Node.js Survey open ended answers

Q1: Which of the following best reflects your role regarding Node.js? - Other

- Architect
- CTO
- enthusiast
- Front end developer
- Front-end developer
- Node js entusiast
- Only interested in further improvements to the project; .NET engineer
- Other (please specify)
- Software Architect

Q4: What type of organization do you work with? - Other

- 100devs
- All kind of industry, scale up
- Banking
- Banking industry
- biotech
- City
- Consulting company
- Criminal Justice System
- Currently Unemployed
- E-com
- enterprise
- Fashion/Tech Company
- Ferry transport
- Finance
- Finance
- Finance company
- Financial
- Financial company
- Financial Company
- Financial Institution
- For me; and open source
- freelance/NGO

- Furniture Retailer
- furniture, energy
- Game studio
- GitHub Organizations
- Government
- Government
- Green Energy
- health
- Health
- Health care
- Health care
- Healthcare
- Healthcare
- Healthcare
- Independent consultant.
- Individual and Startup company
- Industry
- Insurance
- Insurance company
- International Newspaper
- Logistics company
- manufacturing
- Manufacturing enterprise
- Marketing company
- Media
- Media company
- Media company
- Medical company
- Movie studio
- Multi Nationals
- Newspaper
- Newspaper
- none
- Oil sector
- Open Source
- Other (please specify)
- Outsource company
- Outsource dev company
- outsource+outstaff
- Outsourcing company
- Pharmaceutical company
- Product based company
- Publishing company
- Retail

- Retail
- Retail
- Retail
- Safety company
- Sales
- Self employed contractor
- Self-employed
- Service company
- Software house
- Software Incubator
- some case individual, but some case i had team to work..
- Space Company
- Still in search of an opportunity
- student loan finance co
- Technilogy consultant
- Telecom
- Telecom
- Telecommunications company
- Undergraduate students and software hobbyist
- Whosale/Transportation/Supply Chain

Q5: Are you involved in any Node.js community Working Groups? If yes, which one?

This question was not well understood from the responses and there is no real good data in these responses

Q6: What is your primary use case for Node.js? - Other

- Automation
- Automation QA
- Azure Functions
- Bots
- Bots, scripts, parsers...
- Build systems
- Cli tools
- Cli tools

- cli tools, data migrations
- CLI's & Tooling
- CLI/utilities/packages
- CLIs
- Developer tooling
- Developer tooling (CLI tools)
- developing library for end to end testing
- DevX tooling
- I don't neccessarily use it for custom backends (APIs), but because frameworks like Next.js uses it for SSR.
- library / framework development
- linting and legacy build steps
- Nim lang wrapping and extensions
- Quick scripts
- scripting
- Serverless, (aws lambda, gc functions)
- Small tools
- Software engineer in test
- Solving ProjectEuler.net problems
- Some command line applications
- Standalone Products
- Still learning
- Test Automation
- Test automation
- Testing
- Testing
- Testing (jest)
- Tooling
- Tooling
- Tooling, CLIs, etc
- Unit testing
- vite, webpack, code generation, npm
- Web scraping
- WebRTC, NodeOS

Q7: Which package manager do you use? - Other

NOTE: we missed npm in the list to start which explains the large number of npm write ins

- Npm
- both npm and yarn
- Checked in deps
- latest

- nimble
- Nmp
- npm
- npm
- npm
- Npm
- Npm
- npm
- npm
- npm
- npm
- Npm
- Npm
- Npm
- npm
- npm
- Npm
- Npm
- npm
- Npm
- npm
- npm
- Npm
- npm
- npm
- npm
- Npm
- npm
- npm
- npm
- Npm
- Npm
- npm
- npm
- NPM
- NPM
- npm
- npm
- npm
- npm
- npm
- npm
- Npm
- npm

- npm
- NPM
- npm
- npm
- npm
- npm
- npm
- Npm
- Npm
- Npm
- npm
- npm
- npm
- npm
- npm
- npm
- Npm
- npm
- npm
- npm
- npm
- Npm
- npm
- npm
- npm
- npm
- npm
- npm
- npm
- Npm
- Npm
- Npm
- npm & yarn
- npm and pnpm
- npm and pnpm
- npm and yarn
- npm and yarn
- npm, pnpm, and yarn
- Npm, pnpm, yarn
- npm, yarn
- npm, yarn, and pnpm. this question is dumb, people work on more than 1 project.
- npm, yarn, pnpm
- Npm/yarn
- npn

- Other (please specify)
- pnpm,npm,yarn
- regular npm
- yarn for some legacy packags, npm for most cases, pnpm for new greenfield projects

Q9: Which version manager do you use? - Other

- adsf
- apt
- apt
- asdf
- asdf
- asdf
- asdf
- asdf
- asdf
- asdf
- asdf
- Asdf
- asdf
- asdf
- asdf
- asdf
- asdf
- asdf
- asdf
- asdf
- ASDF
- brew
- brew
- Brew
- Containers and Virtual Machines
- corepack
- Corepack
- custom
- Do not use
- Docker
- Docker
- Docker images
- docker images specify the version
- Docker node:* images
- Don't use

- Don't use at all
- Don’t know
- Don’t use
- Fast Node manager
- fem
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- Fnm
- fnm
- Fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- Fnm
- fnm
- fnm
- fnm
- fnm
- Fnm
- fnm
- fnm
- fnm
- fnm
- Fnm
- fnm
- fnm
- fnm
- Fnm
- FNM
- fnm
- fnm
- fnm
- fnm

- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- fnm
- Fnm
- fnm
- Git
- Git
- Homebrew
- https://github.com/jdxcode/rtx
- I don't
- I don't know
- I don't use a version manager as long as nodejs doesn't embed a version manager like NPM
- I don't use a version manager.
- I dont use it, instead I use alter native on Ubuntu or docker with pre installed version
- I prefer not to use one
- I use pnpm to manage multiple node versions
- manually setup in the docker container
- my own bash scripts
- N/A
- n/a
- N/A
- Na
- Nill
- nimble
- nix
- No
- no
- no i install from my dixtro package manager: dnf
- nodeenv

- nodenv
- nodenv
- Nodenv
- nodenv
- nodenv
- nodenv
- None
- None
- None
- none
- None
- none
- None
- none
- None
- None
- None
- None
- none
- None
- None
- None
- None
- none
- None
- None
- none
- None
- None
- None
- None
- None
- None
- None
- None
- none
- none
- None
- None
- none
- None
- None
- None
- none

- none
- None
- none
- None (over docker)
- None of them
- None of these
- None. Always latest lts
- None. I’m really old school with server management
- Not using
- Not using one
- Npm
- npm version
- nvm || laragon
- nvm and pnpm, depending on project
- nvm for windows (same name different project iirc)
- nvm for windows, nodebrew, volta
- nvm-windows
- nvm, but moved to asdf
- nvm.fish
- Only use latest version (operating system)
- Other (please specify)
- pnpm
- pnpm
- pnpm
- Pnpm
- pnpm
- pnpm
- pnpm (it does node version management)
- pnpm env
- PPA / Docker
- rtx
- rtx
- Scoop
- Self buit script
- systemd portable services, Docker
- Unknown
- V

Q10: What operating system is your local development environment? - Other

- All of the above
- Cloud Functions

- ibm i
- Other (please specify)
- Windows (WSL)
- windows with wsl2
- Wsl
- WSL
- WSL
- Wsl
- WSL
- WSL
- WSL
- WSL
- WSL
- WSL - Debian
- WSL 2
- wsl debian
- WSL/Linux
- Wsl2
- wsl2
- WSL2
- WSL2
- WSL2 on Windows 11

Q11: What operating system is your production environment? - Other

- Alpine inside container
- Any that’s used by end user
- AWS Lambda
- AWS Lambda
- Azure
- Azure App Services
- Cloud (Qoddi)
- Cloud app engines.
- Cloud Functions
- Depending on deployment target
- Docker
- Docker (alpine linux)
- Google cloudrun
- ibm i
- IBM i
- Kubernetes (Azure)
- Other (please specify)

- serverless
- Static Sites
- Supporting multiple envs
- Vercel
- Wsl
- A

Q12: What architecture is your production environment? - Other

- ?
- ?
- Any that’s used by end user
- Apple silicon
- AWS Lambda
- Byv
- Cloud
- Depends on deployment target
- Does not know
- don't know
- don't know
- don't know
- Don't know and don't care
- Don’t know
- dont know
- Dont know
- Dunno, something that is on cloud
- GCS E2 vCPU
- i am not sure
- I don't know
- I don't know
- I don't know.
- idk
- idk
- Na
- not sure
- Other (please specify)
- serverless
- Serverless
- Serverless - don’t care
- Serverless, so don't know
- Supporting multiple envs
- Unknown
- Various

- x32
- x86

Q13: How do you get your Node.js executables? - Other

- asdf
- asdf
- asdf
- Bazel
- Brew for macos
- Chocolatey
- Corepack
- Docker
- Docker
- Docker
- Docker
- docker
- Docker
- Docker
- Docker
- Docker
- Docker
- Docker
- docker container
- Docker dev environment
- Docker files
- Docker hub
- Docker image
- Docker image
- Docker image
- Docker image
- docker image
- Docker image
- docker image
- Docker image
- Docker image (e.g. node:lts-alpine)
- Docker image with node.js already installed
- Docker images
- Docker images
- Docker images
- Docker images
- Docker in Prod

- Docker pull
- Docker Pull
- Dockhub
- Fnm
- from official Node Docker image and lightweight custom image based on executables from official
- ibm rpm
- Installed in Dockerfile
- NodeSource
- Nodesource, also whatever GCP has built in
- Official docker image
- official docker images
- Official docker images
- Official Docker images (deployments)
- Official Docker node image
- Other (please specify)
- Pnpm env use
- Pre-built docker images
- Prod env: Docker images
- Scoop
- Using docker
- Using the official Docker Image
- Version manager

Q14: Which of the below technical priorities are important to you? - Other

- a heads of of what the next release will hold, as of now, on the day of node 20 release, it's hard to find info what and when we're going to get
- ABILITY TO EMBED AND BUNDLE NODE.JS RUNTIME 1000%
- All the above
- An event model that works, since existing one just is too complex and buggy
- Backward compatibility with new updates
- C/C++ interop docs and examples, Nim lang interop
- Cloud support
- Compiler which can turn JS to compiled typed language for performance with Rust
- ESM is borked
- Fast and extensible testing module
- HTTP/3
- I want websockets and http3
- Improved performance: image size, start time, execution time

- innovation at a pace that matches real world technology (http3, vulkan, multithreading). node is a js interface to the operating system, its not fulfilling that role in many ways. Also, replace libuv because its slow.
- Long term stability. 5 years would be great
- migrating everything to ESM
- Native TypeScript support
- Numeric precision
- Performance
- providing an end to end solution for projects: built-in testing, linting and bundling capabilities to not rely on third party libraries
- Strictness help incredible, so to encourage typescript

Q15: Please check which of the following are important to you when using Node.js. - Other

- aligning with other web standards to have isomorphic implementation for both the front-end and the back-end
- Android arch support
- async iterator compatible types
- Backward compatibility
- Backwards compatability
- Better CPU intensive task performance
- Better debugging / performance capturing capabilities (such as per-line hotspot info)
- Better dynamic template rendering
- Better performance against competitors (Deno, Bun)
- Better performance; make it harder to write bad code
- Better TypeScript support
- Better typescript support, maybe with paths and baseUrl resolving
- Browser JavaScript runtime parity (same APIs)
- build-in websockets
- Built in hot module reload (not restart) with user space hooks api
- Built-in TypeScript support, and great performances
- clearly specified APIs
- compile to binary, better errors traces descriptions
- Concrete Types for arguments and return types in docs, C API docs
- Ease of use, ease of install
- Esm esm esm esm esm (sorry)
- Extensions with other languages (C/C++/Rust)
- Faster webpack :)
- Full compatibility with Web API's (like Deno)
- Full support for es6 modules
- Full TypeScript support directly in Node.js

- http/3
- HTTP/3
- http/3
- HTTP/3
- Http/3 and websocket
- HTTP3
- http3
- HTTP3
- i dont know
- Improved performances, improved v8 dumps and snapshots. Updated docs to work with streams, servers, timers in a modern promised-based way.
- Legacy support. E.g., Avoiding the hell I'm in with all my Bionic servers with Node v18 builds not working there and everything older going EOL
- Less features, more stability
- Limitation for Enterprise Software development
- Logger Viewer
- Mature 3rd party libraries for backend
- much security
- Native TypeScript
- Native TypeScript Support
- numeric type with arbitrary precision
- Official GPU Wrappers
- Only 32bit Integer support bitwise operator, no 64bit available. I wonder if any more int type can be extend beyond BigInt. Make js being strong type optional to enhance the critical performance. Or maybe this thoughts is not proper here...
- Other (please specify)
- performance
- Performance
- Performance
- Performance
- Performance
- Performance
- Performance
- Performance
- Performance snapshots
- Performances !
- Problems with going back to old projects, install issues, conflicting CJS/ESM modules
- Profiling
- proper tail calls
- Quality and metrics
- Respect more WebApi
- Run nodejs inside nodejs
- Running TypeScript without explicit transpilation step
- Simplify the API and JS features. Remove unnecessary things. Make it easy to learn.

- Speed
- Speed and performance
- startup snapshot
- Startup time
- Strict static types in runtime
- Testing
- Tree shakable single executable application, reduce the binary size; native typescript support
- Types
- Types
- Types (https://github.com/tc39/proposal-type-annotations)
- typescript
- Typescript
- Typescript
- TypeScript
- Typescript first class
- Typescript support
- Typescript support out of the box
- vulkan and window launching would be awesome. webgpu sucks, just use vulkan please
- Websocket
- WebSocket
- WebSocket
- websocket
- Websocket server in stdlib
- WebSockets
- Websockets
- Websockets, HTTP/3,Memory/Processes profiling
- WINTER-CG APIs

Q18 - If you wish to use ESM in an existing application, what have been the pain points or blockers preventing you from doing so (if any)?

- __dirname
- __dirname and other default functions not found in esm. we make workaround with url module.
- __dirname and require cache
- __dirname, __filename
- __fileName __dirName etc. globals available and used in cjs files should have an extra step to get them in esm .mjs files
- - Default imports to CommonJS work differently from every other tools - CommonJS is still the default

- .js file extensions when import a module
- .mjs extension name
- "all or nothing" conversation requirements, __dirname/__filename replacements
- “Named export not found” errors (not sure root cause, however) when package has both default and named exports
- * packaging npm packages * using commons packages in esm projects * use with type safety, configuring typescript causes sometime a lot of pain
- * time * support from all the tooling I use
- 1. Ecosystem does not follow ESM. 2. TypeScript supports it lately (resolved by TS5.0). TypeScript's ecosystem relying on npm still does not support ESM broadly...
- 1. External tools support. Typescript, jest, pm2, ts-node, and many others are just on their way to providing good support for ESM. And you will not be aware that something is not supported until it takes half of your day 2. Mocking dependencies become much harder. `require()` was much better in this sence 3. ESM are not bringing for Node.js anything on top of what CommonJS already had, except of top-level await, which is obviously a very small reason for such a change. The only reason to convert to ESM is to be future-proof
- 1. Jest 2. Aliasing TypeScript extensions combined with package.json exports
- 1. Most of the third party documentation is shown with commonjs syntax, which endorses its use.   2. It isin't the default setting
- 1. problems with mocking native node imports while using jest, jest in general having to rely on experimental VM flag to even start using it with ESM code   2. old libraries / tools using CJS expecting it to not be ESM or my scripts importing them expecting them to be ESM - it would be nice to have more control of which directory gets to have "type":"module" and which not
- 1. Typescript support 2. Observability/APM/instrumentation 3. Compatibility issues
- 1) .mjs file-type 2) typescript not supported 3) problems with using some libs that exported as cjs
- 3rd libs
- 3rd part library support and not being able to use __dirname
- 3rd party libraries
- 3rd party packages
- A little
- A lot of legacy code which takes too much time to refactor (busy with other work).
- A lot of our dependence packages still not use ESM
- A lot of the existing libraries we use don't support it
- A lot of work)
- A pain point that prevented me from adopting ESM in the past was the difference between the adopted import statement syntax and the previously established syntax supported by bundlers/transpilers (webpack, babel). Primarily not supporting pathnames with no extension defined.
- Ability to customize module loading
- Addap all the files with the new imports. And the exports

- additional steps and configuration are required when using with TS (module option set to es6+)
- Ain't broken don't touch it
- All the packages have different module definitions
- All tooling prefers .cjs, node team prefers cjs, easy ioc without build tools
- Amount of technical debt expected
- Amount of work that needs to be invested in
- Angular
- any
- Any
- Any blockers
- Application start syntax checks might be helpful
- Applications that must contain both ESM and CJS or using modules that are ESM/CJS in a CJS/ESM application.
- aren't doing it
- As a _library_ author: _everything_ about this :(It's taken me months to get the Redux libraries to more-or-less be ESM-compatible, because there is _no comprehensive documentation about how to publish compatible libraries properly_.
- As a library author I need to ship both, what's the road map for EOL cjs?
- As I said I just need http3 and websockets
- at least __dirname doesn't work like before it was updated to ESM
- Availability of libraries in ESM format.
- backward compatibility with CJS. I have npm packages and need to have bundler to bundle from ESM to CJS. Many companies are using CJS still.
- Bad support by TypeScript
- Bad UX. Module resolution is a lot more complicated, errors are often not very clear, loaders behaviour is constantly changing, performance is worse than commonjs, no compelling reason to want to switch to ESM as commonjs already does everything ESM does but better.
- BC in many libraries including internal one's
- because CJS is not gonna be deprecated any time soon, most package authors still ship in CJS. because they don't want to lose CJS users. so the package authors focus is still on CJS and the DX of ESM is not good as it can be.   I suggest Node.js should somehow priorities ESM over CJS for users. (e.g: npm init should init an ESM project be default)
- Because not every package supports ESM. Maybe using ESM for mocking testing.
- Big codebase
- Big old codebase
- Bo
- boss and budget
- Broken code coverage
- Bug
- Building binary with pkg
- bundlers

- Bundlers
- Bundlers and open source npm modules support
- bundlers configurations
- Catch 22 bugs
- Changing file extensions is annoying. Babel has found a way to bridge ESM and commonJS for ages. Just disallow top-level await.
- Changing the previous commonjs imports.
- Circular dependency crash server when try to migrate to ESM
- CJS Dependencies
- CJS interoperability
- CJS works fine, ESM doesnt. Poor backwards compatibility. Endless errors i should not be having. Its borked. If im using ESM ill just go to Deno, theirs actually works.
- CLIs not supporting ESM configs
- Code change
- Code format, File structure etc
- Code migrate
- commonjs
- CommonJS interop, outdated libraries on NPM
- CommonJS is the best:)
- Commonjs support
- CommonJS/ESM compat
- Company adoption...
- Company's management
- Compat between esm and commonjs
- Compatibility
- Compatibility
- Compatibility between ESM and CommonJS
- Compatibility between libraries
- Compatibility with CJS.
- compatibility with CommonJS libraries
- Compatibility with node modules is broken for some dependencies
- Compatibility with other dependencies
- Compatibility with other packages. Ensure I can support cjs & esm at the same time for my packages consumers. So far, to achieve that there is no official doc and multiple blogs and tutorial with different approach and none of them are trivial IMHO.
- Compatibility with other tools and libs
- compatiblity to old runtime (like old browser support etc.)
- Complexity to integrate when moving from CommonJS when having the habits of Webpack ease of use
- Configuration
- Configuration, Typescript compatibility, Packages compatibility
- Consistency across packages & easy distribution

- Consistency with modules not all having ESM structure. Especially in our TS projects. The reality is that I don't want to deal with the issues each time there is a conflict. Just fix it.
- convertation from common js modules
- Converting cjs packages to esm
- Converting each file to .mjs extension
- Converting imports
- could not config jest with ESM, some package in node_modules had conflict with ESM, so I had to use .mjs ext instead of global type: module
- coworking with commonjs, TypeScript
- Cross-compatibility, i.e. using code for ESM in one context and without it elsewhere, using packages which don't match module, and understanding what the hell is happening when something goes wrong, and why it ought to be my problem. Mostly, I just want to write JavaScript and for it to work.
- Dealing with legacy code and interop hacks in third party libs
- Dealing with the mix of ESM and CJS, and tooling (like Jest) not supporting ESM
- Default export behaviour
- Definitely Jest and it's poor support for mocking ESM because it's blocked by unstable VM module in Node.js. This situation is bad and looks like there is no solution sadly
- Deleting file import cache isn't possible in esm but possible in cjs require cache
- Dependecies, local scripts
- dependences not support esm
- Dependencies
- Dependencies
- Dependencies
- Dependencies and complicated configuration
- Dependencies are mostly not ready for migration
- Dependencies don't always play nicely with ESM projects
- Dependencies not supporting ESM Runtime (e.g. AWS Lambda) not supporting it
- Dependencies that are not using ESM yet.
- Dependencies that doesn't support ESM
- Dependency with Cjs library. Please drop support to cjs library starting from a fixed future release version. It’s the only way to switch to esm!
- deps
- deps
- difficulty to convert projects because of other dependencies tht do not use it. the ecosystm needs to come to a consensus or better portability
- Documentation
- Documentation, not sure how to do it
- Doesn’t play well with TypeScript imho
- Don't work done library with modules
- Dual format hazards, complexity of defining exports in package.json properly
- dynamic imports

- Dynamic imports are not possible with "caching" and requires some hash in the import to make them work.
- Dynamic synchronous imports. Something as simple as ``` let someModules; if(whatever) someModules = require("someModule.js") ``` is no longer trivially possible.
- Dynamic update files
- Ecosystem inconsistency
- Ecosystem is much better today but still catching up, especially for large apps where swapping to a modern alternative is a complex task.
- Ecosystem is not ready yet, unfortunately :(many projects do not support esm yet, for example Nest.js. Also writing .js extension in every import isn't beautiful.
- Ecosystem moving too slowly
- Effort to achieve interoperability with non-esm modules / libraries (both sourcing and peering)
- Electron
- Electron cannot load esm modules so vscode extension must be bundled.
- ELECTRON FRAMEWORK DONT SUPPORT IT FOR THE MOMENT
- enterprise
- Error handling
- Errors like dynamic import is not supported/module.require is not defined/__dirname when mixing require's from npms and esm
- Errors that are dfficult to understand in the migration process
- ES Module interop and the necessity to extract default from already-default import
- ESM and Commonjs interop
- ESM does not offer a way to manage module cache
- ESM improved a lot, and works very well in a pure-ESM stack.
- ESM is not that flexible for dynamic import and exports I think. And still the compatibility problem. Sure the type support is better in ESM, but I'd rather waiting for more powerful ts support.
- ESM was boy supported in older versions
- ESM without 100% backcompat for CJS/require was a mistake and ESM should be completely abandoned.
- ESM/CJS interop
- Even in Node 20 still no support for in CJS very common items: __filename, __dirname, require.resolve. Available workarounds are convoluted and produce inequivalent results. Standalone files with a node shebang "#!/usr/bin/env node" and no file extension that are directly executed from the command-line will only interpret as CJS. Various solutions offered by the community were closed/dismissed by maintainers without a clear explanation.
- Existing libraries
- Existing modules that do not support ESM (Jest). Typescript support for file extensions.
- Exists tooling
- Exports map are great, no blockers
- External dependencies requiring the usage of CommonJS.

- External libraries
- External Module support
- External npm packages (dependencies), mostly tools
- Fear of breaking an already working app
- few packages had problems with esm but you can always find alternative, also code consistency with older projects in company
- Figuring things out
- File extensions
- file extensions on import index must be index.js
- File system, path
- Finding a doc of proper ESM with a framework like express implementation with good practice. I might need to plan and migrate whole services. And a quality trustable official express project structure will boost the upgrade process to many projects (not only mine)
- Flexibility for backward compatibility
- For company choice I use typescript (that I dislike) but I can't understand the benefit of this changing the module system...
- Framework used written on TypeScript locks me to some convention where ESM isn't applicable. Nowedays I learning alternative development approachs including mainly native Node.js without rich framework.
- Frameworks like Jest, NextJs, Electron are blocking out transition to ESM
- getting it to work
- Getting type info
- Had to use babel to allow better setup of an ESM project with the new standards
- Hard to understand how to make the connection between user side and server side
- Have chosen not to change at this time, because we're switching to typescript instead.
- Have not encountered any use cases for me, so far.
- Have to remember all of that
- Haven't ponder on that yet
- Haven't seen the need for it. What I have seen is some packages converting to it, locking non-ESM projects to outdated versions. I'd say compatibility is an issue here.
- Haven't tried to use yet.
- Haven't tried. There is lack of standard and many configurations when running with typescript
- Having a good un understanding of ESN benefit. And how to migrate with typescrypt ecosystemes and and all packages
- Having to "change the whole world" just to convert from CommonJS modules.
- having to specify the extension is a bit ugly
- How to distribute as dual packages
- https://github.com/nodejs/node/issues/37648
- https://nodejs.org/dist/latest-v20.x/docs/api/vm.html#class-vmmodule status
- huge changes
- I am afrais of ESM being slower than Commonjs, since import is eager whilst require is lazy. All other things being equal, I dont know why would I prefer the slower solution

- I am stuck in synchronous CJS contexts where I cannot import ESM.   ESM interop has been an utter nightmare, where the best option is to simply not try to do both ESM/CJS
- I am used to common js modular style and keep it consistent
- I am using ESM with ts NodeNext, it's fine
- I can import a commons module but not require a esm module. So it’s safer to write commonjs
- I can't make my package ESM-only because some users still use CJS. Which means ESM-only packages can't be used in my packages either.
- I don't own the projects and PO is hesitant to do breaking changes
- I don't see any direct advantages of using ESM over CommonJS modules, especially with Typescript which resolves everything to single import call.
- I don't use common modules, I only use ESM
- i don’t understand why it’s not the default
- I dont like typescript, but when i do, it works still very bad with esm. So I dont like eg nestjs, that is very hard to work with esm and esm packages. Just like to work full native node with esm, than it works very good. So its only typescript and too large dictating frameworks that make it hard
- I find it much less trivial to write unit tests
- I hate that, writing TypeScript, I have to add .js file extensions to imports.
- I have to remember to add the file extension on the import
- I intend to use ESM only in new projects. Please let us need only the .js ext with type:module.
- I like ECMAScript
- I prefer CommonJS
- I prefer esm as default choice instead of custom one
- I tried to port my own app to esm modules and found out that many feature in the current package uses esm as unstable feature and I have many warning about it. Although app syntax and working are correct
- I use typescript
- I use typescript
- I want to use ESM with typescript/babel + hot reload + build tool for building the same source also for Web.   Pain point — hot module replacement without restarting whole app
- I wish it would “just work” without all the configuration, errors, etc.   Maybe libraries use ESM now, and I prefer it.   It’d be nice to be able to use ESM with CommonJS in the same project.
- I work almost exclusively in CoffeeScript, which supports ESM, but it's non-obvious how to get it working.
- I'm mainly using typescript för this
- I'm using TypeScript
- I'm using typescript, JS is not good for scale up project team
- if a cjs module isnt isnt inline requires or like tag templated requires, its no different than esm. esm is dumb. just cut the shit and provide interop out of the box with no messages

and no crazy package.json export fields etc. also , maybe send whoever came up with that dog water to the roof.

- If you have the right packages it is not a problem
- If you wish to use ESM in an existing application, what have been the pain points or blockers preventing you from doing so (if any)?
- import export
- Import extension
- Import issues
- Import json files
- import map : https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script/type/importmap add migration docs for cjs to esm. Specify how to switch from __dirname to new URL();
- import vs require
- Importing CommonJS from ESM, general incompatibility from 3rd party packages.
- imports
- in build
- In ES modules (ESM), there is no direct equivalent to the require.cache object in CommonJS modules.
- In my project I use NestJS, and unfortunately it's Swagger CLI plugin doesn't support ESM
- Inability to use it out of the box in some cases with unclear errors
- Inadequate documentation, some modules support ES6, some only CommonJS
- Incentives to switch when we already hv typescript
- incompatibility with commonjs
- Incompatible peer dependencies
- Incompatible with ecosystem out of the box, requires use of shims
- Incompatible with many packages
- Inconsistency
- integrating with npm packages that haven't migrated yet
- Integrations and self-written libraries
- Internal library support
- Internal problems
- Interop with CommonJS, TypeScript support
- Interoperability between different types of modules
- Interoperability with cjs
- It has been relatively painless lately. But if I recall correctly, once in a while a package doesnt have a named exports. So I have to import the default export and then use the specific named export.
- It is difficult to get en overview what solution is currently the best way to go and consumability by bundlers can be a problem.
- It is hard to get modules play well with legacy commonjs code
- It requires you to go all-in, and lots of tools that I use don’t work with ESM (e.g testing libraries). More cross-compatibility between CommonJS and ESM would be good, e.g supporting __dirname

- It's not backwards compatible
- It's ok in typescript, a hell in Node.
- It's to difficult to understand
- It’s ok
- It’s very hard to support ESM and CJS in a library
- Jest
- jest
- Jest
- jest
- Jest and organization priorities
- Jest can't handle it, and by proxy cosmiconfig and the libraries that use it (blocked by Jest again)
- Jest doesn't support mocks. Other than that the transition was smooth
- Jest has a few limitations on ESM
- Js
- Just the usual, codebase built with commonJS modules
- kill ESM, please
- Lack of APM support, difficulty of exploring a singleton due to deepest code running before shallowest code.
- lack of expirience
- Lack of library support
- Lack of native support by tools and libraries, specially Jest, Typescript and bytenode
- Lack of stable support in popular JavaScript frameworks and libraries (ex. Jest)
- Lack of stble ESM support by testing libraries (Jest, web-ext, karma)
- Lack of support
- Lack of support from existing libraries and differences in Node's ESM module resolution compared to other tools
- Lack of support in packages and general community
- Lack of support in some instances.
- Lack of will
- lack support of third-party library
- Lacking support in common libraries (typescript last time I checked), but also some libraries are forcing ESM on me in their latest version.
- large codebases require a lot of repeated changes in a lot of files
- Lazy load
- Legacy
- Legacy
- Legacy code and interoperability
- Legacy code and unmaintained commonjs packages
- Legacy code refactor is expensive, hard to mock modules for unit tests
- Legacy dependencies. And also existing CJS is good enough for the back-end, where async loading for modules can produce unpredictable behavior.
- Legacy libraries, bundles, dynamic imports

- legacy libraries; interop with tools using non-standard ESM; the unfortunate sticking power of non-standard ESM; I want to be able to use JS/ESM as close to natively as possible so that one day we don't need excessive build tooling and can share more between Node.js, browsers, and any other ECMAScript-adherent environment
- Legacy microservices that use old versions of Node
- Legacy packages, tools (jest, vite), mocks
- Legacy with common modules
- Libraries
- Libraries
- Libraries and packages not interopable with ESM.
- Libraries compatibility
- Libraries don't support ESM, but CJS
- Libraries that do not support
- Libraries that does not supported it
- Libraries that don't support esm
- Libraries, Typescript, compromises made by node for the sake of cjs interop
- library authors don't understand ESM basics like default exports
- Library incompatibility, lack of clear migration path for libraries
- Library support esm
- Libs
- libs with only commonjs support
- Libs, which work in cjs only
- Like one year ago I couldn't mock an ESM module by using node-tap. It might have changed by now.
- Loaders, module interop
- Lots of legacy code
- Lots of packages don‘t work with esm
- Low adoption
- low dependency support
- Low support, command line options, mix between commonjs and esm
- mainly dealing with non-esm modules that dont play well with esm. As an example, code coverage (such as nyc) when combining esm, typescript and node was very bad. Deno was better
- Making the changes to properly switch.
- Managing a large number of dependencies which may or may not support ESM/Commonjs
- mandatory CJS and ESM support → dual-build pain
- Many
- Many Libraries do not support esm. Many unknown errors occur and not easy to solve. Package.json definition is very fragile and it is also different among package managers, please do have a blog post to unify all these defs.
- mess in module resolution, too many different extensions
- Migrating code within custom build setup

- Migrating existing applications and working with a mix of both styles for different parts of a project combined with typescript
- Migration
- migration is a lot of work for bigger projects
- Missing __dirname
- missing APM support in ecosystem
- Mix of esm and cjs
- Mixed codebases
- Mixing with CJS
- mjs extension
- Mocking and many are plugins for legacy CJS packages
- Mocking in test runners
- mocking modules for tests
- Most of libraries play well with CJS, however it's hard to mix CJS & ESM modules in a single project. I guess Node.js still needs more interoperability.
- Mostly libraries compatibility Example: jest and ts-jest
- Mostly observability. From my understanding, open telemetry libraries do not really work well with ESM.
- Motivation to use ESM is too low now.
- Moving end users onto esm friendly tooling, e.g. bundlers
- My old code base and other teams work on the same code base with us
- My pain points are the packages that i installed via pnpm which only supports commonjs and not ESM
- My projects are all ESM only. I would like to see more libraries converted to be ESM-only especially backend libraries that Node maintainers are involved with
- Need in every file builded with typescript compiler write .js extension
- Needing to rewrite everything
- Next.js does not support it natively and lots of libraries are still CJS. I also think the .js specifier when importing files is a blocker because it’s a pretty serious breaking change from before
- No __dirname etc. Support, no easy way to import some non-ESM dependencies
- no autoload
- No clear documentation on how to use it. Is it required to use .mjs extension?
- No clear migration path, or interoperability
- No idea why I would I wish to expend cost for zero benefit
- No quite sure how to use it
- No reason to do this
- no significant blockers
- No standard way of transformation
- No string interpolation
- Non compatible ecosystem
- non-obvious configuration in the presence of a typescript and obsolete libraries
- Not all libraries uses the ESM module
- Not all modules are esm (a lot of modules are still commonjs only) and this gives issues

- Not being the default
- Not considered it
- Not have that much experience. But would love to explore
- Not having access to the cache like we do with require.
- Not intuitive, required runtime arguments
- Not really many. I remember some weird behavior around immediate exports, but I think that ended up being something something unfortunate project on a windows box...
- Not sure why I should
- Not sure, I usually use TypeScript and that solves quite a lot for you
- Not the default yet, colleagues afraid
- Nothing actually, I'm just too comfortable with cjs
- Nothing ever works well in the ecosystem with ESM enabled
- Np
- npm compatibility
- npm packages aren't ready for ESM. For example, jest doesn't have normal support of it, and other test runners are raw and aren't ready for production with ESM
- NX is not fully supporting ESM.
- Obsolete framework we use and can't get rid of easily (sails.js)
- Occasionally third-party modules which don't support esm
- Old but stable and reliable packages work in commonjs, and won't get updates. ESM is good for bundling, but nothing more.
- Old Commonjs code could not compatible with the ESM
- Old dependencies
- Old libraries
- Old libraries
- old libraries doesn't support esm modules and throws errors
- Old library’s
- old libs
- Old libs that doesn’t support esm
- Old packages
- Old project
- Old third party modules compatibility
- Oldest libs
- omg mannny, always suffer with setting a lot of configuration, oh no this library can't work with esm WE NEED TO ENFORCE ECM, and without file extension native support without configuration conflicts please
- Only starting the migration from CommonJS
- Option to work with imports cache like we do with require.cache
- Other existing libraries that is already CommonJS, like ESLint, current ecosystem is a temporary blocker.
- Other library
- Other tools/libraries that didn't support ESM
- Other/older libs not supporting ESM
- Our legacy codebase

- Package
- Package support
- Package support and project setup
- Packages all work differently
- Packages incompatibility
- Packages not supporting it
- Packages that use common.js, inconsistency of typescript packages
- Pain to use in node backend and lack of support for backend libs
- People on the team who want to use TypeScript .ts files and aren't content with .mjs files with type annotations in comments, even though tsc fully understands and type-checks based on those, too
- Poor documentation
- Poor interoperability between ESM and CJS. Lack of better third-party support.
- Poor support of ESM in the existing libraries. Some just didn't work with ESM, for example https://github.com/trpc/trpc and jest (doesn't work with ESM when transpiled from TypeScript)
- Problems with TypeScript
- Problems with TypeScript integration, like path resolving and difficulties with proper setup.
- Project impacted areas
- project time planning
- Proper and understandable Typescript support.
- Proper documentation with standard use cases Interoperability between the two
- Pure Node.js application like HTTP APIs REST (built with Fastify for example) have no real issues to be ESM.   However applications like React with Next.js struggle a bit more when adding "type": "module" in the package.json, as the whole build and "infra" is impacted and config files that goes with it.
- Refactoring
- Reload constants during automated tests
- rename files (js -> cjs) is my path, not good but it is what it is
- rename things
- Renaming all The files
- Require
- Resource to refactor
- Rewriting imports to end with “index” and “.js” extension
- Rewriting the application codebase.
- Scaling
- Serverless is not working with ESM + some libs are not evolved to gain ESM
- Since I have been using TypeScript for over 5 years, switching to ESM is often very easy.
- Size of effort
- So far, I have faced no problem
- some cjs packages don't get named imports picked up correctly

- some frameworks use commonjs to require some myfiles. So I can't make this files esm modules
- Some libraries don't offer support for it
- Some libraries don't support ESM and if even one doesn't then that is a problem
- Some libraries not supporting it yet
- Some libraries still do not fully support ESM
- Some libs are hardly compatible
- Some libs may not have its support.
- Some node modules don‘t Provide esm modules/packages.   Updating Imports to .js extension
- Some of our third party dependencies only experimentally support ESM (e.g., Jest)
- Some old packages like gulp break with esm.
- Some package doesn’t support correctly ESM and no docs with ESM import. All example with require.
- Some packages do not support esm, only commonjs
- some packages do not support it
- some packages like jwt not supporting it yet.
- some packages may not support ESM
- Some packages/libraries
- some projects don't support it like cosmiconfig
- Some tools don't have support for ESM
- Stability of current workflow. Collective studying and organising. Better control direction through components on architecture.
- Stable support for loaders to compile TS on the fly with esbuild
- Still not very sure about the mjs extension.Tutorials don't use it
- Structure
- Stuck on legacy Node version, not all dependencies compatible
- Support in tools and libraries is garbage. Every time I try on a project, I end up so screwed up trying to find integrated versions of things that work.   ESM is the Node.js Python 2 to 3 failure.
- Support of ESM in some tools like Jest, Webpack.
- Supporting cjs and esm in exports
- Supporting old version products
- Synchronous require in the logic (usually lazy load of optional dependencies to help with startup time)
- Technical Debt
- Test coverage tools dont work, incompatibility with observability tools
- Test frameworks, default settings
- That CJS is still around, please deprecate it once and for all so we can move on
- That it's not something working out of the box
- The absence of the __ variables
- The amount of differences between CommonJS and ESM in some cases requires a lot of changes to the code base, and sometimes they are not very trivial.

- the ecosystem lagging behind, cjs packages that havent been updated in 3, 4, 5 years etc
- the extension which is .mjs, we should use same .js
- the need to ship two versions, cjs and esm, together
- The only thing that I can't do is hot-swap i.e. `delete require.cache[...]`
- The refactor of the existing source code
- The sysstem seems broken,Not all dependencies are esm compatible, TS gives a hard time with its config options
- The testing story is more difficult due to immutable bindings + lack of Jest compatibility
- The transition from request methods vs the import, import {} and import * as
- The whole .mjs and .cjs thing is a mess and causes lots of friction. I wish these were never added.
- There are no issues with ESM. The only thing is the old commonjs packages and of course, the slow adoption of the whole community. NodeJS docs and community leaders should promote as much as possible ESM
- There is no specific doc to update existing app
- there need to be more libs that support ESM...
- Third party libraries. Integrations with jest/vitest, etc. import alias
- Third-party libraries
- Time
- Time
- time
- Time
- time
- time
- Time
- Time
- Time constraints
- Time consuming rewrite of code in a large code base. Incompatible packages.
- Time consuming to give the step
- Time to do it. We need stability, this kind of change is a nightmare in big projects.
- Time, and the projects work fine without migrating to it.
- Time, it is on the plan.
- To grasp every little nuance in converting modules, especially with typescript included, is beyond mortal men. It's mostly trial and error until it hopefully works.
- Too many dependencies still using CJS
- too many own modules and libraries who must be converted.
- too many variants
- too much code already written
- Too much handcraft   (Waiting for an IA to automate it)
- Tooling not supporting it
- Tools
- Transitive dependencies being CJS only

- Tried to change our main application a year ago to esm. The main blocker was jest. After a workweek we abandoned the transition
- Troubles of interoperability with old dependencies
- Trying to get it to work in node with old libraries
- TS makes it difficult to transpile CommonJS identifiers into ESM. eg. `import "./myModule"` works when transpiling (and using) CommonJS, but if we wanted to use ESM, all our TS imports would have to use the clunky `import "./myModule.js"` syntax even in TS code
- Types documentation for npm libraries do not always work with ESM imports, so features like intellisense are not always available
- typescript
- TypeScript
- typescript
- typescript
- typescript
- typescript
- Typescript
- typescript / ts-node incompatibilities. test runner incompatibilities.
- Typescript and jest
- Typescript and modules are hard without bundlers
- Typescript build step; extensions in imports
- typescript doesn't work well with it, fastify doesn't support it well
- Typescript ESM compatibility
- TypeScript fails. I think I have enough experience with TypeScript to set it up properly now, but I see no reason to do this
- TypeScript integration
- Typescript integration
- Typescript integration e.g. adding file extensions where they are ".js" but the file is a ts file, it's just confusing.   Also since the ecosystem has other module types, using ESM with some dependencies that are not ESM gets difficult e.g. ESlint which does not support async functionality, so some setups could be broken or blocked from upgrading to ESM. Async importing of ESM modules in CommonJs is a pain.   Json importing with the new assertions is also a pain with typescript as it seems to require some specific tsconfig options
- Typescript support
- TypeScript support and cross-compatibility with CJS codebases
- TypeScript transpiling   gaps in ESM support by linters and test runners (getting there though)
- TypeScript, framework dependency
- Typescript. Specifier resolution.
- Typescript's weird bundling modes, lack of modules that are correctly bundled for both CJS/ESM, lack of availability of Node >= 18 as a Lambda runtime on AWS
- Unavailability of some modules
- uncompatible node.js packages, typescript support for many old packages

- unittest
- Upgrading is painful, many existing tools and libraries don't work.
- Using commonjs lib
- Using ESM with jest in my current projects brings a lot of trouble to run and debug tests. There ins't enough time and team members to find a fix for all of out microservices
- Using old libs without import
- Using typescript or a third party library
- Using typescript with esm is painful as i cannot run the ts files with tsnode since imports need to explicitly end with `.js` extensions
- Vast amount of legacy code
- very annoying
- Very large projects using commonjs
- Waiting for more of the ecosystem to catch up
- waiting to aws sam support
- We have no clear sense of direction or current recommended approaches in migrating a complete application to ESM. There are mixed messages in the community. Some say do it. Others say wait.
- We use typescript maybe in 99,9999% of projects and to be honest built-in ESM is cool but only for libs or personal use like pet-project, where you can have fun with native js.
- Well, the first one is import * as / import * from *, the second one is uncompability with commonjs (had to used really old libraries at some point; but also I don't think Node.js needs to have ability to run them both at the same time - it would be horrifying), the third is no __dirname and __filename constants, the fourth is no way to require directory, the fifth - need to specify the extension of imported file, sixth - always use "type": "module".
- Why change the file extension?
- With Express, a plugin called exegesis is very but does not support EMS for its controllers, but CJS - it's small things like this that hold some projects back
- working with both esm and cjs packages at the same time
- Wrappers like aws lambda otel which are not esm ready
- Writing ESM code in Typescript and transpiling to code that supports nodejs. However, in a monorepo environment which contains Front End application (Angular), I have to maintain separate tsconfig.json file just because nodejs doesn't support proper es version. By having out of box typescript support, such issues could be simplified as seen in Deno.
- You must convert all your code manually from CommonJS to ESM

Q19: Do you have any recurring issues when using Node.js?

- - not being able to pass plain objects to worker threads without doing some tricks with shared buffer arrays.   - the object spread operator is so damn slow.
- - Memory leaks are hard to track down in production - First-class typescript support would be huge
- - missing examples in docs. Sometimes 0 examples for a function
- - Native Typescript support should be enabled out of the box.   - Nodejs should improve the way external packages are imported and the scope those have in the ecosystem. Even if you import one package you end up with 50 and it's hard to keep track of how legit those in-depth packages are.
- "type": "module" should be the default when running npm init. Libraries should not use CJS by default
- (comparing to deno) it feels slow
- 1. Too much memory, and Too slow cause its type interpreted(BigintArray.sort() vs Array.sort()) 2. Sharing variables with Threads
- 1. Weak Platform Integration. For example, modern file systems (btrfs, APFS) supports copy-on-write operation by default, but Node.js' fs.copyFile() does not use it by default even if the platform supports them. We can enable them by opt-in with passing a flag but it does not improve the exist shipped code without any code change.   2. npm cli still slow. I want a more fast package manager that is competitable with pnpm.
- A lot of time spent configuring Typescript with Node.js for full custom projects
- A perf issue between Dataview vs Buffer
- A significant amount of people thinks that node is a joke compared to Java or dotnet. Would be nice to have some arguments against it
- addressing memory usage
- advocating for node to other language champions
- After almost a decade, not so much really.
- all good, no recurring issue
- Always using libraries for core stuff, like Http server, http client, tests, types, linting, dates, etc
- Amount of JS code (including dependencies) and performance characteristics opening the door to tools written in Rust or Go to handle JS (e.g. turborepo, turbopack, esbuild). Ideally, Node would be fast enough to not warrant tooling in compiled languages, provided careful application of algorithms and data structures to adress the task in JS.
- API version inconsistencies, Better native support for window/process management.
- As above (18)
- Async hooks
- Backward compatibility with non ESM packages
- Bad js types
- Bad libraries
- Bad standart input. I must always install 3rd party librares
- Basic typescript types, cold start times

- Being stuck having to support running our code on some very old Node.js versions
- Better doc for service workers with official examples would be great !
- Better logger, template rendering,
- big node modules directory
- Binary dependencies, node-gyp etc
- Breaking changes in JSON Z imports between minor releases
- bring back domains. node process should never restart for unhandled exceptions.
- Bugs. lol.
- Building executables, make them easily shippable!
- Building weird versions sometimes
- Can't access https websites on node 18, had to revert back to earlier versions on my mac environment
- Cannot do heavy calculations
- cannot use node > 14 on my macbook pro m1
- Circular dependency
- CJS
- CJS must die
- Cluster mode and sticky sessions. Slow on high load
- Coming from a C / C++ / Java background, Worker performance seems worse than I would expect. It would be nice if there was something that let you create a worker than inherits a copy of all current imports.
- Commonjs not supported any more with esm
- Constantly upgrading packages and libraries.
- Contributing to Node.js can be very challenging due to extremely flaky CI. I often have to rerun the CI half a dozen times or more to get it to pass. Really doesn't build confidence in the accuracy of those test results…   As for using Node.js, it needs better performance in a lot of critical areas like AsyncLocalStorage, EventTarget, and others. It also could use a lot more development in diagnostics tooling. Other languages like Java are far, far ahead of Node.js in developer UX around diagnostics.
- Corepack. It is an absolute security, reliability, and maintainability nightmare, due to mainly its opaque and silent background operation, hijacking of command-line commands, lack of configurability, problematic install/cache directory choice, and uncooperative maintainers. The whole thing is a grave violation of user-agency by obstructing and usurping control over their own systems, a massive breach of trust due to rampant abuse of system permission/access levels whilst obscuring these actions from the user, and potentially harmful to user files and runtime environments through its malicious-software-like trojan distribution mechanism, user command and control interdictions, operational interference, and file-system vandalism.   Worst of all: since it comes bundled with the precompiled distributions of Node through NPM, we are forced to compile our own builds that exclude this nightmare.   Just bin the project already and bring NPM in as an optional core component. Yarn is dead, PNPM is too broken, package managers are all mostly the same and their differences don't even really matter. Other than all that I'm a huge fan of Node and it has done a lot for me. It really hurts to see this rot developing right inside Node's core.

- Creating browser and node compatible libraries that use web workers
- Cyclic dependency issues which are usually hard to debug and resolve.
- dead dependencies
- Debugging installation failure, observability and monitoring.
- Debugging using documentation
- Debugging, better typescript support.
- Decreasing performance due to each release, lack of http 3 and bugles http 2, lack of multithreaded mechanisms which are more user-friendly
- Dependencies In ESM
- Dependency hell
- Dependency management is still a pain. Very hard to ensure that a legacy app has a reliable upgrade path.
- Dependency Versioning (legacy-peer-deps)
- Deploying
- Deprecated packages, memory limit, node watch bugs, windows apis
- Direct typescript execution
- Do not
- docs is aeful
- Docs missing arguments concrete types and return concrete types, missing Non-Async functions in API (need sync versions too).
- Documentation is not good enough. More divers examples would be nice
- Documentation is sometimes difficult to navigate. Some Core APIs are confusing sometimes, e.g. fs (methods with callbacks, sync methods, async methods / methods returning a promise).
- Does not handle circular dependencies well. Sometimes due to cyclic domain relevant though
- dumb developers from 3-month courses who somehow got a job
- Error Handling
- error handling is terrible & no built in typescript support
- errors wont appear untill runtime (in JS) other than in TS
- es6 features doesn't work OK
- ESM
- ESM in general when using third party packages
- ESM support is infuriating.
- ESM support, non-descriptive error traces (specially for async code and chain of Promises)
- ESM vs CJS
- Even if it could be more related to the package manager, SBOM is currently a priority on my company. It couls be great to have something to obtain a list of all the third-party deps (an nested, like the package lock) but including also all the possible cve and licenses. Like a super package lock + audit
- Features often stay experimental for too much time, even if there are not lot of changes and could already be stable (at least close to be). A good example of that are the methods fs.cp(), fs.cpSync(), fsPromises.cp(). They were introduced in Node.js v16.7.0

and didn't get lot of changes, but they are still marked as experimental (issue https://github.com/nodejs/node/issues/44598).   Also it could very good if Node.js could ease the mocking of builtin core module as the test runner is now stable. mock.method(fs, 'readFile', async () => "Hello World"); Is already good for mocking, but we need something less dependent on the code to be tested.   I want to write tests without knowing what fs methods I call, but still ,they are mocked, and does not read/create files in reality. A good module for fs for example is mock-fs on npm, but unfortunately is not compatible with the current experimental fs.cp() methods...   Better "mock" can be applied to other core modules not only fs.

- For big calculations the application a lot more time then expected even for a single threaded application.
- For every new Node.js version, I have to enable yarn. Also, I have to reinstall all global packages.
- For the most part, my issues are project specific or setup specific.   I know that I run into problems regularly when trying to use core node.js features in TS projects.
- FS API lacks many common needed functions. child_process API is problematic. updating dependencies is hard (incompatible peer deps).
- Get rid of the Y/N prompt when exiting a script when running on Windows
- Good standards are not followed when working whis other team members.   Better error management
- Had one with the native test runner, is solved now, but for me it takes to long to release a bugfit for the coverage
- Hard to do multithreading because threads/workers can't share data. We have to through postMessage which is slow, limitation for sharing complex data.
- Hard to hire giving the fact rust's popular
- Having to reinvent the wheel about logging every time, as there is no real log/console interception
- Having troubles with research for performance optimization
- heap memory runs out easily, I'm constantly using --max-old-space-size=8192
- Heap out of memory
- Honestly, eslint and typescript should be basic features at this point. I can't imagine building things without them.
- Http-requests should be easier, I use axios and cheerio as an alternative
- HTTP/3
- HTTP/3
- HTTP/3 support
- HTTP/3 support
- Http/3, websocket
- https://github.com/nodesource/distributions/issues/1392
- https://redfin.engineering/node-modules-at-war-why-commonjs-and-es-modules-cant-get-along-
- i am scared a bit coin miner is taking over my computer every time i npm install with these crazy high security vulnerability threats, even though its just a regexp

- i can't control application after reach memory limit and crush. hah. in this case process.on('exit') can't prevent crush by cleaning memory and continue to work. i know how it sounds, but...
- I feel the documentation can be more explanatory
- I might need to migrate away from NVM. When last I checked, it doesn't respect .node-version files. We need more adherence to standards across the tools which manage Node.js and the tools that are managed BY Node.js.
- I need http3 and websockets
- I often run into compiler errors when installing packages with native dependencies. Also mode version managers can be finicky.
- I see no reasons for Node.js to make permissions system that Deno has, would be better to add ability to import from URL from the box.
- I used to have a lot of issues with node-gyp, especially as an educator helping multiple students on different architectures set up node on their machines, but this seems to have been resolved recently.
- I would like to have native support of typescript in Node.js
- I'm currently having major issues getting "node-gyp" to work inside qubernerneties
- I’d love if package.json could add another object for global dependencies.
- If node could strip type annotations and run TS directly that would be slay
- If we could make the Nodejs binary somewhat smaller I would be happier
- Im a purist, I want JS, some people around me push for TS... they can all move to zimbabwe and have a nice life away from me
- Import and require issue
- import/require: incompatibilities with packages
- Importing commonjs modules from esm
- importmap, native websocket, create and assert snapshot in nodejs test runner
- In a reflection on my screen, I see a silhouette of Ryan Dahl shaking his head in disapproval whenever I use package.json for custom metadata.
- Incompatible library versions
- Incompatiblity with some libraries when updating to the latest version. Had problem with nuxt at some time after nodejs version upgrade
- Inconsistent or wired behavior around async hooks.
- Initial project setup. Pnp in yarn and pnpm (but not node.js related)
- Installation is still a bit long, requires too many selections and isn't really suited for new users or folks that just need it as a dependency to try out new things or testing. Why not make a GUI like Docker does and give users the ability to manage node from there?
- Installations sucks please learn from rust.
- Installing pulsar-client package on Mac M1
- Interoperability between ESM & CommonJS
- issue node modules size
- Issue that we must have Http/3 and websocket yesterday
- Issue with openssl 1 and node 18

- issues with typescript support issues with ESM modules still hard to work with unexpected exceptions in runtime still hard to detect which user blocked the execution and close it (fail fast)
- It could borrow cohesiveness from the Go project: build, document, manage dependencies, test all with the same binary
- It ships a not-so-good package manager as the default, named npm.
- It's hard to use it with Typescript
- It's related to different packages not updated with old version of node js
- Its good, thx for all your works.
- Its hacky
- kinda, I miss some useful tools from lodash library, but library itself is big and mostly useless, I know you can always do it in vanila node but its not the same, so I would really love to have something like _.get(), or _.pick() in core node.js, also supporting dot notation.
- Lack of a good reflection API
- Lack of async stack trace
- Lack of official Windows ARM binaries.
- Lack of out-of-the-box support for TypeScript
- Lack of runtime observability in the runtime means doing things like performance traces for which bits of code use CPU is nearly impossible, especially compared to other application runtimes like .NET or the jvm
- Lack of SIMD, weird support for parallel computation. Lack of solid parquet/arrow tools.
- Lack of standard web framework, old non-fetch http server (IncomingMessage)
- Lack of standards
- Lack of Tail Call Optimization support
- Lack of tools that helps debugging threads / cluster for Node apps, need to connect dev tools directly to thread, panifull when using pooling
- Lack of types
- Lack of TypeScript and enterprise language features
- Libs
- Limitation for When we are creating apps on Enterprise level. Then nodejs fails to deliver. Because of this Failure Many tech companies moved on other platforms like Java and DotNet. You need to work on i
- Load balancing, multithread
- Loclhost host lookup breaks on Node 18 while doing a fetch request to a localhost address where the Server is only available on ipv4
- LTS is too short.
- LTS too short, security model non existent.
- Make NodeJS faster
- Manage it as a OS service under Windows
- Marjory of developers seems to be confused with Promise/then/await
- maybe if it has its own package similar to PM2 that would be great.
- meanwhile nope
- Mem usage i.c.w. Angular builds

- Memory consumption
- Memory consumption
- memory leak
- Memory Leaks
- memory leaks and/or huge memory usage that might crash app as memory leaks do.
- Memory leaks are everywhere.
- Memory limit leaks
- Minors issues related to external modules
- Missing or complicated support of ESM
- missing standard library better documentation - the current one is complete, but hard to navigate
- Missing Web API's compatibility. In Deno I can take most of the code from web and check it on backend. For instance localstorage - it's super convenient to 'just use it' on backend instead of playing with some file access libraries.
- monitor error or performance
- More ecosystem related. If you're not staying on the edge of the trends, tech debt accumulates (example: when request package was deprecated).
- Mostly libraries incompatibilities, node_modules dependency resolution issues and non reproducible builds.
- Mostly memory dump issue
- Multithreading would be nice for heavy load jobs
- My main gripe is that --cpu-prof doesn't work for workers or child processes
- NAPI too difficult to use
- Native CPP module compilation with nodegyp and electron can be hard sometimes, but it's not related to nodes directly
- need to build from source when on ubuntu
- Need to change Node versions due to tools not being compatible with latest versions. For frontend development I have to keep Node 16 due to issues with Babel version used in my project...
- Needs typescript support out of the box
- New native dependencies complicate version upgrades when building from source
- No any issue but hoping computational language in future
- No native TypeScript support
- No nodeJS is good for me
- No other than ESM (import syntax) configuration and using typescript.   E.g I'd like to be able to just run a .ts file like   `node index.ts` ts-node is supposed to help this but maybe I'm just dumb because I often run into issues with that and module syntax. (Even when using —esm flag)
- No proper inmemory database for quick db testing
- No recursion   No recursion   No recursion   No recursion
- No. But I wish Node.js to have faster native HTTP server to match Bun.js and uWebSockets.js performance
- No. Node is great platform, by the way.

- No. Node.js > Python, Go, etc...... Really love it. When I first tried it a year ago - it was a love at first sight.
- Node documentation is always an issue.
- node stack traces are mostly unusable, fixing them will help immensely
- node-gyp is pain in ass, honestly
- Node.js CI ☠️
- Node.js doesn't update automatically
- Nodejs is not for CPU intensive processes
- Nope, it works remarkably well. Deno is nice and shiny and does some things better (notably, native TS) but not so much I'd be willing to jump ship.
- Not issues but need better tools to understand where the API has more latency.
- Not really. Maybe better support for running node as a service on windows
- Not smooth and easy to write CLI applications or Server-Side Client applications
- Not yet, would be great to have compliance implemented in nodejs core
- Nothing as such some complaints here and there maybe
- npm docs is bad
- Npm version updates and vulnerable packages
- Numeric precision of number type
- Old and new version conflict
- Only on applications requiring multiple versions of node
- Only ones related to still being on node 14 while the rest of the ecosystem passes us by
- Other than having tons of space used by node modules, not really
- Overwhelming complexity of configuring packages for publishing (especially when using native ESM)
- Package-lock
- Packaging an application as a single executable, including the runtime and its dependencies.
- Parallelism
- People often use global node packages in npm scripts, which means you have to install them before installing project dependencies.   I have seen people use typescript's 'tsc' as a global module, when it should really be used as a dev dependency.
- People still believe it's slow
- performance
- Performance analysis and profiling
- Performance compared to other programming languages
- Performance could be better
- Performance debugging
- Performance is poor. I think node.js might start looking at what Bun.js is doing.
- performance issue like heap memory and cpu intensive. issues might be there
- Performance safes money. I love money. I’d love a better performance in Node.js overall
- persistent npm dependency conflicts
- Please add http/3 support
- please do more benchmarks to chut up rust and go techbros

- Please help push to integrate the Types as Comments, so we can finally be rid of Typescript and the horrible build step and non-standard feature implementations.
- Pm2 support is not good. I have deployed an app on cloud in pm2 instance which gets closed randomly a.d doesn't resurrect the processes on startup.   Also, please plan to make a tutorial kind of a portal on nodejs documentation website like Vercel did with Next.js   Found it really useful
- Poore profiling, monitoring tools, add more project specific functionality (like core support of websockets)
- Problem with incorrect module versions, and not being able to run Typescript directly.
- Problem with multithreading
- problems with docs
- Process manager for running node in production. Itegration with IIS.
- ram usage
- random errors mainly related to CJS/ESM mismatch of libraries/applications
- Readable.toWeb has a bug that reads files into memory
- Regular npm install / update / ci confusion by junior developers. Typescript boilerplate compiling/ts-node'ing/etc because of lack of support in node
- Remove common js
- scaling
- security, design patterns/structure of the project node_modules wtf?(a lot dependencies) education for high-quality code TYPES
- Setting up a project for a library vs an app is a bit confusing
- Should improve support for ES6 modules
- Silent errors when using async await and forgetting to catch
- Single threaded
- Single threaded
- Size and complexity of dependencies, libraries insisting on cjs, lack on package manager interop (eg pnpm requires it's own lockfile)
- slow startup time
- Solving packages security problems in node modules
- Some older modules may not be used
- Some problems with node_modules folder and npm-packages. From time to time strange things happen, when some module does not work or works but not correctly. And sometimes the solution is a complete reinstallation of Node.js by deleting the node_modules folder.   I want global node_modules folder like in pnpm or Golang modules.
- some unneeded executions of wmi.exe or windows registry modification when this kind of activity is prohibited in corparate laptop
- Some vulnerability issues
- Sometimes binaries being unavailable for windows/node versions combinations. Issues with building dependency binaries with VS Community, node-gyp, python (the windows / node.js ecosystem).
- Sometimes I have problems with npm, after installing several dependencies the installation process hangs and I have to restart the processes.

- Sometimes running commands becomes slow, not sure what triggers it but in some cases re-cloning the repo the commands are run from solves the issue, it doesn't make sense
- Spawned child process sometimes doesn't respect SIGTERM or SIGKILL signals and ignores exit. Need proper documentation on how to tackle or simple API to quit all child processes when from parent.
- speed
- Spurious crashes when Domain is activated
- stability of the particular node instance
- Startup time, no built-in TypeScript support, mediocre docs
- Stream's api should be better documented
- Streams and HTTP are a nightmare. They changed in v14 and again in v16 in subtle ways that have cost us a tremendous amount of time trying to hunt resource leaks. For example: https://github.com/nodejs/node/issues/42610 and https://github.com/nodejs/node/issues/40775. Lore is that these are old APIs that can't be changed because so much ecosystem relies on their internal state that should never have been exposed, but they're changing and there's obviously inadequate unit tests for these. We really need specified, stable, auto-tested core lib features. I've lost faith in Node.js to be suitable for high-importance production applications. Standard Web APIs (e.g. web streams) give some hope, but they're currently slow and generally backed by the Node.js streams so potentially have the same issues. When we've opened issues and PRs around documentation vs. behavior discrepancies, it seems like no one knows anymore how streams should work in detail, and there's no one taking charge to change that. Maybe bun is the future?
- Streams API is not intuitive. When working with asynchronous functions (eg ETL) is a lot harder to get it right when comparing to golang channels.
- Stuck with async and lopp async 😭
- Submodules version dependencies create technical debt for older apps
- Tests are amazing but needs more features like in other libraries
- Thankfully, no. I'm really enjoyed with quality. Keep going!
- The ESM transition has been our painful python 2->3 moment. Maintaining async state can be messy.
- The lack of builtins for creating and consumming Server-Sent Events.
- The only issue is a social issue where old developers don't like JavaScript & pick technologies that they're more familiar with.
- The ovservable pattern can be sometimes a pain
- The thing that missing is full integration with typescript. Or a guidance of how no JS runs with typescript. Maybe this is more applicable for typescript side
- The time it takes for new V8 versions (with desired new features) to reach LTS versions.
- Too frequent dependency updates & security alerts in library packages because they shipped a CLI tool alongside it
- Too many versions... less, more full releases would be better

- Troubleshooting the app and runtime are just gross. I tend to use webdev tools in chrome - but I have developed libraries for better run time observability because the eco feels so detached.
- TS support
- tty issues in git bash
- types
- Types
- Types
- Types should have first class support.   Monorepo tooling should have first class support
- Types, I need at least basic typing support because it's hard to maintain large applications.
- Typescript + ESM situation is a hell
- Typescript and performances
- Typescript compilation
- typescript getting in the way.
- Typescript should be embedded or at least be enabled as first class citizen.
- TypeScript support still seems patchy. Error messages can be archaic.
- U
- undefined is not a function
- Undocumented changes - rare, but annoying when it happens.
- Unhelpful error traces, especially on async code Low level stdlib that leads to relying on external packages for almost anything
- Updating multiple packages at the same time
- updating Node.js version while keeping all global deps (via NVM)   slow startup time for CLI commands (esp. if they are written in TS and need --loader ts-node/esm/transpile-only
- Verbose standard error messages. Unhelpful error messages without a stacktrace.
- Version Based development, I wish I can develop and switch all lts version
- Want to forget the commonjs fro forever
- what I described above
- Whenever I use stream APIs... I just don't like them very much... Also, the binary size. But I'm not sure that I can help this and I think that this is adequate, I just did not realize it was this big before.
- Whenever packages do their own stuff and bypass the server in .npmrc, and in internal networks where the server mentioned in package-lock.json is not reachable (but another proxy serving packages with the same hash)
- wishing to have typescript "out of the box"
- With nvm somethimes, when change node version the process of loading modules by npm crashes. I resolve the bug changes some paths or moving files, when runs I revert the changes and goes well!
- Workers, CPU and RAM usage after long time
- working with commonjs and ESM always gives the error "cannot use import"   Also, sometimes node processes die in production for lack of memory and there isnt a clear error message for explaing what happened

- working with sockets is a nightmare
- Would be great to have a deno deploy for Node to launch small APIs & stuff
- Would be nice to have more unification like deno, to not need a base project to create a new one fast
- Would like to have better sockets support
- Would really appreciate naive typescript support and easier project setup
- Yep, especially when figuring out how to work with dozens of module import/export approaches. Mixing them together when   migrating to TypeScript or just to a new module system.   I wish there were an ultimate consolidated solution to all these problems.
- yes, documentation is hard to follow
- Yes, fellow developers forget to ‘await’ function calls, which makes it hard to debug sometimes.
- Yes, I think we don't have a vision or template with a really good modern production configured applications or standard which libs we should to use for improve our community of develops!
- Yes, importing CommonJS from ESM
- Yes, it's very slow and sometimes heavily consuming for the CPU
- Yes, javascript is painful to write in. I wish it wasn't the language of the web.
- Yes, the evens for socket streams are a nightmare of complexity, with docs too spread out, and actually logic for some basic operations borderline insane. Also, debuggability is incredibly poor.
- Yes, when using packages there is poor understanding of the mode in which node.js will run my script. If we could have .noderc configuration to be more specific about runtime configuration and ways to process files - would be way more easier
- Yes. Sometimes, when trying to use npm, it doesn't work
- You need a million packages to develop an api