Serverless Computing

Serverless computing is a cloud model where you write and deploy code **without managing servers**. The cloud provider handles provisioning, scaling, patching, and availability—you just focus on the logic.

Think of it as: *"I write functions. The cloud runs them when needed."*

What "serverless" actually means (spoiler: servers still exist)

- Servers **do exist**, but you don't see or manage them
- No VM setup, no capacity planning, no OS maintenance
- You're billed **only when your code runs**

How It Works

You write small **functions** (often called "serverless functions" or "cloud functions") that perform specific tasks. These functions are **triggered** by events like HTTP requests, database changes, file uploads, or scheduled timers. The cloud provider executes your function, **scales** it automatically based on demand, and shuts it down when not in use.

Core building block: Function as a Service (FaaS)

You deploy small, event-driven functions.

Popular examples:

- **AWS Lambda**
- **Azure Functions**
- **Google Cloud Functions**

A function runs when an **event** happens:

- HTTP request
- File upload
- Message in a queue
- Database change
- Cron schedule

Major Serverless Platforms

AWS Lambda - Amazon's serverless platform, the most widely used

Google Cloud Functions - Google's offering

Azure Functions - Microsoft's serverless solution

Cloudflare Workers - Edge computing serverless platform

Vercel/Netlify Functions - Popular for web applications

Typical serverless architecture

arduino Copy code

```
Client → API Gateway → Serverless Function → DB / Queue / Other Services
```

Example:

- User hits `/login`
- API Gateway triggers a Lambda
- Lambda validates user
- Reads/writes to DynamoDB
- Returns response

Types of Serverless

Function as a Service (FaaS) The most common type—individual functions that execute in response to events. Examples: AWS Lambda, Google Cloud Functions.

Backend as a Service (BaaS) Pre-built backend services like authentication, databases, storage that you use without managing servers. Examples: Firebase, AWS Amplify, Supabase.

Serverless Containers Run containerized applications without managing the underlying infrastructure. Examples: AWS Fargate, Google Cloud Run.

Good Fit Use Cases

API backends Build REST or GraphQL APIs without managing web servers. Each endpoint can be a separate function.

Data processing Process files, transform data, generate reports when events occur (like file uploads or database changes).

Scheduled tasks Run cron jobs, send scheduled emails, perform nightly backups without a server running 24/7.

Real-time processing Process streaming data, handle webhooks, respond to IoT device events.

Image/video processing Resize images, generate thumbnails, transcode videos on-demand.

Chatbots and voice assistants Handle requests from Slack, Alexa, or other platforms.

Event pipelines (Kafka, SQS, Pub/Sub)

Where serverless struggles

❌ Long-running processes
❌ Heavy CPU / GPU workloads
❌ Complex stateful workflows (unless you use orchestrators like Step Functions)
❌ Cold start latency (small delay on first request)

Key characteristics

1. Event-driven

2. Auto-scaling

3. Pay-per-use

You pay for:

- Execution time
- Memory used
- Number of invocations

No traffic = almost no cost 💸

4. Stateless

- Each function execution is isolated
- State is stored externally (DB, cache, object storage)

Advantages

1. **No server management**
2. **Automatic scaling**
3. **Pay-per-use pricing** Only pay for actual execution time
4. **Faster time to market** Focus purely on code, not infrastructure. Deploy features more quickly.
5. **Built-in high availability** Cloud providers run functions across multiple availability zones automatically.
6. **Reduced operational overhead** No servers to monitor, no OS updates, no capacity management.

Disadvantages

1. **Cold starts** When a function hasn't run recently, there's latency while the provider spins up the execution environment (can be 100ms to several seconds).
2. **Execution time limits** Most platforms have maximum execution times (AWS Lambda: 15 minutes). Not suitable for long-running processes.
3. **Vendor lock-in** Code often becomes tightly coupled to a specific cloud provider's services and APIs.
4. **Debugging challenges** Harder to debug and test locally since you're running in the cloud provider's environment.
5. **Limited control** Can't customize the underlying infrastructure, runtime environment, or network configuration much.
6. **Cost unpredictability** For high-traffic applications, costs can exceed traditional servers. Need to monitor carefully.
7. **Statelessness** Functions are stateless—they don't retain data between invocations. Must use external storage for persistence.
8. **Resource constraints** Limited memory, CPU, and disk space per function execution.

Best Practices

1. **Keep functions small and focused** Each function should do one thing well. Makes them easier to test, maintain, and reuse.
2. **Handle cold starts** Use provisioned concurrency for latency-sensitive functions, or architect to minimize cold start impact.
3. **Use environment variables** Store configuration separately from code for flexibility and security.
4. **Implement proper error handling** Functions should handle errors gracefully and retry logic when appropriate.
5. **Monitor and log** Use cloud provider tools (CloudWatch, etc.) to track performance and debug issues.
6. **Optimize for cost** Monitor usage, right-size memory allocation, and consider alternatives for steady high-traffic endpoints.

Serverless Architecture Example

Imagine building a photo-sharing app:

- **Upload handler** (Lambda): User uploads photo → function validates and stores in S3
- **Image processor** (Lambda): S3 upload triggers function → generates thumbnails
- **API functions** (Lambda): Handle GET/POST requests for photos
- **Database** (DynamS or Firebase): Serverless database stores metadata
- **Authentication** (Auth0 or Cognito): Serverless auth service
- **CDN** (CloudFront): Serves images globally

Each piece scales independently, you pay only for usage, and there are no servers to manage.