Lecture #11
Serverless

WSMT2023

Introduction to Serverless

Serverless computing is a cloud computing model in which the cloud provider dynamically manages the allocation of machine resources. This eliminates the need for:

- Server administration

- Provisioning

- Maintenance

Freeing up developers to focus on their applications.

History

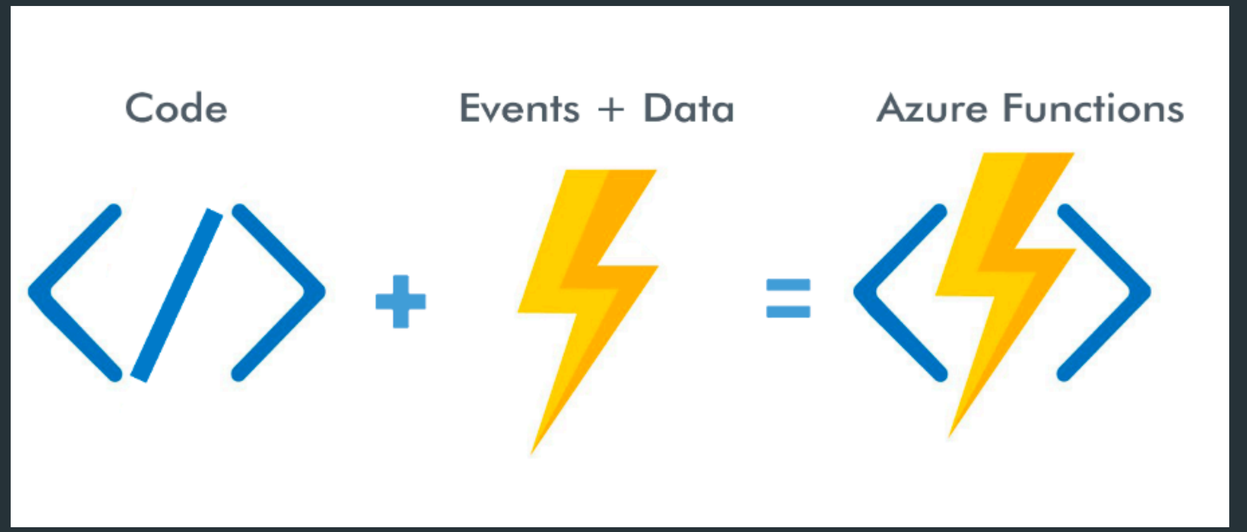
- 2008: Google App Engine.

History

- 2008: Google App Engine.
- 2010: Amazon Web Services (AWS).

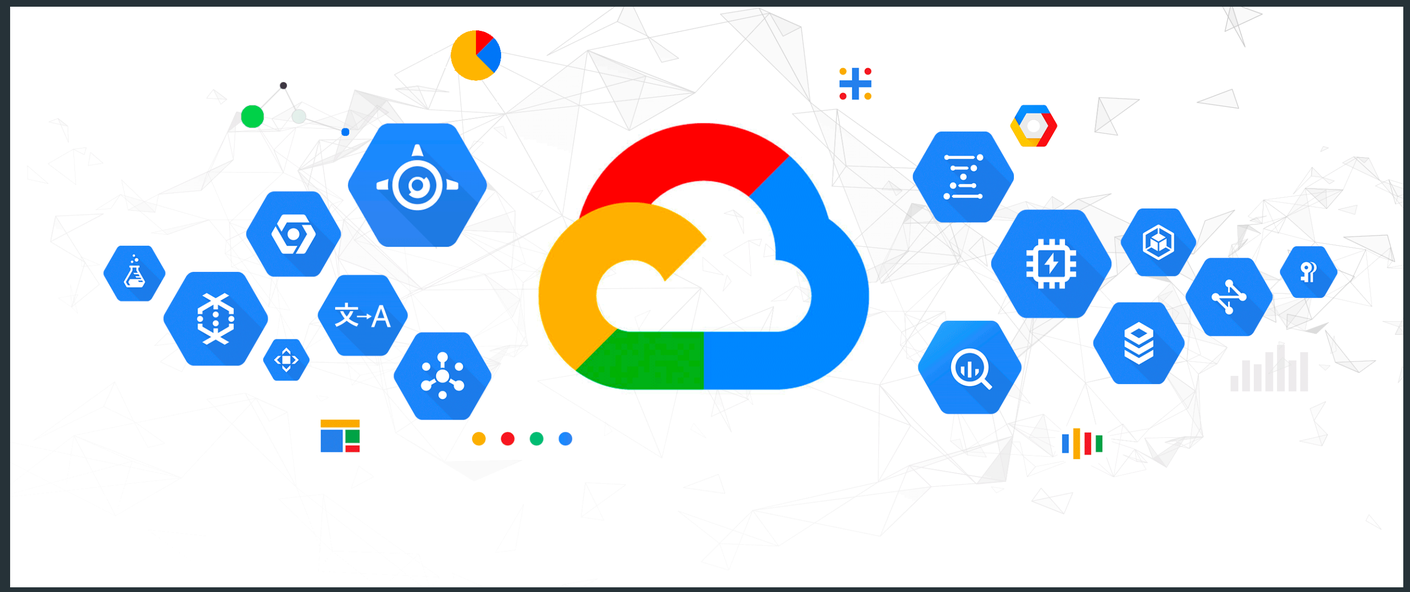
History

- 2008: Google App Engine.
- 2010: Amazon Web Services (AWS).
- 2014: Microsoft Azure launches Functions.



History

- 2008: Google App Engine.
- 2010: Amazon Web Services (AWS).
- 2014: Microsoft Azure launches Functions.
- 2015: Google Cloud Platform (GCP).



History

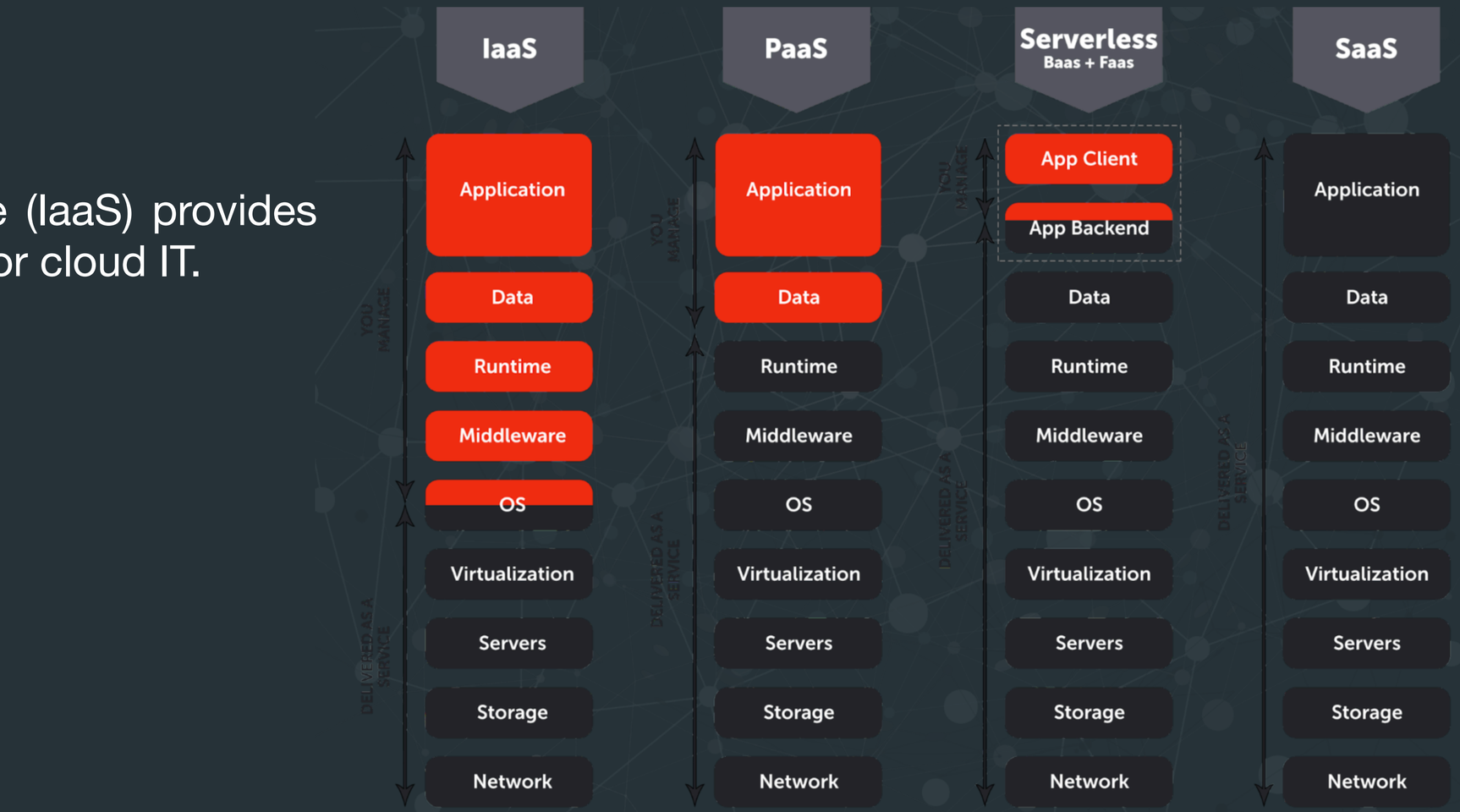
- 2008: Google App Engine.
- 2010: Amazon Web Services (AWS).
- 2014: Microsoft Azure launches Functions.
- 2015: Google Cloud Platform (GCP).
- 2016: IBM Cloud Functions.

Benefits

- It offers a number of benefits over traditional server-based computing, including:
    - Reduced costs
    - Increased scalability
    - Improved agility
- Serverless computing is a good fit for a variety of applications, including:
    - Backend services
    - Event-driven applications
    - Microservices

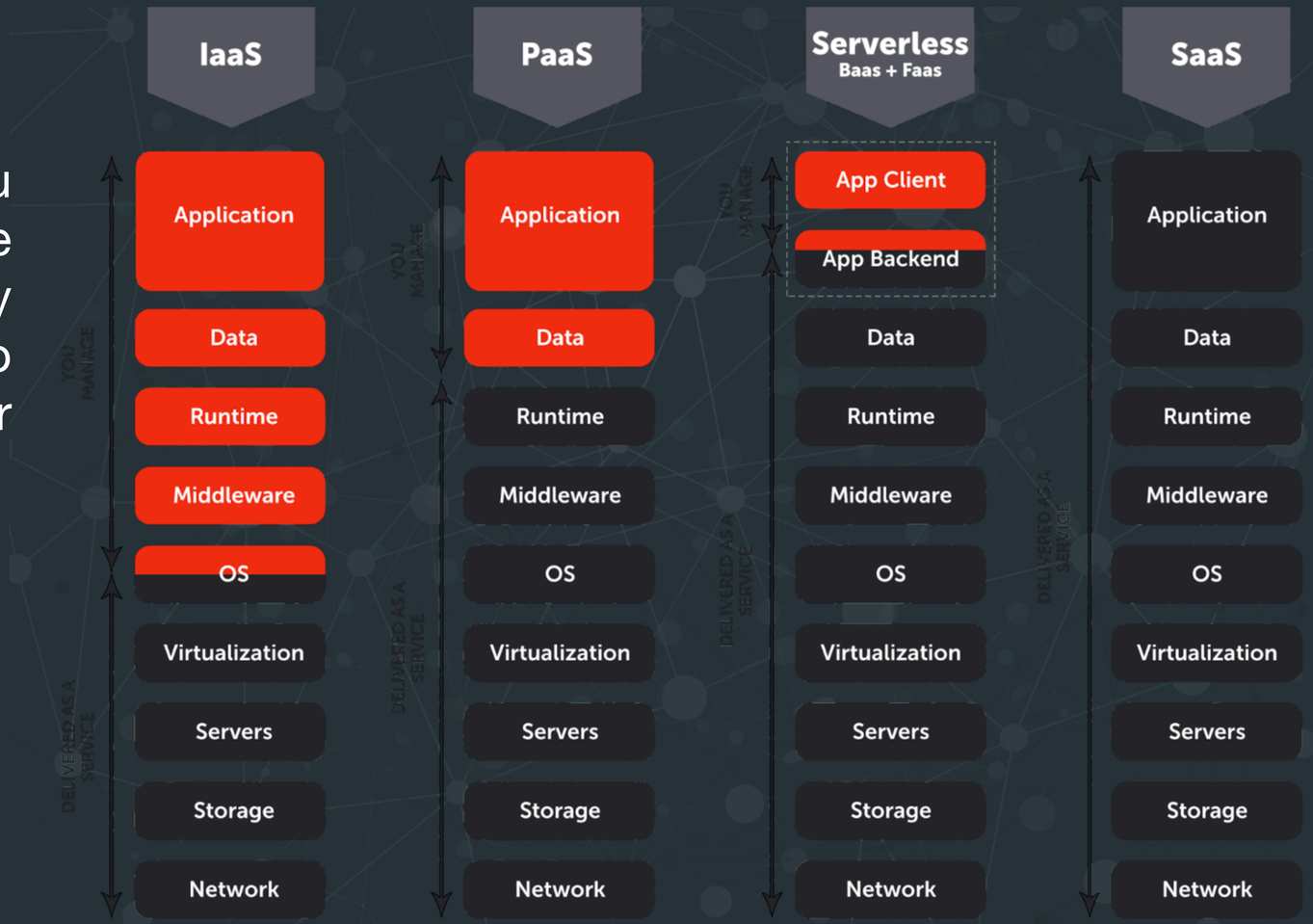
Cloud Computing Models

- Infrastructure as a Service (IaaS) provides the basic building blocks for cloud IT.



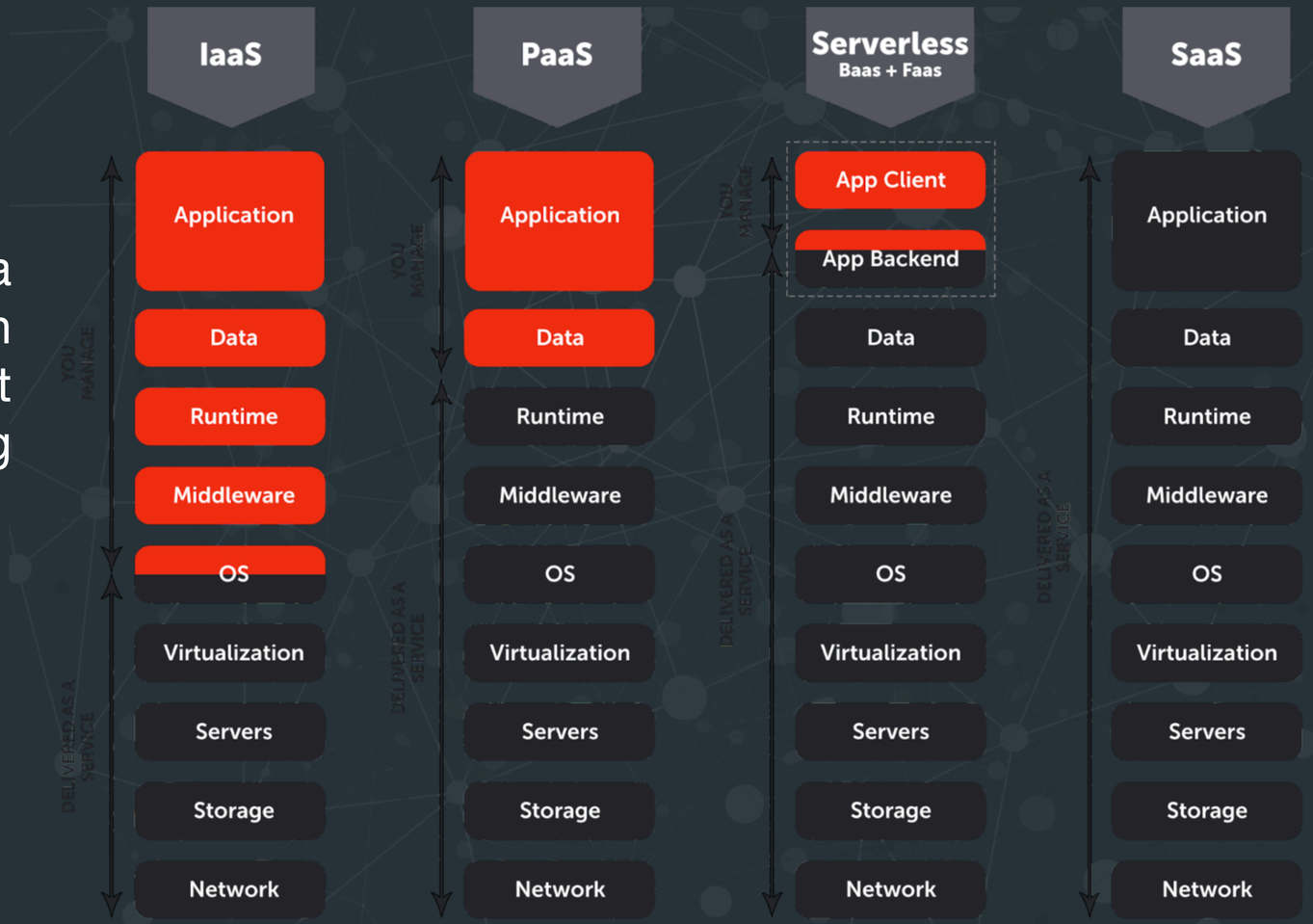
Cloud Computing Models

- Infrastructure as a Service (IaaS).
- Software as a Service (SaaS) provides you with access to a complete software application that is hosted and managed by the cloud provider. You don't need to install or maintain any software on your own computers.



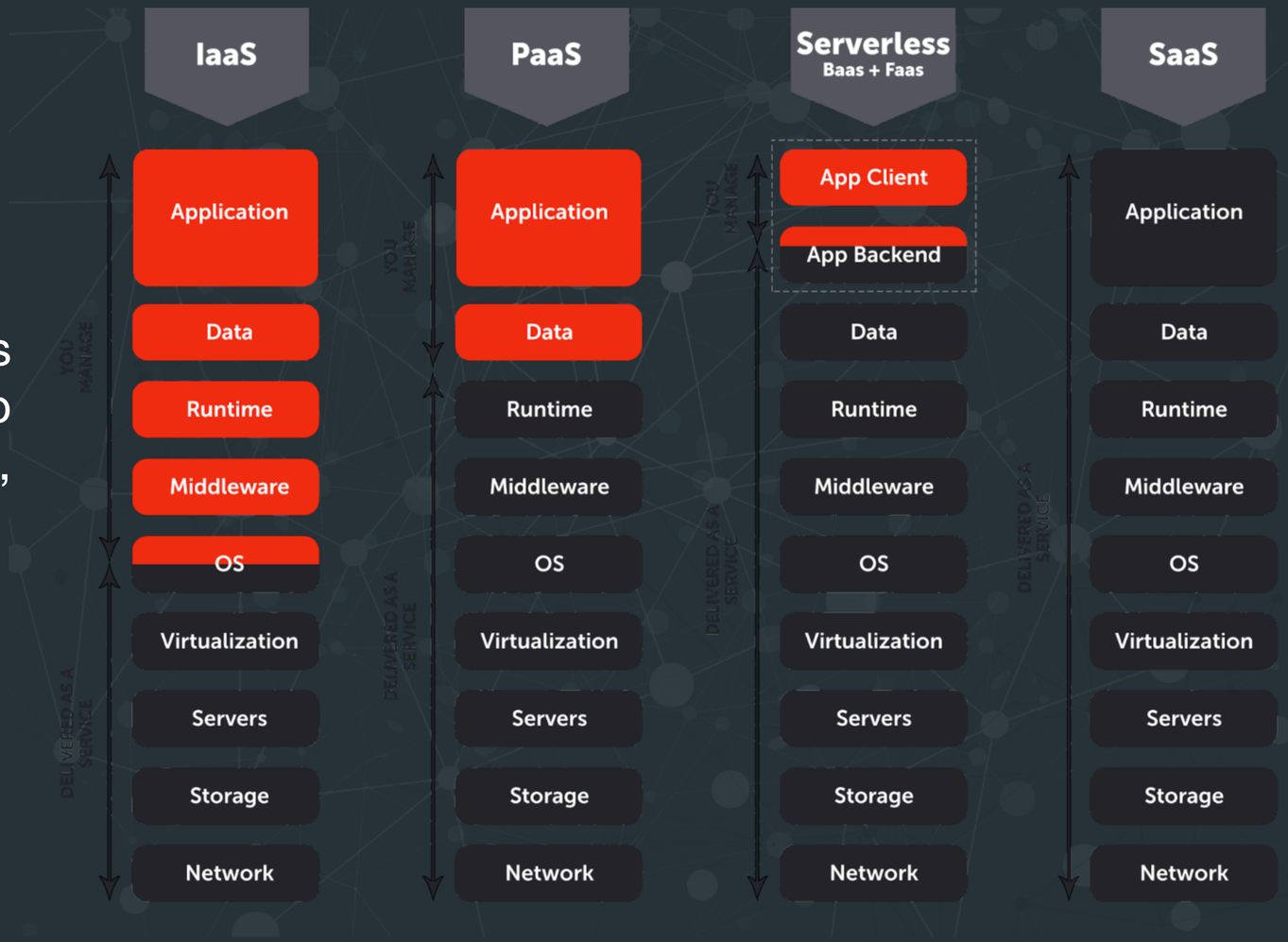
Cloud Computing Models

- Infrastructure as a Service (IaaS).
- Software as a Service (SaaS).
- Platform as a Service (PaaS) - provides a development environment where you can build, test, and deploy applications without having to worry about the underlying infrastructure.



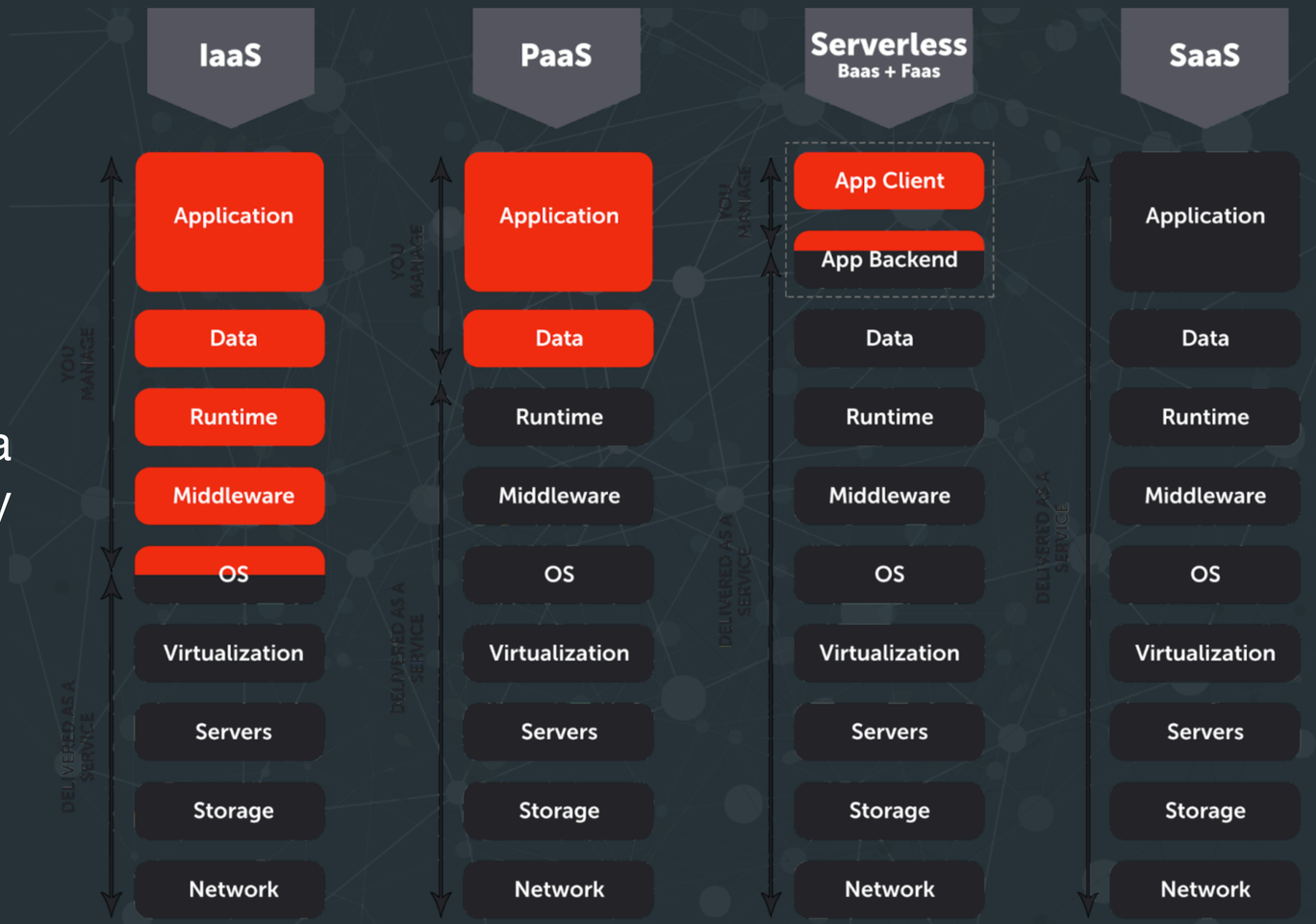
Cloud Computing Models

- Infrastructure as a Service (IaaS).
- Software as a Service (SaaS).
- Platform as a Service (PaaS).
- Backend as a Service (BaaS) provides backend services for mobile and web applications, such as user authentication, push notifications, and data storage.



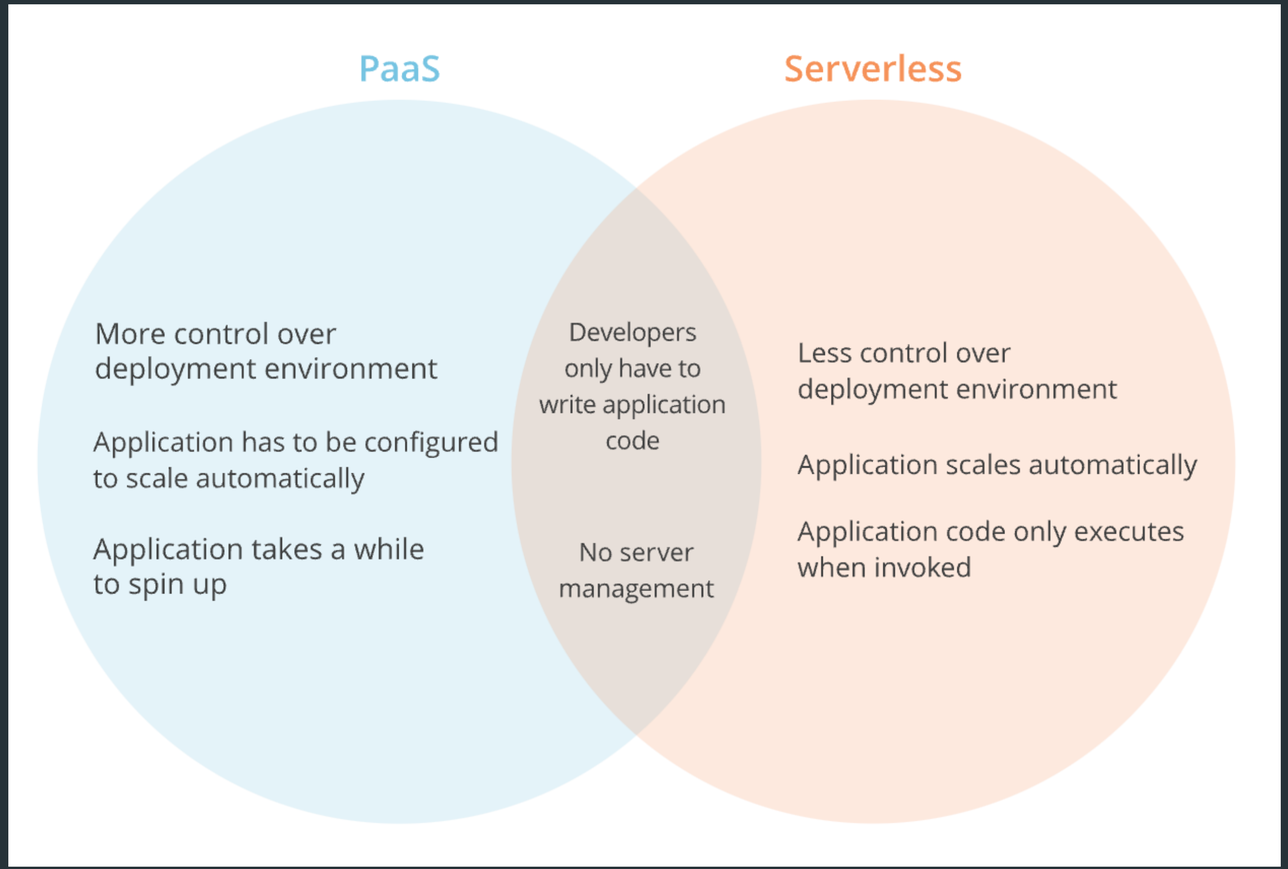
Cloud Computing Models

- Infrastructure as a Service (IaaS).
- Software as a Service (SaaS).
- Platform as a Service (PaaS).
- Backend as a Service (BaaS).
- Function as a Service (FaaS) provides a way to run code without having to worry about managing servers or infrastructure.



Types of Serverless Computing

- Function-as-a-Service (FaaS)
- Backend-as-a-Service (BaaS)
- Platform-as-a-Service (PaaS)

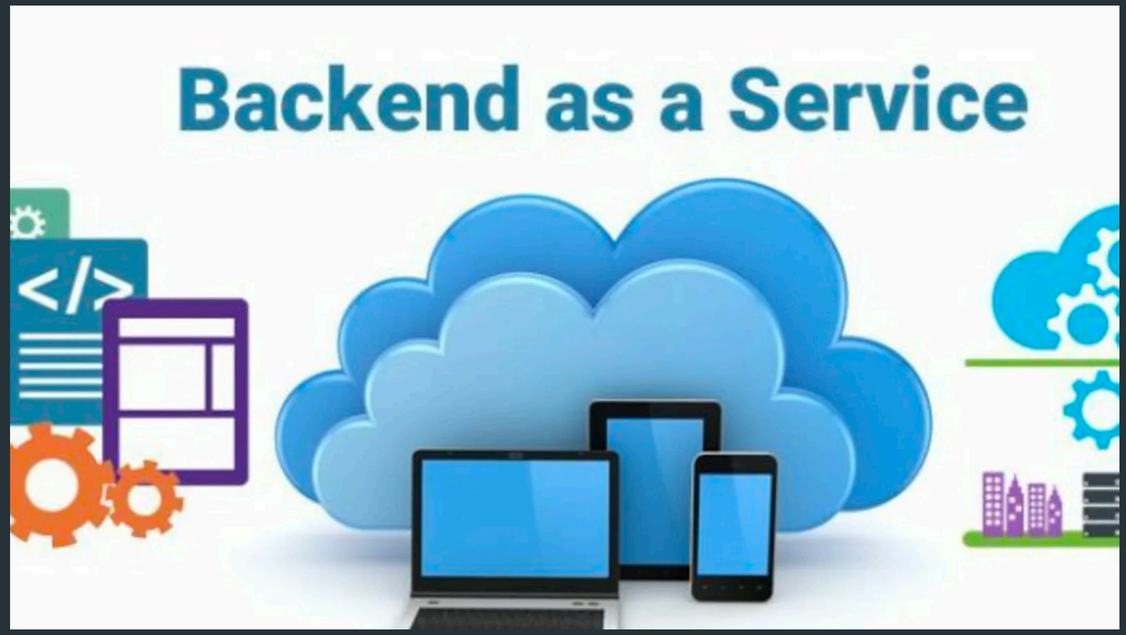


Function-as-a-Service (FaaS)

- The most popular type of serverless computing.
- Allows developers to run code in response to events, such as HTTP requests, database changes, or file uploads.
- Ideal for event-driven applications and for applications that need to be highly scalable and cost-effective.
- Some popular providers include AWS Lambda, Azure Functions, and Google Cloud Functions.

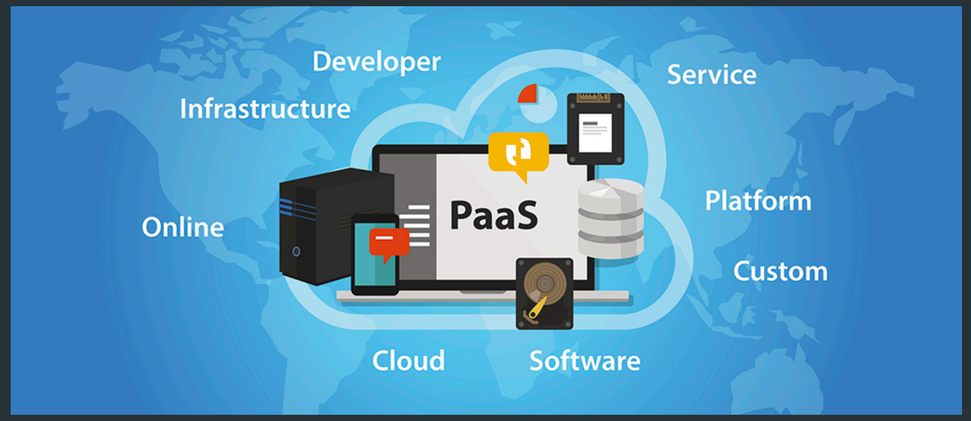
Backend-as-a-Service (BaaS)

- Provides developers with a backend infrastructure, such as databases, storage, and APIs.
- Ideal for developers who want to focus on developing their applications without having to worry about the underlying infrastructure.
- Some popular providers include AWS Amplify, Azure App Service, and Google Cloud Platform App Engine.



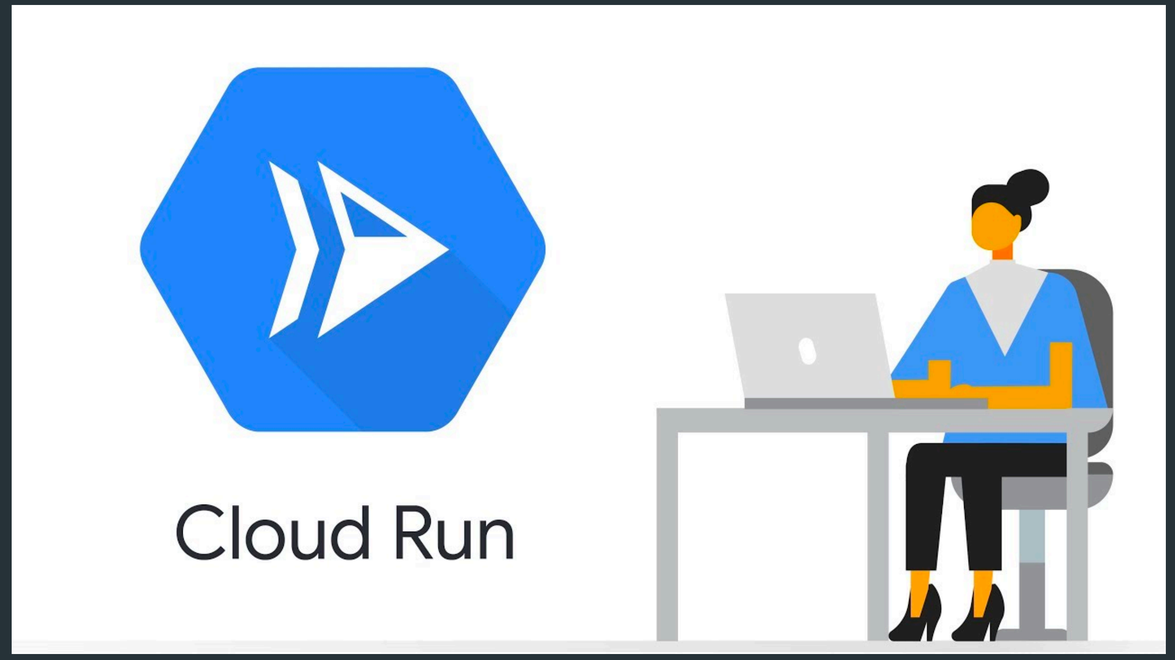
Platform-as-a-Service (PaaS)

- Provides developers with a complete development environment, including a programming language, a runtime environment, and a debugger.
- Ideal for developers who want to quickly and easily develop and deploy applications.
- Some popular providers include AWS Elastic Beanstalk, Azure App Service, and Google Cloud Platform App Engine.



Building Serverless Applications With Google Cloud Run

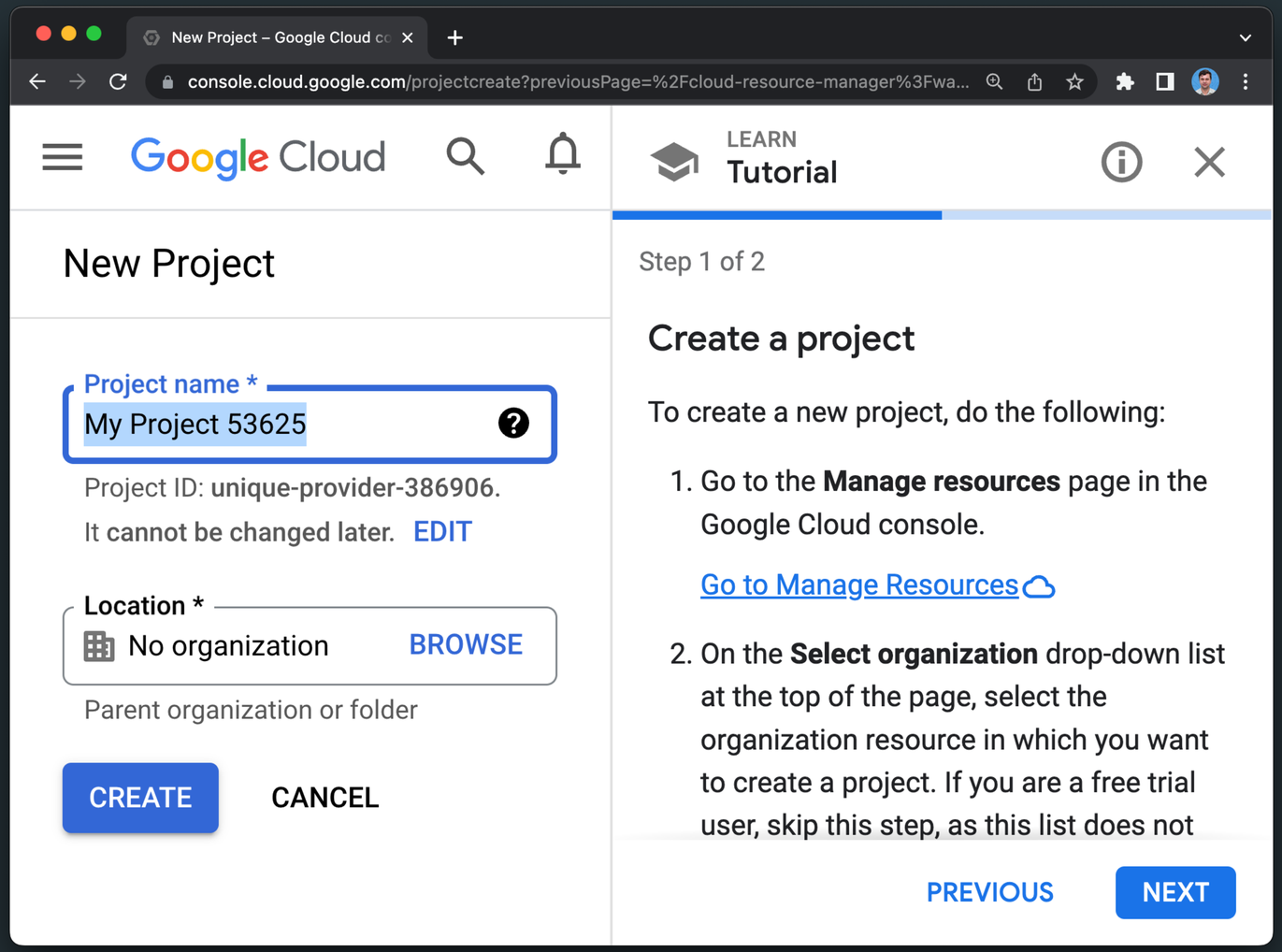
- A serverless compute platform.
- Allows you to run stateless containers.
- Automatically scales your containers based on demand.



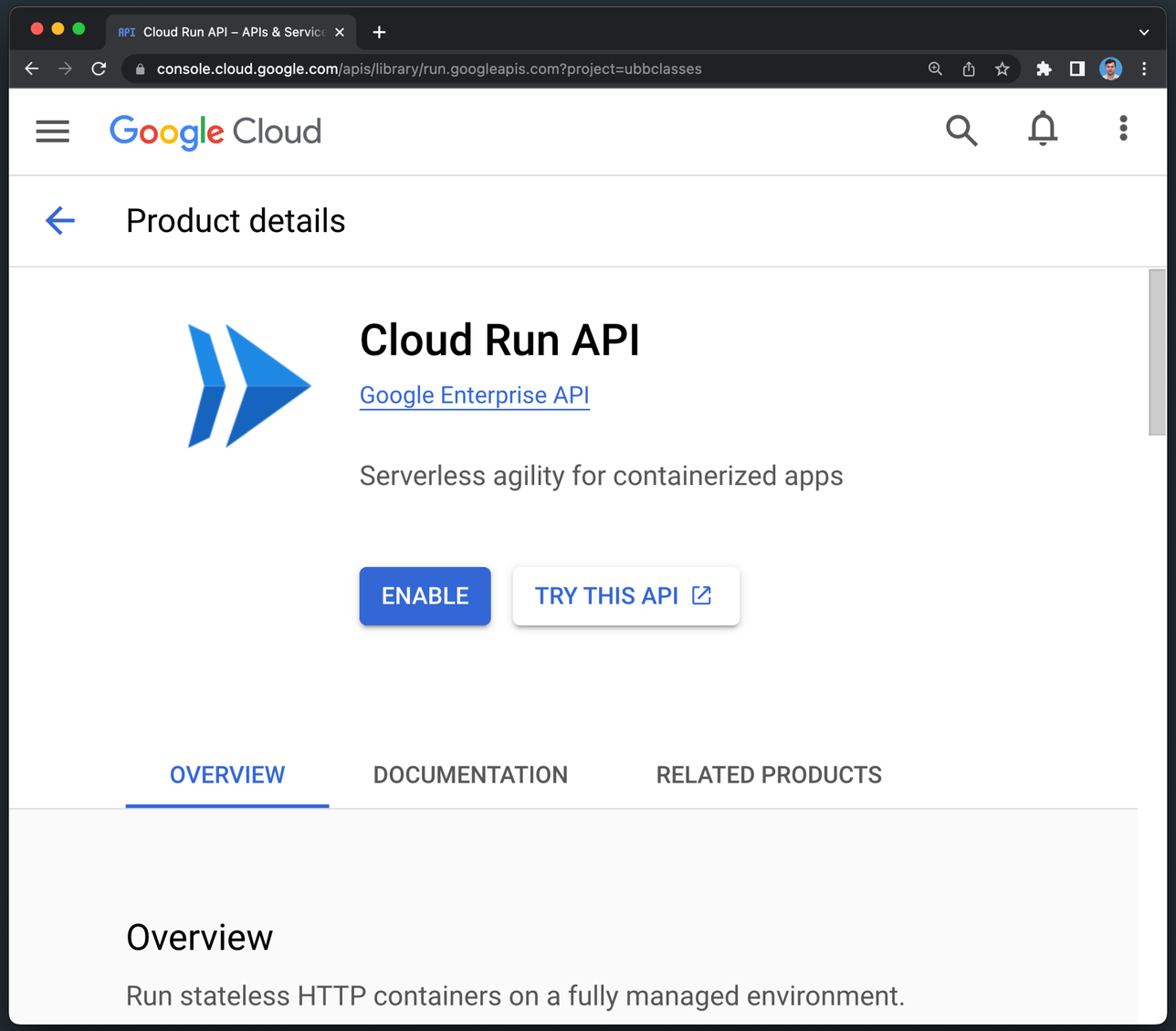
Getting Started

- To get started with Google Cloud Run, you will need to:
  - Create a Google Cloud Platform (GCP) project.
  - Enable the Cloud Run API.
  - Create a Dockerfile.
  - Build your container image.
  - Deploy your container image to Cloud Run.

Create a Google Cloud Platform (GCP) project.



Cloud Run API



Dockerfile

```
FROM python:3.7-alpine

WORKDIR /app

COPY requirements.txt ./

RUN pip install -r requirements.txt

COPY app.py ./

CMD ["python", "app.py"]
```

Container Image

```
docker build -t my-app .
```

Container Image

```
docker build -t my-app .
```

Deploy the Image

```
gcloud run deploy my-app --image=my-app
```

Container Image

```
docker build -t my-app .
```

Deploy the Image

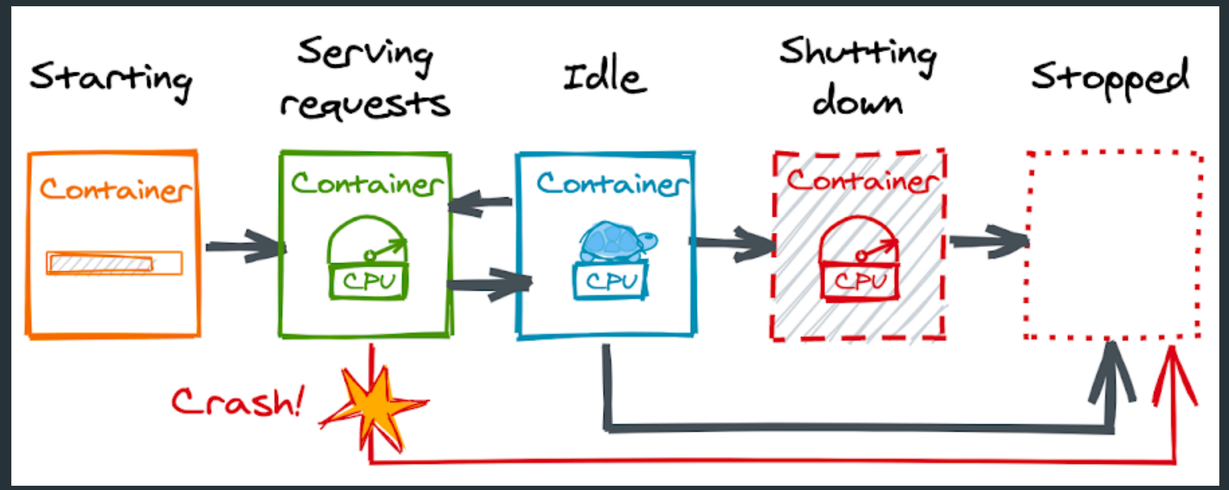
```
gcloud run deploy my-app --image=my-app
```

Access the Image

https://my-app.cloud.run

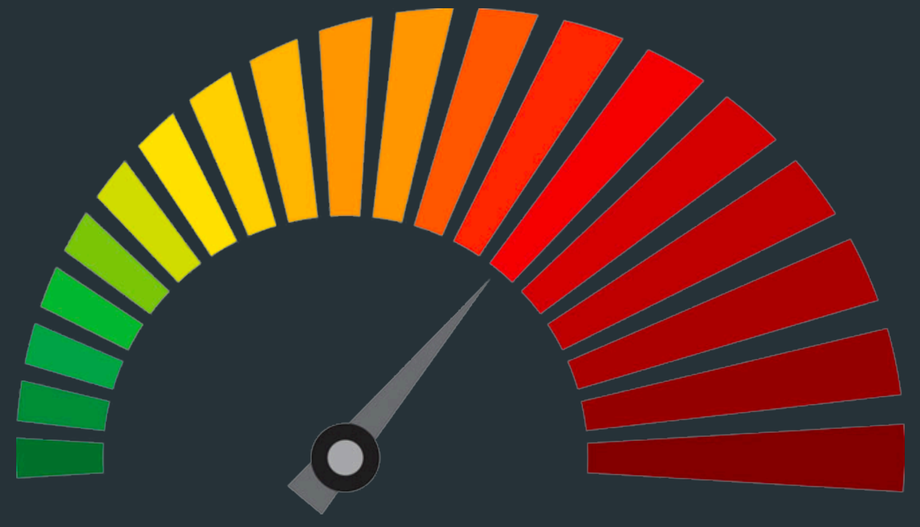
Container Lifecycle

- Creation: The container is created from a container image.
- Start: The container is started and begins listening for requests.
- Run: The container processes requests until it is terminated.
- Stop: The container is stopped and no longer listens for requests.
- Termination: The container is terminated and its resources are released.



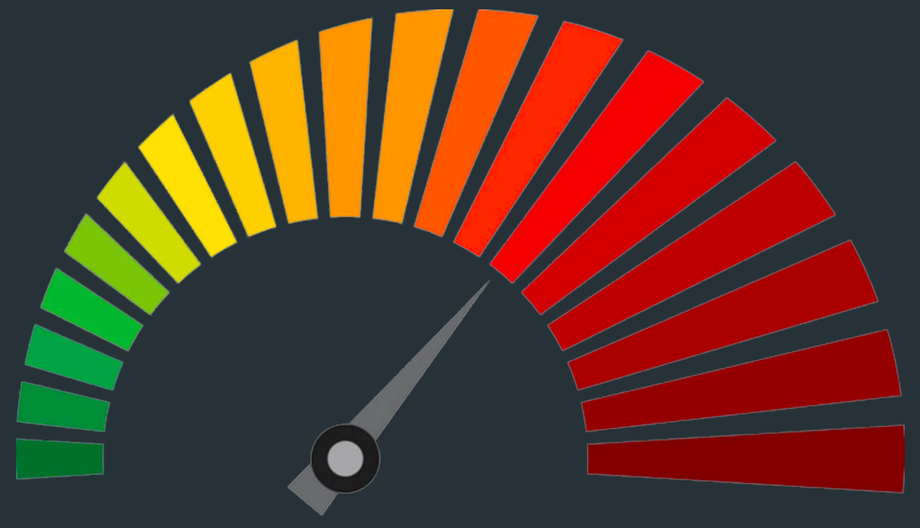
What is CPU Throttling?

- CPU throttling is a feature of Google Cloud Run.
- CPU throttling allows you to control how much CPU your container can use.
- By default, Cloud Run will throttle your container's CPU usage to 50%.



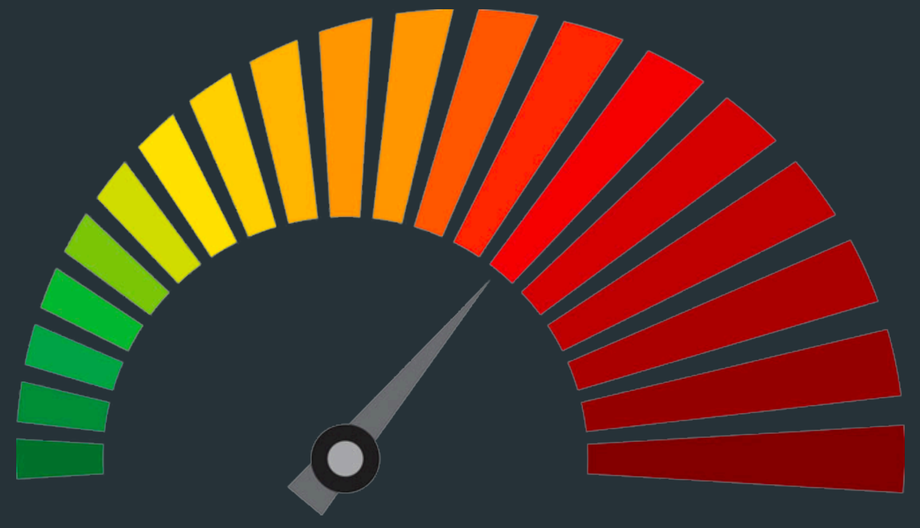
Reasons to Throttling

- Reduce cost.
- Improve performance.



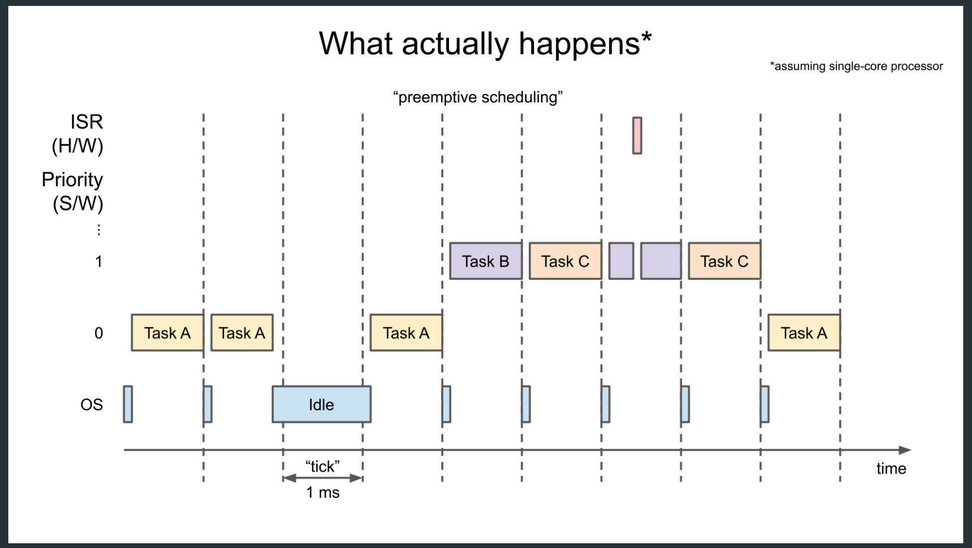
How to Use CPU Throttling

```
runtime: python37

env:
  PORT: 8080

handlers:
- url: /.*
  script: main.py

resources:
  cpu: 25
```



Task Scheduling

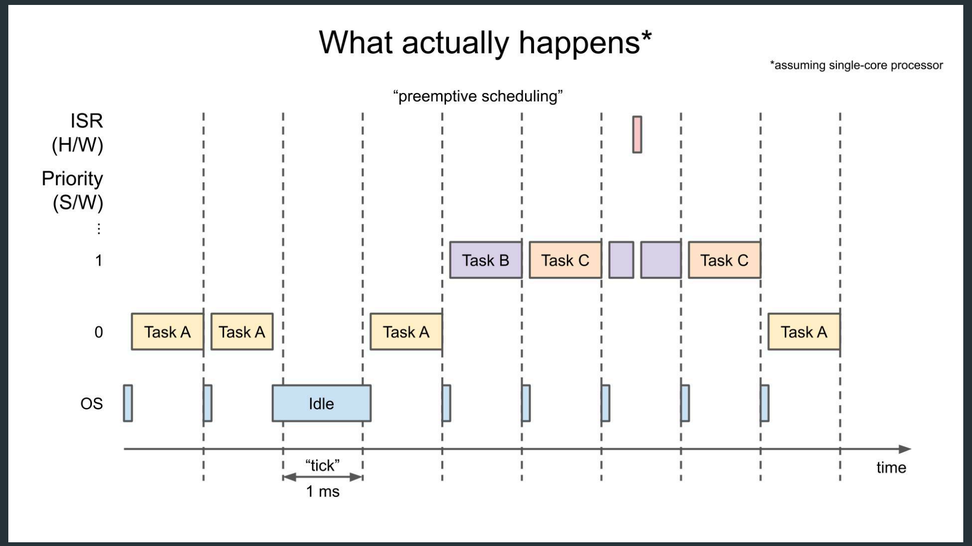
- You can schedule tasks in Cloud Run using the following methods:
  - Cron jobs: Cron jobs allow you to run tasks on a recurring schedule.
  - Event-driven tasks: Event-driven tasks allow you to run tasks in response to events.



Task Scheduling and Throttling

- Throttle the CPU usage of your containers in Cloud Run using the following methods:
  - CPU quota: CPU quota allows you to specify the maximum amount of CPU that a container can use.
  - CPU requests: CPU requests allow you to specify the minimum amount of CPU that a container needs.

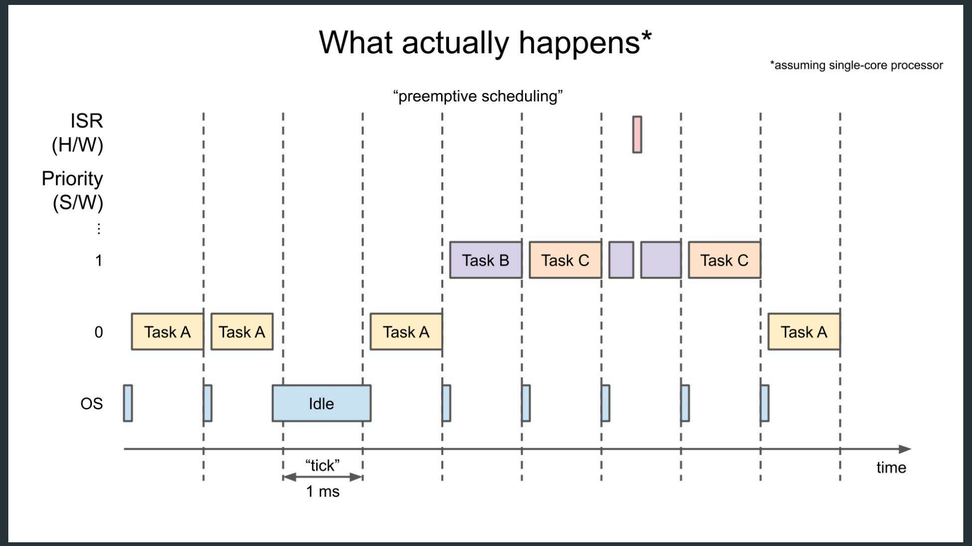
```
cron:
- description: "Run my task every day at 10am"
  schedule: "0 10 * * *"
  command: "gcloud run invoke my-task"
```



Task Scheduling and Throttling

- Throttle the CPU usage of your containers in Cloud Run using the following methods:
  - CPU quota: CPU quota allows you to specify the maximum amount of CPU that a container can use.
  - CPU requests: CPU requests allow you to specify the minimum amount of CPU that a container needs.

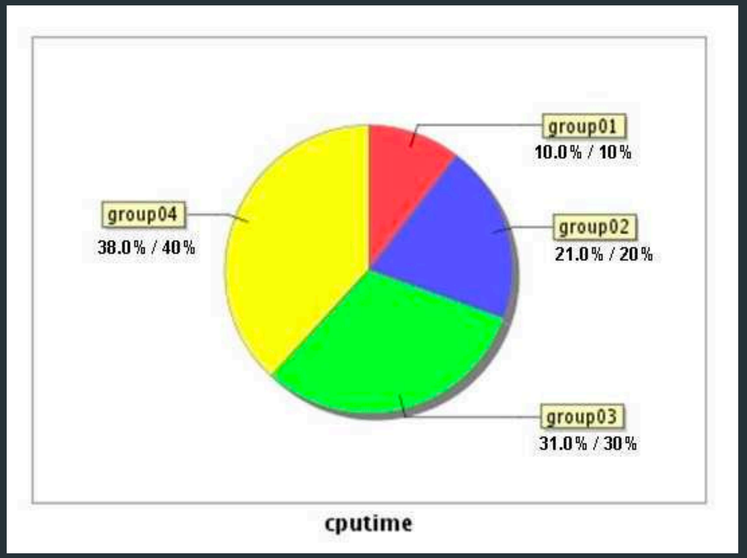
```
cron:
- description: "Run my task every day at 10am"
  schedule: "0 10 * * *"
  command: "gcloud run invoke my-task"
```



CPU Quota

gcloud run set-cpu-quota REGION SERVICE_NAME CPU_QUOTA

- Where:
  - REGION is the region where the service is running.
  - SERVICE_NAME is the name of the service.
  - CPU_QUOTA is the number of CPU cores that you want to allocate to the service.

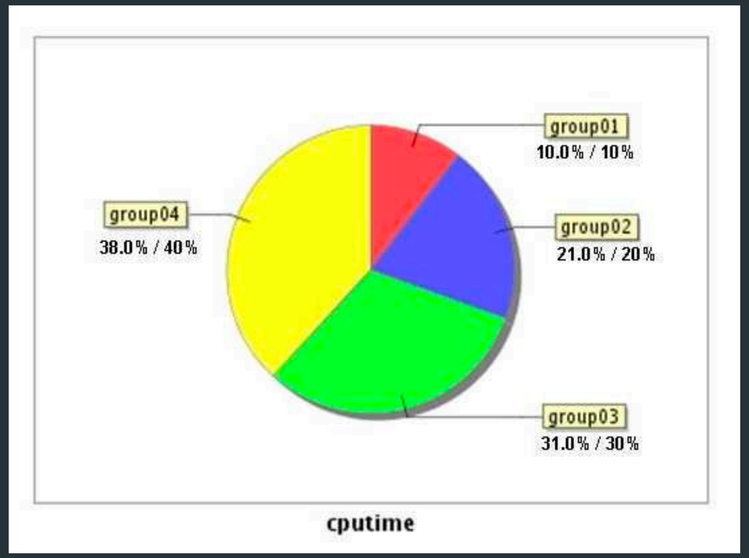


CPU Quota

```
gcloud run set-cpu-quota REGION SERVICE_NAME CPU_QUOTA
```

- Where:
  - REGION is the region where the service is running.
  - SERVICE_NAME is the name of the service.
  - CPU_QUOTA is the number of CPU cores that you want to allocate to the service.

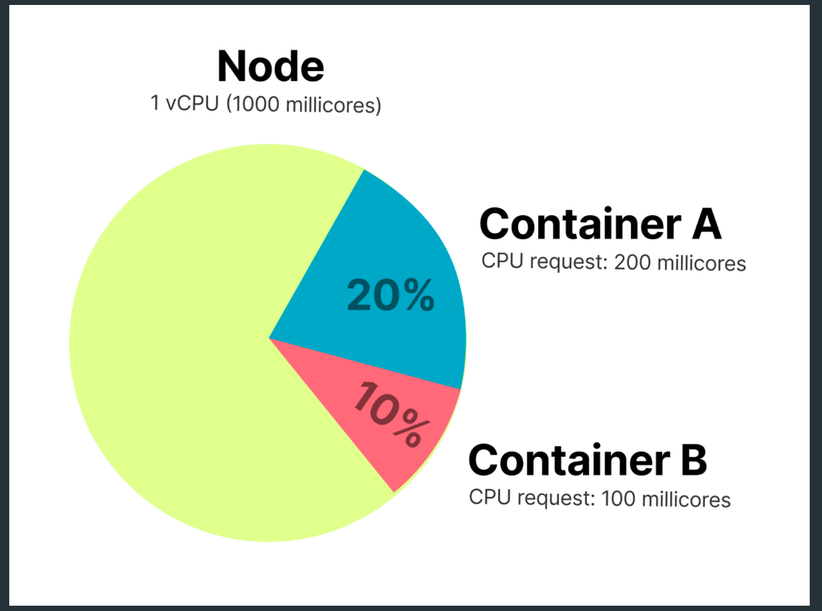
```
gcloud run set-cpu-quota us-central1 my-service 1
```



CPU Requests

```
gcloud run set-cpu-request REGION SERVICE_NAME CONTAINER_NAME CPU_REQUEST
```

- Where:
  - REGION is the region where the service is running.
  - SERVICE_NAME is the name of the service.
  - CONTAINER_NAME is the name of the container.
  - CPU_REQUEST is the number of CPU cores that you want to request for the container.

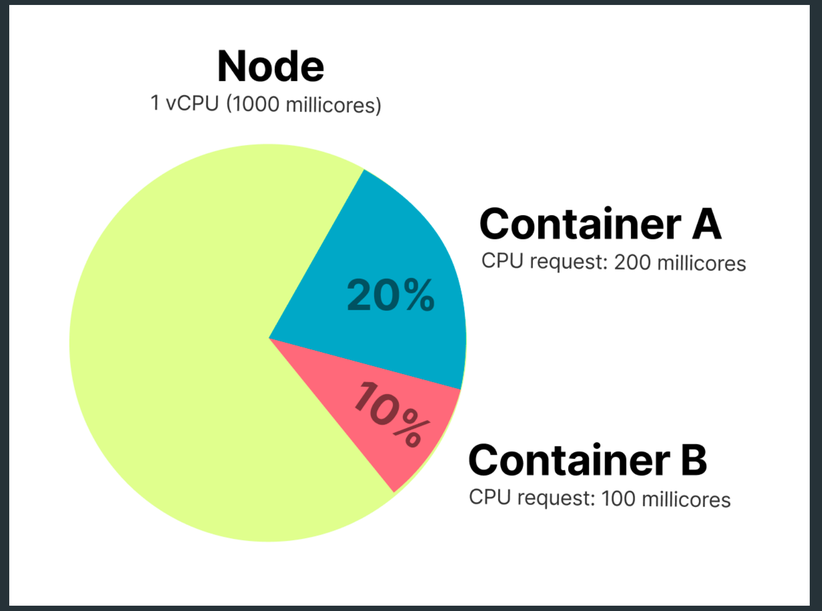


CPU Requests

gcloud run set-cpu-request REGION SERVICE_NAME CONTAINER_NAME CPU_REQUEST

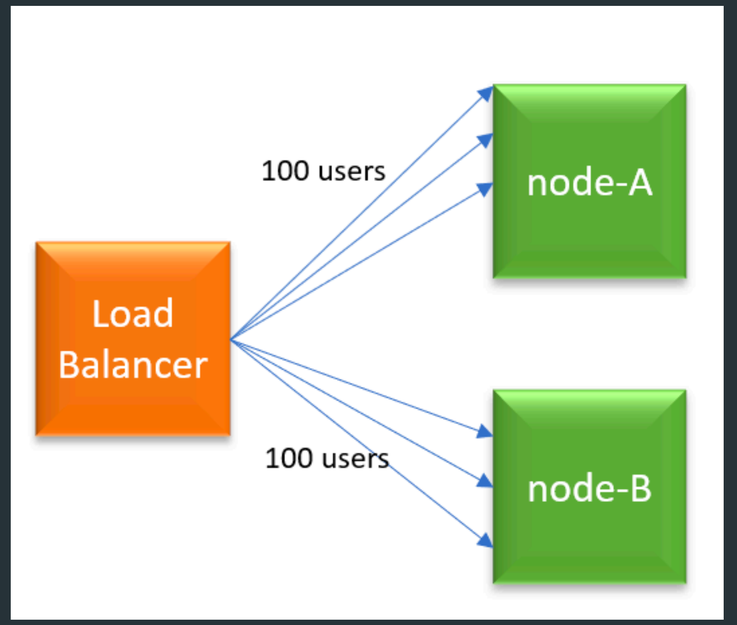
- Where:
  - REGION is the region where the service is running.
  - SERVICE_NAME is the name of the service.
  - CONTAINER_NAME is the name of the container.
  - CPU_REQUEST is the number of CPU cores that you want to request for the container.

gcloud run set-cpu-request us-central1 my-service my-container 0.5



Load Balancing

Cloud Run uses a round-robin load balancer to distribute traffic across your containers. This means that each container will receive an equal number of requests.



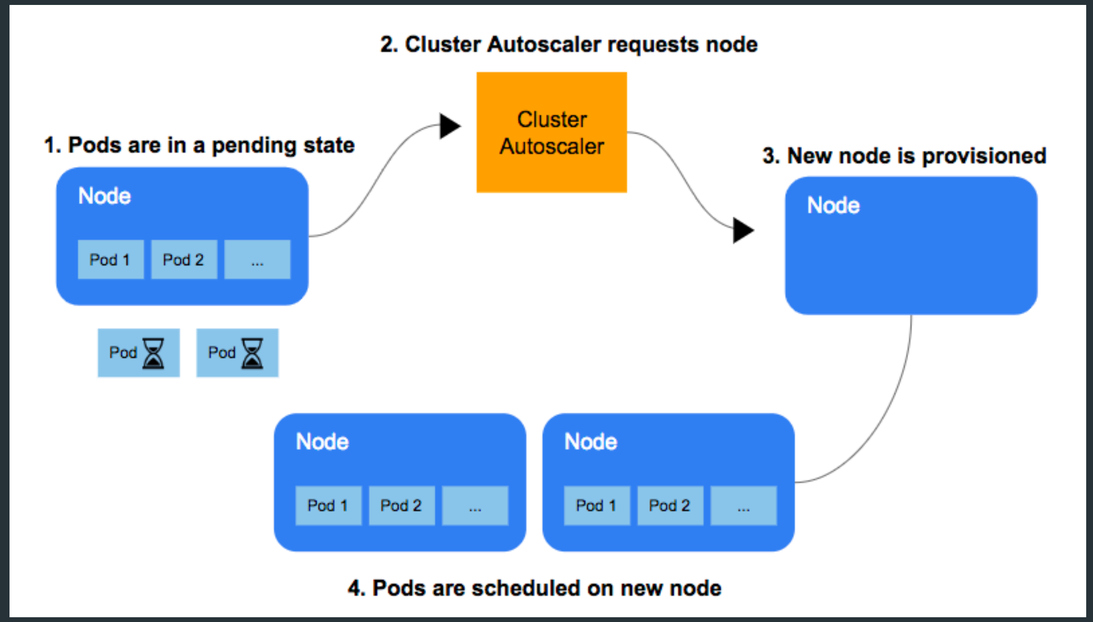
Autoscaler

Autoscaler automatically scales your containers up or down based on demand. The autoscaler uses a variety of factors to determine when to scale, including:

- the number of requests per second.

- the CPU usage.

- the memory usage.



Example

```
loadBalancer:
  enabled: true
  type: ROUND_ROBIN

autoscaler:
  minCount: 1
  maxCount: 10
```

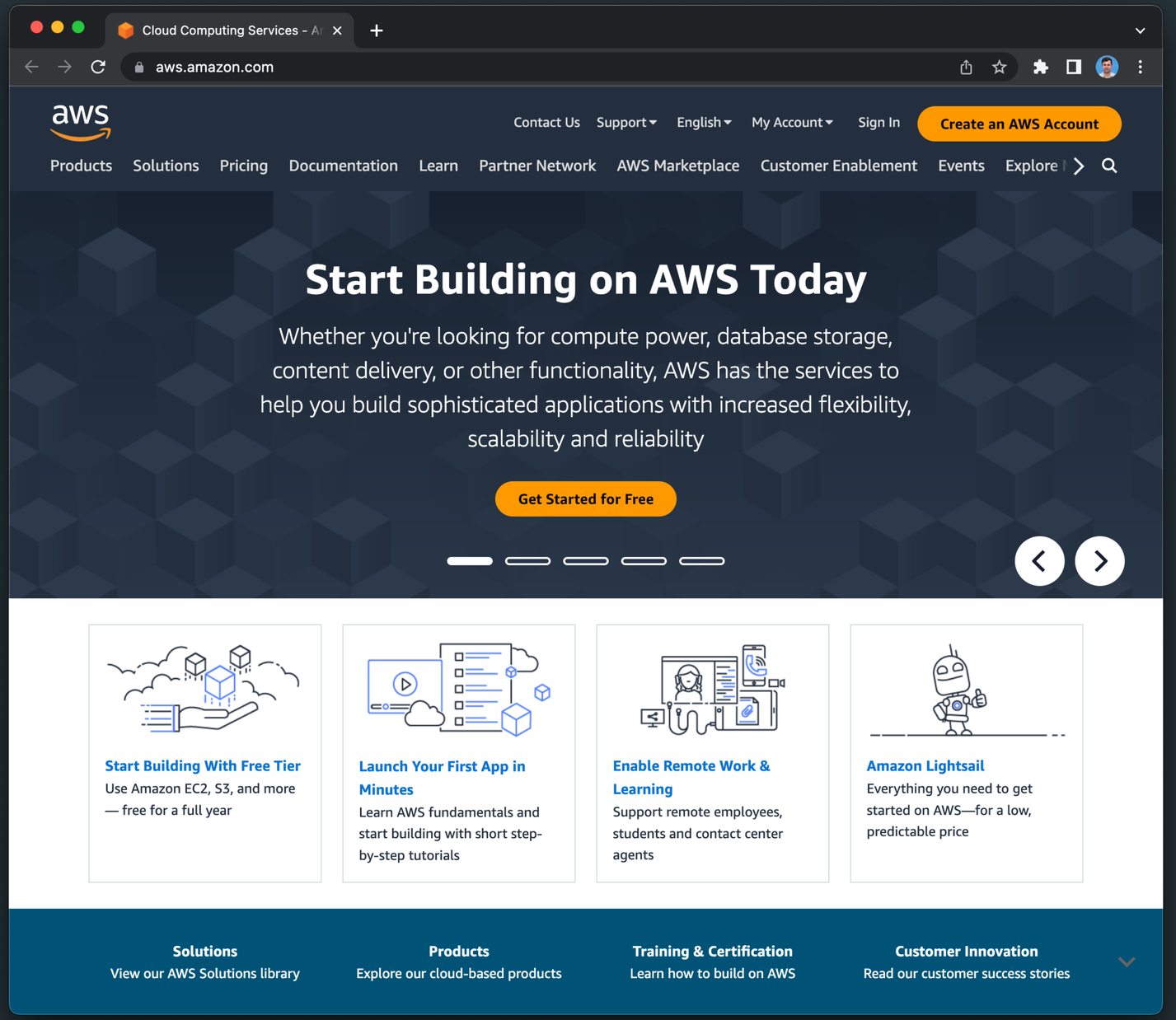
Building Serverless Applications With AWS

- A serverless compute platform.
- Allows you to run stateless containers.
- Automatically scales your containers based on demand.

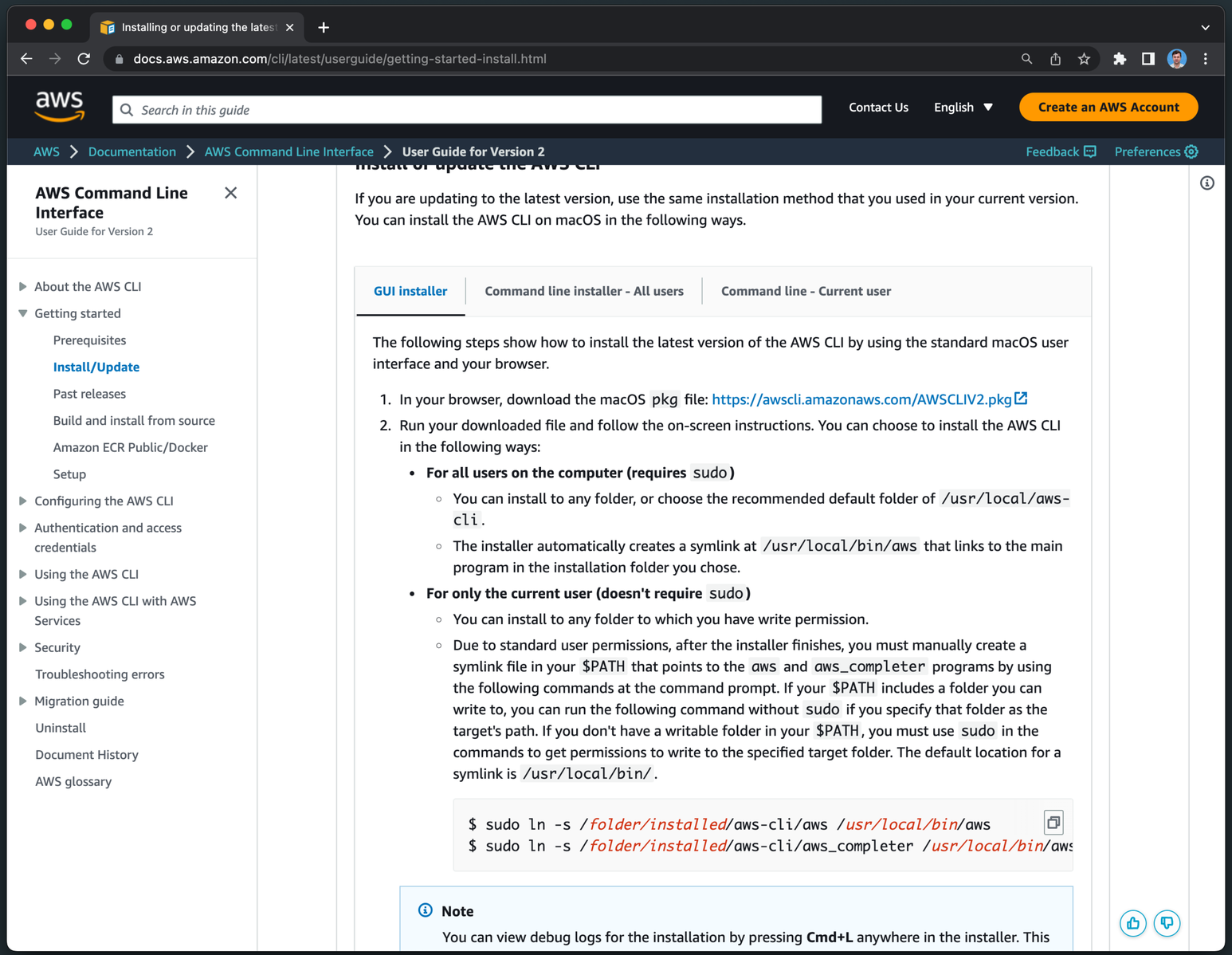
AWS Serverless Services

- **AWS Lambda:** AWS Lambda is a service that allows you to run code without having to provision or manage servers.
- **AWS API Gateway:** AWS API Gateway is a service that allows you to create, publish, monitor, and secure RESTful APIs.
- **AWS DynamoDB:** AWS DynamoDB is a fully managed NoSQL database service.
- **AWS CloudFormation:** AWS CloudFormation is a service that allows you to create and manage AWS resources as a single unit.

Create an AWS account



Install the AWS CLI



Create a Lambda function

```
def hello(event, context):
  return "Hello, " + event['name']
```

Create a deployment package

1. Create a directory for your Lambda function.
2. In the directory, create a file called `index.py` and paste your Lambda function's code into it.
3. If your Lambda function uses any dependencies, add them to the directory.
4. Zip up the directory.

```
mkdir my_function
cd my_function
echo "def my_function(event, context):
  return event['message'] + '!'" > index.py
pip install numpy
zip -r my_function.zip .
```

Create an execution role for your Lambda function

```
aws iam create-role --role-name my-lambda-role --assume-role-policy-document file://trust-policy.json
```

Create an execution role for your Lambda function

```
aws iam create-role --role-name my-lambda-role --assume-role-policy-document file://trust-policy.json
```

Attach the AWSLambdaBasicExecutionRole policy to the role

```
aws iam attach-policy
    --policy-arn arn:aws:iam::aws:policy/AWSLambdaBasicExecutionRole
    --role-name my-lambda-role
```

Create an execution role for your Lambda function

```
aws iam create-role --role-name my-lambda-role --assume-role-policy-document file://trust-policy.json
```

Attach the AWSLambdaBasicExecutionRole policy to the role

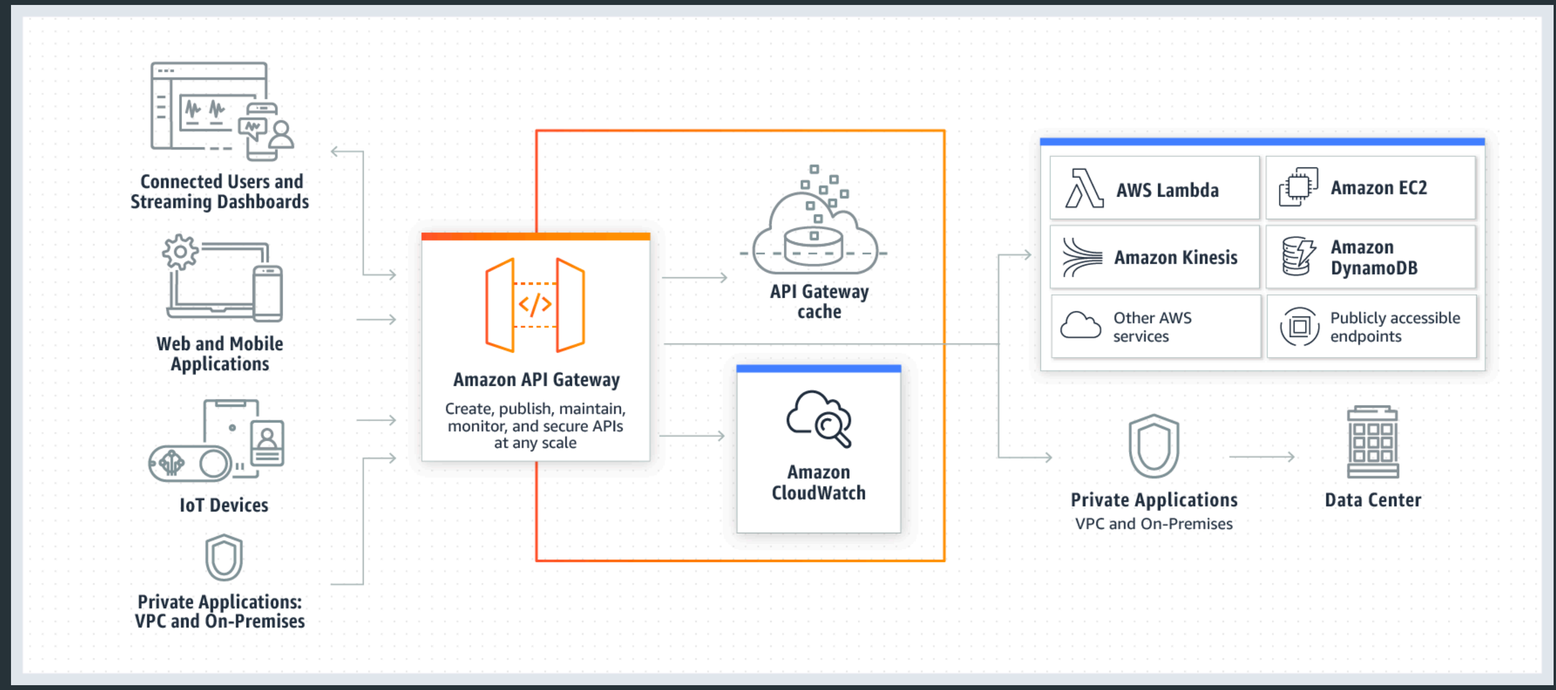
```
aws iam attach-policy
    --policy-arn arn:aws:iam::aws:policy/AWSLambdaBasicExecutionRole
    --role-name my-lambda-role
```

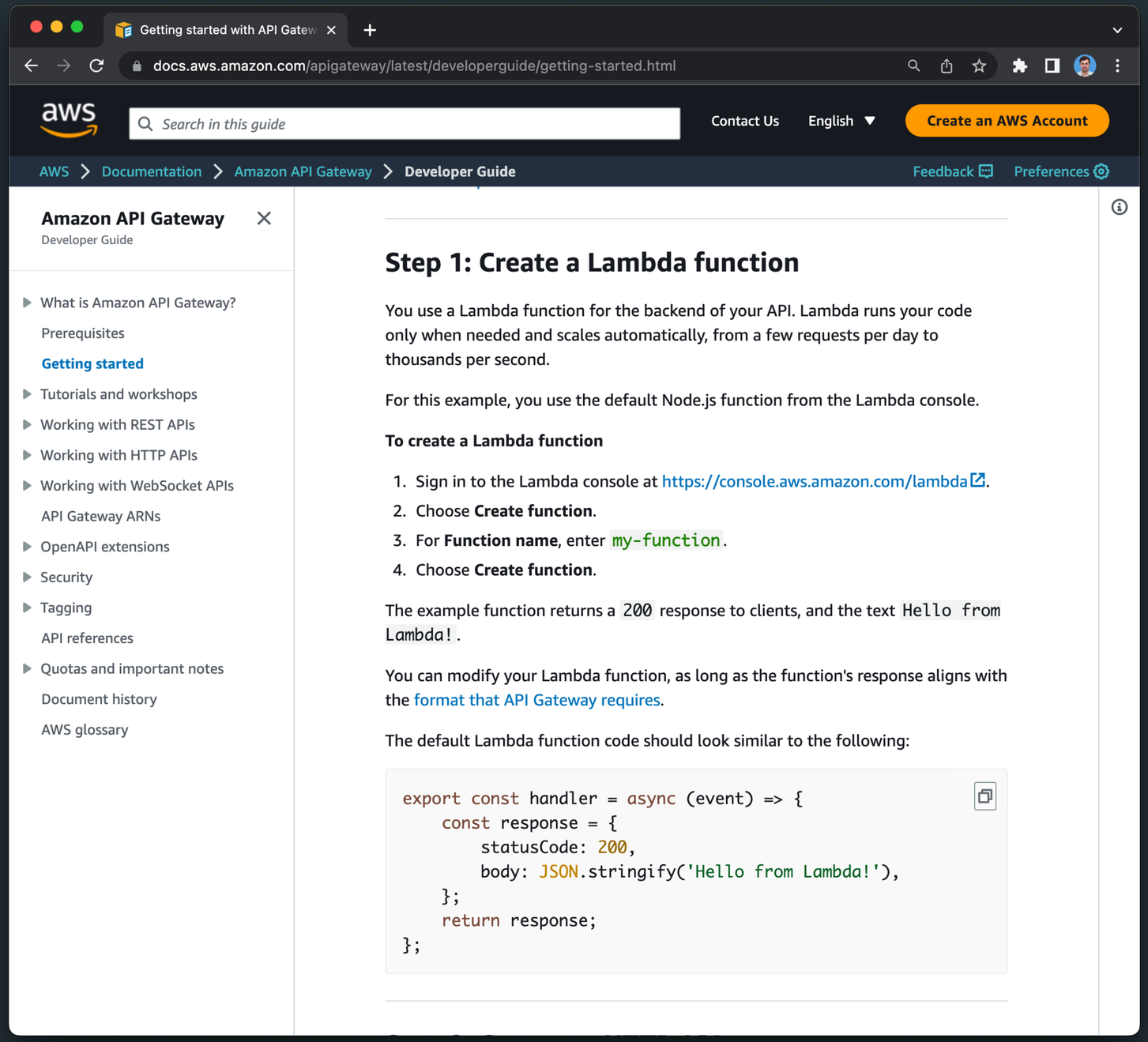
Deploy the Lambda function

```
aws lambda create-function --function-name my-lambda-function
  --runtime python3.8
  --role arn:aws:iam::<your-account-id>:role/my-lambda-role
  --handler index.my_function
  --zip-file file://my_function.zip
```

AWS API Gateway

- Visual editor
- CORS support
- Authorization and access control
- Monitoring and logging
- Deployment




Getting started with API Gatew
docs.aws.amazon.com/apigateway/latest/developerguide/getting-started.html
aws
Search in this guide
Contact Us
English
Create an AWS Account
AWS > Documentation > Amazon API Gateway > Developer Guide
Feedback
Preferences
Amazon API Gateway
Developer Guide
What is Amazon API Gateway?
Prerequisites
Getting started
Tutorials and workshops
Working with REST APIs
Working with HTTP APIs
Working with WebSocket APIs
API Gateway ARNs
OpenAPI extensions
Security
Tagging
API references
Quotas and important notes
Document history
AWS glossary
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 example, you use the default Node.js function from the Lambda console.
To create a Lambda function
1. Sign in to the Lambda console at https://console.aws.amazon.com/lambda.
2. Choose Create function.
3. For Function name, enter my-function.
4. Choose Create function.
The example function returns a 200 response to clients, and the text Hello from Lambda!.
You can modify your Lambda function, as long as the function's response aligns with the format that API Gateway requires.
The default Lambda function code should look similar to the following:
export const handler = async (event) => {
const response = {
statusCode: 200,
body: JSON.stringify('Hello from Lambda!'),
};
return response;
};

Step 2: Create an HTTP API

Next, you create an HTTP API. API Gateway also supports REST APIs and WebSocket APIs, but an HTTP API is the best choice for this exercise. REST APIs support more features than HTTP APIs, but we don't need those features for this exercise. HTTP APIs are designed with minimal features so that they can be offered at a lower price. WebSocket APIs maintain persistent connections with clients for full-duplex communication, which isn't required for this example.

The HTTP API provides an HTTP endpoint for your Lambda function. API Gateway routes requests to your Lambda function, and then returns the function's response to clients.

To create an HTTP API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Do one of the following:
   - To create your first API, for **HTTP API**, choose **Build**.
   - If you've created an API before, choose **Create API**, and then choose **Build** for **HTTP API**.
3. For **Integrations**, choose **Add integration**.
4. Choose **Lambda**.
5. For **Lambda function**, enter `my-function`.
6. For **API name**, enter `my-http-api`.
7. Choose **Next**.
8. Review the *route* that API Gateway creates for you, and then choose **Next**.
9. Review the *stage* that API Gateway creates for you, and then choose **Next**.
10. Choose **Create**.

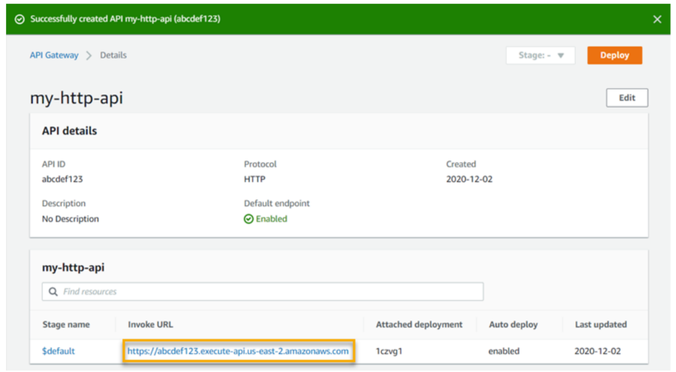
Now you've created an HTTP API with a Lambda integration that's ready to receive requests from clients.

Step 3: Test your API

Next, you test your API to make sure that it's working. For simplicity, use a web browser to invoke your API.

To test your API

1. Sign in to the API Gateway console at https://console.aws.amazon.com/apigateway.
2. Choose your API.
3. Note your API's invoke URL.



4. Copy your API's invoke URL, and enter it in a web browser. Append the name of your Lambda function to your invoke URL to call your Lambda function. By default, the API Gateway console creates a route with the same name as your Lambda function, `my-function`.

   The full URL should look like `https://abcdef123.execute-api.us-east-2.amazonaws.com/my-function`.

   Your browser sends a `GET` request to the API.
5. Verify your API's response. You should see the text `"Hello from Lambda!"` in your browser.

Lecture outcomes

- History
- Cloud Models
- Serverless Types
- Google Cloud Run
- AWS