IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Explore Deployment Strategies for AWS Lambda Functions using AWS CodePipeline and AWS CodeBuild

1Mahesh B. Kadam, 2Ganesh S. Wayal, 3Naganath S. Bagal

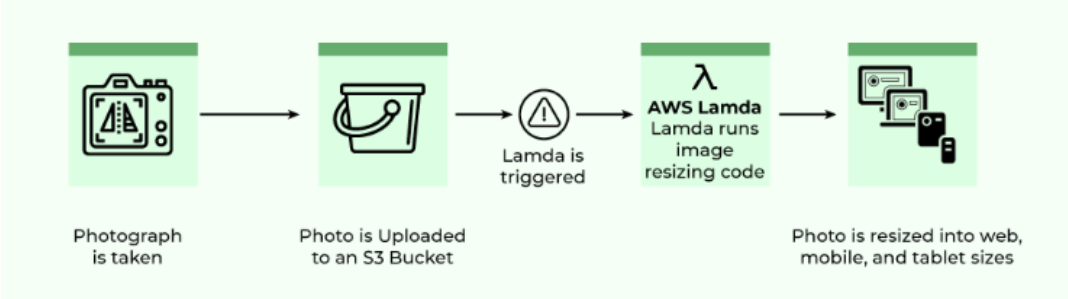
1Student, 2Head of Department., 3Professor

Abstract— An increasing number of businesses are starting their digital transformation journeys in order to forge stronger relationships with their consumers and ultimately achieve long-term success. Embracing DevOps ideas and practices may help technology businesses make the cloud journey easy, efficient, and successful. Many AWS services, notably those related to deployment and monitoring, are based on these concepts, which are intrinsic to the AWS platform. Implementing DevOps methods with AWS provides organisations with a robust platform for collaborative and efficient software development. DevOps is a methodology for enhancing software products via global automation of integration and delivery pipelines, tight cooperation across teams, and quick release cycles. Organisations may gain more efficiency, better cooperation, and more scalability by using AWS services for infrastructure automation, continuous integration, and deployment. Accelerated time-to-market, simplified procedures, and optimized resource utilization are achieved when DevOps concepts are combined with AWS's scalable and flexible infrastructure. Adopting DevOps with AWS has some advantages, such as quicker software delivery, better communication, and the capacity to react to changing needs, but there may be some problems during deployment. As a whole, DevOps with AWS allows businesses to stay ahead of the competition by rapidly delivering high-quality software.

Keywords: AWS, Lambda Function, CodePipeline, CodeBuild, Deployment, Strategy, DevOps

I. INTRODUCTION

The development and deployment of apps has been completely transformed by serverless computing, which also makes resource management and scalability simpler. One of the best serverless computing services is AWS Lambda, which lets you execute code in response to different kinds of events [1,2]. When working on a project that involves many AWS Lambda functions, automating deployment may save time and minimise mistakes. Serverless applications need AWS Lambda functions because they enable developers to execute code without the need to set up or manage servers. An Amazon serverless computing solution called AWS Lambda executes code and takes care of the underlying computer resources, such as (EC2), automatically [3]. It is a computer service that is event-driven. It enables the automated execution of code in response to a variety of events, including state transitions, HTTP requests from the Amazon API gateway, and table changes in Amazon DynamoDB. It also lets the user build their own back-end services and extend to other AWS services with custom logic. Simply create the code, for instance, and submit it as a.zip file or any other kind of container image. Code is executed on high-availability computer infrastructure to operate the service. After that, it handles every administrative task related to that computing resource, including:



A. Deployment Strategies

Your software's delivery method may be defined by your deployment strategy. Depending on their business strategy, organisations adopt various deployment tactics. Some may choose to release completely tested software, while others may seek user input and allow beta testing of features still under development. Several methods of deployment will be discussed later on.

1) In-Place Deployments

This approach involves stopping the programme on all instances in the deployment group, installing the most recent application revision, and then starting and validating the new version of the application. A load balancer may be used to deregister each instance during deployment and then restore them to service after the deployment is finished. If a service outage were to occur, an in-place deployment could be executed all at once, or updates might be made incrementally. One-time, half-time, and all-at-once deployment modes are available in AWS CodeDeploy and AWS Elastic Beanstalk. Blue-Green deployments provide the same deployment methodologies as in-place deployments.

2) Blue-Green Deployments

Deploying an application in a blue-green fashion involves shifting traffic between two identical environments that are running different versions of the programme. This approach is also known as red-black deployment. To reduce the likelihood of interruptions and problems with rollback functionality, blue-green deployments are a great tool to have on hand while updating applications. You may test and monitor the new version of your application before redirecting traffic to it and rolling back on problem detection using blue-green deployments. The old version can still be launched alongside the new version.

3) Canary Deployments

Two increments are used to move the traffic. A blue-green technique that is less daring and uses a phased approach is known as a canary deployment. There are two main approaches: the first involves deploying and testing new application code in a limited setting, and the second involves rolling out the code to the rest of the environment after it has been approved. Smooth Rollouts With a linear deployment, the amount of time that passes between each shift of traffic is exactly equal. Each linear option specifies the percentage of traffic moved and the number of minutes between each increment; you may pick from these predetermined values.

4) All-at-once Deployments

When a deployment is all-at-once, all traffic is moved from the old to the new environment simultaneously.

II. LITERATURE REVIEW

Alteen, et al focuses on the various deployment tactics, capabilities of Elastic Beanstalk, and how to automate deployments for multi-tier systems. With the aid of a continuous integration server and other AWS services, developers may set up continuous builds, tests, and code deployments using the CI/CD pipeline. As a deployment resource, AWS Elastic Beanstalk may be integrated with the CI/CD process. Elastic Beanstalk can retrieve and distribute code from a Git repository using AWS CodeCommit, another CI/CD capability. One of the most important aspects of managing and routing traffic among your instances is load balancing. You should aim for both high performance and high availability when launching apps in your settings. If your application makes use of AWS resources, you can keep tabs on them using AWS Auto Scaling. Consistent, reliable functioning is ensured by the service's automated capacity adjustment. Ech2 instances, Spot Fleets, Amazon ECR jobs, DynamoDB tables, indexes, and Aurora Replicas are all part of the resources that may be managed with the use of scaling strategies. [4].

Challa, Narayana & Devineni, Siva Karthik & Karangara, Rajath presented the rise of cloud computing has changed the face of contemporary IT infrastructures, and one of the most prominent cloud providers is Amazon Web Services (AWS). Unveiling AWS's full potential and showcasing its transformational advantages for consumers and organisations, the study delves into its subtleties. The research examines AWS services from the angles of scalability, dependability, and cost-effectiveness by examining a number of AWS features. In order to maximize operational efficiency and encourage innovation, research aims to determine critical strategies and best practices for using AWS resources. As part of the investigation, we will look at the technical framework, security measures, and integration capabilities of AWS. Stakeholders that want to use all the features of AWS will find this study an essential resource; it will help them understand how AWS can boost digital transformation initiatives and usher in a new age of technological advancement. [5].

Makani, Sai Teja discussed that when it comes to cloud computing, organisations that want to take advantage of the scalability, flexibility, and cost-effectiveness that cloud platforms provide must priorities efficient infrastructure management. A pioneer in the field of infrastructure as code (IaC), HashiCorp's Terraform has become an indispensable tool for automating the setup and administration of cloud resources, especially in the context of the AWS ecosystem. This in-depth analysis delves into the many ways Terraform contributes to the optimization of serverless deployments and the orchestration of AWS cloud architecture. Users may describe the intended state of their infrastructure without detailing the implementation specifics using Terraform's declarative vocabulary, which offers a succinct and easy method to building infrastructure settings. In order to make actions like resource provisioning, modification, and deletion controllable and predictable, terraform keeps track of the infrastructure state via its state management system. Organisations may adopt a multi-cloud approach and use Terraform's capabilities for managing large infrastructure installations since it delivers a consistent workflow and tools across varied cloud environments and supports numerous cloud providers. The modularity, reusability, and support for collaborative workflows that are essential characteristics of Terraform allow organisations to simplify infrastructure provisioning, encourage code reuse, and cultivate team cooperation. Integrating with Terraform Cloud and importing existing infrastructure both increase Terraform's usefulness; the latter offers enterprise functionality for managing infrastructure at scale, as well as centralized state management and communication tools. Terraform plays a crucial part in current DevOps methods and its influence on AWS infrastructure management is explained in this paper via case studies, empirical research, and practical examples. In order to optimise costs, dependability, and productivity, empirical studies show that terraform is effective in optimizing AWS infrastructure management. Terraform Cloud's growing capabilities and connectivity with AWS services are shown by its latest advances, which include integration with AWS Organisations and CI/CD pipelines [6].

Puripunpinyo, Hussachai & Samadzadeh, M. provided useful observation that One of Amazon's most well-known serverless architectures is AWS Lambda, which stands for Amazon Web Services. The Java Virtual Machine (JVM), Python, and JavaScript are the three platforms that it presently supports. Because it supports so many languages other than Java, the JVM has the potential to be the most complex of the three platforms. Conflicts may also arise in projects due to the complicated dependency structure, versioning, and the class loader. Instead of deploying an application that comprises several services, with a serverless architecture each service is deployed as a function. Each deployment artefact used with AWS Lambda must be self-contained, which implies that all resources and dependencies must be included in a single jar file. It's possible that this file size exceeds the maximum that AWS Lambda allows. To create standalone artefacts, developers often use

plugins for build tools; however, these tools often do not know which class and resource files a function requires. Consequently, the artefact is not in its most optimal form. In this work, we show that optimizing an artefact may generally make it run faster and use resources more efficiently. To back up design choices during AWS Lambda development, this article also details the outcome of an anecdotal experiment about the overhead of distant function calls [7].

Villamizar, et al provided useful observation that A number of major internet firms are using the microservice architectural pattern to build, test, deploy, scale, run, and update their huge applications on the cloud. These companies include Amazon, Netflix, and LinkedIn. Adopting this design has several benefits, such as increased agility, self-sufficiency in development, and scalability, but it also comes with significant infrastructure expenditures. This article compares the three methods used to build and launch a web app with identical scalability scenarios in terms of cost: First, a monolithic design; second, a cloud customer-operated microservice design; and third, a cloud provider-operated microservice design. Microservices, according to the test findings, may assist lower infrastructure costs when compared to traditional monolithic designs. There is a 70% or more drop in infrastructure expenses when services designed to install and scale microservices are used. Finally, we go over the difficulties we encountered while launching microservice apps [8].

III. PROPOSED METHODOLOGY

A. Research Design

We have chosen design science research as our framework because it is applicable to many branches of engineering and computer science, focuses on the creation of an artefact, and seeks to advance knowledge that experts in the field can use to solve problems in their work. The created product needs to be an effective technological answer to a pertinent business issue. Effective presentation to both technology- and management-oriented audiences is essential, as is evaluation of the design based on practicality, quality, and efficiency. Researchers may get a better understanding of the issue with the help of the artefact, which allows for re-evaluation and the eventual improvement of the intended artefact. The result is an iterative and continuous process because of the build-and-evaluate loop that is established.

B. CI/CD pipeline

It is essential for organisations to consistently create and distribute new software. Without any explicit cooperation or coordination between teams, it is possible for teams to have dozens, if not hundreds, of separate services that run according to their own schedules. The programme as a whole and individual services might be severely affected by a single poorly executed update to a service. With a CI/CD pipeline, you can ensure that deployments are quick, secure, and consistent. Here's how:

- Continuous Integration (CI) refers to the practice of merging changes as often as feasible into the main branch. A build is created and automated tests are run against it to confirm these changes. The purpose of this method is to make it easier to incorporate new releases.

- When a change completes each step of the production pipeline, it is distributed to consumers as part of Continuous Development (CD). Only a failing test may prohibit a new modification from being pushed to production; no human action is required. Quickening the feedback loop with consumers and reducing release timeframes are the goals of this strategy.

C. AWS CodePipeline

One completely managed continuous delivery option is AWS CodePipeline 25, which lets you model, visualize, and automate software release processes. Software updates go through a release process that is described by a workflow architecture called a pipeline. This process typically consists of at least two steps. A revision is a set of tasks executed on the source location specified for the pipeline, and a stage is a set of activities conducted on an action. It may include data, configuration, build output, or source code. A pipeline may handle several updates concurrently.

D. AWS CodeBuild

In order to create software packages that are prepared for deployment, AWS CodeBuild compiles the source code, executes tests, and provides a fully managed continuous integration service. There is a pay-as-you-go pricing approach, and there's no need to supply, maintain, or scale build servers. You may build a CI/CD pipeline by combining CodeBuild with AWS CodePipeline. The AWS command line interface (CLI) and the console both allow users to set up build projects. The location of the source repository, the runtime environment, the build commands, the container's assumed IAM role, and the compute class needed to conduct the build must all be specified. A buildspec.yml file allows for the optional specification of build commands.

E. Simulation

A proxy's collection of a request and answer is known as simulation. What we term "service virtualization" is the process that lets clients record simulations and then utilise that data to recreate the service. Because they are the fundamental building block of this approach, simulations are crucial. When a request comes in while the proxy is in simulation mode, it must search for another simulation that contains the identical request. Considered areas include:

- Destination
- Query parameters
- Scheme, i.e. HTTP or HTTPS protocol
- Path
- HTTP method
- Path

F. Two units

There are two main parts to the Umarell project: the proxy farm and the Umarell backend.

- Umarell Backend: A client-side application programming interface (API) for managing proxies, simulations, and scenarios using CRUD operations. Data storage that retains information forever, including client data.

- Proxy farm: An environment that provides network access and security to proxies is known as a proxy farm. The proxy farm is engaged whenever a client performs an activity that impacts proxies using the Umarell Backends' API. It offers a few APIs that the Umarell Backend may use to modify proxies.

IV. RESULT AND DISCUSSION

A. Projects organization

Two large regions of Umarell operate independently of one another; they need only exchange information in order to draw upon the services of the other. Because of this separation, which has several advantages, we may store the two components in separate Git repositories on AWS CodeCommit. To get the Hoverfly image from the site spectolabs/hoverfly, another source is DockerHub, an external image repository. The majority of the code is Python code that implements AWS Lambda functions. In order to keep the Lambda instance small and fast, designed functions use as little code as feasible. To make sharing common libraries and utilities simpler, they have been put in Lambda layers.

1) Umarell Backend project

Being a Serverless Framework project, it follows a certain template for its serverless architecture in a configuration file named serverless.yml. This is an entirely serverless component of the application. A lot of FaaS and event-driven paradigm work goes into it [10]. An AWS Lambda function is associated with each endpoint in the customer's API, which is based on AWS API Gateway 3. Serverless Framework is best suited for this scenario. The functions folder contains all of the declared Lambda functions. One handler is included in each file. The ability to break down the handler function into modules, manage smaller functions, and have a distinct and transparent process for each file is a huge boon to maintainability. These roles include:

- A proxy's CRUD operations are ReadProxy, CreateProxy, ModifyProxy, and DeleteProxy.
- The CRUD operations on simulations saved in the database, which include ReadSimulation, CreateSimulation, ModifySimulation, and DeleteSimulation.
- Using the ReadScenario and LoadScenario methods, you may access scenarios saved in the database or import them into the proxy, which is the remote Hoverfly instance.
- When changes are noticed on the table proxies, ProxyWatcher is activated and various actions are conducted based on this.
- Proxy farms provide feedback on Fargate jobs by using UpdateProxy via the Umarell Backend API.

The list of functions that need to be constructed is specified in the template's key functions. Just to illustrate:

```
functions:
  # ...
  CreateProxy:
    handler: functions/api/proxy/create.handler
    role: CreateProxyRole
    package:
      exclude:
        - ./**
      include:
        - functions/api/proxy/create.py
        - proxy/**
        - storage/**
        - utils/**
        - api_builder/**
      individually: true
    events:
      - http:
          path: proxies
          method: post
```

The function is required to be executed whenever a POST request to the path/proxies comes, as specified by the nested key events. This HTTP request is really associated with a call to a Lambda function in practice. The function that must react to the request is defined by its route in the project, as specified by the key handler.

2) Umarell Proxy Farm project

Named app.py, this platform's entry point file primarily instantiates the CloudFormation stack. An object-oriented class describes the stack by declaring each component of the infrastructure in a sequential fashion. With CDK, you may do so with the help of abstract factory-based capabilities, which improves both usability and development speed. To further decouple duties, this project's Lambda functions are the only ones allowed to deal with proxies, which means that only Fargate tasks and Hoverfly instances may be used. The following functions take in arguments from the Umarell Backend (specifically, an event), modify the resources based on certain environment variables, and return the results:

- RunProxy takes a proxy ID and proxy mode as inputs, and then uses the appropriate Hoverfly mode to execute a job.

- Once StopProxy has the ARN of a proxy's task, it will terminate that job.
- An Event Bridge rule initiates ProxyIsRunningFeedback at task start. The event is then reported to the Umarell Backend using its API.
- An Event Bridge rule triggers ProxyHasStoppedFeedback when the job is stopped. The event is then reported to the Umarell Backend using its API.

B. DevOps Pipeline

Git is an open-source distributed version control system that can efficiently manage projects of any size, from very tiny to very big. It is free and open-source. In addition to all that, it is well recognised among developers, has a sizable community, and thorough documentation. With AWS CodeBuild and AWS CodePipeline, among others, you have everything you need to construct a continuous integration and continuous delivery pipeline.

1) CI/CD pipeline

At this moment, it is expected that all pipelines use CD practice. This means that the pipeline will run whenever there is a change in the source location. For developers, this means that the pipeline is activated whenever they submit code to the repository. Figure 1 is a sequence diagram showing the interactions between the AWS services during a code push to a CodeCommit repository.

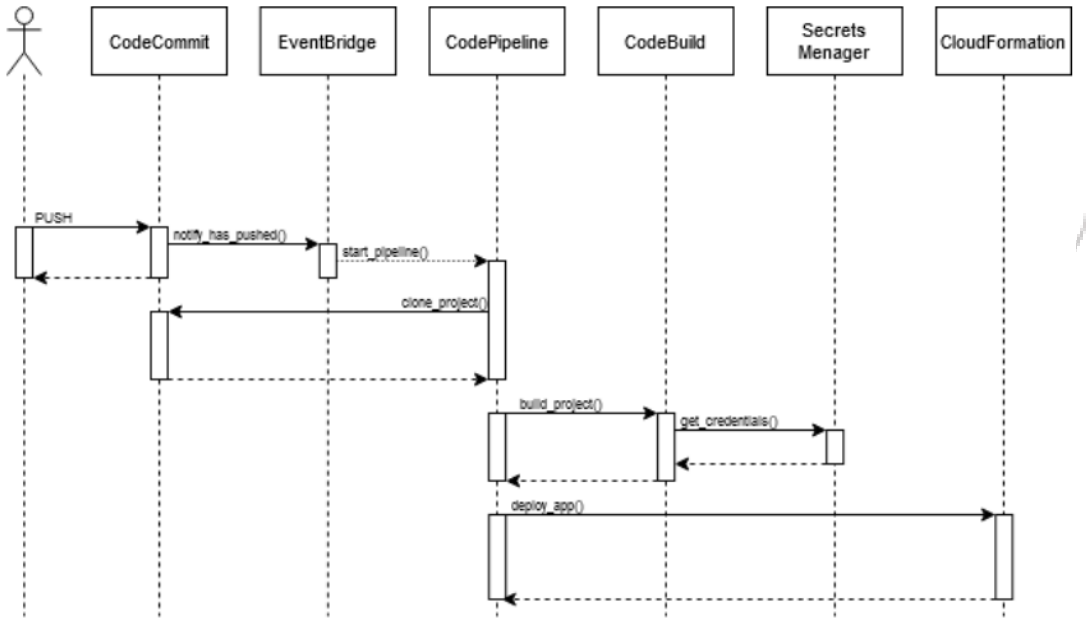


Figure1: CI/CD pipeline sequence diagram beginning

2) Monitoring and observability

Although the pipeline is somewhat basic, it serves to demonstrate the requirements of a serverless application [9], which is acceptable given the project's early state. Because distributed infrastructure components use virtualization and several software abstraction layers, controllability becomes difficult, if not impossible, to achieve. The standard method for understanding how various systems and hardware components is performing is to keep an eye on infrastructure performance indicators and logs. Currently, CloudWatch and X-Ray are the two primary AWS services that are used for these purposes. Lambda has the first one turned on by default. Logging application messages is indeed as simple as publishing to standard output. The CloudWatch services will then gather these and display them on a consolidated dashboard. Then, X-Ray was added to help with tracing function calls, checking performance, debugging, and identifying and fixing performance faults. The Umarell Backend's Lambda and API Gateway services may be enabled with minimum setup, making it a breeze to use.yml:

```
provider:
    iamRoleStatements:
        - Effect: Allow
          Action:
              - xray:PutTraceSegments
              - xray:PutTelemetryRecords
          Resource: "*"
    tracing:
          lambda: true
          apiGateway: true
```

By supplying the build prop to the Function construct, as shown below, X-Ray may be enabled in the Lambda function in CDK:

```
aws_lambda.Function(self, 'LambdaId',
    function_name='LambdaName',
    # other construct parameters
    tracing=aws_lambda.Tracing.ACTIVE)
```

C. Evaluation

1) DevOps pipeline

Having two distinct repositories for the two major pieces allows for two independent source locations, which are used in two separate processes. Continuous Development ensures that the linked pipeline is started whenever a change is detected in one of the two repositories, without affecting the other.

i. Monitoring and observability

Given the project's early state, the pipeline is very simplistic; yet, it adequately demonstrates the requirements for a serverless application.

- Alarms: A specialized group will be formed in the future to use DevOps principles. To ensure that recorded data corresponds to what is in the proxy farm, for instance, if a proxy's status is running, it means that there must be a task in execution, or to inform the arrival of unusual occurrences, such as functions returning a 5xx status code, CloudWatch alarms will be introduced.

- External services: Next, you need to devise plans to ensure you have a complete picture. The integration of CloudWatch logging and analytics with AWS X-Ray trace data may be enhanced. Then, third-party services might be used to alter the acquired data in a different way, resulting in, ideally, more accurate information.

- Responsibility: The placement of proxies determines who is responsible for monitoring proxy farms. Umarell is responsible for monitoring whether the Umarell Proxy Farm is used. Umarell might make available pre-made templates with the necessary elements to make the multi-account solution a breeze to implement.

ii. Testing

At the moment, whenever a change is delivered, testing is done by running unit tests at the microservice level. It serves its purpose; however, it is insufficient for testing the application's overall behaviour due to the lack of evaluation of interactions between the two components. This might be accomplished by adding another pipeline. Two separate tasks could work in tandem to construct the present pipeline, but this would create two separate testing environments. This allows for the execution of system testing, acceptance testing, integration testing, and so on. However, maintaining test data, developing intricate settings, and writing extensive integration suites are all challenges that make this kind of testing particularly challenging for distributed systems like this one. If you're using Umarell, you should put it into action to check the two-way communication layer.

iii. Consequences of serverless

The application layer is close to the developed pipelines. It is unnecessary for it to handle server or virtual machine provisioning or management. By doing so, it may construct potentially massive sections of the infrastructure using the Infrastructure as Code methodology. Benefiting from a product perspective rather than a project one, where success is measured by business KPIs, this shifts the emphasis of operations at higher levels towards challenges facing the company. In a DevOps team, the operations side is in charge of making sure the systems and services are running well. An operations engineer looks at the bigger picture, while a software developer is more concerned with the details.

D. Serverless Computing

Through the use of serverless computing, a collection of cloud services, developers are free to concentrate on creating and releasing code rather than worrying about the underlying infrastructure. Choosing the correct serverless computing type is important for both performance and cost. The cloud service providers included in this research all provide a variety of serverless computing options to meet the demands of their clients.

i. Amazon AWS

Lambda [11] and Fargate [12] are AWS's two main serverless computing offerings. While Fargate is more expensive but offers more configuration options, Lambda is more popular since it provides a fast and inexpensive alternative for serverless computing. Duration and request amount are the two aspects that determine the price of a lambda. Time is calculated by multiplying the quantity of GiB of RAM by the length of the computing process [13]. As seen in table 1, the cost for requests is a flat rate per million requests delivered to Lambda. Three distinct buckets are used to classify the duration cost according to the monthly processing power. The lower the cost per second becomes as more computation is done. Lambdas typically have an average runtime of around 250ms when used as an HTTP endpoint (API).

Table 1: AWS Lambda Pricing

Architecture	Duration	Price per GB-second	Price per 1M requests
x86	First 6 Billion GB-hour / month	$0.06	$0.20
Arm	First 7.5 Billion GB-hour / month	$0.048	$0.20

More precise control over the processing power allocated to each serverless request is possible with Fargate [14]. This implies that the quantity of storage, virtual CPUs, and RAM may be specified by the client. As a result, those three factors determine the Fargate's price. (Not including the 30GiB of free storage) The price of storage is $0.000133 per GiB per hour. The hourly rate for the virtual CPUs is $0.0144309. In conclusion, the ram costs $0.0015838 per GiB per hour. Keep in mind that there are certain restrictions that need a certain minimum ratio of RAM to virtual CPUs. Table 2 displays some of these limits.

Table 2: AWS Fargate Constraints

CPU	Memory Values
0.25 vCPU	0.5 GB, 1 GB, and 2 GB
1 vCPU	Min. 2 GB and Max. 8 GB, in 1 GB increments
4 vCPU	Min. 8 GB and Max. 30 GB, in 1 GB increments
16 vCPU	Min. 32 GB and Max. 120 GB, in 8 GB increments

V. CONCLUSION

We can say that AWS Lambda functions are essential to serverless apps, it may be difficult to manage and deploy several functions. Due to the model's fragmentation, which might result in an inadequately thorough viewpoint, designing serverless architecture can be challenging. Finding best practices and commonalities requires looking at a programme through the lens of a microservices application, identifying shared characteristics, and learning how to manage consistency in a distributed setting. Additionally, this aids businesses in selecting the best cloud computing provider, taking into account deployment difficulties and price models. One of the major challenges of serverless is handling operations, despite the model's seeming simplicity. Although it offers liberation from server management, it nevertheless necessitates the improvement of DevOps methods. Lastly, a proxy farm has to be able to take in instructions (such how to start and stop proxies) and respond with actions (like how to report that a proxy has been launched successfully).

REFERENCES

[1] Serverless Architectures with AWS Lambda. (2019). url: https://docs.aws. amazon. com / lambda / latest / dg / configuration - vpc. html (visited on 08/24/2020) (cit. on p. 12).

[2] Puripunpinyo, H., and Samadzadeh, M. (2017). *Effect of optimizing java deployment artifacts on AWS lambda*. IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 438–443.

[3] Vahidinia, P., & Farahani, B., and Aliee, F. S. (2020). Cold start in serverless computing: Current trends and mitigation strategies. International Conference on Omni-layer Intelligent Systems (COINS), pp. 1–7.

[4] Alteen, N., & Fisher, J., & Gerena, C., & Gruver, W., & Jalis, A., & Osman, H., & Pagan, M., & Patlolla, S., & Roth, M. (2019). Deployment Strategies. 10.1002/9781119549451.ch06.

[5] Challa, N., & Devineni, S. K., & Karangara, R. (2022). A Deep Dive into Amazon Web Services: Unlocking the Potential. Journal of Artificial Intelligence & Cloud Computing, 1, 2-5. 10.47363/JAICC/2022(1)179.

[6] Makani, S. T. (2021). *Deep Dive into Terraform for Efficient Management of AWS Cloud Infrastructure and Serverless Deployment*. 8, 6.

[7] Puripunpinyo, H., & Samadzadeh, M. (2017). *Effect of Optimizing Java Deployment Artifacts on AWS Lambda*. 10.1109/INFCOMW.2017.8116416.

[8] Villamizar, M., & Garcés, O., & Ochoa, L., & Castro, H., & Salamanca, L., & Verano M., Mauricio & C., & Gil, R., Santiago & V., Carlos & Z., Angee & L., Mery. (2016). Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. 10.1109/CCGrid.2016.37.

[9] Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., and Chen, H. (2020). Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. New York, NY, USA: Association for Computing Machinery, p. 467–481. [Online]. Available: https://doi.org/10.1145/3373376.3378512

[10] Ling, W., Ma, L., Tian, C. and Hu, Z. (2019). Pigeon: A dynamic and efficient serverless and faas framework for private cloud. International Conference on Computational Science and Computational Intelligence (CSCI), pp. 1416–1421.

[11] Serverless Computing - AWS Lambda - Amazon Web Services — aws.amazon.com. https://aws.amazon.com/lambda/, [Accessed 23- Apr-2023].

[12] Serverless Compute Engine–AWS Fargate–Amazon Web Services — aws.amazon.com. https://aws.amazon.com/fargate, [Accessed 29-Apr2023].

[13] Serverless Computing – AWS Lambda Pricing – Amazon Web Services — aws.amazon.com. https://aws.amazon.com/lambda/ pricing/ [Accessed 29-Apr-2023].

[14] Serverless Compute Engine–AWS Fargate Pricing–Amazon Web Services — aws.amazon.com. https://aws.amazon.com/fargate/pricing, [Accessed 29-Apr-2023].