Journal of Information Systems Engineering and Management
2022, 7(3)

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Cloud-Native Java Application Development on AWS: Architectural Patterns for EC2, Lambda, and RDS Integration

Prem Reddy Nomula
Northwestern Polytechnic University (alias San Francisco Bay University), USA

ARTICLE INFO

Received: 02 Aug 2022

Accepted: 28 Sept 2022

ABSTRACT

Cloud-native paradigms have significantly transformed enterprise software development by emphasizing scalability, resilience, and rapid deployment. Java continues to be a dominant programming language in enterprise systems, supported by a mature ecosystem and evolving frameworks that align with modern cloud environments. This paper examines architectural patterns for developing cloud-native Java applications on Amazon Web Services (AWS), focusing on the integration of Elastic Compute Cloud (EC2), AWS Lambda, and Amazon Relational Database Service (RDS). The study analyzes multiple architectural approaches, including monolithic deployments, microservices-based designs, serverless computing models, and hybrid architectures that combine these paradigms. It further explores key integration strategies, performance and scalability considerations, and cost optimization techniques relevant to distributed cloud systems. Special attention is given to challenges such as Java runtime performance in serverless environments, database connectivity in stateless architectures, and the coordination of synchronous and asynchronous components. Based on the analysis, a hybrid architectural model is proposed that leverages the strengths of EC2 for long-running services, Lambda for event-driven processing, and RDS for managed data persistence. The findings demonstrate that a balanced integration of these services enables the development of flexible, scalable, and cost-efficient applications. The paper provides a structured foundation for designing robust cloud-native Java systems that meet contemporary enterprise requirements.

Keywords: Cloud-native computing, Java, AWS, EC2, Lambda, RDS, microservices, serverless

1. Introduction

The rapid evolution of cloud computing has fundamentally reshaped the landscape of software engineering, particularly in the design and deployment of enterprise-grade applications. Traditional monolithic systems, once dominant in on-premise environments, are increasingly being replaced by cloud-native architectures that emphasize modularity, scalability, and resilience. This transition is driven by the need for systems that can dynamically adapt to fluctuating workloads, ensure high availability, and support continuous delivery practices in highly competitive digital ecosystems [1].

Journal of Information Systems Engineering and Management
2022, 7(3)

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Cloud-native computing represents more than a shift in infrastructure; it reflects a transformation in architectural philosophy. It promotes the decomposition of applications into loosely coupled microservices, each aligned with specific business capabilities. These services are typically deployed in distributed environments and communicate through lightweight protocols such as RESTful interfaces or messaging systems. This approach enhances fault isolation, enabling systems to remain operational even when individual components fail, while also supporting independent scaling and rapid deployment cycles [2].

Within this evolving paradigm, Java continues to play a central role in enterprise application development. Its sustained relevance is attributed to a mature ecosystem, extensive community support, and a wide range of frameworks that facilitate modern application design. Technologies such as Spring Boot and MicroProfile have simplified the development of lightweight, production-ready services, while advancements in the Java Virtual Machine have improved performance and resource efficiency in containerized and distributed environments [3]. These developments have enabled Java to remain competitive in cloud-native contexts despite the emergence of alternative programming languages.

Amazon Web Services (AWS) provides a comprehensive platform that supports the development and deployment of cloud-native applications. Among its offerings, Elastic Compute Cloud (EC2), AWS Lambda, and Amazon Relational Database Service (RDS) represent key components for building scalable systems. EC2 provides virtualized computing resources with full control over the runtime environment, making it suitable for stateful and long-running services. AWS Lambda introduces a serverless execution model in which code is executed in response to events, eliminating the need for infrastructure management. Amazon RDS complements these services by offering managed relational database solutions that handle operational tasks such as backups, scaling, and maintenance [4].

The integration of these services within a single application architecture introduces both opportunities and challenges. Combining EC2 and Lambda enables a hybrid approach that balances control and abstraction, allowing workloads to be distributed according to their execution characteristics. However, this integration also requires careful consideration of communication patterns, data consistency, and system security. For instance, ensuring efficient database access from stateless Lambda functions to RDS instances necessitates optimized connection management strategies, while maintaining low latency across distributed services demands thoughtful system design.

Furthermore, the increasing adoption of event-driven architectures has amplified the role of serverless computing in cloud-native systems. Event-driven models enable asynchronous processing, decouple system components, and improve scalability. At the same time, they introduce challenges related to monitoring, debugging, and coordination across distributed services. As a result, system architects must carefully evaluate the trade-offs associated with different architectural approaches.

This paper is motivated by the need to provide a structured analysis of how Java-based applications can effectively leverage AWS services to achieve cloud-native characteristics. While existing studies often focus on individual paradigms such as microservices or serverless computing, there is limited consolidated guidance on integrating EC2, Lambda, and RDS within a unified architectural framework.

The objectives of this study are threefold. First, it aims to identify and evaluate architectural patterns for deploying Java applications on AWS using EC2, Lambda, and RDS. Second, it analyzes the trade-offs associated with different integration strategies, particularly in terms of performance, scalability, and cost. Third, it proposes a hybrid reference architecture that synthesizes these insights into a practical model for real-world implementation.

In summary, this introduction establishes the foundation for understanding the convergence of cloud-native principles, Java application development, and AWS service integration. The subsequent

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

sections of this paper explore the underlying concepts, architectural patterns, and practical considerations necessary for designing robust and scalable cloud-native Java systems.

2. Background and Related Work

2.1 Cloud-Native Architecture

Cloud-native architecture represents a significant departure from traditional monolithic and service-oriented architectures by prioritizing scalability, resilience, and rapid delivery. At its core, the cloud-native paradigm is built upon three foundational pillars: microservices, containerization, and DevOps practices. These elements collectively enable systems to be modular, portable, and continuously evolving.

Microservices architecture decomposes applications into a collection of small, autonomous services, each aligned with a specific business capability. Unlike monolithic systems where components are tightly coupled, microservices operate independently and communicate via lightweight protocols such as RESTful APIs or message brokers. This independence facilitates selective scaling, allowing only the services under heavy load to scale, thereby optimizing resource utilization [2]. Furthermore, microservices improve fault isolation; failures in one service are less likely to cascade across the system.

Containerization technologies, particularly Docker, have become integral to cloud-native systems by encapsulating application code along with its dependencies into portable units. Containers ensure consistency across development, testing, and production environments, thereby reducing deployment-related issues. Orchestration platforms such as Kubernetes automate the deployment, scaling, and management of containerized applications, further enhancing system resilience and operational efficiency.

DevOps practices complement microservices and containerization by fostering collaboration between development and operations teams. Continuous Integration and Continuous Deployment (CI/CD) pipelines automate code integration, testing, and deployment processes, enabling rapid and reliable software releases. Infrastructure as Code (IaC) tools, such as AWS CloudFormation and Terraform, allow infrastructure provisioning to be managed programmatically, ensuring repeatability and reducing configuration drift.

Despite these advantages, cloud-native architectures introduce challenges related to service coordination, distributed data management, and observability. Techniques such as service meshes, centralized logging, and distributed tracing have emerged to address these complexities, highlighting the evolving nature of cloud-native ecosystems.

2.2 Evolution of Java in the Cloud

Java has undergone substantial transformation to remain relevant in cloud-native environments. Historically criticized for its relatively high memory footprint and slower startup times, Java has adapted through both language-level improvements and ecosystem innovations.

Frameworks such as Spring Boot have simplified Java application development by providing opinionated configurations, embedded servers, and rapid bootstrapping capabilities. This has enabled developers to build standalone, production-ready microservices with minimal configuration. Similarly, newer frameworks like Micronaut and Quarkus have been designed specifically for cloud and serverless environments, focusing on reduced memory consumption and faster startup times by leveraging ahead-of-time (AOT) compilation techniques.

The Java Virtual Machine (JVM) has also evolved to better support cloud-native workloads. Enhancements in garbage collection algorithms, just-in-time (JIT) compilation, and resource

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

management have improved performance in containerized environments. Additionally, the introduction of container awareness in the JVM allows it to respect resource limits defined by container orchestrators, preventing issues such as memory over-allocation.

Another significant development is the integration of Java with containerization technologies. Java applications are now commonly packaged as Docker images and deployed in orchestrated environments, aligning with modern DevOps practices. Tools such as Jib and Buildpacks further streamline container image creation without requiring deep knowledge of Dockerfiles.

Moreover, the emergence of GraalVM has enabled Java applications to be compiled into native executables, drastically reducing startup time and memory usage. This advancement is particularly relevant for serverless platforms like AWS Lambda, where execution efficiency directly impacts cost and performance [3].

Overall, Java's adaptability and continuous innovation have ensured its sustained relevance in distributed, cloud-native systems.

2.3 AWS Compute and Database Services

Amazon Web Services (AWS) provides a comprehensive ecosystem of cloud services that support diverse architectural paradigms. Among these, EC2, Lambda, and RDS form a foundational triad for building cloud-native Java applications, each representing a distinct abstraction level in cloud computing.

Elastic Compute Cloud (EC2) operates within the Infrastructure-as-a-Service (IaaS) model, offering virtualized computing resources with full control over the operating system, runtime environment, and application stack. This flexibility makes EC2 suitable for stateful applications, legacy system migration, and workloads requiring custom configurations. However, it also imposes responsibility for system maintenance, including patching, scaling, and monitoring.

AWS Lambda, as a Function-as-a-Service (FaaS) offering, abstracts infrastructure management entirely. Developers deploy discrete functions that are executed in response to events, such as HTTP requests, file uploads, or message queue triggers. Lambda automatically scales based on demand, enabling efficient handling of bursty or unpredictable workloads. Nevertheless, its stateless execution model and limitations on execution time necessitate careful application design, particularly for complex or long-running processes.

Amazon Relational Database Service (RDS) provides managed relational database solutions that simplify database administration tasks. By automating backups, patching, replication, and scaling, RDS allows developers to focus on application logic rather than database maintenance. It supports multiple database engines, including MySQL, PostgreSQL, and Oracle, making it versatile for enterprise applications.

The integration of these services enables the construction of hybrid architectures that leverage the strengths of each model. For instance, EC2 can host persistent application services, while Lambda handles event-driven tasks, and RDS manages structured data storage. Research indicates that combining IaaS and FaaS paradigms can lead to optimized resource utilization and cost efficiency, particularly when workloads exhibit varying execution patterns [4].

However, this integration introduces challenges such as network latency, data consistency, and orchestration complexity. Effective solutions often involve asynchronous communication mechanisms, caching strategies, and robust monitoring systems. Additionally, architectural decisions must consider trade-offs between control (offered by EC2) and abstraction (provided by Lambda), as well as the scalability constraints of relational databases.

Journal of Information Systems Engineering and Management
2022, 7(3)

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

In summary, AWS services provide a flexible foundation for cloud-native Java development, but their effective utilization requires a nuanced understanding of their capabilities and limitations.

3. Architectural Patterns

3.1 Monolithic Java Applications on EC2

The monolithic deployment model represents one of the earliest and most straightforward approaches to migrating Java applications to the cloud. In this pattern, a complete Java application—often built using frameworks such as Spring MVC or traditional Java EE—is deployed as a single deployable unit on an Amazon EC2 instance. This approach closely mirrors on-premise deployments, making it particularly attractive for organizations seeking a low-risk entry point into cloud environments.

From an architectural standpoint, all functional modules—such as user management, business logic, and data access—are tightly integrated within a single codebase and runtime. The EC2 instance provides full administrative control over the operating system, middleware, and runtime environment, allowing teams to replicate existing configurations with minimal changes. This level of control is especially beneficial for applications with strict compliance requirements or custom dependencies.

One of the primary advantages of this pattern lies in its simplicity. Deployment pipelines are easier to implement, debugging is more straightforward due to centralized logging, and system behavior is more predictable. Furthermore, legacy enterprise systems that were not originally designed for distributed environments can be migrated without extensive refactoring.

However, the monolithic model presents significant limitations in the context of cloud-native requirements. Scalability is constrained because the entire application must be replicated to handle increased load, even if only specific components are under stress. This results in inefficient resource utilization. Additionally, the tight coupling of components reduces agility, making it difficult to introduce changes without impacting the entire system. Maintenance overhead also increases over time, as updates and patches require redeploying the entire application.

Consequently, while monolithic deployments on EC2 serve as a practical transitional architecture, they are generally not considered optimal for long-term cloud-native strategies.

3.2 Microservices Architecture with EC2 and RDS

The microservices architectural pattern addresses many of the limitations inherent in monolithic systems by decomposing applications into smaller, independently deployable services. In the context of AWS, these services are typically hosted on multiple EC2 instances and interact with a centralized or distributed database managed by Amazon RDS.

Each microservice encapsulates a specific business capability and operates as an autonomous unit, often exposing its functionality through RESTful APIs. This modularity allows development teams to build, test, deploy, and scale services independently, significantly improving development velocity and operational flexibility. For example, a payment service can be scaled independently of a user authentication service, optimizing resource allocation.

The use of RDS in this architecture provides reliable and managed data persistence. Depending on the design, services may share a common database or maintain isolated schemas to ensure loose coupling. The latter approach aligns more closely with microservices principles, as it minimizes dependencies between services and reduces the risk of cascading failures.

A key strength of this pattern is fault isolation. Failures in one microservice are less likely to impact others, enhancing system resilience. Additionally, microservices facilitate the adoption of polyglot

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

architectures, where different services can be implemented using different technologies, although Java often remains the dominant choice in enterprise environments.

Despite these benefits, the microservices approach introduces considerable complexity. Inter-service communication requires careful design, often involving service discovery mechanisms, API gateways, and load balancers. Ensuring data consistency across services becomes challenging, particularly in distributed transactions. Patterns such as eventual consistency and saga orchestration are commonly employed to address these issues [5].

Operational challenges also arise in monitoring and debugging distributed systems. Tools for centralized logging, distributed tracing, and health monitoring become essential components of the architecture. Thus, while microservices provide scalability and flexibility, they demand a higher level of architectural maturity and operational discipline.

3.2.1 Containerized EC2 Deployments with ECS and EKS

In addition to traditional virtual machine-based deployments, containerization has become a widely adopted approach for running Java applications on EC2. While EC2 provides full control over the compute environment, managing dependencies, scaling, and deployment consistency can be complex in large-scale systems. Containerization addresses these challenges by packaging applications and their dependencies into isolated, portable units that can be deployed consistently across environments.

Docker has emerged as the standard tool for containerizing Java applications, particularly those developed using frameworks such as Spring Boot. By encapsulating the runtime environment, libraries, and configuration, Docker ensures consistency between development, testing, and production stages. This approach reduces environment-related issues and simplifies deployment workflows.

To manage containerized workloads at scale, AWS provides orchestration services such as Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS). These services extend the capabilities of EC2 by enabling automated deployment, scaling, and management of containerized applications.

Amazon ECS is a fully managed container orchestration service that integrates closely with EC2. It allows developers to deploy and manage containerized Java applications without handling complex orchestration infrastructure. ECS supports features such as service auto scaling, load balancing, and integration with AWS Identity and Access Management, making it suitable for organizations seeking a streamlined container management solution.

Amazon EKS provides a managed Kubernetes environment, enabling organizations to deploy containerized Java applications using standard Kubernetes tools and practices. This approach is particularly beneficial for teams requiring portability across different cloud platforms or those already familiar with Kubernetes-based workflows. EKS supports advanced deployment strategies, including rolling updates and blue-green deployments, as well as integration with service mesh technologies for enhanced observability and traffic management.

The integration of EC2 with ECS and EKS allows organizations to combine the flexibility of virtual machines with the scalability of container orchestration platforms. In this model, EC2 instances act as the underlying compute layer, while ECS or EKS manages container scheduling, scaling, and networking.

From a Java perspective, containerization improves resource utilization, reduces startup overhead, and supports modern development practices such as continuous integration and continuous deployment. However, adopting container orchestration introduces additional complexity, including cluster management, monitoring, and security configuration.

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Overall, containerized deployments on EC2 using ECS and EKS provide a scalable and efficient approach for managing Java applications in cloud-native environments, enabling organizations to balance operational control with platform-level automation.

3.3 Serverless Java Applications Using Lambda

Serverless computing represents a paradigm shift in application deployment, where developers focus solely on writing code while the cloud provider manages the underlying infrastructure. AWS Lambda exemplifies this model by enabling the execution of Java functions in response to events without requiring server provisioning or management.

In this pattern, applications are decomposed into discrete functions that are triggered by events such as HTTP requests (via API Gateway), file uploads (via Amazon S3), or messages from queues (via Amazon SQS). Each function is stateless and executes independently, allowing for highly scalable and loosely coupled systems.

The primary advantage of serverless architecture is its ability to scale automatically in response to demand. Unlike EC2-based systems, which require explicit scaling configurations, Lambda dynamically allocates resources based on incoming requests. This makes it particularly suitable for workloads with unpredictable or bursty traffic patterns. Additionally, the pay-per-use pricing model ensures cost efficiency, as charges are incurred only for actual execution time.

However, the use of Java in serverless environments introduces specific challenges. One of the most notable is cold start latency, where the initialization time of the Java runtime environment can lead to delays in function execution. This issue is more pronounced compared to lighter runtimes such as Node.js or Python. Techniques such as reducing application size, optimizing dependencies, and using provisioned concurrency can help mitigate these delays [6].

Another challenge is the inherently stateless nature of Lambda functions. Developers must externalize state management to services such as RDS, DynamoDB, or caching layers. This requirement necessitates careful design to ensure efficient data access and minimize latency. Additionally, debugging and monitoring serverless applications can be more complex due to their distributed and ephemeral nature.

Despite these challenges, serverless architectures offer significant advantages in terms of scalability, cost efficiency, and reduced operational overhead, making them an attractive option for modern cloud-native applications.

3.3.1 Lambda Cold Start Optimization for Java

One of the most significant challenges in adopting AWS Lambda for Java-based applications is cold start latency. A cold start occurs when a Lambda function is invoked after a period of inactivity, requiring the cloud platform to initialize a new execution environment. For Java, this initialization includes starting the Java Virtual Machine (JVM) and loading application dependencies, which can result in higher latency compared to lighter runtimes such as Node.js or Python [6].

Cold start latency directly impacts user experience, particularly in latency-sensitive applications such as APIs and real-time systems. Therefore, mitigating this issue became a key focus for enterprise teams.

Several strategies have been developed to address cold start challenges:

- **Provisioned Concurrency:** This feature allows a predefined number of Lambda execution environments to remain initialized and ready to handle incoming requests. By eliminating the need for runtime initialization during invocation, provisioned concurrency significantly reduces response latency. However, it introduces additional cost, as resources are kept active regardless of usage.

Journal of Information Systems Engineering and Management
2022, 7(3)

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

- **GraalVM Native Image Compilation:** GraalVM enables Java applications to be compiled into native executables, eliminating the need for JVM startup during execution. This results in faster startup times and reduced memory consumption. Frameworks such as Quarkus and Micronaut have adopted this approach to optimize Java applications for serverless environments.
- **AWS Lambda SnapStart:** AWS Lambda SnapStart, announced at AWS, provides a significant advancement in reducing cold start latency for Java functions. It works by capturing a snapshot of a fully initialized execution environment and reusing it for subsequent invocations. This approach reduces startup time without requiring major code changes, making it particularly attractive for existing Java applications.

Each of these techniques presents trade-offs between cost, complexity, and performance. Provisioned concurrency ensures predictable latency but increases operational expenses. GraalVM requires changes in build and deployment processes, while SnapStart offers a balanced solution with minimal configuration overhead.

In practice, selecting an appropriate strategy depends on application requirements. For latency-critical services, a combination of provisioned concurrency and SnapStart may be used, whereas cost-sensitive workloads may rely on optimized code and lightweight frameworks to minimize cold start impact.

3.4 Hybrid Architecture: EC2, Lambda, and RDS

The hybrid architectural model combines the strengths of EC2, Lambda, and RDS to create a flexible and efficient system capable of handling diverse workload requirements. Rather than relying on a single compute paradigm, this approach strategically distributes responsibilities across different AWS services based on their respective strengths.

In a typical hybrid architecture, EC2 instances are used to host long-running, stateful services such as core business APIs. These services benefit from the stability and configurability of virtualized environments. Meanwhile, AWS Lambda is employed for event-driven and asynchronous tasks, such as background processing, data transformation, or notification handling. RDS serves as the centralized data store, providing reliable and managed relational database capabilities.

To better understand the trade-offs between different AWS services used in cloud-native Java architectures, Table 1 presents a comparative analysis of EC2, AWS Lambda, and Amazon RDS.

Table 1: Comparison of AWS Services in Cloud-Native Java Architecture

Service	Use Case	Pros	Cons
EC2	Hosting long-running, stateful Java applications (e.g., Spring Boot APIs, microservices)	Full control over environment; supports custom JVM tuning; suitable for legacy migration; stable performance	Requires infrastructure management; slower scaling compared to serverless; potential idle cost
AWS Lambda	Event-driven, short-lived, stateless functions (e.g., background jobs, async processing)	Automatic scaling; pay-per-use pricing; no server management; ideal for burst workloads	Cold start latency (especially for Java); stateless execution; limited execution time; concurrency limits
Amazon RDS	Relational data storage for transactional applications (e.g., user data, orders)	Managed service (backups, patching); supports multiple database engines; high availability options	Limited flexibility compared to self-managed DB; scaling constraints for write-heavy workloads; connection overhead with serverless

Journal of Information Systems Engineering and Management
2022, 7(3)

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

The interaction between these components follows a well-defined workflow. Client requests are initially handled by APIs hosted on EC2, which may perform synchronous operations and return immediate responses. For tasks that do not require immediate completion, the system offloads processing to Lambda functions, often through messaging services such as SQS. These functions execute asynchronously and interact with RDS to store or retrieve data as needed.

This division of responsibilities enables the system to achieve both performance and cost efficiency. EC2 ensures consistent performance for critical services, while Lambda provides elasticity for variable workloads. The use of RDS simplifies data management while maintaining transactional integrity. As shown in Figure 1, the hybrid architecture integrates EC2, AWS Lambda, and Amazon RDS to support both synchronous and asynchronous processing.

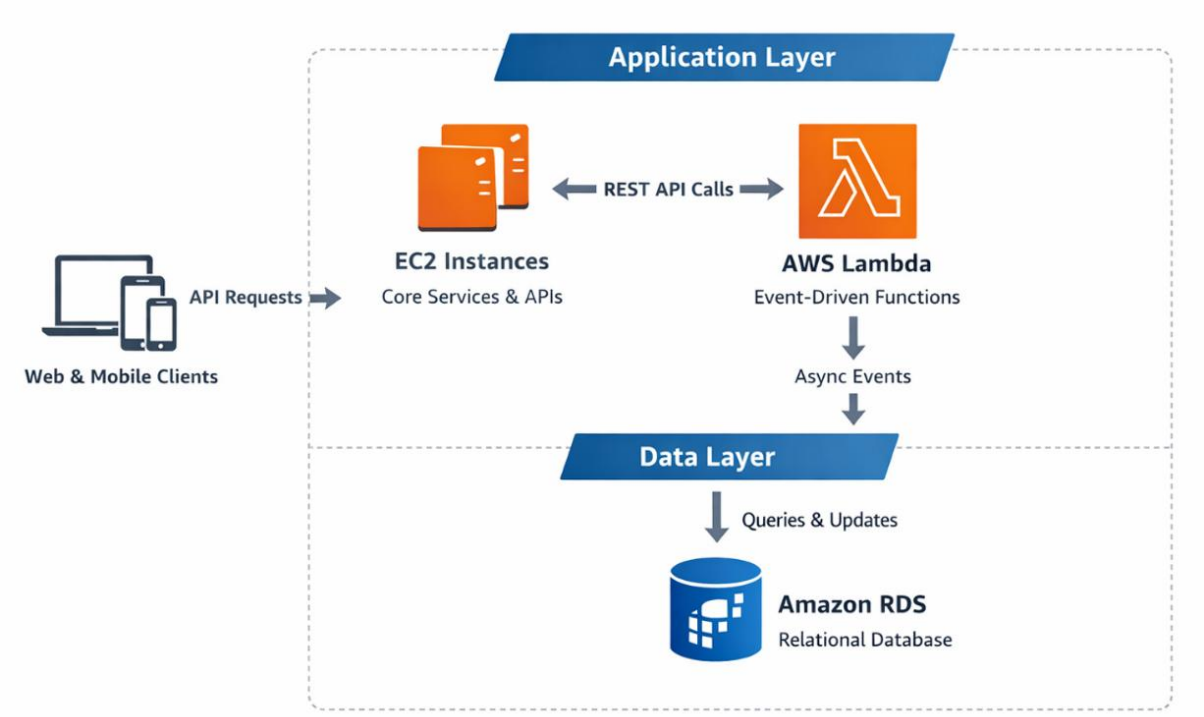

Figure 1: System architecture of a cloud-native Java application integrating EC2, AWS Lambda, and Amazon RDS.

4. Integration Strategies

4.1 Database Integration with RDS

Database integration is a fundamental component of cloud-native Java applications, especially when using Amazon Relational Database Service (RDS) as the primary data storage system. Java applications typically connect to RDS using Java Database Connectivity (JDBC) or higher-level frameworks such as Hibernate and the Java Persistence API (JPA). These frameworks simplify database interaction by allowing developers to work with objects instead of writing complex SQL queries, which improves development speed and code maintainability.

In EC2-based environments, database connection pooling is commonly managed using libraries such as HikariCP or Apache DBCP. These tools maintain a pool of reusable database connections, which reduces the overhead of repeatedly opening and closing connections. Proper configuration of

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

parameters such as pool size, timeout settings, and connection lifetime is essential to ensure efficient performance.

When integrating RDS with AWS Lambda, additional challenges arise. Lambda functions are short-lived and do not maintain persistent connections, which can lead to frequent creation of new database connections. Under high load, this behavior can overwhelm the database and increase response time.

To address this issue, solutions such as Amazon RDS Proxy are used. RDS Proxy acts as an intermediary that manages database connections efficiently, reducing the number of direct connections to the database and improving overall performance [7]. Another approach is to design applications in a way that minimizes direct database interactions, such as batching operations or using caching mechanisms.

Advanced strategies such as read replicas and database partitioning can also be applied to improve performance and distribute load. However, these techniques must be carefully implemented to maintain data accuracy and consistency.

4.1.1 Spring Boot as the Integration Layer

Spring Boot plays a central role in unifying the different components of a cloud-native Java application. It acts as the integration layer that connects compute services, messaging systems, and data storage while providing a consistent programming model across environments.

One of the key advantages of Spring Boot is its support for dependency injection, which simplifies the management of application components. In EC2-based deployments, Spring's inversion of control container manages the lifecycle of services, repositories, and configuration classes, enabling modular and maintainable system design. When extending this model to serverless environments, special consideration is required to align dependency injection with the execution lifecycle of AWS Lambda functions. Frameworks such as Spring Cloud Function allow developers to adapt Spring-managed beans into Lambda-compatible handlers, ensuring that business logic remains reusable across both server-based and serverless deployments.

Spring Cloud AWS further enhances integration by providing abstractions for interacting with AWS services. It simplifies access to resources such as Amazon RDS, messaging services, and configuration management by reducing the need for low-level API calls. This allows developers to focus on application logic while leveraging managed cloud services through familiar Spring programming patterns.

Configuration management is another critical aspect where Spring Boot provides significant value. Externalized configuration enables applications to adapt to different environments without code changes. Properties can be managed through environment variables, configuration files, or centralized services, ensuring flexibility and consistency across deployments. This approach is particularly important in distributed systems where services run across EC2 instances and Lambda functions.

In addition, Spring Boot supports integration with secure credential management practices. Database credentials and service access configurations can be externalized and managed through secure mechanisms rather than being hardcoded within the application. This aligns with best practices for cloud-native security and enhances system maintainability.

Overall, Spring Boot serves as the foundational layer that binds together EC2-based services, serverless functions, and database interactions. By providing a unified development model, it enables consistent application behavior across different execution environments while simplifying integration, configuration, and lifecycle management.

Journal of Information Systems Engineering and Management
2022, 7(3)

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

4.2 API Management and Event Handling

API management plays a central role in connecting different components of a cloud-native system. AWS API Gateway is commonly used to expose RESTful APIs and acts as the main entry point for client requests. It provides features such as request routing, rate limiting, authentication, and monitoring.

In Java-based applications, API Gateway typically forwards requests to backend services running on EC2 or triggers Lambda functions for processing. For real-time operations, requests are handled synchronously by EC2-based services, while background or non-urgent tasks are processed asynchronously using Lambda functions. Event-driven communication is further supported by services such as Amazon Simple Queue Service (SQS) and Amazon Simple Notification Service (SNS), enabling decoupled and scalable system design.

As illustrated in Figure 2, the request flow follows a client-to-service interaction pattern in which API Gateway routes incoming requests to AWS Lambda functions, which then interact with Amazon RDS for data processing and persistence.

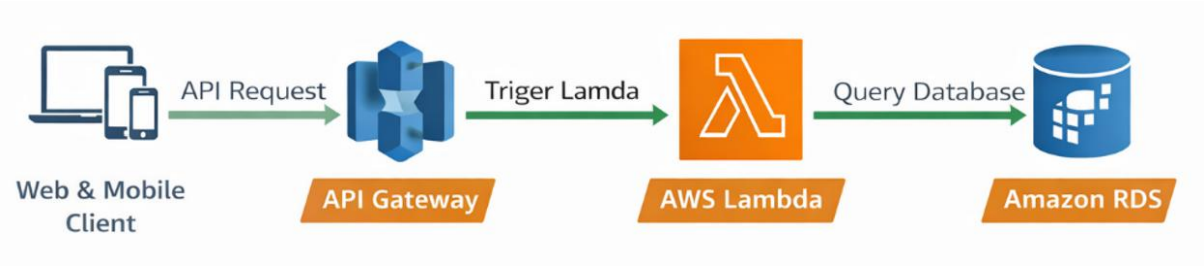

Figure 2: Request flow diagram illustrating client interaction through API Gateway to AWS Lambda and Amazon RDS.

4.3 Security and Access Control

Security is a critical aspect of cloud-native application design, particularly when multiple services interact across distributed environments. AWS provides several tools and services to ensure secure access and data protection.

AWS Identity and Access Management (IAM) is used to define permissions for users and services. In a Java application, IAM roles are assigned to EC2 instances and Lambda functions, allowing them to access other AWS services securely. The principle of least privilege is followed, meaning each component is given only the permissions it needs to perform its tasks.

Network security is managed using Virtual Private Clouds (VPCs), which create isolated environments for application resources. Within a VPC, resources can be placed in separate subnets to control access. For example, public-facing services can be placed in public subnets, while databases such as RDS are placed in private subnets. Security groups and network access rules are used to control traffic between these components.

Data protection is ensured through encryption. AWS supports encryption for data stored in RDS and for data transmitted between services. Secure communication protocols such as HTTPS and TLS are used to protect data during transmission.

Monitoring and auditing tools such as AWS CloudTrail and Amazon CloudWatch provide visibility into system activity. These tools help detect unusual behavior, support compliance requirements, and improve system security.

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Despite the availability of strong security features, incorrect configurations can still lead to vulnerabilities. Therefore, automated security checks and continuous monitoring are essential to maintain a secure cloud-native environment.

5. Performance and Scalability

Performance and scalability are central considerations in the design of cloud-native Java applications. The ability of a system to handle increasing workloads efficiently while maintaining acceptable response times is a defining characteristic of cloud-native architectures. In AWS-based environments, scalability is achieved through a combination of compute elasticity, event-driven execution, and database optimization. This section examines how EC2, Lambda, and RDS contribute to system performance and the strategies required to balance their capabilities.

5.1 EC2 Scaling

Amazon EC2 provides flexible scaling capabilities that allow applications to dynamically adjust compute resources in response to workload variations. The primary mechanism for achieving this is through Auto Scaling Groups (ASGs), which automatically increase or decrease the number of EC2 instances based on predefined policies and performance metrics.

Scaling policies can be configured using various triggers, such as CPU utilization, memory usage, network traffic, or custom application-level metrics. For instance, when CPU utilization exceeds a specified threshold, additional instances are launched to distribute the load. Conversely, when demand decreases, instances are terminated to reduce costs. This elasticity ensures that applications maintain consistent performance without over-provisioning resources.

Load balancing is typically integrated with Auto Scaling through services such as Elastic Load Balancer (ELB), which distributes incoming traffic across multiple EC2 instances. This not only improves performance but also enhances fault tolerance by ensuring that failed instances are automatically replaced and traffic is rerouted to healthy nodes.

However, EC2 scaling is not instantaneous. Instance provisioning and initialization can introduce latency, particularly during sudden traffic spikes. To mitigate this, predictive scaling and pre-warmed instances are often employed. Additionally, stateless application design is recommended to facilitate horizontal scaling, as it allows any instance to handle any request without dependency on local state.

From a Java perspective, performance optimization on EC2 involves tuning JVM parameters, optimizing thread management, and leveraging caching mechanisms such as Redis or in-memory caches. These optimizations ensure efficient resource utilization and minimize response times under high load.

5.2 Lambda Scaling

AWS Lambda introduces a fundamentally different scaling model based on automatic concurrency management. Unlike EC2, where scaling must be explicitly configured, Lambda functions scale automatically in response to incoming events. Each request triggers a separate execution environment, allowing the system to handle a virtually unlimited number of concurrent invocations.

This model is particularly advantageous for workloads with unpredictable or bursty traffic patterns. For example, an e-commerce platform experiencing sudden spikes during promotional events can rely on Lambda to scale instantly without prior provisioning. The pay-per-execution pricing model further enhances cost efficiency by charging only for actual usage.

Despite these benefits, Lambda scaling is subject to concurrency limits, which define the maximum number of simultaneous executions allowed per account or function. Exceeding these limits results in

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

throttling, where additional requests are delayed or rejected. Therefore, careful monitoring and configuration of concurrency quotas are essential to ensure reliable performance.

Cold start latency is another important consideration, especially for Java-based Lambda functions. When a function is invoked after a period of inactivity, AWS must initialize a new execution environment, leading to increased response times. Techniques such as provisioned concurrency, which keeps a pool of pre-initialized environments ready, can significantly reduce cold start delays.

Furthermore, efficient resource utilization in Lambda requires minimizing function execution time and memory usage. This can be achieved by optimizing code, reducing dependency size, and leveraging lightweight frameworks. In high-throughput systems, combining Lambda with event-driven services like SQS allows for controlled scaling and improved resilience.

5.3 RDS Scaling

While compute services such as EC2 and Lambda can scale horizontally with relative ease, database systems often present a scalability bottleneck in distributed architectures. Amazon RDS addresses this challenge by offering both vertical and horizontal scaling mechanisms, although each comes with trade-offs.

Vertical scaling involves increasing the computational capacity of a database instance by upgrading to a more powerful instance type with greater CPU, memory, and storage capabilities. This approach is straightforward and does not require significant architectural changes. However, it has inherent limits and may involve downtime during instance resizing.

Horizontal scaling is achieved through the use of read replicas, which allow read operations to be distributed across multiple database instances. This significantly improves performance for read-heavy workloads, as queries can be offloaded from the primary instance. In Java applications, read/write splitting can be implemented at the application or middleware level to direct queries appropriately.

Despite these capabilities, maintaining data consistency across replicas introduces complexity. Replication lag can lead to stale data being returned in read operations, which may not be acceptable for certain applications. Additionally, write operations must still be handled by the primary instance, limiting scalability for write-intensive workloads.

To address these limitations, caching strategies are often employed to reduce database load. Technologies such as Amazon ElastiCache (Redis or Memcached) can store frequently accessed data in memory, significantly improving response times and reducing the number of database queries.

Connection management is another critical factor in RDS performance. As discussed earlier, excessive connections from distributed services—particularly Lambda functions—can overwhelm the database. Solutions such as connection pooling and RDS Proxy help mitigate this issue by efficiently managing connections.

In highly distributed systems, architectural patterns such as database sharding and eventual consistency may be adopted to further enhance scalability. However, these approaches require careful design to ensure data integrity and application correctness.

In summary, achieving optimal performance and scalability in cloud-native Java applications requires a holistic approach that considers the strengths and limitations of EC2, Lambda, and RDS. While compute services can scale dynamically to meet demand, database systems often require additional strategies to prevent bottlenecks. By combining appropriate scaling techniques, caching mechanisms, and architectural patterns, developers can build systems that are both responsive and resilient under varying workloads.

Journal of Information Systems Engineering and Management
2022, 7(3)

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

6. Cost Optimization

Cost optimization is a key advantage of cloud-native systems, as cloud platforms provide flexible pricing models that allow organizations to align expenses with actual usage. In AWS-based Java applications, effective cost management requires a strategic combination of compute, storage, and database services, along with careful architectural decisions.

One of the most significant cost benefits comes from the use of AWS Lambda. Since Lambda follows a pay-per-execution model, organizations are charged only for the compute time consumed during function execution. This makes it highly cost-effective for intermittent, event-driven, or unpredictable workloads. For example, tasks such as file processing, notifications, or background jobs can be handled by Lambda without maintaining continuously running servers, thereby eliminating idle resource costs.

In contrast, Amazon EC2 operates on a provisioned resource model, where instances run continuously regardless of utilization. While this may appear less cost-efficient, AWS offers pricing options such as Reserved Instances and Savings Plans, which provide significant discounts in exchange for long-term commitments. These options are particularly suitable for stable workloads that require consistent performance, such as core application services and APIs. By carefully selecting instance types and commitment plans, organizations can achieve predictable and optimized pricing.

Amazon RDS also contributes to overall cost efficiency, but it requires thoughtful configuration. Choosing the appropriate database instance size based on workload requirements is essential to avoid over-provisioning. Storage optimization, including the selection of storage types and allocation of capacity, further impacts cost. Features such as automated backups, multi-zone deployments, and read replicas improve reliability and performance but may increase expenses if not used judiciously.

A hybrid architecture that combines EC2, Lambda, and RDS often provides the best cost-performance balance. In such a model, long-running and stable services are hosted on EC2, while variable or burst workloads are handled by Lambda. This ensures that resources are used efficiently without unnecessary overhead. Additionally, techniques such as auto scaling, caching, and efficient database design contribute to reducing operational costs.

Cost monitoring tools such as AWS Cost Explorer and billing alerts play an important role in maintaining financial control. Continuous evaluation of resource usage and optimization of configurations are necessary to sustain cost efficiency over time [8].

7. Case Study

To illustrate the practical application of the discussed architectural patterns, this section presents a case study of an enterprise retail system that transitioned from a traditional monolithic architecture to a hybrid cloud-native model using AWS services.

7.1 Initial Architecture

The original system was a monolithic Java application deployed on a single or limited number of servers. All functionalities, including user management, product catalog, order processing, and payment handling, were tightly integrated within a single codebase. While this design was sufficient for moderate workloads, it presented several challenges:

- Limited scalability during high traffic periods such as seasonal sales
- Difficulty in deploying updates without affecting the entire system
- High maintenance overhead and reduced flexibility

Journal of Information Systems Engineering and Management
2022, 7(3)

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

7.2 Migration Strategy

To address these limitations, the organization adopted a phased migration approach toward a cloud-native architecture. The new system was designed using a hybrid model that integrates EC2, Lambda, and RDS.

- Core APIs on EC2: The main application interfaces, including user authentication and product management, were deployed on EC2 instances. This ensured stable performance and allowed for controlled scaling using Auto Scaling Groups.
- Event-Driven Processing with Lambda: Order processing, notification services, and background tasks were migrated to AWS Lambda. These functions were triggered by events such as order placement or payment confirmation, enabling asynchronous processing and improved responsiveness.
- Data Persistence with RDS:All transactional data, including customer information and order details, was stored in Amazon RDS. The database was configured with read replicas to handle increased read traffic and improve performance.

7.3 Outcomes and Benefits

The transition to a hybrid cloud-native architecture resulted in several measurable improvements:

- Improved Scalability: The system was able to handle peak demand periods more effectively. EC2 instances scaled to manage user requests, while Lambda automatically handled spikes in background processing tasks.
- Reduced Infrastructure Costs: By shifting intermittent workloads to Lambda and optimizing EC2 usage with reserved pricing, the organization significantly reduced operational expenses.
- Enhanced System Resilience: The separation of components into independent services improved fault isolation. Failures in one part of the system did not disrupt the entire application, leading to higher overall availability.
- Faster Deployment Cycles: The modular architecture enabled independent updates and deployments, reducing downtime and accelerating the release of new features.

8. Discussion

The integration of Amazon EC2, AWS Lambda, and Amazon RDS forms a powerful foundation for building cloud-native Java applications. However, the combination of these services introduces important trade-offs that must be carefully evaluated during system design. While each service provides distinct advantages, their interaction within a distributed architecture adds complexity in areas such as performance, maintainability, and operational control.

One of the primary considerations is the balance between control and abstraction. EC2 offers a high degree of control over the computing environment, allowing developers to configure operating systems, runtime environments, and application settings. This level of control is essential for applications with specific performance requirements or legacy dependencies. However, it also increases operational responsibility, as tasks such as patching, scaling, and monitoring must be managed explicitly.

In contrast, AWS Lambda provides a highly abstracted, serverless execution model that eliminates the need for infrastructure management. This significantly reduces operational overhead and enables rapid scaling in response to demand. However, this convenience comes at the cost of reduced control over the execution environment. Limitations such as execution time constraints, statelessness, and

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

cold start latency can impact application design and performance, particularly for Java-based workloads.

Amazon RDS further contributes to the trade-off between simplicity and flexibility. As a managed database service, RDS simplifies administrative tasks such as backups, patching, and replication. This allows development teams to focus on application logic rather than database maintenance. However, compared to self-managed databases, RDS may impose restrictions on customization, tuning, and advanced configurations. Additionally, as a centralized relational database, it can become a performance bottleneck in highly distributed systems if not properly optimized.

Another important aspect is system complexity. Hybrid architectures that combine EC2, Lambda, and RDS require careful coordination between synchronous and asynchronous components. Designing efficient communication patterns, managing data consistency, and ensuring reliable error handling are non-trivial tasks. Distributed systems also demand advanced monitoring and debugging tools to maintain visibility across multiple services.

From a performance perspective, achieving optimal results requires a balanced distribution of workloads. Long-running and stateful services are best suited for EC2, while short-lived and event-driven tasks benefit from Lambda. RDS must be carefully scaled and optimized to handle concurrent access from multiple services. Failure to align workloads with the appropriate service can lead to inefficiencies and increased costs.

Looking forward, emerging technologies offer opportunities to further enhance cloud-native architectures. Container orchestration platforms such as Kubernetes provide greater flexibility in managing containerized applications, enabling more consistent deployment across environments. Similarly, serverless database solutions and distributed data stores are gaining attention as alternatives to traditional relational databases, offering improved scalability and reduced operational complexity.

Future research should focus on improving interoperability between different cloud services, optimizing performance in hybrid environments, and developing standardized architectural frameworks. Additionally, advancements in Java runtime technologies and cloud-native frameworks are expected to further reduce the gap between traditional and serverless computing models.

9. Conclusion

Cloud-native Java application development on AWS represents a convergence of mature programming practices and modern cloud technologies. The effective use of services such as EC2, AWS Lambda, and Amazon RDS enables the creation of scalable, resilient, and cost-efficient systems capable of meeting the demands of contemporary enterprise applications.

This study has examined key architectural patterns, integration strategies, and performance considerations associated with these services. It highlights that no single architectural approach is universally optimal; rather, the selection of an appropriate design depends on specific application requirements, including workload characteristics, performance expectations, and cost constraints.

The hybrid architecture, which combines EC2 for stable and long-running services, Lambda for event-driven and scalable workloads, and RDS for managed data persistence, emerges as a practical and balanced solution. This approach leverages the strengths of each service while mitigating their individual limitations, enabling efficient distribution of responsibilities across the system.

Furthermore, the adoption of best practices such as efficient resource utilization, dynamic scaling strategies, secure access control, and robust monitoring mechanisms is essential for the success of

e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

cloud-native systems. These practices ensure that applications not only perform efficiently under varying workloads but also maintain reliability and security in distributed environments.

In conclusion, the design of cloud-native Java applications on AWS requires careful architectural planning and continuous optimization. By applying established principles and integration strategies, organizations can develop robust and adaptable systems capable of supporting evolving digital services and business requirements. As cloud technologies continue to advance, the ability to effectively integrate multiple computing paradigms will remain a critical factor in achieving long-term system efficiency and scalability.

References

[1] Mell, P., & Grance, T. (2011). The NIST definition of cloud computing. National Institute of Standards and Technology.
[2] Pahl, C. (2015). Containerization and the PaaS cloud. IEEE Cloud Computing, 2(3), 24–31.
[3] Walls, C. (2021). Spring Boot in Action (2nd ed.). Manning Publications.
[4] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., ... & Suter, P. (2017). Serverless computing: Current trends and open problems. Research Advances in Cloud Computing, 1–20.
[5] Newman, S. (2021). Building Microservices (2nd ed.). O'Reilly Media.
[6] Spillner, J. (2017). Serverless computing: State-of-the-art and research challenges. IEEE Cloud Computing, 4(4), 66–71.
[7] Fowler, M. (2018). Patterns of Enterprise Application Architecture. Addison-Wesley.
[8] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., ... & Zaharia, M. (2010). A view of cloud computing. Communications of the ACM, 53(4), 50–58.
[9] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In Present and Ulterior Software Engineering (pp. 195–216). Springer.
[10] Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., & Casallas, R. (2015). Infrastructure cost comparison of running web applications in the cloud using AWS Lambda and monolithic and microservice architectures. IEEE/ACM CCGrid.
[11] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C. C., Khandelwal, A., Pu, Q., ... & Stoica, I. (2019). Cloud programming simplified: A Berkeley view on serverless computing.
[12] Wang, L., Li, M., Zhang, Y., Ristenpart, T., & Swift, M. (2018). Peeking behind the curtains of serverless platforms. USENIX ATC.
[13] Eivy, A. (2017). Be wary of the economics of "serverless" cloud computing. IEEE Cloud Computing, 4(2), 6–12.
[14] Zhang, Q., Chen, M., Li, L., & Chen, Y. (2018). Cost-effective scheduling of cloud workflows using spot instances. Future Generation Computer Systems, 86, 1020–1032.
[15] Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: A systematic mapping study. IEEE Cloud Computing, 5(6), 56–65.
[16] Kratzke, N., & Quint, P. (2017). Understanding cloud-native applications after 10 years of cloud computing. Journal of Systems and Software, 126, 1–16.
[17] Leitner, P., Cito, J., & Stöckli, E. (2019). Modelling and managing deployment costs of microservice-based cloud applications. IEEE Transactions on Services Computing.
[18] Shafiei, M., Khonsari, A., & Derakhshi, H. (2019). Performance modeling of cloud applications using queuing theory. Journal of Cloud Computing, 8(1).
[19] Adzic, G., & Chatley, R. (2017). Serverless computing: Economic and architectural impact. ACM SIGPLAN Notices, 52(1), 884–889.
[20] Castro, P., Ishakian, V., Muthusamy, V., & Slominski, A. (2019). The rise of serverless computing. Communications of the ACM, 62(12), 44–54.