Serverless AWS Lambda Explained

what is serverless (lamda in aws) ?

explain in details so that i am able to understand and donot have doubt on this.

Great question, Piyush. **Serverless (AWS Lambda)** is one of those concepts that *sounds confusing* at first, but once it clicks, it actually becomes very logical.
I'll explain it **from zero → internals → real flow → pros/cons → when to use**, so you won't have doubts.

🧠 First, clear the biggest myth

❌ "Serverless means there are no servers"

Wrong.

✅ Reality:

- **Servers DO exist**
- **You just don't manage them**

AWS manages:

- Servers
- OS
- Scaling
- Patching
- Availability

You manage:

- **Only your code**

This is why it's called **server-less (from developer's perspective)**.

🧩 What is Serverless?

📌 Definition (simple)

Serverless is a cloud execution model where you run code without provisioning or managing servers, and you pay only when your code runs.

In AWS, the most popular serverless service is:

🔥 AWS Lambda

> Lambda lets you run small pieces of code (functions) in response to events.

🧱 Traditional Server-based App vs Serverless

🖥️ Traditional (EC2 / VM based)

You must:

- Create server
- Choose CPU, RAM
- Install Java, Docker, etc.
- Deploy app
- Scale manually
- Pay **24×7**, even if no traffic

⚡ Serverless (Lambda)

You:

- Upload a function
- AWS runs it **only when needed**
- Auto-scales
- Pay **per execution**

🧪 Simple analogy (VERY important)

🍽️ Restaurant analogy

Traditional server

- You rent the whole restaurant

- Pay full rent
- Even if no customers come

Serverless

- You pay **per plate ordered**
- No orders → no cost
- Kitchen auto-scales when rush comes

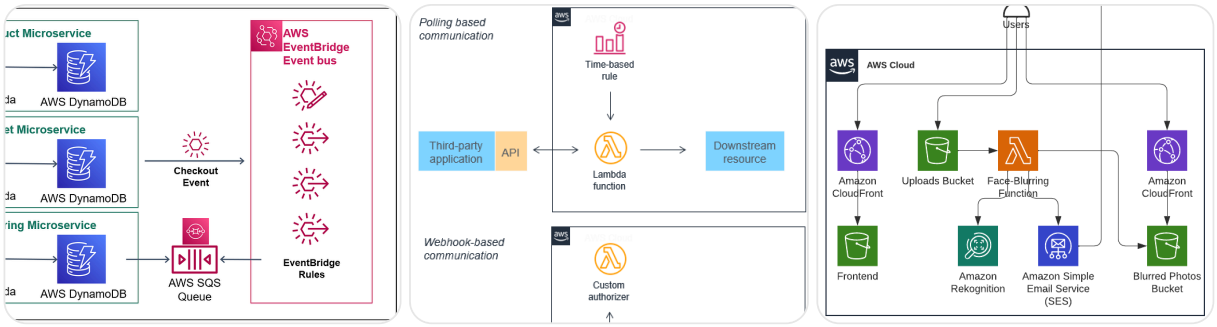
🧩 What exactly is AWS Lambda?

📌 AWS Lambda is:

- A **function execution service**
- Event-driven
- Stateless
- Auto-scaling
- Short-running (seconds/minutes)

🧠 Lambda works like this

```text
Event happens
    ↓
AWS starts a container
    ↓
Your function code runs
    ↓
Returns response
    ↓
Container may be reused
```



🧩 What is a Lambda Function?

A **Lambda Function** is:

- A single-purpose piece of code
- Written in:
  - Java ✅
  - Python
  - Node.js
  - Go
- Has:
  - **Handler method** (entry point)

Java Example (VERY simple)

```java
public class HelloLambda {
    public String handleRequest(String input) {
        return "Hello " + input;
    }
}
```

AWS calls `handleRequest()` when an event occurs.

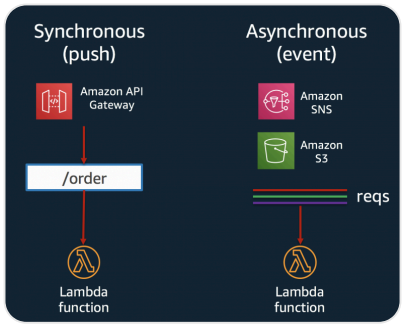
🔔 What triggers a Lambda? (Events)

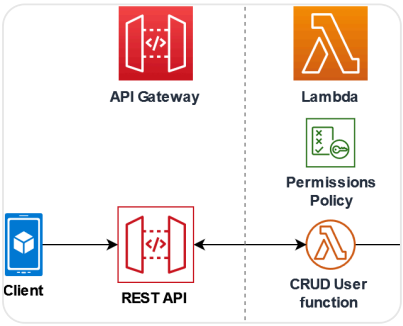
Lambda is **event-driven**.

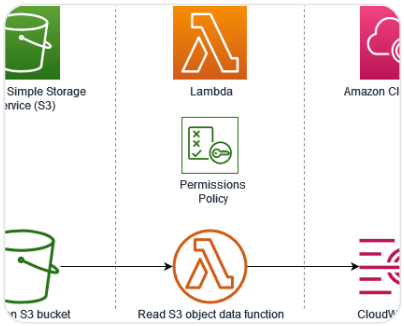
Common triggers:

- HTTP request → API Gateway

- File upload → S3
- Message → SQS
- Stream → Kafka / DynamoDB
- Cron → EventBridge







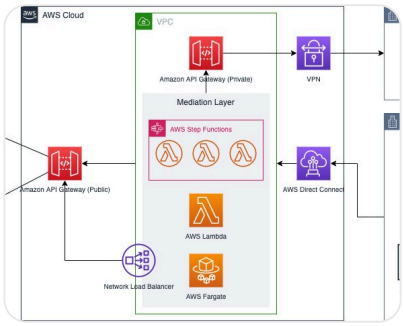
🌐 Lambda + API Gateway (MOST IMPORTANT USE CASE)

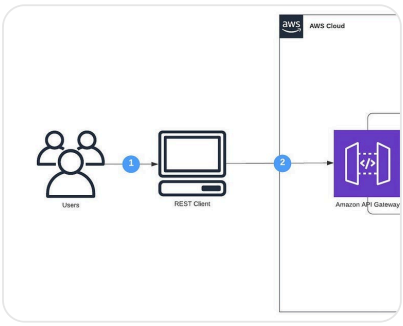
This is how **serverless backend APIs** are built.

Flow:

```
Client (Browser / Mobile)
      ↓
API Gateway (HTTP endpoint)
      ↓
AWS Lambda (business logic)
      ↓
Response
```

👉 No server
👉 No Spring Boot running 24×7
👉 No Tomcat





⏱️ Lambda execution lifecycle (VERY IMPORTANT)

1️⃣ Cold Start

- AWS creates container
- Loads runtime (Java is slower)
- Initializes code

⏱️ Happens when:

- First request
- After inactivity

2️⃣ Warm Start

- Container already exists
- Much faster

⚠️ Java note (important for you)

Java Lambdas:

- Cold start slower than Node/Python
- But acceptable with:
  - Smaller JAR
  - Proper memory config

📦 Stateless nature (CRITICAL CONCEPT)

Lambda is **stateless**.

❌ You cannot rely on:

- Instance memory
- Local variables
- Local files

✅ You must store state in:

- DynamoDB
- RDS

- Redis
- S3

📏 Limits you MUST know

Limit	Value
Max execution time	**15 minutes**
Memory	128 MB – 10 GB
Disk (/tmp)	512 MB
Concurrent executions	1000 (default)

💰 How pricing works (simple)

You pay for:

1. **Number of executions**
2. **Execution time**
3. **Memory allocated**

Example:

- 1M requests
- Each runs 200ms
- You pay **only for that time**

No traffic = ₹0

✅ Advantages of Serverless

- ✔ No server management
- ✔ Auto-scaling
- ✔ High availability
- ✔ Cost efficient for variable load
- ✔ Faster development

❌ Disadvantages (be honest)

❌ Cold starts (Java)
❌ Harder debugging
❌ Vendor lock-in
❌ Not ideal for long-running jobs
❌ Stateless complexity

🧠 When SHOULD you use Lambda?

Perfect for:

- REST APIs
- Webhooks (payments, Razorpay 😉)
- Background jobs
- File processing
- Event-driven systems
- Cron jobs

🚫 When NOT to use Lambda?

- Long-running services (>15 min)
- Heavy ML training
- Stateful systems
- Extremely low-latency systems

🧩 Serverless ≠ Lambda only

Serverless ecosystem includes:

- Lambda (compute)
- API Gateway (HTTP)
- DynamoDB (DB)
- SQS / SNS (messaging)
- EventBridge (events)

Together → **fully serverless architecture**

🧠 Final mental model (lock this in)

> **Serverless = "I write business logic. Cloud handles everything else."**

If you want next:

- **Lambda vs Spring Boot**
- **Lambda + API Gateway complete flow**
- **Cold start deep dive**
- **How webhooks use Lambda**
- **When to choose Lambda vs EC2 vs ECS**

Just tell me 👍

what is container orchestration in system design?

explain in details so that i am able to understand and donot have doubt on this.

Perfect topic to ask next, Piyush 👍
Container orchestration is a *core* system-design concept, and I'll break it down **slowly, from fundamentals → problems → solutions → real-world flow**, so you can build a rock-solid mental model with **zero confusion**.

🧱 Step 1: First, what is a container? (30-second recap)

A **container** is a lightweight package that contains:

- Your application code
- Runtime (JVM / Node / Python)
- Libraries & dependencies
- Config

👉 Runs **the same everywhere** (laptop, server, cloud).

Most popular container tool:

- Docker

🧨 Step 2: The REAL problem (why orchestration exists)

Imagine this setup 👇

- You have **50 Docker containers**
- App must be:
  - Highly available
  - Scalable
  - Fault tolerant
- Traffic keeps changing
- Servers crash
- Containers crash

❌ Without orchestration, YOU must:

- Start containers manually
- Restart crashed containers
- Decide where containers run
- Scale up/down
- Load balance traffic
- Handle deployments
- Monitor health

This becomes **impossible at scale**.

🧠 This leads to...

📌 What is Container Orchestration?

✅ Simple definition

Container orchestration is the automated management of containers at scale.

It handles:

- Deployment
- Scaling
- Networking

- Load balancing
- Self-healing
- Configuration
- Rollouts & rollbacks

👉 **You define WHAT you want. The system ensures it stays that way.**

🤖 Think of orchestration like a "container manager"

🎼 Orchestra analogy

- Containers = musicians
- Servers = stage
- Orchestrator = conductor

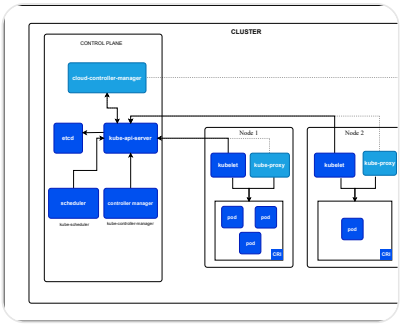
The conductor ensures:

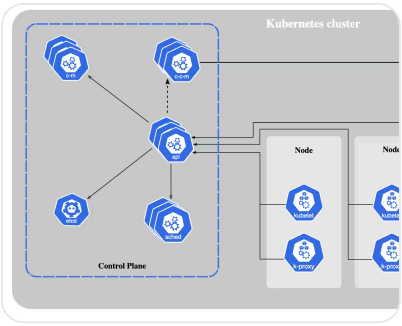
- Right musicians play
- If one stops → replaced
- Sound reaches audience properly

🧩 Step 3: Most popular orchestrator

🔥 Kubernetes (K8s)

- Industry standard
- Created by Google
- Used by AWS, Azure, GCP





🧱 Step 4: Kubernetes mental model (VERY IMPORTANT)

Kubernetes manages a cluster

```java
Cluster
 ├── Master / Control Plane
 └── Worker Nodes (servers)
```

Each **worker node** runs containers.

🧩 Core Kubernetes components (one by one)

1️⃣ Pod (smallest unit)

A **Pod**:

- Wraps **one or more containers**
- Containers share:
  - Network
  - Storage

👉 Kubernetes does **NOT** run containers directly.
👉 It runs **Pods**.

2️⃣ Node (machine)

A **Node**:

- Physical or virtual server
- Runs:
  - Pods
  - Container runtime (Docker/containerd)

3️⃣ Deployment (desired state)

A **Deployment** defines:

- How many replicas?
- Which image?
- Update strategy?

Example:

```
text

"I want 3 copies of my app always running"
```

Kubernetes ensures this state **continuously**.

4️⃣ Service (networking)

Pods:

- Are dynamic
- Can die & restart
- IPs change

A **Service**:

- Gives stable IP / DNS
- Load balances traffic across Pods

5️⃣ Load Balancer (external traffic)

- Routes internet traffic → cluster
- Connects users → services

🔄 Step 5: Self-healing (KEY FEATURE)

If:

- Pod crashes ❌
- Node goes down ❌

Kubernetes:

- Detects failure
- Creates new Pods
- Reassigns traffic

👉 No human needed.

📈 Step 6: Auto-scaling

Horizontal scaling

- Increase number of Pods
- Based on CPU / memory

Vertical scaling

- Increase Pod resources

Traffic spike?

👉 Kubernetes scales automatically.

🔁 Step 7: Rolling deployments (ZERO downtime)

When you deploy new code:

1. Start new Pods
2. Gradually shift traffic
3. Kill old Pods

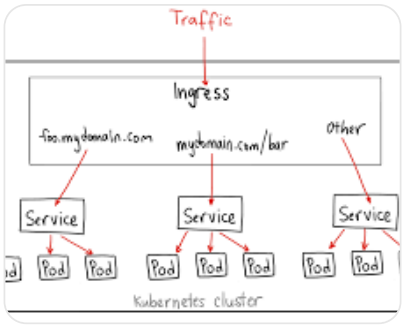
If something breaks:

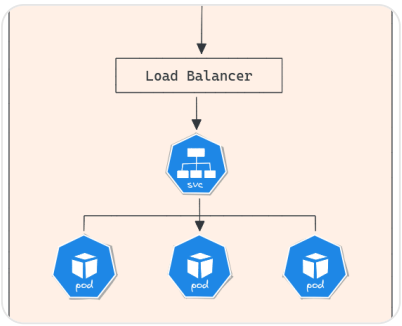
👉 **Rollback instantly**

🧠 Step 8: Real production flow (API example)

text

```
User Request
    ↓
Load Balancer
    ↓
Kubernetes Service
    ↓
Pod 1 / Pod 2 / Pod 3
    ↓
Application
```





🆚 Container orchestration vs Serverless

Feature	Orchestration	Serverless
Server control	Yes	No
Execution	Long-running	Short-lived
State	Can be stateful	Stateless
Scaling	Config-based	Automatic
Example	Kubernetes	AWS Lambda

👉 Kubernetes = **you manage infra behavior**
👉 Serverless = **infra disappears**

🧠 When do we NEED container orchestration?

Use it when:

✔ Microservices architecture
✔ Long-running services
✔ High traffic apps
✔ Custom networking needs
✔ Stateful workloads

Examples:

- Netflix
- Uber
- Payment systems

- Backend APIs (Spring Boot)

🚫 When NOT needed?

❌ Small app
❌ Simple backend
❌ Event-driven logic only
❌ Short-lived tasks

👉 Use serverless instead.

🧠 Final lock-in summary

> **Container orchestration ensures that your containers are always running, reachable, scalable, and healthy — automatically.**

Or even simpler:

> **"I say what I want. Kubernetes makes it happen."**

If you want next (highly recommended):

- **Kubernetes vs Docker vs ECS**
- **How Spring Boot runs inside Kubernetes**
- **How scaling works internally**
- **Kubernetes YAML explained simply**
- **Real system design interview explanation**

Just tell me 🚀