AWS App Runner vs. AWS Fargate vs. Lambda Containers: A Comparison for Running Containers Without Servers

Dylan O'Donnell
Information Technology Management
SETU
County Carlow
C00272192@setu.ie

Abstract*—Using containers and serverless computing is growing fast. This gives developers several choices on cloud platforms like Amazon Web Services (AWS) for running their applications without managing the servers underneath. Picking the best service can be tricky. This report looks at three key AWS services for running containers without servers: AWS App Runner, AWS Fargate (used with both Elastic Container Service - ECS and Elastic Kubernetes Service - EKS), and AWS Lambda using container images. We compare how each service is built, how it works day-to-day, how it scales, its networking options, and how much it costs. We point out the best uses for each service and discuss the trade-offs. The aim is to give developers and architects a straightforward way to choose the right AWS serverless container service for their application's needs, their preferred way of working, and their budget. This comparison shows the differences in how easy each service is to use, how flexible it is, how it performs, and the effort needed to run it.

***Keywords*—Cloud Computing, AWS, Serverless, Containers, AWS App Runner, AWS Fargate, AWS Lambda, Microservices, Platform-as-a-Service, Container Orchestration, ECS, EKS.**

INTRODUCTION I

Modern software development relies heavily on containerisation, using tools like Docker to package applications and their dependencies [1]. This simplifies deployment and ensures consistency across different environments. Additionally, serverless computing, popularised by AWS Lambda's Function-as-a-Service (FaaS) model, offered the promise of running code without managing any underlying infrastructure [2] These two powerful trends inevitably merged, leading to the demand for running containerised applications with the same operational ease as serverless functions. This gave rise to "serverless containers" – technologies that execute containers without requiring users to provision, manage, or patch servers [3].

Amazon Web Services (AWS), as a leading cloud platform, responded to this demand by offering multiple services in this space. AWS App Runner was introduced as a highly abstracted service, aiming for maximum simplicity – developers provide source code or a container image, and AWS handles everything else to run it as a web service [4] .AWS Fargate provides the serverless compute capacity for AWS's mainstream container orchestrators, ECS and EKS, removing the need for users to manage the worker nodes within their clusters [5]. More recently, AWS Lambda was enhanced to support running functions packaged as container images, offering more flexibility for dependencies and runtimes within the established event-driven model [2]

This variety of options presents a significant choice for developers and architects. Each service has different strengths, weaknesses, and operational models. Selecting the most appropriate service is crucial for balancing ease of use, cost, performance, and control. Misunderstanding the trade-offs can lead to inefficient architectures or unexpected operational burdens. This report aims to clarify these choices by comparing App Runner, Fargate, and Lambda Containers. The motivation is to equip cloud practitioners with the knowledge to make informed decisions when deploying containerised applications on AWS without managing servers.

Our contribution lies in a focused analysis comparing these three specific services across key technical dimensions:

1. We detail and contrast their underlying system architectures and operational models.
2. We explore suitable use cases and scenarios for each, providing practical examples.
3. We discuss the critical trade-offs, covering aspects like ease of use, flexibility, performance, networking, cost, and ecosystem integration.

This report is based on publicly available AWS documentation, technical whitepapers, and standard industry practices.

LITERATURE REVIEW II

The evolution towards serverless containers builds upon several foundational cloud computing concepts. Initially, cloud adoption was driven by Infrastructure-as-a-Service (IaaS), where providers like AWS offered virtual machines (e.g., EC2) [6]. This provided flexibility but retained the burden of operating system management, patching, and scaling. Platform-as-a-Service (PaaS) offerings emerged to simplify deployment by abstracting the underlying infrastructure further, but often imposed limitations on runtime environments and control [7].

Containerisation, primarily popularised by Docker, provided a lightweight alternative to full virtualisation. Containers package an application and its dependencies, ensuring consistency ("it works on my machine") and improving resource density compared to VMs. However, managing containers at scale efficiently requires Container Orchestration Engines (COEs). Kubernetes, an open-source project initiated by Google, became the standard [8]. AWS developed its own orchestrator, Elastic Container Service (ECS) [9] and later offered a managed Kubernetes service, Elastic Kubernetes Service (EKS) [10]. While powerful, these

orchestrators traditionally required users to manage a cluster of worker nodes (EC2 instances) where the containers ran.

Serverless computing, used by AWS Lambda, represented a shift. It focused on event-driven code execution (FaaS) where the provider fully managed the underlying compute infrastructure, scaling, and availability [11]. Users paid only for the execution time. The desire to combine the FaaS operational model with the flexible packaging of containers led to the development of serverless container platforms.

AWS Fargate, launched in 2017, was AWS's first major step in this direction. It acted as a "serverless compute engine" initially for ECS, and later for EKS, allowing users to define tasks/pods with resource requirements (vCPU, memory) and run them without managing any EC2 instances for the data plane. Fargate handled the provisioning and scaling of the underlying compute capacity automatically.

AWS App Runner, launched in 2021, targeted an even higher level of abstraction. Inspired by PaaS solutions like Heroku or Google Cloud Run, it aimed to make deploying web applications and APIs from source code (via connections to Git repositories) or container images (from ECR) extremely simple, bundling load balancing, TLS, scaling, and deployment pipelines.

AWS Lambda, launched in 2020, it added support for packaging functions as container images (up to 10GB) instead of just zip archives. This addressed limitations related to large dependencies, custom binaries, or specific runtime requirements within the Lambda execution model, offering container benefits within the familiar event-driven framework.

While numerous articles compare specific aspects like FaaS vs. Containers [12]or benchmark container startup times, a comparative analysis focusing specifically on the architectural trade-offs and use-case suitability of AWS's three main serverless container approaches – App Runner, Fargate, and Lambda Containers – provides practical value for practitioners navigating these options.

SYSTEM ARCHITECTURE III

1. *AWS App Runner*

AWS App Runner represents the highest level of abstraction among the three services. Its architecture is designed around simplicity and developer experience, allowing quick deployment of web applications and APIs with minimal configuration.

Key Architectural Components:

1. **Service Infrastructure:** App Runner provisions and manages all necessary infrastructure components, including compute resources, load balancers, and networking. These components remain completely hidden from users.
2. **Build Pipeline:** For source code deployments, App Runner includes an integrated build service that automatically converts source code into container images using buildpacks or Dockerfiles.
3. **Auto Scaling:** The service includes automatic scaling based on concurrent requests and CPU utilisation. Scaling parameters can be configured but with fewer options than Fargate.
4. **Networking Layer:** App Runner automatically provisions an application load balancer, assigns a public domain, and manages TLS certificates. It provides public endpoints by default, with VPC connectivity as an optional configuration.
5. **Operational Model:** The service follows a fully managed approach where developers only need to provide source code or container images. All operational aspects like health checks, logging, and metrics are built-in and require minimal configuration.

2. *AWS Fargate*

Key Architectural Components:

1. ***Orchestration Integration:*** Fargate works as a compute provider for ECS and EKS. Users define tasks (ECS) or pods (EKS) while Fargate manages the underlying compute resources.
2. **Networking Model:** Fargate containers run in their own isolated environment with dedicated kernel, CPU, and memory resources. They can be configured to run within VPCs, with support for private subnets, security groups, and network interfaces.
3. **Task Definition:** Users create detailed task definitions specifying container images, resource requirements, IAM roles, volumes, and networking. This provides precise control over the runtime environment.
4. **Auto Scaling:** Scaling is managed through separate AWS services - Application Auto Scaling for ECS tasks or Kubernetes Horizontal Pod Autoscaler for EKS pods. This offers sophisticated scaling policies based on various metrics.
5. **Operational Model:** While Fargate eliminates server management, users still manage the container orchestration layer. This requires more configuration but provides greater control over deployment strategies, service discovery, and load balancing.

3. *AWS Lambda*

Key Architectural Components:

1. **Execution Environment:** Lambda functions using container images run within the same secure and isolated execution environments as traditional zip-packaged functions. The underlying infrastructure remains fully managed by AWS.
2. **Container Image Support:** Lambda allows packaging functions as container images up to 10GB, compared to the 250MB limit for zip archives. These images must implement the Lambda Runtime API and adhere to specific base image requirements.
3. **Concurrency Model:** Lambda maintains its concurrency model with container images, allowing functions to scale based on demand, with controls for reserved and provisioned concurrency.
4. **Event-Driven Architecture:** Unlike the other services, Lambda remains fundamentally event-driven. Container-based functions respond to specific triggers rather than serving continuous web traffic directly (unless fronted by API Gateway).

5. **Cold Starts:** Container image functions experience cold starts similar to standard Lambda functions, though potentially longer due to larger image sizes. AWS has implemented optimisations to minimise this impact.

USE CASES IV

AWS App Runner Use Cases

Rapid Deployment of Web Applications

App Runner is ideal for teams seeking the fastest path from code to a production web service. Its fully managed nature makes it particularly well-suited for:

- Startups and small teams with limited DevOps resources
- Full-stack web applications needing quick iteration cycles
- Developer environments and staging deployments
- API services with moderate traffic patterns

Example Scenario: A digital agency developing client websites can use App Runner to rapidly deploy and demonstrate web applications directly from their Git repositories. The automatic build pipeline converts source code to running applications without managing build servers or deployment pipelines.

AWS Fargate Use Cases

Production-Grade Microservices Architectures

Fargate is well-suited for complex microservices deployments requiring:

- Advanced orchestration capabilities
- Sophisticated deployment strategies
- Service discovery and mesh capabilities
- Fine-grained security controls
- Integration with extensive AWS ecosystem

Example Scenario: An e-commerce platform can leverage Fargate with ECS for its microservices architecture, using service discovery for inter-service communication, application load balancers for traffic routing, and sophisticated auto-scaling policies to handle variable traffic patterns.

AWS Lambda with Container Images Use Cases

Event-Driven Processing with Complex Dependencies

Lambda with container images is ideal for event-processing scenarios requiring:

- Custom runtimes or system libraries
- Large dependency packages
- Compatibility with existing container workflows
- Short-lived, event-triggered execution

Example Scenario: A media processing service that needs to transform images or videos upon upload can use Lambda with container images. The container can include specialised libraries like OpenCV or FFmpeg for media processing, triggered when new files arrive in S3.

LIMITATIONS AND CHALLENGES VI

While the serverless container services from AWS offer significant advantages, they also present important limitations and challenges that practitioners should consider when making architectural decisions.

A. App Runner Limitations

1. Limited Customisation

- Restricted networking options compared to Fargate
- Limited integration with certain AWS services
- Fixed deployment patterns with fewer customisation options
- Constraints on container configuration and runtime environment

2. Operational Challenges

- Less mature service with fewer advanced features
- Regional availability limitations compared to other services
- Limited visibility into underlying infrastructure for debugging
- Potential vendor lock-in due to proprietary deployment model

B. Fargate Limitations

- **Complexity Overhead**

- Steep learning curve for ECS/EKS concepts
- Complex configuration requirements
- Numerous interconnected services to manage
- More operational overhead compared to App Runner

- **Cost Management**

- Pricing based on allocated resources rather than actual usage
- Potential for over-provisioning and cost inefficiency
- Additional costs for related services (load balancers, NAT gateways)
- No sub-minute billing granularity

C. Lambda Containers Limitations

Runtime Constraints

- 15-minute maximum execution time
- Memory limitations (up to 10GB)
- Limited CPU allocation tied to memory configuration
- Temporary storage constraints (/tmp limited to 512MB)

2. Container Limitations

- Requirement to implement Lambda Runtime API
- Restricted container environment
- Cold start penalties, especially for larger images
- No persistent local storage between invocations

CONCLUSION VII

This comparative analysis of AWS App Runner, AWS Fargate, and AWS Lambda with container images demonstrates that each service occupies a distinct position in the serverless container ecosystem, with specific strengths and limitations that make them suitable for different use cases.

AWS App Runner provides the simplest developer experience with minimal configuration required, making it ideal for teams seeking rapid deployment of web applications without operational complexity. Its streamlined approach comes at the cost of customisation flexibility but offers an excellent entry point into serverless containers.

AWS Fargate delivers the most comprehensive control and integration capabilities, positioning it as the optimal choice for production-grade microservices architectures with complex requirements. While requiring more configuration and expertise, it provides the necessary flexibility for sophisticated enterprise applications.

AWS Lambda with container images successfully bridges the gap between flexible container packaging and the operational simplicity of serverless computing. Its event-driven model and consumption-based pricing make it particularly well-suited for variable workloads and event processing pipelines.

For practitioners making architectural decisions, the key factors to consider include:

- Application traffic patterns and predictability
- Operational resources and expertise available
- Required level of control over infrastructure
- Integration requirements with other services
- Cost model alignment with usage patterns

As cloud computing continues to evolve, these serverless container services will likely converge in capabilities while maintaining their distinct operational models. Understanding their fundamental architectural differences enables practitioners to build cloud-native applications that best leverage the strengths of each approach while mitigating their limitations.

REFERENCES

REFERENCES VIII

[1] "Docker Basics," [Online]. Available: https://www.docker.com/.

[2] AWS, "AWS Lambda," 2025. [Online]. Available: https://aws.amazon.com/lambda/.

[3] "AWS Serverless Concepts," 2025. [Online]. Available: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-concepts.html.

[4] "AWS App Runner," 2025. [Online]. Available: https://docs.aws.amazon.com/apprunner/latest/dg/getting-started.html.

[5] "AWS Fargate," 2025. [Online]. Available: https://aws.amazon.com/fargate/.

[6] "AWS IaaS," [Online]. Available: https://aws.amazon.com/what-is/iaas/.

[7] "AWS Types of cloud computing," [Online]. Available: https://aws.amazon.com/types-of-cloud-computing/.

[8] "Kubernetes," 2025. [Online]. Available: https://kubernetes.io/.

[9] "AWS ECS," 2025. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html.

[10] "AWS EKS," 2025. [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html.

[11] "FaaS guide," [Online]. Available: https://www.geeksforgeeks.org/function-as-a-service-faas-system-design/.

[12] "Serverless (FaaS) vs. Containers - when to pick which?," 6 October 2017. [Online]. Available: https://www.serverless.com/blog/serverless-faas-vs-containers.