# What is AWS Lambda?

You can use AWS Lambda to run code without provisioning or managing servers. Lambda runs your code on a high-availability compute infrastructure and manages all the computing resources, including server and operating system maintenance, capacity provisioning, automatic scaling, and logging. You organize your code into Lambda functions.
When using Lambda, you are responsible only for your code. Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources to run your code. Because Lambda manages these resources, you cannot log in to compute instances or customize the operating system on provided runtimes.

## When to use Lambda

- **Stream processing:** Use Lambda and Amazon Kinesis to process real-time streaming data for application activity tracking, transaction order processing, clickstream analysis, data cleansing, log filtering, indexing, social media analysis, Internet of Things (IoT) device data telemetry, and metering.
- **Web applications:** Combine Lambda with other AWS services to build powerful web applications that automatically scale up and down and run in a highly available configuration across multiple data centers
- **Mobile backends:** Build backends using Lambda and Amazon API Gateway to authenticate and process API requests. Use AWS Amplify to easily integrate with your iOS, Android, Web, and React Native frontends.
- **IoT Backend:** Build serverless backends using Lambda to handle web, mobile, IoT, and third-party API requests.
- **File processing :** Use Amazon Simple Storage Service (Amazon S3) to trigger Lambda data processing in real time after an upload.
- **Database Operations and Integration:** Use Lambda to process database interactions both reactively and proactively, from handling queue messages for Amazon RDS operations like user registrations and order submissions, to responding to DynamoDB changes for audit logging, data replication, and automated workflows.
- **Scheduled and Periodic Task:** Use Lambda with EventBridge rules to execute time-based operations such as database maintenance, data archiving, report generation, and other scheduled business processes using cron-like expressions.

# How Lambda works

Because Lambda is a serverless, event-driven compute service, it uses a different programming paradigm than traditional web applications. The following model illustrates how Lambda fundamentally works:

1. You write and organize your code in Lambda functions, which are the basic building blocks you use to create a Lambda application.
2. You control security and access through Lambda permissions, using execution roles to manage what AWS services your functions can interact with and what resource policies can interact with your code.
3. Event sources and AWS services trigger your Lambda functions, passing event data in JSON format, which your functions process (this includes event source mappings).
4. Lambda runs your code with language-specific runtimes (like Node.js and Python) in execution environments that package your runtime, layers, and extensions.

## 1. Lambda functions and function handlers

In Lambda, functions are the fundamental building blocks you use to create applications. A **Lambda function** is a piece of code that runs in response to events, such as a user clicking a button on a website or a file being uploaded to an Amazon Simple Storage Service (Amazon S3) bucket. You can think of a function as a kind of self-contained program with the following properties. A **Lambda function handler** is the method in your function code that processes events. When a function runs in response to an event, Lambda runs the function handler. Data about the event that caused the function to run is passed directly to the handler. While the code in a Lambda function can contain more than one method or function, Lambda functions can only have one handler.

To create a Lambda function, you bundle your function code and its dependencies in a deployment package. Lambda supports two types of deployment package, .zip file archives and container images.

- A function has one specific job or purpose
- They run only when needed in response to specific events
- They automatically stop running when finished

## 2. Lambda execution environment and runtimes

Lambda functions run inside a secure, isolated execution environment which Lambda manages for you. This execution environment manages the processes and resources that are needed to run your function. When a function is first invoked, Lambda creates a new execution environment for the function to run in. After the function has finished running, Lambda doesn't stop the execution environment right away; if the function is invoked again, Lambda can re-use the existing execution environment.
The Lambda execution environment also contains a runtime, a language-specific environment that relays event information and responses between Lambda and your function. Lambda provides a number of managed runtimes for the most popular programming languages, or you can create your own.
For managed runtimes, Lambda automatically applies security updates and patches to functions using the runtime.

## 3. Events and triggers

You can also invoke a Lambda function directly by using the Lambda console, AWS CLI, or one of the AWS Software Development Kits (SDKs). It's more usual in a production application for your function to be invoked by another AWS service in response to a particular event. For example, you might want a function to run whenever an item is added to an Amazon DynamoDB table.
To make your function respond to events, you set up a trigger. A trigger connects your function to an event source, and your function can have multiple triggers. When an event occurs, Lambda receives event data as a JSON document and converts it into an object that your code can process. You might define the following JSON format for your event and the Lambda runtime converts this JSON to an object before passing it to your function's handler.

## 4. Lambda permissions and roles

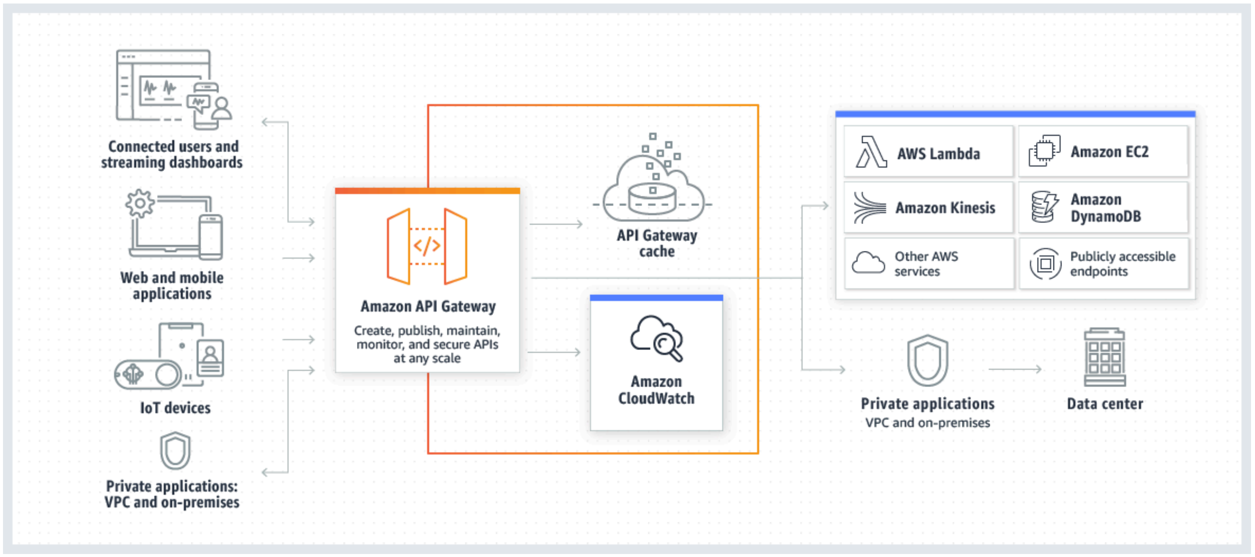
For Lambda, there are two main types of permissions that you need to configure:

- Permissions that your function needs to access other AWS services
- Permissions that other users and AWS services need to access your function

# What is Amazon API Gateway?

Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale. API developers can create APIs that access AWS or other web services, as well as data stored in the AWS Cloud. you can create APIs for use in your own client applications. Or you can make your APIs available to third-party app developers.

## 1. Architecture of API Gateway


Connected users and streaming dashboards
Web and mobile applications
IoT devices
Private applications: VPC and on-premises
Amazon API Gateway
Create, publish, maintain, monitor, and secure APIs at any scale
API Gateway cache
Amazon CloudWatch
AWS Lambda
Amazon EC2
Amazon Kinesis
Amazon DynamoDB
Other AWS services
Publicly accessible endpoints
Private applications
VPC and on-premises
Data center

This diagram illustrates how the APIs you build in Amazon API Gateway provide you or your developer customers with an integrated and consistent developer experience for building AWS serverless applications. API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls. These tasks include traffic management, authorization and access control, monitoring, and API version management.
API Gateway acts as a "front door" for applications to access data, business logic, or functionality from your backend services, such as workloads running on Amazon Elastic Compute Cloud (Amazon EC2), code running on AWS Lambda, any web application, or real-time communication applications.

## 2. Features of API Gateway

Amazon API Gateway offers features such as the following:

- Support for stateful (WebSocket) and stateless (HTTP and REST) APIs.
- Powerful, flexible authentication mechanisms, such as AWS Identity and Access Management policies, Lambda authorizer functions, and Amazon Cognito user pools.
- Canary release deployments for safely rolling out changes.
- CloudTrail logging and monitoring of API usage and API changes.
- CloudWatch access logging and execution logging, including the ability to set alarms.
- Integration with AWS WAF for protecting your APIs against common web exploits.
- Integration with AWS X-Ray for understanding and triaging performance latencies.

## 3. Types of APIs:

---

### 1. HTTP API (Modern, Lightweight)

**What it is:**

- A newer, simpler, faster version of REST API.
- Designed for low-latency use cases with basic routing and Lambda/HTTP backends.

**Key Features:**

- Supports JWT authorizers (like Cognito).
- Native integration with AWS Lambda, HTTP backends.
- Much faster and cheaper than REST APIs.
- Simplified configuration — less overhead.

**Limitations:**

- No usage plans / API keys.
- No request/response transformation
- No integration with AWS service (like DynamoDB directly).

**Use Cases:**

- Microservices
- Simple REST APIs
- Serverless applications

---

## 2. REST API (Classic, Full-Featured)

**What it is:**

- The original and full-featured API Gateway.
- Supports more advanced features than HTTP API.

**Key Features:**

- Request and response transformations using Velocity Template Language (VTL).
- API keys, usage plans, throttling, quotas.
- Custom authorizers (Lambda-based), IAM, Cognito.
- Supports AWS Service integrations (e.g., S3, DynamoDB).
- Fine-grained control of caching, logging, headers, etc.

**Downsides**:

- Higher latency and cost compared to HTTP API.
- More complex to configure.

**Use Cases:**

- External public APIs
- Legacy APIs
- APIs needing fine-tuned transformations or service integrations

## 3. WebSocket API (Real-Time, Bidirectional)

**What it is:**

- Designed for real-time, bidirectional communication over a persistent WebSocket connection.
- Unlike HTTP/REST, it maintains an open connection.

**Key Features:**

- Supports connect, disconnect, and message routes.
- Push messages from backend to clients.
- State is maintained through connection IDs.

**Limitations:**

- More complex to build.
- Limited to apps that really need long-lived connections.

**Use Cases:**

- Chat apps
- Real-time notifications
- Online gaming
- Collaborative tools (e.g., whiteboards)

### 🔗 Example flow:

```
Client <--- WebSocket ---> API Gateway ---> Lambda
                     ↑
        Sends msg back to client
```

---

## 4. Private REST API (Internal-Only)

**What it is:**

- A REST API that is only accessible from within your VPC.
- Not exposed to the public internet.
- Uses VPC endpoints (powered by AWS PrivateLink).

**Key Features:**

- Only accessible from:
  - VPCs
  - Direct Connect
  - Peered VPCs
- No public exposure
- Full REST API feature set

**Limitations:**

- Not accessible from internet unless routed via proxy or VPN.
- Setup involves VPC endpoint and routing rules.

**Use Cases:**

- Internal microservices
- Backend APIs between services in your AWS account
- Compliance and security-sensitive workloads

## Choosing the Right API Type

| You want... | Use This |
|---|---|
| Fast, cheap, simple Lambda-based API | HTTP API |
| Full control, custom auth, legacy | REST API |
| Real-time bidirectional communication | WebSocket API |
| Secure internal-only API | Private REST API |

## Comparison Table

| Feature | HTTP API | REST API | WebSocket API | Private REST API |
|---|---|---|---|---|
| API Type | HTTP | HTTP | WebSocket | HTTP |
| Pricing | ✅ Lower cost | ❌ Higher cost | ⚖️ Medium | ❌ Higher |
| Performance | ✅ Faster | ❌ Slower | ✅ Persistent | ❌ Slightly slower |
| Request/Response Transformation | ❌ Limited | ✅ Full VTL | ❌ Not applicable | ✅ Full VTL |
| API Keys/Usage Plans | ❌ No | ✅ Yes | ❌ No | ✅ Yes |
| JWT/Cognito Auth | ✅ Yes | ✅ Yes | ✅ Custom | ✅ Yes |
| AWS IAM Auth | ✅ Yes | ✅ Yes | ❌ No | ✅ Yes |
| AWS Service Integration | ❌ No | ✅ Yes | ❌ No | ✅ Yes |
| VPC Access | ❌ No | ✅ (public) | ❌ No | ✅ Private only |
| Ideal Use | Serverless apps | Public APIs | Real-time apps | Internal APIs |

# API endpoint types for REST APIs in API Gateway

## 1. Edge-optimized API endpoints

An edge-optimized API endpoint typically routes requests to the nearest CloudFront Point of Presence (POP), which could help in cases where your clients are geographically distributed. This is the default endpoint type for API Gateway REST APIs.
Edge-optimized APIs capitalize the names of HTTP headers (for example, Cookie).
CloudFront sorts HTTP cookies in natural order by cookie name before forwarding the request to your origin. For more information about the way CloudFront processes cookies, see Caching Content Based on Cookies.
Any custom domain name that you use for an edge-optimized API applies across all regions.

## 2. Regional API endpoints

A Regional API endpoint is intended for clients in the same region. When a client running on an EC2 instance calls an API in the same region, or when an API is intended to serve a small number of clients with high demands, a Regional API reduces connection overhead.
For a Regional API, any custom domain name that you use is specific to the region where the API is deployed. If you deploy a regional API in multiple regions, it can have the same custom domain name in all regions. You can use custom domains together with Amazon Route 53 to perform tasks such as latency-based routing. For more information, see Set up a Regional custom domain name in API Gateway and Set up an edge-optimized custom domain name in API Gateway.
Regional API endpoints pass all header names through as-is.

## 3. Private API endpoints

A private API endpoint is an API endpoint that can only be accessed from your Amazon Virtual Private Cloud (VPC) using an interface VPC endpoint, which is an endpoint network interface (ENI) that you create in your VPC. For more information, see Private REST APIs in API Gateway.
Private API endpoints pass all header names through as-is.

# Choose between REST APIs and HTTP APIs

REST APIs and HTTP APIs are both RESTful API products. REST APIs support more features than HTTP APIs, while HTTP APIs are designed with minimal features so that they can be offered at a lower price. Choose REST APIs if you need features such as **API keys, per-client throttling, request validation, AWS WAF integration, or private API endpoints.** Choose HTTP APIs if you don't need the features included with REST APIs.

The following sections summarize core features that are available in REST APIs and HTTP APIs.

## Endpoint type

The endpoint type refers to the endpoint that API Gateway creates for your API.

| Endpoint types | REST API | HTTP API |
|---|---|---|
| Edge-optimized | Yes | No |
| Regional | Yes | Yes |
| Private | Yes | No |

## Security

API Gateway provides a number of ways to protect your API from certain threats, like malicious actors or spikes in traffic.

| Security features | REST API | HTTP API |
|---|---|---|
| Mutual TLS authentication | Yes | Yes |
| Certificates for backend authentication | Yes | No |
| AWS WAF | Yes | No |

## Authorization

API Gateway supports multiple mechanisms for controlling and managing access to your API.

| Authorization options | REST API | HTTP API |
|---|---|---|
| IAM | Yes | Yes |
| Resource policies | Yes | No |
| Amazon Cognito | Yes | Yes 1 |
| Custom authorization with an AWS Lambda function | Yes | Yes |
| JSON Web Token (JWT) 2 | No | Yes |

1 You can use Amazon Cognito with a JWT authorizer.
2 You can use a Lambda authorizer to validate JWTs for REST APIs.

## API management

Choose REST APIs if you need API management capabilities such as API keys and per-client rate limiting

| Features | REST API | HTTP API |
|---|---|---|
| Custom domains | Yes | Yes |
| API keys | Yes | No |
| Per-client rate limiting | Yes | No |
| Per-client usage throttling | Yes | No |

## Monitoring

API Gateway supports several options to log API requests and monitor your APIs.

| Feature | REST API | HTTP API |
|---|---|---|
| Amazon CloudWatch metrics | Yes | Yes |
| Access logs to CloudWatch Logs | Yes | Yes |
| Access logs to Amazon Data Firehose | Yes | No |
| Execution logs | Yes | No |
| AWS X-Ray tracing | Yes | No |

## Integrations

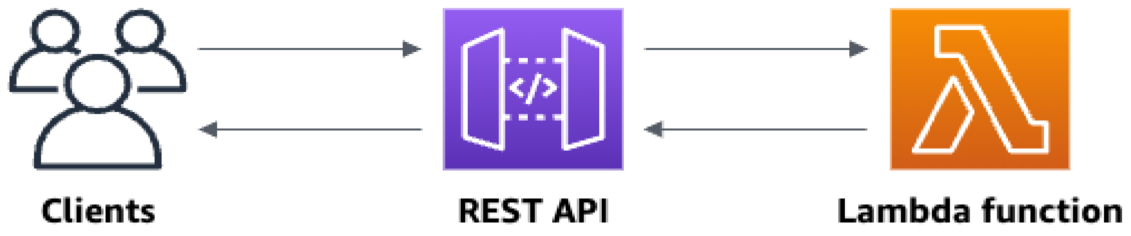
Integrations connect your API Gateway API to backend resources.

| Feature | REST API | HTTP API |
|---|---|---|
| Public HTTP endpoints | Yes | Yes |
| AWS services | Yes | Yes |
| AWS Lambda functions | Yes | Yes |
| Private integrations with Network Load Balancers | Yes | Yes |
| Private integrations with Application Load Balancers | No | Yes |
| Private integrations with AWS Cloud Map | No | Yes |
| Mock integrations | Yes | No |

# REST API + LAMBDA

First, you create a Lambda function using the Lambda console. Next, you create a REST API using the API Gateway REST API console. Then, you create an API method and integrate it with a Lambda function using a Lambda proxy integration. Finally, you deploy and invoke your API.

When you invoke your REST API, API Gateway routes the request to your Lambda function. Lambda runs the function and returns a response to API Gateway. API Gateway then returns that response to you.



To complete this exercise, you need an AWS account and an AWS Identity and Access Management (IAM) user with console access.

## Step 1: Create a Lambda function

You use a Lambda function for the backend of your API. Lambda runs your code only when needed and scales automatically, from a few requests per day to thousands per second. For this exercise, you use a default Node.js function in the Lambda console.

*To create a Lambda function*

1. Sign in to the Lambda console at [https://console.aws.amazon.com/lambda](https://console.aws.amazon.com/lambda).
2. Choose Create function.
3. Under Basic information, for Function name, enter my-function.
4. For all other options, use the default setting.
5. Choose Create function.

**The default Lambda function code should look similar to the following:**

```
export const handler = async (event) => {
  const response = {
    statusCode: 200,
    body: JSON.stringify('The API Gateway REST API console is great!'),
  };
  return response;
};
```

## Step 2: Create a REST API

Next, you create a REST API with a root resource (/).

*To create a REST API*

1. Sign in to the API Gateway console at [https://console.aws.amazon.com/apigateway](https://console.aws.amazon.com/apigateway).
2. Do one of the following:
   - To create your first API, for REST API, choose Build.
   - If you've created an API before, choose Create API, and then choose Build for REST API.
3. For API name, enter my-rest-api.
4. (Optional) For Description, enter a description.
5. Keep API endpoint type set to Regional.
6. For IP address type, select IPv4.
7. Choose Create API.

## Step 3: Create a Lambda proxy integration

Next, you create an API method for your REST API on the root resource (/) and integrate the method with your Lambda function using a proxy integration. In a Lambda proxy integration, API Gateway passes the incoming request from the client directly to the Lambda function.

*To create a Lambda proxy integration*

1. Select the / resource, and then choose Create method.
2. For Method type, select ANY.
3. For Integration type, select Lambda.
4. Turn on Lambda proxy integration.
5. For Lambda function, enter my-function, and then select your Lambda function.
6. Choose Create method.

## Step 4: Deploy your API

Next, you create an API deployment and associate it with a stage.

*To deploy your API*

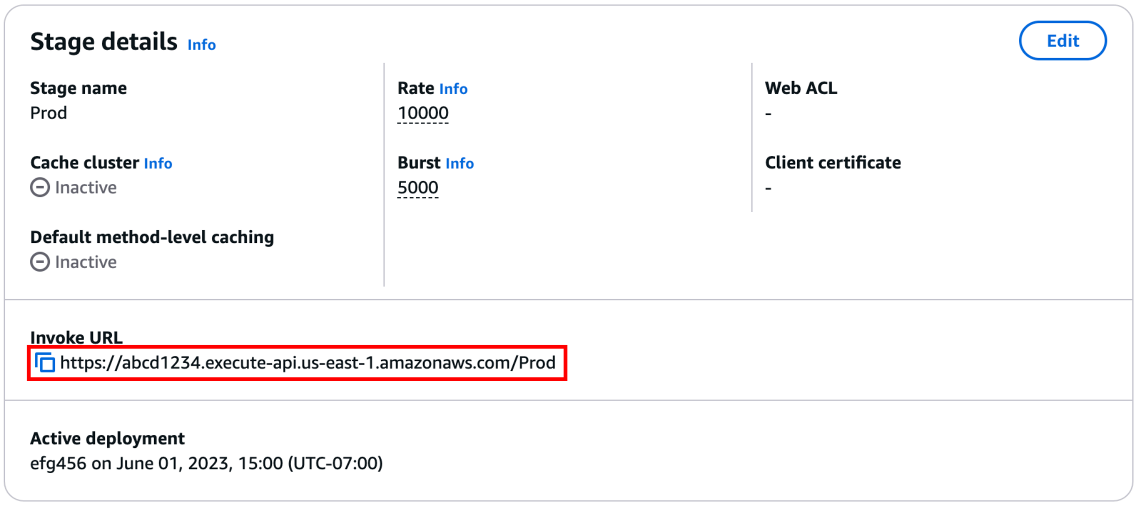
1. Choose Deploy API.
2. For Stage, select New stage.
3. For Stage name, enter Prod.
4. (Optional) For Description, enter a description.
5. Choose Deploy.

Now clients can call your API. To test your API before deploying it, you can optionally choose the ANY method, navigate to the Test tab, and then choose Test.

## Step 5: Invoke your API

*To invoke your API*

1. From the main navigation pane, choose Stage.
2. Under Stage details, choose the copy icon to copy your API's invoke URL.



Stage name: Prod
Cache cluster: Inactive
Default method-level caching: Inactive
Rate: 10000
Burst: 5000
Web ACL: -
Client certificate: -
Invoke URL: https://abcd1234.execute-api.us-east-1.amazonaws.com/Prod
Active deployment: efg456 on June 01, 2023, 15:00 (UTC-07:00)

3. Enter the invoke URL in a web browser.
   The full URL should look like
   https://abcd123.execute-api.us-east-2.amazonaws.com/Prod.
   Your browser sends a GET request to the API.
4. Verify your API's response. You should see the text "The API Gateway REST API console is great!" in your browser