Assessing the Migration from FaaS to IaaS: Cost, Performance, and Challenges in AWS

Julián Casaburi[1][0009−0005−1660−5930], Mario Matías Urbieta[1][0000−0002−4508−1209], and Sergio Firmenich[2][0000−0001−9502−2189]

[1] Facultad de Informática - Universidad Nacional de La Plata, Buenos Aires, Argentina
juliancasaburi@gmail.com, murbieta@lifia.info.unlp.edu.ar
[2] Universidad Loyola Andalucía, Spain
sdfirmenich@uloyola.es

Abstract. In cloud-native environments, service model selection is critical for optimizing both operational and economic outcomes. This study investigates the migration from a serverless Function-as-a-Service (FaaS) model, specifically AWS Lambda, to a monolithic solution deployed on Amazon EC2. We examine this transition to evaluate cost savings, performance improvements, and architectural considerations across various scenarios. Our findings indicate that migrating to Infrastructure-as-a-Service (IaaS) can offer notable cost benefits in specific contexts, though it also introduces infrastructure management requirements. This work provides insights into migration decisions and practical considerations when transitioning from FaaS to IaaS-based models.

Keywords: Cloud Computing · Function as a Service (FaaS) · Infrastructure as a Service (IaaS) · AWS Lambda · Monolithic Architecture · Performance Benchmarking · Cost Optimization.

1 Introduction

Function-as-a-Service (FaaS), a serverless computing model, allows developers to execute code fragments without managing the underlying infrastructure. AWS Lambda [9] is a prominent example of FaaS, offering automatic scaling, high availability, and a usage-based pricing model. However, as applications grow in complexity or scale, FaaS can face limitations, making it less suitable or even inadequate for certain use cases.

Real-world migration cases reinforce the limitations of FaaS for high-demand applications. A notable example is Amazon Prime Video's transition from AWS Lambda to a monolithic solution in 2023 [15]. Prime Video initially developed a serverless solution for its video quality monitoring service, orchestrated by AWS Step Functions and powered by Lambda functions. By consolidating operations into a monolithic solution on Amazon EC2, Prime Video achieved an approximately 90% cost reduction.

Given the above, this work explores the migration from AWS Lambda to a monolithic solution through an analysis of key considerations and an empirical evaluation, motivated by the following factors:

- **Cost efficiency:** FaaS can incur higher costs in high-demand scenarios compared to other solutions. This is due to FaaS pricing being based on the number of invocations and execution time, which can lead to escalating costs as traffic increases.
- **Resource utilization:** FaaS can lead to underutilized resources for I/O-intensive tasks or those with high inactivity, such as database queries or API calls. Each function instance handles a single request, regardless of its resource needs. Migrating to IaaS enables multithreaded processing of multiple requests within a single instance, improving resource efficiency.
- **Service model limitations:** AWS Lambda imposes a maximum execution time of 15 minutes per function instance. In some cases, long-running tasks cannot be split into multiple function instances, making Lambda unsuitable. Even when splitting is feasible, additional costs arise due to AWS Step Functions, which charge per state transition [10].

In this study, we introduce an approach to migrate components and assets from a FaaS architecture to a monolithic one in order to reduce costs. The contribution is manyfold: we discuss current costs related to FaaS solutions. Additionally, we propose steps to refactor FaaS assets to run as a monolithic solution, discuss the practical challenges introduced by the migration, and, finally, we introduce a preliminary cost-saving analysis after the migration to evaluate the financial benefits. To evaluate these aspects, we implemented multiple test scenarios, executing load tests with varying request rates and workload characteristics. These scenarios include high and constant traffic loads, unpredictable spikes, I/O-bound processing, and long-running tasks that exceed Lambda's execution limits. The results provide insights into when and why such migrations may be beneficial or impractical, isolating real features from unexpected variables that could affect the assessment.

This work is structured as follows: Section 2 presents related work. Section 3 introduces pricing model concepts. Section 4 discusses key migration considerations, Section 5 examines migration challenges and Section 6 reports the results of the analyzed scenarios. Finally, Section 7 presents the main conclusions and discusses potential areas for future work.

2 Related Work

The migration from monolithic architectures to microservices [1,16], as well as to FaaS [19,21] (a particular case of microservices), has been extensively researched in the literature, detailing challenges and best practices. However, the reverse direction—migrating FaaS solutions to other types—has been less explored. In this regard, a noteworthy study was conducted in 2020 by BBVA, a multinational financial services company, comparing AWS Lambda with Amazon EC2 [2]. Their

study challenged the prevailing assumption that FaaS is universally more cost-effective by examining realistic traffic patterns and workload distributions. The findings indicated that the cost advantages of Lambda diminish in high-traffic scenarios characterized by constant or long-running processes. While Lambda proved to be cost-efficient and low-maintenance for sporadic or low-traffic use cases, EC2 was more economical for stable and high-volume traffic. Lambda's invocation-based pricing model quickly led to costs surpassing those of a highly available EC2 setup.

Notably, following BBVA's cost study and Prime Video's migration, AWS introduced a tiered pricing model for Lambda in August 2022 [3]. This new pricing structure offers differentiated rates based on the volume of GB-seconds consumed. For example, a Lambda function running on an x86 architecture with 2048 MB of memory, an average duration of 60 seconds, and 75 million monthly invocations would experience the following cost changes: USD 150,015.30 without tiered pricing and USD 145,015.29 with tiered pricing. This results in a monthly saving of USD 5,000.01. Therefore, after this update, Lambda costs can become more competitive for large workloads. These modifications may impact the conclusions of previous cost analyses, justifying a reassessment of various scenarios. Thus, this work conducts a cost and performance analysis across different types of scenarios and evaluates the associated challenges, aiming to provide a better understanding of when and why this type of migration may be advantageous.

3 Pricing Models

This section details the key aspects of each service's pricing structure, laying the groundwork for the cost comparisons and analyses presented later in this paper.

3.1 AWS Lambda

The AWS Lambda pricing model [4] is based on requests and duration (in "GB-seconds"). For example, a function using 1 GB of memory for 1 second consumes 1 GB-second. Memory allocation ranges from 128 MB to 10,240 MB, with CPU resources proportional to memory. More memory can potentially reduce execution time and lower costs.

AWS Lambda features a tiered pricing structure [3] that offers discounts based on the total number of GB-seconds consumed, with savings of up to 20% for the highest usage tier. This pricing model is differentiated by architecture (x86 and ARM64) and region.

3.2 Amazon EC2

The comparison with Amazon EC2 in this analysis is particularly relevant because EC2 offers full compatibility with the broader AWS ecosystem, facilitating seamless integration with other AWS services commonly used in conjunction with AWS Lambda.

Amazon EC2 users are billed for the entire capacity of their instances, regardless of actual utilization. Pricing depends on various configuration factors, including instance type, CPU, RAM, network capabilities, and storage options.

AWS offers several payment plans to cater to different needs [5]. In this work, we analyze the following two:

- **On-Demand:** Pay by the second with a minimum charge of 60 seconds, suitable for unpredictable workloads.
- **Reserved Instances:** Commit to 1- or 3-year terms to receive significant discounts, ideal for stable, long-term usage.

3.3 Conclusion

In summary, the pricing models reveal that AWS Lambda is particularly advantageous for workloads with infrequent access. However, AWS Lambda can incur higher costs for workloads characterized by consistent high volume or significant processing demands. Organizations must acknowledge that while managed services alleviate infrastructure management, they come at a premium. Therefore, assessing usage patterns thoroughly is essential for making informed decisions regarding AWS Lambda's suitability for specific workloads.

4 Migration Key Concepts

Although alternative approaches exist, such as open-source FaaS frameworks and other serverless services like AWS Fargate, this analysis focuses on the monolithic approach.

Transforming a FaaS-based system into a monolithic one involves modifying architectural decisions to consolidate functionalities into a single cohesive application. This process requires a thorough analysis of the application's requirements to ensure equivalent behavior.

Regarding the following concepts, while the monolithic solution can be deployed on various platforms—whether in on-premises environments, cloud services such as Amazon EC2 (virtualized or bare metal IaaS), or AWS Elastic Beanstalk (PaaS)—the focus here will be on EC2 (IaaS).

For a comparative overview, a table summarizing the considerations discussed in this section is available in the project repository [11].

4.1 Functional Requirements

Code When considering code migration from FaaS to IaaS, the structure of the original Lambda-based solution plays a significant role. Specifically, the number of Lambda functions directly impacts the migration complexity. We can identify two extremes:

- **Lambdalith:** A single Lambda function handles all endpoints, covering multiple aspects. Migrating this type of solution to a monolithic architecture is simpler; however, it is recommended to separate the logic into distinct modules to ensure code maintainability.
- **Single-Purpose Functions:** Functions are decoupled and highly focused on a specific domain aspect. For each endpoint, the solution may consist of a single function. Migrating this type of solution introduces a higher level of complexity, as the interaction flow of the functions must be carefully analyzed.

State AWS Lambda is stateless, meaning any data that needs to persist between invocations must be stored externally (e.g., AWS DynamoDB, AWS S3). In contrast, monolithic solutions can be either stateful or stateless. In a monolithic solution using horizontal scaling, persistent sessions (sticky sessions) can be used at the load balancer level. Ideally, the application should be stateless, meaning each instance should not rely on locally stored data or the state of other instances to process requests.

Event-Driven Programming A key pattern in AWS Lambda is event-driven programming, where functions automatically execute in response to events generated by other services, such as uploading a file to S3 or inserting a record into DynamoDB. To replicate an event-driven system in an IaaS environment, message queues can be implemented. For example, to migrate a workflow where uploading a file to an Amazon S3 bucket triggers a Lambda function, to a monolithic application hosted on Amazon EC2 instances, S3 can be configured to publish a message to an Amazon SQS queue. The monolithic application can then consume this message via polling. Alternatively, S3 events can be sent to Amazon EventBridge, which can route them based on rules to various targets, such as an API endpoint in the application itself.

Rate Limiting Rate limiting sets limits on the number of requests allowed within a specific time interval. Once this limit is reached, additional requests may be rejected, delayed, or processed differently to prevent system overload and performance degradation. When developing solutions with AWS Lambda, rate limiting can be implemented for each function using services like Amazon API Gateway. In contrast, within a monolithic application, rate limiting implementation depends on whether the application runs on a single instance or is horizontally scaled. For single instances, in-memory rate limiting can be used, while for horizontally scaled applications, external databases like Redis can be leveraged. Furthermore, rate limiting can be enforced at the reverse proxy level using solutions like NGINX or managed services like Amazon API Gateway.

Database Transactions In a Lambda-based solution with multiple functions orchestrated by AWS Step Functions, the SAGA pattern [7] is used to manage

distributed transactions and maintain consistency. This pattern breaks a transaction into a series of smaller steps, where each service executes a portion of the transaction and coordinates with others through messages and events. If an error occurs, the pattern ensures that changes up to the failure are undone using compensatory actions. When migrating to a monolithic application, the distributed transaction logic must be consolidated.

Authentication In AWS Lambda-based solutions, authentication is typically managed through cloud services such as Amazon Cognito or Auth0, providing robust and scalable identity and access management. When migrating to a monolithic architecture, several options are available for handling authentication:

- **Option 1: Retain the Existing Authentication Service**. This approach can be achieved through API calls or SDKs.
- **Option 2: Implement a Single Sign-On (SSO) System**. The SSO mechanism allows users to authenticate once and access multiple applications or services without re-entering credentials. Various open-source solutions, such as Red Hat's Keycloak, can be used to implement SSO.
- **Option 3: Authentication Managed by the Monolithic Solution**. In this case, an authentication policy needs to be implemented, such as using the JSON Web Token (JWT) standard.

4.2 Non-Functional Requirements

Logging and Monitoring Effective logging and monitoring are essential non-functional requirements for diagnosing issues, ensuring system reliability, and gaining insights into application performance. Logging is the process of recording events or actions within a system, providing a crucial audit trail for troubleshooting and analysis.

In AWS Lambda, function logs are automatically captured and centralized in Amazon CloudWatch Logs, simplifying log management. For monolithic applications, especially horizontally scaled ones, a centralized logging solution is equally critical. Logs from each instance should be aggregated in a repository for effective analysis and monitoring. This can be achieved using tools like Logstash, Splunk, or cloud services such as Amazon CloudWatch Logs. Beyond logging, comprehensive monitoring is necessary to track performance metrics, detect anomalies, and ensure the overall health of the application. Solutions like Amazon CloudWatch or Prometheus can be used to monitor key performance indicators (KPIs) and trigger alerts for proactive issue resolution.

Permissions for Accessing AWS Services In AWS Lambda-based solutions, permissions to access other services (e.g., S3, DynamoDB, or SQS) are managed through AWS Identity and Access Management (IAM) policies directly attached to each Lambda function. This model allows for granular access control, adhering to the principle of least privilege. When migrating to a monolithic solution,

if hosted on AWS using EC2, permissions can be managed by attaching an IAM role to the EC2 instance with policies granting the necessary access. For a monolithic solution hosted on another cloud provider or on-premises, integration with AWS services requires securely managing access credentials.

High Availability Lambda functions are highly available by default, as they are deployed across multiple Availability Zones (AZs). In a monolithic solution deployed on IaaS, high availability can be achieved with the appropriate deployment configuration. It should be noted that not all application use cases require high availability. Scenarios where high availability is unnecessary and cost savings can be achieved include temporary testing, internal use, or limited-use applications that are not critical.

For Amazon EC2, configuring the distribution of instances across multiple Availability Zones (AZs) is essential to achieve availability similar to Lambda. Auto Scaling Groups (ASG) can automatically manage the number of EC2 instances based on defined conditions such as CPU load or scheduling and can span multiple AZs within a region.

5 Migration Challenges

Various operational and organizational issues may arise during the migration and subsequent operation. For each issue identified, we also propose preliminary solutions to mitigate potential negative impacts. A summary can be found in the repository [11]. The challenges are described next.

5.1 Downtime During System Updates

In a monolithic solution, updates tend to be riskier, as they can introduce downtime if errors are detected during development. A failure in one component can compromise the stability of the entire application, potentially causing service disruptions [13]. To minimize this risk, a blue/green deployment strategy is recommended, allowing traffic to be redirected to the stable version in case of issues with the new version.

5.2 Increase in Operational Costs Due to Infrastructure Management and Maintenance

Migrating to IaaS increases operational costs due to the need to manage and maintain the underlying infrastructure. To mitigate these costs, the following preliminary solutions are proposed:

- **Automated Infrastructure Provisioning:** Use Infrastructure as Code (e.g., Terraform) to automate the provisioning and configuration of infrastructure resources, ensuring consistency and reducing the potential for manual errors.

- **Dynamic Scalability:** Configure Auto Scaling Groups to automatically adjust the number of EC2 instances based on demand or use similar mechanisms in other cloud environments to optimize resource allocation.
- **Automated Monitoring and Proactive Management:** Set up automatic alerts and corrective scripts to address issues such as server failures, minimizing the need for manual intervention.

5.3 Resource Underutilization Due to Horizontal Scaling

Horizontal scaling can lead to resource underutilization, as the last instance added to an Auto Scaling Group often remains underutilized. Consequently, costs increase without fully leveraging the instance's capacity. As a preliminary solution, it is recommended to configure Auto Scaling Groups with a larger number of smaller instances rather than fewer larger ones, to achieve more efficient utilization.

5.4 Increased Provisioning Time for Horizontal Scaling

Unlike AWS Lambda, which scales almost instantly, deploying on IaaS requires starting new instances as demand increases. Provisioning delays can impact the application's responsiveness to traffic spikes, affecting availability and performance. The proposed solutions include:

- **Predictive Scaling Combined with Dynamic Scaling:** Implement predictive scaling based on historical load patterns or anticipated events, allowing pre-scaling to handle demand spikes together with dynamic scaling.
- **AMI Optimization:** Minimize initialization time by integrating dependencies and configurations within the Amazon Machine Image (AMI), instead of relying on instance startup scripts.

5.5 Increased Security Risks

AWS Lambda's sandboxed environment significantly reduces the attack surface, whereas deploying on IaaS infrastructure exposes servers to additional security risks. For example, failing to apply regular operating system updates may leave the server vulnerable to known exploits. Specific risk factors include:

- Direct server management via SSH access, in contrast to AWS Lambda, which does not provide such access.
- Potential misconfiguration of firewall rules, which could increase exposure to threats.

To mitigate these risks, the following solutions are proposed:

- **Abstraction and Segmentation via a Reverse Proxy:** Implement a reverse proxy (e.g., NGINX or AWS Elastic Load Balancer) as an intermediary to hide the IP addresses of origin servers, making them more resilient against attacks such as Distributed Denial of Service (DDoS). Attackers would only target the proxy, which can be hardened and scaled to withstand attacks.

- **Automated Operating System Updates:** Use tools like AWS Systems Manager Patch Manager to schedule and automate OS patches and updates.
- **Regular Updates of Languages, Frameworks, and Dependencies:** Frequent updates reduce vulnerability by incorporating the latest security mitigations.
- **Automated Code Security Analysis:** Integrate static code analysis and dependency checks into the CI/CD pipeline to identify vulnerabilities before deployment.
- **SSH Access Security:** Disable password-based SSH access, allowing only public key authentication, or remove SSH access entirely if not required.

6 Results

This section evaluates the feasibility of migration by measuring differences in performance and cost in scenarios with different functional requirements and traffic patterns. For each scenario, the application code, Infrastructure as Code templates, and load testing scripts are available in the repository [11] to ensure the reproducibility of the tests. The following methodology was applied:

- **Application of Migration Patterns**: For each scenario, the applicable migration concepts discussed in Section 4 were used. For instance, all scenarios incorporated the approaches for code migration and high availability.
- **Load Testing**: Load tests were conducted using K6 [12] on an EC2 m5.2xlarge instance in a separate VPC within the same region as the applications, collecting metrics such as response times and failure rates. Each test was repeated three times, and results were averaged for consistency.
- **Languages and Runtime Environments**: All applications were developed in JavaScript using the Node.js runtime environment (version 20) [18].
- **Monolithic Applications**: For monolithic applications, the PM2 process manager was used in cluster mode [14] to run multiple instances of the application in parallel. This configuration leverages available hardware resources by distributing the load across multiple CPU cores.
- **Instance Selection**: Instance types were selected based on their ability to handle the required number of requests per second for each scenario while maintaining cost-effectiveness. For scenarios using burstable instance types, the standard credit mode was configured. Tests confirmed that the credits earned consistently exceeded the credits consumed, ensuring sustained CPU performance during the tests.
- **Cost Optimization in Lambda**: Optimal resource configurations for each scenario were identified using the AWS Lambda Power Tuning application [6], as recommended in the Amazon Web Services documentation.
- **Exclusions**: The comparison excludes AWS Lambda's "Always free" tier, API Gateway costs (which are identical for both solutions), and data transfer costs, as they are billed similarly for AWS Lambda and Amazon EC2.

- **Lambda Function Duration**: The cost of the Lambda solution was calculated using the average function execution duration as reported by Amazon CloudWatch.
- **Load Balancer Costs**: For scenarios involving horizontally scaled EC2 instances, the costs include both the fixed and variable fees associated with the AWS Application Load Balancer (ALB).

6.1 Cost-Driven Migration Scenarios

The following section presents detailed results for three scenarios, each representing different load patterns and workloads that could influence the feasibility of migration.

Scenario 1: High and Constant Traffic Pattern In this scenario, we tested a CPU-intensive workload with sustained traffic. To do so, we implemented a simple single-endpoint application that performs base64 encoding of text data. The following configurations were used: **EC2 Instance Type** t3.small, **Lambda Configuration** 128 MB of RAM, and **Request Rate** 100 RPS, maintained using k6's constant arrival rate executor.

Table 1 lists the performance and cost results for both solutions. Under a constant load of 100 RPS, the monolithic solution outperformed AWS Lambda in terms of average response times. In this configuration, without considering high availability, on-demand EC2 instances resulted in a 73.76% cost saving, while reserved instances provided an 83.48% reduction compared to Lambda. Furthermore, Figure 1 represents the break-even point for the two solution types, with and without high availability.

If high availability is a mandatory requirement, with a horizontal scaling configuration including at least two active instances running continuously (such as t3.micro, costing USD 54.97 per month) and an AWS Application Load Balancer (USD 39.79 per month), the monthly savings are reduced to 4.38% for on-demand instances and 14.25% for reserved instances. It is important to note that the ALB incurs a fixed base cost of USD 16.43 per month. As the request volume increases, the fixed cost of the ALB becomes a smaller portion of the total, optimizing resource usage and improving cost efficiency. Additionally, to address resource underutilization, we used smaller instances for horizontal scaling, as noted in Section 5.

It is also important to highlight that potential savings can be achieved if the API Gateway service is unnecessary in the monolithic solution, depending on the required features such as rate limiting, authentication, payload validation, and transformation. For example, in the tested scenario, the monthly cost of using API Gateway is USD 262.80 for an HTTP API or USD 919.80 for a REST API.

Scenario 2: Unpredictable Traffic In this scenario, an application experiencing sudden traffic spikes was tested to evaluate how well monolithic architectures deployed on IaaS handle abrupt increases in demand. To do so, we implemented a single-endpoint application that calculates the 100,000th Fibonacci

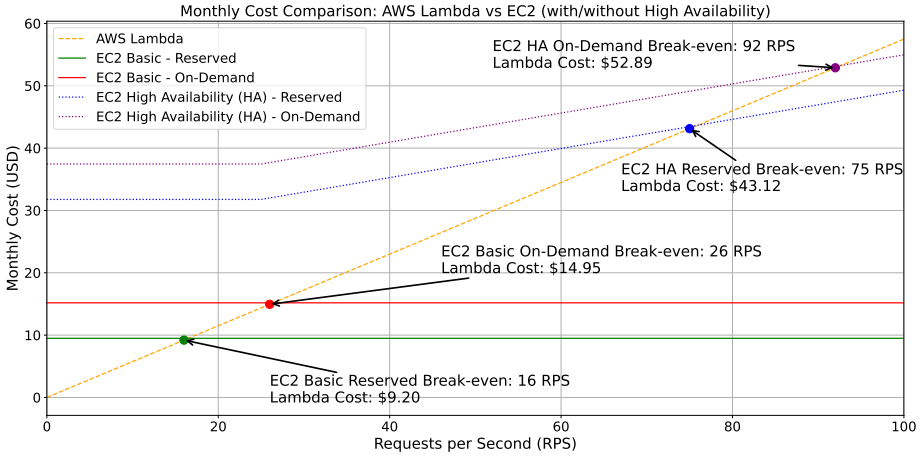

Fig. 1. Break-even point of different solution types for Scenario 1: High and constant traffic pattern (RPS)

number, making it CPU-bound. The following configurations were used: **EC2 Instance Type** t3.small, **Lambda Configuration** 1536 MB of RAM, and **Request Rate** 10 RPS and 20 RPS.

Table 1 lists the performance and cost results for both solutions. The monolithic solution consistently showed lower response times than Lambda for both test rates, while also offering a significant cost advantage. Despite having four instances running at all times, the Lambda solution remains more expensive, resulting in a cost reduction of 43.82% when using EC2 instances with auto-scaling and an AWS Application Load Balancer. Additionally, the break-even point is shown in Figure 2.

Scenario 3: I/O-Bound Workload In this scenario, an I/O-bound application was tested. As discussed in the introduction (1), Lambda's execution model incurs costs even during idle periods, such as when waiting for responses from third-party APIs, database queries, or other I/O operations. This waiting time can significantly reduce Lambda's cost efficiency in such use cases. This scenario was proposed as a potential area for future testing in the comparative study by Villamizar et al. [20]. To simulate an I/O-bound workload, we implemented an application that simulates fetching data from an external API. Upon receiving a request, it initiates three parallel API calls, each introducing a fixed delay of 50 ms. Once all responses are received, the application processes the data by converting the text to uppercase and appending a status message. This workload reflects real-world scenarios in which a significant portion of execution time is spent waiting for I/O operations to complete.

The following configurations were used: **EC2 Instance Type** m5.large, **Lambda Configuration** 128 MB of RAM, and **Request Rate** 4000 RPS.

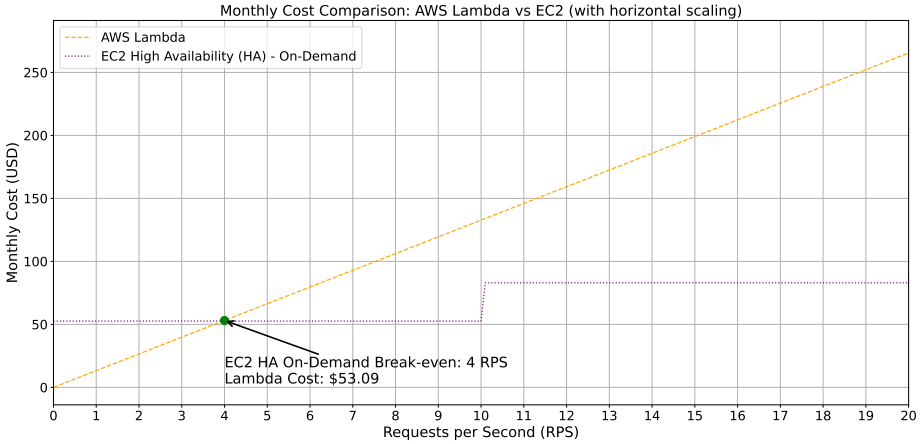

Fig. 2. Break-even point for different solution types in Scenario 2: Unpredictable Traffic

Table 1 presents the detailed performance and cost results for both solutions. As illustrated there, the monolithic solution demonstrated lower response times compared to Lambda. Furthermore, the Lambda solution incurred significantly higher costs, resulting in substantial monthly savings of 70.69% for on-demand EC2 instances and 72.10% for reserved instances when adopting the monolithic approach. Additionally, the break-even point is shown in Figure 3.

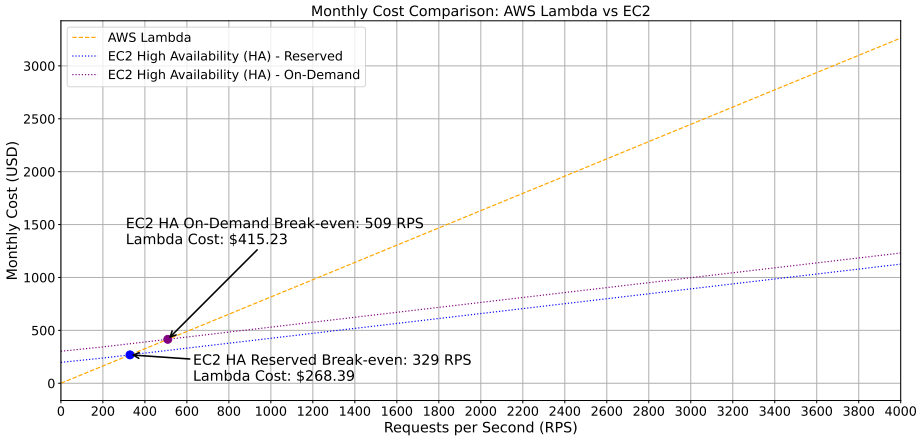

Fig. 3. Break-even point for different solution types in Scenario 3: I/O-Bound Workload

6.2 Migration Scenario Driven by Architectural Limitations - Scenario 4: Long-Running Application

The proposed example for this scenario involves a function triggered by the upload of video files to an Amazon S3 bucket. The function downloads the file to the ephemeral storage of the instance, re-encodes it using FFmpeg to reduce its bitrate and size, and then uploads the processed file to another S3 bucket for storage.

If a video with a duration of 43 minutes and 32 seconds, dimensions of 1280x720, a bitrate of 584 kbit/s, and a frame rate of 30 fps is uploaded to the input bucket, the video exceeds the 15-minute limit mentioned in the introduction (1). Consequently, the video processing remains incomplete. The original solution was designed for videos with a duration limit but now requires an extension of this limit. As a result, it becomes necessary to migrate to a different solution that supports long-running processing. We tested two alternatives using the following configurations: **EC2 Instance Type** c6a.4xlarge, **Lambda Configuration** 3008 MB of RAM (for all functions, both in the single Lambda solution and the orchestrated solution).

The first alternative is a Lambda-based solution orchestrated by AWS Step Functions. In this approach, the video is split into smaller chunks, each processed individually, and then reassembled. While this method allows for longer processing times compared to a single Lambda function, it incurs additional costs due to state transitions and multiple Lambda invocations.

In contrast, the monolithic EC2 solution involves message handling via SQS and processing through EC2 instances. Horizontal scaling is managed by a target tracking policy [8], which adjusts based on the queue length (measured using the ApproximateNumberOfMessagesVisible metric reported by SQS). To mitigate the challenge of increased provisioning time during horizontal scaling in IaaS, we implemented AMI Optimization, building the Amazon Machine Image with all necessary dependencies, including a pre-installed FFmpeg environment, to reduce instance initialization time.

The results, listed in Table 1, show that the monolithic solution is faster and less expensive.

6.3 Cost and Performance Comparison

Table 1 presents a cost and performance comparison of the solutions discussed in the previous four scenarios. The results highlight that, while AWS Lambda provides a highly scalable and maintenance-free execution environment, it is not always the most cost-effective option.

7 Conclusions and Future Work

This study analyzed the migration of a FaaS solution deployed on AWS Lambda to a monolithic solution on Infrastructure-as-a-Service (IaaS), highlighting the challenges, benefits, and key considerations.

Table 1. Cost and Response Time Comparison Between AWS Lambda and EC2 On-Demand Instances

Scenario	Lambda Cost	EC2 Cost	Lambda Time	EC2 Time
1 (Single EC2 instance)	$57.49	$15.18	25.79 ms	7.04 ms
1 (Two EC2 instances)	$57.49	$54.97	25.79 ms	7.23 ms
2	$147.90	$83.08	220.36 ms	178.92 ms
3	$3,723.00	$1,090.80	73.01 ms	59.20 ms
4	$0.06483	$0.02856	3 min 34 s	2 min 48 s

In the tested scenarios, infrastructure cost savings of up to 80% were achieved when features such as high availability were not required, as seen in Scenario 1. Although savings were lower in other cases, they remained significant. Therefore, it is recommended to use AWS Lambda for internal or low-demand tasks, while for other cases, architects should carefully evaluate requirements and costs before making a decision. Finally, since the tests were conducted in a production-ready AWS setup focusing on costs, the evaluation of other scenarios is suggested for a more comprehensive cost analysis.

We plan a few lines of research as future work. In this work, we focused on a monolithic solution but the feasibility of migrating to self-managed FaaS solutions, such as OpenFaaS [17], could be analyzed to determine whether they offer a better cost-performance ratio compared to AWS Lambda and EC2. On the other hand, a catalog of different scenarios could be developed based on those discussed in this work, to explore a broader range of experimental configurations to further strengthen empirical validation. This could involve testing with more diverse workload types, traffic patterns, and application complexities. Finally, migration is a major concern for any development team, so both manual and automated approaches should be analyzed. This may require developing tools to automate the migration process, thereby reducing manual effort and minimizing associated errors.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M.: Decomposition of monolith applications into microservices architectures: A systematic review. IEEE Transactions on Software Engineering **49**(8), 4213–4242 (2023). https://doi.org/10.1109/TSE.2023.3287297
2. Alda Rodríguez, Á., Álvarez, F., Díaz López, G., Evgeniev, M., Horrillo, P.: Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures (2018), https://www.bbva.com/en/innovation/economics-of-serverless/, accessed: March 21, 2024
3. Amazon Web Services: Introducing tiered pricing for aws lambda (2022), https://aws.amazon.com/blogs/compute/introducing-tiered-pricing-for-aws-lambda/, accessed: February 19, 2024

4. Amazon Web Services: Aws lambda pricing (nd), https://aws.amazon.com/lambda/pricing/, accessed: February 19, 2024
5. Amazon Web Services: How aws pricing works (nd), https://docs.aws.amazon.com/whitepapers/latest/how-aws-pricing-works/amazon-ec2.html, accessed: April 1, 2024
6. Amazon Web Services: Profiling functions with aws lambda power tuning (nd), https://docs.aws.amazon.com/lambda/latest/operatorguide/profile-functions.html, accessed: August 8, 2024
7. Amazon Web Services: Saga pattern (nd), https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html, accessed: December 4, 2024
8. Amazon Web Services: Target tracking scaling policies for amazon ec2 auto scaling - amazon web services (nd), https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-target-tracking.html, accessed: October 19, 2024
9. Amazon Web Services: What is aws lambda? (nd), https://docs.aws.amazon.com/lambda/latest/dg/welcome.html, accessed: February 19, 2024
10. Amazon Web Services: What is aws step functions? (nd), https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html, accessed: March 19, 2024
11. Casaburi, J., Urbieta, M., Firmenich, S.: Github repository - applications, load testing scripts, and infrastructure as code (2025), https://github.com/juliancasaburi/icwe-faas
12. Grafana: Grafana k6 (2024), https://k6.io/, accessed: July 16, 2024
13. Kaloudis, M.: Evolving software architectures from monolithic systems to resilient microservices: Best practices, challenges, and future trends. International Journal of Advanced Computer Science and Applications (IJACSA) **15**(9) (2024). https://doi.org/10.14569/IJACSA.2024.0150901
14. Keymetrics: Pm2 documentation - cluster mode (2024), https://pm2.keymetrics.io/docs/usage/cluster-mode/, accessed: July 16, 2024
15. Kolny, M.: Scaling up the prime video audio/video monitoring service and reducing costs by 90% (2023), https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90, accessed: February 19, 2024
16. Newman, S.: Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media (2019)
17. OpenFaaS: Openfaas - serverless functions, made simple (2024), https://www.openfaas.com/, accessed: December 14, 2024
18. OpenJS Foundation: Node v20.9.0 (lts) (2024), https://nodejs.org/en/blog/release/v20.9.0, accessed: February 19, 2024
19. Pedratscher, S., Ristov, S., Fahringer, T.: M2faas: Transparent and fault tolerant faasification of node.js monolith code blocks. Future Generation Computer Systems **135**, 57–71 (2022). https://doi.org/10.1016/j.future.2022.04.021
20. Villamizar, M., Garcés, O., Ochoa, L., et al.: Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures. Service Oriented Computing and Applications **11**, 233–247 (2017). https://doi.org/10.1007/s11761-017-0208-y
21. Würz, H., Kramer, M., Kaster, M., Kuijper, A.: Migrating monolithic applications to function as a service. Software: Practice and Experience **53**(12), 2353–2373 (2023). https://doi.org/10.1002/spe.3263