# Serverless Computing with AWS Lambda: Best Practices for Scalable Enterprise Applications

**Ritesh Kumar**

Independent Researcher
New Jersey, USA
ritesh2901@gmail.com

**Abstract**

**Serverless computing has transformed cloud application development by enabling organizations to build scalable, event-driven applications without managing infrastructure. AWS Lambda, a leading Function-as-a-Service (FaaS) platform, allows developers to run code in response to events while AWS handles provisioning, scaling, and execution. This paper presents best practices for event-driven applications using AWS Lambda, emphasizing performance optimization, security, cost efficiency, and observability. Key areas include cold start mitigation, concurrency management, IAM security, API Gateway integration, and compliance automation. A real-world use case demonstrates how AWS Lambda can be used for real-time compliance enforcement by leveraging AWS services such as S3, DynamoDB, SNS, and Step Functions. Intended for enterprise architects, DevOps engineers, and developers, this paper provides actionable insights to achieve higher performance, stronger security, and lower operational costs in serverless architectures.**

**Keyword: Serverless computing, AWS Lambda, Function-as-a-Service (FaaS), Cloud security, Compliance automation**

## I. INTRODUCTION

### A. *What is Serverless Computing?*

Serverless computing is a cloud-native execution model in which cloud providers dynamically manage resources for executing applications, eliminating the need for provisioning or maintaining infrastructure [1]. Unlike traditional computing models, where applications run on pre-allocated servers or containers, serverless functions execute on demand in response to events, scaling automatically based on workload.

#### 1) *Key Principles of Serverless Computing:*

- *Event-Driven Execution:* Code is triggered by events such as API requests, file uploads, database updates, or scheduled tasks.
- *Automatic Scaling*: The cloud provider manages scaling, allocating compute power dynamically based on incoming requests.
- *No Infrastructure Management:* Developers do not need to provision or maintain servers, reducing operational overhead.
- *Pay-as-You-Go Pricing:* Costs are based on actual execution time and resources used rather than pre-allocated infrastructure.
- *Statelessness:* Each function execution is independent, requiring external storage (e.g., databases or object storage) for state persistence.

*2) Comparison with Traditional Architectures*

Serverless computing is gaining traction across industries, providing **cost savings**, **operational efficiency**, and **simplified development workflows** for organizations adopting cloud-native architectures.

| **Feature** | **Traditional Servers** | **Containers** | **Serverless (FaaS)** |
|---|---|---|---|
| Provisioning | Manual | Automated | Fully managed by cloud provider |
| Scalability | Limited by server capacity | Scales with orchestration tools (e.g., Kubernetes) | Auto-scales instantly |
| Maintenance | Requires OS and patch updates | Requires container updates | No maintenance required |
| Cost Model | Pay for provisioned capacity | Pay for running containers | Pay only for execution time |
| Execution Time | Persistent | Long-running | Short-lived (often limited to minutes) |

***TABLE 1. COMPARISON OF TRADITIONAL SERVERS, CONTAINERS AND SERVERLESS***

*B. Evolution of Serverless Computing*

Serverless computing has evolved as a natural extension of cloud computing, driven by the need for better resource utilization and reduced operational complexity.

*1) Key Milestones in Serverless Adoption:*

*a)* ***2014****:* AWS introduces Lambda, allowing event-driven function execution without provisioning servers [2].

*b)* ***2015–2016****:* Adoption grows as AWS expands integration with services like API Gateway, DynamoDB, and S3.

*c)* ***2017****:* Other cloud providers enter the space (Azure Functions, Google Cloud Functions). AWS adds support for more languages (Go, Ruby, .NET Core).

*d)* ***2018****:* AWS introduces Lambda Layers for better code reuse and AWS Step Functions for orchestrating serverless workflows.

*2) Enterprise Adoption Trends*

*a) Serverless-first applications:* More organizations are designing applications natively for serverless instead of migrating existing workloads.

*b) Hybrid architectures:* Many companies combine serverless with containers and traditional computing to balance flexibility and control.

c) *Security and compliance focus:* As adoption grows, enterprises are implementing IAM, encryption, and VPC integration to protect serverless workloads.

### C. *Why AWS Lambda?*

AWS Lambda is **one of the most mature and widely adopted** serverless computing platforms, enabling developers to execute code **without managing servers**

Key features of AWS Lambda:

1) *Fully Managed Infrastructure:* No need to provision or maintain servers.
2) *Automatic Scaling:* Functions scale instantly with workload demand.
3) *Cost Efficiency: Pay only for execution time (billed per millisecond).*
4) *Deep AWS Integration:* Works seamlessly with AWS services like S3, DynamoDB, API Gateway, SQS, and SNS.
5) *Multi-Language Support:* Currently supports Node.js, Python, Java, Go, .NET Core, and Ruby.
6) *Security & Compliance:* Supports IAM roles, encryption, and VPC networking for enterprise security.

### D. *Scope of the Paper*

This paper explores **best practices for designing scalable, secure, and cost-efficient serverless applications** using AWS Lambda.

1) *Event-Driven Application Design:* Architectural patterns and AWS service integrations.
2) *Performance Optimization:* Cold start mitigation, concurrency management, and efficient monitoring.
3) *Security Best Practices*: IAM roles, VPC networking, multi-tenancy considerations.
4) *Cost Optimization Strategies:* Reducing execution time, optimizing dependencies, and managing scaling.
5) *Real-World Use Case*: Implementing a compliance automation system using AWS Lambda.

| AWS Service | Integration Type | Use Case Example |
|---|---|---|
| Amazon API Gateway | HTTP Trigger | Creating RESTful APIs with Lambda as a backend |
| Amazon S3 | Event Driven | Processing file uploads, image resizing, data transformation |
| Amazon DynamoDB | Streams | Real-time data processing and change tracking |
| Amazon SNS & SQS | Messaging | Asynchronous event-driven architectures |
| AWS Step Functions | Orchestration | Managing multi-step workflows and function chaining |
| CloudWatch Event | Scheduled Execution | Running periodic tasks, scheduled compliance checks |

**TABLE 2. AWS LAMBDA INTEGRATION SERVICES**

## II. AWS LAMBDA: KEY CONCEPTS AND COMPONENTS

### A. How AWS Lambda Works

AWS Lambda is an **event-driven, serverless computing service** that allows developers to execute code **without provisioning or managing infrastructure**. Lambda functions are triggered by events from various AWS services, scaling automatically to handle incoming requests.

#### 1) Execution Model

*a) Event-Driven Architecture:* AWS Lambda is designed to respond to events from sources such as API Gateway, S3, DynamoDB, SNS, and SQS.

*b) Stateless Function Execution:* Each invocation is independent, meaning no persistent state is maintained across function executions.

*c) Ephemeral Compute:* Lambda functions run in a short-lived environment, executing for **up to 15 minutes per invocation**.

*d) Automatic Scaling:* AWS automatically provisions and deallocates compute resources based on demand, ensuring functions scale up or down as needed.

#### 2) Lambda Invocation Process

*a) An event triggers the Lambda function.*

*b) AWS allocates a compute environment (if a new execution environment is needed).*

*c) Lambda loads the function code and dependencies.*

*d) The function executes the code and returns a response.*

*e) The execution environment is frozen (kept warm for reuse) or deallocated after inactivity.*

#### 3) Supported Languages:

*AWS Lambda supports multiple programming languages – Node.js (Javascript), Python, Java, Go, .NET Core (C#), Ruby*

### B. Integration with AWS Services

AWS Lambda integrates seamlessly with various AWS services to create **fully managed, event-driven applications** [7].

### C. AWS Lambda Execution Environment

AWS Lambda executes functions in a **managed runtime environment** optimized for short-lived workloads.

#### 1) Compute Resources for Lambda Execution

*a) Memory Allocation: Configurable from 128MB to 3GB (affects CPU allocation).*

*b) Execution Timeout:* Up to 15 minutes per function invocation.

*c) Ephemeral Storage: Functions have 512MB of temporary storage in /tmp directory.*

*d) Networking: Functions can run publicly or inside an Amazon VPC for private network access.*

*2) Invocation Models*

*AWS Lambda functions can be **invoked synchronously or asynchronously**, depending on the use case.*

| Invocation Type | Description | Use Case Example |
|---|---|---|
| Synchronous | The caller waits for Lambda to return a response. | API requests via API Gateway, real-time applications. |
| Asynchronous | The event is queued, and Lambda processes it later. | SNS notifications, S3 event triggers. |
| Stream-based | Lambda polls event streams and processes records in batches. | DynamoDB Streams, Kinesis Data Streams. |

**TABLE 3. AWS LAMBDA INVOCATION MODELS AND USE CASES**

## III. BEST PRACTICES FOR DESIGNING SCALABLE EVENT-DRIVEN APPLICATIONS USING AWS LAMBDA

AWS Lambda enables organizations to build **scalable, event-driven applications** with minimal operational overhead. However, to maximize **performance, reliability, and cost efficiency**, it is essential to follow best practices when designing serverless applications. This section explores **event-driven architecture patterns, cold start mitigation strategies, concurrency management, and cost optimization techniques**.

### *A. Event-Driven Architecture Patterns*

AWS Lambda follows an **event-driven model**, where functions are executed in response to specific triggers [3], [10]. This architecture allows applications to be **highly scalable and loosely coupled**, improving maintainability and fault tolerance.

*1) Common Event-Driven Patterns in AWS Lambda Applications*

| Pattern | Description | AWS Service Used |
|---|---|---|
| API Backend | Lambda functions handle HTTP requests, replacing traditional web servers | API Gateway + Lambda |
| Event Processing | Functions trigger based on real-time events like file uploads or database updates. | S3, DynamoDB Streams, SNS, SQS |
| Data Streaming | Lambda functions consume and process streaming data in near real-time. | Kinesis, DynamoDB Streams |
| Workflow Automation | Lambda functions execute sequential or parallel tasks using managed workflows. | AWS Step Functions |

**TABLE 4. EVENT-DRIVEN PATTERNS IN AWS LAMBDA APPLICATIONS**

### B. *Performance Optimization Techniques*

#### 1) *Cold Start Mitigation Strategies*

A **cold start** occurs when AWS **initializes a new execution environment** for a Lambda function, increasing the function's response time. Cold starts are more noticeable for functions triggered **infrequently** or running inside a **VPC** [4], [6], [8].

Strategies to Reduce Cold Starts:

*a) Keep Lambda Functions Warm:* Use scheduled invocations (CloudWatch Events) to periodically trigger functions and prevent execution environments from being shut down [5].

*b) Optimize Deployment Package Size:* Smaller packages reduce initialization time and remove unnecessary dependencies and use AWS Lambda Layers for shared code.

*c) Choose the Right Runtime:* Node.js and Python generally have lower cold start times compared to Java and .NET Core.

*d) Avoid VPC Where Possible:* Running a function inside a VPC increases cold start latency due to Elastic Network Interface (ENI) provisioning.

#### 2) *Concurrency and Scaling Considerations*

AWS Lambda **automatically scales** by launching multiple execution environments to handle concurrent requests [1]. However, improper concurrency settings can **affect performance and cost**.

Concurrency Controls in AWS Lambda

*a) Unreserved Concurrency (Default):* Lambda scales dynamically **without a fixed limit**, potentially consuming all available AWS account resources.

*b) Reserved Concurrency:* Ensures that a specific function has **guaranteed execution capacity**, preventing starvation from other functions.

*c) Provisioning Parallel Executions:* For high-volume event processing, **use SNS or SQS** to distribute workloads across multiple Lambda functions.

Best Practices for Managing Concurrency

*a) Use SQS Queues to smooth out traffic spikes and prevent throttling.*

*b) Monitor CloudWatch metrics (e.g., ConcurrentExecutions, Throttles) to detect scaling issues.*

*c) Limit execution time to reduce resource contention and improve throughput.*

### C. *Cost Optimization Strategies*

AWS Lambda follows a **pay-per-use pricing model**, where costs are determined by:

a) Execution Time (Billed per 1ms of execution).
b) Memory Allocation (Higher memory = higher cost).
c) Number of Requests (Each invocation incurs a small cost).

Best Practices for Cost Optimization

#### 1) *Optimize Execution Time*

- Reduce function duration by avoiding unnecessary computations [12].
- Use asynchronous processing (SNS, SQS) instead of waiting for synchronous operations.

*2) Optimize Memory Allocation*

- Higher memory allocations increase CPU power (AWS assigns proportional CPU).
- Use benchmark testing to find the optimal memory setting (e.g., 512MB vs. 1024MB).

*3) Reduce Deployment Package Size*

- Use Lambda Layers for shared dependencies instead of bundling them inside each function.
- Use compiled dependencies (e.g., precompiled Python packages) to speed up execution.

*4) Use Compute Savings Plans (If Available)*

- AWS Compute Savings Plans allow reserved pricing for predictable workloads, reducing overall Lambda costs.

## IV. SECURITY CONSIDERATIONS FOR AWS LAMBDA

Security is a critical aspect of designing **serverless applications**. While AWS Lambda **manages infrastructure security**, developers must implement **proper access controls, data encryption, and network security** to protect serverless workloads. This section outlines best practices for **IAM roles, API Gateway security, VPC integration, and multi-tenant security considerations**.

### *A. IAM Roles & Least Privilege Access*

AWS Identity and Access Management (IAM) controls **who can invoke Lambda functions** and what AWS services they can access. **Overly permissive IAM roles** increase the risk of security breaches [3], [9].

**Best Practices for IAM Role Management**

*1) Follow the Principle of Least Privilege (PoLP)*

- Assign only the necessary permissions to Lambda functions [3].
- Avoid wildcard (”*”) permissions in IAM policies.

*2) Use Separate IAM Roles for Different Lambda Functions*

- Do NOT use a single IAM role for all functions—each function should have a unique role with specific permissions.

*3) Restrict Cross-Account Execution*

- If functions must be invoked from another AWS account, use resource-based policies to limit access.

*4) Secure Environment Variables*

- Store sensitive data (e.g., API keys, credentials) in AWS Systems Manager Parameter Store or AWS Secrets Manager, not directly in environment variables.

### B. *API Gateway Security Best Practices*

When using **API Gateway to expose Lambda functions**, it is essential to implement **authentication, authorization, and rate limiting** to prevent unauthorized access.

Best Practices for Securing API Gateway with AWS Lambda

#### 1) *Use AWS IAM for Internal APIs*

- For internal applications, require IAM authentication to restrict API Gateway access to authorized AWS users.

#### 2) *Use Amazon Cognito for Public APIs*

- Amazon Cognito User Pools enable secure authentication for web and mobile apps.
- Supports OAuth2, OpenID Connect, and JWT-based authentication [4].

#### 3) *Enable Request Validation and Throttling*

- Validate request parameters, headers, and payloads to prevent malformed requests.
- Set API Gateway rate limits to prevent abuse.

### C. *VPC Integration and Network Security*

By default, AWS Lambda functions run in a secure AWS-managed environment, but some applications require private network access. Running Lambda inside a VPC allows access to:

a) Private AWS resources (e.g., RDS, ElastiCache).
b) On-premises systems via AWS Direct Connect or VPN.

Challenges of Running Lambda in a VPC

a) Higher cold start times due to Elastic Network Interface (ENI) provisioning.
b) Limited internet access (requires NAT Gateway or VPC Endpoints for outbound connections).

Best Practices for Secure VPC Integration

c) Minimize Lambda functions inside VPC unless private network access is required.
d) Use VPC Endpoints to connect securely to AWS services without internet access.
e) Restrict security groups to only allow necessary traffic.

### D. *Data Security and Encryption*

AWS Lambda processes **sensitive data** in many applications, requiring proper encryption and storage security.

Best Practices for Data Protection

#### 1) *Enable Encryption for Data at Rest and in Transit*

- Use AWS KMS to encrypt S3, DynamoDB, and environment variables.
- Ensure HTTPS (TLS 1.2+) for API Gateway requests.

#### 2) *Avoid Hardcoding Secrets*

- Store credentials in AWS Secrets Manager instead of Lambda environment variables.

3) *Monitor Data Access Logs*

- Enable CloudTrail logging to track data access events.

### E. *Multi-Tenant Security Considerations*

For SaaS applications running on AWS Lambda, **multi-tenancy** introduces additional security risks, including **data leakage between tenants**.

Best Practices for Tenant Isolation

1) *Use Separate IAM Policies for Each Tenant*

- Define fine-grained IAM permissions per tenant.
- Avoid cross-tenant access in API Gateway authorizers.

2) *Implement Data Segmentation*

- Use DynamoDB partition keys or separate S3 buckets per tenant.
- Encrypt tenant-specific data with separate KMS keys.

3) *Limit Function Execution Per Tenant*

- Use reserved concurrency limits to prevent one tenant from consuming all resources.

## V. REAL-WORLD USE CASE: IMPLEMENTING A REAL-TIME COMPLIANCE AUTOMATION SYSTEM USING AWS LAMBDA

### A. *Business Challenge*

Enterprises operating in **regulated industries** such as **finance, healthcare, and manufacturing** must comply with various standards like **HIPAA, GDPR, and ISO 27001**. Ensuring compliance requires **continuous monitoring, real-time policy enforcement, and automated audits**.

1) *Key Compliance Challenges*

a) *Manual compliance checks lead to delays and human errors.*
b) *Traditional infrastructure is costly and difficult to scale for real-time event processing.*
c) *Security risks arise from misconfigured resources or unauthorized data access.*

2) *Objective*

To design a **real-time compliance monitoring system** using **AWS Lambda** that can:

a) *Automate compliance enforcement across cloud resources.*
b) *Detect non-compliant events in real-time and take corrective actions.*
c) *Reduce infrastructure costs while maintaining high availability.*

### B. *Solution Architecture*

The **compliance automation system** is designed as an **event-driven serverless application**, leveraging AWS Lambda for **real-time detection, logging, and remediation** of non-compliance incidents.

High-Level Architecture:

1) *Event Detection:*
   - AWS resources generate events (e.g., S3 object creation, IAM policy changes, API Gateway requests).
   - AWS CloudTrail, Config, and S3 event notifications trigger compliance checks.
2) *Processing with AWS Lambda:*
   - AWS Lambda validates compliance policies.
   - Non-compliant events are logged in DynamoDB for auditing.
   - Remediation actions are triggered via SNS notifications.
3) *Alerting & Reporting:*
   - AWS SNS sends real-time alerts for critical compliance violations.
   - AWS Step Functions orchestrate multi-step compliance enforcement workflows.

AWS Services Used in the Architecture:

| AWS Service | Purpose |
|---|---|
| AWS Lambda | Processes compliance events, enforces policies |
| Amazon S3 | Stores compliance-related audit logs |
| AWS Config | Monitors AWS resources for non-compliance |
| Amazon DynamoDB | Stores compliance check results |
| Amazon SNS | Sends alerts for violations |
| AWS Step Functions | Orchestrates automated remediation workflows |

**TABLE *5*. AWS SERVICES USED IN THE COMPLIANCE AUTOMATION ARCHITECTURE**

### C. *Implementation Steps*

1) *Step 1: Setting Up S3-Triggered Compliance Checks*
   - Configure S3 Event Notifications to trigger Lambda when files are uploaded.
   - Lambda function checks for required encryption settings and access control policies.
   - If non-compliant, the function logs the event and notifies administrators.

2) *Step 2: Enforcing IAM Policy Compliance with AWS Config*
   - AWS Config continuously monitors IAM policy changes.
   - When a non-compliant IAM policy is detected, AWS Lambda:
   - Logs the violation in DynamoDB.
   - Triggers an SNS alert to security teams.
   - Automatically removes excessive permissions if configured.

3) *Step 3: Storing Compliance Logs in DynamoDB*

- Lambda writes compliance violations to DynamoDB for auditing.
- Queries can retrieve historical compliance reports.

4) *Step 4: Sending Real-Time Alerts with SNS*
- When non-compliance is detected, Lambda publishes a message to an SNS topic.
- Security teams receive email/SMS alerts for high-risk violations.

5) *Step 5: Automating Remediation with Step Functions*
- AWS Step Functions can orchestrate multi-step remediation workflows.

### D. *Result & Benefits*

1) *Scalability*
- AWS Lambda scales dynamically to handle large compliance workloads.
- No need for dedicated servers, reducing operational complexity.

2) *Cost Efficiency*
- The pay-per-use model eliminates costs associated with idle resources.
- Compliance checks run only when triggered, minimizing expenses.

3) *Security & Compliance*
- Real-time violation detection and auto-remediation reduce compliance risks.
- AWS-native security tools (IAM, KMS, Config, SNS) enhance protection.

### E. *Monitoring, Debugging, and Observability*

1) *Logging & Monitoring with CloudWatch*
- Track function execution time, errors, and throttling metrics.
- Set CloudWatch Alarms for error rate detection.

2) *Distributed Tracing with AWS X-Ray*
- Visualize function execution across multiple AWS services.

3) *Handling Failures*
- Use Dead Letter Queues (DLQs) for failed asynchronous executions.
- Implement custom retry policies to improve fault tolerance.

## VI. CONCLUSION & FUTURE OUTLOOK

### A. *Key Takeaways*

AWS Lambda has revolutionized **event-driven computing** by enabling developers to build **highly scalable, cost-efficient, and fully managed** applications without provisioning or maintaining infrastructure. However, designing an **optimized, secure, and reliable** serverless application requires careful planning.

Summary of Best Practices

1) *Scalable Event-Driven Design*
- Use API Gateway, S3, DynamoDB, SNS, and SQS to create modular, event-driven architectures.
- Leverage AWS Step Functions for workflow automation.

2) *Performance Optimization*
- Reduce cold start impact by warming up functions and minimizing package size.

- Optimize memory allocation to balance speed and cost efficiency.

3) *Security and Compliance*
   - Follow least privilege IAM policies and restrict Lambda execution roles.
   - Secure data using AWS KMS encryption and VPC integration when necessary.
   - Implement tenant isolation strategies in multi-tenant SaaS applications.
4) *Observability & Debugging*
   - Use CloudWatch Logs & Metrics to monitor Lambda performance [11].
   - Implement AWS X-Ray tracing to identify bottlenecks.
   - Handle failures with DLQs and custom retry policies.
5) *Cost Optimization*
   - Reduce execution time by optimizing code efficiency and avoiding unnecessary dependencies.
   - Manage concurrency limits to prevent excessive Lambda execution costs.

By following these best practices, organizations can build **robust, secure, and efficient AWS Lambda applications** that scale seamlessly and optimize cloud costs.

### B. *Future Trends in Serverless Computing*

1) ***Lower Cold Start Latency*** *through improved runtime performance [7].*
2) ***Advanced Monitoring & AI-driven Observability*** *tools.*
3) ***Increased Enterprise Adoption*** *for* ***compliance and security automation****.*

## REFERENCES

[1] Amazon Web Services, Inc., AWS Lambda Developer Guide, Seattle, WA, USA: Amazon Web Services, 2019. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html DOI: 10.5555/3279552.

[2] Amazon Web Services, Inc., Serverless Architectures with AWS Lambda, AWS Whitepaper, Seattle, WA, USA: Amazon Web Services, 2019. [Online]. Available: https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf.

[3] Amazon Web Services, Inc., Security Best Practices for AWS Lambda, AWS Whitepaper, Seattle, WA, USA: Amazon Web Services, 2019. [Online] Available: https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html.

[4] Amazon Web Services, Inc., Building Scalable Applications with AWS Lambda, AWS Architecture Blog, Seattle, WA, USA, 2019. [Online] Available: https://aws.amazon.com/blogs/architecture/building-scalable-serverless-applications-with-aws-lambda/.

[5] G. Cavo, P. Patel, and J. Sundaresan, "Event-Driven Architectures with AWS Lambda," *AWS re:Invent Conference*, Las Vegas, NV, USA, Dec. 2018. [Online]. Available: https://reinvent.awsevents.com/.

[6] M. Roberts, "Serverless Architectures," ThoughtWorks, 2018. [Online]. Available: https://martinfowler.com/articles/serverless.html.

[7] AWS Step Functions Documentation, AWS, 2018. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html.

[8] A. Singh, "Optimizing AWS Lambda cold start performance," *AWS Compute Blog*, Mar. 2019. [Online]. Available: https://aws.amazon.com/blogs/compute/optimizing-aws-lambda-cold-start-performance/.

[9] Amazon Web Services, Inc., *AWS Security Best Practices for IAM Roles and Policies*, AWS Whitepaper, Seattle, WA, USA, 2019. [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html.

[10] M. Fowler, "Serverless Architectures: Design Patterns and Best Practices," *Martin Fowler Blog*, 2019. [Online]. Available: https://martinfowler.com/articles/serverless.html.

[11] Amazon Web Services, Inc., *Monitoring and Observability for AWS Lambda Applications*, AWS Whitepaper, Seattle, WA, USA, 2019. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/monitoring.html.

[12] Amazon Web Services, Inc., "Best Practices for Performance Optimization in AWS Lambda," *AWS Compute Blog*, Mar. 2019. [Online]. Available: https://aws.amazon.com/blogs/compute/best-practices-for-working-with-aws-lambda-functions/.