Step-by-Step Self-Study Guide

(Newbie → Basic Pro)

WHITE MATRIX

PREPARED BY:

WHITE MATRIX
TRAINING TEAM

👋 Hi
Quick intro
Why learn Lambda?

AWS Lambda lets you run code without managing servers. Pay only while your code runs. It scales automatically. Use it to build APIs, background jobs, file processors, scheduled tasks and more — fast.

Before you start (prerequisites)

Make sure you have the basics:

- Comfortable with JavaScript (Node.js) or Python (pick one runtime to begin).
- Basic command line (terminal) skills.
- An AWS account (use a student/free tier account) and the ability to create an IAM user.
- Node.js (or Python) is installed locally.
- Optional but helpful: Git.

Mini task (prep):
Create an AWS account and an IAM user with programmatic access (Access Key + Secret), and install/configure AWS CLI locally:

```bash
npm install -g aws-cli   # or use package manager
aws configure          # enter Access Key, Secret, region (eg: us-east-1), default output json
```

?? What is Lambda (short, clear)

Think: small functions that run on demand in the cloud. Lambda executes your code when triggered (HTTP call, file upload, schedule, message), and you pay per execution.

01 Hello Lambda (Console) Your first function

Goal:
Create a Lambda function in the AWS Console and run it.

Do this:

1. Go to AWS Console → Lambda → Create function → Author from scratch.
    - Runtime: Node.js or Python.
    - Create a basic execution role (Lambda basic execution policy).
2. Paste a simple handler:

Node.js example

```
exports.handler = async (event) => {
 console.log('Event:', JSON.stringify(event));
 return { statusCode: 200, body: JSON.stringify({ message: 'Hello from Lambda!' }) };
};
```

3. Create a test event and click *Test*. Check CloudWatch Logs for console output.

Mini project:
"Hello Lambda" — function returns `Hello, <yourname>`. Trigger via Console test.

02 Local dev & simple deployment (SAM or Serverless CLI)

Goal:
Move from Console to local dev + repeatable deployment.

Options: AWS SAM (recommended for beginners) or Serverless Framework.

Do this:

1. Go to AWS Console → Lambda → Create function → Author from scratch.
   - Runtime: Node.js or Python.
   - Create a basic execution role (Lambda basic execution policy).
2. Paste a simple handler:

Node.js example

```
exports.handler = async (event) => {
 console.log('Event:', JSON.stringify(event));
 return { statusCode: 200, body: JSON.stringify({ message: 'Hello from Lambda!' }) };
};
```

3. Create a test event and click *Test*. Check CloudWatch Logs for console output.

Mini project:
"Hello Lambda" — function returns `Hello, <yourname>`. Trigger via Console test.

02 Local dev & simple deployment (SAM or Serverless CLI)

Goal:
Move from Console to local dev + repeatable deployment.

Options: AWS SAM (recommended for beginners) or Serverless Framework.

Key commands (SAM):

```
sam init                    # create template
sam build
sam local invoke HelloWorldFunction       # run locally (requires Docker)
sam local start-api              # run API locally
sam deploy --guided               # deploy to AWS
```

Mini project:
Create a simple Lambda + API Gateway using SAM that serves `/hello` returning JSON.

Why:
local iteration → faster dev and consistent deployments.

03 HTTP APIs Lambda + API Gateway

Goal:
Build a REST endpoint with API Gateway → Lambda.

- Create a Lambda that handles an API event (path, query string).
- Configure API Gateway (HTTP API or REST API) to invoke Lambda.
- Test with `curl` or a browser.

Example handler (Node):

```
exports.handler = async (event) => {
  const name = event.queryStringParameters?.name || 'Stranger';
  return { statusCode: 200, body: JSON.stringify({ msg: `Hello, ${name}` }) };
};
```

Mini project:
Build a simple To-Do backend:

- `GET /tasks` returns tasks from an in-memory array (later connect to DynamoDB).
- `POST /tasks` adds a task (accepts JSON).

04 S3 trigger
process uploads

Goal:
Make Lambda respond to S3 uploads.
Use case examples: extract text, parse CSV, virus scan, generate metadata.

Flow:
upload file → S3 trigger event → Lambda reads object via S3 SDK → process → write results to another bucket or DynamoDB.

Node snippet (read S3 object):

```
const AWS = require('aws-sdk');
const s3 = new AWS.S3();

exports.handler = async (event) => {
  const record = event.Records[0];
  const Bucket = record.s3.bucket.name;
  const Key = decodeURIComponent(record.s3.object.key.replace(/\+/g,' '));
  const obj = await s3.getObject({ Bucket, Key }).promise();
  const text = obj.Body.toString('utf-8');
  console.log('File content:', text.slice(0,200));
};
```

Mini project:
"CSV Uploader": upload a CSV → Lambda parses and inserts rows into DynamoDB table.

05 Scheduled tasks (EventBridge / CloudWatch Events)

Goal:
Run tasks on a schedule (cron) — backups, cleanup, reports.

- Create a CloudWatch Events / EventBridge rule → target: Lambda.
- Lambda runs on schedule, performs job (e.g., clean old records in DB).

Mini project:
Scheduled "Daily summary" function that aggregates a count from a DynamoDB table and saves a summary to S3.

06 Using DynamoDB (serverless DB)

Goal:
Persist data – use DynamoDB with Lambda.

- Create DynamoDB table, set primary key.
- Lambda uses AWS SDK to put/get items.

Node example (Dynamo put):

```
const AWS = require('aws-sdk');
const ddb = new AWS.DynamoDB.DocumentClient();

await ddb.put({ TableName: 'Tasks', Item: { id, title, done: false } }).promise();
```

Mini project:
Convert the To-Do backend to store tasks in DynamoDB (GET/POST/DELETE endpoints).

07 Environment variables config & secrets

Goal:
Configure your function for different environments.

- Use Lambda environment variables for config (DB table name, bucket name).
- For secrets (DB passwords), use AWS Secrets Manager and grant Lambda permissions to read secrets.

Mini project:
Add an environment variable `TASKS_TABLE` and update Lambda to use it. For secret access, mock a secret read (or use Secrets Manager in later steps).

08 Observability:
Logging, Metrics, Tracing

Goal:
Learn how to debug & monitor Lambda.

- Use `console.log()` (Node) / `print` (Python) → CloudWatch Logs.
- Create custom CloudWatch Metrics (or use Logs Insights).
- Optional: enable **AWS X-Ray** tracing for distributed tracing.

Mini project:
Add structured logs to your To-Do functions and create a CloudWatch Log Insights query to count errors.

09 Error handling, retries, DLQ

Goal:
Make functions robust.

- Understand sync vs async invocation: sync returns errors to the caller; async retries may occur.
- Configure **Dead-Letter Queues (DLQ)** (SNS or SQS) for failed async invocations.
- Implement idempotency in your Lambda to prevent duplicate processing.

Mini project:
Create a handler that writes to SQS on failure, and a separate Lambda that consumes the DLQ and alerts.

10 Security, IAM & least privilege

Goal:
Secure Lambda operations.

- Lambda needs an execution role. Give only required permissions (S3 read, DynamoDB put).
- Don't embed credentials. Use Secrets Manager or Parameter Store.
- Limit network access using VPC only when necessary (but note VPC cold start costs).

Mini project:
Create an IAM role with only DynamoDB access and attach to your Lambda. Verify restricted permissions work.

11 Packaging, size limits & performance

Goal:
Understand deployment packages, layers and cold starts.

- Keep package small (< 50 MB zipped for direct upload; use Layers for common large libs).
- For heavy native dependencies (e.g., image libs), use Lambda Layers or container images.
- Increase memory to improve CPU and execution speed (memory affects CPU proportionally).

Mini project:
Move a shared utility library into a Lambda Layer and reference it from two functions.

12 Advanced features (overview)

- Lambda Layers — share libraries.
- Provisioned Concurrency — reduce cold starts for latency-sensitive functions.
- Step Functions — orchestrate multiple Lambdas in workflows.
- Container image support — package functions as Docker images (useful for large binaries).

Capstone mini project: Build a small production-like serverless app:

- Frontend (static site) hosted on S3/CloudFront.
- Backend: API Gateway → Lambda (CRUD for tasks) → DynamoDB.
- File uploads: S3 trigger → Lambda processes and stores metadata.
- Scheduled job: daily summary via EventBridge.
- Deploy via SAM or Serverless Framework, add basic monitoring and alarms.

Quick cheat-sheet (commands & snippets)

AWS CLI configure

```bash
aws configure
```

SAM basic flow

```bash
sam init
sam build
sam local start-api   # test locally (docker)
sam deploy --guided
```

Invoke a Lambda

```bash
aws lambda invoke --function-name MyFunction --payload '{"key":"value"}' out.json
```

View logs

```bash
aws logs filter-log-events --log-group-name /aws/lambda/MyFunction --limit 50
```

Best practices & gotchas

- Keep functions single-purpose.
- Keep package size small; prefer native Node libs.
- Design for idempotency (retries happen).
- Avoid long (> 15 min) tasks — Lambda has timeouts.
- Monitor costs (many tiny executions can add up).
- Use versioning & aliases for safe deployments.
- Use Infrastructure as Code (SAM/Serverless/Terraform) — avoid manual console changes.

Suggested learning path (step order recap)

- Hello Lambda (Console) — basic test.
- Local dev + SAM — repeatable deploys.
- Lambda + API Gateway — build REST endpoints.
- S3 trigger processing.
- DynamoDB integration.
- Scheduled tasks (EventBridge).
- Observability (CloudWatch, X-Ray).
- Security (IAM, least privilege).
- Packaging, Layers, performance tuning.
- Orchestration (Step Functions) and capstone project.
- (Each step above has its mini-project — follow them in order.)

Recommended resources

- AWS Lambda Developer Guide (official AWS docs)
- AWS Serverless Application Model (SAM) docs
- Serverless Framework docs (alternate deployment tool)
- FreeCodeCamp / Traversy Media / Net Ninja — search YouTube for “AWS Lambda tutorial” (project-based videos)
- AWS workshops & tutorials (Hands-on Labs)
- AWS re:Invent talks (for deeper patterns)

Final Tips for Students

- **Start small and keep it simple:** Don't try to build everything at once. Begin with a single function running end-to-end (for example, API Gateway → Lambda → CloudWatch logs). Once that works, you'll have the confidence to build more complex pipelines.

- **Work like a professional:** Always commit your code to GitHub or GitLab. Document your steps in a README file, so anyone (even your future self) can repeat the process. Use deployment tools like AWS SAM or the Serverless Framework to package and redeploy easily, instead of relying only on the AWS Console.

- **Be mindful of the free tier:** AWS Free Tier gives you a generous limit, but resources like S3 storage or API Gateway calls can add up. Monitor your usage with CloudWatch and set a billing alert to avoid surprises.

- **Build step by step:** Don't jump straight into a large project. First, combine two simple mini-projects (like API + DynamoDB). Once you're comfortable, add an S3 trigger, then schedule tasks with CloudWatch Events. Think of it as Lego blocks—add one piece at a time.

- **Test often and log everything:** Use `print()` or `logger.info()` in your Lambda functions and check CloudWatch logs regularly. Debugging in Lambda is easier when you have clear, consistent logs.

- **Focus on real-world scenarios:** Instead of just doing "Hello World," try to build use cases that reflect real business needs—like image processing, notifications, or a simple task scheduler. This will not only improve your learning but also look impressive in your portfolio.

- **Document and share your work:** Keep track of your progress in a personal blog, GitHub repository, or LinkedIn post. Sharing what you build helps you learn better and also attracts the attention of recruiters and mentors.

<Thank_You>