ChatGPT

Migrating an AWS Lambda Project (Node.js 24) to ECMAScript Modules (ESM)

Node.js 24 ESM Support on AWS Lambda and Lambda@Edge

- **Full ESM Support:** AWS Lambda **fully supports ECMAScript modules** in Node.js 24.x. The Node 24 runtime can load both CommonJS and ESM code, and AWS **recommends using ES modules** to take advantage of modern features like top-level `await` [1]. By default, files with a `.mjs` extension are treated as ESM, while `.js` files are treated as CommonJS unless you set a flag in package metadata [2].
- **Lambda@Edge Compatibility:** As of November 2025, **Lambda@Edge also supports the Node.js 24 runtime**, meaning you can run Node 24 Lambdas at CloudFront edge locations [3]. This includes the ability to use ES module syntax in your Lambda@Edge functions. In practice, a Lambda@Edge function using Node 24 behaves the same as any Lambda function in terms of module loading. There is no inherent limitation preventing ESM usage at the edge – you simply deploy your function code as ESM and specify runtime `nodejs24.x` (in the US-East-1 region, per Lambda@Edge requirements).
- **Handler Syntax:** One change with Node 24 (unrelated to ESM vs CJS, but relevant for migration) is that **callback-based handlers are no longer supported**. All asynchronous Lambdas must use the modern async/`await` pattern [4]. Ensure your functions export an async handler (e.g. `export const handler = async (event) => { ... }`) rather than using `exports.handler = (event, ctx, cb) => { ... }`. This applies to both regional Lambdas and Lambda@Edge functions on Node 24.

Deploying ESM-Based Lambda@Edge Functions

- **Packaging ESM for Lambda:** To deploy an ES module Lambda (including Lambda@Edge functions), you need to ensure AWS Lambda recognizes your code as an ES module. There are two common ways to do this:
- **Use `.mjs` Extension:** Rename your handler file to use the `.mjs` extension (for example, `index.mjs` instead of `index.js`). Node will then treat this file as an ES module by default [2]. In your CloudFormation/Terraform config, the Lambda handler can still be specified as `index.handler` (Lambda will load `index.mjs` if present) [5] [1].
- **Use `"type": "module"`:** Alternatively (or additionally), include a `package.json` **with** `"type": "module"` in your function's deployment package. This setting tells Node to interpret **all `.js` files as ES modules** (so you could keep the file named `index.js`) [6]. If you take this approach, ensure the `package.json` is packaged with your Lambda code (and not filtered out).
- **Lambda@Edge Deployment:** There are no special steps beyond the above for Lambda@Edge. You will deploy your function code (with the ESM configuration) to the `us-east-1` **region**, and associate it with your CloudFront distribution as usual. The deployment artifact (a ZIP file) should contain either the `.mjs` file or the `package.json` marking modules as ESM. **No additional**

configuration is required on the CloudFront side – if the Lambda function is Node 24 and its code is an ES module, it will run as such.

- **Inline Code Note:** If you were writing Lambda code **inline in CloudFormation** (using the `ZipFile` property), by default that code is named `index.js` and treated as CommonJS. In Node 24, AWS added a way to handle this: you can set the `NODE_OPTIONS` environment variable to `--experimental-detect-module` for that function, which allows Node to auto-detect ESM vs CJS in `.js` files [7]. This is mainly a corner case – for a full project, it's usually better to package the code properly with the `.mjs` extension or a package.json.

AWS Powertools (TypeScript) Compatibility with ESM

- **Powertools v2 Supports ESM:** AWS Powertools for AWS Lambda (TypeScript) has **official ESM support** as of its v2.x releases. The library is dual-published so that you can import it using ES module syntax (`import { Logger, Tracer, Metrics } from '@aws-lambda-powertools/logger'`, etc.) in a Node.js 18+ or 24 Lambda [8]. AWS's announcement of Node 24 notes that Powertools (TypeScript) works on Node 24, which implicitly includes ESM usage [9]. If you're on an older 1.x version of Powertools, plan to upgrade to the latest v2 to get smooth ESM compatibility.
- **Using Powertools in ESM:** In practice, Powertools v2 lets you leverage top-level `await` and ES `import` statements. For example, you could do: `import { getSecret } from '@aws-lambda-powertools/parameters/secrets'; const apiKey = await getSecret('my-key');` at the module's top level to fetch a secret during the Lambda init phase [10]. This was not possible with CommonJS modules, and is a key benefit of moving to ESM.
- **CommonJS Fallback (X-Ray SDK):** One caveat: the Powertools Tracer utility depends on the AWS X-Ray SDK, which is **CommonJS-only**. The Powertools team has addressed this by recommending a small workaround when bundling: you can **polyfill** `require` **in an ESM context** for that specific need [11]. In practice, this means adding a line in your bundle (or using a bundler plugin/banner) like:

```
import { createRequire } from 'module';
const require = createRequire(import.meta.url);
```

  This allows you to use `require()` within an ES module file if needed (here it would enable the X-Ray SDK to be loaded). The Powertools documentation provides an example of injecting this code via esbuild's banner option [12]. If you use the Powertools layers instead of bundling, ensure you pull the **Node 24 compatible layer** ARN – the Lambda Powertools layers are published per runtime and v2 layers should support Node 18 and above.
- **Outcome:** Aside from the X-Ray note, **AWS Powertools works well under ESM**. Logging, metrics, and other utilities will function the same. Just import them normally and be sure to tree-shake (Powertools v2 is more tree-shakable). If you face any import issues, double-check that you didn't accidentally mix `require` and `import` – everything in your code using Powertools should be using ESM import syntax once you migrate.

Build & Configuration Changes for ESM Migration

- **TypeScript Configuration:** If your project is in TypeScript, update your **tsconfig** to target ESM. For Node.js 24, set `"module": "NodeNext"` or `"ES2022"` (NodeNext is ideal for full Node ESM support) and ensure `"moduleResolution": "NodeNext"`. This tells TypeScript to output native ESM code and use `.mjs` or package type info when resolving modules.

- **ESBuild Bundling Settings:** You'll need to adjust your bundler (ESBuild in this project) to output ESM:
- **Output Format:** Set `format: "esm"` in the ESBuild config/options so that the bundle is in ESM format (uses `import/export` in the output) [13]. Also use `platform: "node"` and an appropriate `target` (e.g. `node18` or `es2020`) to ensure compatibility with Node 24.
- **File Extension:** ESBuild by default might produce a `.js` file. You have two options: **a)** tell ESBuild to use a `.mjs` extension for the output, or **b)** include the `"type": "module"` marker. For example, if you're using AWS SAM CLI or CDK with ESBuild, you can configure the output format as ESM and still get a `.js` file – in that case, be sure the package.json with `"type": "module"` is in the zip [2] [13]. Some tools (AWS SAM's build workflow, CDK's NodejsFunction) now allow specifying the output **format as ESM**, and will automatically adjust the Lambda metadata. In AWS SAM, you'd do: `BuildProperties: { Format: ESM, EntryPoints: ["src/index.ts"] }` and SAM will handle including the right extension or package.json flag [14].
- **Tree Shaking:** One advantage of moving to ESM is smaller bundles. ESBuild will tree-shake dead code more effectively with ESM import/export semantics. (CommonJS `require` could hide usage, whereas ESM imports are static.) Make sure tree-shaking is not disabled. By default ESBuild bundles will tree-shake, but you can explicitly set `--tree-shaking=true` (or in config `treeShaking: true`) for peace of mind. This can substantially cut bundle size – e.g., only pulling in parts of AWS SDK v3 or Powertools that you actually use [15]. Smaller bundles generally mean **faster cold starts** and less memory usage.
- **Bundling CJS Modules:** With ESM output, ESBuild can still bundle CommonJS dependencies (it wraps them or inlines them as needed). However, if you run into a module that *dynamically* uses `require` at runtime in a way ESBuild can't analyze, you may need to mark it as external or use the `banner` trick discussed for Powertools. In practice, most packages (including the AWS SDK v3, Powertools, etc.) bundle fine. If using the **AWS SDK v3**, note that it is modular and ESM-ready; you should import only the clients/commands you need (and consider **not** bundling the entire SDK if you don't need to – AWS SDK v3 is included in the Node 24 runtime by default, or you can bundle selectively for performance).
- **Project Package.json:** Ensure your root `package.json` has `"type": "module"` (if you want all files to default to ESM). Also update your **scripts** if any dev tools assumed CommonJS (for example, if using Mocha or Jest, ensure they support ESM or use their ESM transpile modes). For Lambda specifically, the package.json's main role is the `type` flag – it's not required to list dependencies for runtime since you're bundling, but including it in the deployment can be useful for clarity.
- **Lambda Function Configuration:** In your IaC templates, update the **Handler** if necessary. Generally, you will keep the same handler string (e.g. `index.handler`). AWS Lambda will find `index.mjs` if it exists when you use `index.handler` (the runtime will attempt to load `index.mjs` or `index.js` as needed). If you explicitly name the file differently, adjust accordingly (for example, you could set Handler to `app.mjs.handler` if your file is named `app.mjs` – though usually sticking to `index` is simplest). Remember to change the runtime to `nodejs24.x` if you haven't already.

Potential Compatibility and Performance Considerations

- **Dependency Compatibility:** The vast majority of popular Node libraries have added ESM support, but a few haven't. After migrating, **watch for any runtime errors** like `ERR_REQUIRE_ESM` or

`ERR_MODULE_NOT_FOUND`. These usually indicate a module import issue. If a dependency only supports CommonJS (and you can't easily replace it), you can still use it by dynamic import: e.g. `const pkg = await import('commonjs-only-package')` or by using `createRequire`. This adds a bit of complexity, so weigh if it's easier to find an alternative library that supports ESM. In our context, AWS Powertools and AWS SDK are fully ESM-compatible.
- **Cold Start Time: Cold start performance should not degrade by moving to ESM** – in fact, it can improve. With ESM, you have the opportunity to do more work during the **init phase** (before the handler runs) thanks to top-level await. AWS demonstrated that an ES module Lambda could load configuration at init and thus respond faster on first invocation, reducing p99 cold-start latency by ~43% in their example [16]. Also, because your bundle can be smaller (unused code dropped), there's less code for the runtime to load on cold start. Any differences in the Node module loader (ESM vs CJS) at runtime are negligible compared to these benefits.
- **Node.js 24 Quirks:** The Node 24 runtime introduced a new **Runtime Interface Client** and stripped out some legacy features (like the callback-style completion) [4]. Ensure your code doesn't use deprecated Node Lambda patterns (e.g. `context.done` or `callbackWaitsForEmptyEventLoop`, which are removed in 24 [17] [18]). These changes are one-time adjustments. They aren't directly related to ESM, but you'll encounter them as you switch runtimes.
- **AWS Lambda@Edge Limits:** Running ESM at Lambda@Edge doesn't change the existing limits (memory 128 MB and timeouts of 5s or 30s depending on trigger) [19] [20]. Be mindful that any initialization code you add (with top-level await) in an Edge function still counts toward the overall execution time each invocation (especially for Viewer Request/Response which have 5s max). In other words, don't do anything at init that consistently takes a very long time, or you risk edge timeouts. Typically, initialization is quick (e.g. loading a config value, or setting up an SDK client).
- **Initial Launch Performance:** AWS noted that new runtimes can have slightly higher cold start until they "warm up" in the system [21]. Since Node.js 24 was recently added, you might see cold starts on the higher side for the first few weeks of usage. This isn't an ESM problem, but a general note for any brand new runtime. Over time, as Node 24 is used more, AWS's internal optimizations (like cached microVMs) will make cold starts as fast as previous runtimes. If your project is very latency-sensitive, you might consider enabling Provisioned Concurrency on critical functions during this period to keep cold starts zero.

Step-by-Step Migration Plan

1. Update AWS Lambda Runtimes: In your Terraform configuration, set the Lambda runtimes to `nodejs24.x` for all functions, including those used in Lambda@Edge. (AWS Lambda Node 24 is available in all regions and supported at Edge [3].) Deploying this runtime ensures the environment can run ESM code. Verify that your deployment pipeline (Terraform/OpenTofu) allows Node 24 – you may need to upgrade the AWS provider if it wasn't aware of Node 24 yet.

2. Upgrade Dependencies and Tools: Align your project dependencies with Node 24 and ESM: - Upgrade **AWS Powertools (TypeScript)** to the latest v2.x release to get ESM support [8]. Check the Powertools release notes for any minor breaking changes in v2 (mostly around import paths for middleware, etc., which you can adjust). - If you use other AWS libraries or SDK v3 clients, use the latest versions. AWS SDK v3 is included in the Node 24 runtime, but you can also bundle a newer version if needed. - Update any build tooling (ESBuild, TypeScript, etc.) to versions that fully support Node 24 and ESM output. For example, ensure ESBuild is reasonably up-to-date (v0.18+ should be fine; there were earlier bugs with ESM bundling in older versions).

3. Convert Code to ESM Syntax: This is a core step: - **Module Exports:** Rewrite any CommonJS export definitions. For each Lambda handler file, replace `exports.handler = ...` or `module.exports = { handler: ... }` with an **ES module export**. The typical pattern is to use named exports – e.g. `export const handler = async (event) => { ... }`. Ensure that the Handler value in your Terraform config matches the file name and export name (usually `index.handler` if you export `handler`). - **Imports:** Find all `require()` statements and convert them to `import` statements. E.g. `const AWS = require('aws-sdk')` becomes `import AWS from 'aws-sdk';` (and note that AWS SDK v3 uses named imports for clients: `import { S3Client } from '@aws-sdk/client-s3';`). In TypeScript, you might need to adjust import syntax slightly (e.g. import * as module vs default imports) depending on the library. - **File Extensions:** If you decided to use `.mjs` for your files, rename them accordingly. Update any internal imports between your files to use the new filenames. (If files have no extension in import paths, Node's ESM loader won't automatically add extensions – you either include the extension in the import, or use a bundler that resolves it. Bundling with ESBuild will handle extension resolution for the bundle.) - **Top-Level Await:** As part of ESM conversion, consider if there are initialization tasks that make sense to do at the top level. For example, establishing a database connection, reading a config from S3 or SSM Parameter Store, or instantiating an AWS SDK client can be done outside the handler. With ESM, you can even `await` asynchronous setup at the top level [22]. This can improve cold start behavior (the one-time init happens before the function is invoked). It's not mandatory, but it's a benefit you might take advantage of during the refactor.

4. Adjust Build Configuration (ESBuild/Webpack): Modify your build steps to produce ESM-compatible output: - **ESBuild CLI/Script:** Add the `--format=esm` flag if using CLI, or `format: 'esm'` in the build script config [13]. Ensure `platform: 'node'` is set so that ESBuild knows to bundle for Node (this handles Node built-ins correctly). Also, consider setting `mainFields: ['module','main']` so that if libraries provide an ESModule build, ESBuild will prefer it – this yields smaller bundles [23]. - **Output Extension:** Configure ESBuild's output extension if possible. There isn't a direct flag for extension, but if you specify an output file name ending in `.mjs`, ESBuild will honor it. For example, `esbuild index.ts --bundle --outfile=index.mjs --format=esm --platform=node`. If you use a plugin or script, you might manually rename the file after bundling. If this is cumbersome, then rely on `"type": "module"` in package.json as mentioned. The key is to avoid ending up with an ESM bundle named `.js` without a module type indicator. - **Bundle Verification:** After building, **inspect the output** (the bundle file). It should start with `import` statements or have the top-level code as an ES module (if everything is bundled into one file, ESBuild might convert imports to its internal loading mechanism – that's fine). The absence of `require(` at the top level is a clue that it's ESM. If you used the Powertools tracer, verify that the `createRequire(import.meta.url)` snippet is present in the bundle (if you added the banner). This ensures the X-Ray SDK can load. - **Source Maps:** Not directly related to ESM, but if you generate source maps, ensure they're configured as you want (you might disable them for production for smaller package size, or include them if needed for debugging). Just remember that even if source maps are off, stack traces in AWS may look different under ESM (they'll show module URLs). This is normal.

5. Deployment Package Changes: When using Terraform, you might be zipping artifacts manually or via a build script: - Include the `package.json` (with the `"type": "module"` field) in the zip if using that mechanism. Double-check your `.npmignore` or zip script isn't excluding it. - If you have **Lambda layers** (e.g., a layer for FFmpeg or other binaries), those are unaffected by ESM – they remain as they are. But if you had a **layer with Node code** (e.g., a layer containing Powertools or other libs), ensure that code is Node 24 compatible (you might just remove such a layer and bundle the code directly, since Powertools is now in your bundle). - Make sure the **handler name** in your Terraform config matches the new setup. For example, if you renamed the file to `index.mjs` and are still using `index.handler`, that's fine. If you chose a different naming scheme, update `aws_lambda_function.handler` accordingly.

6. Testing and Verification: Before deploying to production: - **Local Test:** Use `npm run build` to produce the new artifact, and then invoke the Lambda handler locally (you can use the AWS SAM CLI's `sam local invoke`, or simple Node execution if no AWS services are needed, or a small harness script). This will catch any immediate syntax or import errors. Common issues at this stage would be forgetting an extension in an import path (if not bundling fully) or encountering a dependency error. - **Unit/Integration Tests:** Run your test suite. If you were using a test runner that doesn't support ESM, you may need to switch or upgrade it. For instance, Jest prior to v28 had limited ESM support – you'd want the latest version or switch to something like Vitest. Mocha has an `--loader` flag for ESM. This is beyond the Lambda itself, but important for your development workflow. - **Deploy to a test environment:** Deploy the Terraform stack to a dev or staging environment. Ensure the Lambdas deploy successfully and **create a few test invocations** (e.g., call the APIs or triggers that use these Lambdas). Watch CloudWatch Logs for any errors. A common error could be `ERR_MODULE_NOT_FOUND` if the Lambda couldn't find the handler – this would indicate a packaging/handler naming issue. If you see `Warning: To load an ES module, set "type": "module"` or similar, it means the runtime thought your file was CommonJS – likely the package.json with type:module wasn't included or the file wasn't .mjs. Fix that and redeploy.
- **Edge Functions:** For Lambda@Edge, after deployment, invalidate the CloudFront cache or simply exercise the functionality to trigger the Edge Lambda. Verify through CloudFront logs or Lambda logs that it ran. (Note: CloudFront will cache the association, so if you replaced a Node 16 function with a Node 24 one, ensure the distribution got updated properly. There might be a slight delay in replication.)

7. Monitor and Optimize in Production: Once the ESM-based Lambdas are live: - Monitor memory and duration metrics. ESM by itself doesn't usually increase memory usage – if anything, reduced code might use slightly less memory. If you do notice any spikes, profile what might be loading at init. It's possible you pulled in a large dependency at the top-level. Tree-shaking usually mitigates this, but keep an eye out. - Observe cold start logs (you can instrument cold start by logging something in a branch that runs only on first invocation – e.g., `if (process.env.AWS_LAMBDA_INITIALIZATION_TYPE === "on-demand") console.log("Cold start")`). Compare cold start times before vs after. You may find them similar or improved, especially if you optimized what runs at startup. If any function's cold start is higher, consider enabling Provisioned Concurrency or investigating if the bundle size can be reduced further (e.g., remove unused font libraries or other large packages).
- Take advantage of the new Node 24 features in the long run – for example, the **AWS SDK v3 is included** in Node 24 so you might not even need to bundle AWS clients (you can rely on the runtime's copy to save space). Also Node 24 has improved performance for certain APIs and a newer V8, which is a bonus.

8. Rollback Plan: Have a rollback strategy just in case. Ideally, keep the last known-good CommonJS deployment artifacts or a Git branch. If something goes wrong that you can't quickly fix, you can redeploy the previous version to stabilize. However, given that Node.js 24 and ESM are well-supported by AWS, any issues are likely to be fixable with configuration rather than requiring a full rollback. Common fix scenarios: adding a missing `"type": "module"`, adjusting an import path, or including a polyfill.

Gotchas and Additional Tips

- `require()` **in ESM:** If you *must* use `require` (for example, importing JSON files or a module that isn't ESM), remember that in an ES module context `require` is not defined by default. Use `createRequire` from Node's `'module'` package to create a require function [12]. Powertools' docs explicitly mention this for the X-Ray SDK case [11]. This is preferable to enabling experimental flags globally.
- **ESM and AWS SDK v2:** If by chance you were still using AWS SDK v2, note that Node 18+ runtime no longer includes it (and Node 24 definitely doesn't include v2). This is a good time to ensure you've migrated to AWS SDK v3, which has better tree-shaking and native ESM support. If you had to use v2 for something, you'd have to bundle it yourself (and it's CJS, which would bloat your package). Avoid that if possible.
- **Testing Lambda@Edge Locally:** Lambda@Edge functions can be tricky to test locally because they run in CloudFront. But since they are just Lambda functions, you can invoke their handler with a sample CloudFront event. After migration, test that your Edge function handlers (viewer/origin requests, etc.) still manipulate the event object correctly and return it. Nothing about that logic should change with ESM – the cloud event structure is the same – but it's good to validate. The code examples in the Lambda@Edge Node 24 article showed using `exports.handler = async (event) => { ... }` for Edge triggers [24]. You'll be doing `export const handler = async (event) => { ... }` now. The behavior (adding headers, modifying the request/response) remains the same.

Conclusion

In summary, **migrating the project to ESM on Node.js 24 is definitely feasible and comes with benefits**. Node 24 on Lambda (including Lambda@Edge) fully supports ES modules, and AWS's tooling and libraries (like Powertools) have evolved to first-class support for ESM [3] [8]. The migration will involve updating your syntax (imports/exports), tweaking build configurations to output ESM (`format: "esm"` and `"type": "module"` indicators), and testing thoroughly.

By following the steps above, you'll convert the project to ESM and likely enjoy cleaner code and potentially faster cold starts (thanks to smaller bundles and top-level await) [16]. AWS's official guidance even encourages using ES modules for Lambda due to these advantages [1]. Just watch out for a few compatibility niggles (like any lingering CommonJS-only dependencies), and you should be able to complete the migration with relatively little pain. With Node.js 24 LTS and ESM, your project will be using the latest JavaScript features on AWS Lambda, positioning it for easier maintenance and better performance going forward.

Sources:

- AWS Lambda Node.js 24 announcement (Lambda@Edge support and Powertools mention) [3]
- AWS Lambda documentation – *"CommonJS and ES Modules"* (recommends ESM, using `.mjs` or `"type": "module"`) [1] [2]
- AWS Powertools for TypeScript – Upgrade guide (ESM support in v2, require workaround) [8] [11]
- AWS Compute Blog – *"Using Node.js ES modules and top-level await in AWS Lambda"* (performance benefits of ESM/top-level await) [16] [25]

- AWS Compute Blog – Node.js 24 runtime updates (cold start considerations) [21]
- Optimization Blog – *“Optimize TypeScript Bundles for AWS Lambda with ESBuild”* (advice to use ESM format and `"type": "module"`) [13]

---

[1] [2] [5] [6] Define Lambda function handler in Node.js - AWS Lambda
https://docs.aws.amazon.com/lambda/latest/dg/nodejs-handler.html

[3] [9] AWS Lambda adds support for Node.js 24 - AWS
https://aws.amazon.com/about-aws/whats-new/2025/11/aws-lambda-nodejs-24/

[4] [7] [17] [18] [21] Node.js 24 runtime now available in AWS Lambda | AWS Compute Blog
https://aws.amazon.com/blogs/compute/node-js-24-runtime-now-available-in-aws-lambda/

[8] [10] [11] [12] [14] Upgrade guide - Powertools for AWS Lambda (TypeScript)
https://docs.aws.amazon.com/powertools/typescript/2.0.2/upgrade/

[13] [15] [23] Optimize TypeScript Bundles for AWS Lambda with ESBuild
https://cajuncodemonkey.com/posts/bundles-for-aws-lambda-with-esbuild/

[16] [22] [25] Using Node.js ES modules and top-level await in AWS Lambda | AWS Compute Blog
https://aws.amazon.com/blogs/compute/using-node-js-es-modules-and-top-level-await-in-aws-lambda/

[19] [20] [24] I Tried Node.js 24 with Lambda@Edge | DevelopersIO
https://dev.classmethod.jp/en/articles/lambda-edge-node-js-24/