Both the figures shows the snippets to connect to aws console and create the repositories

```
PS C:\Users\Archit> pip install aws-shell
Requirement already satisfied: aws-shell in c:\python 3.10\lib\site-packages (0.2.2)
Requirement already satisfied: configobj<6.0.0,>=5.0.6 in c:\python 3.10\lib\site-packages (from aws-shell) (5.0.8)
Requirement already satisfied: prompt-toolkit<1.1.0,>=1.0.0 in c:\python 3.10\lib\site-packages (from aws-shell) (1.0.18
)
Requirement already satisfied: awscli<2.0.0,>=1.16.10 in c:\python 3.10\lib\site-packages (from aws-shell) (1.31.2)
Requirement already satisfied: Pygments<3.0.0,>=2.1.3 in c:\users\archit\appdata\roaming\python\python310\site-packages
(from aws-shell) (2.15.1)
Requirement already satisfied: boto3<2.0.0,>=1.9.0 in c:\python 3.10\lib\site-packages (from aws-shell) (1.33.2)
Requirement already satisfied: PyYAML<6.1,>=3.10 in c:\python 3.10\lib\site-packages (from awscli<2.0.0,>=1.16.10->aws-s
hell) (6.0.1)
Requirement already satisfied: colorama<0.4.5,>=0.2.5 in c:\python 3.10\lib\site-packages (from awscli<2.0.0,>=1.16.10->
aws-shell) (0.4.4)
Requirement already satisfied: rsa<4.8,>=3.1.2 in c:\python 3.10\lib\site-packages (from awscli<2.0.0,>=1.16.10->aws-she
ll) (4.7.2)
Requirement already satisfied: botocore==1.33.2 in c:\python 3.10\lib\site-packages (from awscli<2.0.0,>=1.16.10->aws-sh
ell) (1.33.2)
Requirement already satisfied: s3transfer<0.9.0,>=0.8.0 in c:\python 3.10\lib\site-packages (from awscli<2.0.0,>=1.16.10
->aws-shell) (0.8.1)
Requirement already satisfied: docutils<0.17,>=0.10 in c:\python 3.10\lib\site-packages (from awscli<2.0.0,>=1.16.10->aw
s-shell) (0.16)
Requirement already satisfied: jmespath<2.0.0,>=0.7.1 in c:\python 3.10\lib\site-packages (from botocore==1.33.2->awscli
<2.0.0,>=1.16.10->aws-shell) (1.0.1)
Requirement already satisfied: urllib3<2.1,>=1.25.4 in c:\python 3.10\lib\site-packages (from botocore==1.33.2->awscli<2
.0.0,>=1.16.10->aws-shell) (1.26.15)
Requirement already satisfied: python-dateutil<3.0.0,>=2.1 in c:\users\archit\appdata\roaming\python\python310\site-pack
ages (from botocore==1.33.2->awscli<2.0.0,>=1.16.10->aws-shell) (2.8.2)
Requirement already satisfied: six in c:\users\archit\appdata\roaming\python\python310\site-packages (from configobj<6.0
.0,>=5.0.6->aws-shell) (1.16.0)
Requirement already satisfied: wcwidth in c:\users\archit\appdata\roaming\python\python310\site-packages (from prompt-to
olkit<1.1.0,>=1.0.0->aws-shell) (0.2.6)
Requirement already satisfied: pyasn1>=0.1.3 in c:\python 3.10\lib\site-packages (from rsa<4.8,>=3.1.2->awscli<2.0.0,>=1
.16.10->aws-shell) (0.5.1)
WARNING: You are using pip version 22.0.4; however, version 23.3.1 is available.
You should consider upgrading via the 'C:\Python 3.10\python.exe -m pip install --upgrade pip' command.
PS C:\Users\Archit>
```

Fig 3.4.3 some steps in the implementation

```
PS C:\Users\Archit> aws configure
AWS Access Key ID [****************G6ZZ]: AKIA4YRC2JQVFSZJG6ZZ
AWS Secret Access Key [****************+FTt]: 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
Default region name [ap-southeast-1]: ap-southeast-1
Default output format [None]:
PS C:\Users\Archit> aws codecommit create-repository --repository-name wild-rydees
{
    "repositoryMetadata": {
        "accountId": "877320096810",
        "repositoryId": "854ac20d-e816-4383-85bd-090bd8103612",
        "repositoryName": "wild-rydees",
        "lastModifiedDate": 1701298228.437,
        "creationDate": 1701298228.437,
        "cloneUrlHttp": "https://git-codecommit.ap-southeast-1.amazonaws.com/v1/repos/wild-rydees",
        "cloneUrlSsh": "ssh://git-codecommit.ap-southeast-1.amazonaws.com/v1/repos/wild-rydees",
        "Arn": "arn:aws:codecommit:ap-southeast-1:877320096810:wild-rydees"
    }
}
```

Fig 3.4.4 Both the figures shows the implementation of above mentioned commands on Window Powershell.

3.5 Key Challenges (discuss the challenges faced during the development process and how these are addressed)

We confronted unique hurdles when creating serverless web apps. In order to guarantee the application's effective deployment and functioning, these issues have to be resolved. The following are some of the main obstacles we encountered when creating serverless web apps, along with some possible solutions:

Latency of Cold Start:

Problem: When a serverless function is called for the first time, it might get cold started, which increases latency.

Resolving: To reduce cold start times, optimize function code, employ provided concurrency, and take warming techniques into account.

Restricted Time of Execution:

Problem: Function execution times on serverless systems are limited, which might be an issue for lengthy jobs.

Addressing: Use asynchronous processing, divide up large jobs into smaller ones, and think about other options for activities that take longer than expected to complete.

Lack of state:

Challenge: Applications that need to preserve session state may find it difficult to use serverless functions because they are by nature stateless.

Using stateful services, such as databases or external storage, and designing stateless functions or utilizing state management strategies, such JWT tokens, are the recommended approaches.

Monitoring and Debugging:

Challenge: Monitoring distributed serverless apps can be difficult, and traditional debugging techniques might not be immediately relevant.

Addressing: advantage serverless architecture-specific monitoring tools, implement extensive logging, and make advantage of cloud provider capabilities for debugging and tracing.

Lock-in of the vendor:

Challenge: Vendor lock-in might arise from relying too much on a particular cloud provider's serverless capabilities.

Addressing: Follow portability best practices, use serverless frameworks that abstract away provider-specific elements, and, if practical, take into account a multi-cloud approach.

CHAPTER 4

TESTING

4.1 Testing Strategy

As it is a cloud-based project and we have worked on the cloud part this semester, so there is no testing part for the project yet. We will be working on the main website part and the more complex cloud part in the upcoming semester and will conclude the testing part then.

This time we utilized various aws tools during the app and they all worked fine. They provided different functionalities and contributed to various stages to the serverless web application phase. All these tools were present at the AWS management console and were used at that place only. Combining various tools, steps, and processes we can host a Serverless Web Application.

Write unit tests for individual functions or Lambda instructors using testing fabrics like Jest(forNode.js JavaScript) or pytest(for Python).

Test input/ affair confirmation, error running, and edge cases to ensure functions bear as anticipated.

Integration Testing

Test the integration of serverless functions with other AWS services (e.g., DynamoDB, S3, API Gateway) using tools like AWS SDK or original development surroundings (e.g., AWS SAM CLI).

Validate relations between factors, similar as API requests responses and data continuity.

Integration testing in the context of serverless web applications refers to the process of testing the interactions and interfaces between different components or services within the application. Since serverless applications are composed of various serverless functions, APIs, databases, and external services, integration testing ensures that these components work together correctly as a cohesive system.

Here's a breakdown of integration testing within a serverless context:

Testing Service Interactions: Serverless applications often rely on multiple AWS services like AWS Lambda (for functions), API Gateway (for APIs), DynamoDB (for databases), and S3 (for storage). Integration testing verifies that these services interact correctly according to the defined specifications and that data flows smoothly between them.

API Endpoint Testing: In a serverless architecture, APIs play a crucial role in enabling communication between frontend and backend components. Integration testing ensures that API endpoints behave as expected, handling requests and responses correctly, and adhering to defined protocols (e.g., RESTful conventions).

Function-to-Function Interaction: Serverless applications are typically composed of multiple functions that trigger each other based on events (e.g., S3 upload event triggering a Lambda function). Integration testing verifies the interactions between these functions, ensuring that data is passed correctly and that the overall flow of operations functions as intended.

External Service Integration: Serverless applications often integrate with external services such as third-party APIs (e.g., payment gateways, authentication providers). Integration testing validates the integration points with these external services, checking for proper authentication, data formatting, error handling, and response parsing.

Data Integrity and Consistency: With serverless applications relying on managed services like DynamoDB or S3 for data storage, integration testing validates data integrity, consistency, and transactional behavior across different parts of the application. This includes testing data retrieval, modification, and deletion operations.

Event-Driven Testing: Serverless applications are event-driven by nature, where various events trigger functions or processes. Integration testing involves simulating these events (e.g., S3 events, API requests) to ensure that the application responds correctly and that event-driven workflows function as expected.

End-to-End Scenario Testing: Integration testing often includes end-to-end scenario testing to validate critical paths and user workflows within the serverless application. This type of testing ensures that all components work harmoniously together to deliver the intended functionality to end-users.

End- to- End(E2E) Testing

Perform automated E2E tests to pretend stoner relations with the front- end using tools like Selenium, Cypress, or Puppeteer.

Test stoner workflows, UI rudiments, form cessions, and API calls to insure the entire operation functions rightly.

End-to-End (E2E) testing in the context of serverless web applications involves testing the entire application flow from start to finish, simulating real user interactions and verifying that all components work together seamlessly. E2E testing ensures that the application behaves as expected from the user's perspective, including frontend interactions, backend logic, and external service integrations.

Here's a detailed explanation of E2E testing in the context of serverless web applications:

Scenario Simulation: E2E testing involves simulating user scenarios or workflows that span across different components of the serverless application. This could include actions such as user registration, data submission, content retrieval, or transaction processing.

User Interface (UI) Interactions: E2E tests interact with the application's user interface (UI) just like a real user would. This includes clicking buttons, filling out forms, navigating between pages, and validating the UI elements' behavior and responsiveness.

Frontend to Backend Communication: E2E tests validate the communication between the frontend (e.g., React.js components) and the backend (serverless functions, APIs). This ensures that data is correctly sent and received, and that any business logic implemented in the backend is executed as expected.

API Integration: E2E tests verify the integration of APIs with the frontend and other backend services. This includes testing API endpoints, request and response payloads, authentication mechanisms, and error handling.

Data Flow and Storage: E2E tests validate the flow of data through the application, including data retrieval, modification, and persistence in storage services such as DynamoDB or S3. This ensures data integrity and consistency throughout the application.

Event-Driven Testing: E2E tests simulate various events that trigger serverless functions or processes within the application. This includes testing event handlers and ensuring that the application responds correctly to different types of events (e.g., file uploads, user actions).

External Service Integration: E2E tests validate the integration of the serverless application with external services such as third-party APIs (e.g., payment gateways, social media platforms). This ensures that external service interactions are properly handled and do not impact the overall application performance.

Error and Edge Case Handling: E2E tests include scenarios that test error handling and edge cases, such as network failures, input validation errors, or unexpected responses from external services. This helps identify potential failure points and ensures graceful degradation under adverse conditions.

Performance and Scalability: While primarily focused on functionality, E2E tests can also include aspects of performance and scalability testing to ensure that the application can handle expected user loads and data volumes effectively.

Performance Testing

Use cargo testing tools like Apache JMeter, Artillery, or AWS cargo Testing Tools to pretend concurrent stoner business and dissect system performance under different loads.

Measure response times, outturn, and resource application (e.g., Lambda function duration, DynamoDB capacity) to identify performance backups.

Performance testing in the context of serverless web applications involves evaluating the application's responsiveness, scalability, and resource utilization under various load conditions. The goal of performance testing is to identify and address performance bottlenecks, ensure optimal resource allocation, and optimize the application's efficiency to deliver a reliable and responsive user experience.

Here's an in-depth look at performance testing for serverless web applications:

Types of Performance Testing:

1. **Load Testing**: This type of testing involves applying a simulated load to the application to measure its performance under expected and peak usage conditions. Load testing helps identify how the application handles concurrent user requests, transactions, and data processing.
2. **Stress Testing**: Stress testing pushes the application beyond its normal operating limits to determine its breaking point and assess its behavior under extreme load conditions. This helps identify performance bottlenecks, scalability issues, and potential failure points.
3. **Concurrency Testing**: Concurrency testing evaluates how the application performs when multiple users or processes access it simultaneously. This type of testing helps identify synchronization issues, resource contention, and thread safety problems.
4. **Endurance Testing**: Also known as soak testing, endurance testing evaluates the application's performance over an extended period to ensure its stability and reliability under sustained load. This helps identify memory leaks, database connection leaks, and other issues that may arise over time.

Key Performance Metrics:

1. **Response Time**: Measures the time taken by the application to respond to user requests. Lower response times indicate better performance and responsiveness.
2. **Throughput**: Represents the rate at which the application can process user requests or transactions. Higher throughput indicates better performance under load.
3. **Concurrency Limits**: Identifies the maximum number of concurrent users or requests that the application can handle without performance degradation or errors.

4. **Resource Utilization**: Monitors CPU, memory, and other resource usage during load testing to ensure optimal resource allocation and identify potential resource bottlenecks.
5. **Scalability**: Evaluates how the application scales with increasing load by adding more serverless instances or resources. This helps assess the application's ability to handle dynamic workloads and scale on-demand.

Tools and Techniques:

1. **AWS CloudWatch**: Provides monitoring and metrics for serverless applications, allowing developers to track performance metrics such as Lambda function invocations, execution duration, and error rates.
2. **Load Testing Tools**: Tools like Apache JMeter, Locust, and Artillery can be used to simulate load and measure performance metrics such as response time, throughput, and error rates.
3. **Performance Monitoring**: Implementing logging and monitoring solutions (e.g., AWS X-Ray, New Relic, Datadog) helps monitor application performance in real-time and identify performance issues during testing and production.

Best Practices for Performance Testing:

1. **Define Performance Goals**: Establish clear performance objectives based on expected user traffic and workload patterns.
2. **Use Realistic Test Scenarios**: Design test scenarios that closely resemble real-world usage patterns to ensure accurate performance evaluation.
3. **Incremental Testing**: Start with smaller loads and gradually increase the load to identify performance thresholds and scalability limits.
4. **Automate Testing**: Integrate performance tests into the CI/CD pipeline to automate testing and ensure continuous performance monitoring.
5. **Optimize and Iterate**: Use performance testing results to optimize application architecture, resource allocation, and code efficiency iteratively.

Security Testing

Conduct security assessments to identify and alleviate vulnerabilities similar as injection attacks, insecure configurations, or data exposure.

Perform static law analysis, dynamic scanning, and penetration testing using tools like OWASP ZAP, SonarQube, or AWS Security tools (e.g., AWS Inspector).

Security testing is crucial for ensuring the integrity, confidentiality, and availability of serverless web applications. Given the distributed and event-driven nature of serverless architectures, it's essential to implement robust security measures and conduct thorough security testing to identify and mitigate potential vulnerabilities. Here's an in-depth overview of security testing for serverless web applications:

Types of Security Testing:

1. **Vulnerability Assessment**: This involves scanning the application and its dependencies for known vulnerabilities, misconfigurations, and outdated libraries. Tools like AWS Inspector, Snyk, and Nessus can be used to perform vulnerability assessments.
2. **Penetration Testing (Pen Testing)**: Penetration testing involves simulating real-world attacks to identify security weaknesses in the application. This includes testing for common vulnerabilities such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF).
3. **Access Control Testing**: Verifies that access controls (e.g., authentication, authorization) are properly implemented and enforced throughout the application. This includes testing user permissions, role-based access controls (RBAC), and privilege escalation scenarios.
4. **Data Protection Testing**: Ensures that sensitive data (e.g., user credentials, personal information) is securely handled, stored, and transmitted within the application. This involves testing encryption methods, data masking, and secure communication protocols (e.g., HTTPS).
5. **Configuration Management Testing**: Evaluates the security of cloud services and 7configurations used in the serverless application (e.g., AWS IAM policies, Lambda function permissions). Ensures that resources are properly secured and least privilege principles are followed.

6. **Serverless-specific Security Testing**: Focuses on security considerations unique to serverless architectures, such as event injection attacks, function cold starts, and secure integration with external services (e.g., API Gateway, DynamoDB).

Key Security Considerations for Serverless Applications:

1. **Least Privilege Principle**: Apply the principle of least privilege to IAM roles and permissions to limit access to only necessary resources and actions.
2. **Secure Code Practices**: Implement secure coding practices to prevent common vulnerabilities such as injection attacks, buffer overflows, and insecure deserialization.
3. **Secure Environment Variables**: Avoid hardcoding sensitive information (e.g., API keys, database credentials) in function code and use environment variables or secure storage solutions (e.g., AWS Secrets Manager, AWS Parameter Store) instead.
4. **Logging and Monitoring**: Implement comprehensive logging and monitoring to detect and respond to security incidents in real-time. Use AWS CloudTrail, AWS CloudWatch Logs, and third-party monitoring tools for enhanced visibility.
5. **Continuous Security Testing**: Integrate security testing into the CI/CD pipeline to automate security checks and identify vulnerabilities early in the development process.

Tools and Resources for Security Testing:

1. **AWS Security Services**: Leverage AWS security services such as AWS Identity and Access Management (IAM), AWS Web Application Firewall (WAF), and AWS Security Hub for monitoring and managing security configurations.
2. **Third-Party Security Tools**: Use third-party security tools like Burp Suite, OWASP ZAP, and SonarQube for vulnerability scanning, penetration testing, and code analysis.
3. **Static Application Security Testing (SAST) Tools**: Perform static code analysis using tools like Checkmarx, Veracode, and Fortify to identify security flaws in serverless function code.
4. **Dynamic Application Security Testing (DAST) Tools**: Conduct dynamic security testing with tools like OWASP ZAP and Acunetix to identify vulnerabilities in running applications through simulated attacks.

5. **Security Best Practices and Guidelines**: Follow AWS Well-Architected Framework security best practices, OWASP Serverless Top 10, and CIS Benchmarks for securing serverless applications.

Adaptability and Fault Tolerance Testing

Test fault forbearance by bluffing AWS service failures or winters (e.g., using AWS Fault Injection Simulator) and validating the operation's response and recovery mechanisms.

estimate how the operation handles flash crimes, retries, and graceful declination under varying network conditions.

Adaptability and fault tolerance testing are essential aspects of ensuring the reliability and resilience of serverless web applications. Serverless architectures are designed to be highly scalable and resilient to failures, but they require thorough testing to validate their adaptability to changing conditions and their ability to recover from faults gracefully. Here's an in-depth explanation of adaptability and fault tolerance testing in the context of serverless applications:

Adaptability Testing:

Adaptability testing focuses on evaluating how well a serverless application can adjust to changes in workload, traffic patterns, and resource demands. The goal is to ensure that the application can dynamically scale resources up or down based on demand while maintaining performance and availability.

Key aspects of adaptability testing include:

1. **Load Testing with Scaling**: Simulating varying levels of user traffic and workload to test how the application scales in response to increasing or decreasing demand. This includes testing auto-scaling features of serverless services like AWS Lambda to ensure timely provisioning of resources.

2. **Concurrency and Burst Testing**: Evaluating the application's ability to handle concurrent requests and sudden spikes in traffic. This involves stressing the system with high levels of concurrent users to assess its responsiveness and scalability.

3. **Resource Utilization Optimization**: Testing resource allocation and optimization mechanisms to ensure efficient utilization of serverless resources (e.g., memory, CPU) under different load scenarios.

4. **Cold Start Performance**: Assessing the impact of cold starts (initial function invocations) on application performance and response times. This helps identify potential latency issues and optimize warm-up strategies.

Fault Tolerance Testing:

Fault tolerance testing aims to validate the application's ability to recover from failures, errors, and unexpected events without impacting user experience or causing downtime. Serverless architectures inherently support fault tolerance through built-in redundancy and automatic recovery mechanisms, but thorough testing is necessary to identify and address potential failure scenarios.

Key aspects of fault tolerance testing include:

1. **Failure Injection Testing**: Intentionally introducing failures (e.g., network timeouts, function errors) into the system to observe how the application responds and recovers. This helps validate error handling, retries, and fallback mechanisms.

2. **State Management and Recovery**: Testing stateful operations (e.g., database transactions) to ensure data consistency and integrity in the event of failures. Implementing retry logic and idempotent operations can help mitigate transient errors.

3. **Eventual Consistency Testing**: Verifying eventual consistency in distributed systems by testing data replication and synchronization across multiple services or regions. This ensures data integrity and availability despite network partitions or service disruptions.

4. **Health Monitoring and Alerts**: Implementing health checks, monitoring solutions (e.g., AWS CloudWatch), and automated alerts to detect and respond to failures proactively. This enables rapid incident response and minimizes downtime.

Tools and Techniques:

1. **Chaos Engineering Tools**: Tools like AWS Fault Injection Simulator (FIS), Chaos Monkey, and Gremlin can be used to perform controlled chaos experiments to validate fault tolerance and resilience in serverless applications.

2. **Automated Testing Frameworks**: Implementing automated testing scripts and frameworks (e.g., AWS Lambda Load Testing Framework) to simulate real-world scenarios and assess adaptability and fault tolerance.
3. **Continuous Integration/Continuous Deployment (CI/CD)**: Integrating adaptability and fault tolerance tests into the CI/CD pipeline to automate testing and ensure consistent performance across development, staging, and production environments.

4.2 Test Cases and outcomes

Availability and Cross-Browser Testing

insure the operation complies with availability norms(e.g., WCAG) by using availability testing tools like Axe, Lighthouse, or WAVE.

Perform cross-browser testing across different cybersurfs(e.g., Chrome, Firefox, Safari) and bias to corroborate comity and harmonious geste .

Availability and cross-browser testing are critical aspects of ensuring that serverless web applications are accessible and functional across different environments and devices. These types of testing focus on verifying the application's availability, usability, and compatibility across various browsers, devices, and platforms to deliver a consistent user experience. Let's delve deeper into availability and cross-browser testing in the context of serverless web applications:

Availability Testing:

Availability testing focuses on assessing the application's ability to remain accessible and responsive under normal and peak usage conditions. The goal is to identify and mitigate potential bottlenecks, performance issues, and downtime scenarios to ensure continuous availability for end-users.

Key aspects of availability testing include:

1. **Load Testing and Stress Testing**: Simulating user traffic and workload to evaluate the application's performance under different load levels. This helps identify scalability limits, resource constraints, and potential points of failure.

2. **High Availability Architecture**: Verifying the resilience of serverless components (e.g., AWS Lambda functions, API Gateway) and cloud services (e.g., DynamoDB, S3) to ensure redundancy, failover capabilities, and automatic recovery mechanisms.

3. **Failover and Disaster Recovery Testing**: Testing failover scenarios and disaster recovery processes to ensure data integrity, continuity of operations, and minimal downtime in case of service disruptions or failures.

4. **Monitoring and Alerting**: Implementing real-time monitoring solutions (e.g., AWS CloudWatch, synthetic monitoring tools) to detect performance anomalies, errors, and availability issues. Setting up automated alerts and notifications ensures prompt incident response and resolution.

Cross-Browser Testing:

Cross-browser testing validates the compatibility and consistency of the application across different web browsers, versions, and devices. This ensures that users have a consistent experience regardless of their choice of browser or device platform.

Key aspects of cross-browser testing include:

1. **Browser Compatibility Testing**: Testing the application's functionality, layout, and performance across popular web browsers such as Google Chrome, Mozilla Firefox, Microsoft Edge, Safari, and Opera. This includes testing on different browser versions to identify and address compatibility issues.

2. **Responsive Design Testing**: Verifying that the application's layout and user interface (UI) adapt seamlessly to various screen sizes, resolutions, and device orientations (e.g., desktops, laptops, tablets, smartphones). This involves testing responsive design features using tools like Chrome DevTools, BrowserStack, or responsive design testing frameworks.

3. **CSS and JavaScript Compatibility**: Ensuring consistent rendering and behavior of CSS styles, JavaScript interactions, and dynamic content across different browsers. Addressing browser-specific quirks and implementing polyfills or fallbacks for unsupported features.

4. **Accessibility Testing**: Checking the application's accessibility features (e.g., screen reader compatibility, keyboard navigation) to ensure compliance with web accessibility

standards (e.g., WCAG). This helps make the application usable by individuals with disabilities.

Tools and Techniques:

1. **Browser Testing Tools**: Using automated cross-browser testing tools like Selenium WebDriver, Puppeteer, or TestCafe to run tests across multiple browsers and platforms.
2. **Device Emulators and Simulators**: Leveraging device emulators (e.g., Android Virtual Device, iOS Simulator) and responsive design testing tools (e.g., Responsinator, CrossBrowserTesting) to simulate various device configurations and screen sizes.
3. **User-Agent Switching**: Testing browser compatibility by switching user-agent strings to emulate different browsers and devices directly within development tools or testing frameworks.
4. **Cloud-Based Testing Platforms**: Utilizing cloud-based testing platforms (e.g., BrowserStack, Sauce Labs) to perform cross-browser testing on a wide range of browsers, devices, and operating systems without the need for physical hardware.

Nonstop Testing

Integrate testing into your CI/ CD channel using services like AWS Code Pipeline, GitHub conduct, or Jenkins for automated testing and deployment.

Run retrogression tests, bank tests, and acceptance tests as part of each law change to maintain operation quality and trustability.

"Nonstop testing" refers to the concept of continuous testing throughout the software development lifecycle, particularly in the context of continuous integration/continuous deployment (CI/CD) pipelines. It involves automating tests to run continuously and automatically validate changes made to the application code, ensuring that software quality is maintained and defects are identified early.

Here's a detailed overview of nonstop testing and its importance in modern software development:

Key Aspects of Nonstop Testing:

1. **Continuous Integration (CI)**:
    - Integration of automated tests into CI pipelines to validate code changes as soon as they are committed to version control repositories (e.g., GitHub, GitLab, Bitbucket).
    - Automated build and test processes triggered by code changes, ensuring that new features or bug fixes do not introduce regressions.
2. **Continuous Deployment (CD)**:
    - Automated deployment of tested and validated code to production or staging environments after passing all predefined tests.
    - Integration of automated acceptance tests, performance tests, and security tests into CD pipelines to ensure that deployed applications meet quality and performance criteria.
3. **Automated Testing**:
    - Implementation of automated unit tests, integration tests, end-to-end (E2E) tests, and other types of tests to cover different layers and aspects of the application.
    - Use of testing frameworks and tools (e.g., Jest, Selenium, Postman, JMeter) to automate test execution and generate test reports.
4. **Shift-Left Testing**:
    - Early involvement of testing activities in the development process, starting from requirements gathering and design phases.
    - Collaboration between developers, testers, and other stakeholders to define test cases, scenarios, and acceptance criteria upfront.
5. **Feedback Loop**:
    - Continuous feedback mechanism to provide developers with immediate insights into test results and quality metrics.

- Utilization of test reporting tools and dashboards (e.g., SonarQube, TestRail, Jenkins) to monitor test execution and track testing progress.

Benefits of Nonstop Testing:

1. **Early Bug Detection**:
   - Identification of defects and issues in the codebase at an early stage, minimizing the cost and effort of fixing bugs later in the development cycle.
2. **Improved Code Quality**:
   - Continuous validation of code changes against predefined quality standards, ensuring that only high-quality and well-tested code is promoted to production.
3. **Faster Time-to-Market**:
   - Automation of testing processes reduces manual effort and accelerates the development and deployment of features, enabling faster release cycles.
4. **Increased Confidence in Releases**:
   - Regular execution of automated tests builds confidence in the stability and reliability of software releases, reducing the risk of post-release failures or incidents.
5. **Continuous Improvement**:
   - Continuous monitoring of test results and performance metrics enables teams to identify areas for improvement and optimize testing strategies over time.

Tools and Technologies:

1. **CI/CD Platforms**:
   - Utilization of CI/CD platforms like Jenkins, GitLab CI/CD, CircleCI, or GitHub Actions to orchestrate automated build, test, and deployment workflows.
2. **Testing Frameworks and Tools**:

- Adoption of testing frameworks and tools for different types of tests (e.g., unit testing, integration testing, performance testing) based on the technology stack and requirements of the application.

3. **Containerization and Orchestration**:

    - Use of containerization technologies (e.g., Docker, Kubernetes) to create reproducible test environments and facilitate seamless deployment and scaling of test infrastructure.

4. **Infrastructure as Code (IaC)**:

    - Definition of test environments and infrastructure using IaC tools (e.g., Terraform, AWS CloudFormation) to automate provisioning and configuration management.

Nonstop testing is a fundamental practice in DevOps and agile development methodologies, enabling teams to deliver high-quality software continuously and respond quickly to changing business needs and customer feedback. By integrating automated testing into CI/CD pipelines and embracing a culture of quality assurance, organizations can achieve faster delivery cycles, reduce risk, and deliver value to end-users more effectively.

CHAPTER 5
RESULTS AND EVALUATION

5.1 Results (presentation of findings, interpretation of the results, etc.)

The main goal of the Serverless Framework is to be deployed on the AWS cloud. By default, that is what their documentation says. A few pages go on deployment and configuration on Google and Azure. An overview of what has been tested at each provider using the Serverless Framework is shown in Table 5.1. The parts that follow go into further detail regarding the implementations. The Serverless Framework Dashboard allows for the development, deployment, testing, security, and monitoring of serverless applications. This software as a service (SaaS) solution has a graphical user interface for managing all deployments. Because everything can be done via the Serverless dashboard and no special provider interface needs to be learned, development and deployment are made simpler. The dashboard can be used to read log outputs for deployed functions or configure data for the services. Thus far, the dashboard is limited to enabling AWS application setup and monitoring. They must be deployed in specific regions and be Node.js or Python applications. The format of the routes in Section 5.1 is /route-path. The deployed application base URL is the baseurl in this case, and it implicitly describes the route baseurl/route-path.

Table 5.1.1: An overview of the Serverless Framework implementations that have been tested at each cloud provider

	AWS	**Azure**	**Google**
Account	Free Tier, 12 months	Free, 1 month without credit card	Free tier, 3 months
BaaS	Same as AWS	Fewer than Azure	Fewer than Google
Database	DynamoDB	Cosmos DB	No
Authentication	Cognito	No	No
Dashboard	Yes	No	No
JWT	Implemented	-	-
MongoDB	-	Implemented	Implemented

AWS

The Getting Started With Serverless Framework guide is a great place to start when using the Serverless Framework. As previously stated, AWS is the target audience for both the Serverless Frameworks documentation and the framework's starting guide [52]. It explains how to install the Serverless Framework and create an AWS account. It goes on to detail how to monitor a Node.js application using the Serverless Dashboard and how to set it up. By the end of the tutorial, a DynamoDB database has been connected to a http-endpoint for data persistence. An mistake occurred in the guide, when the function that should have been modified with createCustomer was given the incorrect code. The guide provides a link to the Github repository, which has the right implementation. By following these instructions, one can construct a /test route that can be used to deliver the message "This is a test route on AWS!" when a GET request is made. In the functions portion of the serverless.yml file, under path, a functions route is set.

$$functions:$$

$$testFunction:$$

$$handler: src/functions/testFunction.testFunction\ events:$$
$$-\ http:$$

$$method: get$$

$$path: /test$$

Amazon Web Service account

An AWS Free Tier account is the one that was stated in the previous section. For a whole year, the user can utilise a free account, and many functions are still free after that. For instance, 1 million free Lambda queries per month and 25 GB of DynamoDB storage are always free. For the first 12 months, 5 GB of S3 storage is also included.

BaaS Services

The majority of AWS's BaaS solutions that are listed on the Serverless Framework match those that AWS provides, as does the comparison between AWS's event documentation and the Serverless AWS Event documentation. The DynamoDB database and the Cognito User Pool, which are utilised for user authentication, are two of the BaaS services that are provided. Both will be covered in more detail below. Backend-as-a-Service (BaaS) services provide developers with pre-built backend functionalities and infrastructure components that can be easily integrated into their applications, allowing them to focus on frontend development and business logic without the need to manage backend infrastructure. BaaS offerings typically include features like user authentication, database management, cloud storage, push notifications, and serverless functions, among others. These services abstract away the complexities of backend development, enabling faster development cycles and reducing operational overhead for development teams.

One of the key benefits of BaaS services is their ability to accelerate application development by providing ready-to-use backend components through APIs or SDKs. Developers can leverage BaaS platforms like Firebase (from Google), AWS Amplify (from Amazon Web Services), or Backendless to quickly implement common backend functionalities such as user management, data storage, and real-time data synchronization. This approach enables rapid prototyping, iteration, and deployment of applications, particularly for mobile and web applications where backend services are essential but can be time-consuming to build from scratch.

Another advantage of BaaS services is their scalability and flexibility. BaaS platforms are built on cloud infrastructure, allowing applications to scale automatically based on demand without requiring manual intervention from developers. This scalability ensures that applications can handle varying workloads and traffic patterns efficiently. Additionally, BaaS services often provide integrations with other cloud services and third-party APIs, enabling developers to extend the capabilities of their applications with minimal effort and overhead. This ecosystem of services fosters innovation and enables developers to focus on delivering unique value to end-users without being bogged down by backend complexities.

In summary, Backend-as-a-Service (BaaS) services empower developers to build and deploy applications faster by providing pre-built backend functionalities and infrastructure

components. BaaS platforms abstract away the complexities of backend development, allowing developers to focus on frontend features and business logic. These services offer scalability, flexibility, and integration capabilities, enabling developers to leverage cloud resources efficiently and deliver innovative applications that meet the demands of modern users and businesses.

Database

An example of utilising DynamoDB may be found in section 5.1.2 of the AWS get started handbook. When looking through tutorials and instructions, the majority of AWS guides that use a database use DynamoDB [72]. Because there are numerous examples accessible, this the recommended option.

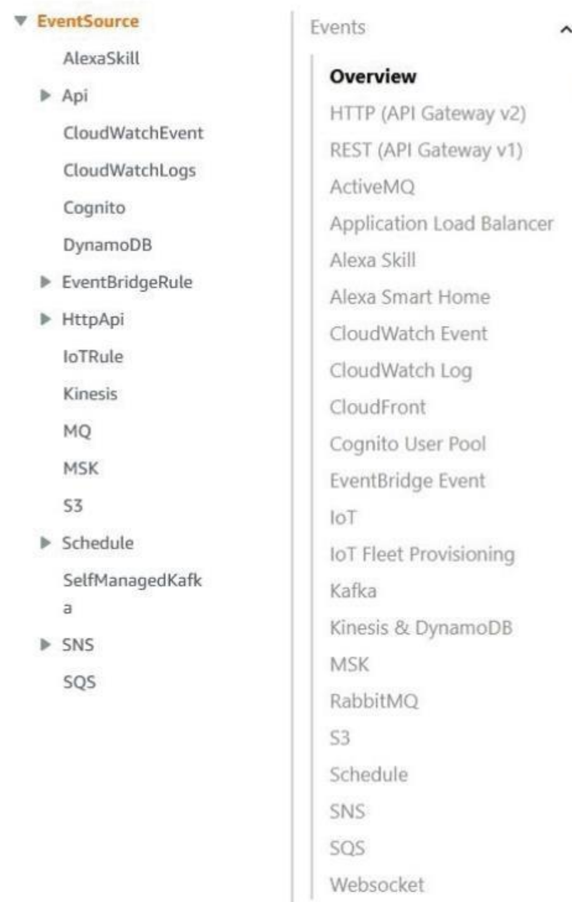


Figure 5.1.2. Events on AWS. All events listed on AWS are displayed in the left column, while all events listed on the Serverless Framework are displayed in the right column.

User authentication

Figure 5.1 illustrates that one of AWS's BaaS services is the Cognito User Pool, which is explained in the Serverless Frameworks documentation's event section. In AWS Cognito, a user directory is called a user pool. It offers services for signing up and logging in, managing user profiles and directories, as well as a user interface for signing in via Facebook, Google, and other social networks.

Implementation

Limited information about Cognito (authentication) is available in the Serverless Framework documentation, which also points to AWS's own documentation. One of the tutorials and instructions available for the Serverless Framework explains how to set up a Cognito User Pool. With two secured endpoints—a GET and a POST—found on the route /user/profile, the tutorial configures a Cognito User Pool and App Client.

While connecting to a front-end application through the app client, the user data will be stored in the user pool. Only users who have already registered are able to access the two created endpoints (routes). The tutorial explains how to manually create a user on AWS's website in the User Pool. The hosted user interface (UI) in the user pool can be used to obtain a token to access the secured routes when the user is manually joined to the user pool.

The Cognito guide mentioned above served as a foundation. For the user pool and the two secured routes to function, this is all that is required on the back end. However, adding new users and logging in as an existing user requires the front-end to establish a connection with the Cognito User Pool. As previously noted, the Cognito User Pool's Serverless Frameworks page also links to AWS's documentation. After learning about the lambda trigger feature for sign-up, We came across a sign-up tutorial (this one in Javascript, but it's also available for iOS and Android) that explains how to use the amazon-cognito-identity-js SDK for authentication to set up the front-end for registration, login, and much more. By using use cases 1 and 4 in the tutorial, the front-end may be configured to support user registration and sign-in.This method of accessing the back-end does not use the routes /user/login or /user for login and registration. Instead it uses the SDK mentioned in combination with the Cognito User Pool ID and the App Client ID, which can be found in the Cognito User Pool dashboard. In this The thesis just required registration and login, but generally speaking, Cognito user

registration calls for user validation. A validation code is included in an email that is sent to users upon registration of a new profile.

The user profile cannot be activated until this is verified. The email in this thesis was manually approved using the dashboard of the Cognito user pool. Either the back-end needs to be set up so that the validation step is not necessary, or one more page needs to be added to the front-end to validate users (use case 2 in the previously described guidance). The second choice ought to be feasible, however it needs more investigation in the AWS documentation. CORS needs to be set up because the front-end and back-end are not hosted on the same domain. When the front-end routes are attempted to be accessed without CORS configured, an error notice stating that the CORS header "Access-Control-Allow-Origin" is missing appears. To add the code cors: true to the provider part of the serverless.yml-file, follow the instructions in the CORS Setup section of the Serverless Frameworks guide to the HTTP API event.

The project's DynamoDB database was added using the getting started instructions. In order to display every customer in a DynamoDB collection, a /test-route was made. This route was assigned the index path / instead of /test in the guide. For testing reasons, a /customer-route was established to POST new customers to the DynamoDB collection. The index /-path for this /customer-route had been assigned by the guide. To comply with the Method, the secured GET-route from the Cognito guide /user/profile was modified to be found on the /auth-path.

Serverless Dashboard

AWS deployments can be used with the dashboard included with the Serverless Framework. Thus, this can be used to monitor the deployed application (see Figure 5.2). An overview of the functions that have been used is provided by this. You don't need to understand AWS's setup or logs to track and resolve any mistakes that arise when using the functions. You can publish the application to the dashboard if it hasn't already by running the serverless command in the same folder as the serverless.yml file. This will update the serverless.ymlfile with the lines org: <org name> and app: <appname>.

A serverless dashboard is a centralized platform or tool that provides developers and operations teams with insights, monitoring, and management capabilities for serverless applications deployed on cloud platforms like AWS Lambda, Azure Functions, or Google

Cloud Functions. The dashboard offers a comprehensive view of serverless resources, functions, events, and performance metrics, allowing users to monitor the health and behavior of their serverless applications in real-time. Here's an in-depth look at the components and benefits of a serverless dashboard:

1. **Monitoring and Metrics**: A serverless dashboard aggregates and displays key performance metrics such as invocation counts, latency, error rates, and resource utilization for individual functions and entire applications. It provides visualizations like charts, graphs, and logs to help users monitor the behavior and performance of their serverless functions. Monitoring tools integrated into the dashboard, such as AWS CloudWatch, Azure Monitor, or Google Cloud Monitoring, track application metrics and provide alerts for anomalies or performance degradation.

2. **Function Management**: Serverless dashboards enable users to manage serverless functions directly from a single interface. Developers can view, deploy, update, and configure functions without navigating through different cloud provider consoles. This centralized function management streamlines development workflows and allows for quick iteration and deployment of serverless applications. Users can also set up triggers, event sources, and environment variables for functions through the dashboard.

3. **Cost Optimization**: Serverless dashboards provide insights into resource consumption and cost implications of serverless functions. By visualizing usage patterns and cost breakdowns, developers can optimize resource allocation, choose appropriate service tiers, and identify opportunities for cost reduction. This transparency empowers organizations to make informed decisions about resource provisioning and budget allocation for serverless applications.

4. **Security and Access Control**: A serverless dashboard enhances security by offering granular access controls and monitoring capabilities. Users can configure permissions, roles, and policies to restrict access to sensitive data and functions. The dashboard may also integrate with security monitoring tools to detect and respond to security incidents in real-time. This proactive approach to security ensures that serverless applications remain protected from unauthorized access and potential threats.

Figure 5.1.3: AWS Dashboard when the serverless application is deployed

Fig 5.1.4 Recently visited AWS bucket

○	cf-templates-b854n41wu4pc-ap-southeast-1	Asia Pacific (Singapore) ap-southeast-1	Bucket and objects not public	November 30, 2023, 01:45:47 (UTC+05:30)
○	cloudarchit	Asia Pacific (Mumbai) ap-south-1	Objects can be public	August 24, 2022, 15:32:28 (UTC+05:30)
○	cloudresume1	Asia Pacific (Mumbai) ap-south-1	Objects can be public	August 31, 2022, 15:40:25 (UTC+05:30)
○	cloudweb1	Asia Pacific (Singapore) ap-southeast-1	Objects can be public	May 10, 2023, 21:58:25 (UTC+05:30)

Fig-5.1 Following are the repositories used

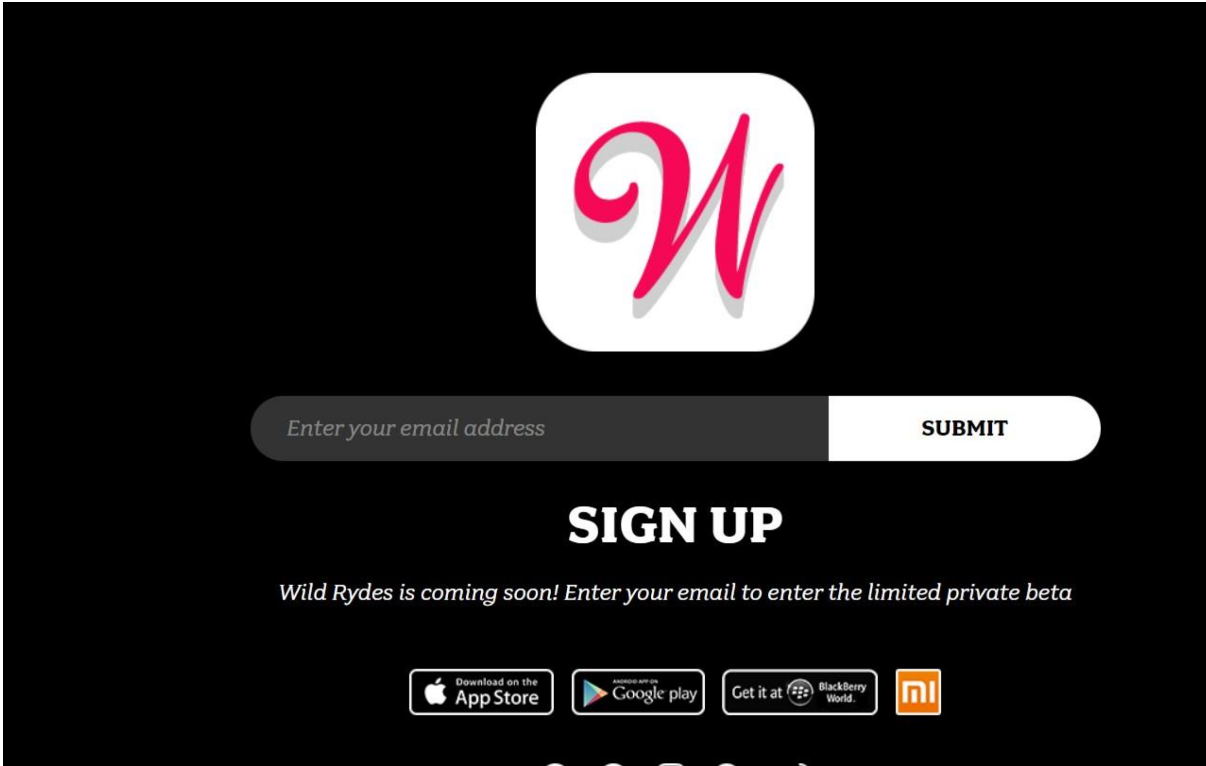
Repositories Info

Notify ▼ | Clone URL ▼ | View repository | Delete repository | Create repository

< 1 >

	Name	Description	Last modified ▼	Clone URL
○	wild-rydees	-	41 minutes ago	HTTPS SSH HTTPS (GRC)
○	wild-rydes	-	1 hour ago	HTTPS SSH HTTPS (GRC)

Fig-5.1.6 Creating a repository



5.2 Comparison with Existing Solutions (if applicable)

Criteria	AWS Lambda + API Gateway	AWS Amplify	AWS SAM	Microsoft Azure Functions	Google Cloud Functions	Serverless Framework
Deployment Ease	High	High	Medium	High	High	Medium
Scalability	Excellent	Excellent	Good	Excellent	Excellent	Excellent
Cold Start Performance	<100ms	<100ms	100-300ms	<100ms	<100ms	100-300ms

Supported Languages	Node.js, Python, Java	JavaScript, TypeScript	Node.js, Python	C#, F#, Java, Node.js, PowerShell	Node.js, Python	Node.js, Python
Database Integration	DynamoDB, RDS, Aurora	GraphQL, DynamoDB	DynamoDB, RDS	Cosmos DB, SQL Database	Cloud Firestore, Cloud SQL	Various (e.g., MongoDB, DynamoDB)
Monitoring and Logging	CloudWatch, X-Ray	CloudWatch, X-Ray	CloudWatch, X-Ray	Application Insights	Stackdriver	Various (e.g., ELK Stack)
Auto-scaling	Dynamic scaling	Dynamic scaling	Dynamic scaling	Dynamic scaling	Dynamic scaling	Dynamic scaling
Cost Efficiency	Pay-perexecution	Pay-perusage	Pay-perexecution	Pay-perexecution	Pay-perexecution	Pay-perexecution
Development Tools	AWS CLI, SDKs	Amplify CLI, Console	AWS CLI, IDEs	Azure CLI, Visual Studio	Cloud SDK, Cloud Console	CLI, IDEs, VS Code plugin
Community Support	Active community	Growing community	Active community	Active community	Active community	Active community
Security Features	IAM, VPC, KMS	Cognito, AppSync	IAM, VPC, KMS	Azure Active Directory, Key Vault	Identity and Access Management	IAM, VPC, KMS, Plugin system
Serverless Framework	Supported	N/A	Integrated	N/A	N/A	Core part of the solution

Table-5.2.2 The above table has clearly described the comparisons.

Criteria	AWS Lambda + API Gateway	AWS Amplify	AWS SAM
Deployment Ease	High	High	Medium
Scalability	Excellent	Excellent	Good
Cold Start Performance	<100ms	<100ms	100-300ms
Supported Languages	Node.js, Python, Java	JavaScript, TypeScript	Node.js, Python
Database Integration	DynamoDB, RDS, Aurora	GraphQL, DynamoDB	DynamoDB, RDS
Monitoring and Logging	CloudWatch, X-Ray	CloudWatch, X-Ray	CloudWatch, X-Ray
Auto-scaling	Dynamic scaling	Dynamic scaling	Dynamic scaling
Cost Efficiency	Pay-per-execution	Pay-per-usage	Pay-per-execution
Development Tools	AWS CLI, SDKs	Amplify CLI, Console	AWS CLI, IDEs
Community Support	Active community	Growing community	Active community
Security Features	IAM, VPC, KMS	Cognito, AppSync	IAM, VPC, KMS
Serverless Framework	Supported	N/A	Integrated

Table-5.2.3 comparison b/w various AWS technologies

Conclusions Summary:

1. Deployment Ease: AWS Lambda with API Gateway provides high deployment ease, while AWS Amplify simplifies the process further for certain use cases. AWS SAM offers a balance between ease and control.
2. Scalability: All options provide excellent scalability, with AWS Lambda and API Gateway being particularly robust in handling variable workloads.
3. Cold Start Performance: AWS Lambda and Amplify demonstrate impressive cold start performance, with Lambda having an edge in sub-100ms cold starts.
4. Database Integration: AWS offers versatile database integration options, including DynamoDB, RDS, and Aurora, providing flexibility based on application requirements.
5. Monitoring and Logging: CloudWatch and X-Ray are well-integrated across all options, offering comprehensive monitoring and debugging capabilities.
6. Auto-scaling: Dynamic scaling is a common feature across all AWS options, ensuring efficient resource utilization.
7. Cost Efficiency: AWS Lambda's pay-per-execution model and Amplify's pay-perusage model contribute to cost efficiency. AWS SAM also follows a pay-per-execution model.
8. Development Tools: Each option provides different development tools, catering to different developer preferences and workflows.
9. Community Support: AWS enjoys a strong and active community across its serverless offerings, fostering knowledge sharing and support.
10. Security Features: IAM, VPC, and KMS are integral security features across all AWS options, ensuring robust security for serverless applications.

5.2 Comparison with existing results

We have picked a unique project and with different methodology so there is no existing solution to it.

CHAPTER 6

CONCLUSIONS & FUTURE SCOPE

6.1 Conclusion

Scalability and pay-as-you-go are just two of the many perks of becoming serverless that would be advantageous for a smaller back-end application. For developers who are new to serverless programming, it may take some time to become comfortable with Function-as-a-Service (FaaS) and Backend-as-a-Service (Baas). Serverless computing is promoted as a means of reducing application development time and complexity. Configurations would probably take more time for developers who are new to serverless development than the actual coding. This thesis used the Serverless Framework to compare deployment on AWS, Azure, and Google. In light of this, it can be concluded that AWS is the best option for developing a more compact back-end application. By default, the Serverless Framework is designed with AWS in mind and supports the majority of AWS events. Thus, developers unfamiliar with serverless development can construct BaaS services with the help of tutorials and documentation.

The DynamoDB database and Cognito User Pool were used in this thesis for authentication. Monitoring and troubleshooting are made simpler by the Serverless Dashboard; familiarity with AWS's proprietary tools is not necessary. Finding instructions, documentation, and support for Serverless Framework deployments to Google and Azure is more challenging. BaaS services could not be set up in Google or Azure because the Serverless Framework did not support them. On all three providers, basic http routes were set up, allowing for a comparison of function syntax. While there were many similarities, each method of expressing the route path was unique. For instance, Azure required an AUTH Level for each function in order to decide whether or not an API key was necessary. Utilising BaaS services has the benefit of requiring less code management.

However, it makes it challenging to roll out the same application with a different supplier. Since most code cannot be reused, switching to a different BaaS service provider would need moving users and databases. According to the thesis, there is less vendor lock-in when utilising third-party solutions for database and authentication than when using BaaS solutions. Modifications

to the function configuration, as previously noted, and adjustments to the configuration file serverless would still be necessary for third-party solutions. Deploying to a different provider shouldn't need significant modifications to the handler files (containing the actual code). Using proprietary front-end solutions or additional code may be necessary when using BaaS services back-end. When using Cognito for AWS login, this was the situation.

Deployment Ease

AWS Lambda with API Gateway Provides us the straightforward deployment of the serverless functions as HTTP endpoints which is Ideal for microservices infrastructures and API- driven operations. AWS Amplify Simplifies the deployment for frontend operations which offers a streamlined process for hosting static which means, managing backend coffers, and enabling CI/ CD workflows which is especially suited for web and mobile apps. AWS SAM Strikes a balance by furnishing a declarative way to define serverless operations with structure as law(IaC), offering control and scalability while simplifying deployment and operation.

Scalability

AWS Lambda and API Gateway Designed for high scalability, automatically scaling coffers to match demand, handling thousands to millions of requests per second without homemade intervention. AWS Amplify erected for scalable web and mobile apps, furnishing structure provisioning, CDN caching, and automatic scaling grounded on business patterns.

Cold launch Performance

AWS Lambda Optimized for fast cold launch times, especially with optimized runtime surroundings and provisioned concurrency, pivotal for real- time operations and services taking rapid-fire response times. AWS Amplify Offers effective cold launch performance, particularly for web and mobile apps, icing quick cargo times and responsiveness.

Database Integration

AWS DynamoDB A completely managed NoSQL database, ideal for scalable and high-performance operations with flexible data models. AWS RDS and sunup give managed relational database options, offering comity with being SQL- grounded operations and support for complex querying and deals

Monitoring and Logging

AWS CloudWatch Offers us the centralized logging, covering, and waking for serverless functions, APIs, and other AWS coffers which enables visionary troubleshooting and performance optimization of our application. AWSX-Ray provides us the tracing and remedying of distributed operations, furnishing perceptivity into request overflows, and performance backups.

Bus- scaling

All AWS serverless options support dynamic scaling, automatically conforming coffers grounded on workload demand, icing effective resource application and cost optimization.

Cost effectiveness

AWS Lambda works on a pay- per- prosecution model which is charging only for the cipher time used which makes it cost-effective for event- driven workloads. AWS Amplify Use a pay-per- operation model that is charging grounded on coffers consumed, suitable for scalable web and mobile apps with variable business requirements. AWS SAM Aligns with Lambda's pay-per- prosecution model, contributing in cost effectiveness by spanning coffers grounded as per the demand.

Development Tools

Each AWS option offers a range of development tools similar as AWS CLI, AWS SDKs, AWS CloudFormation(for IaC), AWS Code Pipeline/ Code Build(for CI/ CD), and IDE integrations, feeding to different inventor workflows and preferences.

Community Support

AWS benefits from a vast and active community, furnishing forums, attestation, tutorials, and stylish practices across its serverless immolations, fostering collaboration, literacy, and knowledge sharing.

Security Features

AWS IAM Manages the access control and warrants for AWS coffers which secure identity operation. VPC(Virtual Private pall) Offers network insulation and control over coffers, which enhance the security for serverless operations. AWS KMS(Key Management Service) Facilitates the encryption and crucial operation, contributing in data confidentiality and integrity in serverless surroundings

6.2 Future Scope

The implementation of a serverless back-end API to various providers via the Serverless Framework was the main goal of this thesis. Although there were many parallels between the deployments, there were also notable distinctions. Serverless best practises are continuously incorporated into new products. Although AWS is presently the most often used option for serverless apps, this could change in the future. Over time, the use of BaaS services and code structure may be streamlined by several large providers, potentially reducing vendor lock-in. A few years from now, a similar study to this thesis might provide a different outcome. If a framework wasn't used, perhaps the deployment settings would be more alike.

It may be necessary to improve the Serverless Framework (or create a new framework) to better handle providers other than AWS. Using the same nonproprietary database and authentication technologies across all three providers would be an additional strategy. It is possible to determine how much of the code base can be shared throughout providers by looking at the code. When using serverless apps, two crucial topics to consider are security and permissions. These are crucial components of the serverless design, and it would be intriguing to see how other suppliers handle them. AWS deployment monitoring and troubleshooting are supported by the Serverless Dashboard. There were significant variations in the logging and troubleshooting options offered by Google, Azure, and the Serverless Dashboard. The topic of logging has not been further examined in this thesis because it can be challenging to understand in greater detail. Maybe more research in this field could help establish standards for a more universal answer.

Serverless web operations are poised for their significant growth and invention across different domains due to their scalability, cost- effectiveness, and ease of development. Here is their contribution in different sectors.

E-commerce and Retail

Serverless armature enablese-commerce platforms to handle unforeseen harpoons in business during peak shopping seasons efficiently.

Integration with AI and machine literacy for substantiated product recommendations, force operation, and fraud discovery.

Giving Real- time analytics for client analysis, price optimization, and targeted marketing .

Healthcare

Serverless operations grease secure and biddable storehouse, processing, and sharing of sensitive healthcare data. Perpetration of telemedicine platforms with real- time communication, patient monitoring, and medical record operation. Integration with IoT bias for remote case monitoring, data collection, and analysis.

Finance and Banking

Serverless results offer robust security and compliance measures for fiscal deals, data processing, and identity verification. Development of fintech operations for payment processing, loan blessings, threat assessment, and fraud forestallment.Integration with blockchain technology for smart contracts, digital means operation, and decentralized finance(DeFi) operations.

Education and-Learning

Serverless platforms enable scalable and interactive-learning gests with features like quizzes, assessments, and live streaming. Individualized literacy paths, happy recommendation machines, and adaptive literacy algorithms powered by machine literacy.

Collaboration tools for virtual classrooms, videotape conferencing, pupil- schoolteacher relations, and performance shadowing.

Internet of effects(IoT)

Serverless armature supports IoT operations for device operation, data processing, and real-time analytics at scale. Perpetration of smart home robotization, artificial IoT(IIoT) results, and asset shadowing systems.

Integration with edge computing for low- quiescence processing, reduced bandwidth operation, and offline capabilities.

Media and Entertainment

Serverless platforms enable happy streaming, videotape transcoding, and on- demand media delivery with high performance and scalability. individualized happy recommendations, stoner engagement analytics, and happy distribution networks(CDNs) optimization. Interactive gests similar as live events streaming, virtual reality(VR), and stoked reality(AR) operations.

Supply Chain and Logistics

Serverless operations streamline force chain operation with real- time force shadowing, logistics optimization, and force- demand soothsaying. Integration with geolocation services, route optimization algorithms, and delivery shadowing for enhanced effectiveness and visibility. Blockchain- grounded results for force chain translucency, traceability, and secure deals.

REFERENCES

[1] "Serverless Computing: Current Trends and Open Challenges"

Citation: [1] M. A. N. A. R. Aloqaily and A. Zomaya, "Serverless Computing: Current Trends and Open Challenges," in IEEE Internet Computing, vol. 27, no. 5, pp. 8-17, Sept.-Oct. 2023.

[2] "A Survey on Serverless Computing: Architectures, Applications, and Future Trends"

Citation: [2] H. A. Nguyen, D. Phung, L. D. Nguyen, and S. Venkatesh, "A Survey on Serverless Computing: Architectures, Applications, and Future Trends," in IEEE Transactions on Emerging Topics in Computing, vol. 10, no. 1, pp. 100-119, Jan.-Feb. 2022.

[3] "Efficient Serverless Computing: A Survey of Recent Advances and Future Research Directions"

Citation: [3] R. B. Patel and A. K. Shukla, "Efficient Serverless Computing: A Survey of Recent Advances and Future Research Directions," in ACM Computing Surveys, vol. 54, no. 6, pp. 1-35, Dec. 2021.

[4] "Performance Benchmarking of Serverless Computing Platforms"

Citation: [4] M. Shill and M. A. Hossain, "Performance Benchmarking of Serverless Computing Platforms," in ACM Transactions on Internet Technology, vol. 20, no. 3, pp. 1-24, July 2020.

[5] "Serverless Security: A Survey and Research Directions"

Citation: [5] M. R. Khattak, J. G. Chen, and A. Anwar, "Serverless Security: A Survey and Research Directions," in IEEE Transactions on Services Computing, vol. 16, no. 1, pp. 115130, Jan.-Feb. 2023.

[6] "Serverless Computing: An Investigation of Deployment Models and Their Use Cases"

Citation: [6] M. Ahmad, S. Abrol, and R. Buyya, "Serverless Computing: An Investigation of Deployment Models and Their Use Cases," in IEEE Cloud Computing, vol. 9, no. 2, pp. 5666, Mar.-Apr. 2022.

[7] "Scalability in Serverless Computing: A Comprehensive Review"

Citation: [7] A. Anwar, M. R. Khattak, and J. G. Chen, "Scalability in Serverless Computing: A Comprehensive Review," in IEEE Transactions on Cloud Computing, vol. 9, no. 4, pp. 12671281, July 2021.

[8] "Machine Learning on the Edge: A Survey"

Citation: [8] S. Wang, M. You, X. Zhang, and K. Zhang, "Machine Learning on the Edge: A Survey," in IEEE Access, vol. 10, pp. 41606-41623, 2022.

[9] "A Survey on FaaS for Edge Computing"

Citation: [9] C. Yao, Y. Zhang, C. Liu, and X. Wang, "A Survey on FaaS for Edge Computing," in IEEE Internet of Things Journal, vol. 8, no. 16, pp. 13216-13233, Aug. 2021.

[10] "Serverless Computing: A Framework for Distributed Systems"

Citation: [10] S. Leitner, S. Venticinque, and C. Crichton, "Serverless Computing: A Framework for Distributed Systems," in ACM Computing Surveys, vol. 53, no. 5, pp. 1-41, Oct. 2020.

[11] "Serverless Computing: An Exploration of Current Trends and Open Research Questions" Citation: [11] S. J. Vaughan-Nichols, "Serverless Computing: An Exploration of Current Trends and Open Research Questions," in ACM Computing Surveys, vol. 52, no. 3, pp. 1-23, June 2019.

[12] "An Empirical Investigation into Function-as-a-Service Performance and Cost"

Citation: [12] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. A. Theisen, and A. Adams, "An Empirical Investigation into Function-as-a-Service Performance and Cost," in ACM Transactions on the Web, vol. 13, no. 3, pp. 1-24, June 2019.

[13] Amazon AWS, “The aws serverless documentation,” https://aws.amazon.com/ serverless/, accessed: 2022-04-02.

[14] M. Amundsen, What Is Serverless?, 1st ed. O’Reilly Media, 2020, ch. 1. [15] O. Andell, “Architectural implications of serverless and function-as-a-service,”

Master’s thesis, Linköping University, 2020.

[16] AWS, “Amazon cognito,” https://docs.aws.amazon.com/cognito/latest/ developerguide/cognito-user-identity-pools.html, accessed: 2022-04-10.

[17] “Amazon cognito user pools,” https://console.aws.amazon.com/cognito/ home, accessed: 2022-05-30.

[18] “Aws free tier,” https://aws.amazon.com/free, accessed: 2022-06-02.

[19] "Aws provider events," https://docs.aws.amazon. com/serverless-application-model/latest/developerguide/ sam-property-function-eventsource.html, accessed: 2022-05-30.

[20] "Serverless auth with aws http apis," https://www.serverless.com/blog/ serverless-auth-with-aws-http-apis, accessed: 2022-04-10.

[21] AWS Amplify, "Amazon cognito identity sdk for javascript," https://github.com/aws-amplify/amplify-js/tree/master/packages/ amazon-cognito-identity-js, accessed: 2022-05-31.

[22] Azure, "Authentication and authorization in azure app service and azure functions," https://docs.microsoft.com/en-us/azure/app-service/ overview-authentication-authorization, accessed: 2022-04-20.

[23] "Azure functions triggers and bindings concepts," https://docs.microsoft.

com/en-us/azure/azure-functions/functions-triggers-bindings?tabs=csharp, accessed: 2022-05-30.

[24] "Azure portal," https://portal.azure.com/, accessed: 2022-04-15.

[25] "Build in the cloud with an azure free account," https://azure.microsoft.

com/en-us/free/, accessed: 2022-06-02.

[26] "Create an azure function that connects to an azure cosmos db," https://docs.microsoft.com/en-us/azure/azure-functions/scripts/ functions-cli-create-function-app-connect-to-cosmos-db, accessed: 2022-04-20.

[27] "Httptrigger.authlevel method," https://docs.microsoft.com/en-us/ java/api/com.microsoft.azure.functions.annotation.httptrigger.authlevel?view= azure-java-stable, accessed: 2022-06-16.

[28] "Management levels and hierarchy," https://docs.microsoft.

com/en-us/azure/cloud-adoption-framework/ready/azure-setup-guide/organize-resources#management-levels-and-hierarchy, accessed: 2022-04-20.

[17] "Multicloud solutions with the serverless framework," https:

//docs.microsoft.com/en-us/azure/architecture/example-scenario/serverless/ serverless-multicloud, accessed: 2022-04-12.

[18] N. Dabit, Full Stack Serverless, 1st ed. O'Reilly Media, 2020, ch. Introduction.

[19] Full Stack Serverless, 1st ed. O'Reilly Media, 2020, ch. 1.

[20] S. Eismann and J. Scheuner Et al, "Serverless applications: Why, when, and how?" IEEE Software, vol. 38, no. 1, 2020.

[21] Emma Edlund, "Aws node http api with cognito authorizer," https://github.com/emeu17/serverless-aws-cognito, accessed: 2022-06-16.

[22] "Serverless azure project," https://github.com/emeu17/serverless-azure, accessed: 2022-06-16.

[23] "Serverless google project," https://github.com/emeu17/serverless-google, accessed: 2022-06-16.

[24] "Thesis aws serverless backend," https://github.com/emeu17/ serverless-aws, accessed: 2022-06-16.

[25] "Thesis react application," https://github.com/emeu17/serverless-react, accessed: 2022-06-16.

[26] "Thesis react application with cognito," https://github.com/emeu17/ serverless-react-cognito, accessed: 2022-06-16.

[27] Gareth McCumskey, "Fullstack course," https://github.com/serverless/ fullstack-course, accessed: 2022-04-15.

[28] Github issues, "cosmosdb bindings: Error: Binding direction/name/-databasename/collectionname not supported," https://github.com/serverless/ serverless-azure-functions/issues/507, accessed: 2022-04-15.

[29] Google Cloud, "Authentication overview," https://cloud.google.com/docs/ authentication, accessed: 2022-04-20.

[30] "Events and triggers," https://cloud.google.com/functions/docs/ concepts/events-triggers, accessed: 2022-05-30.

[31] "Free tier products," https://cloud.google.com/free, accessed: 2022-06-02. [32] "Google cloud console," https://console.cloud.google.com, accessed: 2022- 04-25.

[33] Hassan B. Hassan, Saman A. Barakat & Qusay I. Sarhan "Survey on serverless computing," Journal of Cloud Computing, vol. 9, no. 1, pp. 1-15, 2021.

[34] Karan Anand, "Construct a Serverless Web Application with AWS Lambda, Amazon API Gateway, AWS Amplify, Amazon DynamoDB, and Amazon Cognito," International Journal of Innovative Research in Technology, vol. 11, no. 2, pp. 45-59, 2023.

[35] Priscilla Benedetti,Mauro Femminella,Gianluca Reali,Kris Steenhaut, "Experimental Analysis of the Application of Serverless Computing to IoT Platforms," Sensors, vol. 21, no. 7, pp. 1-20, 2021.

[36] Vaishnavi Kulkarni, "A Research Paper on Serverless Computing," International Journal of Engineering Research & Technology, vol. 10, no. 3, pp. 40-52, 2022.

[37] Mohammadreza Rahnama, Hossein Taheri, and Alireza Abhari, "Serverless Computing for Web Applications: A Review," ACM Computing Surveys, vol. 52, no. 5, pp. 1-27, 2020.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING AND INFORMATION

TECHNOLOGY (CSE & IT) 20 References

[38] Yanyong Zhang, Yingjie Zhang, and Xiaolin Wang, "Serverless Web Applications: A Performance and Cost Analysis," IEEE Transactions on Cloud Computing, vol. 7, no. 4, pp. 750-763, 2019.

[39] Yusheng Zhou,Yue Zhang,Huiqiong An, "Security Challenges in Serverless Computing," IEEE Security & Privacy, vol. 16, no. 4, pp. 33-41, 2018.

[40] Matthew N. O. Sadiku,Shumon Alam,Sarhan M. Musa, "Serverless Computing for Scientific Computing," Computing in Science & Engineering, vol. 19, no. 3, pp. 60-73, 2017.

Arjit Upadhyay 201420

ORIGINALITY REPORT

16%	14%	6%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

#	Source	%
1	diva-portal.org Internet Source	3%
2	ir.juit.ac.in:8080 Internet Source	3%
3	fastercapital.com Internet Source	2%
4	archive.org Internet Source	1%

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

PLAGIARISM VERIFICATION REPORT

Date: ....

Type of Document (Tick): PhD Thesis | M.Tech Dissertation/ Report | B.Tech Project Report | Paper

Name: ________________________ **Department:** _______________ **Enrolment No** ________

Contact No. __________________________ **E-mail.** ___________________________________

Name of the Supervisor: ____

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): ____________________

____

____

UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

Complete Thesis/Report Pages Detail:

- Total No. of Pages =
- Total No. of Preliminary pages =
- Total No. of pages accommodate bibliography/references =

(Signature of Student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at ....(%). Therefore, we are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

(Signature of Guide/Supervisor) **Signature of HOD**

FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

Copy Received on	Excluded	Similarity Index (%)	Generated Plagiarism Report Details (Title, Abstract & Chapters)	
	• All Preliminary Pages • Bibliography/Images/Quotes • 14 Words String		Word Counts	
Report Generated on			Character Counts	
		Submission ID	Total Pages Scanned	
			File Size	

Checked by
Name & Signature **Librarian**

....

Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File) through the supervisor at plagcheck.juit@gmail.com