Comprehensive Architectural Evaluation and Migration Strategy for ECMAScript Module Implementation in Serverless Media Processing Pipelines

The shift from CommonJS to ECMAScript Modules represents a fundamental transition in the JavaScript ecosystem, moving from a legacy, synchronous module resolution system to a standardized, asynchronous, and statically analyzable paradigm. For a project as technically sophisticated as the media downloader under evaluation—which integrates Node.js 22.x, TypeScript, and a single-table DynamoDB architecture within a pnpm monorepo—this migration is not merely a syntactic update but a structural evolution.[1] The feasibility of such a migration depends on a nuanced understanding of the AWS Lambda runtime, the constraints of the CloudFront Edge network, and the specific dependencies that define the core business logic of the application.

Structural Anatomy of the Media Downloader Ecosystem

The current state of the repository indicates a high level of technical maturity. The project is deployed using OpenTofu, a community-driven fork of Terraform, which manages a serverless stack comprising API Gateway, Lambda, S3, and DynamoDB.[1] The application logic is built on a "convention over configuration" tenant, prioritizing the minimization of external dependencies while maximizing the utilization of native AWS services.[1]

At the center of the database layer is ElectroDB, a type-safe ORM designed for single-table DynamoDB patterns, which powers the entity definitions for users, sessions, accounts, and media files.[1] The authentication layer is handled by Better Auth 1.4.3, for which the project owner has developed a custom ElectroDB adapter.[1] This specific detail is critical for an ESM migration, as both ElectroDB and Better Auth must be evaluated for their ESM compatibility and their behavior in a statically linked module environment.

Component	Current Implementation	ESM Migration Priority
Runtime	Node.js 22.x (Managed)	High: Upgrade to Node.js 24.x [2]

Module Format	Hybrid (TS/CJS)	High: Native ESM with type: module [3]
Authentication	Better Auth + ElectroDB	Medium: Ensure ESM entry points are used [4]
Infrastructure	OpenTofu (Terraform)	Medium: Update runtime and build targets [5]
Bundling	Webpack / esbuild	High: Switch to ESM-only output [6]
Testing	Jest with ESM support	Low: Already ESM-compatible [1]

The project already demonstrates an awareness of ESM, as evidenced by its use of Jest with explicit ESM support and the presence of .mjs configuration files for modern tooling like ESLint.[1] This hybrid posture simplifies the migration, as many of the development-time hurdles associated with ESM testing have already been addressed. However, the production runtime and the bundling pipeline require a concerted effort to reach a pure ESM state, particularly when considering the user's desire to move logic to the CloudFront Edge.

The Evolution of the Node.js 24 Runtime and RIC Improvements

The release of the Node.js 24 runtime in late 2025 has provided the ideal environment for this migration. This runtime introduces a new implementation of the Runtime Interface Client (RIC), written in TypeScript, which streamlines the integration between function code and the Lambda service.[2] A significant change in Node.js 24 is the removal of support for legacy callback-based function handlers.[7] This forces a shift toward the async/await pattern, which is natively supported and encouraged in ESM environments.

For the media downloader, the migration to Node.js 24 offers several advantages. The new RIC reduces the overhead of the Lambda execution environment, while support for ESM-native features like top-level await allows for faster initialization.[7] Furthermore, Node.js 24 introduces improved handling of unresolved promises, ensuring that Lambda no longer waits for background work that is not explicitly awaited once the handler returns.[7] This consistency is vital for maintaining the performance profile of media processing tasks, which often involve complex asynchronous chains.

The mathematical model for cold start latency in this environment can be defined by the sum of environment setup time, module evaluation time, and handler execution. By utilizing ESM

and top-level await, we can shift the complexity of module evaluation into the initialization phase:

$$T_{total} = T_{env\_setup} + T_{init\_eval} + T_{handler}$$
In a CJS environment, T_{init_eval} often involves synchronous require() calls that block the main thread. In an ESM environment, T_{init_eval} can leverage asynchronous loading, and when combined with Provisioned Concurrency, the impact of T_{init_eval} on the end-user request is virtually eliminated.[9]

Feasibility of Lambda@Edge Migration

The user's query specifically highlights the desire to migrate to ESM at the Edge. Lambda@Edge allows code to execute at CloudFront edge locations, but it operates under a set of constraints that differ markedly from standard Lambda functions.

Hardware and Networking Constraints

Lambda@Edge functions are constrained by architecture and memory. While standard Lambda functions can leverage the arm64 (Graviton) architecture for better price-performance, Lambda@Edge remains limited to x86_64.[11] This has implications for the media downloader's current build pipeline, which may need to be adjusted if it currently targets arm64.

Constraint	Lambda@Edge (Viewer)	Lambda@Edge (Origin)	Standard Lambda
Memory	128 MB	3,008 MB	10 GB
Timeout	5 Seconds	30 Seconds	15 Minutes
Code Size	1 MB (Compressed)	50 MB (Compressed)	250 MB (Uncompressed)
Env Vars	Not Supported	Not Supported	Fully Supported
Layers	Not Supported	Not Supported	Fully Supported

.[11]

The most significant hurdle for the media downloader is the lack of support for Lambda

Layers at the Edge.[11] The current project uses a Lambda Layer to deploy the yt-dlp binary, which is central to its media downloading functionality.[1] Because Layers are unsupported at the Edge, this binary would have to be bundled directly into the function package. Given that viewer-facing events are capped at 1MB, bundling a binary of that size is likely impossible. Even for origin-facing events with a 50MB limit, the 30-second timeout of Lambda@Edge would likely be insufficient for long-running video downloads.[12]

Strategic Partitioning of Logic

The research suggests that a full migration of the entire media processing pipeline to the Edge is impractical due to these resource limits. However, the project's custom authorizer and Feedly webhook handler are prime candidates for Edge migration.[1] By moving authentication logic to the Edge, the project can reject unauthorized requests at the point of presence (PoP), saving the costs of invoking more expensive, heavy-duty processing Lambdas and reducing latency for the user.[11]

Technical Implementation of the ESM Migration

The transition to a pure ESM state involves a multi-step process targeting the package configuration, the source code syntax, and the build infrastructure.

Package Manifest and Monorepo Configuration

The first step is to modify the root package.json to include "type": "module".[3] In a pnpm monorepo, this change should be cascaded through all workspace packages to ensure a consistent module system. This prevents the "dual package hazard," where a single application might inadvertently load two versions of the same library—one CJS and one ESM—leading to state inconsistencies and increased memory usage.[4]

For tools that require CommonJS configuration files, the extensions must be explicitly changed to .cjs.[9] The project's existing .dependency-cruiser.cjs file is already correctly named to signal its CJS nature to the Node.js loader.[1] Any other configuration files, such as those for Webpack or ESLint, should be reviewed and renamed if they contain module.exports syntax.

Syntax Transformation and Global Variable Replacements

CommonJS globals such as __dirname and __filename are unavailable in ESM.[18] This is an intentional design choice to ensure that JavaScript modules work consistently across different environments, including browsers and Node.js.[19] The recommended approach for Node.js 20.11 and later is to use the native import.meta.dirname and import.meta.filename properties.[20]

For legacy compatibility or cross-platform consistency, the project can use the following

boilerplate:

JavaScript

```
import { fileURLToPath } from 'url';
import { dirname } from 'path';

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);
```

.[19]

The migration also requires replacing require() with import statements. For TypeScript, this involves ensuring that the compilerOptions.module is set to ESNext or NodeNext.[22] The AWS SDK v3, which the project already uses, is natively optimized for this.[23] By using named imports like import { S3Client } from "@aws-sdk/client-s3", the bundler can perform tree-shaking, which is critical for staying within the size limits of Lambda@Edge.[23]

Handling the Lack of Environment Variables at the Edge

A significant challenge identified in the research is that Lambda@Edge does not support environment variables.[11] For a project that relies on DynamoDB ARNs and API keys, this requires a workaround. The standard industry practice for 2025 involves using CloudFront Origin Custom Headers.[14]

Static configuration values can be set as custom headers in the CloudFront origin settings and accessed via the request.origin.custom object in the Lambda code.[26] For more dynamic configurations, the project can leverage the CloudFront KeyValueStore (KVS), which provides a low-latency, globally replicated store for key-value pairs that functions can access without the overhead of a full AWS SDK call to SSM or Secrets Manager.[12]

Build Engineering and Bundling Optimization

The project utilizes both Webpack and esbuild.[1] For an ESM migration, esbuild is the superior choice due to its native support for ESM and its ability to handle tree-shaking with significantly less overhead than Webpack.[6]

esbuild Configuration for ESM Output

To produce a valid ESM bundle for AWS Lambda, the esbuild configuration must be

specifically tuned:

- **Format**: Set to esm to produce standard ES module output.[6]
- **Target**: Use es2022 or later to support top-level await.[6]
- **Out Extension**: Set the output extension to .mjs. This is critical for the Node.js runtime to identify the file as an ES module regardless of the package.json setting.[6]
- **Banner**: Because some dependencies may still use require() internally, injecting a banner can provide a shim: import { createRequire } from 'module'; const require = createRequire(import.meta.url);.[6]

The Impact of Bundling on Cold Start Performance

Optimizing the bundle is not just a deployment requirement but a performance imperative. Data suggests that smaller deployment packages result in faster cold starts because the Lambda service has to read less data from the internal storage layer.[6] In a media downloader, reducing the cold start latency of the authorizer or the webhook processor directly translates to a more responsive user experience.

Bundling Strategy	Bundle Size	Cold Start (Approx)
CJS (No Bundling)	~15 MB	1.62 s
CJS (Bundled)	~5 MB	1.35 s
ESM (Bundled + Tree-shaking)	~3 MB	1.22 s

.[6]

The use of mainFields: ["module", "main"] in esbuild ensures that the bundler prioritizes the ESM version of third-party libraries (like Zod or Ramda), which often provides better tree-shaking opportunities.[25]

Database and Authentication Considerations

The project's reliance on ElectroDB and Better Auth warrants a deeper dive into how these libraries behave in an ESM-native environment.

ElectroDB and Single-Table Design

ElectroDB is designed to handle complex relational patterns in DynamoDB through a single-table architecture.[1] Because ESM supports static analysis, the ElectroDB entity

definitions can be more effectively bundled. This is particularly beneficial when the application needs to load specific collections or entities for a given function without pulling in the entire database schema.[1]

During migration, it is important to verify that the ElectroDB client is initialized using top-level await if the connection or schema validation requires any asynchronous setup.[9] This ensures that the database client is "warm" before the handler receives its first request.

Better Auth and Custom Adapters

The project includes a custom ElectroDB adapter for Better Auth.[1] Better Auth 1.4.3 is a modern framework, and its architecture is generally compatible with ESM. However, the custom adapter must be tested for its behavior when bundled. The transition to ESM might expose issues with how the adapter handles session management or cryptographic operations if it relied on CJS-specific behaviors for module caching.[4]

For the authentication logic specifically, the absence of environment variables at the Edge is a critical concern. If the Better Auth secret is stored in an environment variable, it must be moved to KVS or injected at build time using esbuild's define property.[14] This ensures that the Edge-based authorizer can validate tokens without requiring a trip back to the origin for configuration data.

Deployment and Infrastructure Management via OpenTofu

The media downloader project utilizes OpenTofu for its infrastructure as code, which provides a flexible way to manage the transition from standard Lambda to Lambda@Edge.[1]

Updating Lambda Runtimes and Handlers

In the OpenTofu configuration, the aws_lambda_function resource for each component must be updated to use the nodejs24.x runtime.[2] If the build process now outputs .mjs files, the handler property must reflect this change (e.g., index.handler becomes index.mjs.handler depending on the bundler configuration).[6]

Managing Edge Deployments

Lambda@Edge requires a more complex deployment workflow than standard Lambda. The function must be created in the us-east-1 region, and a specific version must be published and associated with a CloudFront distribution.[14] OpenTofu can automate this by using a separate provider for us-east-1 and utilizing the aws_lambda_function_event_invoke_config or the aws_cloudfront_distribution resource to set up the triggers.[32]

Deployment Parameter	Standard Lambda	Lambda@Edge
Region	Any supported region	Must be us-east-1 [32]
Versioning	Optional (can use $LATEST)	Required (must use numbered version) [14]
Logging	Single CloudWatch Group	Regional CloudWatch Groups
Concurrency	Reserved/Provisioned	Regional Pool [11]

.[11]

For the yt-dlp binary layer, the project uses a null_resource and a data.archive_file to manage the layer's lifecycle.[1] This mechanism remains valid for standard Lambda functions even after migrating the code to ESM. The ESM code will simply import the binary from the /opt/bin/ directory as it did before, though it must now use import.meta.dirname to construct the correct absolute path to the binary.[19]

Performance Metrics and Operational Benchmarking

The transition to ESM is often justified by the performance gains in cold start latency and execution duration. In a serverless media processing application, these gains can be significant.

Cold Start Mitigation with Top-Level Await

The use of top-level await allows developers to perform heavy initialization tasks—such as establishing a DynamoDB connection or loading the ElectroDB schema—during the "Init" phase of the Lambda lifecycle.[9] When Provisioned Concurrency is used, this initialization occurs before any request is received, essentially reducing the cold start latency to zero for the end-user.[9]

In Node.js 24, the RIC is optimized to handle these initialization tasks more efficiently. Benchmarks from 2025 show that Node.js 24 consistently outperforms earlier runtimes in both cold and warm execution, particularly when using modern ESM syntax.[35] This is critical for the Feedly webhook handler, which must process requests quickly to avoid timeouts from the Feedly API.

Cost Implications of ESM and Edge Migration

Lambda@Edge pricing is based on duration and request count, similar to standard Lambda, but with different unit costs and no free tier.[11]

Billing Metric	Standard Lambda (us-east-1)	Lambda@Edge
Per 1M Requests	$0.20	$0.60 [32]
Per GB-Second	$0.0000166667	$0.00005001 [11]

.[11]

While Lambda@Edge is more expensive per invocation, the architectural benefit of rejecting unauthorized requests at the Edge can lead to overall cost savings by preventing these requests from ever reaching the more expensive media processing functions and the S3/DynamoDB origins.[11]

Security and Dependency Integrity in the Migration

The project emphasizes security through pnpm lifecycle protection and sops for secret management.[1] An ESM migration must maintain these standards.

Auditing the Dependency Graph

ESM's static nature makes it easier to audit the dependency graph for vulnerabilities. Tools can trace the import statements more reliably than they can trace dynamic require() calls.[4] The project's use of pnpm with security hardening is already a best practice that should be maintained throughout the migration.[1]

Managing Secrets in an ESM Monorepo

The use of sops for secrets management involves decrypting .env files and injecting them into the environment.[1] For standard Lambda, this remains the most secure method. For Lambda@Edge, where environment variables are unavailable, the build pipeline must be adjusted to inject these secrets during the bundling process using esbuild's define or banner features, ensuring that sensitive data is handled securely during the transformation from CJS to ESM.[6]

Conclusion and Recommended Strategy

Based on the evaluation of the aws-cloudformation-media-downloader project and the current state of the Node.js 24 ecosystem, a migration to ESM is not only possible but strategically advisable. The migration will unlock significant performance benefits, particularly through the use of top-level await and tree-shaking, while future-proofing the codebase against the deprecation of legacy CommonJS runtimes.[6]

Summary of Strategic Recommendations

The following roadmap provides a structured path for the migration:

1. **Environment Preparation**: Update the local and CI/CD environments to Node.js 24. Modify the root package.json to include "type": "module" and ensure that all workspace packages are aligned.[2]
2. **Syntax Refactoring**: Use automated tools to replace require() with import and module.exports with named export statements. Implement import.meta.dirname for path resolution, particularly for the /opt/ directory used by the yt-dlp binary layer.[18]
3. **Edge Partitioning**: Migrate the Feedly webhook authorizer and request routing logic to Lambda@Edge. Use CloudFront KVS or Origin Custom Headers to manage configuration values in the absence of environment variables.[11]
4. **Bundling Optimization**: Standardize on esbuild for all Lambda builds. Configure the build to output .mjs files with tree-shaking enabled to minimize the footprint of the Edge-based functions.[6]
5. **Infrastructure Updates**: Update OpenTofu scripts to reflect the new runtime and handler filenames. Ensure that Edge-based resources are correctly versioned and associated with the CloudFront distribution in the us-east-1 region.[5]

By adopting this strategy, the media downloader project will leverage the full power of modern serverless architectures, providing a more responsive, efficient, and maintainable platform for media processing.[6]

Works cited

1. j0nathan-ll0yd/aws-cloudformation-media-downloader: A Terraform project for downloading media (e.g. YouTube videos) via Feedly - GitHub, accessed December 21, 2025, https://github.com/j0nathan-ll0yd/aws-cloudformation-media-downloader
2. AWS Lambda adds support for Node.js 24, accessed December 21, 2025, https://aws.amazon.com/about-aws/whats-new/2025/11/aws-lambda-nodejs-24/
3. How to update Lambda Functions from Common JS to ECMAScript - AndMore Dev, accessed December 21, 2025, https://www.andmore.dev/blog/update-commonjs-to-esm/
4. CommonJS vs. ES Modules | Better Stack Community, accessed December 21, 2025, https://betterstack.com/community/guides/scaling-nodejs/commonjs-vs-esm/

5. Building Lambda functions with Node.js - AWS Documentation, accessed December 21, 2025, https://docs.aws.amazon.com/lambda/latest/dg/lambda-nodejs.html
6. Build & Deploy Tips(The Ultimate Guide to AWS Lambda Development Chapter 2), accessed December 21, 2025, https://georgemao.medium.com/build-deploy-tips-the-ultimate-guide-to-aws-lambda-development-chapter-2-60f7b3c0b0cc
7. Node.js 24 runtime now available in AWS Lambda | AWS Compute Blog, accessed December 21, 2025, https://aws.amazon.com/blogs/compute/node-js-24-runtime-now-available-in-aws-lambda/
8. Lambda@Edge - Noise, accessed December 21, 2025, https://noise.getoto.net/tag/lambdaedge/
9. Using Node.js ES modules and top-level await in AWS Lambda | AWS Compute Blog, accessed December 21, 2025, https://aws.amazon.com/blogs/compute/using-node-js-es-modules-and-top-level-await-in-aws-lambda/
10. AWS Cloud Development Kit – AWS Compute Blog, accessed December 21, 2025, https://aws.amazon.com/blogs/compute/category/developer-tools/aws-cloud-development-kit/feed/
11. Lambda@Edge – Use Cases, Pricing & Examples (2025) - StormIT, accessed December 21, 2025, https://www.stormit.cloud/blog/lambda-at-edge/
12. Restrictions on Lambda@Edge - Amazon CloudFront, accessed December 21, 2025, https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/lambda-at-edge-function-restrictions.html
13. Serverless Framework - AWS Lambda Events - CloudFront, accessed December 21, 2025, https://www.serverless.com/framework/docs/providers/aws/events/cloudfront
14. AWS Lambda@Edge - Optimizely, accessed December 21, 2025, https://docs.developers.optimizely.com/feature-experimentation/docs/aws-lambda-at-edge
15. Dynamically Configure Your Lambda@Edge Functions - DEV Community, accessed December 21, 2025, https://dev.to/aws-builders/dynamically-configure-your-lambda-edge-functions-2pkp
16. A Step-by-Step Guide to Migrating AWS Lambda Functions to ECMAScript Modules (ESM), accessed December 21, 2025, https://businesscompassllc.com/a-step-by-step-guide-to-migrating-aws-lambda-functions-to-ecmascript-modules-esm/
17. Move on to ESM-only - Anthony Fu, accessed December 21, 2025, https://antfu.me/posts/move-on-to-esm-only
18. Why I Finally Switched to ES Modules in Node.js — And How You Can Too - Medium, accessed December 21, 2025, https://medium.com/@ahmedrao609/why-i-finally-switched-to-es-modules-in-n

ode-js-and-how-you-can-too-b98cdbdae7ab
19. ES Modules __dirname Fix: Complete Guide (2025) | by Devin Rosario - Medium, accessed December 21, 2025, https://devin-rosario.medium.com/es-modules-dirname-fix-complete-guide-2025-b068a076712c
20. Alternative for __dirname in Node.js when using ES6 modules - Stack Overflow, accessed December 21, 2025, https://stackoverflow.com/questions/46745014/alternative-for-dirname-in-node-js-when-using-es6-modules
21. __dirname is back in Node.js with ES - Sonar, accessed December 21, 2025, https://www.sonarsource.com/blog/dirname-node-js-es-modules/
22. Top-level await in AWS Lamba with TypeScript - DEV Community, accessed December 21, 2025, https://dev.to/oieduardorabelo/top-level-await-in-aws-lamba-with-typescript-1bf0
23. Migrate from version 2.x to 3.x of the AWS SDK for JavaScript, accessed December 21, 2025, https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/migrating.html
24. Migrating from AWS SDK v2 to v3 in Node.js AWS Lambda Functions | Edstem Technologies, accessed December 21, 2025, https://www.edstem.com/blog/aws-sdk-v2-v3-in-node-lambda/
25. Optimize TypeScript Bundles for AWS Lambda with ESBuild - Cajun Code Monkey, accessed December 21, 2025, https://cajuncodemonkey.com/posts/bundles-for-aws-lambda-with-esbuild/
26. AWS Lambda@Edge Nodejs "Environment variables are not supported." - Stack Overflow, accessed December 21, 2025, https://stackoverflow.com/questions/54828808/aws-lambdaedge-nodejs-environment-variables-are-not-supported
27. Using await at the top level in ES modules - Matt Smith, accessed December 21, 2025, https://allthingssmitty.com/2025/06/16/using-await-at-the-top-level-in-es-modules/
28. Drop the layers, bundle up with ESBuild instead - AndMore Dev, accessed December 21, 2025, https://www.andmore.dev/blog/layerless-esbuild-lambda/
29. (aws-lambda-nodejs): Wrong @aws-sdk bundling when using format: OutputFormat.ESM and externalModules: [] · Issue #29310 · aws/aws-cdk - GitHub, accessed December 21, 2025, https://github.com/aws/aws-cdk/issues/29310
30. Node.js Lambda Package Optimization: Decrease Size and Increase Performance Using ES Modules | by Levi9 Serbia | levi niners articles | Medium, accessed December 21, 2025, https://medium.com/levi-niners-crafts/node-js-lambda-package-optimization-decrease-size-and-increase-performance-using-es-modules-100b392b7732
31. Top-level await vs lazy-loading to cache a result in a Nodejs Lambda : r/aws -

Reddit, accessed December 21, 2025, https://www.reddit.com/r/aws/comments/1j2n3ab/toplevel_await_vs_lazyloading_to_cache_a_result/
32. AWS Lambda@Edge Pricing, Features & Best Alternatives (2025) - srvrlss, accessed December 21, 2025, https://www.srvrlss.io/provider/amazon-lambda-edge/
33. Customize at the edge with Lambda@Edge - Amazon CloudFront - AWS Documentation, accessed December 21, 2025, https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/lambda-at-the-edge.html
34. AWS Lambda now supports ES Modules and Top-Level Await for Node.js 14, accessed December 21, 2025, https://aws.amazon.com/about-aws/whats-new/2022/01/aws-lambda-es-modules-top-level-await-node-js-14/
35. How do you like your AWS Lambda Concurrency – 2025 revision - NordHero, accessed December 21, 2025, https://www.nordhero.com/posts/aws-lambda-concurrency-2025revision/
36. AWS Lambda end of support for Node.js 18 and Node.js 16 extended to March 2026, accessed December 21, 2025, https://sinovi.uk/articles/aws-lambda-end-of-support-for-nodejs-18