Shipping Node.js packages in 2025

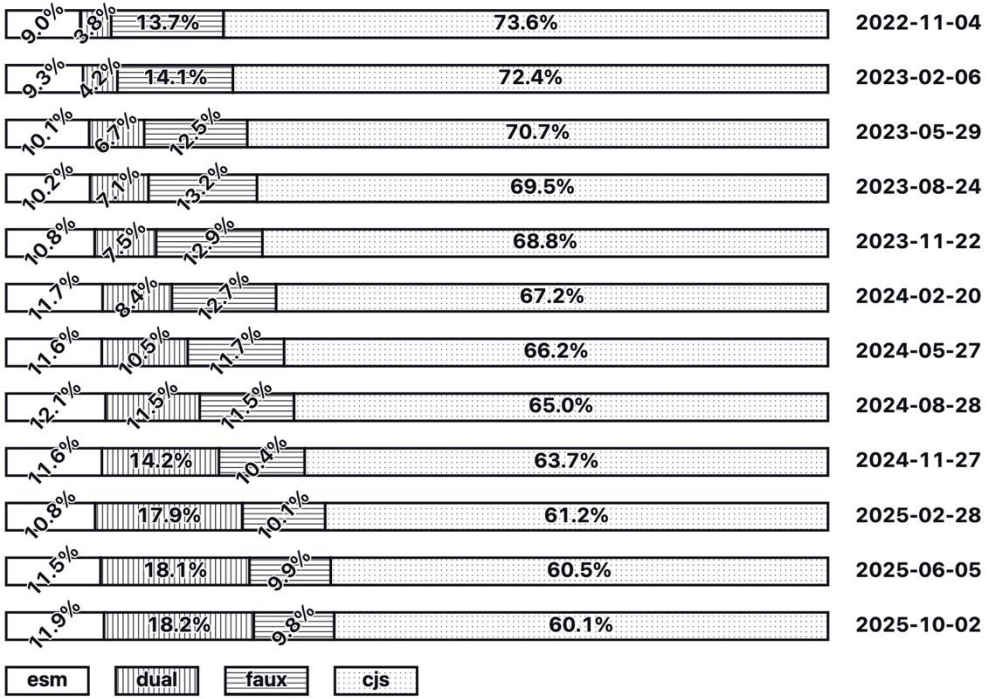
Joyee Cheung

About me

- Joyee Cheung
- Working at Igalia, sponsored by Bloomberg
- Member of Node.js Technical Steering Committee & V8 Committer
- Improving module loading, among other things

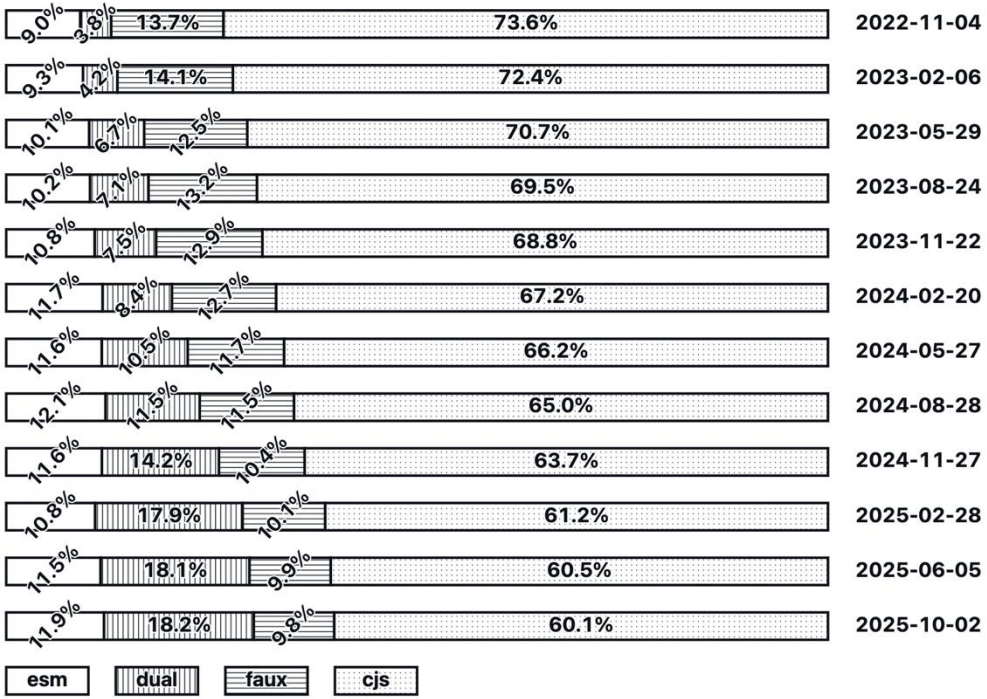
How Node.js packages have been shipped

State of high-impact packages in the npm registry



https://github.com/wooorm/npm-esm-vs-cjs/

State of high-impact packages in the npm registry



- ESM was stabilized in Node.js in 2020
- CommonJS is still the majority (slowly migrating)
- ESM is rising
- The most popular way to ship ESM is to ship dual

https://github.com/wooorm/npm-esm-vs-cjs/

Why not shipping ESM as-is?

It used to break compatibility with users running CommonJS (even if it's transpiled from ESM)

```
// ESM provider: logger.mjs
export default function log() {}

// CJS consumer cannot load ESM via require()
require('./logger.mjs'); // ❌ Throws ERR_REQUIRE_ESM!
// CJS consumer can load ESM via import(),
// but it returns a promise and only works in async code
import('./logger.mjs').then((namespace) => { namespace.log() });
```

Writing ESM != shipping ESM

- Packages, frameworks and tools transpile ESM to CommonJS: faux ESM

```
// Users write: handler_loaded_by_cjs_framework.ts
import { bar } from 'framework';
import { foo } from 'external_esm';
export default function handler() { return bar(foo()); }
```

Writing ESM != shipping ESM

- Packages, frameworks and tools transpile ESM to CommonJS: faux ESM
- Don't always work with real ESM
- Vicious loop: more deps/user code transpiled, more user code/deps stuck to CJS

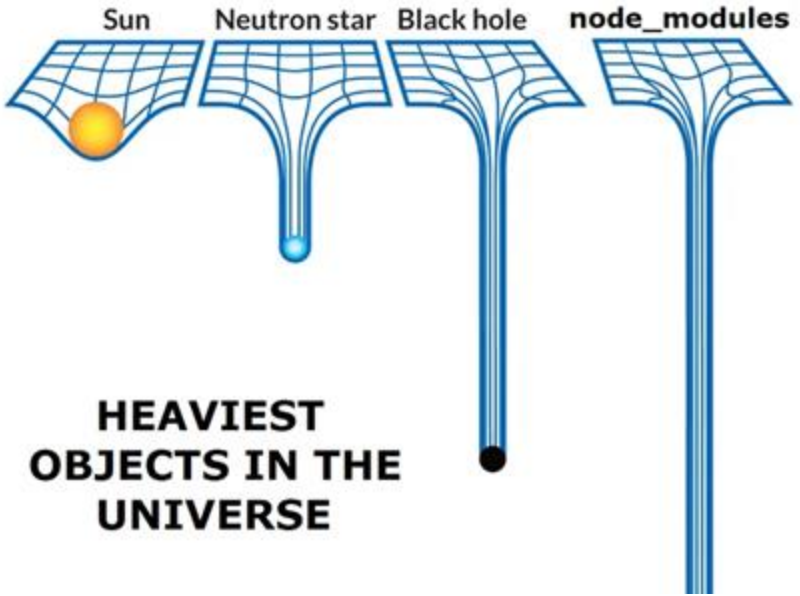
```
// Frameworks run: handler_loaded_by_cjs_framework.js
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
exports.default = handler;
const framework_1 = require("framework");
// Throws ERR_REQUIRE_ESM from code authored in ESM⁉️
const external_esm_1 = require("external_esm");
function handler() { return (0, external_esm_1.foo)(); }
```

Dual package

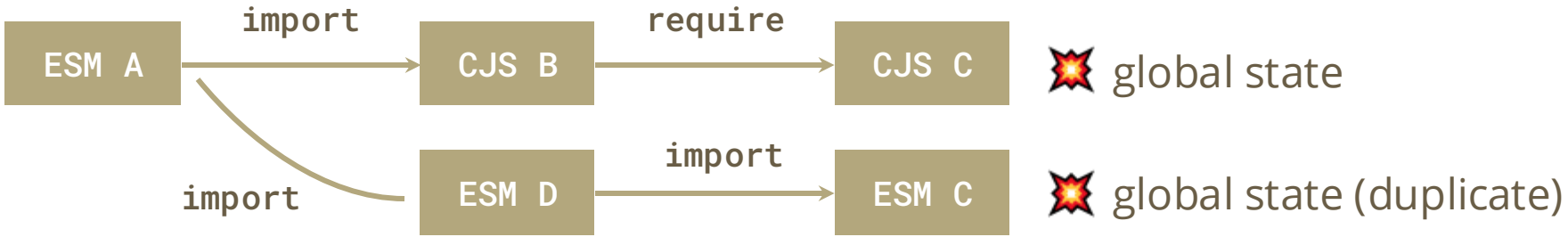
- Many packages ship two copies to support both consumers: supply ESM to ESM, CJS to CJS via conditional exports
- Doubled the size of node_modules...

Dual package

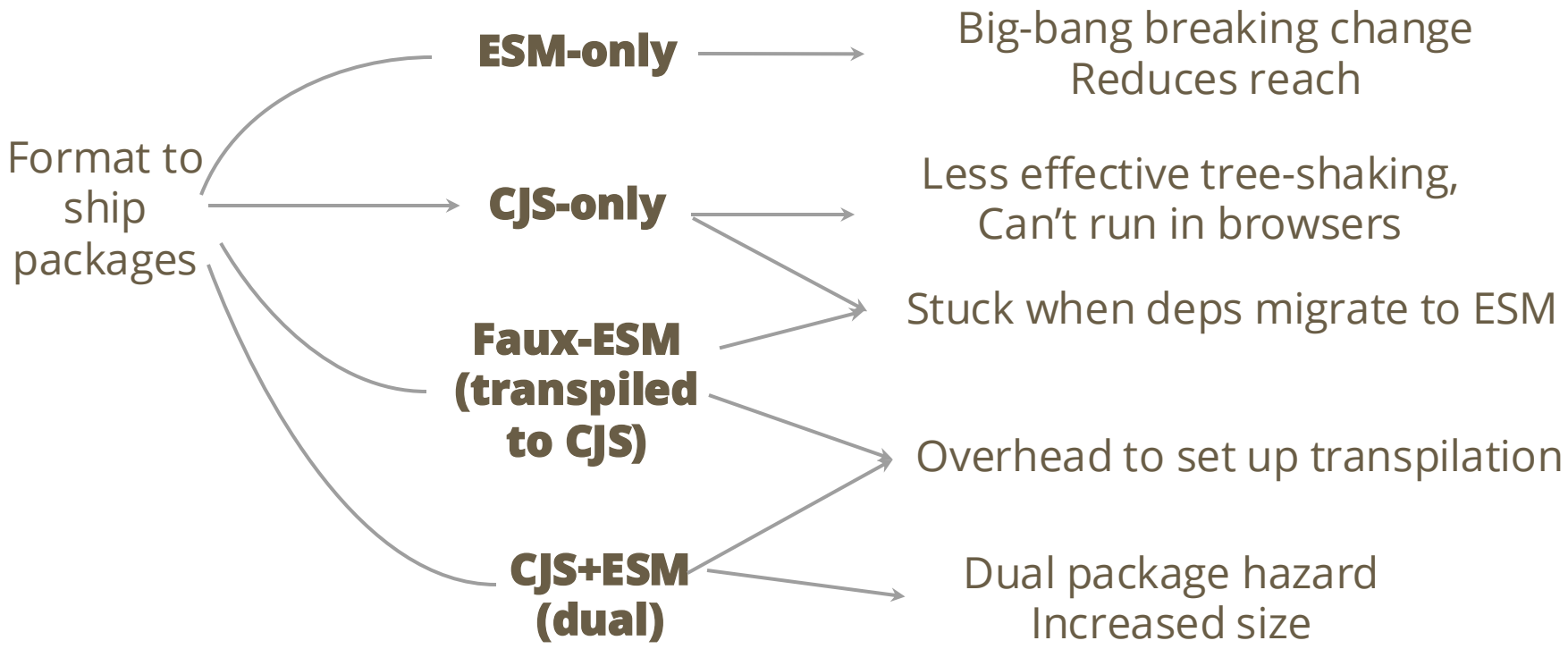
- Many packages ship two copies to support both consumers: supply ESM to ESM, CJS to CJS via conditional exports
- Doubled the size of node_modules...



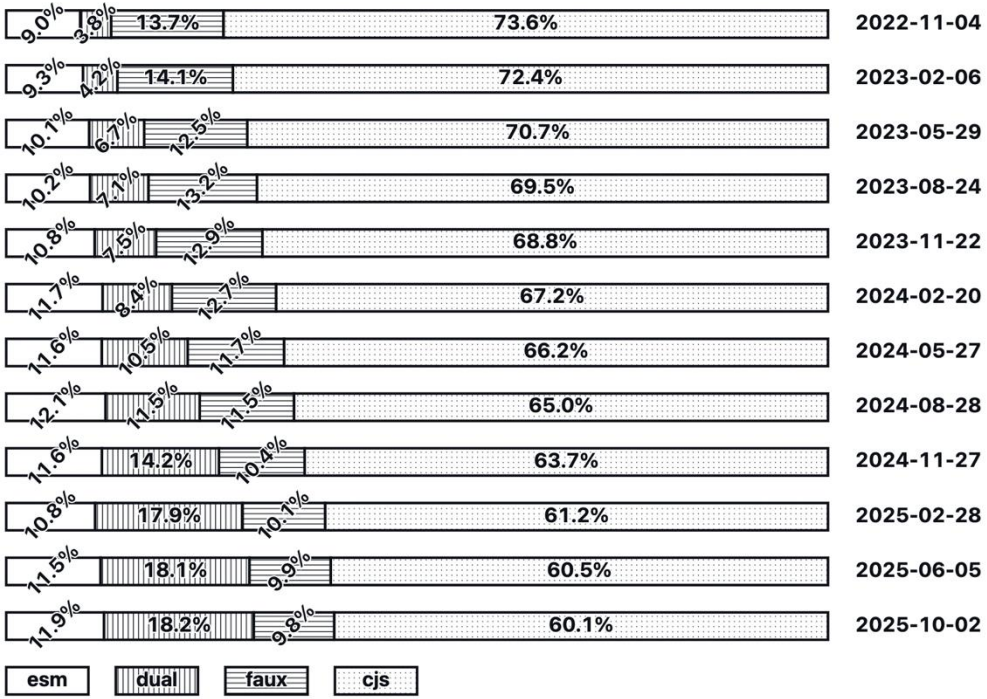
Dual package hazard



Pick your poison



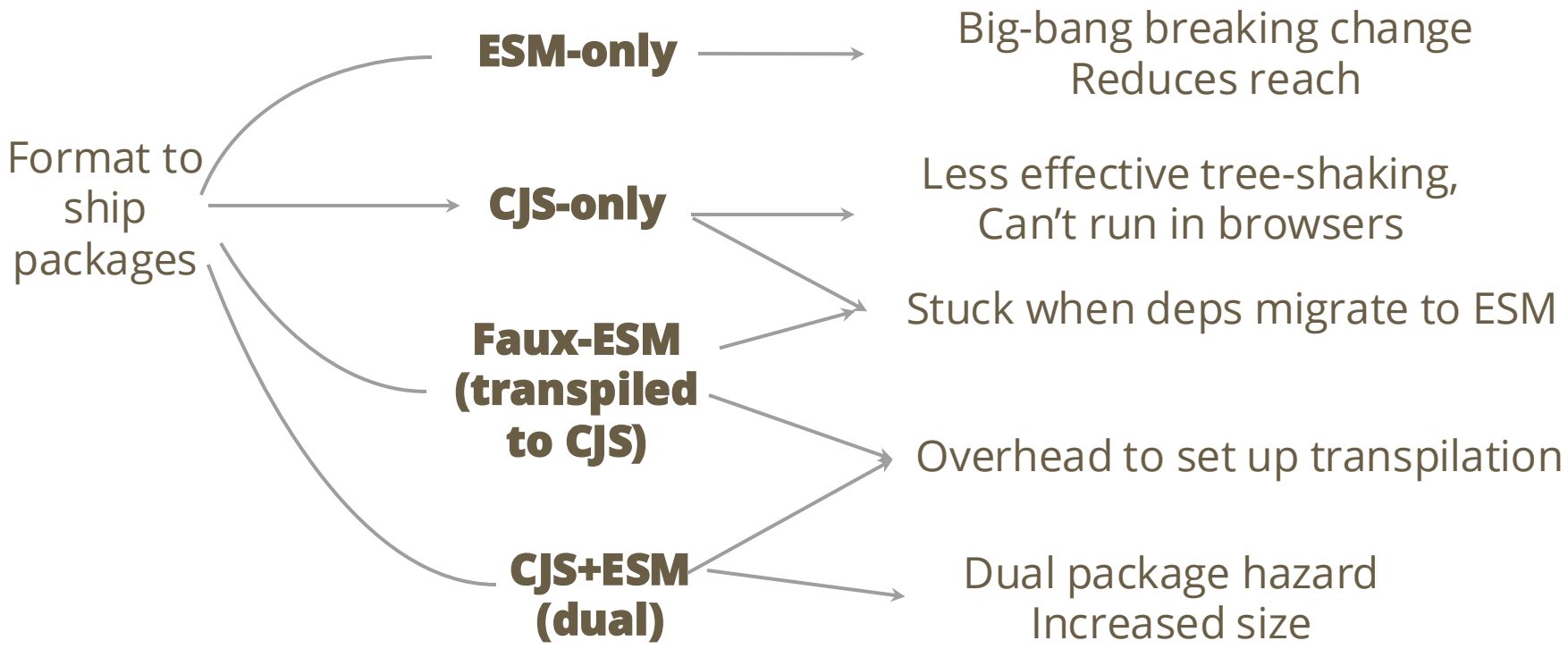
Impact of lack of require(esm)



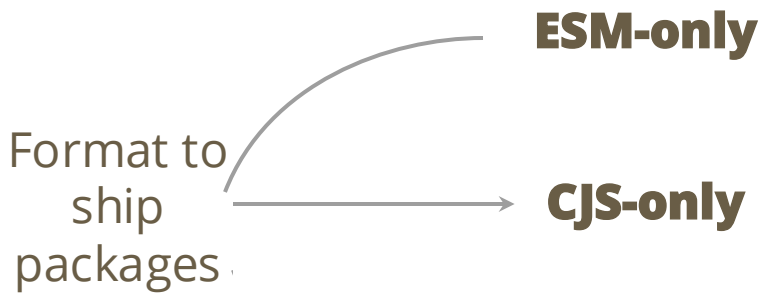
- This effectively made ESM a less desirable **execution** format
- Demotivated adoption of ESM

https://github.com/wooorm/npm-esm-vs-cjs/

What if ESM can be loaded by require()... 🤔

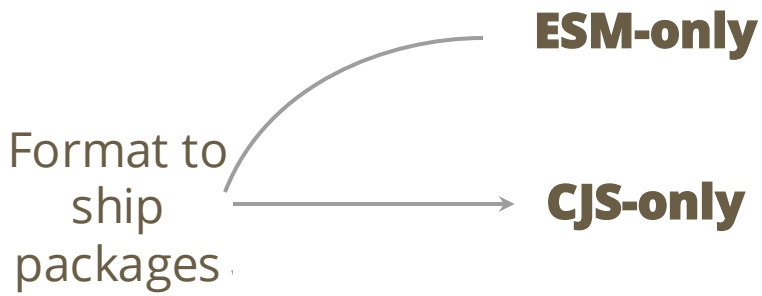


What if ESM can be loaded by require()... 🤔



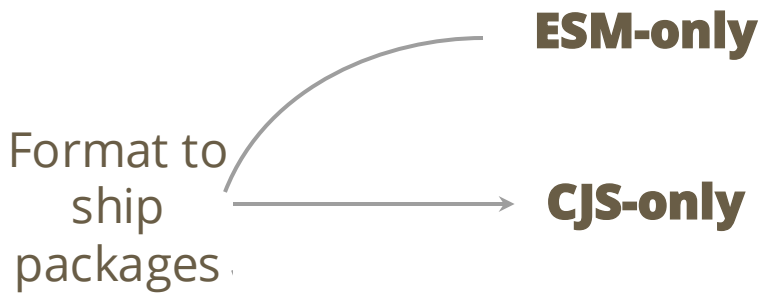
No longer necessary

What if ESM can be loaded by require()... 🤔



No more interop issues
Enables incremental upgrade

What if ESM can be loaded by require()... 🤔



No more vicious loops!

“ESM is async, but require() is sync”?

require

The CommonJS module `require` always treats the files it references as CommonJS.

Using `require` to load an ES module is not supported because ES modules have asynchronous execution. Instead, use `import()` to load an ES module from a CommonJS module.

- Node.js documentation stated this from v12 until v20.0.0
- There was an attempt to implement require(esm) in 2019 but implementation was unsafe (allowed running top-level await in sync code) and abandoned
- Limited debate among people who knew how it worked
- People not involved in the development (e.g. old me) just took what the documentation claimed and thought it was not possible by design

“ESM is async, but require() is sync”?

- Late 2023, fixing a leak and looking into V8’s part of ESM by chance
- What V8 and the spec had contradicted what the Node.js documentation claimed
- https://tc39.es/ecma262/#sec-innermoduleevaluation

```
for (Handle<SourceTextModule> m : exec_list) {
  if (m->has_toplevel_await()) {
    MAYBE_RETURN(ExecuteAsyncModule(isolate, m), Nothing<bool>());
  } else {
    if (!ExecuteModule(isolate, m).ToHandle(&unused_result)) {
      AsyncModuleExecutionRejected(isolate, m, exception);
    } else {
      // Resolve to undefined synchronously if the module itself nor its
      // dependencies contain top-level await. // ...
      JSPromise::Resolve(capability, isolate->factory()->undefined_value());
    }
  }
}
```

ESM without top-level await is synchronous

require(esm) without top-level await is pretty *straightforward* (at least theoratically)

```
// Pseudo code - this needs access to native V8 APIs.
function requireESM(specifier) {
  const linkedModule = fetchModuleGraphAndLinkSync(specifier);
  if (linkedModule.hasTopLevelAwaitInGraph()) {
    throw new ERR_REQUIRE_ASYNC_MODULE;
  }
  const promise = linkedModule.evaluate();
  // This is guaranteed by the ECMAScript specification.
  assert.strictEqual(getPromiseState(promise), 'fulfilled');
  assert.strictEqual(unwrapPromise(promise), undefined);
  // The namespace is guaranteed to be be fully evaluated at this point if the
  // module graph contains no top-level await.
  return linkedModule.getNamespace();
}
```

Up to Node.js to make it synchronous

ESM without top-level await is synchronous

require(esm) without top-level await is also not that unorthodox on the Web

```
// Pseudo code - this needs access to native V8 APIs.
function requireESM(specifier) {
  const linkedModule = fetchModuleGraphAndLinkSync(specifier);
  if (linkedModule.hasTopLevelAwaitInGraph()) {
    throw new ERR_REQUIRE_ASYNC_MODULE;
  }
  const promise = linkedModule.evaluate();
  // This is guaranteed by the ECMAScript specification.
  assert.strictEqual(getPromiseState(promise), 'fulfilled');
  assert.strictEqual(unwrapPromise(promise), undefined);
  // The namespace is guaranteed to be be fully evaluated at this point if the
  // module graph contains no top-level await.
  return linkedModule.getNamespace();
}
```

V8 already implemented it because **service workers** also have this semantics

ESM without top-level await is synchronous

top-level await is mostly meant for code that only import other modules, not code supposed to be imported by external code controlled by others

```
// Pseudo code - this needs access to native V8 APIs.
function requireESM(specifier) {
  const linkedModule = fetchModuleGraphAndLinkSync(specifier);
  if (linkedModule.hasTopLevelAwaitInGraph()) {
    throw new ERR_REQUIRE_ASYNC_MODULE;
  }
  const promise = linkedModule.evaluate();
  // This is guaranteed by the ECMAScript specification.
  assert.strictEqual(getPromiseState(promise), 'fulfilled');
  assert.strictEqual(unwrapPromise(promise), undefined);
  // The namespace is guaranteed to be be fully evaluated at this point if the
  // module graph contains no top-level await.
  return linkedModule.getNamespace();
}
```

Only 6/5000 top high-impact packages actually have TLA, 5 of them only added it during migration to ESM and have alternatives

ESM without top-level await is synchronous

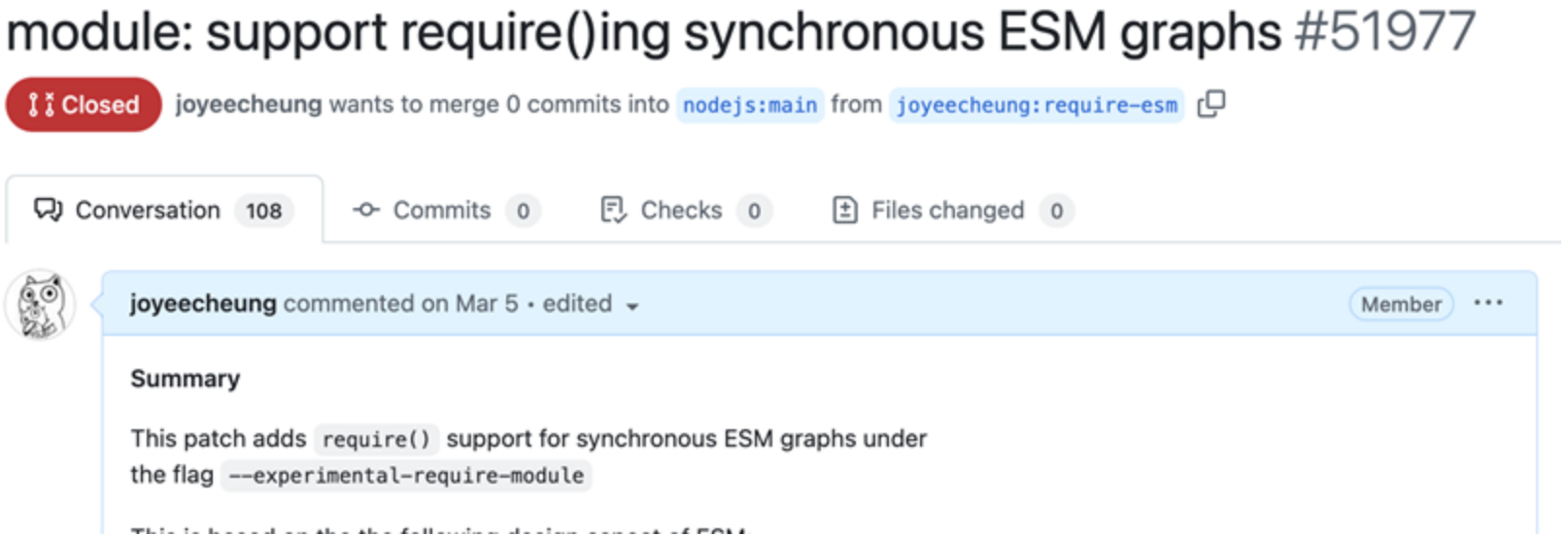
require(esm) without top-level await has some theoretical guarantees

```
// Pseudo code - this needs access to native V8 APIs.
function requireESM(specifier) {
  const linkedModule = fetchModuleGraphAndLinkSync(specifier);
  if (linkedModule.hasTopLevelAwaitInGraph()) {
    throw new ERR_REQUIRE_ASYNC_MODULE;
  }
  const promise = linkedModule.evaluate();
  // This is guaranteed by the ECMAScript specification.
  assert.strictEqual(getPromiseState(promise), 'fulfilled');
  assert.strictEqual(unwrapPromise(promise), undefined);
  // The namespace is guaranteed to be be fully evaluated at this point if the
  // module graph contains no top-level await.
  return linkedModule.getNamespace();
}
```

Guaranteed by the specification

Making require(esm) happen

- (Re)started implementation in 2024, sponsored by Bloomberg
- Experimental release in 22.0.0
- Unflagged: ^20.19.0 || >=22.12.0



require(esm) is here!

- All active LTS now supports require(esm)!
- Packages that do not support EOL Node.js versions can count on it now
- Many popular packages have started to drop dual and ship ESM-only after require(esm) was unflagged
  - Vite
  - Yargs
  - Babel
  - Storybook
  - Unjs packages..
  - Various tinylibs..
  - Various eslint plugins (h/t e18e)

LLMs still didn't get the notice

I have a Node.js package written in ESM. I want both Common JS and ESM users to be able to load my package. How should I publish it?

Thought for 17s

Great goal — you can publish a single package that works for both `require()` and `import`. The modern, reliable way is:

- build **two artifacts** (ESM + CJS),
- expose them via **conditional exports** in `package.json`,
- ship **one set of types**,
- and (optionally) keep a legacy `main` for very old tooling.

Below is a battle-tested template you can copy.

The 2025 update

Preparing ESM code shipped in dual/faux format

1. No top-level await in the module code shared to external users - normally, dual/faux already don’t have it

Preparing ESM code shipped in dual/faux format

1. No top-level await in the module code shared to external users - normally, dual/faux already don't have it
2. Check extensionless exports

```
// ❌ These are faux-ESM only - Node.js ESM does not support extensionless import
// It only worked when the it was transpiled to require()
import { foo } from './lib/index'; // Transpiled to require('./lib/index')
import { bar } from './lib/dir'; // Transpiled to require('./lib/dir')

// 🟢 Update it to full path
import { foo } from './lib/index.js';
import { bar } from './lib/dir/index.js'
```

Preparing ESM code shipped in dual/faux format

TypeScript source code commonly omits the extension for multi-purpose (build time/run time) resolutions

```
// Build tools resolves to lib/index.ts for type checking at build time
import { foo } from './lib/index';
// and transpiles to
const { foo } = require('lib/index'); // only resolves to .js in CommonJS require()

// ❌ ERR_MODULE_NOT_FOUND due to missing extension if transpiled to ESM
import { foo } from './lib/index';
```

Preparing ESM code shipped in dual/faux format

To tanspile TypeScript to JS-in-JS for shipping, use `rewriteRelativeImportExtensions: true` (requires TS >=5.7)

```
// Used to be transpiled to require('./lib/index'), worked with .ts in tools,
// .js at run time
import { foo } from './lib/index';

// => Extend to full .ts in the source code
import { foo } from './lib/index.ts';

// rewriteRelativeImportExtensions rewrites it to .js for output to be run
import { foo } from './lib/index.js';
```

From Dual to ESM-only: updating package.json

Variant 1: require + import (⚠ prone to dual-package hazard, but still popular)

```
{
  "type": "module",
  "exports": {
    ".": {
      // Supply CommonJS version to require() - need to make sure that
      // states can be duplicated or shared states live in the same graph
      "require": "./dist/index.cjs",
      // Supply ESM version to import and others
      "import": "./index.js",
      "default": "./index.js"
    }
  }
}
```

From Dual to ESM-only: updating package.json

Variant 1: ESM everywhere, remove CommonJS distribution, update engines field

```
{
  "type": "module",
  "exports": {
    // No more splits!
    ".": "./index.js",
  },
  // In Node.js, drop support for versions that do not support require(esm)
  // by default (may require bumping major version for the package).
  "engines": {
    "node": "^20.19.0 || >= 22.12.0"
  }
}
```

From Dual to ESM-only: updating package.json

Variant 2: CommonJS on Node.js, ESM in other environment

```
{
  "type": "module",
  "exports": {
    ".": {
      // In Node.js, always provide transpiled CommonJS to avoid
      // dual-package hazard
      "node": "./dist/index.cjs",
      // On any other environment, use the ESM version (e.g. for
      // better tree-shaking when bundled for browsers
      "default": "./index.js"
    }
  }
}
```

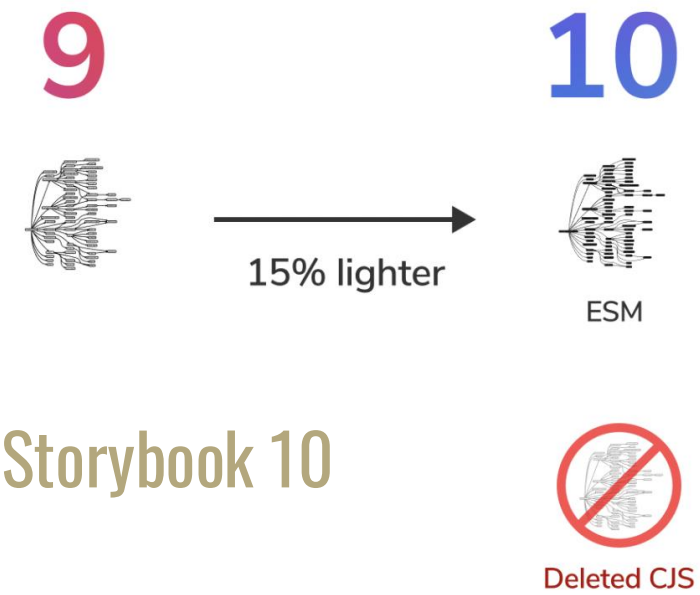
From Dual to ESM-only: updating package.json

Variant 2: ESM everywhere, remove CommonJS distribution, update engines field

```
{
  "type": "module",
  "exports": {
    // No more splits!
    ".": "./index.js",
  },
  // In Node.js, drop support for versions that do not support require(esm)
  // by default (may require bumping major version for the package).
  "engines": {
    "node": "^20.19.0 || >= 22.12.0"
  }
}
```

From Dual to ESM-only: dropping CommonJS

- Other package-specific cleanup, if necessary
- Delete the transpile-to-CommonJS build step
- No more dist-cjs/... or dist/cjs/... or dist/**.cjs copies



From Faux-ESM to ESM-only: updating package.json

Most common variant: “module” for bundlers, “main” for Node.js

```
{
  // Most faux-ESM packages don’t have “type” field and rely on the
  // “commonjs” default for .js files loaded in Node.js

  // The “module” export condition may work similarly, but top-level
  // “module” field is popular for supporting legacy bundlers
  "module": "./dist-es/index.js",
  // Using “main” instead of “exports” for legacy support
  "main": "./dist-cjs/index.js"
}
```

From Faux-ESM to ESM-only: dropping CommonJS

Update all CommonJS pointers to ESM, remove CommonJS distribution, update engines field

```
{
  // Add “type”: “module” for Node.js to load .js as ESM
  "type": "module",
  // Same as before
  "module": "./dist-es/index.js",
  // Update “main” to point to ESM now
  "main": "./dist-es/index.js",
  // In Node.js, drop support for versions that do not support require(esm)
  // by default (may require bumping major version for the package).
  "engines": {
    "node": "^20.19.0 || >= 22.12.0"
  }
}
```

From Faux-ESM to ESM-only: dropping CommonJS

…or, it’s time to drop legacy support and just use “exports”!

```
{
  // Add “type”: “module” for Node.js to load .js as ESM
  "type": "module",
  // “exports” helps hiding internal files (may require bumping major version)
  ”exports": {
    ”.": ”./dist-es/index.js"
  },
  // In Node.js, drop support for versions that do not support require(esm)
  // by default (may require bumping major version for the package).
  "engines": {
    "node": "^20.19.0 || >= 22.12.0"
  }
}
```

From CommonJS to ESM: updating package.json

```
my-package/
 ├── index.js // CJS -> ESM syntax
 ├── utils.cjs // Files that have to stay in CommonJS
 └── package.json  // add "type": "module"
```

```
{
  // Add “type”: “module” for Node.js to load .js as ESM,
  // and leave files that must be CommonJS for some reason to use .cjs
  "type": "module"
}
```

From CommonJS to ESM: updating package.json

```
my-package/
  ├── index.mjs // CJS -> ESM syntax
  ├── utils.js // Files pending to be migrated
  └── package.json  // use "type": ”commonjs" and flip when it’s done
```

```
{
  // Or if you need to do it gradually, convert on a file-by-file basis as .mjs
  // or configure the tools you use to emit them as .mjs (e.g. mts)
  "type": ”commonjs"
}
```

From CommonJS to ESM: updating imports

```
// CommonJS
const { log } = require('log');
const fs = require('fs');
const { foo } = require('./lib');

// ESM
import { log } from 'log';
import fs from 'node:fs'; // Built-ins must be prefixed with node:
import { foo } from './lib/index.js'; // Must be full paths
```

From CommonJS to ESM: special context variables

```
if (require.main === module) {
  // It's run as entrypoint
}
// Equivalent to
if (import.meta.main) { // Requires Node.js ^22.18.0 || >= 24.2.0
  // It's run as entrypoint
}
// Requires Node.js ^20.11.0 || >= 21.2.0
const __filename = import.meta.filename;
const __dirname = import.meta.dirname;
```

From CommonJS to ESM: named exports only

```
// If there are only named exports..
exports.func = function func() {}
const foo = { };
exports.bar = foo;
exports.baz = 'some string’;

// converts to
export function func() { }
export { foo as bar };
export const baz = 'some string';
```

From CommonJS to ESM: default exports only

```
// If there are only default exports...
module.exports = class Logger {};
// converts to
export default class Logger{};

// Or
module.exports = object;
// converts to
export { object as default };
```

From CommonJS to ESM: what if we have both?

```
// When the package is provided through CJS
module.exports = class Logger {};
// Assign static properties explicitly for CJS named import detection from ESM
module.exports.log = function log() {}

// ESM user gets..
import { log } from 'log';
import Logger from 'log';
// In ESM, default export is placed separately from named exports 🤔
await import('log'); // { default: Logger, log: log }

// CJS user gets..
const { log } = require('log');
const Logger = require('log');
```

From CommonJS to ESM: watch out for default exports

```
// If the package migrates to ESM..
export default class Logger{};
export function log() { }
Logger.log = log;

// ESM user gets..
import { log } from 'log';
import Logger from 'log';
// In ESM, default export is placed separately from named exports 🤔
await import('log'); // { default: Logger, log: log }

// CJS user gets..
const { log } = require('log');
const Logger = require('log'); // ❌ Oops, it's also { default: Logger, log: log }!
const Logger = require('log').default; // Have to unwrap it from .default..
```

From CommonJS to ESM: watch out for default exports

Use the special “module.exports” string named export to customize result returned by require(esm)

```
// Migrate to ESM
export default class Logger{};
export function log() { }
Logger.log = log;
export { Logger as 'module.exports' }; // Customize for require(esm) in Node.js

// ESM user gets..
import { log } from 'log';
import Logger from 'log';

// CJS user gets the same as before
const { log } = require('log');
const Logger = require('log'); // Returns 'module.exports' string export if it exists
```

What if I need sync dynamic loading at the top-level?

```
if (typeof module === 'object' && module.exports) {
  // If it's Node.js CommonJS, do some extra tuning
  const os = require('node:os');
}
```

What if I need sync dynamic loading at the top-level?

```
if (typeof module === 'object' && module.exports) {
  // If it's Node.js CommonJS, do some extra tuning
  const os = require('node:os');
}

// When converted into ESM, one might reach for TLA + dynamic import, then it
// becomes asynchronous and gets complicated...
try {
  const os = await import('node:os'); // Do some tuning with OS info
} catch { /* Fallback */ }
```

What if I need sync dynamic loading at the top-level?

```
if (typeof module === 'object' && module.exports) {
  // If it's Node.js CommonJS, do some extra tuning
  const os = require('node:os');
}

// When converted into ESM, one might reach for TLA + dynamic import, then it
// becomes asynchronous and gets complicated...
try {
  const os = await import('node:os'); // Do some tuning with OS info
} catch { /* Fallback */ }

// Use process.getBuiltinModule() instead to have dynamic synchronous loading in ESM
if (globalThis.process?.getBuiltinModule) {
  const os = globalThis.process.getBuiltinModule('os');
// Do some tuning with OS info
}
```

What if I need sync dynamic loading at the top-level?

```
// For non-built-ins, createRequire() gives you working alternative to
// https://github.com/tc39/proposal-import-sync
import { createRequire } from 'node:module';
const require = createRequire(import.meta.url);

const foo = condition ?
  require('./conditionA.js') : require('./conditionB.js');

// To feature-detect require(esm) and fall back accordingly
import process from 'node:process';
if (!process.features?.require_module) {
  // require(esm) is not available, do something less fancy
}
```

WIP official guide

https://github.com/nodejs/package-examples/

Thanks!