A Study on Cost Optimization in FaaS Computing with The Most Favorable Resource Size

Bibus Poudel[a], Saroj Pandey[b]
[b]*sarojpandey@kcc.edu.np*
[a b]*Kantipur City College, Kathmandu, 44600, Nepal*

Abstract
Cloud computing has been overgrown, leading to the transformation of workloads from on-premises server rooms into public cloud environments. This change in the workflow has benefited developers by shifting the responsibility of server and infrastructure management to a cloud service provider. Along with these services, a new emerging paradigm for cloud computing is introduced called serverless computing, where the user does not need to manage any server infrastructure. FaaS is a serverless computing model, that allows developers to deploy individual functions to the cloud, where cloud providers take care of resource management tasks such as resource provisioning, deployment, and auto-scaling. This paper focuses on choosing the most favorable resource size for the AWS lambda function for optimal cost by using a reference table. It allows developers to take a reference while choosing the memory size for the lambda function. The output was evaluated on 10 different serverless functions on AWS. Initially, these functions were set up using the default memory size, and during the evaluation, the referred memory size was chosen. Based on the evaluation of test functions, it selects the optimal memory size for 80.0% of the serverless functions, which results in an average speed up of 142.26% while also decreasing average costs by 10.57%.

Keywords: function as a service; microservices; lambda function optimization, dynamic profiling; event-driven computing; lambda cost-performance tradeoff

1. INTRODUCTION

The traditional approach to application development requires upfront investment in physical servers, leading to high costs and slow scaling. Cloud computing emerged as a game-changer, offering on-demand access to configurable computing resources over the internet [1]. This shift reduced management burdens and allowed developers to focus on their core competencies. However, serverless computing promises an even more transformative leap. By abstracting away server management entirely, it enables rapid deployment and eliminates the need for infrastructure expertise. [2] This research delves deeper into this emerging paradigm, analyzing how serverless platforms handle management and function placement. Additionally, it explores the motivations driving further research in this area and provides a clear roadmap by outlining its objectives and structure [3].

Serverless computing revolutionizes development by removing server management burdens. Developers deploy code on cloud platforms, leveraging pay-as-you-go pricing, automatic scaling, and infrastructure offloading [9]. This shift comes with operational complexity and new challenges, particularly cost optimization. Choosing the right resource size for each function significantly impacts both cost and performance, necessitating a deep understanding of optimization strategies for serverless architectures. This research explores serverless computing, focusing on its goals, architecture, and benefits. It then focuses on the crucial challenge of cost optimization, examining techniques for optimal resource allocation and cost efficiency. By investigating this innovative paradigm, we aim to empower developers and contribute to the advancement of serverless application development. Since the

early years of computer science and the first networked computers, IT engineers have gained large amounts of valuable experience that has become the base knowledge to develop modern internet infrastructure.

Virtualization, the invisible architect, lays the foundation for the diverse cloud landscapes we navigate today [4][8]. Businesses and developers alike have a spectrum of options, each catering to specific needs. For those wielding the reins, Infrastructure as a Service (IaaS) offers raw computing power, akin to renting building blocks for any software endeavor - think Amazon EC2, Google Compute Engine, or Azure VMs [4]. Containers as a Service (CaaS) delivers pre-packaged applications in containers, ready for deployment with platforms like Docker Swarm or Kubernetes [5]. It's the perfect blend of flexibility and ease. Developers seeking a nurturing haven can turn to Platform as a Service (PaaS), where platforms like Heroku or AWS Elastic Beanstalk provide development tools and libraries, freeing them to focus on crafting beautiful applications. But if simplicity reigns supreme, Software as a Service (SaaS) is your answer [6]. Think Slack, ServiceNow, or Dropbox—ready-to-use applications accessible with a click, leaving maintenance and updates to the cloud providers [7]. Finally, serverless computing emerges as the latest trend, pushing boundaries with its pay-as-you-go model and eliminating server management. It's perfect for event-driven architectures and dynamic scaling, a true testament to the cloud's ever-evolving potential [11].

Function as a Service (FaaS) platforms run small units of code executed by a response to an event. The event can be, for example, an HTTP request, an operation in a database, or a message in a message queue. Users deploy code to FaaS platforms, and the code is executed on-demand based on the events. The service provider handles scaling and infrastructure, so the service appears serverless to the user [15]. The functions are executed in stateless ephemeral containers, where the lifetime of a single function execution environment can be in a matter of milliseconds. FaaS functions are stateless because it cannot be guaranteed that any state information is 3 preserved between function invocations. The service provider decides how long is the lifetime of a single function instance [15].

1.1 Motivation

The relationship between the memory size of a serverless function, the cost per function execution, and the function execution time is quite counter-intuitive. A common assumption is that a higher memory size results in a faster execution at a higher price, since the allocated CPU, I/O, network, etc. capacity scales linearly with the selected memory size [9, 17]. However, this is not the case due to the pricing scheme most cloud providers employ, where the cost of an execution is calculated based on the consumed GB-s of memory, that is, the execution time multiplied by memory size. Additionally, cloud providers often include a static overhead charge per execution, which tends to be negligible compared to the consumed GB/s charge. For a more in-depth discussion of the pricing model behind serverless functions, it is referred to [18]. As an example, a function that runs on AWS for three seconds with a memory size of 512 MB would cost:

3s * 0.5 * 0.00001667$ + 0.0000002$ = 0.0000252$

Here, 0.00001667$ is the AWS-specific price per consumed GB-s and 0.0000002$ the static overhead charge (0.7% of the total execution cost). Increasing the memory size increases the cost per second, but also decreases the execution time more resources are allocated. As each function’s execution time scales differently with additional resources, every function has a unique cost/performance trade-off.

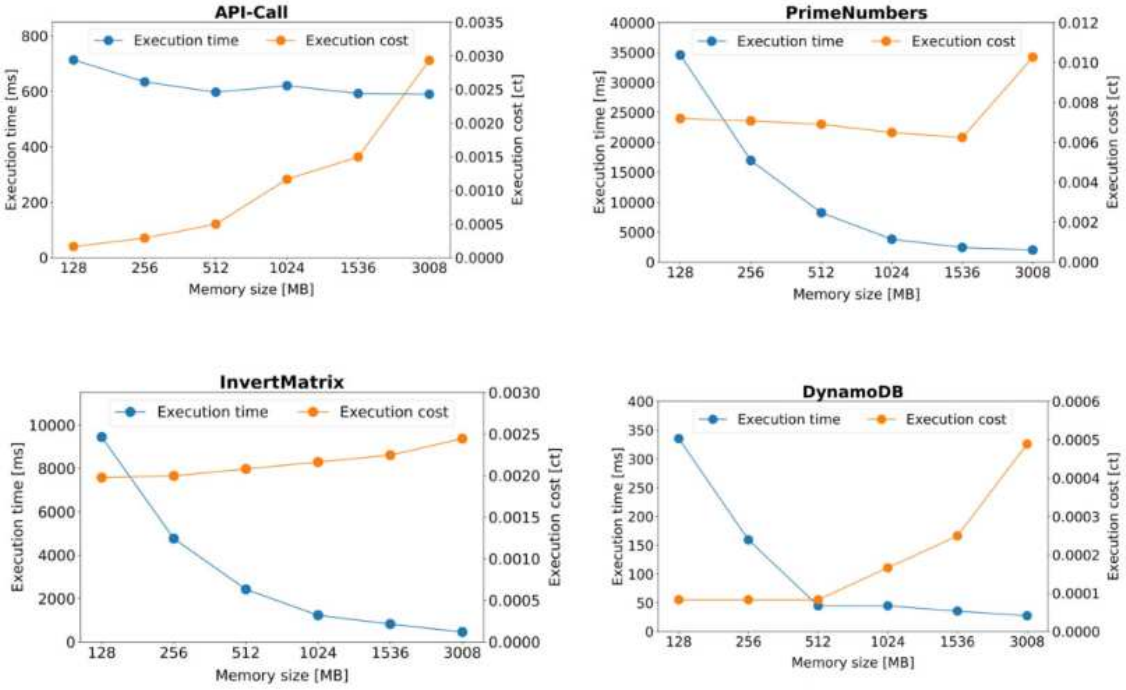

Fig. 1: The mean execution time and cost of four serverless functions (adapted from [18])

Fig. 1 exemplifies how the execution time and cost per execution vary for four different functions with different memory sizes based on data from [11]. The function InvertMatrix creates and inverts a random matrix. Here, we can see that increasing the memory size from 128MB to 256MB decreases the execution time by 49.6%, with only a 1% increase in cost. For larger memory sizes, the execution time still decreases almost linearly. The second function, PrimeNumbers, calculates the first million prime numbers a thousand times, which is another CPU-intensive task. Interestingly, the execution time of this function scales super-linearly with increased memory sizes up to 2048MB, which results in a 92.9% faster execution with simultaneously 13.3% reduced costs. Using a memory size of 3008MB further speeds up the execution time, but it increases the execution cost. The third function, DynamoDB, executes three queries against a DynamoDB table, which is a serverless database. Here, the execution time decreases roughly linearly from 128MB to 512MB, resulting in an 86.6% decreased execution time at a similar cost. However, further increasing the memory only slightly reduces the execution time while increasing costs by 587.5%. Lastly, the API-Call function calls an external API. Here, increasing the memory has minimal impact on the execution time and only increases the cost per execution. Based on these results, we can conclude that: i) the impact of memory size configurations on execution time differs from function to function, ii) predicting the execution time for a memory size is challenging, as even two seemingly CPU-intensive and two network-intensive functions behave differently, and iii) selecting an appropriate memory size is important as it can drastically improve performance at a similar or reduced cost.

1.2 Objectives

- To determine the most favorable memory size for lambda function to optimize cost and performance.
- To compare the impact of chosen memory size in lambda functions.

2. METHODOLOGY

The research attempts to address the problem of developers choosing the right memory size for the serverless function. This introduced an approach to choosing the most favorable memory size for serverless functions using a reference table with differing cost and time tradeoffs.

2.1 Workflow

First, a large set of monitored data of serverless functions was collected from various open-source tools like Registry of Open Data on AWS (RODA), UCI Machine Learning Repository, DataDog, etc. Secondly, these data were filtered, and the most efficient data was kept. Afterward, data patterns were analyzed, and the execution time and cost of functions with different memory sizes were calculated. To solve the cost and time, multi-objective optimization problem, a tradeoff factor was defined that combined the objectives into a single score.

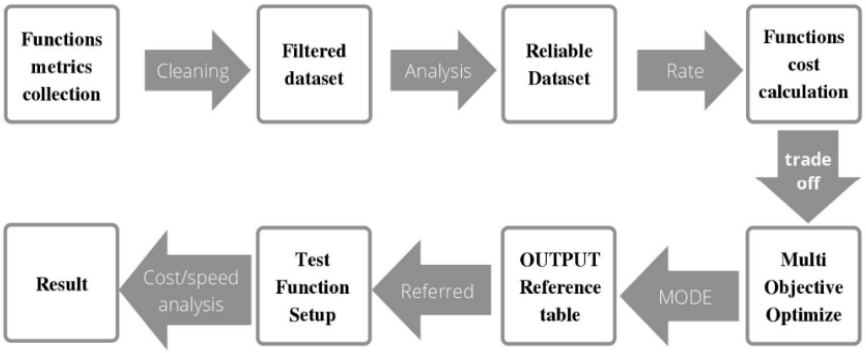

Fig. 2: Workflow overview of the proposed approach

Afterward, the memory size with the minimum total score was taken as the most favorable one. The test for memory size with a minimum score was conducted on 12000 lambda functions, and the most reiterated memory size was classified based on three tradeoff factors along with a reference table created as the final output.

2.2 Data collection

To derive how different memory sizes influence execution time and cost, a large dataset covering various types of functions is required. So secondary data is preferred, as primary data would be costly (in terms of both money and time) and could be obtained only after running functions on the AWS cloud. With the help of the Cloud Watch repository, Registry of Open Data on AWS (RODA), UCI Machine Learning Repository, datadog, and codeocean 12893 lambda functions, their resource consumption metrics were collected. These function segments are the most common tasks in serverless applications.

2.3 Dataset cleaning

All the function data collected from various sources was not of good quality. Some had old configuration settings, pricing models, and different vendors' data and were incomplete as well. So, data cleaning was essential to the research. Data cleaning is the process of detecting and correcting corrupt or inaccurate records from a record set, table, or database. It refers to identifying incomplete, incorrect, inaccurate, or irrelevant parts of the data and then replacing, modifying, or deleting the dirty or coarse data. Pandas was used in the data cleaning process, which is the popular Python library that is mainly used for data processing purposes like cleaning, manipulation, and analysis. Pandas stands for “Python Data Analysis Library.”. After removing the unnecessary data, only 12,000 lambda functions were considered for further analysis.

2.4 Dataset Analysis

To analyze the internal consistency and reliability of the dataset collected after filtering, Cronbach's alpha is used. Cronbach's alpha (〈), also known as tau-equivalent reliability or coefficient alpha, is a reliability coefficient that provides a method of measuring the internal consistency of tests. First, a 10% sample population was taken from a total population of 12,000 lambda functions. After that, the Anova two-factor without replication is calculated for 1200 samples in an Excel sheet. A two-factor ANOVA, also known as factorial analysis, is an extension of the one-way analysis of variance. In a two-factor analysis, there are two variables, rather than one, as in a single-factor analysis. Based on the assumption that both variables, or factors, affect the dependent variable, each factor contains two or more classes, and the degree of freedom for each variable is one less than the number of levels [12] [13]. After the ANOVA test, the sum of squares (SS), degree of freedom (df), mean square (MS), F, P-value, and Fcrit (Fcritical) are obtained for rows and columns. To calculate Cronbach's alpha (〈), the MS of error and row ratio are subtracted by one.

α = 1-(error ms/ rows ms)

The α value obtained was 0.74, The general rule of thumb for Cronbach's alpha is that a value of 0.70 and above is good, 0.80 and above is better, and 0.90 and above is best, which interprets our value to be acceptable.

Table 1: Cronbach's Alpha Anova parameters value

Source of Variance	SS	df	MS	F	P-Value	F crit
Rows	520743592.471963	1200	433952.99	3.84676	8.77995874163018E	1.075233335
Columns	1285584645.88029	5	257116929.2	2279.20	0	2.2155891
Error	676859976.2	6000	112809.99			
Total	2483188215	7205				

Cronbach's Alpha=0.7400409776

2.5 Cost Calculation

With AWS Lambda, we pay only for what we use. Charges are based on the number of requests for our functions and the time it takes for our code to execute. Lambda registers a request each time it starts executing in response to an event notification or invoke call, including test invokes from the console. We are charged for the total number of requests across all our functions. Duration is calculated from the time our code begins executing until it returns or otherwise terminates, rounded up to the nearest 1 ms [10]. The price depends on the amount of memory we allocate to our function. The price for each lambda function is calculated using 4 factors:

i. The number of times each function is executed per month (e.g., 1,000,000 executions/month).
ii. The memory is allocated to the function by application developers. The CPU resources allocated to the function represent an implicit parameter. This parameter value is proportional to the function's allocated memory (i.e., a 256 MB function is automatically allocated twice the CPU speed of a 128 MB function).
iii. The duration is how long the function runs.
iv. The price per 1 GB of memory and 1 second of execution. For AWS Lambda, the price of 1 GB and 1 second is 0.00001667$/GB-s.

The cost for each lambda function is calculated considering a single run. The cost calculation formula for a lambda function is

Total cost = execution time [second] * memory size [GB] * Price per memory

consumed [$] + static overhead cost)

As an example, a function that runs on AWS for three seconds with a memory size of 512 MB would cost:

3 * 0.5 * 0.00001667$ + 0.0000002$ = 0.0000252$

where 0.00001667$ is the AWS-specific price per consumed GB-s and 0.0000002$ is the static overhead charge (0.7% of the total execution cost).

Using the above formula, the cost for 12000 lambda functions was calculated with execution time, price, and memory size as parameters; however, the static overhead cost was not considered.

3. EVALUATION AND RESULTS

To test the reference table, 10 different lambda functions were created on the lambda function console with varying time and space complexity. These functions were written in javascript on the AWS console IDE and used node.js runtime environment architecture x86_64. Index.handler was used as a handler to simulate the real-world backend API services. Each function was passed with JSON data for testing. At first, these functions were configured on the default memory size. The execution time and cost of these functions were taken from Cloud Watch, which is a monitoring and management service that provides data and actionable insights for AWS, hybrid, and on-premises applications and infrastructure resources. Secondly, these functions were reconfigured using the reference table, and the monitored execution time and cost were recorded. To investigate if the predictions are accurate enough to determine the optimal memory size, the author applied the optimization approach using the execution time predictions and compared the default memory size to the optimal memory size determined based on the measured execution time. The author ran the optimization test for three different tradeoff parameters, = 0.75, which prioritizes cost, = 0.5, which shows no preference, and = 0.25, which prioritizes performance. The result obtained was used to measure changes in cost and speed.

3.1 Cost and Speed analysis

The research investigates what the actual benefits of using the memory sizes selected by the approach are, i.e., how much cost can be saved and how much the function execution can be sped up. To quantify these benefits, the author calculates the relative change in cost and execution time between the memory size selected by the approach of 1024 MB, which is the most dominant and favorable memory size for all the tradeoff factors, i.e., 0.75, 0.5, and 0.25. Table 2 shows the average percentage cost savings and execution time speedup obtained by switching to the memory sizes recommended by our approach. In the evaluation of 10 lambda functions, the cost of 8 functions out of 10 was saved, and the function execution time decreased for all 10 functions. Overall, the average function execution speedup for all ten functions was 142.26% and the overall average cost savings was 10.57%. This shows that the speedup is higher than cost savings, which does not show a balance of priorities for both. As cost savings from the memory size optimization are generally lower than execution time speedups, it seems that the tradeoff value is more inclined towards 0.25 in the optimization result. Therefore, the preferred memory size has more advantages over speed. Another observation that can be made from the result of negatively marked functions is that the lesser the difference in execution time between the default and referred memory sizes, the more the optimized result is not obtained. To summarize, this approach referred to memory size functions saving on average 10.57% costs and speeds up the functions by 142.26%. This highlights the importance of selecting an appropriate memory size and, therefore, the benefits of an approach for memory size optimization.

Table 2: Cost and speed analysis of 10 test functions.

Test Functions	Cost savings (%)	Speedup (%)
Factorialofanumber	11.92	150.00
Helloworld	18.77	152.94
FibinacciArray	56.73	167.03
RemoveDuplicateandcountlenth	-55.86	111.11
Sudokusolver	37.71	160.40
Matrixmultiplication	11.92	150.00
Permutationofarray	25.15	155.56
checkvalidparenthesis	36.64	160.00
Sortarrayandfindmaxgap	84.96	175.59
longestpalindrom	-122.21	40.00

3.2 Findings and discussion

Serverless functions automate resource provisioning, deployment, instance management, and auto-scaling. The last resource management task that developers are still in charge of is resource sizing, i.e., selecting how many resources are allocated to each worker instance. This paper introduced an approach to choosing the most favorable memory size for serverless functions using a reference table with differing cost and time tradeoffs. First, a large set of serverless functions monitored data collected from various open-source tools. Secondly, these data were filtered and the most efficient data were kept. Using these, data patterns were analyzed, and then the execution time and cost of functions with different memory sizes were calculated. The task of then selecting the most favorable memory size is a multi-objective optimization problem, which resulted in a Pareto front of optimal memory sizes. To solve this multi-objective optimization problem, a tradeoff factor was defined that combined the objectives into a single score. Afterward, the memory size with the minimum total score was taken as the most favorable one. The test for memory size with the minimum score was conducted on 12000 lambda functions, and the most reiterated memory sizes were classified based on three tradeoff factors: 0.25, 0.5, and 0.75, and a reference table was created as the final output.

In the evaluation, 10 different lambda functions were created with a default memory size at the beginning, and their execution time and cost were measured. After that, these functions were reconfigured using the referred memory size by research and monitored. The approach of this paper selected the optimal memory size for 80.0% of the serverless functions, which results in an average speed up of 142.26% while simultaneously decreasing average costs by 10.57%.

4. CONCLUSION

This study presented an approach to choosing the optimal memory size of serverless functions using a reference table that balances cost and time tradeoffs. Through data collection, filtering, and analysis, a Pareto front of optimal memory sizes was obtained, and a tradeoff factor was defined to combine the objectives into a single score. The test conducted on 12000 lambda functions showed that the approach selected the optimal memory size for 80.0% of the serverless functions, resulting in an average speed up of 142,26% and a decrease in average cost by 10.57%. The developed reference table can help developers select the most optimal memory size for their serverless functions. thereby achieving both cost efficiency and performance improvements. This research contributes a novel approach for selecting the optimal memory size for serverless functions, which can result in significant improvements in performance and cost efficiency.

Acknowledgments

I would like to express my heartfelt gratitude to my supervisor, Sr. Asst. Prof. Saroj Pandey, for his constant encouragement, inspiration, and invaluable knowledge that made this research possible. I am also deeply thankful to the Department of IT at Kantipur City College for their unwavering support and endless encouragement during this study.

References

[1] Adzic G., Chatley R. Serverless Computing: Economic and Architectural Impact. ESEC/FSE 2017 Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Pages 884-889, 2017.

[2] CNCF Serverless Working Group, 2018. CNCF WG-Serverless Whitepaper v1.0. Accessed 4.4.2023. https://github.com/cncf/wg-serverless/raw/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf.

[3] Jonasm E.et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. University of California at Berkeley, Electrical Engineering and Computer Sciences, 2019.

[4] Lunhao V., Aili L. Research on the Application of Virtualization Technology in High-Performance Computing. 2012 IEEE Symposium on Electrical & Electronics Engineering (EEESYM), Pages 386-388, 2012.

[5] Ivanov V. Implementation of DevOps pipeline for Serverless Applications. Master's thesis, Aalto University, School of Science, Espoo, Finland, 2018. Accessed 8.4.2022. https://aaltodoc.aalto.fi/bitstream/handle/123456789/32432/master_Ivanov_Vitalii_2018.pdf

[6] Amazon Web Services. AWS Lambda Limits. Accessed 11.4.2023. https://docs.aws.amazon.com/lambda/latest/dg/limits.html.

[7] Namiot D., Sneps-Sneppe M.. On Micro-services Architecture. International Journal of Open Information Technologies vol. 2, no. 9, pages 24-27, 2014.

[8] Li Z., Kihl M., Lu Q., Andersson J. A. Performance Overhead Comparison between Hypervisor and Container Based Virtualization. (AINA), 2017 IEEE 31st International Conference on. IEEE, pages 955–962, 2017.

[9] Roberts M., 2018. Serverless Architectures. Accessed 8.4.2022. https://martinfowler.com/articles/serverless.html.

[10] Alex Casalboni. 2020. AWS Lambda Power Tuning. https://github.com/alexcasalboni/aws-lambda-power-tuning

[11] Baldini I., et al. Serverless Computing: Current Trends and Open Problems. IBM Research, New York, USA, 2017.

[12] Maarten Speekenbrink, 2023. "Statistics: Data analysis and modeling", accessed 14-27, December 2023.

[13] SA Glantz and BK Slinker. 2021. Primer of Applied Regression & Analysis of Variance, ed. McGraw-Hill, Inc., New York.

[14] Tarek Elgamal. 2018. Costless: optimizing the cost of serverless computing through function fusion and placement.2018 IEEE/ACM Symposium on Edge Computing (SEC). IEEE, 300–312

[15] Joel Scheuner and Philipp Leitber, et al. 2022. Function as a service performance evaluation: A multivocal literature review: Journal of Systems and Software 170 (2022), 110708.

[16] Lewis J., Fowler M., 2014. Microservices. Accessed 8.4.2022. https://martinfowler.com/articles/microservices.html.

[17] Tarek Elgamal. 2018. Costless: Optimizing cost of serverless computing through function fusion and placement. In 2018 IEEE/ACM Symposium on Edge Computing (SEC). IEEE, 300–312

[18] Dragoni N., et al. Microservices: Yesterday, Today, and Tomorrow. Present and Ulterior Software Engineering. Springer, Cham, pages 195-216, 2017.