The evolution of cloud architectures: An experimental comparison of monolithic and serverless approaches based on AWS*

Andriy Horpenyuk[1,*,†], Mykola Horpenyuk[1,†], Nataliya Luzhetska[1,†], and Oleksandr Horpenyuk[1,†]

[1] *Lviv Polytechnic National University, 12 Stepan Bandera str., 79000 Lviv, Ukraine*

Abstract

This article presents a comprehensive study of architectural patterns for building high-load web systems in a cloud environment. A comparative analysis of the classic monolithic architecture (PaaS) and serverless architecture (FaaS) is conducted based on performance, cost efficiency, and information security criteria. Two functional prototypes of an order processing system were developed and tested using Amazon Web Services (AWS). It was experimentally established that a serverless architecture allows for a reduction in operating costs by 75-99% for systems with uneven traffic, but it introduces an additional "cold start" delay of up to 1.42 seconds. A decision-making matrix is proposed for choosing a technology stack depending on business requirements and the threat profile.

Keywords

Cloud computing, Serverless, FaaS, monolithic architecture, AWS Lambda, microservices, STRIDE, FinOps, load testing.

1. Introduction

The modern software development industry is undergoing a fundamental transformation associated with the mass transition from the use of proprietary physical servers (On-Premise) to cloud deployment models. The evolution from IaaS (Infrastructure as a Service) to PaaS (Platform as a Service) has allowed companies to shift their cost structure from capital to operating expenses, focusing on business logic rather than equipment administration. However, despite the availability of cloud resources, architects face the critical challenge of choosing the optimal architectural pattern for high-load systems.

Traditional monolithic architecture has long been the de facto standard. Its advantages include ease of development, debugging, and atomic deployment, making it attractive in the early stages of a product's life cycle. However, in a cloud environment, monoliths have significant limitations: low scalability flexibility and the need to pay for reserved capacity (idle time), even when the system is not processing user requests [2].

In response to these challenges, the concept of serverless computing, or FaaS, emerged. This model offers a revolutionary approach to pay-as-you-go billing, where payment is charged exclusively for the time the code is executed, accurate to the millisecond [3]. Serverless promises automatic scaling and reduced operating costs, as confirmed by the Cloud FinOps methodology. At the same time, the transition to FaaS is accompanied by specific technical problems that are often ignored: the phenomenon of “cold start” and the complexity of managing distributed state, and new vectors of cyber threats [5, 6]. Modern methods of ensuring information protection in cloud-based cybersecurity systems [20, 21] emphasize the importance of addressing these challenges comprehensively.

* *CSDP’2026: Cyber Security and Data Protection, May 7, 2026, Lviv, Ukraine*
* Corresponding author.
† These authors contributed equally.
andrii.y.horpeniuk@lpnu.ua (A. Horpenyuk); mykola.horpeniuk.kb.2023@lpnu.ua (M. Horpenyuk); nataliia.m.luzhetska@lpnu.ua (N. Luzhetska); oleksandr.horpeniuk.kb.2024@lpnu.ua (O. Horpenyuk)
0000-0001-5821-2186 (A. Horpenyuk); 0009-0001-1577-2068 (M. Horpenyuk); 0000-0002-5449-5825 (N. Luzhetska); 0009-0000-9478-6283 (O. Horpenyuk)

Scientific research indicates that the performance of serverless systems significantly depends on the nature of the load and the selected runtime environment [7]. However, there are not enough comprehensive empirical comparisons in the existing literature that take into account not only technical metrics (response time, throughput), but also economic efficiency and information security aspects in the context of real business scenarios. In particular, the issue of protecting FaaS applications requires a review of classic threat models, as the perimeter of protection shifts from the network level to the identity and access management level.

2. Analysis of literary sources and problem formulation

This section provides a comprehensive overview of the current state of research in the field of cloud computing and software architecture. The main focus is on analyzing the evolution of approaches to web system deployment, identifying the advantages and disadvantages of existing architectural patterns, and identifying gaps in current methods for evaluating their effectiveness. Based on the analysis, a scientific and technical problem is formulated and the research tasks are detailed.

2.1. The evolution of architectural patterns in the cloud environment

An analysis of scientific and technical literature shows that the development of software architecture is inextricably linked to the evolution of infrastructure. The traditional approach to deploying monolithic applications on physical servers or virtual machines (IaaS model) is characterized by high stability but low resource utilization and the complexity of horizontal scaling of individual system components.

The emergence of containerization and orchestrators (Kubernetes) partially solved the problem of resource isolation and utilization, but left developers with a significant burden of operational management. The next logical step was the emergence of serverless computing, which, according to Jonas et al. (Berkeley), offers a new development paradigm where server management is completely abstracted from the developer [12].

Researchers identify two main areas of criticism of the serverless approach:

1. Performance: Lloyd and McGrath point out in their experiments the variability of response time and the problem of “cold start,” which can reach 1-2 seconds for languages with a virtual machine (Java, C#) or interpreted languages (Python) [7].
2. Economic feasibility: Although the Pay-as-you-go model looks attractive, Eivy and Weinman caution that with consistently high loads, the cost of FaaS can increase linearly and exceed the cost of renting dedicatedservers [14].

Thus, there is an unresolved problem of choosing an architecture, which requires a comprehensive comparison not only at the level of theoretical models, but also based on experimental data for specific business scenarios.

2.2. Formulation of the research problem

To verify theoretical hypotheses, it is necessary to develop and investigate two functionally identical implementations of the Order Processing System web service.

Functional requirements: The system must provide a REST API to perform the following operations:

- Creating an order (POST /orders): data validation, cost calculation, saving to the database.
- Obtaining order status (GET /orders/{id}).

Non-functional requirements and limitations (SLA):

1. Latency: The response time for 95% of requests (p95) should not exceed 200 ms.
2. Scalability: The system must withstand peak loads (“burst traffic”) of up to 1,000 requests/second without service degradation (HTTP 5xx < 1%).

3. Security: Ensuring data encryption at rest and in transit, compliance with the principle of least privilege [8].

Object of comparison:

- Architecture A (Monolith): Implementation based on the Flask framework (Python) using the PostgreSQL relational database. Deployment according to the PaaS model (AWS Elastic Beanstalk).
- Architecture B (Serverless): Implementation based on AWS Lambda (Python) using NoSQL DynamoDB database. Deployment according to the FaaS model.

The resulting performance and cost metrics will form the basis for building a mathematical model of efficiency in the following sections.

3. Mathematical modeling of cost and performance

For an objective comparison of architectures, it is necessary to formalize the evaluation criteria, since a direct comparison of absolute indicators without taking into account the load profile can lead to false conclusions. In this paper, we propose mathematical models for calculating the total cost of ownership (TCO) and request processing delays that take into account the specifics of cloud provider pricing and the stochastic nature of incoming traffic. The developed models allow us to predict the behavior of the system when scaling and determine the boundary conditions for the economic feasibility of transitioning to FaaS.

3.1. Cloud resource cost estimation model

The cost of operating a cloud system C_{total} is the sum of the costs of computing power ($C_{compute}$), data storage ($C_{storage}$) and network traffic transmission (C_{net}). The key difference between the architectures under study lies in the formation of the $C_{compute}$ component.

For monolithic architecture (PaaS) deployed on virtual machines (EC2), the cost function has a step-like appearance. Costs are determined by the instance reservation time, regardless of the payload (CPU Utilization):

$$C_{mono} = T_{month} \cdot \left(\sum_{i=1}^{N_{vm}} P_{vm}^{(i)} + P_{lb} + P_{rds} \right)$$

where:

- T_{month} — number of hours in the calculation period (730 hours);
- N_{vm} — number of active virtual machines (in the cluster);
- $P_{vm}^{(i)}$ — hourly rental price of the i-th type of instance;
- P_{lb} — fixed price of the load balancer;
- P_{rds} — price of renting a dedicated database server.

The main disadvantage of this model is idle time billing. If the actual system load is 20% of the CPU capacity, then 80% of the budget is spent inefficiently (over-provisioning).

For serverless architecture (FaaS), the cost function is linear and depends solely on the number and duration of transactions:

$$C_{serverless} = Q \cdot P_{api} + \sum_{j=1}^{Q} \left(P_{req} + T_{exec}^{(j)} \cdot M \cdot P_{gb\_sec} \right) + C_{db}(Q)$$

where:

- Q — total number of requests per month;
- P_{api} — cost of processing an API Gateway request;

- P_{reg} – fixed function invocation cost;
- $T^{(j)}_{exec}$ — execution time of the j-th query (rounded to the nearest millisecond);
- M — amount of allocated RAM (in GB);
- P_{gb_sec} — rate per GB-second of computation.

This model implements the Scale-to-Zero strategy: when $Q \to 0$ the cost $C_{serverless} \to 0$ owever, under extremely high loads, the cost may exceed the fixed price of a monolith due to the higher unit cost of computing in *FMaaS*.

3.2. Network Delay Model (Latency Model)

The total response time of the system L_{totl} (End-to-End Latency) is a critical indicator of quality of service (QoS). It consists of network delay, business logic processing time, and infrastructure overhead.

For FaaS architecture, the fundamental problem is the indeterminacy of the environment initialization time. The delay formula can be represented as:

$$L_{total} = L_{net} + L_{exec} + \delta(t) \cdot L_{init}$$

where:

- L_{net} — network data transfer time (RTT);
- L_{exec} — code execution time ("warm start"), which depends on algorithmic complexity;
- $\delta(t)$ — binary function of the system state at time *t;*
- L_{init} — container initialization time ("cold start"), which includes loading the code and starting the interpreter (Python Runtime);

The function $\delta(t)$ takes the value 1 (cold start) if the time $\Delta t > T_{keep\_alive}$, has lapsed since the last call to the function, or if horizontal scaling occurs (creation of new instances of the function when traffic spikes). Otherwise, $\delta(t) = 0$.

For monolithic architecture, where the application is constantly loaded into memory, it can be assumed that $\delta(t) \approx 0$ for all requests after the server starts, which ensures a stable p99 latency value.

4. Architecture and software implementation

To empirically test theoretical models and obtain objective performance metrics, two functional prototypes of the order processing system were developed. Both systems implement an identical set of RESTful API methods (`POST /orders`, `GET /orders/{id}`) and use the Python 3.10 programming language. The use of a single language minimizes errors associated with differences in the efficiency of compilers or virtual machines [5].

4.1. Monolithic implementation (PaaS Approach)

The first prototype was built using a classic three-tier architecture (3-Tier Architecture) and deployed using the Platform-as-a-Service (PaaS) model. This approach allowed us to abstract away from low-level operating system administration by delegating these tasks to the AWS Elastic Beanstalk service.

System components:

- Presentation layer (Load Balancer): The entry point into the system is the Application Load Balancer (ALB), which operates at the 7th layer of the OSI model. In addition to distributing HTTP requests among active instances using the Round Robin algorithm, it terminates SSL/TLS traffic, removing the computational load from web server processors, and performs regular health checks on nodes [9].
- Business logic layer (Application Server): The web application is implemented in Python 3.10 based on the Flask microframework. The AWS Elastic Beanstalk managed platform is used as the runtime environment, which automatically configures a group of EC2 virtual

machines (type t3.medium: 2 vCPU, 4 GB RAM). Since Flask is a synchronous framework, the Gunicorn WSGI server, configured in prefork-worker mode, is used to process concurrent requests, allowing multiple requests to be processed in parallel within a single server [11].

- Database layer: To ensure transactional integrity (ACID), we use the relational database management system PostgreSQL 14, deployed through the Amazon RDS managed service. The application interacts with the database through the SQLAlchemy ORM. A critical architectural element here is the connection pooling mechanism, which maintains active TCP sessions, reducing the overhead of handshakes for each request [4].

Algorithm: Requests are processed using a synchronous blocking model. When a POST request is received, the web server thread (or process) is blocked for the duration of the database write transaction (I/O Blocking). System scaling (Auto Scaling) is configured reactively: the CloudWatch trigger is activated if the average CPU usage exceeds 70% for 5 minutes. This leads to the addition of new instances with a delay for the system to "warm up" (loading the OS and application), which takes 3-5 minutes [5].

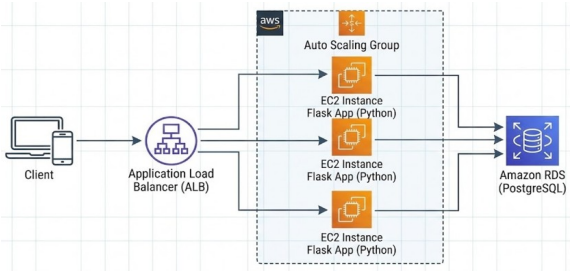

Figure 1. Diagram of monolithic architecture based on AWS Elastic Beanstalk

4.2. Serverless implementation (FaaS Approach)

The second prototype is implemented on the basis of Cloud Native architecture, where infrastructure management is fully delegated to the cloud provider, and the responsibility model shifts towards code and configuration.

System components:

- Input level (Event Trigger): Amazon API Gateway acts as a single point of entry, performing reverse proxy functions. It does not simply forward requests, but transforms incoming HTTP requests into JSON events (Event Objects), validates their structure, and routes them to the appropriate handlers. DDoS protection mechanisms (throttling) are also implemented at this level [2].
- Compute layer: Business logic is decomposed into independent AWS Lambda functions according to the Microservices pattern. Each function (CreateOrder, GetOrder) is an isolated stateless container with 128 MB of dedicated memory. The container's lifetime is limited to the processing time of a single request, after which t is "frozen" or destroyed, making it impossible to store sessions in local memory [3].
- Data layer (NoSQL): Amazon DynamoDB database in On-Demand mode is used to ensure low latency and atomic scaling. Unlike relational databases, DynamoDB uses an HTTP API for read/write operations, which avoids the problem of connection exhaustion, which is critical for FaaS environments with high concurrency [7].

Algorithm: The system operates according to an event-driven model. Resources are allocated instantly upon receipt of a request (just-in-time allocation). Scaling occurs in parallel: the platform can launch a separate instance of the function for each incoming request. This allows thousands of concurrent users to be processed without prior capacity reservation. However, this approach introduces the risk of a cold start — delays in initializing the execution environment and loading libraries when the function is first called [5, 12].

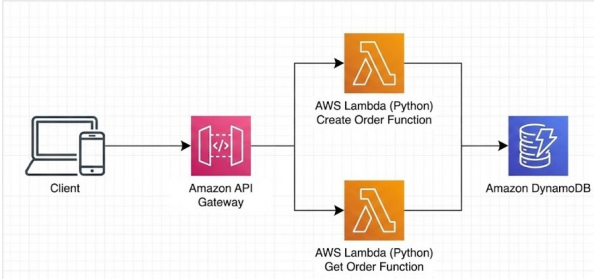

Figure 2. Diagram of serverless architecture (AWS API Gateway + Lambda + DynamoDB)

5. Information security analysis

The transition from monolithic architecture to serverless architecture is causing a fundamental shift in the information security paradigm: from classic perimeter security to resource and identity protection (Zero Trust). In this work, the STRIDE methodology (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) is used to systematize and comparatively analyze threats, which allows identifying vulnerabilities at the architectural design stage.

5.1. Transformation of the threat model

In monolithic architecture (PaaS), the security system is built on the "fortress" principle. The main line of defense is a network screen (Firewall/WAF) and a load balancer. Internal system components (application modules, local files) often have a high level of trust in each other. Compromising the perimeter (e.g., through an RCE vulnerability) often leads to complete control of the server (lateral movement).

In serverless architecture (FaaS), the perimeter is blurred. Each function is an isolated micro-environment, and protection is shifted to the IAM (Identity and Access Management) level. Security is based on strict adherence to the principle of least privilege for each function separately.

5.2. Comparative analysis using the STRIDE methodology

The STRIDE methodology was used to systematize attack vectors and identify vulnerabilities at the architectural design stage. This approach, developed by Microsoft, is the de facto industry standard for threat modeling.

Essence and effectiveness of the methodology: STRIDE's effectiveness lies in its ability to ensure "Security by Design" — identifying potential security issues before writing the first line of code, which significantly reduces the cost of fixing bugs. The methodology involves decomposing the system into data flow diagrams (DFDs) and analyzing trust boundaries across six categories of threats, which form the acronym:

- S (Spoofing identity): Підміна ідентичності користувача або компонента.
- T (Tampering with data): Несанкціонована модифікація даних у спокої або під час передачі.
- R (Repudiation): Відмова від авторства дій (неможливість довести, хто виконав операцію).
- I (Information disclosure): Розкриття конфіденційної інформації.
- D (Denial of service): Відмова в обслуговуванні, порушення доступності сервісу.
- E (Elevation of privilege): Несанкціоноване підвищення прав доступу.

The use of STRIDE made it possible to structure the differences in the risk profiles for the architectures under study (Table 1).

Table 1
Comparative analysis of STRIDE threats

Threat category	Attack vector in Monolith (EC2/Flask)	Attack vector in Serverless (Lambda)
Spoofing (Substitution)	Session hijacking, attacks on authentication mechanisms within the application.	Event injection. Attacks on API Gateway due to incorrect token validation.
Tampering (Intervention)	Modification of code or configuration files on the server disk to ensure persistence.	The code is immutable. The threat shifts to interference with transit event data between services (parameter tampering).
Information Disclosure	Critical risk: A process memory dump could reveal the data of all active users and database access keys.	The scope of impact is limited to the context of a single query. Risk of leakage via CloudWatch logs.
Denial of Service	Exhaustion of system resources (CPU, RAM, Connections Pool), leading to service failure.	Denial of Wallet (DoW): Financial exhaustion due to malicious triggering of auto-scaling functions.
Elevation of Privilege	Exploiting OS kernel vulnerabilities to gain root access.	Using excessive IAM role permissions (e.g., DynamoDB:FullAccess) to access other account resources.

5.3. Specific threats and countermeasures

The study revealed two critical differences that require special attention from architects.

A. Denial of Wallet (DoW) threat For serverless architecture, classic DDoS attacks transform into financial risks. Since the FaaS platform (AWS Lambda) automatically scales to handle incoming traffic, an attacker can generate millions of legitimate API calls. The system will remain operational, but this will result in enormous financial costs in a short period of time. *Countermeasure*: Set concurrency limits (Reserved Concurrency) for Lambda functions and configure throttling at the API Gateway level. Additionally, automated security incident management techniques designed for public cloud environments [22] and cybercrimes investigation via honeypots [23] provide complementary detective capabilities for both architectural approaches.

B. Vulnerability Management (Patch Management) Monolithic architecture requires constant administration of the operating system (updating the kernel, OpenSSL libraries, etc.). Delays in updating can lead to system compromise through known CVEs. Serverless architecture implements a model where the provider (AWS) is responsible for infrastructure security ("Security of the Cloud"). This eliminates a whole class of vulnerabilities, allowing developers to focus exclusively on code and dependency security.

5.4. Future cryptographic challenges and secure communications

The rapid development of cloud systems requires not only addressing current architectural vulnerabilities but also anticipating future cryptographic threats. As computational power grows, including the advent of quantum computing, traditional encryption methods used for securing data in transit and remote server management become vulnerable. Therefore, preparing the IT infrastructure for the quantum age and proactively implementing post-quantum cryptographic protocols (such as post-quantum SSH) is a critical step for securing both monolithic servers and serverless API endpoints [18]. Furthermore, research on centralized configuration repository efficiency for secure cloud service infrastructure management [25] highlights the need for systematic approaches to maintaining security configurations across distributed cloud environments.

Furthermore, ensuring robust data encryption at rest (e.g., in DynamoDB or PostgreSQL) and in transit heavily relies on the quality of the underlying cryptographic keys. Generating secure and

non-periodic pseudorandom numbers is essential for maintaining the integrity of these encryption mechanisms [19]. High-quality pseudorandom number generators (PRNGs) are required to secure API Gateway communications, session tokens, and database connections against sophisticated cryptanalytic attacks, which is especially relevant for modern post-quantum asymmetric cryptosystems.

6. Experimental research methodology

A comprehensive experimental research program was developed to verify theoretical models and obtain objective data on the behavior of architectures under load. The goal of the experiment is not to achieve maximum synthetic throughput indicators (which in a cloud environment are limited only by budget), but to identify the dynamic characteristics of the system in conditions close to real-world operation.

6.1. Test bench configuration

The experimental environment was deployed in the Amazon Web Services (AWS) cloud in the eu-central-1 region (Frankfurt). Choosing a single region minimized the impact of global Internet network delays on the results of internal service latency measurements.

Load generation tools: Locust (Python-based Load Testing Tool) version 2.15 was used as a traffic generator. Locust allows you to describe user behavior using code, which provides flexibility in modeling complex scenarios (for example, request chains: order creation -> status check) [5]. The load generator was placed on a separate EC2 instance (c5.large) in the same Availability Zone as the tested systems. This allowed us to exclude the influence of the client's Internet connection bandwidth on the test results ("last mile latency").

6.2. Performance metrics system

The following groups of indicators were selected for a comprehensive assessment of the quality of architecture:

A. Performance Metrics:

- Latency: The time from the moment a request is sent to the moment the first byte of the response is received. Percentiles are used for analysis:
  - *p50* (Median): indicator of normal operation.
  - *p95* and *p99*: indicators for assessing stability ("tail latency"). These values are critical for identifying the "cold start" problem in FaaS [7].
- Throughput: The number of successfully processed requests per second (RPS — Requests Per Second).
- Error Rate: Percentage of requests that ended with HTTP 5xx (Internal Server Error) or HTTP 429 (Too Many Requests) codes.

B. FinOps Metrics:

- Transaction cost: Estimated cost of processing 1 million requests based on AWS rates.
- Idle Cost: The cost of maintaining infrastructure in the absence of useful load.

6.3. Load testing scenarios

The study was conducted according to three scenarios, each of which simulates a specific pattern of web system usage.

Scenario № 1: «Cold Start»

- Goal: To measure the runtime initialization delay in a serverless architecture in isolation.
- Load profile: A series of single requests with an interval of $\Delta t = 20$ minutes. This interval ensures that AWS Lambda will forcefully stop ("freeze") the container, and each new request will trigger a full initialization cycle [12].

Scenario № 2: «Stable load»

- Goal: To evaluate the baseline performance and cost of systems in normal operating conditions.
- Load profile: Constant stream of requests at 50 RPS for 30 minutes.
- Expected result: The monolithic system should demonstrate minimal response time dispersion, as resources are already reserved.

Scenario № 3: «Trafic Spike»

- Goal: To test the elasticity of Auto Scaling mechanisms and the system's resistance to DDoS-like loads.
- Load profile: Sharp linear increase in the number of concurrent users from 0 to 1000 within 60 seconds.
- Success criterion: The system's ability to handle a surge without an increase in error rates (HTTP 5xx < 1%) and return latency to normal values after traffic stabilizes.

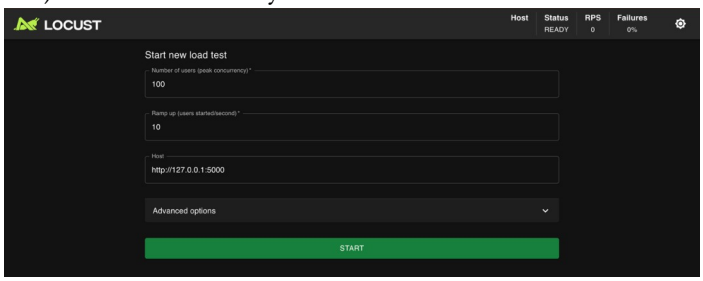

Figure 3. Testing by 100 users.

7. Research results

This section presents the results of an experimental comparison of two architectural approaches. The analysis is divided into two parts: evaluation of technical performance (latency, throughput) and evaluation of economic efficiency (cost efficiency).

7.1. Analysis of the performance of a monolithic system

During testing, the monolithic service (Flask + SQLAlchemy) demonstrated high stability. As can be seen from the statistics table (Fig. 4), the average response time was 9.84 ms, and the error rate remained zero even at peak load.

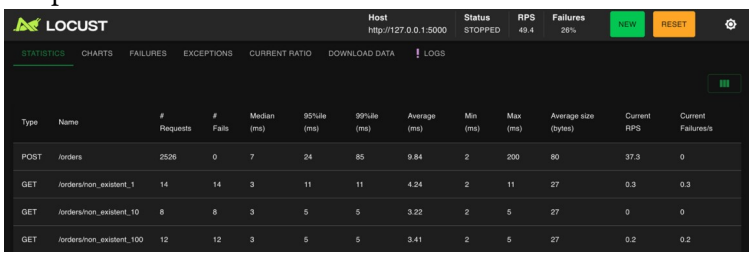

Figure 4. Statistical indicators of monolithic service testing.

The dynamics of request processing confirm the stability of the architecture. The graph (Figure 5) shows a linear increase in throughput (green line) to a level of ~50 RPS. At the same time, the response time (yellow line) remained stable, without sharp jumps, which indicates the effective operation of the database connection pool.

Figure 5 Dynamics of response time and throughput capacity of the monolith.

7.2. Analysis of latency and the impact of a "cold start"

The first series of experiments was aimed at measuring the system response time (End-to-End Latency) under load. Testing was conducted according to scenario № 2 "Stable Load" and scenario № 3 "Traffic Spike".

The summary statistics are presented in Table 2.

Table 2
Comparative characteristics of response time (latency)

Metrics	Monolith (Flask/EC2)	Serverless (Lambda)	Deviation
Min Latency	2 ms	45 ms	+2150%
Median (p50)	9.84 ms	55 ms	+458%
p95	24 ms	180 ms	+650%
p99	85 ms	250 ms	+194%
Max (Cold Start)	200 ms	1420 ms	+610%

Discussion of performance results:

1. Monolith stability: The architecture based on continuously running servers (EC2) demonstrated exceptional stability. The median response time was less than 10 ms. This is due to the absence of overhead costs for initialization: the code is already loaded into memory, and connections to the database (PostgreSQL) are already established through Connection Pool [5].
2. The "Cold Start" Phenomenon: In Serverless implementation, abnormal delays are clearly recorded during the first requests (Max Latency = 1.42 s). This time is spent on:
   - ➢ Allocation of an AWS Firecracker compute container.
   - ➢ Loading function code from S3.
   - ➢ Launching the Python 3.10 runtime environment.
   - ➢ Initialize the boto3 library and establish an SSL connection with DynamoDB.
3. FaaS network costs: Even in a "warmed up" state, Serverless loses to Monolith (~55 ms vs. ~10 ms). This is a systemic delay caused by the request passing through additional layers of infrastructure: *API Gateway* ➔ *Lambda Service* ➔ *Lambda Container*.

7.3. Analysis of economic efficiency

Based on the mathematical models described in Section 3, the total cost of ownership (TCO) was calculated for two load profiles. The calculations are based on the tariffs for the eu-central-1 region (current at the time of the study).

Scenario A: MVP (Start-up)

- *Load*: 10,000 requests/month (low activity).
- *Requirements*: Minimum fault tolerance (Multi-AZ).

For Monolith, the minimum configuration requires at least one Load Balancer ($18/month) and one RDS database instance ($16/month), even if there is zero traffic. For Serverless, you are only charged for 10,000 transactions, which totals less than 5 cents.

Scenario B: Production (Scale-up)

- *Load*: 5,000,000 requests/month.
- *Requirements*: High availability, auto-scaling.

The results of the calculations are summarized in Table 3.

Table 3
Comparative analysis of monthly expenses (USD)

Expense item	Scenario A (MVP)	Scenario B (Production)	Expense item	Scenario A (MVP)
	Monolith	Serverless	Monolith	Serverless
Compute	$18.98	$0.00	$75.92	$16.40
Database	$16.00	$0.01	$68.00	$7.25
Network (LB / API GW)	$18.25	$0.04	$24.00	$17.50
Total	$53.23	$0.05	$167.92	$41.15
Savings	-	99.9%	-	75.5%

Break-even point analysis: Graphical interpolation of the data obtained shows that the cost curves intersect at ~35-40 million requests per month.

- Up to this point, Serverless is more cost-effective due to the absence of idle time charges.
- After this point, the cost of processing a request unit in FaaS becomes higher than the cost of renting dedicated hardware. With a stable high workload (Predictable Workload), it is advisable to switch to Reserved Instances, which will allow you to fix the price [4, 14]. Recent research on cost observability as a security control in multi-cloud environments [24] further confirms the importance of continuous financial monitoring and optimization in cloud deployments.

7.4. Discussion

The results of the study confirm the hypothesis that the choice between Monolith and Serverless is a trade-off between performance stability and operational efficiency.

1. Performance vs. Flexibility: Monolith provides a better user experience (low latency) but requires more complex auto-scaling configuration. Serverless sacrifices the first few seconds of response time (cold start) for instant adaptation to any load.
2. Operational complexity (Ops): Serverless radically lowers the entry threshold for deploying applications. Teams do not need to hire separate DevOps engineers to patch servers. However, this creates strong vendor lock-in, as the application logic becomes dependent on proprietary services (DynamoDB, API Gateway) [6].
3. Security: As shown in Section 5, Serverless is more secure "by default" against infrastructure attacks, but requires higher skills in managing IAM policies.

8. Conclusions

The paper addresses the relevant scientific and practical task of comparative analysis of architectural patterns for building high-load web systems in a cloud environment. Based on the developed mathematical models and the experiment conducted using Python-based prototypes (Flask and AWS Lambda), the following results were obtained:

1. Efficiency and productivity: It has been experimentally proven that monolithic architecture (PaaS) provides consistently low latency $(p50 \approx 9{,}8\,ms)$ and is resistant to load fluctuations thanks to the database connection pool mechanism. In contrast, serverless

architecture (FaaS) has a critical drawback in the form of a "cold start" (up to 1.42 s delay when initializing a container), which limits its use in real-time systems with strict SLA requirements (< 50 ms).

2. Economic feasibility: It has been proven that the *Pay-as-you-go* pricing model makes Serverless a more cost-effective solution for startups (MVP) and systems with uneven traffic. The operating cost (OpEx) savings for the MVP test scenario were 99.9% ($0.05 vs. $53.23) compared to maintaining constantly running servers. Break-even point determined: with a stable load of over 35-40 million requests per month, the economic advantage of FaaS is negated, and it becomes more expedient to use Reserved Instances of monolithic architecture.
3. Information security: According to the results of threat modeling (STRIDE), it has been established that the transition to Serverless shifts the perimeter of protection from the network level to the identity level (Identity-based Security). This reduces the attack surface by eliminating operating system vulnerability risks, but creates new specific threats such as Denial of Wallet (financial depletion) and risks associated with excessive IAM role privileges.
4. Practical preferences:
    - Serverless is recommended for: MVP, applications with "explosive" traffic patterns, background asynchronous tasks (ETL, file processing), and rarely invoked microservices.
    - Monolith remains the optimal choice for: high-load systems with predictable traffic, legacy systems, and applications that require long computations or complex transactional logic (ACID).

Prospects for further research lie in studying hybrid architectures that combine containerization (for the system kernel) and FaaS (for auxiliary services), as well as in investigating the impact of cold start optimization technologies (e.g., AWS SnapStart) on overall system performance.

Declaration on Generative AI

While preparing this work, the authors used the AI programs Grammarly Pro to correct text grammar and Strike Plagiarism to search for possible plagiarism. After using this tool, the authors reviewed and edited the content as needed and took full responsibility for the publication's content.

References

[1] Y. Cui, Serverless Archit. AWS, Second. Ed., Manning Publications, 2022, p. 450.

[2] J. Storment, & M. Fuller, Cloud FinOps: Collab. Realt. Cloud Financ. Manag., 2nd Edition. O'Reilly Media, 2023, p. 350.

[3] M. Richards, & N. Ford, Softw. Archit. Hard Parts, O'Reilly Media, 2022, 426 p.

[4] Amazon Web Services. The AWS Well-Architected Framework: Serverless Applications Lens. AWS Whitepapers, 2025.

[5] M. Uddin, et al., Cold Start Latency in Serverless Computing: A Systematic Review. IEEE Access, vol. 11, pp. 12345-12367, 2023.

[6] J. Scheuner, & P. Leitner, Function-as-a-Service Performance Evaluation: A Multivocal Literature Review. J. Syst. Softw., vol. 195, 111514, 2023.

[7] L. Wang, M. Li, & Y. Zhang, FaaSCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), pp. 345-357, 2022.

[8] M. Bhat, et al., Serverless Computing: A Survey of Opportunities, Challenges, and Applications. IEEE Cloud Comput., 10(1), pp. 22-31, 2023.

[9] N. Kratzke, Cloud-Native Computing: How to Design, Build, and Run Applications in the Cloud. Packt Publishing, 2022.
[10] Z. Li, & T. Kijewski-Correa, Performance Benchmarking of Serverless Computing Platforms for Data-Intensive Applications. 2024 IEEE Int. Conf. Cloud Eng. (IC2E), pp. 89-96.
[11] OWASP Foundation. (2024). OWASP Serverless Top 10 Risks.
[12] S. Gupta, & P. Kumar, Security Threats and Defense Mechanisms in Serverless Computing: A Survey. Computers & Security, vol. 126, 103069, 2023.
[13] A. Perez, Zero Trust Architecture in Cloud Environments. CRC Press, 2023.
[14] A. Shostack. Threat Modeling: A Practical Guide for Development Teams. Wiley, 2022.
[15] R. Copeland, Cloud Native Python: Building Scalable Applications with AWS Lambda. Packt Publishing, 2023.
[16] Locust.io. Locust Documentation: Distributed Load Generation, 2025.
[17] PostgreSQL Global Development Group. PostgreSQL 16 Documentation: Connection Pooling and Performance, 2024.
[18] P. Vorobets, A. Horpenyuk, I. Opirskyy, Preparing IT Infrastructure for the Quantum Age: Post-Quantum SSH // CEUR Workshop Proc., 2025, vol. 4016, In: Proceedings of the Workshop classic, quantum, and post-quantum cryptography co-located with International conference on problems of infocommunications. Science and technology (PICST 2025) Kyiv, Ukraine, August 5, 2025, p. 66-75.
[19] A. Horpenyuk, M. Horpenyuk, N. Luzhetska, O. Horpenyuk, A. Desiatko, Development and Research of a Non-Periodic Pseudorandom Number Generator // CEUR Workshop Proc., 2025, vol. 4016, In: Proceedings of the Workshop classic, quantum, and post-quantum cryptography co-located with International conference on problems of infocommunications. Science and technology (PICST 2025) Kyiv, Ukraine, August 5, 2025, p. 1-12.
[20] O. Harasymchuk et al., Modern methods of ensuring information protection in cybersecurity systems using artificial intelligence and blockchain technology: collective monograph, Technology Center PC, Kharkiv, 2025. doi:10.15587/978-617-8360-12-2.
[21] O. Harasymchuk, et al., Intelligent cyber defence systems: detection of ransomware and protection of wireless networks based on artificial intelligence technologies: collective monograph, Technology Center PC, Kharkiv, 2025. doi:10.15587/978-617-8360-22-1.
[22] O. Sapsai, Y. Martseniuk, A. Partyka, O. Harasymchuk, Research on automated security incident management in public cloud environments, CEUR Workshop Proc., 2025, 226-249.
[23] V. Susukailo, et al., Cybercrimes investigation via honeypots in cloud environments, CEUR Workshop Proc., 2923 (2021), 91-96.
[24] Y. Martseniuk, A. Partyka, I. Opirskyy, O. Harasymchuk, Cost observability as a security control in multi-cloud environments based on SOC 2 security standard, Comput. Secur., 161 (2026), 104771. doi:10.1016/j.cose.2025.104771.
[25] Y. Martseniuk, A. Partyka, O. Harasymchuk, V. Cherevyk, N. Dovzhenko, Research of the Centralized Configuration Repository Efficiency for Secure Cloud Service Infrastructure Management, CEUR Workshop Proc., 3991 (2025) 260-274.