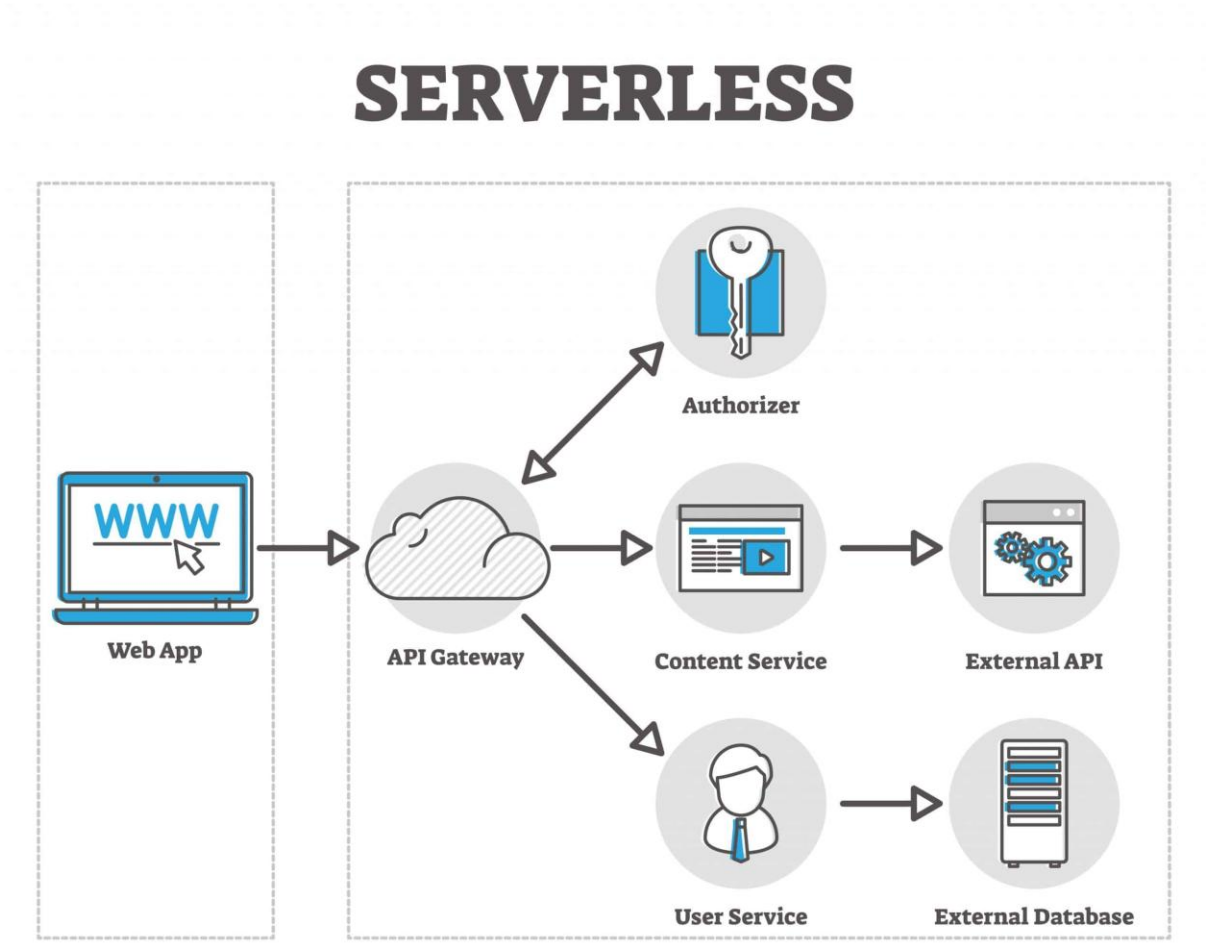
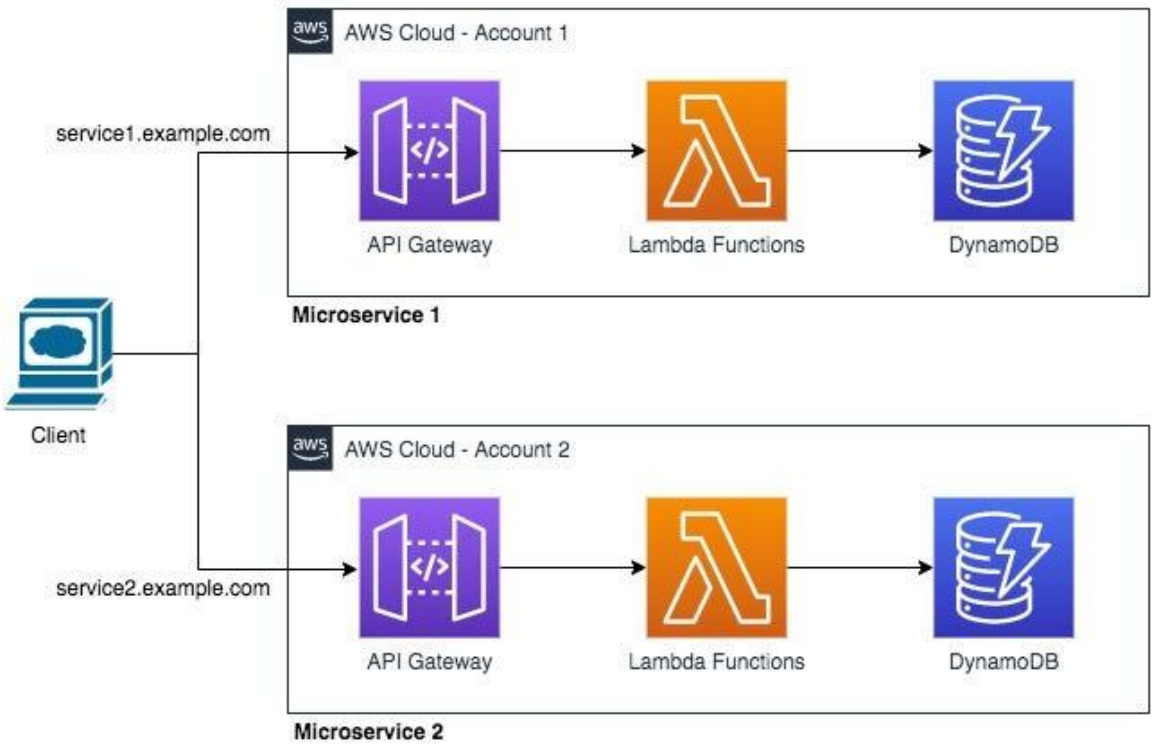
Module 7: Serverless Architectures — Detailed Notes

---

1. What is Serverless? When to Use It?





1.1 What is Serverless?

Serverless is a cloud-native execution model where:

- You **don't manage servers**
- Infrastructure is fully abstracted
- Applications run in small units called **functions**
- Auto-scaling and availability are provided by the cloud provider
- Billing is based on **actual execution time**

It does **not** mean "no servers"—it means **you don't manage them**.

1.2 Serverless Characteristics

Feature	Description
No server management	Cloud provider handles compute
Auto-scaling	Functions scale instantly based on traffic
Event-driven	Triggered by HTTP, timers, queues, storage events

Feature	Description
Pay-per-use	Charged only for execution time
Stateless compute	Functions cannot persist data locally
Short-lived	Functions usually run for short durations

1.3 When to Use Serverless?

Best use cases:

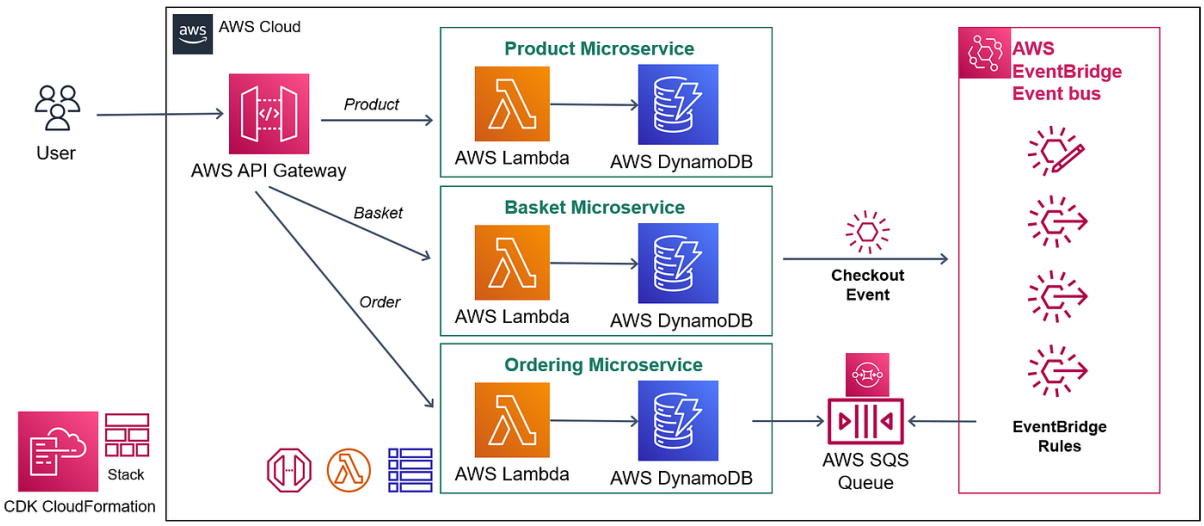
- APIs with unpredictable loads
- Event-driven workloads
- Cron jobs / scheduled tasks
- Stream processing (IoT, logs)
- Chatbots, notification services
- Data transformation pipelines
- Lightweight backends (Auth, ML inference)

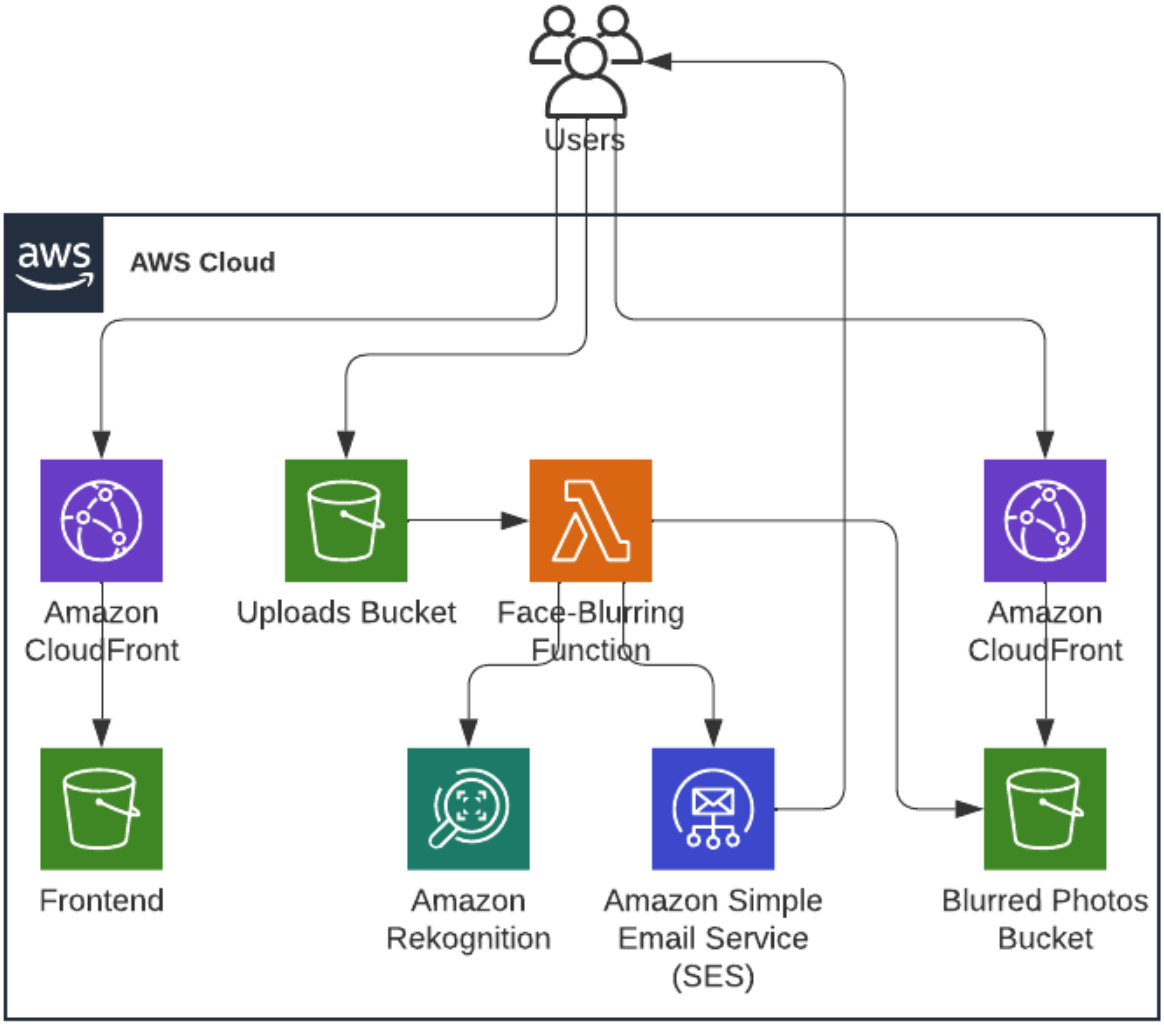
Avoid for:

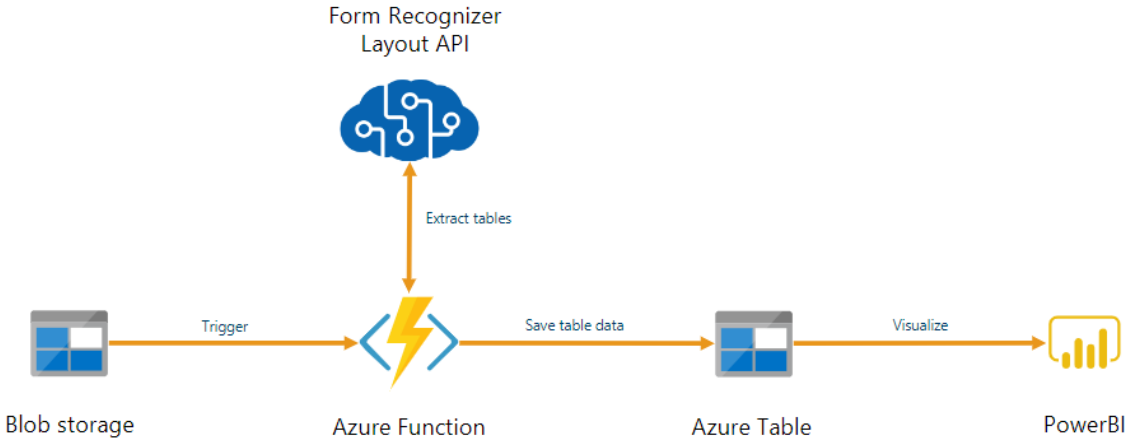
- Long-running jobs
- Applications needing GPUs long-term
- Persistent connections (e.g., WebSockets) without managed services

---

2. AWS Lambda, Azure Functions, Google Cloud Functions







6

2.1 AWS Lambda

Key Features:

- Triggers: S3, API Gateway, DynamoDB Streams, EventBridge
- Supports many runtimes (Python, Node, Java, Go, .NET, Ruby)
- Integrated with IAM, VPC, CloudWatch
- Layers for dependency packaging

Example Lambda (Python):

```
def lambda_handler(event, context):
    print("Hello from Lambda!")
    return {"status": "success"}
```

Basic Deployment:

```
aws lambda create-function \
  --function-name myfunction \
  --runtime python3.11 \
  --role arn:aws:iam::1234556789:role/lambda-role \
  --handler index.lambda_handler \
  --zip-file fileb://function.zip
```

2.2 Azure Functions

Triggers:

- HTTP
- EventHub
- CosmosDB
- ServiceBus
- Timer
- Storage (Blob/Queue)

Programming Models:

- Consumption plan (serverless)
- Premium plan
- Dedicated plan

Example Function (JavaScript):

```
module.exports = async function (context, req) {
    context.res = { body: "Hello Azure Functions!" };
```

}

2.3 Google Cloud Functions

Triggers:

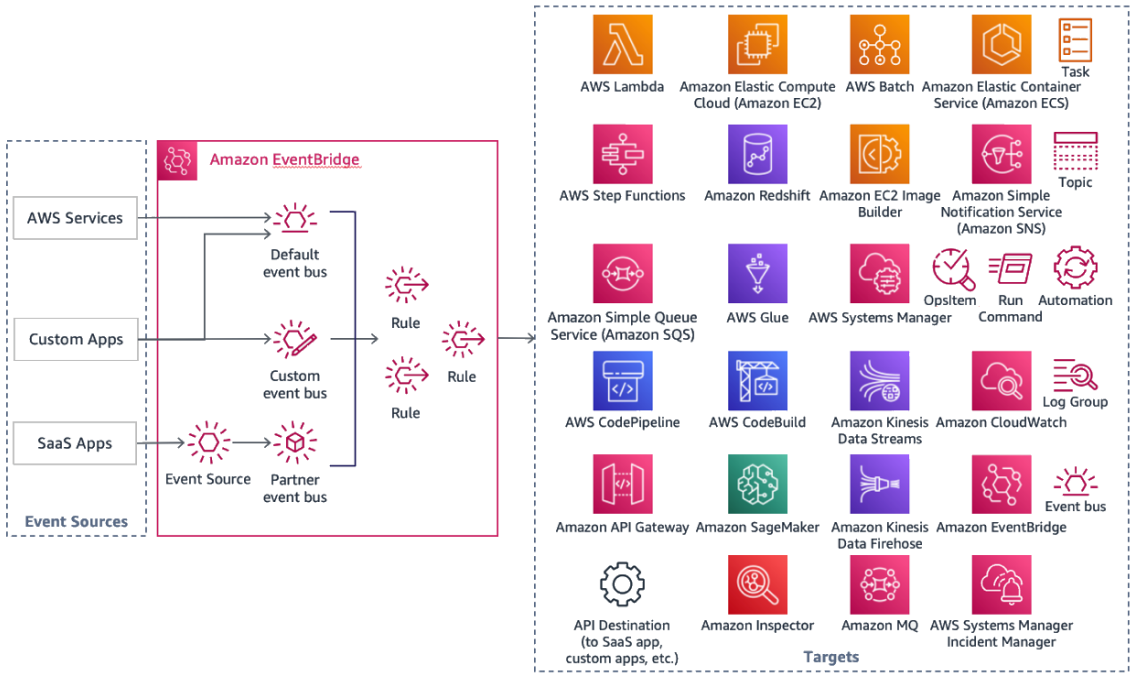
- Pub/Sub
- HTTP Triggers
- Cloud Storage events
- Firestore events

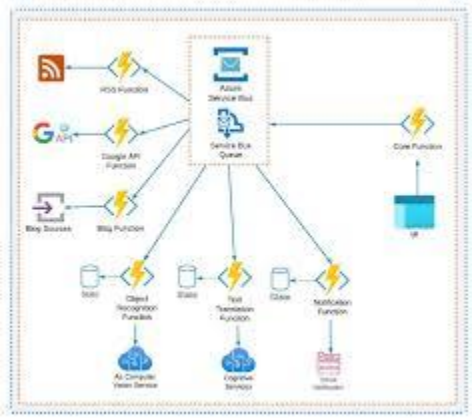
Example (Python):

```
def hello_world(request):
    return "Hello from Google Cloud Functions!"
```

---

3. Event-Driven Design Patterns





6

Common patterns that work well with Serverless:

3.1 Pub/Sub Events

Use cases:

- Messaging pipelines
- IoT data ingestion

AWS: SNS/SQS
Azure: EventHub/ServiceBus
GCP: Pub/Sub

3.2 Data Ingestion & Processing

Pattern:

- Events → Queue/Stream → Lambda → Database

Example:
S3 upload → Lambda → DynamoDB insert

3.3 API Gateway + Functions Pattern

Used for:

- Microservices
- REST APIs
- Mobile app backends

Flow:

Client → API Gateway → Function → Database

3.4 Fan-Out & Fan-In Pattern

After event arrives:

- One event → multiple parallel functions (Fan-Out)
- Aggregate results at end (Fan-In)

Example:

SNS Topic → Many Lambdas

3.5 Orchestration vs Choreography

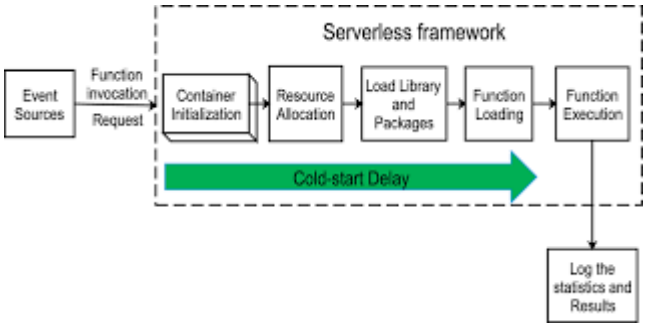
Choreography (Event-based):
Events trigger functions independently.

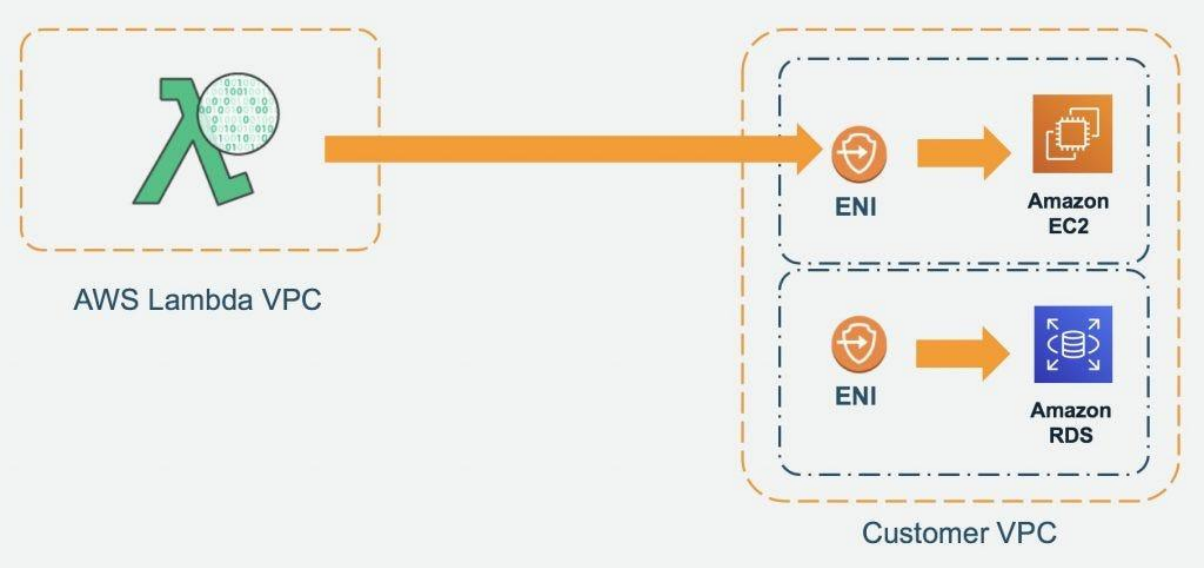
Orchestration (Managed workflow):

- AWS Step Functions
- Azure Durable Functions
- GCP Workflows

---

4. Security, Cold Starts & Cost Optimization





6

4.1 Security Considerations

You must secure:

Identity & Access

- Use Principle of Least Privilege (PoLP)
- Use IAM Roles (Lambda) / Managed Identities (Azure)

Secrets Management

- AWS Secrets Manager
- Azure Key Vault
- GCP Secret Manager

Network Security

- Put functions in private subnets
- Use VPC connectors for DB access
- Restrict API Gateway access via:
    - IAM
    - JWT Auth
    - OAuth 2.0
    - IP whitelisting

Data Protection

- Encrypt data at rest (KMS/Key Vault)

- Enable TLS 1.2 for APIs

4.2 Understanding Cold Starts

Cold start happens when:

- A function is invoked after a long idle period
- Cloud provider must initialize environment
- Impact more on languages like Java/.NET compared to Node/Python

Cold Start Impact:

- Increased latency for first request
- Bad for user-facing apps

How to reduce cold starts:

- Use **provisioned concurrency** (Lambda)
- Choose lightweight runtimes (Node/Python)
- Keep functions small
- Avoid heavy libraries
- Reduce VPC-attached function unless needed

4.3 Cost Optimization Techniques

Right-size memory

More memory = faster execution (cost may reduce)

Function Timeouts

Set proper timeout to avoid cost spikes.

Use Event Filters

Reduce unnecessary executions.

Avoid chatty functions

Batch data to reduce triggers.

Use CloudWatch/Stackdriver for monitoring

Review invocation patterns.

Use Step Functions instead of chaining lambdas

Reduces duplicated executions.

Storage Optimization

Use IA (Infrequent Access) tiers for intermediate storage.

Module 7 Summary

Topic	**Summary**
Serverless	Event-driven, auto-scaling, pay-per-use
AWS Lambda	Most mature FaaS platform
Azure Functions	Deep integration with Microsoft services
Google Cloud Functions	Pub/Sub-centric design
Event Patterns	Pub/Sub, API Gateway, Fan-Out/In
Security	IAM, VPC, secret management
Cold Starts	Startup delay for idle functions
Cost Optimization	Right-sizing, batching, concurrency control