

Engineer

serverless

running code without managing the servers it runs on



What it is

Serverless is a way of running code in which the team writes and deploys the application and a provider runs, scales and maintains the machines underneath it. The CNCF Serverless Working Group's whitepaper, which uses the longer name serverless computing, describes it as the idea of building and running applications without anyone on the team having to manage a server.

The name overstates the case, and the whitepaper says so in plain words: "Servers are still required to run a serverless platform." The servers still exist, and so does operations work. What changes is who does it. The team no longer chooses machine sizes, installs the operating system, patches it or decides how many copies to run. It hands over code or a container image and a description of when that code should run.

The whitepaper divides the field into two parts. Functions-as-a-service, or FaaS, is event-driven computing: small pieces of logic that run in response to something happening. The thing that starts a function is called a trigger, and it is usually the result of an event such as an incoming HTTP request or a change in a database. Backend-as-a-service, or BaaS, is the other part: third-party services, reached through an API, that replace whole parts of an application's functionality. Functions are expected to be stateless and short-lived, so the BaaS side is where the state lives.

Two products show the range. AWS Lambda describes itself as a serverless compute service that runs code without provisioning or managing servers, and its original form is the event function: a handler plus a trigger. Azure Container Apps calls itself a service that runs application containers while the provider operates the infrastructure. The first takes a function, the second takes a whole container, and both leave the machines to the provider.

Use the concept in a decision

Put “What is serverless?” to work in a real decision. We can help relate the concept to the system you are building or buying. Servers sized for what you actually use, moved into the cloud or back out of it, with the monthly bill worked out first.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Ran 20 Docker apps on Google Cloud and AWS — 20 containerised applications deployed and watched

Ran 20 Docker apps on Google Cloud and AWS Get in touch →

Why it matters

The billing model is the first reason teams look at it. The whitepaper describes serverless pricing as pay-as-you-go, charging only for the time code is running, with no charge when nothing is running. Lambda bills an event function per request plus the execution time it uses, measured in gigabyte-seconds. A service that is busy for an hour a day pays for that hour, not for a server waiting through the other twenty-three.

The second reason is scaling in both directions. Lambda creates and destroys execution environments as traffic rises and falls, and the whitepaper describes functions whose instances drop to zero when there is no work. Microsoft notes that Container Apps scales on HTTP traffic, events, or CPU and memory load, and that most applications can scale to zero.

The third reason is the work that leaves the team’s hands. AWS lists server maintenance, capacity provisioning, scaling and patching among the jobs Lambda does for you. For a small team, those are the chores that fill a week without shipping anything, and with a serverless platform they become the provider’s responsibility.

The whitepaper’s use cases show where this fits best: reacting to database changes, processing sensor input from devices, handling streams of data, and short scheduled tasks of the kind that would otherwise be a cron job. Microsoft’s list for Container Apps adds API endpoints, background processing jobs and microservices.

The same whitepaper also warns that serverless may look the most economical option at first, and asks teams to weigh the full range of costs before committing. Per-use billing rewards work that is bursty and idle most of the time.

Work that keeps a machine busy around the clock is a different calculation, and debugging, which the whitepaper says can be harder than on IaaS or PaaS offerings, is a cost too.

What it looks like in practice

AWS's documentation for Python handlers has an example that receives an order, writes a text receipt and stores it in an Amazon S3 bucket. A shortened version of it looks like this:

```
import os

import boto3

# Runs in the Init phase, once per execution environment.
s3_client = boto3.client("s3")


def lambda_handler(event, context):
    order_id = event["Order_id"]
    bucket_name = os.environ.get("RECEIPT_BUCKET")
    receipt_content = (
        f"OrderID: {order_id}\n"
        f"Amount: ${event['Amount']}\n"
        f"Item: {event['Item']}"
    )
    key = f"receipts/{order_id}.txt"
    s3_client.put_object(Bucket=bucket_name, Key=key, Body=receipt_content)
    return {"statusCode": 200, "message": "Receipt processed successfully"}
```

The handler takes exactly two arguments. The event carries the data to process, and a JSON event arrives as a Python dictionary, so this one expects input of the following shape:

```
{"Order_id": "12345", "Amount": 199.99, "Item": "Wireless Headphones"}
```

The context carries information about the invocation itself. If the file is called `lambda_function.py`, the handler is named `lambda_function.lambda_handler`, which is the default the Lambda console gives a new function. The bucket name comes from an environment variable rather than from the code, which is how AWS recommends passing operational settings, and the code holds no credentials: Lambda supplies them to the SDK through environment variables of its own.

The placement of the client matters. For on-demand event functions without SnapStart or Managed Instances, Lambda runs an execution environment through three phases: Init, Invoke and Shutdown. Code outside the handler runs once, during Init, and AWS limits the initial Init attempt to 10 seconds. The handler then runs for each event, within the function's timeout. When an environment is reused for the next event, which AWS calls a warm start, the client created during Init is still there and is not built again.

Serverless does not have to mean functions. On this site, the portfolio entry on provisioning an Azure estate as code lists two services, `api` and `www`, deployed to Azure Container Apps, with their container images kept in the project's own Azure Container Registry. Microsoft calls Container Apps a serverless platform for containers, so those two services are where that estate is serverless. Every environment is described by one set of Bicep templates that take parameters, which is why the entry can treat a new environment as a pull request.

Going the other way can be the right call too. The entry on reviewing infrastructure and moving workloads off AWS starts from an AWS setup sized far beyond what the business used and still carrying services nobody needed. The unused services were removed, the remaining capacity was adjusted, and the workloads moved off AWS. Serverless billing is one way to stop paying for idle capacity; reviewing and removing unused infrastructure is another. A comparison of the real load and costs would be needed to decide which costs less; the entry records changes rather than a like-for-like cost ratio.

Where it goes wrong

Most serverless failures come from treating a function like a small server that is always on, which it is not.

Cold starts add delay to some requests. When no environment is ready, Lambda has to download the code, set up an environment and run the Init code before the handler sees the event. AWS says cold starts typically affect under 1% of invocations and last from under 100 milliseconds to over a second, that they are more common in development and test functions, and that the Init time is billed. The fix is to keep Init small, importing only what the function needs and loading rarely used objects lazily. Where a delay matters, provisioned concurrency keeps a chosen number of environments initialised in advance.

State kept inside the environment leaks or disappears. Each invocation runs independently, and an environment may or may not be reused. AWS also replaces environments every few hours for runtime updates, even under constant traffic, and a reset after a crash or a timeout does not clear the `/tmp` directory. Data from one event can therefore surface during another, or vanish without warning. The fix is to keep user data and events out of the environment altogether, and to put anything that must persist in a database or storage service.

The same event is processed twice. A function can receive duplicate events, and AWS's best-practice list asks for idempotent code that validates each event and handles a duplicate the same way as the original. In the example above the receipt key is derived from the order id, so a second run overwrites the same object instead of creating a second receipt. Any side effect that cannot be repeated safely, such as sending an email or charging a card, needs its own record of what was already done.

Hard limits end the work mid-way. For event functions outside Lambda Managed Instances, an invocation can run for at most 900 seconds, or 15 minutes. A synchronous request and a buffered synchronous response are each limited to 6 MB, an asynchronous request to 1 MB, and environment variables to 4 KB in total. Exceeding a limit is an error rather than a slowdown. The fix is to design around the limits from the start: split long jobs into steps, keep large data in storage and pass a reference to it, and move work that does not fit to a container platform or a virtual machine.

Concurrency runs out before traffic does. Lambda's default quota is 1,000 concurrent executions per Region, while API Gateway's default allows 10,000 requests per second, so the front door can accept more requests than the functions behind it can run. New accounts start with reduced concurrency and memory quotas, which AWS raises automatically based on usage. The fix is to request an increase before launch and to load test at the expected peak.

A function that triggers itself runs up the bill. A function that invokes itself, or starts a process that invokes it again, can produce a flood of invocations, and every one is billed. AWS's advice is to avoid recursive invocations, and if an unexpected volume appears, to set the function's reserved concurrency to `0` at once, which throttles all invocations until the code is fixed.

The programming model ties you to one platform. The whitepaper names the platform's programming model, its event and messaging interface and its BaaS offerings as sources of lock-in. The fix is to keep the business logic in ordinary functions that the handler only calls, and to prefer platforms such as Container Apps that run a standard container image from any registry when moving later is a real possibility.

Further reading

- CNCF Serverless Whitepaper v1.0 — the working group's definitions of serverless, FaaS and BaaS, and drawbacks.
- What is AWS Lambda? — what Lambda manages for you, its function types and its billing model.
- Define Lambda Event Function handler in Python — the receipt example in full, handler naming and the code best practices.
- Understanding the Lambda execution environment lifecycle — the Init, Invoke and Shutdown phases, cold starts and environment reuse.
- Lambda quotas — concurrency, memory, timeout and payload limits, and how to raise them.
- Azure Container Apps overview — Microsoft's serverless platform for containers and how it scales to zero.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is serverless?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/serverless/