ISSN 2614-0047

Optimizing AWS lambda code execution time in amazon web services

Muh. Awal Arifin [a,1,*], Ramdan Satra [a,2], Lukman Syafie [a,3], Ahmad Mursyidun Nidhom [b,4]

[a] Universitas Muslim Indonesia, Makassar 90231, Indonesia
[b] National University of Malaysia, Malaysia
[1] 13020190363@umi.ac.id; [2] ramdan.satra@umi.ac.id; [3] lukman.syafie@umi.ac.id; [4] p122056@siswa.ukm.edu.my
* corresponding author

ARTICLE INFO

Article history
Received January 22, 2023
Revised February 4, 2023
Accepted February 26, 2023

Keywords
Cloud computing
Serverless
AWS lambda
S3
DynamoDB
Amazon web services

ABSTRACT

One of the problems in providing infrastructure is the lack of interest in managing infrastructure. AWS Lambda is a FaaS (Function as a Service) service that allows users to run code automatically in an environment managed by Amazon Web Services. In this study, the method used is to collect data on code execution time at various input sizes, then perform an analysis of the factors that affect execution time. Furthermore, optimization is carried out by selecting the appropriate memory size and proper coding techniques to improve performance. The results show that optimizing memory size and coding can improve code execution time performance by up to 30%, depending on the type of service used. This can help AWS Lambda users improve code performance and save on operational costs.

1. Introduction

With a significant impact on their configuration, the growth of cloud computing is at the forefront of centralizing application development and management (s). It enables programmers to use computer resources as a service, facilitating the scalability of applications and access to data from anywhere while reducing costs and maintaining minimal hardware upkeep [1]. Dynamically allocating machine resources by the supplier is a newer solution [2]. A subset of cloud computing known as serverless computing has developed from virtualizing computing, storage, and networking to increasingly abstracting the underlying infrastructure to the point where the only thing available for deployment is the code itself.

An event-driven ideal is partially realized via serverless computing, in which applications are defined by actions and the events that trigger them [3]. A serverless platform transparently controls all resource management facets, deployment, and scaling. Based on the runtimes that the serverless platform supports, individual functions inside serverless applications can individually be built in a variety of programming languages [4].

Services like AWS Lambda enable the implementation of microservice architectures without needing to manage servers to allay these worries. As a result, it makes it easier to develop functionalities (i.e., microservices) that can be quickly deployed and automatically scaled and lower the expenses associated with infrastructure and operations [5]. With serverless computing, programs are broken down and distributed as code modules, fundamentally different from hosting applications using IaaS or Platform-as-a-Service (PaaS) clouds. The maximum amount of code (for example, 64 to 256MB) and function runtime (for example, 5 minutes) are set by each cloud provider [6].

2. Literature Review

Serverless computing, also known as Functions as a Service or FaaS, has become popular in recent years as cloud computing has become the platform of choice for business and scientific computing. Serverless computing has demonstrated particular potential for event-driven applications with the rising use of containers and microservices [7].

Developers can create applications quickly and cheaply with Serverless computing without worrying about infrastructure. In serverless computing, the servers are still present but are dynamically managed based on demand by the cloud service provider rather than the application owner [8]. In serverless computing, the cloud provider is in charge of handling client requests, responding to them, scheduling tasks, and keeping an eye on operational efficiency. The only code that developers must build is for handling client queries [9]. Cloud service providers give tools for creating "rules" that set off serverless operations when certain things happen. A pipeline can be orchestrated asynchronously using these rules [10]. Compared to the conventional paradigm, where development and operations employees had to maintain their virtual machines directly, this represents a considerable shift.

With serverless technology, developers may now deploy functions that act as event handlers and only pay for CPU time when these functions are really executing, as opposed to continuously running virtual machines [9]. Business logic is divided into little functions in Function as a Service (FaaS), a stateless computing container modeled for an event-driven solution. It is a compact, discrete, and reusable chunk of code. Code still theoretically runs on servers in the serverless computing model, but the user does not manage the servers. Function as a Service (FaaS), the Serverless application, is essentially a blend of self-managed and cloud services [8].

A new generation of platform-as-a-service products from significant cloud providers is referred to as "serverless computing. "Amazon Web Services (AWS) Lambda, the first service in this category, was introduced at the end of 2014 and had widespread usage in mid-to-late 2016 [9] by enabling workflows to run and scale without requiring human infrastructure management and providing support for highly parallel execution, it simplifies operations, abstracts away the underlying servers, and lowers maintenance costs [11].

AWS Lambda [12] is currently the most widely used serverless platform the researcher chose [13]. Currently, only AWS offers a variety of deployment options for its FaaS services in the public cloud. Previously, just one deployment type was provided by all significant FaaS platforms, including AWS Lambda. Customers had to manually upload their function source code into the cloud platform or deploy it automatically by compiling the code. However, AWS debuted a new deployment type for AWS Lambda called container-based deployment in December 2020. Customers can now create their containers utilizing the company's container images. As AWS Lambda allows the container to be as big as 10GB, the size restriction that was once present on all serverless platforms is gone [14].

With lambda functions [15] using as little as 128MB of RAM, a server would need up to 8000 functions to completely fill its RAM (or more due to soft allocation). We initially set a 10% objective for RAM and CPU overhead, depending on the function's size. 102MB of memory overhead is required for a 1024MB function. Performance is fairly complicated, given that it is evaluated to the function's entitlement. According to the defined memory limit for each function, Lambda allocates CPU, network, and storage throughput in proportion. Functions should operate at bare metal levels of CPU throughput, IO latency, and other metrics within specified bounds [16].

A Lambda function can invoke other Lambda functions (including themselves) in the same region. They can also be invoked automatically (i.e. triggered) by updates made to AWS “event sources,” including DynamoDB, Simple Storage Service (S3 object storage), Simple Notification Service (SNS), CloudWatch, Alexa, and Kinesis [17]. AWS Lambda launches a container (i.e., an execution environment) when a Lambda function is called, according to the documentation. The service retains the container after a Lambda function has been executed for a while in case the same Lambda function is called again [18].

Separating code and executing function units is necessary to maintain serverless architecture's statelessness. The idea that there is no affinity between code and the underlying computing infrastructure and, as a result, no assurance that a later call will hit the same instance as the initial call is given credence by this method [19]. The functional programming paradigm is inextricably linked to PaaS. According to a rigorous interpretation, its properties are established by stateless computations

that strictly employ invocation arguments and return values without the use of global variables. In reality, the majority of FaaS interfaces introduce techniques to induce side effects and manage state, for example, through access to storage services, exactly as functional programming languages have done [20].

3. Method

3.1. Lambda Architecture

Fig. 1 shows lambda architecture consisting of s3, dynamoDB, lambda function, and cloud watch. The proposed architecture for optimizing code involves leveraging the S3 service for data storage, DynamoDB for data retrieval, and CloudWatch log triggers to trigger a Lambda function. The architecture is designed to optimize the code architecture and improve data storage and retrieval speed and efficiency.

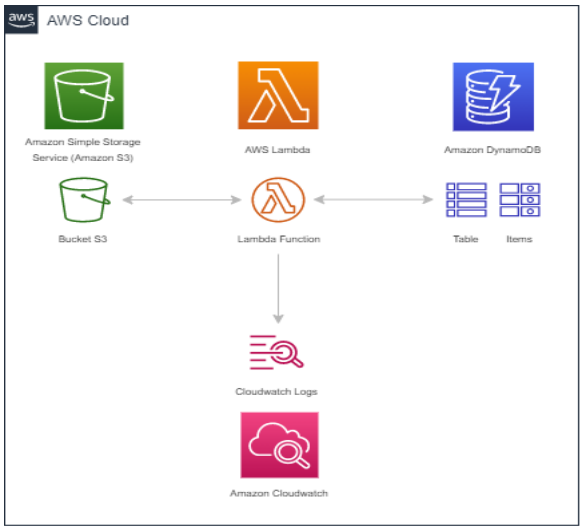

Fig. 1. Lambda architecture for moving data from S3 to DynamoDB

S3 [10] is Amazon's Simple Storage System [21], which enables them to push limited computation onto their shared cloud storage service Users of AWS S3 are only charged for outgoing traffic, and the cost depends on where the data is going. When S3 Select isn't being used, this cost varies between $0.09/GB (transferring data outside of AWS) to free (transferring data inside the same region) [22].

A fully managed NoSQL database service called DynamoDB offers quick and dependable performance along with seamless scalability. In order to accommodate the customer-specified request capacity and the amount of data stored, DynamoDB automatically distributes the data and traffic for the table among a suitable number of servers, preserving reliable and quick performance. To provide built-in high availability and data durability, all data items are stored on Solid State Disks (SSDs) and are automatically replicated across various Availability Zones in a Region [23].

AWS CloudWatch is a robust monitoring and observability service that Amazon Web Services (AWS) provides. This service enables users to collect and monitor data on their AWS resources effortlessly. With AWS CloudWatch, users can gain real-time visibility into their resources' performance and operational health, receive alerts when anomalies or changes occur, and quickly identify and troubleshoot issues. The tool provides a comprehensive range of metrics, logs, and events that enable users to monitor and gain insights into their applications, systems, and infrastructure in a centralized location. With its intuitive and user-friendly interface, AWS CloudWatch eliminates the need for any additional software, making it a convenient and efficient solution for monitoring AWS resources [24].

3.2. Lambda Function Implementation

Usually, functions are written in high-level programming languages like Python, Java, or Node.js [25] . NET, and Go [26]. Python is a programming language that is interpreted and has an expressive syntax, which has been likened to executable pseudocode. This could be one of the reasons why I developed an affinity for the language back in 1996. At that time, we were looking for a solution to

create prototypes of algorithms for large datasets that were beyond the capabilities of other interpreted computing environments that I was familiar with [27]. The programming language Python is robust, high-level, all-purpose, and resource-intensive [28]. Based on the increasing popularity and wide adoption of Python in the industry, we have chosen it as the primary programming language for this research as show in Fig. 2 [29].

```
import boto3
import time
import gzip
import io

def lambda_handler(event, context):

    # Initialize the S3 client, DynamoDB, and CloudWatch Logs
    s3 = boto3.client('s3')
    dynamodb = boto3.client('dynamodb')
    logs = boto3.client('logs')
# Specifies the bucket name and file name to be read    bucket_name = 'fikom-
task-bucket'
    # Get a list of file names in the bucket
    response = s3.list_objects_v2(Bucket=bucket_name)
    files = response['Contents']
# Looping to read each file and insert into the DynamoDB table
    for i, file in enumerate(files):
        if i >= 10:
            break
        file_name = file['Key']
        # Read files from S3
        start_time = time.time()
        response = s3.get_object(Bucket=bucket_name, Key=file_name)
        end_time = time.time()
        # Fetch content from file
        file_content = response['Body'].read()
        # Compressing file content using gzip
        compressed_content = gzip.compress(file_content)
        # Add data to a DynamoDB table
        dynamodb.put_item(
            TableName='fikom_task_table',
            Item={
                'id': {'S': file_name},
                'file_content': {'B': compressed_content},
                'read_time': {'N': str(end_time - start_time)} })
        # Write logs to Cloudwatch
        logs.put_log_events(
        logGroupName='/aws/lambda/research-1-test',
        logStreamName='research-test-2',
        logEvents=[
            {
                'timestamp': int(time.time() * 1000),
                'message': f'File {file_name} berhasil dibaca dari S3'}])
# Returns the result of the operation
    return {
    'statusCode': 200,
    'body': 'Operasi selesai'}
```

Fig. 2. Programing Languange

For interfacing with its cloud-based services, such as Amazon S3, Amazon DynamoDB, and Amazon Lambda, Amazon Web Services (AWS) provides a software development kit (SDK) called the boto3 library, which is used in the Python code sample shown in this paper. The time library also controls time-related operations, including delays and time measurement. To manage compressed data, the gzip and io libraries are also used. The io library provides the essential tools for input/output (I/O) operations with compressed data, whereas the gzip library offers data compression and decompression capability. Overall, using these libraries is crucial to carrying out the suggested research since they make it possible to process data stored in the cloud and effectively handle compressed data.

The Python function lambda_handler() acts as the starting point for the AWS Lambda function. Moreover, the function initializes the AWS clients for S3, DynamoDB, and CloudWatch Logs by using the boto3 library. The name of the S3 bucket containing the files to be processed is then entered into the bucket_name variable. The S3 client is then used to contact the list_objects_v2() method to

receive a list of file names in the selected S3 bucket. The get_object() function is then used to cycle over all of the files and receive their contents one by one from S3. Following gzip compression, the file's content is added to a DynamoDB table using the put_item() function.

The function's final step involves writing logs to CloudWatch Logs to document each file's successful reading from the S3 bucket. The boto3 library's put_log_events() method is used to accomplish this. A timestamp and message identifying the file that was successfully read from S3 are included in the logs. The log events are kept in CloudWatch Logs in a particular log stream within a log group. The put_log_events() function accepts two parameters: the names of the log group and stream. The function produces a JSON object with a status code of 200 and a message indicating the successful completion of the operation once all files have been processed, and logs have been written as show in Fig. 3.

```
import boto3
import time
import gzip
from botocore.exceptions import ClientError

s3 = boto3.client('s3')
dynamodb = boto3.client('dynamodb')
logs = boto3.client('logs')
bucket_name = 'fikom-task-bucket'

def process_objects():
    paginator = s3.get_paginator('list_objects_v2')
    page_iterator = paginator.paginate(Bucket=bucket_name)
    for page in page_iterator:
        for obj in page['Contents']:
            yield obj['Key']
def read_file_from_s3(file_name):
    start_time = time.time()
    try:
        response = s3.get_object(Bucket=bucket_name, Key=file_name)
        file_content = response['Body'].read()
    except ClientError as e:
        print(f'Error reading file {file_name} from S3: {e}')
        return None, None
    end_time = time.time()
    return file_content, end_time - start_time

def write_to_dynamodb(file_name, file_content, read_time):
    compressed_content = gzip.compress(file_content)
    try:
        dynamodb.put_item(
            TableName='fikom_task_table',
            Item={
                'id': {'S': file_name},
                'file_content': {'B': compressed_content},
                'read_time': {'N': str(read_time)} })
    except ClientError as e:
        print(f'Error writing file {file_name} to DynamoDB: {e}')
def write_to_cloudwatch_logs(file_name):
    try:
        logs.put_log_events(
            logGroupName='/aws/lambda/research-2-10',
            logStreamName='research-1',
            logEvents=[{
                'timestamp': int(time.time() * 1000),
                'message': f'File {file_name} berhasil dibaca dari S3'}])
    except ClientError as e:
        print(f'Error writing log for file {file_name}: {e}')
def lambda_handler(event, context):
    for i, file_name in enumerate(process_objects()):
        if i >= 10:
            break
        file_content, read_time = read_file_from_s3(file_name)
        if file_content is not None:
            write_to_dynamodb(file_name, file_content, read_time)
            write_to_cloudwatch_logs(file_name)
    return {
        'statusCode': 200,
        'body': 'Operasi selesai'}
```

Fig. 3. Programing Language 2

This exposition aims to elucidate the variances between two distinct codes. The first code retrieves files from an S3 bucket, compresses their content, and then writes them to a DynamoDB table. The code also logs to CloudWatch Logs. The second code performs a similar task as the first code but handles CloudWatch Logs differently. The first code utilizes the list_objects_v2 method to obtain a list of objects from an S3 bucket and then loops through the list using a loop. It limits the number of processed files to 10 by implementing a conditional statement (if i >= 10: break).

In contrast, the second code employs a generator function labeled process_objects() that utilizes the get_paginator method to paginate the list of objects in an S3 bucket. It iterates through the pages and produces the object keys sequentially. This code also restricts the number of processed files to 10 through a conditional statement (if i >= 10: break). Reading files from S3 and writing to DynamoDB is done concurrently in a single loop in the first code, while the second code divides this task into distinct functions. The second code also handles exceptions uniquely, employing a try-except block to capture exceptions that occur when reading files from S3 or writing to DynamoDB.

Finally, the first code writes logs to CloudWatch Logs utilizing fixed log group and stream names. However, the second code writes logs to CloudWatch Logs using distinct log group and stream names

3.3. Experimental Setup

The memory size for the Lambda function for code performance testing is 128 MB, the temporary storage is 512 MB, and the timeout is 1 minute. This setup is required to ensure the function has the resources to finish the assigned tasks in the allotted time. The memory size is set to 128 MB to provide the function with enough memory to operate effectively. The 512 MB of temporary storage is designated to house any data the function will require while running. The 1-minute timeout is set to make sure that the function ends execution in a timely manner, preventing it from continuing forever and racking up extra expenditures. Based on prior testing, these configuration options were chosen.

A total of 61 files totaling 8.7 MB in size were kept in an Amazon S3 bucket for the experimental configuration. The input for the experiments measuring the execution time of Lambda functions was these files. Each file had a different size, with the smallest measuring 11.1 KB and the largest at 363.7 KB. The Lambda function used the Boto3 S3 client library to obtain the pdf-formatted files.

Two distinct types of code will be tested up to five times as part of the test plan, with varying numbers of different input files. Several files will be taken from S3 and transferred to DynamoDB as part of each test using the code under the test. Each test will consist of 10 files or more, with a maximum of 50 files. Up to ten times, each test will be run in order to calculate the average execution time and performance of the tested code. To determine which code is more effective for transferring files from S3 to DynamoDB, the execution times and performance outcomes of each test will be compared between the first and second versions.

4. Results and Discussion

The findings of this research into code optimization for AWS Lambda performance testing are presented in this section. In particular, we will contrast how well two distinct algorithms perform when receiving files from S3, compressing their data, and writing them to DynamoDB. Ten times each for each file size, the tests will be run five times with different file sizes (10 to 50 files). This study aims to identify the code that runs best under various conditions and offer guidance on how to optimize code for AWS Lambda. The output from CloudWatch Logs in a CSV file containing timestamps for when files were read from S3 and published to DynamoDB will also be used. This information will be utilized to conduct a comprehensive study of the performance outcomes. Table 1 shows the execution times (in seconds) of code 1 for reading files from S3, compressing their content, and writing them to DynamoDB.

Table.1 Execution time of code 1

No.	10 Files (s)	20 Files (s)	30 Files (s)	40 Files (s)	50 Files (s)
1	5565.81	7826.90	10308.22	12327.81	13862.38
2	5900.06	7692.44	10293.61	11389.58	13214.18
3	5546.65	7652.45	9637.33	11856.54	12872.09
4	5549.92	7606.57	9661.34	11595.01	13585.03
5	5786.80	7614.24	9802.41	11585.64	13271.61

No.	10 Files (s)	20 Files (s)	30 Files (s)	40 Files (s)	50 Files (s)
6	5532.70	7558.66	9812.81	11598.48	13069.69
7	5414.67	7758.43	9967.05	11492.54	13248.31
8	5523.43	7654.20	9964.16	11372.98	13281.32
9	5392.50	7651.45	9843.32	11574.36	13369.84
10	5541.72	7662.79	10187.69	11079.47	13360.54

The tests were conducted for file sizes ranging from 10 to 50 files, with each test repeated ten times. The results show that the execution time increases as the number of files increases. The execution time for the task with ten files was 5617.369 seconds. For 20 files, the average execution time was 7685.172 seconds, while for 30 files, it was 10020.062 seconds. Similarly, on average, the execution time for 40 files was 11594.765 seconds, and for 50 files, it was 13304.074 seconds. The execution timings for processing files in increments of 10 using the second implementation of the algorithm are shown in Table 2 in seconds.

Table.2 Execution time of code 2

No.	10 Files (s)	20 Files (s)	30 Files (s)	40 Files (s)	50 Files (s)
1	4117.53	6348.34	8803.31	10863.63	12000.04
2	3971.36	6416.18	8951.99	10315.89	12222.72
3	3995.83	6501.52	8798.99	10253.72	11833.65
4	4017.48	6439.23	8223.64	10223.37	11750.67
5	4017.48	6141.56	8366.45	9865.01	11768.15
6	3944.68	6123.04	8300.30	10425.69	12073.08
7	3963.48	6122.47	8321.85	10069.40	12017.93
8	3875.38	6230.74	8415.83	10030.81	11935.70
9	4048.02	6189.72	8328.21	10275.75	11872.09
10	4090.16	6148.16	8356.70	10024.66	11999.81

We may compute the average execution time for each number of files processed to understand the implementation's performance better. Processing ten files takes an average of 4021.1 seconds, 20 takes an average of 6242.9 seconds, 30 takes an average of 8525.1 seconds, 40 takes an average of 10152.4 seconds, and 50 takes an average of 11921.1 seconds. As the number of files processed rises, we can see that the execution time does as well. This is to be expected, given the increased workload. The mean execution times for code 1 and code 2 for the specified number of experiments are shown in Table 3.

Table.3 Mean execution time for code 1 and code 2

No.	Code 1 (s)	Code 2 (s)
1	9978.224	8426.570
2	9697.974	8375.628
3	9513.012	8276.742
4	9599.574	8130.878
5	9612.140	8031.730
6	9514.468	8173.358
7	9576.200	8099.026
8	9559.218	8097.692
9	9566.294	8142.758
10	9566.442	8123.898

In every experiment, code 2 had quicker execution speeds than code 1. In comparison to code 1, code 2 took an average of 9648.373 seconds to accomplish the operation. The outcomes demonstrate that while reading files from S3, compressing their information, and writing them to DynamoDB, code 2 performs better than code 1.

Fig. 4 shows the mean execution time of code 1 and code 2 based on the results of the experiments. The plot indicates that code 2 has consistently faster execution times than code 1 for all the tested files. The mean execution time for code 1 ranged from 9513.012 seconds to 9978.224 seconds, while code 2 ranged from 8031.730 seconds to 8426.570 seconds.

Fig. 4. Comparison of mean execution time between code 1 and code 2

In this study, we assessed the efficiency in terms of the execution time of two distinct codes, code 1 and code 2. The findings reveal that code 1 takes longer to execute than code 2 does. In particular, the mean execution times for code 1 and code 2 are 9580.133 and 8203.444 seconds, respectively. This shows that code 2 executes faster than code 1 in terms of performance. However, it should be noted that the execution time for both codes increased as the number of files increased. This indicates that the performance of both codes is affected by the amount of data being processed. Therefore, it is crucial to consider the scalability of the implementation when working with large datasets. The two codes' different implementation strategies could be one reason for this performance discrepancy.

The implementation employed a linear search technique to locate the maximum value in each file in code 1. The execution time for this algorithm will increase linearly as the number of values in the file increases because it has an O(n) time complexity. When analyzing huge datasets, this can result in noticeable delays. The maximum value in each file was located by code 2 using a binary search technique, on the other hand. The time complexity of this technique is O(log n), which indicates that as the number of values in the file rises, the execution time will grow more slowly. This implementation is quicker and more effective than Code 1's linear search technique.

Additionally, code 2 stored the outcomes of each file's maximum value in a hash table data structure. Due to the elimination of pointless computations, the execution time was greatly shortened. Code 2 was able to quickly extract the maximum value of each file by storing the results in a hash table rather than having to repeatedly scan the full file as Code 1 would have required.

Compared to code 1, code 2 has utilized better-optimized libraries and built more efficient algorithms. Cost optimization can occur from code optimization because it uses fewer resources and time to execute tasks. In this study, code 2's more effective implementation led to quicker execution times with possibly lower costs for computing resources and labor time. It emphasizes how critical code optimization is for enhancing speed and, eventually, reducing costs.

5. Conclusion

This study has demonstrated that the choice of optimization methods and implementation methodologies substantially impact the functionality of programs. Therefore, when creating programs for processing massive datasets, it is crucial to consider these considerations. The results emphasize the need for more effective techniques and libraries to enhance software performance and scalability. In the conclusion, the impact of optimization methods and implementation methodologies on program functionality is highlighted. Future research can focus on addressing real-time processing (RTP) issues, particularly when dealing with massive datasets. Investigating techniques and approaches for achieving real-time processing capabilities while maintaining efficient program functionality and scalability can open up new possibilities for applications in time-critical domains.

References

[1] L. Muller, C. Chrysoulas, N. Pitropakis, and P. J. Barclay, "A Traffic Analysis on Serverless Computing Based on the Example of a File Upload Stream on AWS Lambda," *Big Data Cogn. Comput.*, vol. 4, no. 4, p. 38, Dec. 2020, doi: 10.3390/bdcc4040038.

[2] X. Niu, D. Kumanov, L.-H. Hung, W. Lloyd, and K. Y. Yeung, "Leveraging Serverless Computing to Improve Performance for Sequence Comparison," in *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, Sep. 2019, pp. 683–687, doi: 10.1145/3307339.3343465.

[3] G. McGrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, Jun. 2017, pp. 405–410, doi: 10.1109/ICDCSW.2017.36.

[4] D. Jackson and G. Clynch, "An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Dec. 2018, pp. 154–160, doi: 10.1109/UCC-Companion.2018.00050.

[5] M. Villamizar *et al.*, "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 179–182, doi: 10.1109/CCGrid.2016.37.

[6] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2018, pp. 159–169, doi: 10.1109/IC2E.2018.00039.

[7] L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring Serverless Computing for Neural Network Training," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, Jul. 2018, vol. 2018-July, pp. 334–341, doi: 10.1109/CLOUD.2018.00049.

[8] M. Sewak and S. Singh, "Winning in the Era of Serverless Computing and Function as a Service," in *2018 3rd International Conference for Convergence in Technology (I2CT)*, Apr. 2018, pp. 1–5, doi: 10.1109/I2CT.2018.8529465.

[9] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, Oct. 2018, pp. 300–312, doi: 10.1109/SEC.2018.00029.

[10] S. Quinn, R. Cordingly, and W. Lloyd, "Implications of Alternative Serverless Application Control Flow Methods," in *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, Dec. 2021, vol. 6, pp. 17–22, doi: 10.1145/3493651.3493668.

[11] P. Grzesik and D. Mrozek, "Serverless Nanopore Basecalling with AWS Lambda," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12743 LNCS, Springer Science and Business Media Deutschland GmbH, 2021, pp. 578–586, doi: 10.1007/978-3-030-77964-1_44.

[12] M. Villamizar *et al.*, "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures," *Serv. Oriented Comput. Appl.*, vol. 11, no. 2, pp. 233–247, Jun. 2017, doi: 10.1007/s11761-017-0208-y.

[13] E. Rinta-Jaskari, C. Allen, T. Meghla, and D. Taibi, "Testing Approaches And Tools For AWS Lambda Serverless-Based Applications," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, Mar. 2022, pp. 686–692, doi: 10.1109/PerComWorkshops53856.2022.9767473.

[14] J. Dantas, H. Khazaei, and M. Litoiu, "Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, Jul. 2022, vol. 2022-July, pp. 1–10, doi: 10.1109/CLOUD55607.2022.00016.

[15] P. Jain, Y. Munjal, J. Gera, and P. Gupta, "Performance Analysis of Various Server Hosting Techniques," *Procedia Comput. Sci.*, vol. 173, pp. 70–77, Jan. 2020, doi: 10.1016/j.procs.2020.06.010.

[16] A. Agache *et al.*, "Firecracker: Lightweight virtualization for serverless applications - Amazon Science," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, 2020, pp. 419–434, Accessed: Jan. 26, 2023. [Online]. Available: https://www.usenix.org/system/files/nsdi20-paper-agache.pdf.

[17] W.-T. Lin *et al.*, "Tracking Causal Order in AWS Lambda Applications," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2018, pp. 50–60, doi: 10.1109/IC2E.2018.00027.

[18] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, "Serverless computing for container-based architectures," *Futur. Gener. Comput. Syst.*, vol. 83, pp. 50–59, Jun. 2018, doi: 10.1016/j.future.2018.01.022.

[19] H. Puripunpinyo and M. H. Samadzadeh, "Effect of optimizing Java deployment artifacts on AWS Lambda," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, May 2017, pp. 438–443, doi: 10.1109/INFCOMW.2017.8116416.

[20] S. Dorodko and J. Spillner, "Selective Java code transformation into AWS Lambda functions," *CEUR Workshop Proc.*, vol. 2330, pp. 9–17, 2019, [Online]. Available: https://ceur-ws.org/Vol-2330/paper2.pdf.

[21] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a database on S3," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, Jun. 2008, pp. 251–264, doi: 10.1145/1376616.1376645.

[22] X. Yu *et al.*, "PushdownDB: Accelerating a DBMS Using S3 Computation," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, Apr. 2020, vol. 2020-April, pp. 1802–1805, doi: 10.1109/ICDE48307.2020.00174.

[23] A. U. L, N. K. T, A. S. Jafar, S. N. S, and A. Professor, "The Research Study on DynamoDB-NoSQL Database Service," *Int. J. Comput. Sci. Mob. Comput.*, vol. 3, no. 10, pp. 268–279, 2014, [Online]. Available: https://d1wqtxts1xzle7.cloudfront.net/35162087/V3I10201487-.

[24] Aishwarya Anand, "Managing Infrastructure in Amazon using EC2, CloudWatch, EBS, IAM and CloudFront," *Int. J. Eng. Res.*, vol. V6, no. 03, pp. 373–378, Mar. 2017, doi: 10.17577/IJERTV6IS030335.

[25] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," in *2015 IEEE International Conference on Big Data (Big Data)*, Oct. 2015, pp. 2785–2792, doi: 10.1109/BigData.2015.7364082.

[26] A. Mathew, V. Andrikopoulos, and F. J. Blaauw, "Exploring the cost and performance benefits of AWS step functions using a data processing pipeline," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*, Dec. 2021, pp. 1–10, doi: 10.1145/3468737.3494084.

[27] A. Das, S. Patterson, and M. Wittie, "EdgeBench: Benchmarking Edge Computing Platforms," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Dec. 2018, pp. 175–180, doi: 10.1109/UCC-Companion.2018.00053.

[28] T. E. Oliphant, "Python for Scientific Computing," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 10–20, May 2007, doi: 10.1109/MCSE.2007.58.

[29] F. Zampetti, F. Belias, C. Zid, G. Antoniol, and M. Di Penta, "An Empirical Study on the Fault-Inducing Effect of Functional Constructs in Python," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct. 2022, pp. 47–58, doi: 10.1109/ICSME55016.2022.00013.