Does Serverless Have Servers?

BYTEBYTEGO
NOV 16, 2023 · PAID

125 1 13 Share ···

Serverless computing refers to the paradigm of building and running applications without managing the underlying servers. With serverless, the cloud provider abstracts away the servers, runtimes, scaling, and capacity planning from the developer. The developer focuses on the application code and business logic, while the cloud provider handles provisioning servers, auto-scaling, monitoring, etc., under the hood. Over the past decade, this "no-ops" approach has greatly simplified deploying cloud-native applications.

The term "serverless" is misleading since servers are still involved. But from the developer's perspective, there are no visible servers to manage. The cloud provider handles the servers, operating systems, and runtimes. This model allows developers to deploy event-driven, auto-scaling functions and APIs without worrying about the infrastructure.

Some key enablers of serverless computing include FaaS (Functions-as-a-Service), BaaS (Backend-as-a-Service), containers, microservices, and auto-scaling. This newsletter will dive deeper into serverless concepts, architecture, user cases, limitations, and more.

The Evolution of Serverless

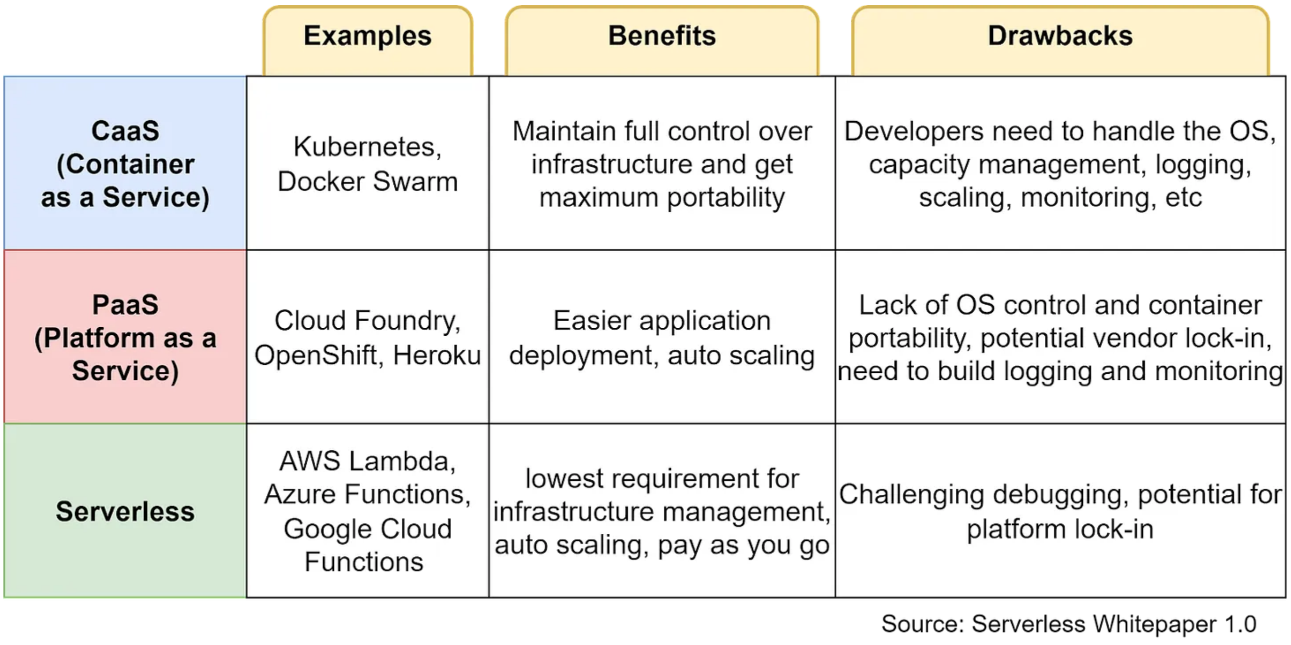
Serverless computing gained prominence around 2014 with the release of AWS Lambda. It allowed developers to run event-driven applications without having to provision backend servers. Functions would auto-scale based on load while AWS handled the infrastructure.

The release of Azure Functions and Google Cloud Functions further popularized this FaaS model.

With the rise of Docker and Kubernetes between 2013 and 2014, containerization and microservices made it easy to deploy serverless applications and accelerated the adoption of serverless computing.

In 2018, the Cloud Native Computing Foundation (CNCF) published the *Serverless Whitepaper 1.0.* The whitepaper positioned serverless as a new cloud-native approach compared to Containers-as-a-Service (CaaS) and Platform-as-a-Service (PaaS).

The table below lists some key benefits and drawbacks of each model. CaaS has full control over the infrastructure but requires more effort in monitoring, logging, and capacity management. PaaS facilitates easier application deployment and auto-scaling but doesn't have container portability. Serverless has the lowest requirement for infrastructure management, with autoscaling, reduced ops, and pay-per-use billing as crucial benefits.



	Examples	Benefits	Drawbacks
CaaS (Container as a Service)	Kubernetes, Docker Swarm	Maintain full control over infrastructure and get maximum portability	Developers need to handle the OS, capacity management, logging, scaling, monitoring, etc
PaaS (Platform as a Service)	Cloud Foundry, OpenShift, Heroku	Easier application deployment, auto scaling	Lack of OS control and container portability, potential vendor lock-in, need to build logging and monitoring
Serverless	AWS Lambda, Azure Functions, Google Cloud Functions	lowest requirement for infrastructure management, auto scaling, pay as you go	Challenging debugging, potential for platform lock-in

Source: Serverless Whitepaper 1.0

Today, serverless has become a popular option for startups to quickly build and iterate on applications for product-market fit. The benefits of adopting serverless computing include:

1. Cost savings: With serverless computing, providers adopt a pay-as-you-go billing model based on usage. Developers only pay for the specific compute resources utilized when the functions run. There is no need to over-provision capacity upfront and pay for idle resources. This billing model enables efficient resource

utilization and significant cost savings compared to provisioning servers or virtual machines with fixed capacity.

2. Faster time-to-market: With serverless computing, developers don't need to spend time on infrastructure management tasks like capacity planning. Developers can focus on writing code and business logic, allowing them to deploy applications quickly.
3. Auto-scaling: Serverless applications can automatically scale up and down based on traffic patterns. Developers can initially provision minimal compute resources. As incoming requests increase, the platform automatically allocates more resources to handle the load. When traffic decreases, excess capacity is released.
4. Expanding developer skills: Serverless computing abstracts away infrastructure and operational concerns by modularizing computation and storage into functions and services. It enables front-end developers to work on simple backend services as well.

Next, let's look at the key concepts in serverless computing.

FaaS and Event-Driven Architecture

The term "serverless" has two related but distinct meanings:

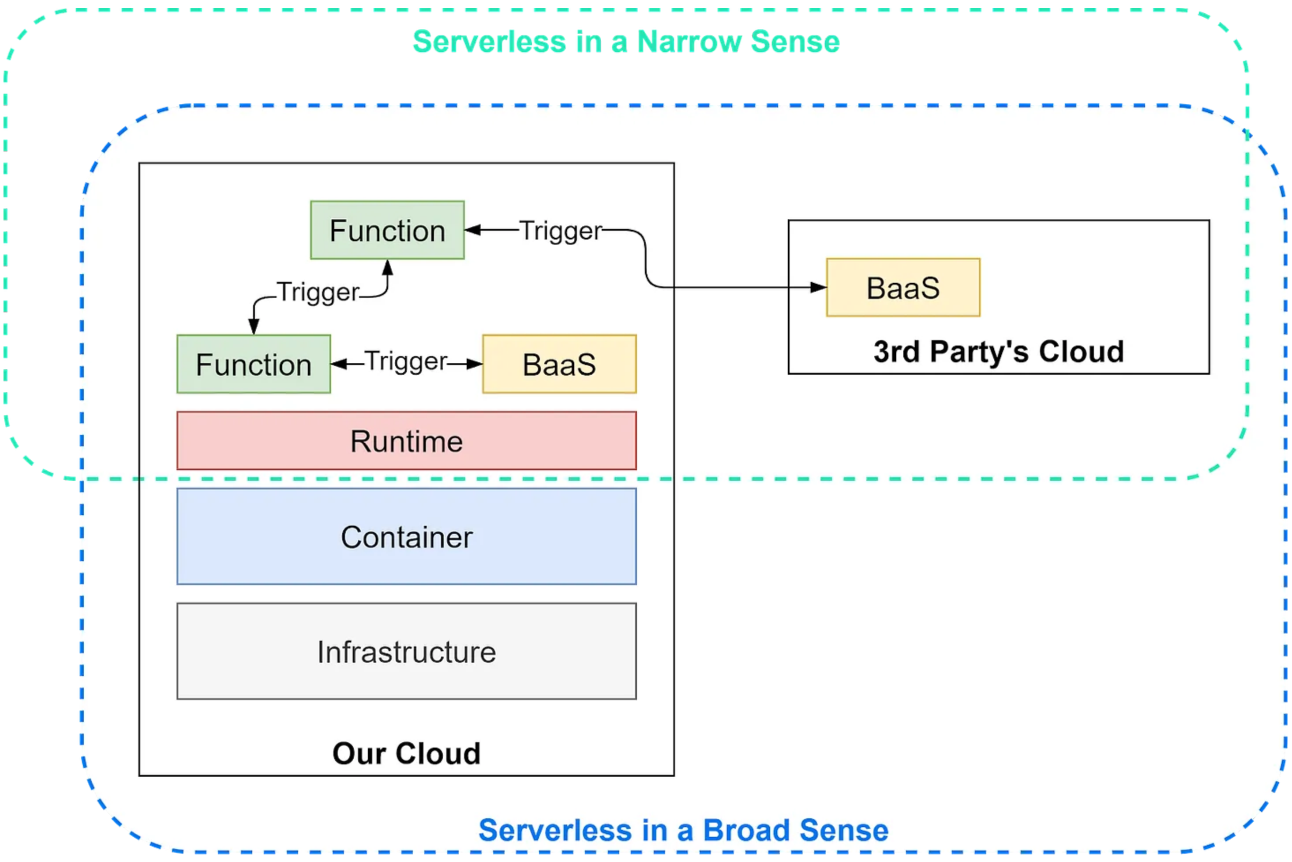
In a narrow sense, serverless refers specifically to FaaS. With FaaS, developers deploy auto-scaling function code without provisioning servers. The functions are executed on-demand based on event triggers.

FaaS functions are stateless and ephemeral. BaaS is combined with FaaS to provide API-based services maintained by the cloud provider and offer stateful capabilities like storage, databases, etc.

More broadly, serverless implies a "NoOps" approach where developers don't manage any backend servers or infrastructure. This concept includes FaaS, BaaS, and other managed cloud services like databases, storage, etc.

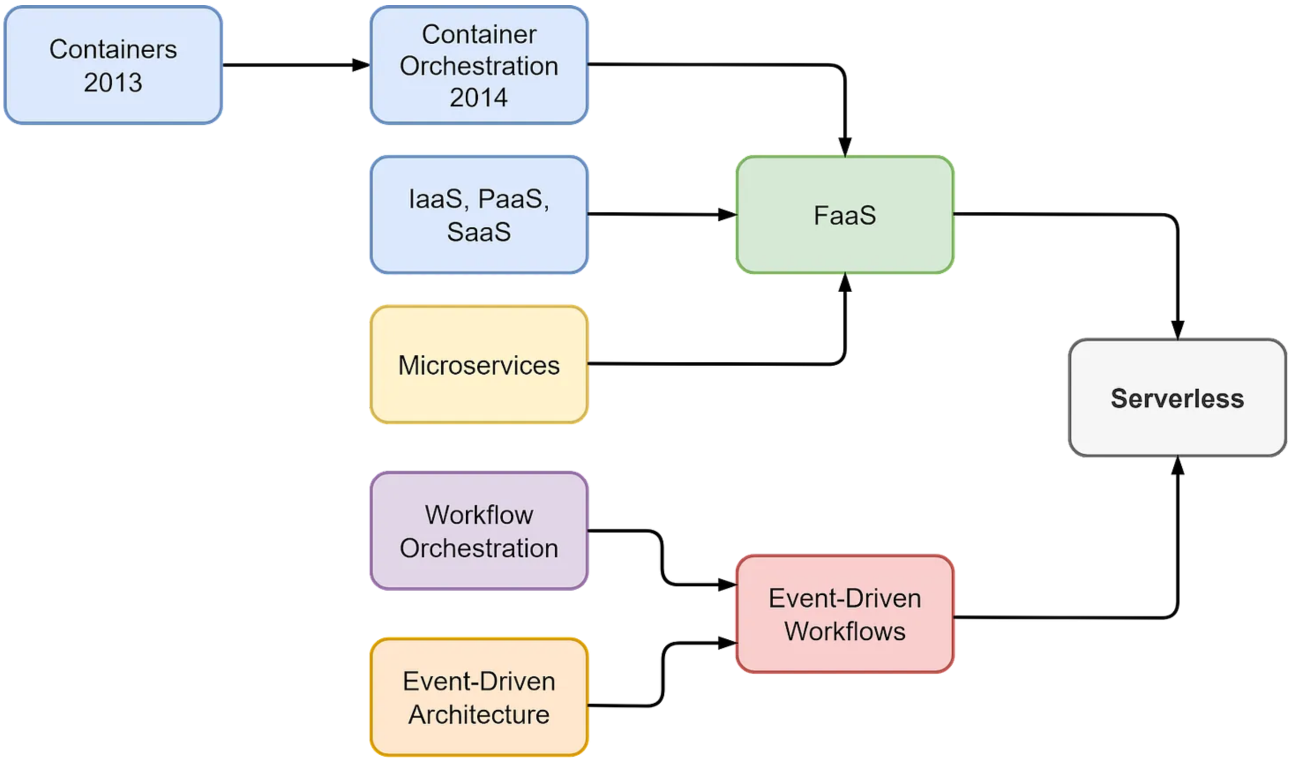
In this sense, serverless means the servers are abstracted from the developer. The cloud provider handles all server provisioning, scaling, availability, etc. Developers just use the services through APIs and don't operate the backend.

The diagram below explains the scope of the two types of serverless.



Functions and event triggers are two key components of cloud-native architecture. The diagram below shows their evolution.

The evolution of containers, PaaS, and microservices paved the way for the development of FaaS. The event-driven workflow defines the system as a predefined set of steps where we can embed business logic. The two architectural developments make serverless computing a popular choice.



In the next section, we will review how FaaS works.

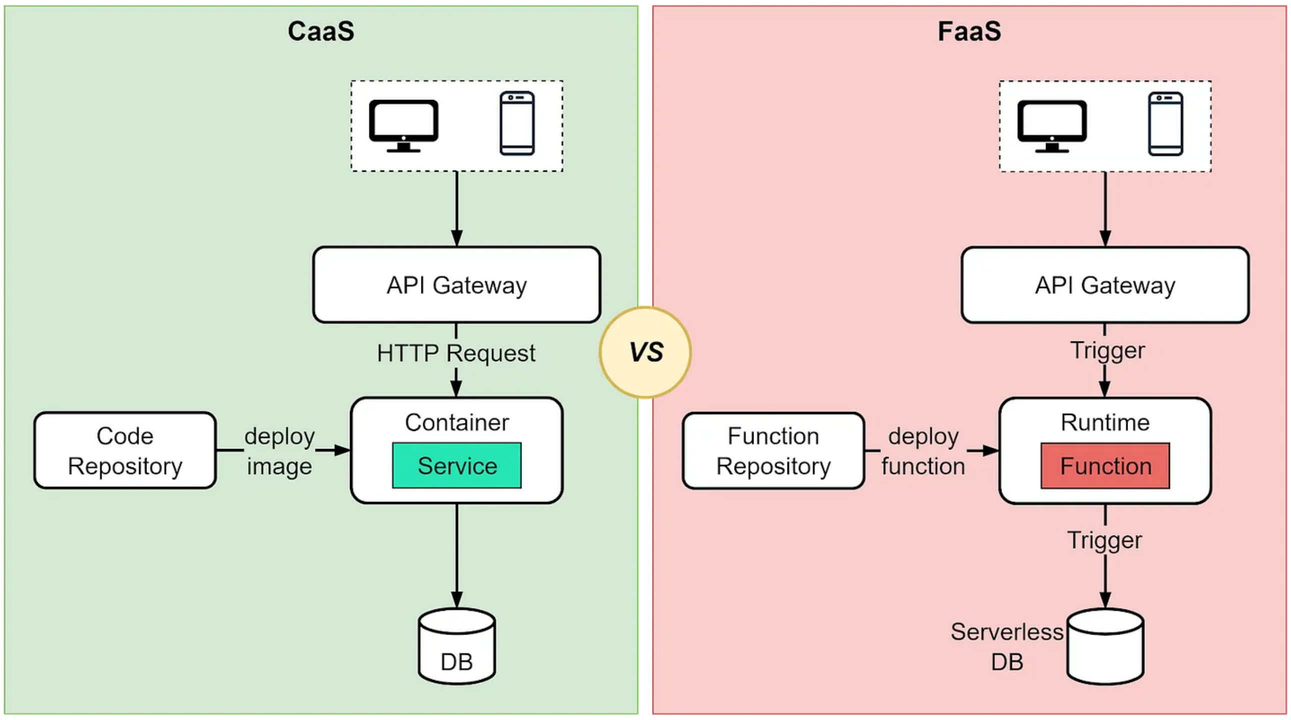
FaaS

Unlike CaaS, FaaS handles the runtime environment so developers can focus solely on the function-based application logic. The diagram below illustrates how FaaS works differently from regular service processes.

With CaaS, we bundle dependencies into containers like Docker. We must manage the entire application context and runtime environment.

With FaaS, the cloud provider defines the runtime environment. We just code the function logic without worrying about dependencies or runtime management. When a function finishes executing, the runtime is destroyed along with the function. Functions interact via event triggers and scale automatically based on demand - new instances spin up to handle more requests and scale down when traffic decreases.

The key difference is that FaaS abstracts away runtime management and scaling, allowing developers to focus on writing code. We only pay for resources once a request triggers function execution. CaaS gives more control over dependencies and runtime but requires configuring and managing the environment



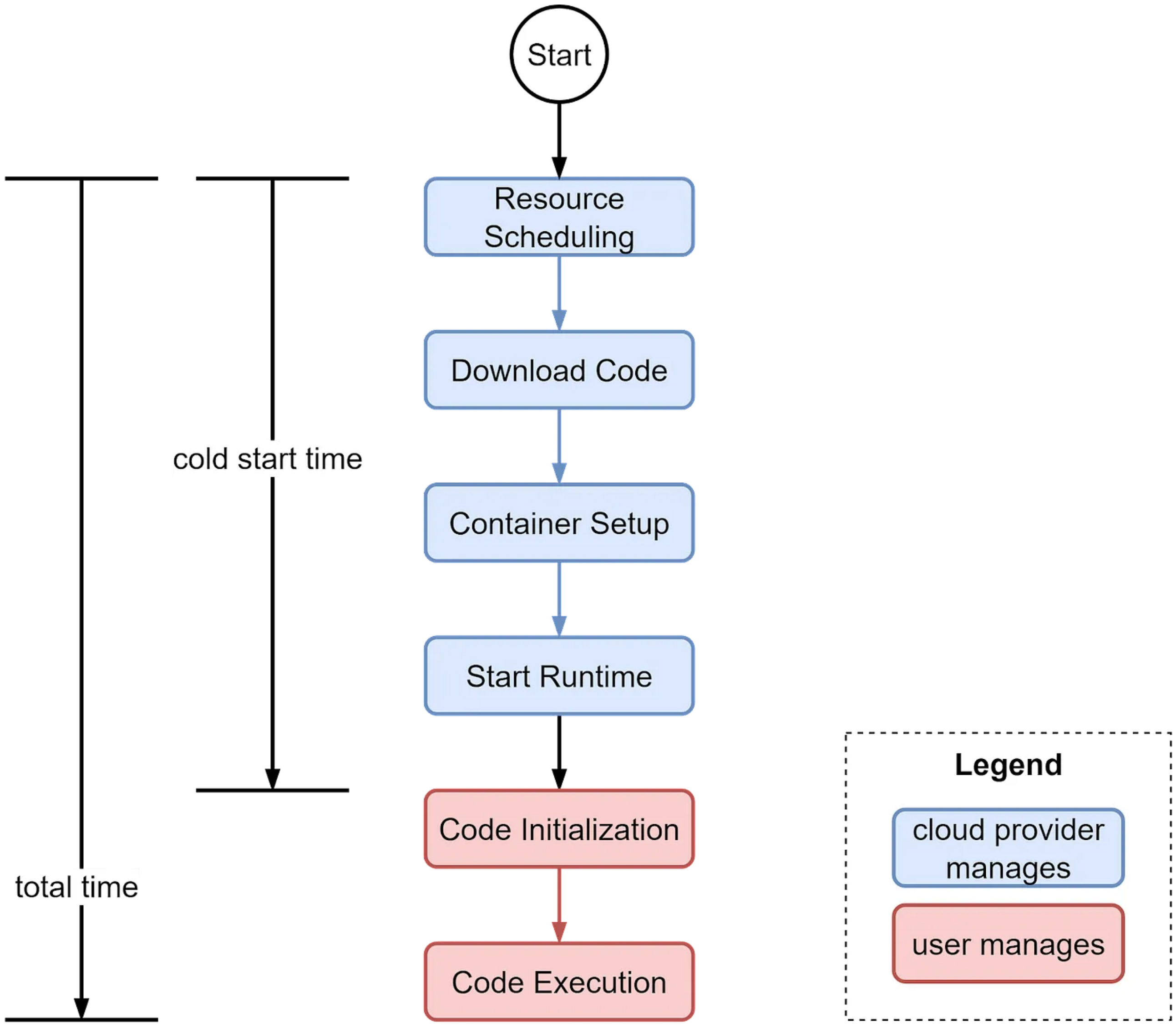
One of the main benefits of FaaS is enabling rapid scaling through fast cold starts compared to traditional servers. But how fast are FaaS cold starts in reality?

Cold Start Process

With traditional servers, restarting and launching apps took minutes. In contrast, FaaS functions can begin executing much faster:

- Function complexity - Simple functions with few dependencies can start in tens of milliseconds.
- Larger functions with significant code and many third-party dependencies may take seconds to cold start.

The diagram below outlines the cold start process. Blue boxes are managed by the cloud provider, and red boxes by the function owner. Cold start time is measured from calling the function to completing instance preparation.



While FaaS cold starts are much faster than traditional servers, the latency is still too long for applications requiring sub-10 millisecond response times. For these low-latency use cases, cold starts present challenges around unpredictability and tail latency spikes.

There are optimization techniques to help mitigate cold start latency:

- Cloud providers pre-download function code and dependencies to optimize cold starts.
- Reserving instances avoids cold starts for subsequent invocations, though instances may still be destroyed after periods of inactivity. Reserving instances incurs extra costs.
- Caching memory/disk snapshots with services like AWS SnapStart improves cold starts.

Monitoring and measuring cold start latency is vital for user-facing applications to understand and improve performance.

Cost Model

The pay-per-use model of FaaS lowers initial costs since we only pay when requests come in. Let’s examine the cost model and understand why functions are so popular.

There are several cost dimensions to consider:

1. Number of function invocations: More function calls equals higher costs, so expenses scale with traffic fluctuations.
2. Duration of each invocation: Charges are based on how many CPU-seconds are used per invocation.
3. Memory allocated to the function: The memory allotted to a function impacts its cost. More memory typically increases charges, proportional to the total allocation.
4. CPU and memory scaling: Some providers tie CPU cores to memory so that more memory can increase processing power. Optimizing memory improves performance and increases overall costs.

The core pricing dimensions are similar across providers, but each could have additional factors like network usage and usage tiers that influence total costs. Still, the pay-per-use model drives the significant cost benefits of FaaS.

The real value of FaaS emerges in orchestrating services and data at high throughput and low latency. The lightweight, event-driven model is well-suited for connecting intermittent IoT devices. For example, home appliances and gadgets increasingly utilize FaaS to handle data transmission tasks.

In a typical serverless architecture, FaaS is combined with complementary services like Backend-as-a-Service (BaaS) to enable stateful data processing and storage needs.

Next, we will explore how BaaS complements FaaS, enabling full-stack serverless designs that blend stateless and stateful applications.

BaaS

Backend-as-a-Service (BaaS) provides API-based services that replace an application's core backend functionality. BaaS creates an abstraction layer that renders the backend operations effectively "serverless" to developers by providing these APIs as auto-scaling, managed services. This abstraction simplifies and accelerates development, as developers can leverage APIs for complex tasks without managing servers.

In serverless architectures, BaaS complements stateless FaaS. FaaS excels at executing logic in response to events, while BaaS fills gaps like database interactions. For example, developers can utilize BaaS services for data persistence to avoid the need to manage database operations directly in the stateless FaaS environment.

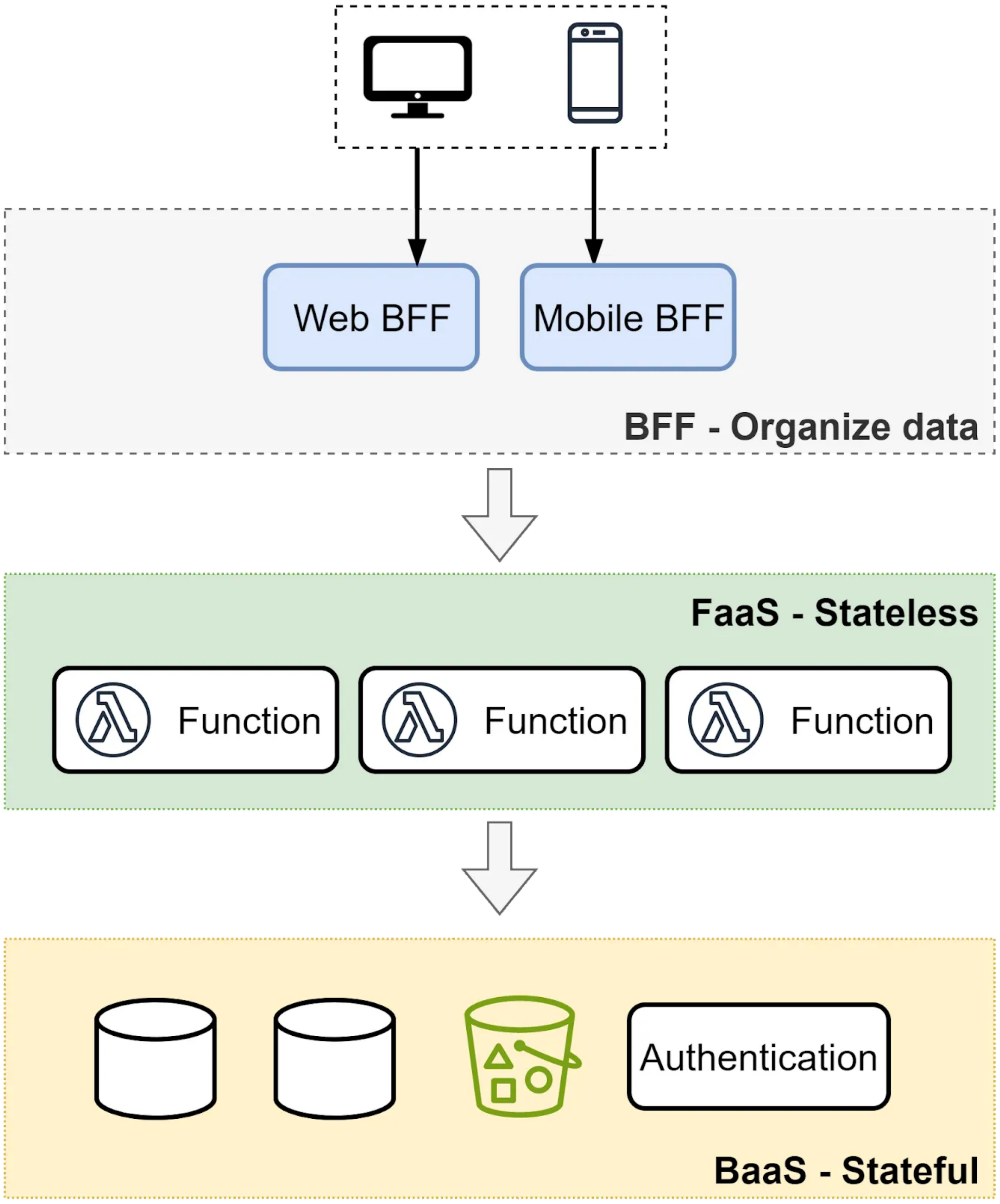
BaaS evolution is often contrasted with the microservices approach. Transforming monolithic backend services into BaaS APIs effectively transforms them into auto-scaling, NoOps interfaces for data and services. Typical BaaS use cases include authentication, remote updating, cloud storage, and database management.

The "NoOps" concept emerged alongside serverless computing. With NoOps, developers can focus on new features while leaving operations to cloud providers.

Combining the two pillars of serverless computing - FaaS and BaaS - a typical architecture is:

- FaaS handles business logic and processing tasks responding to events
- BaaS provides scalable data handling and backend services
- Backend for Frontend (BFF) processes and organizes the data generated from the FaaS layer, tailoring it for consumption by diverse front-end devices

This layered architecture enables a clear separation of concerns: the front end focuses on user experience while the BFF layer manages data processing traditionally done by model and controller.



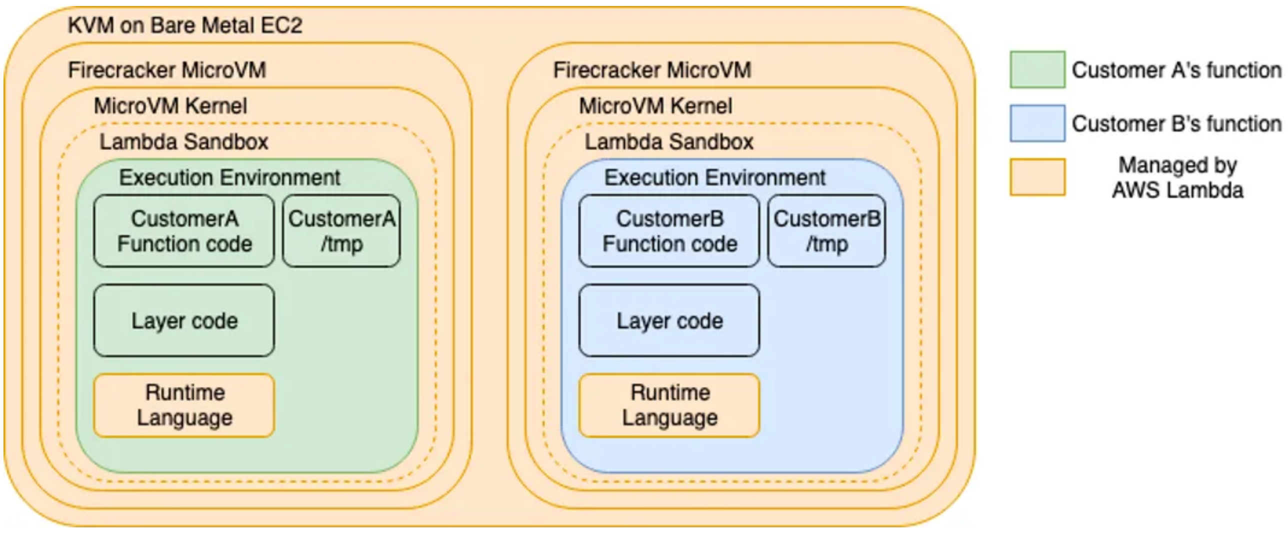
In the next section, let's deep dive into how functions are implemented from the cloud service provider's perspective.

Serverless Has Servers

The concept of serverless is from the cloud users' perspective. In reality, FaaS and BaaS functions still need to run on servers managed by cloud providers.

Firecracker MicroVMAWS

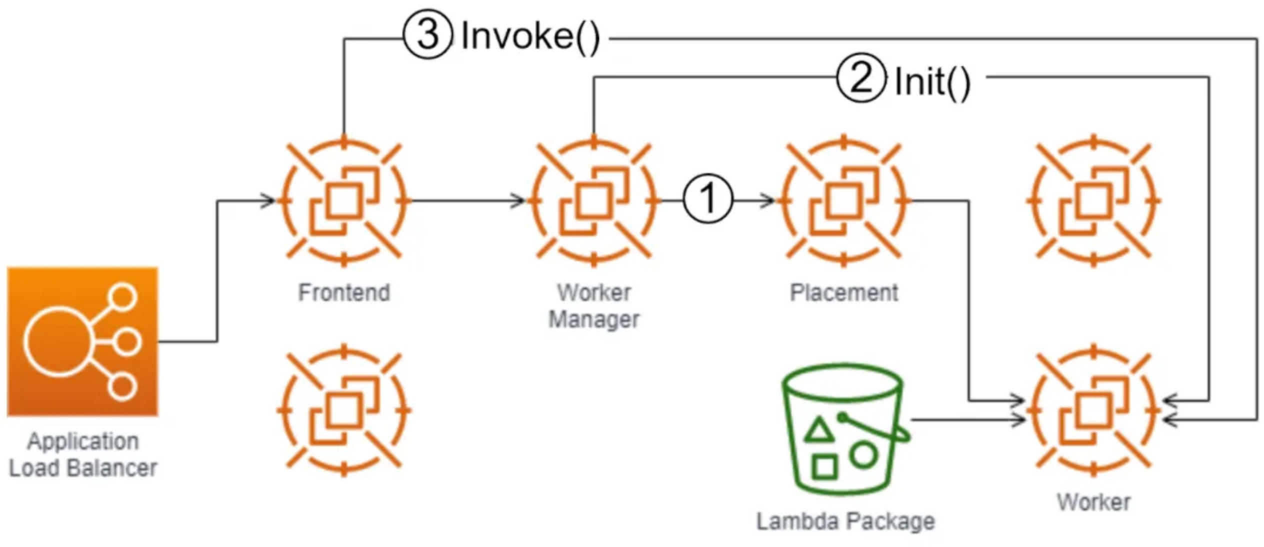
Lambda uses Firecracker lightweight virtual machines (microVMs) to isolate and run Lambda functions. Firecracker is an open-source virtualization technology developed by AWS and written in Rust.The diagram below shows the AWS Lambda execution environment. Lambda functions run in microVMs on AWS EC2 instances called Lambda Workers. Each Worker can run multiple isolated microVMs to enable multi-tenant container and function-based services. Users only need to worry about their function code (the green and blue areas) while AWS handles the underlying infrastructure



Source: Lambda executions by AWS docs

Next, let's look at how the functions are initialized and invoked. There are two modes: synchronous execution and asynchronous execution.

Synchronous Execution



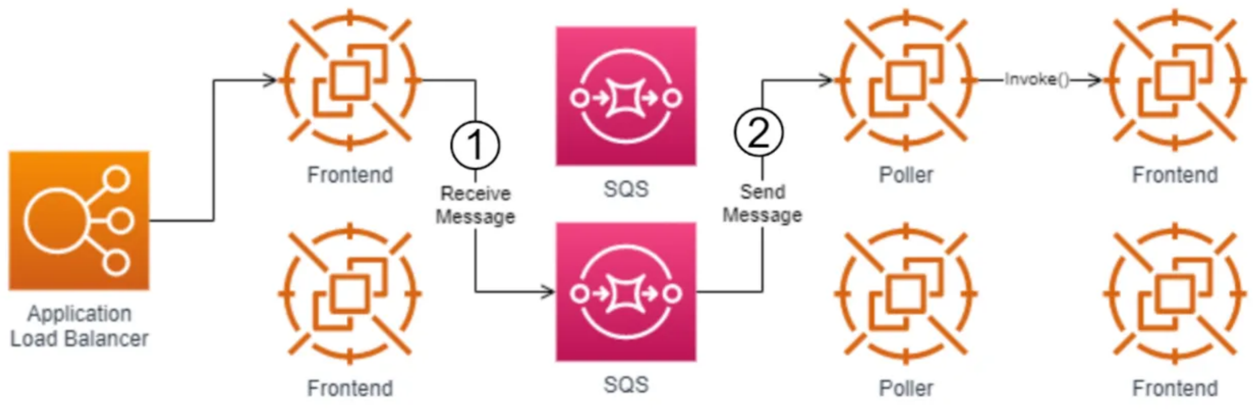
Source: https://www.bschaatsbergen.com/behind-the-scenes-lambda/

Step1: "The Worker Manager communicates with the Placement Service to allocate a microVM on a Worker node and returns the location.

Step 2: "The Worker Manager calls Init() to initialize the function for execution by downloading the code package from S3 and setting up the Lambda runtime".

Step 3: The Frontend Worker synchronously invokes the function by calling Invoke().

Asynchronous Execution



Step 1: The Application Load Balancer forwards the invocation event to an available Frontend Worker, which places the event on an internal SQS queue.

Step 2: Poller processes take events from the queue and synchronously invoke the function on a Frontend Worker. The Frontend Worker follows the synchronous invocation call pattern we covered above.

While serverless platforms like AWS Lambda remove infrastructure management, they have some limitations. We should weigh the tradeoffs between PaaS, CaaS, and FaaS.

Limitations

FaaS and BaaS abstract away infrastructure management but can make debugging challenging. When errors occur, debugging is difficult without visibility into the underlying implementation. Limited logs may not provide enough context to identify root causes.

It's tempting to redesign everything as serverless but avoid over-engineering. Abuse of small functions leads to complex architectures that are hard to maintain. For example, Amazon Prime Video returned to monolithic architecture and saved 90% in costs. Our free newsletter covered this before. The lesson is that architecture changes with time and requirements. We should adopt the "serverless first" mindset but not default to "serverless only."

Runtime environments also differ across cloud providers. Code working on AWS Lambda may not work on Google Cloud Functions. While open-source cross-vendor frameworks help, some vendor lock-in is inevitable.

Serverless Framework

The Serverless Framework (serverless.com), initially released in 2015, was the first open-source, cross-cloud serverless deployment tool. It provides a CLI and hosted dashboard to streamline developing, deploying, and operating serverless architectures.

Key features:

- It supports all major cloud providers.
- It enables deploying to multiple clouds for redundancy.

- It works with many runtimes, including Nodes.js, Python, Java, Go, and more.

These frameworks lower the serverless adoption barrier, especially for larger companies. They cultivate an ecosystem of plugins and integrations that benefit companies of all sizes. The open-source nature also ensures portability across cloud vendors. These frameworks reduce vendor lock-in risks and simplify leveraging serverless architectures.

Summary

Serverless computing has revolutionized cloud application development over the past decade. By abstracting away infrastructure management, the serverless model enables developers to focus on writing code and business logic. The combination of event-driven FaaS and managed BaaS simplifies building full-stack applications with minimal ops overhead.

The auto-scaling, pay-per-use aspects of serverless have proven ideal for startups looking to build and iterate on products quickly. The ability to leverage managed services from cloud providers also accelerates time-to-market. While limitations around debuggability and vendor lock-in persist, open-source tools continue to evolve to mitigate these tradeoffs.

As serverless frameworks and best practices mature, more large enterprises will adopt serverless where appropriate. Striking the right balance between serverless and traditional architectures will be key for optimizing development velocity, operational overhead, and costs. Serverless is an indispensable part of the cloud-native toolkit for developers.

125 Likes · 13 Restacks

1 Comment

Write a comment...

1 more comment...