https://doi.org/10.31891/csit-2026-1-13

Yurii GUNCHENKO
Head of the Computer Systems and Technologies Department, Doctor of Technical Sciences, Professor, Odesa I.I. Mechnikov National University, Odesa, Ukraine
e-mail: gunchenko@onu.edu.ua
https://orcid.org/0000-0003-4423-8267
Alla KAMIENIEVA
Associate Professor of the Computer Systems and Technologies Department, PhD in Technical Sciences, Odesa I.I. Mechnikov National University, Odesa, Ukraine.
e-mail: alla.kameneva@onu.edu.ua
https://orcid.org/0000-0002-9970-9081
Maryna IEPIK
Associate Professor of the Computer Systems and Technologies Department, PhD in Technical Sciences, Odesa I.I. Mechnikov National University, Odesa, Ukraine.
e-mail: m.iepik@onu.edu.ua.
https://orcid.org/0000-0001-9021-3680,

Received: 13/12/2026
Accepted:24/02/2026
Published:26/03/2026

UDC 004.75

RESEARCH ON THE AWS LAMBDA PERFORMANCE FOR USER AUTHENTICATION IN CROSS-PLATFORM CLOUD APPLICATIONS

The paper is devoted to investigating the effectiveness of using the serverless AWS Lambda architecture to implement a scalable user authorization system in a cloud environment. The authorization function is deployed as a cross-platform component that interacts with the managed Amazon Aurora PostgreSQL relational database and uses the AWS Systems Manager service to securely store configurations. The function's performance depending on environment parameters (memory capacity, cold/warm start) was investigated, as well as the JWT token generation implementation for user identification. The results can be applied in the development of cloud applications focused on high scalability, portability, and secure database operations.

Keywords: AWS Lambda, user authorization, serverless computing, Amazon Aurora PostgreSQL, JWT tokens, cloud function performance, cross-platform applications.

Introduction

In modern conditions of digital technology rapid development, ensuring secure and scalable user authorization in cloud applications is becoming particularly important. With the growth in the number of web and mobile clients, traditional server solutions no longer meet the requirements for flexibility, performance, and cost-effectiveness. In this regard, it is important to implement architectural approaches that allow computing resources to be scaled according to load, while ensuring high availability and data security. One such solution is serverless architecture, offered by Amazon Web Services (AWS) through the AWS Lambda service.

Serverless computing allows software functions to be performed without the need to manage infrastructure, which is particularly important for cross-platform application developers. In the authorization context, this enables the creation of modular, independent, and easily scalable components that can be integrated into mobile or web applications regardless of the operating system or client environment. This provides reliable access to the centralized databases, such as Amazon Aurora PostgreSQL, which guarantees data security, transactional integrity, and SQL standard support.

The use of AWS Lambda in combination with a managed database and configuration management system (AWS Systems Manager Parameter Store) opens up new opportunities for implementing modern approaches to account management, secret storage, and access token generation, including JWT. This system allows you to flexibly process incoming requests, validate account data, create cryptographically secure access tokens, and record event logs – all without a constantly running server environment.

Unlike traditional server architectures, where request handling depends on preconfigured instances, AWS Lambda performs functions on demand. This not only reduces costs, but also increases system elasticity, as it automatically adapts to load changes. However, such flexibility requires careful analysis of performance under different conditions, including cold and warm starts, as well as different capacities of allocated memory.

Recent years have seen a rapid increase in interest in serverless computing, which has led to numerous investigations into Function-as-a-Service (FaaS) architectures. Works [1], [2]

discuss the key advantages of serverless models, including automatic scaling, reduced infrastructure costs, adaptability to uneven loads, and convenience for developers. Of particular importance is the support for event-driven architecture, which allows the creation of reactive services, including on-demand authorization functions.

Deploying authorization mechanisms in a cloud environment requires integration with managed databases. Papers [3] and [4] analyze the effectiveness of using Amazon Aurora as a scalable and PostgreSQL-compatible solution for storing account data. The authors focus on its high availability, automatic backup, transaction support, and integration with other AWS services. They highlight the importance of using managed configurations and secrets through AWS Systems Manager Parameter Store [5], which lets you securely store passwords, keys, and other sensitive parameters.

Modern user authorization methods described in [6], [7] are based on the use of access tokens, in particular JWT (JSON Web Token), which support cross-platform interaction between the client and the server. This allows for the implementation of stateless APIs, which is especially important for serverless functions that do not store user sessions between calls. The works also note the need to adhere to the principles of secure authentication and restriction of access rights to the database.

The problem of AWS Lambda performance is discussed in [8] and [9], where the authors investigate the impact of cold starts, allocated memory capacity, and number of function calls on response latency. Special attentions are paid to optimizing the execution time of tasks related to database access and token generation. The paper [10] provides a comparative analysis of AWS Lambda with other FaaS solutions (Azure Functions, Google Cloud Functions), which also confirms the AWS architecture's competitiveness in authorization tasks.

In the context of cross-platform compatibility, it is worth mentioning publications [11], [12], which focus on the advantages of developing universal cloud backends capable of serving both mobile and web clients. Through the use of RESTful API, authorization functions, and unified data formats (JSON), it is possible to create a single microservice that adapts to the needs of different platforms.

Hence, a review of existing investigations demonstrates that the combination of AWS Lambda, Amazon Aurora, and JWT technologies is a promising approach for building scalable, secure, and cross-platform authorization systems. However, the question of accurate performance analysis and determination of optimal configurations for such systems remains open, which explains the relevance of this paper.

The purpose of this work is to evaluate the AWS Lambda performance for implementing a scalable user authorization system in terms of integration with a database and cross-platform clients. The paper analyzes the role of managed services in improving the security, stability, and performance of such solutions. The results of this investigation can be used as a basis for developing authorization modules in cloud applications focused on mobile platforms, web interfaces, and multi-component systems.

Scalable user authorization cloud components

System architecture

Scalable authorization in the cloud environment requires coordinated interaction between computing functions, databases, and secure configuration storage. Three main AWS platform components were used in the investigation: AWS Lambda, Amazon Aurora PostgreSQL, and AWS Systems Manager Parameter Store (SSM). Their functional role and technical integration within the authorization module are described.

The implemented authorization system architecture is based on a combination of serverless computing, a managed database, and a centralized configuration store. The user sends account data via an HTTP request, which is processed by API Gateway. The request invokes an AWS Lambda function, which retrieves parameters from AWS Systems Manager Parameter Store, connects to the Amazon Aurora PostgreSQL database, performs account data verification, and generates a JWT token. This token is returned to the client in the response. The entire system is built on AWS, ensuring high availability, scalability, and security.

Figure 1 shows the user authorization system architecture in AWS. The client (mobile or web application) sends a request via API Gateway to AWS Lambda. The Lambda function interacts with Amazon Aurora PostgreSQL to check account data and stores the configuration in SSM Parameter Store. After successful verification, a JWT token is returned.

Database

A managed relational database, Amazon Aurora PostgreSQL, which is compatible with PostgreSQL (a high-performance database that provides automatic scaling, backup, and low latency), is used to store account data. In the context of authorization, it performs username and password verification.

For secure password storage, the *crypt()* function from the *pgcrypto* module is used, which allows you to store hashed passwords and verify them without decrypting them.

Secure configuration storage

Confidential information, such as database passwords or secrets for signing tokens, is stored in AWS SSM Parameter Store. This avoids hard-coding secrets in the code and allows for centralized management of access to them.

The authorization system relies on serverless computing, centralized secret storage, and a managed database, which together form a reliable, scalable, and secure architecture.

AWS Lambda: on-demand code execution

AWS Lambda is a serverless service that allows you to run functions in response to events (such as an HTTP request from a client) without the need to manage infrastructure. In this system, the Lambda function processes authorization requests: it reads the account data, checks them against the database, and, if successful, generates a JWT token. Figure 2 shows the authorization function flowchart.

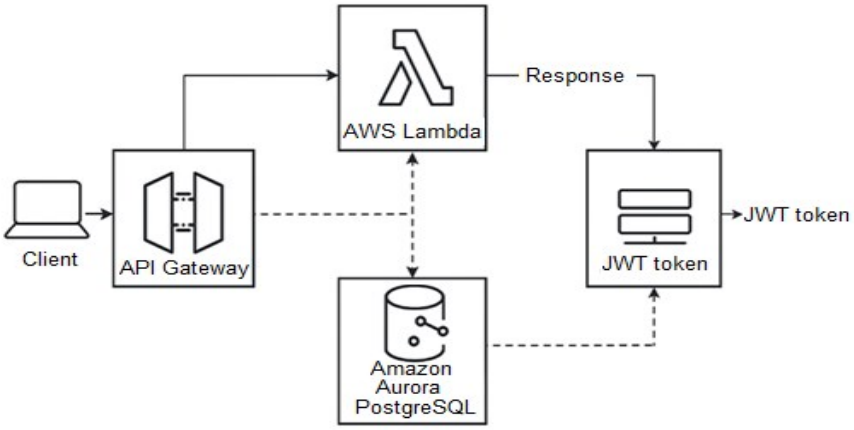

Fig. 1. User authorization system architecture in AWS

The authorization function operates as an independent computing microservice that does not require constant activity and automatically scales according to the load.

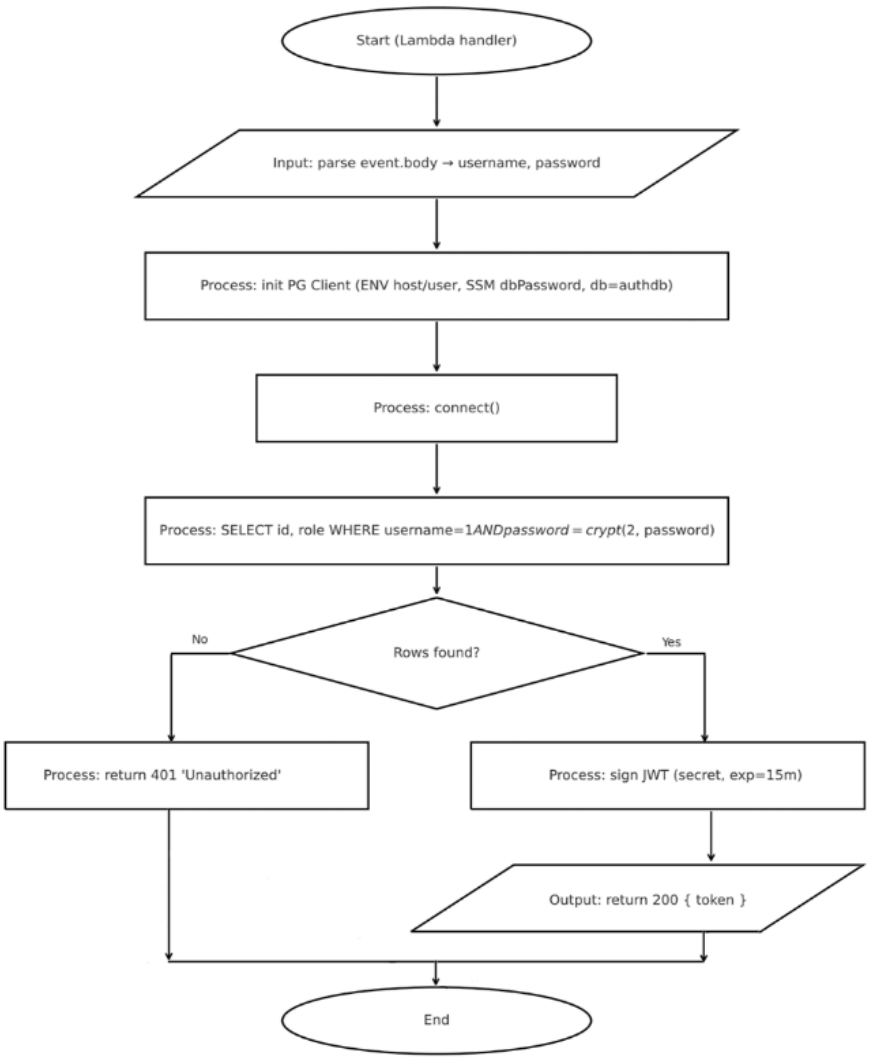

Fig.2. Authorization function flowchart

This section provides an overview of the leading AWS cloud platform, with a focus on its server computing services and managed databases. AWS Lambda has demonstrated high flexibility and scalability for executing event-driven code without the need to manage infrastructure. Particularly useful features include automatic scaling, support for multiple programming languages, integration with other AWS services, support for private networks, and containerization.

Amazon Aurora provides reliable, high-performance, and scalable data storage with automatic replication, read load balancing, and high availability. Its compatibility with MySQL/PostgreSQL, support for serverless configuration (Aurora Serverless v2), and separation of storage and compute capabilities make it easy to integrate into complex, scalable systems.

Hence, the combination of AWS Lambda and Aurora provides you with a modern, efficient architecture for creating productive and reliable applications, especially for authorization and data management tasks.

Experimental evaluation of AWS Lambda function performance for user authorization

The memory capacity's impact on the time and cost of executing a function using the Argon2 algorithm

The dependence of the AWS Lambda function performance, which performs password hashing using the Argon2 algorithm, on the allocated memory capacity (the RAM capacity of the function) is considered. Argon2 is a modern cryptographic algorithm recommended for secure password storage, which is used in the authorization function to verify the correctness of account data.

To evaluate this, a set of experiments was conducted with different memory capacities, ranging from 128 MB to 2048 MB.

For each configuration, the following was measured:

1) function execution time in the case of a cold start (first environment load),
2) execution time in the case of a warm start (repeated call without reinitialization),
3) execution cost in both cases, calculated according to the official AWS Lambda pricing model.

Table 1

Function execution speed dependence on RAM capacity

RAM	Cold start		Warm start	
	Execution time, ms	Cost, USD	Execution time, ms	Cost, USD
128	4064.47	8.5354E-06	3185.10	6.6887E-06
256	2074.28	8.7120E-06	1585.09	6.6574E-06
512	1026.10	8.5166E-06	756.31	6.2773E-06
1024	503.27	8.4047E-06	375.26	6.2668E-06
2048	260.35	8.6697E-06	183.46	6.1093E-06

Execution time analysis: cold start. Analysis of the results in Table 1 shows an exponential decrease in function execution time with increasing RAM capacity. For Argon2, which is a resource-intensive algorithm, low memory capacity (128 MB) results in unacceptably high latency – over 4 seconds. Increasing the capacity to 2048 MB reduces the time to 260 ms, which corresponds to a 15-fold acceleration.

Hence, using small memory capacities (<512 MB) for cryptographic functions significantly impairs performance and is not recommended for real-time authorization services.

Execution time analysis: warm start. The data in Table 1 shows a similar trend for a warm start: when the function is called again (warm start), the execution time decreases from 3185 ms to 183 ms. Although the gain is less noticeable due to the reuse of the initialized environment, the effect remains significant – up to a 17-fold acceleration.

Conclusion: for consistently low latency during warm starts, it is recommended to use at least 1024 MB of memory, with 2048 MB being optimal.

Interestingly, the cost of performing the function remains almost unchanged for all RAM levels. For a cold start, the value ranges from 8.4 to 8.7 × 10^{-6} USD, despite the fact that the memory capacity increases 16 times. This is because as the RAM capacity increases, the function runs much faster, so the time is reduced and the overall speed remains unchanged.

Conclusion: increasing memory capacity has almost no impact on cost, but significantly improves performance, so it is advisable even in budget scenarios.

AWS Lambda performance with SSM

The impact of allocated memory capacity on the AWS Lambda function's performance and cost when accessing the AWS Systems Manager Parameter Store (SSM) service is considered. This component is used in the authorization function to securely obtain configuration parameters (such as database passwords or secret keys). Unlike computations within the function itself, accessing SSM requires an external request, access verification via IAM, decryption of the value via KMS, and transmission of the result to Lambda, which significantly increases the overall latency.

Table 2

Function execution speed dependence on RAM capacity

RAM	Cold start		Warm start	
	Execution time, ms	Cost, USD	Execution time, ms	Cost, USD
128	8036.04	16.8757E-06	3363.09	7.0625E-06
256	4007.14	16.8300E-06	1610.78	6.7653E-06
512	1967.15	16.3274E-06	822.84	6.8295E-06
1024	970.15	16.2015E-06	428.57	7.1571E-06
2048	540.81	18.0088E-06	233.53	7.7765E-06

Execution time analysis: cold start. Analysis of the results in Table 2 shows a typical exponential dependence: at 128 MB, the function takes more than 8 seconds to execute, while at 2048 MB, it takes only 540 ms. This is almost a 15-fold reduction in response time. However, the absolute delay values are higher than with Argon2 hashing because the SSM request involves network interaction, authorization, KMS decryption, and response transmission.

Conclusion: SSM is one of the most "difficult" authorization stages. Reducing the delay is only possible by increasing the memory capacity or pre-caching values in the function code.

Execution time analysis: warm start. As shown in Table 2, the time is significantly reduced during a warm start: from 3363 ms (128 MB) to 233 ms (2048 MB). This indicates that even when restarting the function, performance when accessing SSM significantly depends on the allocated memory, albeit to a lesser extent than when initializing the environment.

Conclusion: even with a warm start, it is recommended to use at least 1024 MB of memory to ensure acceptable latency in the authorization API.

Evaluation of the execution cost. The cost for calls to SSM is the highest among all stages investigated. On average, it ranges from 16 to 18 $\times 10^{-6}$ USD for a cold start (Table 2). This is due not only to the execution duration, but also to the fact that calls to SSM activate auxiliary services, in particular KMS and IAM. Even with a warm start, the cost decreases only slightly, for example, with 2048 MB, only from 18.0 to 7.7 $\times 10^{-6}$ USD.

Conclusion: due to the high cost and delay in using SSM, it should be limited to only critically important parameters, while others should be cached within the function or stored as encrypted layers (Lambda layers).

Aurora PostgreSQL database query performance

In the experiment, the AWS Lambda function connects to the Aurora PostgreSQL database and executes a SELECT query to verify user account data. Aurora runs in Serverless v2 mode in a single availability zone. The experiment aims to evaluate how changing the memory capacity affects connection time, query execution time, overall latency, and function cost.

The graph in Figure 3 shows that as RAM increases, the function execution time is significantly reduced in both scenarios. In the case of a cold start, the function runs slower, but even then, the increase in RAM reduces the delay by 13 times (from ~8790 ms to ~659 ms). In a warm start, the reduction in execution time is even more effective – from ~3623 ms to ~297 ms.

At the same time, the cost of the function execution remains almost stable (Fig. 4), fluctuating within the same order of magnitude (approximately 8–9 $\times 10^{-6}$ US dollars). This indicates the advisability of increasing memory capacity, as it provides a significant improvement in performance without a proportional increase in costs.

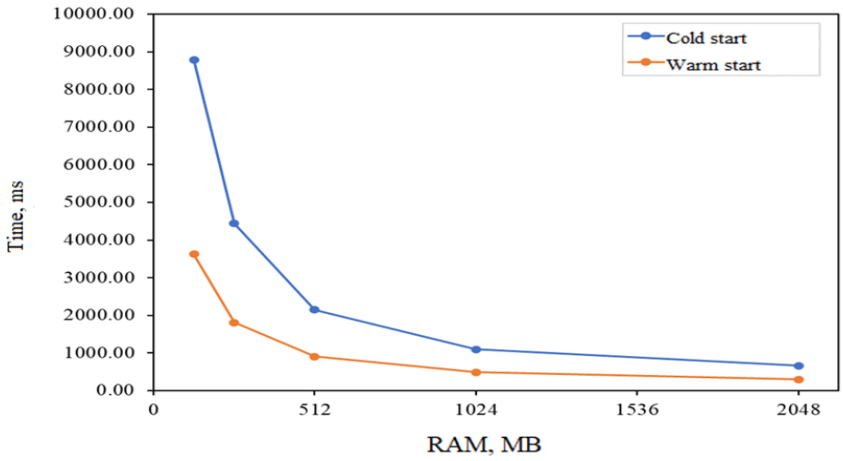

Fig. 3. Graph of function execution speed dependence on RAM capacity

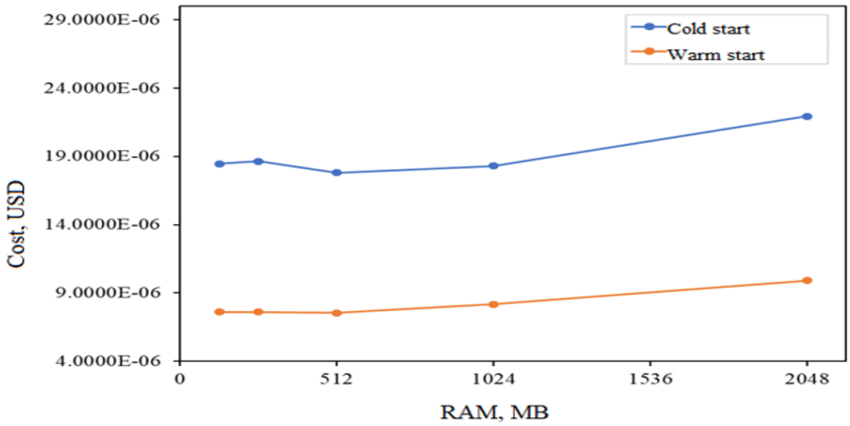

Fig. 4. Graph of function execution cost dependence on RAM capacity

JWT token generation: performance and costs

Experiment essence: JSON Web Token (JWT) is used to transfer authorization information between the client and server. After successfully verifying the account data, the Lambda function generates a signed token that stores the user ID, role, creation time, and expiration time. Token generation involves cryptographic encryption (HMAC or RSA), JSON serialization, and response return via API Gateway. The experiment aims to determine how the amount of memory (RAM) allocated to the function affects the JWT generation time and its cost.

The results shown in Table 3 indicate that as RAM capacity increases, token generation time decreases significantly in both cold and warm start scenarios. In a cold start, there is a decrease in time from ~9518 ms (128 MB) to ~710 ms (2048 MB), i.e., more than 13 times. In a warm start, from ~3520 ms to ~289 ms, which is almost a 12-fold improvement.

In addition, the function execution cost remains stable – it does not increase linearly with memory capacity, but fluctuates within the same order of magnitude: approximately 7 to 10 $\times 10^{-6}$ USD. This demonstrates the effectiveness of scaling: more RAM allows for significantly shorter execution times without a significant increase in costs.

Hence, for JWT generation, it is most appropriate to use RAM in the range of 1024–2048 MB, which ensures minimal delays with a negligible impact on the overall function cost. As a result, all four components (argon2, SSM, DB, JWT) confirm that increasing memory significantly reduces execution time with almost no impact on warm start cost.

Table 3

Function execution speed dependence on RAM capacity

RAM	Cold start		Warm start	
	Execution time, ms	Cost, USD	Execution time, ms	Cost, USD
128	9518.49	19.9888E-06	3519.04	7.3900E-06
256	4778.19	20.0684E-06	1826.59	7.6717E-06
512	2295.47	19.0524E-06	926.12	7.6868E-06
1024	1168.74	19.5180E-06	492.66	8.2273E-06
2048	709.64	23.6310E-06	288.93	9.6212E-06

Below is a summary table comparing all four stages of Lambda authorization function execution by key metrics: time (ms) and cost (USD), for both cold and warm starts, at maximum configuration (2048 MB).

Table 4

Performance of Lambda function stages (2048 MB)

Execution stage	Cold start, ms	Warm start, ms	Cold start cost, USD	Warm start cost, USD
Argon2	260.35	183.46	8.6697E-06	6.1093E-06
SSM	540.55	233.25	1.8003E-05	7.7688E-06
Database	658.18	312.14	2.1921E-05	1.0392E-05
JWT	709.47	289.62	2.3625E-05	9.6452E-06

Conclusions from the summary table:

1. Fastest component: Argon2 with a warm start – only 183 ms.
2. Slowest with a cold start: JWT – over 700 ms, due to cryptography initialization.
3. Most expensive component: JWT with a cold start – 23.6 $\times 10^{-6}$ USD.
4. Least sensitive to RAM: SSM, because delays depend more on external calls.
5. Optimal strategy: for real production load, it is worth using pre-allocated instances or Lambda containers with pre-initialized dependencies, while setting the memory to at least 1024–2048 MB.

Results and Discussion

As a result of the experimental evaluation of the Lambda authorization function's performance, four key stages were investigated: password hashing (argon2), SSM access, interaction with the PostgreSQL database, and JWT generation. A comparison of execution time and cost was performed for cold and warm start scenarios with 2048 MB of memory.

The longest stage during a cold start was the JWT token generator (over 700 ms), which is related to the cryptographic dependencies initialization. The fastest execution was observed for Argon2 during a warm start. Cost analysis confirmed that the function with the selected configuration remains cost-effective and benefits significantly from the use of "warm" instances.

Overall, the results indicate that AWS Lambda is a viable option for authorization tasks, especially when combined with AWS managed services. The best performance is achieved when using a high memory configuration (1024–2048 MB) and provisioned concurrency. This reduces latency, avoids unpredictable cold starts, and ensures stable, scalable operation of the authorization service.

The paper explores the possibilities of using AWS Lambda serverless architecture in combination with Amazon Aurora managed database to implement a scalable user authorization system. The analysis confirms that this combination enables high performance, automatic scaling, flexible resource management, and reduced infrastructure costs. With support for multiple programming languages and architectures, AWS Lambda integrates well with cross-platform applications, including mobile, web, and IoT solutions. Aurora, in turn, provides a stable and scalable relational database with high availability, which is critical for processing user requests and storing account data.

The investigation confirmed the performance of the serverless AWS Lambda architecture for implementing modern user authorization in a cloud environment. The results of the experiments demonstrated that correctly selected environment parameters, particularly the RAM capacity, significantly impact the function speed without significantly increasing the execution cost. Integration with the Amazon Aurora managed database and secure configuration storage in SSM ensures the reliability, flexibility, and scalability of the system.

The conclusions obtained can be used as a basis for developing productive, cost-effective, and cross-platform authorization services in real production environments. This opens up prospects for further investigation into automatic scaling, cost optimization, and improving the resilience of cloud architectures to next-generation workloads.

Introduction

Mental health is a growing issue today, as more and more people struggle with stress, anxiety, depression, and insomnia. Lack of support, high levels of psychological distress, and limited access to effective treatments exacerbate the problem. Addressing mental health issues may require a holistic approach that combines medical treatment with more natural and accessible methods. One such method is music, which can improve mood and reduce stress and tension. This research is directly related to several Sustainable Development Goals. In particular, the research is fully relevant to SDG 3 Good Health, as finding new, effective, and accessible ways to support mental health is an important challenge today. It is also relevant to SDG 4 Good Education, as knowledge about how music affects mental health can inform educational programs and increase psychological literacy. The connection with SDG 10 Reducing Inequality is important, as music is accessible to almost everyone, regardless of age or social status, and can serve as a universal means of emotional support. In addition, caring for the mental well-being of society helps to build a more cohesive and resilient environment, which is in line with the principles of SDG 16 Peace and justice.

Available scientific data confirm that the regular use of music as a means of emotional self-regulation helps reduce stress and anxiety, which defines it as one of the most accessible ways to strengthen mental health.

The impact of music is individual and depends on musical preferences, the tempo of the composition and the emotional state of the person. In addition, the same music can affect different people differently due to individual tastes and different psycho-emotional states, and regular listening to certain tracks can enhance the therapeutic effect. That is why this emphasizes the personalized selection of recommendations. Therefore, technologies that can automatically select musical recommendations according to a person's psycho-emotional state are becoming relevant.

Available scientific data confirm that the regular use of music as a means of emotional self-regulation helps reduce stress and anxiety, which defines it as one of the most accessible ways to improve mental health [1]. The study [2] combines advances in music research from neuroscience, psychology, and psychiatry to connect the specific foundations of music in human biology with its specific therapeutic applications.

ADDITIONAL INFORMATION

AUTHOR CONTRIBUTIONS

Conceptualization, A.K.; methodology, A.K. and Y.G.; software, M.I.; validation, Y.G. and M.I.; investigation, A.K. and Y.G.; resources, M.I.; writing - original draft preparation, M.I.; writing - review and editing, M.I.; visualization, A.K.; supervision, Y.G.; project administration, Y.G. All authors have read and agreed to the published version of the manuscript.

DECLARATION ON THE USE OF GENERATIVE ARTIFICIAL INTELLIGENCE TOOLS

The authors acknowledge the use of artificial intelligence tools, specifically ChatGPT and Grammarly, to check grammar and spelling, formatting the manuscript. After using these services, the authors reviewed and edited the content and take full responsibility of this publication's content.

REFERENCES

1. Baldini, I., Castro, P., Chang, K. et al. Serverless computing: Current trends and open problems. *Research Advances in Cloud Computing*, 2017, pp. 1–20.
2. Spillner, J., Mateos, C., Monge, D. A. FAASCAPES: towards a taxonomy for serverless computing in cloud environments. *IEEE Cloud Computing*, 2019, 6(5), pp. 48–57.
3. Wu, H., Shah, A., Gupta, R. Aurora: Design and Performance of the Next-Generation MySQL-Compatible Cloud Database. *AWS re:Invent Conference*, 2017.
4. Baeza-Yates, R. et al. Scalable Databases in the Cloud: Data Models and Performance. *Proceedings of the VLDB Endowment*, 2018, 11(12), pp. 2106–2109.
5. AWS Documentation. AWS Systems Manager Parameter Store – Best Practices. [Online]. Available: https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-parameter-store.html
6. Jones, M., Bradley, J., and Sakimura, N. JSON Web Token (JWT). *IETF RFC 7519*, 2015. [Online]. Available: https://tools.ietf.org/html/rfc7519
7. Choudhury, O. et al. Enabling secure and scalable authentication with JWT in distributed systems. *Journal of Cloud Computing*, 2020, 9(1), p. 24.
8. Wang, L., Zhang, M., Ristenpart, T., Swift, M. D. Peeking Behind the Curtains of Serverless Platforms. *USENIX ATC*, 2018, pp. 133–146.
9. McGrath, G., Brenner, P. Serverless computing: Design, implementation, and performance. *IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017.
10. Spillner, J. Comparing FaaS Performance: AWS Lambda, Azure Functions, and Google Cloud Functions. *arXiv preprint*, arXiv:2003.04867, 2020.
11. Richards, M. *Software Architecture Patterns*. O'Reilly Media, 2015. (Chapter on Microservices and Backend for Frontend).
12. Taivalsaari, A., Mikkonen, T. A Roadmap to Web Applications: From Web 1.0 to Web 4.0. *Web Information Systems and Technologies*, 2017, pp. 318–336.

Юрій ГУНЧЕНКО, Алла КАМЄНЄВА, Марина ЄПІК
Одеський національний університет імені І.І. Мечникова

ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ AWS LAMBDA ДЛЯ АВТЕНТИФІКАЦІЇ КОРИСТУВАЧІВ У КРОС-ПЛАТФОРМНИХ ДОДАТКАХ

У роботі досліджено ефективність використання безсерверної архітектури AWS Lambda для реалізації масштабованої системи авторизації користувачів у хмарному середовищі. Функція авторизації застосовується у вигляді кросплатформного компоненту, який взаємодіє з керованою реляційною базою даних Amazon Aurora PostgreSQL і використовує сервіс AWS Systems Manager для безпечного зберігання конфігурацій. Розглянуто продуктивність функції авторизації залежно від параметрів середовища (обсяг пам'яті, холодний/теплий старт). Реалізовано генерацію JWT-токенів для ідентифікації користувачів. Результати дослідження можуть бути застосовані при розробці хмарних додатків, орієнтованих на високу масштабованість, портативність і безпечну роботу з базами даних.

Ключові слова: AWS Lambda, авторизація користувачів, безсерверні обчислення, Amazon Aurora PostgreSQL, jwt токени, продуктивність хмарних функцій, кросплатформні застосунки.