AWS Decision guide

# AWS Fargate or AWS Lambda?

# AWS Fargate or AWS Lambda?: AWS Decision guide

# Table of Contents

# AWS Fargate or AWS Lambda?

**Understand the differences and pick the one that's right for you**

| | |
|---|---|
| **Purpose** | To explore whether AWS Fargate or AWS Lambda meet your needs for a serverless compute service. |
| **Last updated** | November 15, 2024 |
| **Covered services** | • AWS Fargate<br>• AWS Lambda |

## Introduction

Before you get started exploring whether you choose AWS Lambda or AWS Fargate as your serverless compute service, you probably have considered the broader range of AWS compute services (covered in the Choosing an AWS compute service decision guide) and narrowed it down to these two choices because they provide:

- **Reduced operational overhead:** Both Lambda and Fargate abstract away server management, reducing the need for patching, maintenance, and capacity planning.
- **Pay-per-use pricing:** You only pay for the compute resources you actually use, potentially lowering costs for variable workloads.
- **Faster deployment:** Typically offers quicker deployment times compared to provisioning and configuring EC2 instances.
- **Built-in high availability:** Both services handle infrastructure redundancy automatically.
- **Simplified compliance:** Reduced attack surface and built-in security features can ease compliance efforts.
- **Focus on code:** Developers can concentrate more on writing application code rather than managing infrastructure.

While Lambda and Fargate are both serverless options, there are significant differences between them:

**AWS Fargate** is a serverless compute engine for containers, primarily used with Amazon ECS. It automatically manages your infrastructure, allowing you to focus on deploying and scaling containerized applications. Fargate is ideal for long-running applications, microservices, or batch processing—where you need fine-grained control over resource allocation (CPU, memory) and want to avoid managing underlying servers.

**AWS Lambda** is a serverless computing service that automatically runs your code in response to events, and manages the underlying compute resources. It's best suited for event-driven applications, such as processing files uploaded to Amazon S3, responding to HTTP requests, or running scheduled tasks. Lambda is also well-suited for stream processing and data processing applications due to its ability to scale automatically in response to events and handle high volumes of data in real time. Lambda can process data streams from sources like Amazon Kinesis or Amazon DynamoDB, allowing for efficient, serverless data transformations, filtering, and analytics without managing infrastructure. Lambda is designed for short-lived tasks (up to 15 minutes) and is billed based on the number of requests and execution time, making it cost-effective for sporadic workloads.

If your project involves event-driven, short-duration tasks or unpredictable workloads, AWS Lambda might be the better fit. If you need to run containerized applications with specific resource needs (or you require persistent processes), AWS Fargate would be more appropriate.

The following table gives a more detailed look at some of the differences between these services to get you started.

| Feature | AWS Fargate | AWS Lambda |
| --- | --- | --- |
| Execution model | Container-based, serverless compute | Event-driven, serverless functions |
| Supported languages | Any language that can run in a container | Supported languages: Node.js, Python, Java, C#, Go, Ruby, and PowerShell. You can also build a custom runtime to implement an AWS Lambda function in the language of your choice. |

| | | |
|---|---|---|
| Use case | Long-running, containerized applications | Short-duration, event-driven tasks |
| Scaling | Automatic scaling based on desired task count | Automatic scaling per request |
| Cold start | 35 seconds to 2 minutes | 100 ms to 2 seconds |
| Execution time limit | No hard limit | 15 minutes maximum |
| Memory allocation | Up to 120 GiB | Up to 10 GiB |
| CPU allocation | Up to 16 vCPU | Proportional to memory, up to 6 vCPU |
| Networking | Runs in VPC, can use ENIs | Can run in AWS managed VPC or attached to a customer-managed VPC using AWS Hyperplane |
| State Management | Containers in Fargate can maintain state across requests as long as the container is running, making it possible to handle sessions, cache data, or keep in-memory state without needing external storage. External storage is recommended for critical data. | Stateless by design (state must be managed externall y, for example, Amazon S3, Amazon DynamoDB, Amazon EFS) |
| Container support | Supports containers | Limited container support (via container image deploymen ts) |
| Orchestration | Integrated with Amazon ECS | No orchestration required |
| Pricing model | Per second billing for vCPU and memory used | Per invocation and duration (GB-seconds) |

| | | |
|---|---|---|
| Concurrency limits | Based on cluster capacity | 1000 concurrent executions by default (can be increased) |
| Event-driven invocation | Requires additional setup | Native support for various AWS event sources |
| Cold start mitigation | Lazy loading images with Seekable OCI can speed up starting Fargate tasks | Provisioned concurrency available |
| Package size limit | No specific limit (containe r size limited by configured ephemeral storage, 200 GiB maximum) | 250 MB unzipped, including layers, 10GB for container image deployments |

# Differences between Fargate and Lambda

Explore the differences between Fargate and Lambda in a number of key areas.

Languages supported

**Fargate**: AWS Fargate is a container orchestration service, meaning that it supports any programming language or runtime environment that can be packaged into a Docker container. This flexibility allows developers to use virtually any language, framework, or library that suits their application needs. Whether you're using Python, Java, Node.js, Go, .NET, Ruby, PHP, or even custom languages and environments, Fargate can run them as long as they are encapsulated in a container. This broad language support makes Fargate ideal for running diverse applications, including legacy systems, multi-language microservices, and modern cloud-native applications.

**Lambda**: AWS Lambda offers native support for a more limited set of languages compared to Fargate, specifically designed for event-driven functions. As of now, Lambda officially supports the following languages:

- Node.js
- Python
- Java

- Go
- Ruby
- C#
- PowerShell

Lambda also supports custom runtimes, which allow you to bring your own language or runtime environment, but this requires more setup and management compared to using the natively supported options. If you choose to deploy your Lambda function from a container image, you can write your function in Rust by using an AWS OS-only base image and including the Rust runtime client in your image. If you're using a language that doesn't have an AWS-provided runtime interface client, you must create your own.

Event-driven invocation

**Lambda** is inherently designed for event-driven computing. Lambda functions are triggered in response to a variety of events, including changes in data, user actions, or scheduled tasks. It integrates seamlessly with many AWS services, such as Amazon S3 (for example, invoking a function when a file is uploaded), DynamoDB (for example, triggering on data updates), and API Gateway (for example, handling HTTP requests). The Lambda event-driven architecture is ideal for applications that need to respond immediately to events without requiring persistent compute resources.

**Fargate** is not natively event-driven, but with some additional boilerplate logic, it can integrate with event sources such as Amazon SQS and Kinesis. Whereas Lambda handles the bulk of this integration logic for you, you will have to implement this integration yourself using the APIs for these services.

Runtime/use cases

**Fargate** is designed to run containerized applications, providing a flexible runtime environment where you can define the CPU, memory, and networking settings for your containers. Since Fargate operates on a container-based model, it supports long-running processes, persistent services, and applications with specific runtime requirements. The containers in Fargate can run indefinitely, as there is no hard limit on execution time, making it ideal for applications that need to be up and running continuously.

**Lambda**, on the other hand, is optimized for short-lived, event-driven tasks. Lambda functions are executed in a stateless environment where the maximum execution time is capped at 15 minutes. This makes Lambda well-suited for scenarios like file processing, real-time data

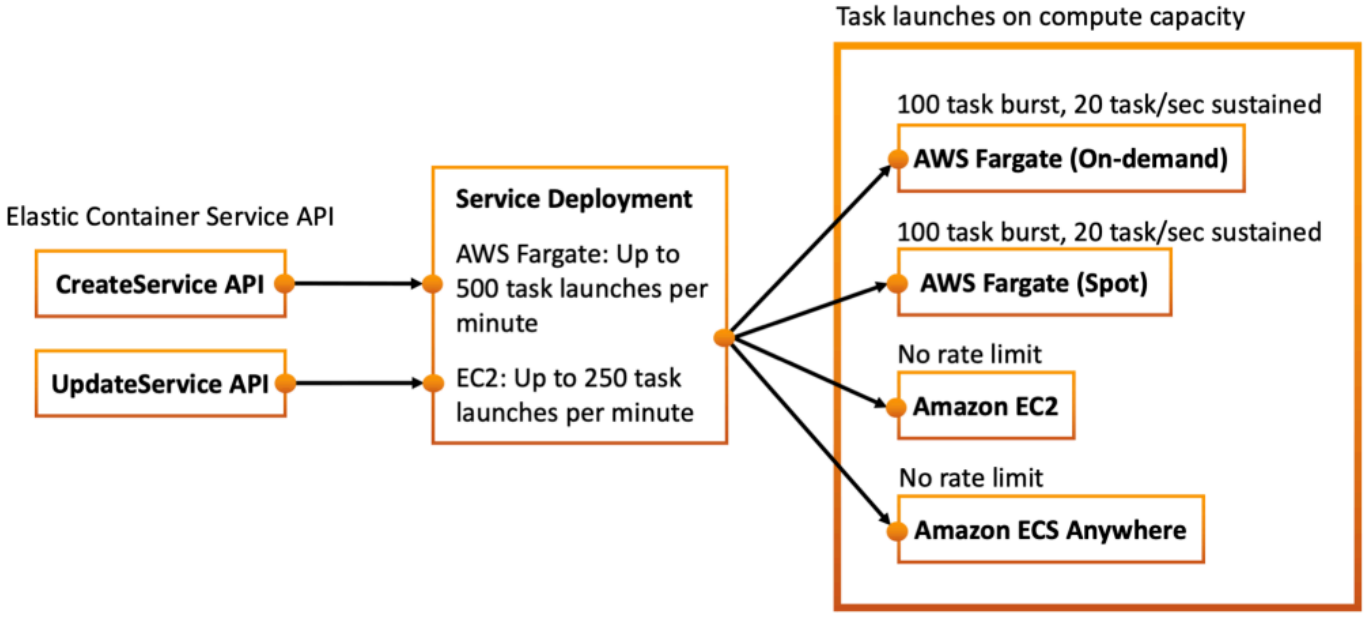
streaming, and HTTP request handling, where the tasks are brief and don't require long-running processes.

In Lambda, the runtime environment is more abstracted, and there is less control over the underlying infrastructure. Lambda's stateless nature means that each function invocation is independent, and any state or data that needs to persist between invocations must be managed externally (for example, in databases or storage services).

Scaling

**Fargate** scales by adjusting the number of running containers based on the desired state defined in your container orchestration service (Amazon ECS). This scaling can be done manually or automatically through Amazon EC2 Auto Scaling. This blog post offers further details on how.

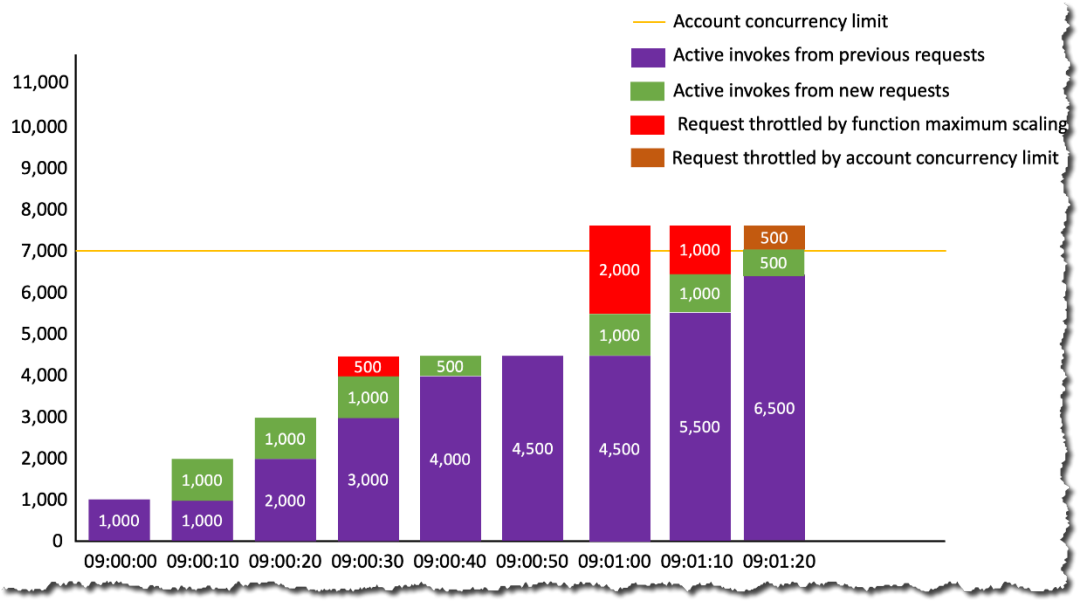
In Fargate, each container runs in its isolated environment, and scaling involves launching additional containers or stopping them based on the load. The Amazon ECS service scheduler is able to launch up to 500 tasks in less than a minute per service for web and other long-running services.




For **Lambda**, *concurrency* is the number of in-flight requests that your AWS Lambda function is handling at the same time. This differs from concurrency in Fargate, where each Fargate task can handle concurrent requests as long as there are available compute and network resources. For each concurrent request, Lambda provisions a separate instance of your execution environment. As your functions receive more requests, Lambda automatically handles scaling

the number of execution environments until you reach your account's concurrency limit. By default, Lambda provides your account with a total concurrency limit of 1,000 concurrent executions across all functions in an AWS Region, and you can request a quota increase if needed.

For each Lambda function in a Region, the concurrency scaling rate is 1,000 execution instances every 10 seconds, up to the maximum account concurrency. As explained in this blog, if the number of requests in a 10 second period exceeds 1,000, the additional requests will be throttled. The following graph demonstrates how Lambda scaling works assuming an account concurrency of 7000.




Cold start and cold-start mitigation

**Lambda** can experience cold starts, which occur when a function is invoked after being idle for some time. During a cold start, the Lambda service needs to initialize a new execution environment, including loading the runtime, dependencies, and the function code. This process can introduce latency, particularly for languages with longer initialization times (for example, Java, or C#). Cold starts can impact the performance of applications, especially those requiring low-latency responses.

To mitigate cold starts in Lambda, several strategies can be employed:

- **Minimize function size**: Reducing the size of your function package and its dependencies can decrease the time needed for initialization.
- **Increase memory allocation**: Higher memory allocations increase CPU capacity, potentially reducing initialization time.

- **Keep functions warm**: Periodically invoking your Lambda functions (for example, using CloudWatch Events) can keep them active and reduce the likelihood of cold starts.
- **Lambda SnapStart**: Use Lambda SnapStart for Java functions to reduce startup time.
- **Provisioned concurrency**: This feature keeps a specified number of function instances warm and ready to serve requests, reducing cold start latency. However, it increases costs as you're paying for the provisioned instances even if they're not actively handling requests.

**Fargate** is not generally impacted by cold starts in the same way as Lambda. The time it takes to start a Fargate task is directly correlated to the time it takes to pull the container images defined in the task from the image registry. Fargate also supports lazy loading of container images that have been indexed with Seekable OCI (SOCI). Lazy loading container images with SOCI reduces the time taken to launch Amazon ECS tasks on Fargate. Fargate runs containers that remain active for as long as needed, meaning they're always ready to handle requests. However, if you need to start new containers in response to scaling events, there might be some delay while the containers are initialized, but this is typically less significant compared to Lambda cold starts.

Memory and CPU options

**Fargate** provides granular control over both memory and CPU resources for your containerized applications. When you launch a task in Fargate, you can specify the exact CPU and memory requirements based on the needs of your application. The CPU and memory allocations are independent, allowing you to choose combinations that best suit your workload. For instance, you can select CPU values ranging from 0.25 vCPUs to 16 vCPUs and memory from 0.5 GB to 120 GB per container, depending on your configuration.

This flexibility is ideal for running applications that require specific performance characteristics, such as memory-intensive databases or CPU-bound computation tasks. Fargate allows you to optimize your resource allocation to balance cost and performance effectively.

In **Lambda**, memory and CPU are linked, with the CPU automatically allocated in proportion to the amount of memory you select. You can choose memory allocations between 128 MB and 10 GB, in 1 MB increments. The CPU scales with the memory, up to 6 vCPU, meaning higher memory settings result in more CPU power, but you don't have direct control over the CPU allocation itself.

This model is designed for simplicity, allowing developers to quickly adjust memory settings without needing to manage CPU configurations. However, it might be less flexible for

workloads that require a specific balance between CPU and memory resources. Lambda's model is suitable for tasks where you want straightforward scaling based on memory needs, but it might not be optimal for applications with complex or highly specific resource demands.

Networking

When you deploy tasks in **Fargate**, they run in an Amazon VPC (Amazon Virtual Private Cloud), giving you full control over the networking environment. This includes configuring security groups, network access control lists (ACLs), and routing tables. Each Fargate task gets its own network interface, with a dedicated private IP address, and can be assigned a public IP address if needed.

Fargate supports advanced networking features such as load balancing (using AWS Elastic Load Balancing), VPC peering, and direct access to other AWS services within the VPC. You can also use AWS PrivateLink for secure, private connectivity to supported AWS services, without traversing the internet.

By default, **Lambda** functions are run in a managed network environment without direct control over network interfaces or IP addresses. However, Lambda can be attached to a customer-managed VPC using AWS Hyperplane, enabling you to control access to resources inside your VPC.

When Lambda functions are attached to a customer-managed VPC, they inherit the VPC's security groups and subnet configurations, allowing them to interact securely with other AWS services (like RDS databases) within the same VPC.

The Lambda service uses a Network Function Virtualization platform to provide NAT capabilities from the Lambda VPC to customer VPCs. This configures the required elastic network interfaces (ENIs) at the point where Lambda functions are created or updated. It also enables ENIs from your account to be shared across multiple execution environments, which allows Lambda to make more efficient use of a limited network resource when functions scale.

Since ENIs are an exhaustible resource and there is a soft limit of 250 ENIs per Region, you should monitor elastic network interface usage if you are configuring Lambda functions for VPC access. Lambda functions in the same AZ and same security group can share ENIs. Generally, if you increase concurrency limits in Lambda, you should evaluate if you need an elastic network interface increase. If the limit is reached, this causes invocations of VPC-enabled Lambda functions to be throttled.

Pricing model

**Fargate** pricing is based on the resources allocated to your containers, specifically the vCPU and memory you select for each task. You are billed per second, with a one-minute minimum charge, for the CPU and memory that your containers use. The costs are directly tied to the resources your application consumes, meaning you pay for what you provision, regardless of whether the application is actively processing requests. Fargate is well-suited for predictable workloads where you need specific resource configurations and can optimize costs by adjusting the allocated resources. Additionally, there might be additional charges for related services, such as data transfer, storage, and networking (for example, VPC, Elastic Load Balancing).

**Lambda** has a different pricing structure that is event-driven and pay-per-execution. You are charged based on the number of requests your functions receive and the duration of each execution, measured in milliseconds. Lambda also factors in the amount of memory you allocate to your function, with costs scaling based on the memory used and the execution time. The pricing model includes a free tier, offering 1 million free requests and 400,000 GB-seconds of compute time per month, which makes Lambda particularly cost-effective for low-volume, sporadic workloads.

The Lambda pricing model is ideal for applications with unpredictable or bursty traffic patterns, as you only pay for actual function invocations and execution time, without the need to provision or pay for idle capacity.

# Use

Now that you've read about the criteria for choosing between AWS Fargate and AWS Lambda, you can select the service that meets your needs, and use the following information to help you get started using each of them.

AWS Fargate

- **Learn how to create an Amazon ECS Linux task for the Fargate launch type**

  Get started with Amazon ECS on AWS Fargate by using the Fargate launch type for your Linux tasks.

  Explore the guide

- **Learn how to create an Amazon ECS Windows task for the Fargate launch type**

Get started with Amazon ECS on AWS Fargate by using the Fargate launch type for your Windows tasks.

Explore the guide

- **Getting started with Fargate and Amazon EKS**

  This guide describes how to get started running your pods on AWS Fargate with your Amazon EKS cluster.

  Explore the guide

- **AWS Fargate pricing**

  Use this guide to understand how vCPU, memory, storage, and operating system configurations impact AWS Fargate pricing.

  Explore the guide

- **AWS Fargate frequently asked questions**

  Get answers to common questions about AWS Fargate capabilities, and best practices for implementation.

  Explore the guide

AWS Lambda

- **Create a serverless file-processing app**

  A step-by-step walkthrough of setting up and using Amazon SNS. It covers topics such as creating a topic, subscribing endpoints to a topic, publishing messages, and configuring access permissions.

  Explore the guide

- **Serverless Developer Guide**

  This guide helps you develop a better conceptual understanding of serverless application development, and how various AWS services fit into together to create *application patterns* that form the core of your cloud applications.

  Explore the guide

- **Serverless Land**

  This site brings together the latest information, blogs, videos, code, and learning resources for AWS Serverless. Learn to use and build apps that scale automatically on low-cost, fully managed serverless architecture.

  Explore the site

- **AWS Lambda pricing**

  Use this guide to estimate expenses and optimize costs based on function usage and configuration. It includes a pricing calculator to calculate your AWS Lambda and architecture cost in a single estimate.

  Explore the guide

- **AWS Lambda frequently asked questions**

  Get answers to common questions about AWS Lambda capabilities, and best practices for implementation.

  Explore the guide

# Document history for AWS Fargate or AWS Lambda?

The following table describes the important changes to this decision guide. For notifications about updates to this guide, you can subscribe to an RSS feed.

| Change | Description | Date |
| --- | --- | --- |
| Initial release | Initial release of the decision guide. | November 15, 2024 |