Q3 - What type of organization do you work with? - Other

Academy of Science
between positions
Bootcamp
Charity
Community
Company, startup, individual
Consulting
contractor, mostly Public Sector/enterprise clients
Corporate consultant
Digital Agency
Digital Studio
education center
Education Institution
Factory
Federal Research Lab
Entertainment
Freelancer
Freelancer
Freelancer
Government
Government
Government
Government
Government
Government
government
Government
Government
Government
Government
Government
Education
Company
Freelancer
Startup
Independent contractor
Individual & national health governing body
individual + school
industry
Learning to code

Media
military
News organization
NGO
no organization or job yet
Non-profit NGO
Nonprofit
Nonprofit
Open Finance
Open Source
Open source project
OSS
OSS
Own business
own consultant, working for government
Private for Family
Public Body
Research institute
Scale up
scaleup
Scaleup
School
school
Self taught
Services based company
Small Business
Company
Startup, Consultancy, Company, Individual, Open Source
State Government
student
Teleperformance
Tourism
University, Government, Company
Warner Bros
We have clients in all levels, from startup to fortune 100.
Worker Cooperative

Q4 - If working in a company, in which sector does your company operate?

apparel EC
A/B Testing
Ad Agency
Ad tech
adtech
Advertising
advertising
Advertising
Advertising
Aerospace
Agricultural technology
Agriculture
Agriculture
Agriculture
Agriculture
Agriculture
Agriculture / Climate
Agro
AI
AI
Airline
Analytics
Animal Healthcare
Art gallery
Arts
Augmented Reality
Automotive
Automotive
automotive industry
Backend focused
Beauty
Beauty Cosmetics
Biotech
Biotech
blockchain
Branding agency
Building
Business
Climate
Clothing brand
Commerce
Compliance
Computer Games

Computer science
Computerized Maintenance Management System
Construction
Consultancy
Consultancy
Consultant
Consulting
Consulting
Consulting
Consulting
consulting
data broker
data exchange
Delivery
Digital Branding
Digital signage
Discount services & retail
Distributor
E commerce
E-com
e-commerce
E-commerce
E-commerce
e-commerce
E-commerce
e-commerce
E-commerce
Earth Observation
Ecommerce
Ecommerce
ecommerce
eCommerce
ecommerce
Ecommerce
ECommerce
eCommerce
Ecommerce food
Edtech
EdTech and Payments
Education
Education
Education
Education
Education

Education
education
Education
Education
education
Education
education
Education
Education
Education
Education
Education
Education
Education
Education
Education
Education (JKU)
Education / Non-profit
Education, edtech
Education, Legal Tech, Food, Sports
Electricity Wholesale
Engineering
Engineering
Entertainment
Entertainment
Entertainment
Entertainment
Entertainment
entertainment
Entertainment
Entertainment Services
Entertainment/Consumer Services
Environmental Protection
ERP
Events & 3D interactives simulations
Fashion Industry and Photography
Final Mile Logistics
Finance
Fitness
Food
Food delivery service
Food Service Software
Gambling
gambling

Game Server
GameDev
Gamedev
Games
Games
Games
Gaming
Gaming
Gaming
gaming/blockchain
government
Government/Social Services
Grocery
Health & Safety
Healthcare Applications/Tech
Hospitality
Hospitality
Hospitality
Hospitality
Hospitality
Hospitality
HR
HR
Human Resources
Human Resources
Human rights
Various
iGaming
Inbound Marketing
Independent
Insurance
Insurance
Insurance
Insurance
insurance
Insurance
Insurance
Insurance
Insurance
Intellectual Property
IOT
IoT
IT Consultancy
Knowledge Management

Law
Law enforcement
Law enforcement analytics
Legit all these things too damm big haha
Live streaming
Local governments
Logistics
logistics
Logistics
Logistics
Logistics
Logistics
Logistics
Logistics
logistics
Logistics
logistics
Logistics
Logistics
Logistics & Retail
Luxury
Manufacturing
Market Research
Marketplace
Martech
Media
Media
Media
Media
Media industry
military
military
Mining/resources
Motorcycle Rental Business
Multi Sports & Hospitality
Music
Music
Music production

Navigation / Physical Address
News
News Publication
News service
NGO support

Non Profit
Non profit advocacy
Online Gambling
Online Travel
Our clients are in virtually every industry.
outsource
Outsource
Outsource
outsource
outsourcing (all above)
politics
Process R&D
Professional services
public services
Purchasing, Construction
Recruiting
regulatory body
renewable resources/energy
Rental accommodation
Research & Development
Retail
Retail
Retail
Retail
Retail
Retail
Retail
Retail
Retail
Retail
Retail
Reuse & recycle
Robotics
Saas
sales
Science
Security
Semiconductors
Services
Several
Shipping
ShortPoint Inc. | Intranet Pages Builder
Sleep
Smart City

Smart home device OEM
Social
Social media
Software
Software
Software house
Softwares for Vacation Rental
Sport
sport evernts
Sports
Statistics
Study abroad
Talent Vetting
Telecom
Tourism
Tourism
Tourism
toy
Toy
Transport
Transport
Transportation
Transportation
Transportation Mobility
Travel
travel
Travel
travel
Travel
Travel
Travel
Travel
Travel Tech
Uni Projects
US Department of Defense
Waste management
Web agency
Web designer full stack
Web dev
web development
Web3
Windows & Doors

Q5 - Which of the following best reflects your role regarding Node.js?

1-4 equally
All but Node Core and Investments
All of the above (Sans core contributor)
Application developer & Library & package author
Application developer AND Library & package author
Application developers and library & package authors
back end developer
Compiler developer targeting JavaScript
Desktop Application
DevRel
Full stack web developer
Full Stack. Library, Dev-Ops
I'm an app developer and open source library author
Learning JavaScript
Marketing
Multiple - Author Libs and Packagers, App Developers and operators
Proxy
Software Tests and analysis
Solution Architect
Sometimes I am fulfilling multiple, if not all the above, roles regarding Node.js, per contract
Student
Startup
Test Automation Engineer
Tinkerer; local front-end dev & design
Application operators, application developers, library & package authors
Webdev

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

Development of APIs with Microservices including extensive communication via MessageBrokers
1,6,7 + classic backend SSR apps
All
all
all above in regards to backend
All cases
All of the above

All of the above
All of the above
All of the above
All of the above
All of the above
All of the above
all of the above
All of the above
All of the above
all of the above
All of the above
All of the above (except proxy)
All of them above
All of these.
all set of above except proxy
APIs with Serverless and Socket based applications
at least 5 points from list above
Automation and glue using cloud-native tools like AWS Lambda
Back end applications
Back-end operations
Back-end Servers Development
Backend infrastructure
Backend Processes
both static frontend and backend with Express.js
Building a web server
Building embedded apps
Building front end applications with meta frameworks (Nuxt, Next.js, etc.)
Building peer-to-peer distributed systems
Business critical applications and application servers.
Can be some, or all, of the above
Cordova
deploying front end applications & script+automation
Deploying front end applications and Script and automation
Desktop application with node embedded
Developing APIs and Discord bots
Developing front (serverless, static) and back end apps
Developing utilities for everyone to use
Development of a good old monolith with server rendered templates
Development of APIs serverless, Development of APIs with microservices, CLI tools
Development of APIs with Dynolith
Development of APIs with Server
Development of backend and frontend (equally)
development of database driver
Development of PhotoStructure

electron
Empowering people to own their own place on the web and seize the means of communication to protect their freedom of speech and privacy.
estudos
Everything
Everything above
everything above except proxy
FE apps, BE APIs, scripts and automation
Front end applications and servers. Deployed on VPS and serverless
full stack
Full stack app development
Full stack development
Full stack development (no discernable bias)
Full stack front end and back end
Full stack, backend node API and front end web components
Full Stack: back-end (APIs, server-side), front-end
Full stack: frontend and backend applications
Full stack: server runtime and building the frontend
Full web application stack
Full-Stack JavaScript Development.
Full-stack web app and api
Full-Stack web development
Fullstack apps, MERN
Fullstack developer
fullstack node express react oracle mongo kafka and everything in the middle :(
Fullstack node.js and frontend application
fullstack, no specific specialization from the above, a bit of everything
Fullstack: ReactJS front end applications, and server side systems
fun
Game Development
Hybrid app development (based on nw.js framework)
I use it for everything

I've been working on a relay connector for (eg. IPFS), that establishes secure routing for people behind firewalls or inside a subnet. I set up a RDNIS device that helps to contain my audio routing across ipv4, and integrates nicely with microsoft's pseudo teredo tunneling adapter, and that's alongside other mesh style connectors to try and force quieter bluetooth behavior. There's also some heavy port swapping across the edge, which makes it extremely hard to identify, even with cloudflare integration. :-D

In house server-side rendering framework for React apps
IoT
IoT, Building dynamic front-end and back-end applications

Just for fun
Learning
Libraries
Library authoring + Devops for Node.js applications
Make websites
Making websites with Express
Many from the list
Many of the options above
maybe All
monolit application
Monolith
monolithic
monorepos (many packages)
Most of the above equally

Most of the above

Multiplatform apps
Multiple - API Devs, Static Front End Dev, Microservices, serverless, Testing, Deploying, CLI, Automation
multiple of the above
Nearly all
Nearly every use case listed here
NestJS
NextJS, but also bots and APIs
NLP
Node js
Packages
Robots and video games
Sending millions of requests to ddos websites
Single-Page Applications, this did not appear to be included in "static front end"
Support of server-based Javascript processing
Time Pass
Tracing npm modules
typings and discord bots
Web Apps and VSCode Extensions
Web dev in general (server-based MPAs and front end development)
Web Server on myself cloud server, cli, API with Serverless, proxy
whole enterprise application
Writing game servers and website backends
Writing website projects that require a backend and I want it to be in JS

Q7 - What operating system is your local development environment?

All
Android
android
Android
Android
Android
Android 13
Android/windows
Chrome
chromebook
Develop on MacOS, Windows and WSL
FreeBSD
GNU/Linux and macOS
Linux and Mac
linux for personal + macos for work
Linux on remote workspace
Linux with Docker and Windows with WSL (50-50)
Linux with Podman
Linux, Windows and macOS
macOS, macOS with Docker, Linux, and Linux with Docker
Mainly Windows w/ WSL, although sometimes docker.
multiple of the above
ReactOS
Windows + macOs
Windows and MacOS
Windows and macOS
Windows with docker and wsl
Windows with virtual box
Windows with WSL and Docker
Windows with WSL, Linux servers and macOS
Windows, Linux

Q8 - What operating system is your production environment?

All

all
All
all of above
all of the above
All of the above
All of the above (cross-platform desktop app)
Android
Android
Android
Android
Android
Android
Android
any, runs in a browser
AWS
AWS Cloudfront/S3
AWS lambda
AWS Lambda
AWS serveless
Both Windows and Linux including occasional Docker instances
Chromebook
Dual boot Windows and Linux
Everywhere people run tsc/tsserver, so all of the above
Firebase
FreeBSD
Google app engine
Google Cloud Run (Docker)
iOS, Android WebView apps, React Native Apps
is windows but sometimes LInux(RPi os ot Ubuntu)
kubernetes
kubernetes
kubernetes, serverless, Linux
Lambda
Lambda, ECS with Docker
Linux
Linux Azure
Linux Kubernetes
Linux with and without docker, as well as OS used by GCP cloud functions and/or AWS Lambdas use
Linux with docker and AWS Lambda
Linux with K8S
Linux with Kubernetes
Linux with Podman
Linux, Windows, macOS

Mobile
Multiple
multiple of the above
None at the moment.
Not using for backend.
Our CLI runs on Mac, win, linux
ReactOS
Serverless (Vercel)
Serverless + React
serverless lambda
Serverless lambda
Serverless on AWS
That varies Linux/Unix but sometimes with or without Docker, in rare situations also Windows without Docker
Ubuntu
Ubuntu
various
Vercel
vercel
vercel/netlify
Whatever the users of my libraries are deploying on
Windows and Linux both
Windows and MacOS
Windows with WSL2
Windows, Linux, Docker
Windows, macOS and GNU/Linux

Q9 - What is your architecture in which you are running Node.js?

Aarch64
All
all desktop
All of the above!
Android 13
Apple M chips
Apple Silicon M3 (I think it is arm based, but not sure)
arm local, x64 deploy
arm, x64
Arm. I use this runtime on raspberry pi 5
armv8, x86_64
Azure (I don`t know the hardware specs)

Browser env
Depends
Everywhere people run tsc/tsserver, so all of the above
various
x62 & arm
x64 (x86) and arm64
x64 & arm/aarch64
X64 + arm
X64 and arm
x64 and arm
x64 and ARM
x64 and arm
x64 and arm64
x64 and arm64
x64 for dev, x64 + arm64 fro prod
x64 local, arm prod
x64, arm
x64, but you asked about the next 10 years... planning to switch to arm in production
x86
X86 and arm
x86-64 and aarm
X86, Arm, SiFive

Q10 - How do you get your Node executables?

Ansible, which downloads binaries directly
asdf
asdf
asdf,nix
Author
aws lambda
Brew + nvm + deb packages
bun
bun
bun
Company manages it, it downloads automatically.
Depends, sometimes official installer, sometimes package managers
devbox
Direct download for Windows dev, apt for live
Docker
Docker
docker

Docker
Docker
Docker
Docker
docker container
Docker Hub
docker image
Docker image
Docker image
Docker image
Docker image
docker image
Docker Images
Docker registry
dockerhub for prod
fnm
fnm
from Docker Hub
gitpod
Homebrew
Included in workspace image
mise
mise
mise (former rtx-cli) asdf rust rewrite
mise or asdf
mostly winget, sometimes official installer
Multiple ways but mostly using prebuilt containers with Node installed
nix
nix / nvm
nix flake
nix packages
Node.js Docker image
Nodesource
Nvm, docker image
Official installer for windows, Apt for Linux
pnpm env
pnpm env command
podman images
Using docker image
Using nodemon
vercel node downloader
Volta
winget
FNM

Q11 - Which package manager do you use?

all (except for cnpm)
All of them above
All
both npm and pnpm
bun
bun
bun
bun
bun
bun
bun
bun
bun
Bun
bun
Bun
bun
Bun
bun
bun
bun
bun
bun
bun
bun
bun
bun
bun
bun
bun
bun
Bun
bun
Bun
Bun
bun
bun
bun
Bun

bun - primary, pnpm - secondary
bun 🙈 but also yarn1/3 and pnpm
bun install
espresso
git or none
i dont know
I dont know
I dont know
Mainly NPM, but it can depend if I'm working with people who insist on using yarn, PNPM, etc, etc...
multiple of the above
npm & pnpm
npm & pnpm
npm and bun
npm and yarn
npm via bun
npm, bun
npm, pnpm
Npm, yarn
npm, yarn v1, pnpm
npm, yarn v1, pnpm
NPM, yarn v1, pnpm depending on project
Npm, yarn, bun
npx
pnpm & bun
pnpm and bun
pnpm and bun, depending on project
rush
yarn
Yarn strict
yarn v4
yarn v4
yarn v4
yarn v4
Yarn v4
yarn v4
yarn v4
yarn v4
yarn v4
yarn v4
yarn v4
Yarn v4
yarn v4
Yarn v4

yarn v4
yarn v4
yarn v4
yarn v4
yarn v4
yarn v4
Yarn4.1.1

Q12 - Which version manager do you use?

pnpm
pnpm
pnpm
pnpm
pnpm
pnpm
pnpm
pnpm env
pnpm env
Pnpm env
pnpm env
pnpm env
pnpm env cli
proto
proto
proto
scoop
snap
vfox
vfox
winget

Q13 - How do you manage the package manager for your project?

.nvmrc file
.tools-version or .nvmrc locally. or config file from serverless or container for prod. I should probably just set and env var for this.
All of the above
all of the above
Android

All of above
Depending on which package manager I'll either use option (a) or (b)
depends on project
direnv
fnm allows me to use the npm associated with the node version specified in the .nvmrc file of the current project
https://github.com/bazelbuild/rules_nodejs

I always create a new lxd container and do all my node installations within. Why? Because when nothing works, then I can just delete the container without having to worry about 100,000 extra files that get installed into hidden folders every time I invoke npm on the command line.
I don't

I get the current LTS version of node js and the supporting version of npm whenever I start a new project.
I install locally per project
nvm
N
I try to not use a package manager, for security validation reasons.
I use a custom command to switch to the node version specified in .nvmrc with nvm
I use built in npm version corresponding project's node.js version
I use NPM by choice but Yarn V3 if client uses it.
I use one version installed globally for all my projects, with asdf for occasional legacy projects
I use one version installed globally for my all projects, but I also use nvm for some specific projects.
NPM
Install locally for every project (As every project I have require diferent versions)
Installing in different project folder everytime
Just set the engine property in the package file and change to the correct version of node when working on a project.
Locally installed with small .bat file redirects
manually
manually using NVM
maven-download plugin
mise
Mise
mvn
nix
nix dev flakes
nixos
No
None/NA
not sure
nothing

Npm
nvm
nvm
nvm
nvm, github/vallyian/nuse
Other (please specify)
pnpm
pnpm env
rush
rush self-manages itself
NPM
Try to use the latest version as much as possible
two version installed globally
use lts version or node-sass strongly dependent versions(e.g. node14)
varies per repository
Volta does this for me
We use nvm, and whatever version of npm is bundled with the version of node in the .nvmrc
yarn checked into the source code
Yarn has a way to install the version in the project directory itself
Yarn is installed in the repo

Q14 - Which of the current technical priorities are important to you? Select all that apply.

A standard library in order to avoid too much dependencies
Android
asyc/await based libraries
Better cold-start
better out of the box typescript support, better commonjs and esm support in same file/project,
Better tools around finding performance bottle necks like high CPU usage, High Memory usage or thrashing
Building a Wallet
Built in linter, formatting and types (Like Go/Deno) VS Code App
Built in support to typescript
C++ Addon
CJS/ESM interop, native TS support
CJS/ESM interoperability (to support migration of packages)
Cold start, deployment size
compatibility
Direct TypeScript support

easy to learn as a frontend dev
ESM / CJS interop, as a Nest.js user using ESM is very painful
Facilitate debugging. Someway to get up debuing a nodejs app quickly.
Faster builds in CI/CD
Faster startup time
Graphical Development
GraphQL / solving the nodejs/npm/global installed packages versioning mixture when developing in a week in multiple application folders
gRPC
help replacing deprecated dependencies
HTTP/3
I need to get some effective security policies in place for Windows, since it does have a pretty advanced defense posture.
Improve ESM and CJS interoperability.
Improved AddOns support for modern C++ in node-gyp (modules, concepts,..). Make N-API easier for other compiled languages, still a bit too much DYI integration.
Integrade Unit Test envinronment
intl
JS Compiler
JSON5 support for package.json
Keeping the API SMALL and SIMPLE and avoiding bloat
Low app bootstrap cost
Low cold start time (for serverless)
Machine Learning
make self version manager built it
Making ESM / CJS stuff not a mess
merged CJS and ESM syntax
monorepo to organize code
More Powerful API
multi-threading performance improvements for cpu-intensive tasks.
native C++ addons
Native TypeScript support
native typescript support
Native TypeScript support and assorted tooling
Number of concurrent requests a NodeJS server can handle.
Other (please specify)
performance
Performance
Performance
Performance
Performance
performance
performance
Performance

performance and native easy monorepo setup like in bun.js
performance comparing to golang
performance, speed of development
Performance!
Pure performance
RAM consumption
Reduced cold start time
Sandboxing
Security Update
Self-containment: still easily runnable in 10+ years
Single threaded performance
Small cold starts for Lambda
Speed of execution
Stability
Stability - fewer breaking changes.
stable apis and slower release cycle.
support typescript
Support for running typescript code without transpilation (- decorators)
support for running TypeScript programs directly
System integration (display a window, run pseudo bash)
Testing
Thread support
TLS and SSL editing
Typescript
TypeScript
typescript
Typescript
TypeScript
TypeScript
Typescript
TypeScript
Typescript ootb support, performances
Typescript Support
Typescript Support
Typescript support
TypeScript support
Typescript support
Typescript support
VM
Web APIs
Websocket server
WebSocket Support, including Servers, similar to the ws NPM package
websockets and webtransport
WebTransport + simpler installation (one-file like Deno)

Windows integration
Workers

Q15 - What is important to you? Select all that apply.

An expanded standard library
An official package manager that fixes the problems `npm` currently has. I think `pnpm` is a good candidate and should be the de-facto package manager

Backward compatibility & long-term stability

Better support for modern standard JS development practices, such as TypeScript
break the 4Gb limit and make node a contender to Java in term of reliability / tweakings + also eat Python

built-in TypeScript support, better npm experience
Bun
CJS as a first-class citizen even as ESM joins the party
code reusable and maintainable
compatibility
Compatibility
Creating a single executable file which will be light footprint
Ensure ongoing C++ usage is safe, easier integration of other safer system languages in end-to-end story.
Faster builds
faster runtime and npm
Fully compatible web standards implemented (I don't to use ESM or another standards a second-class citizen, web standard first please)
Guidance on how to write for and properly bubble up errors
HTTP/3
Language Features
Less bloat
Linter & Prettier to have a standard of how code should look for every project.
Memory management
Memory safety and the ability to easily debug memory issues
Modern Web API
More alignment with bun and deno pushing the boundaries forward. Module Federation pattern should be experimented more and pushed and made easier to implement.
Move away from primordials
Native reliable coverage reports for tests
Native support for typescript
Native TypeScript support and assorted tooling

Standard and JS/ES, no CJS MJS…

NodeJS server load performance. Can it handle 100k concurrent requests or not? This often time becomes a question for whether or not to use something else like Go.

Office NodeJS app for VSCode (Linting etc like Go/Deno)
Packaging modules, right now it's quite hard to do it right with ESM, CJS, TypeScript etc...
Performance
Performance
Performance
performance
Performance
Performance and easy to work with
Performance/Speed
Provide compiled binary programs for multiple platforms
QUIC Support
Regardling the ability to embed and bundle the Node.js runtime, having a predictable URL structure (as currently exists) means we can do this at install time. So please don't change that. Although reduction in binary sizes would be very welcome :)
Stability of C interface
support typescript
typescript
typescript first-class citizen
TypeScript support
Typescript support
Typescript, monorepo
versions manage
Web browser-compatible APIs

Q17 For those of you wishing to use ESM in an existing application, what have been the pain points or blockers preventing you from doing so (if any)?

no any
__dirname __filename, node18 is alive but not fully supporting ESM
__filename __dirname
- Could not `require` ESM (which was fixed recently) - TypeScript support for ESM is bad (Have to specify extensions, moduleResolution breaks a lot of projects) - Frameworks are not ready (e.g. Next.js) - Some faux ESM modules which does not provide named exports properly (e.g. emotion)

- jest - problems with file extensions in Typescript
- legacy dependencies which are not using esm
- No support for absolute paths (had to write custom loader) - Importing CJS packages that manipulate imports (ex. Winston, EventEmitter2) - Adding .js to all imports (should have a codemod) - Jest to Vitest migration
- Top-level await - require and import in nodejs are not compatible - __dirname dosent exist if I use ESM in node.js
- Transitioning to hybrid (when updating an existing CJS project) - Trying to get 2 or more projects written in either format to work well together

.js is a must in imports, that confusion that this work without .js in some frameworks like next.js / vite but doesn’t in esm node
"type": module..... should be the default not CommonJS
`.js` extention required during esm import
1. It is not supported by all libraries and frameworks we use 2. Need to spend time to adjust imports to add the trailing ".js" extension 3. CJS -> ESM compatibility through "await import"
1. Need extra variables after import
1. The amount of CommonJS files in the project. 2. Differences between `import()` and `require()`, mainly that `require()` just returns whatever is exported from there, and that `import()` returns an intermediary
3rd party libraries being incompatible (i.e. jest, etc.) Issues with importing esm in CJ's.
3rd party packages still using commonjs
A bunch of fairly core tools don't seem to support ESM nicely, at least not out of the box (e.g. jest, eslint, sometimes tsc on a bad day). Often these errors are fairly odd and poorly documented, and every time I've tried this I've got fed up and gone back to CJS.
A lot of documentation is still focused on CommonJS first. Jest still has no ESM support.
A lot of libraries doesn't support new way of importing and some of them are broken because of this.
Ability to do incremental migration - use both ESM & CJS in the same project, it's release in 22 .. waiting for it to be LTS and start migrating slowly to ESM.
Ability to mix CJS and ESM, and which file format to use in which situation with a mix of tools working together.
Added complexity configuring transpilation
adding .js to all relative module imports.   packaging application, now we don't package the applications as executable because of ESM but want.   In general, our switch to ESM was because of one ESM dependency and inability to load ESM from common.
Adding ESM in a project that's already using CJS. Interoperability between ESM and CJS
Adding in configuration to support it. Old-style modules not supporting ESM.
Additional configuration that needs to apply to dev/prod and test modules separately. It should be native with 0 additional configuration like in bun.js
adoption in community npm tools and packages
After an upgrade, we were able to do it, using ts-node.
Age of the runtime
all of it
All the various configuration options especially in combination with TypeScript

already using ESM via typescript
Already using it, but it's too fiddly and requires the use of a bundler.
Amount of existing code
An old project that uses vue2 is a pain to migrate. The vue2 usage for a frontend makes it challenging to migrate certain backend features.
Any
any
anyone good
apk
As a developer of frameworks and libraries, I hope that Node official can provide a clear direction, not be mediocre. The current ESM ecosystem compatibility is too poor, we need to consider how the existing applications and library ecosystems continue to evolve. Otherwise for library developers, they can only be forced to write in TS, and then compile into both CJS and ESM at the same time, but this is actually not good, it's not just about package size issues, it also makes things confusing.
As somebody who developed in NodeJS then worked on Elixir and came back to NodeJS, the schism of ESM and CommonJS is super confusing and its relationship to package.json's `module` attribute seems to preclude unsupported Node.JS dependencies from being installed or working, which makes the entire ecosystem so much more confusing than it was in years past. Occasionally I'll install a package only to find it's not supported because there isn't ESM support.
Async dynamic imports, cjs <-> esm, module cache reset, path resolving
Attempting to use ESM-only packages from CommonJS.
Autocomplete imports. Maybe a syntax like this would help: from 'x' import { y }
backward compatibility with some packages
Backwards compatibility for existing lib, clear why how to use esm with TypeScript
Backwards compatibility with old packages.
Big bang conversion requirements and complexity of intertwined package.json vs other config requirements
Big refactors when moving from "require" to "import".
Browser compatibility + Module system integration Integrating ESM with existing module systems like CommonJS or AMD can be complex
Browser compatibility
Build process was set up in CJS and no one has tried to upgrade because so far CJS has caused no problems and all the synchronous requires in the code base just seem like a mess to upgrade.
Build tooling configuration
bundlers config to support commonjs dependancies and sub-dependancies
Cache control
Can not simply and directly run the code. The community (like other libraries) does not take extra consideration of ESM
Can't mix require/import.
Cannot be mixed with CJS
Cannot load addon (*.node) with import syntax.

Cannot mix require() and import, both have entirely different syntax as well (export vs module.exports) so end up having to reformat all my code and potentially any libraries I am using
Certain tools don't support ESM out of the box or at all (example: Jest) and require the project to be transpiled each time which makes running tests slow. Granted we have switched to Vitest and it works great.
Cjs
CJS
CJS
cjs & esm compatability & painful migration
Cjs and having to support legacy browsers until ~2y, plus our components/islands are consumed in dozens and dozens of apps that update at different cadence
Cjs compat is complicated
CJS ESM incompatibility
CJS interoperability
cjs libraries
CJS library support
cjs libs
cjs users until recently could not consume esm
CJS/ESM interop nightmare
cjs/mjs interop is not good enough, using type: module breaks things sometimes though it has been better lately. I think it should just work (easier said than done I know)
combiming csm and esm modules
Combination of ESM and CJS dependencies in a ESM TypeScript project is not always working properly.
combination of TypeScript, jest, ts-node for running tests is not ideal, when I want to add some ESM-only dependency. Right now, I'm locked to some older versions of modules that went ESM-only.
common js
common js/ESM interop is painful
commonjs + packages that require TypeScript in monorepos
CommonJS and ESM interoperability can be tricky
commonjs compatibility
CommonJS integration still painful. Old npm packages moving to esm breaks backward compatibility, which is hard as a library maintainer
CommonJS interoperability
CommonJS interoperability, especially require(esm). Lack of clear communication from Node.js team that CJS should be considered legacy, and focus should be on ESM-only future.
CommonJS interp
commonjs syntax
CommonJS tooling and/or libraries with half baked ESM support
commonjs vs esm support on same file/project
Compability wirh tests frameworks as jest

Comparability.   I strongly think there needs to be a "change over date" when CJS is marked deprecated
Compatability with commonjs imports. You either need to ensure everything is esm, or nothing is.
compatibility
Compatibility
Compatibility
Compatibility
Compatibility for some existing NPM packages.
compatibility issues between cjs / esm
Compatibility issues with many packages, some of them ending up unmaintained
Compatibility issues with third-party npm packages
compatibility of esm with node versions, unclear errors.
Compatibility of the runtime, eg lambdas
Compatibility when importing old packages
Compatibility with CJS
Compatibility with CJS, a lot of errors come from mixed usage of both, sadly most libs already fully switched (looking at you sindresorhus) to ESM so some wont be able to update the libs as long as full support is not there.
Compatibility with CommonJS
Compatibility with CommonJS Modules: Integrating ESM with existing CommonJS modules can be challenging, especially when dealing with modules that rely heavily on Node.js-specific features like require() and module.exports. This can lead to compatibility issues and require careful migration strategies.   Module Resolution: ESM introduces a different module resolution mechanism compared to CommonJS. While ESM offers more flexibility with module paths, it can sometimes lead to confusion and unexpected behavior, particularly when resolving modules across different directories or third-party packages.   File Extensions: ESM requires file extensions (e.g., .js, .mjs) for module imports, which can be cumbersome, especially when working with a large codebase or migrating from CommonJS, where file extensions are optional.   Tooling Support: Some Node.js tools and libraries may not fully support ESM, or they may require additional configuration to work seamlessly with ESM. This includes build tools, testing frameworks, and third-party libraries.   Node.js Core Modules: While Node.js core modules have been gradually updated to support ESM, certain core modules and functional concepts are not fully compatible with ESM, requiring workarounds or alternative approaches that impede efficiency.   Third-Party Packages: Not all third-party packages have migrated to ESM, which can be a significant barrier when building applications that rely on external dependencies. Developers may need to resort to workarounds such as using CJS interop or finding alternative ESM-compatible packages.   Debugging: Debugging ESM code can be challenging, especially when dealing with runtime errors related to module loading and resolution. Tools like node --experimental-modules offer some debugging support, but it may not be as robust as debugging CommonJS modules.   Documentation and Community Support: As ESM adoption continues to grow, the availability of comprehensive documentation and community support may vary across different tools and libraries. Developers may need to rely on experimental features or undocumented APIs, leading to uncertainty and potential maintenance issues.   Performance

Overhead: ESM may introduce a performance overhead compared to CommonJS, particularly during module loading and resolution. While this overhead may be negligible for most applications, it can become more noticeable in performance-sensitive scenarios or large codebases.   Learning Curve: Switching to ESM requires developers to familiarize themselves with new syntax and best practices, especially if they have been primarily working with CommonJS. This learning curve can slow down development initially and may require additional training or resources.
compatibility with dependencies
Compatibility with legacy packages. Having to write "type": "module" in package.json.
Compatibility with libraries that don't support ESM
Compatibility with npm packages and in my case ts-node to run TS in dev environment
compatibility with pkgs
Compatibility with testing lib like jest and other third party dependencies
compatibility with tooling: typescript, bundlers, linters, jest
compatible issue. Bun has a better solution than node
Compatible packages
Complexity of setup for authoring libraries. Complexity when using packages from the public registry.
configuration and boilerplate
Configuration
Configuration problems in order to set up ESM correctly in pure JS apps
Configure typescript
Conflicts with CJS modules
Confusion on how to use packages that doesnt support it
Consistency.
Converting CJS components for use in ESM projects and ensuring ESM module backwards compatibility
Coordination of all libraries/dependencies together (libs, TypeScript, our code, ...).
Cost of migrating big codebase from CJS to ESM.
Couldn't use require before node 22. Sometimes theres a frontend bundler that gives the js additional features that the bundler is able to do that node cant so need to rewrite it often if not using typescript.
Coworkers that are clueless about the difference
Creation of hybrid packages.   Using it in general.
Cross cjs/mjs usage
cross compat between cjs & esm, with tools like webpack and whatnot
Cross compatibility issues with ESM and non-ESM code. Also, janky requirements for imports when exporting ESM from TypeScript... like having to include file extensions in the import.
cross compatibility of JS code between web browsers in various (partley older) versions and NodeJS.
Cross-compatibility with third-party libraries in CommonJS
CSM interop and general lack of clarity around what will work
Custom loaders should not require extra files (for the loader). An app should be able to start with: "node index.js"   and from there it should be able to load whatever and however it

wants. Sync and Async. That is only possible with CommonJS modules and monkey patching. Sadly.   Thanks.
—------------------------------ Michael note, got to here

Cyclical dependencies, it was oir fault of poor design
Dealing with cjs libraries
Default exports
Definitely interoperability of ESM with CJS is a huge pain-point. Going all ESM was nogo for many years due to weird issues with all the tooling. That was very fragile. Current best fit for most was still CJS (after compiling with TS for example) as it was the least painful, but we missed some innovations. Some libraries went into ESM-only, which prevented us from using it. That was the status up until recently when Node 22 introduced experimental support for importing synchronous ESM into CJS, which is a huge deal. I can't imagine why such an obvious quality change, and rather simple one, took so long, but that should solve most of the issues.
Deleting import cache
Dependencies
Dependencies
Dependencies
dependencies that do not support ESM
dependencies that don't support it
Dependencies! Some dependencies support ESM, but not CJS and some support CJS, but don't work very well with ESM
Dependency compatibility, often around whether import extensions (".js") are required
Dependency on packages using common js
Depending on the import type, package root is slightly different, so projects sometimes have multiple "default" objects, i'd rather this be standardized
Different ecosystem tools which do not support ESM completely
Different module types and conflicts between those at runtime and as developer experience. Like having 3 different .js files in same project due to library requirements or application structure.
difficult configuration to use ESM in the whole project .i.e. it could work in some files and not in others and sometimes it can't run like normal js imports
Difficult to manage imports, mock them for testing proposes. Not native import
Difficulties with testing. Specifically a lack of support from Jest (it's unstable + issues with spying on exports)
don't know
Dot js in import paths in typescript
Down compilation steps
Dual module hazard, accidental inclusion of both ESM/CJS version of a package.
Dual package hazard
dynamic imports
Dynamic imports are always a gamble, and having to destructure ESM imports.
eabuild don’t support decoratirs

ecosystem
Ecosystem fragmentation, particularly with libraries supporting either CJS or ESM, but not both. Often I find myself in situations where some only support ESM and others only support CJS, which leaves me unable to intuitively use both
ecosystem support
Ecosystem support inconsistency
ecosystem variability in ESM support, especially TypeScript-compiled module compatibility with PureScript, Elm, ReasonML-compiled modules...
Electron
Electron maintainers refuse to update to esm
electron, vscode extensions
ERR_REQUIRE_ESM Top Level Await Synchronized Conditional Import
Error messages are very intransparent. Hard to tell if the issue is in typescript config, bundler version, node setup etc
eslint/autocomplete support
ESM <> CJS interoperability
ESM and CJS interop
ESM is not still widely supported so some public repositories still encounter bugs when running on ESM. There I never use ESM anymore, I tend to use the CJS module syntax
ESM is slower and not as flexible or ergonomic, and almost every advantage can be had with authoring in ESM while transpiling to CJS.
ESM is the JS standard (with ten 10 years) and it still a second-class citizen: you need to specify module in package.json, old packages are still using CJS because nodejs it is not forcing to use ESM so you need bundlers to help with this...
ESM Loaders
ESM or whatever, it is not my really concern, I juste want code works with my ES2015+ syntax (typescript)
ESM should be the default.
ESM support for the exports syntax is not standardised between implementations... Some don't like it when no `default` is present, others it doesn't matter. The ESM implementation docs are not clear
esm-cjs interop
Esm/cjs compat Ecosystem still have ton of cjs module
Especially importing JSON files directly
everyone using something different
Everything. From tooling to the runtime itself. Everyone have a different resolution and getting it work with Typescript with different node versions(esm or cjs)
Everything. It’s so hard to get a project with ESM set up and running with mixed tooling and library support.
Executable (using shebang) fails to run files without extension. Overall “require” is better integrated.
Existing libraries not migrated
Existing packages not adopting ESM because ESM isn't as widely adopted as it should be, causing packages not to adopt ESM.. etc.

experimental flags
External dependencies
External depends / can't import ESM in existing CJS projects for easier migration (node22 fixes this) / running a script and getting a "can't import in non module"... like bun just works in all these cases 😭
External libraries
Familiarity
faux modules, and tooling that lacks full esm support.
Fear and terror
File extension names
For my browser deps i use a bundler that supports ESM, and for node I'm typically resigned to just use the commonJS require syntax.
For those of you wishing to use ESM in an existing application, what have been the pain points or blockers preventing you from doing so (if any)?
Found best to move to bundling the service to reduce dependency graph needing to be resolved and load in production
fragmented workflows between ESM and CJS
Framework/library supoort typescript integration
Frameworks not adopting it (nestje)
Frameworks or test libraries not supporting it natively: jest, too many workarounds to make it work
Frameworks that don’t support it (fully), such as Next.js and Jest
Full support by development and test tools, such as Jest. Module mocking is especially hard with ESM.
Full support of ESM in Node.js & libraries
General confusion due to the mix of packages available with CJS vs ESM, and what I need to do to be compatible with ESM. Feels like a lot of confusion with very little benefit to my use-case.
Getting my other team members on board. Fixing the typescript build and npm workspaces.
Getting ts to work with esm in a npm workspace setup, and still want good testing and watch experience, is next to impossible with esm for some reason
Gradual integration from CJS
Ha! Everything.
Hard to mix with CommonJS
Hard to spend effort to convert all existing CJS codes to be ESM-compatible
hard to use due to commonjs dependencies
Hard to use ESM with legacy CJS
Having a (transitive) dependecy that still doesn‘t support esm
having dependencies demanding ESM while others will not work with ESM - catch 22
having to add .js to every import
Having to change all imports to include file extensions, having to change file extensions, having to import typescript files as “.js”, interop with CJS.
having to configure bundler
Having to do this bothers me the most: import pkg from 'package'   const { thing } = pkg

having to name all backend files .mjs
Having to set the package.json to type module to be able to use them correctly/ESM modules not importing with my editor (vscode) correctly for type hints.
Having to set the type: module in package.json and some packages not being prepared for ESM.
Having to specify file extensions, and consuming ESM-only packages/modules within CJS applications. The fact that ESM was so incredibly poorly implemented and that you fragmented the ecosystem.
Having to specify the module model beforehand
Having to use .mjs files.
Having to use different file suffix like .mjs .cjs, I don't want to know the difference. I just want top level await & import/export to work out of the box.
Having to use the flag --experimental-vm-modules to use jest testing library with ESM
Holy, a lot. Everything here is a mess frankly.
hosting/docker
I am using it already
I am using it but I am migrating some libraries to make them work
I cannot easily migrate project to esm, and cannot able to consume esm only pkgs to my non esm projects
I change all code von cjs to esm
i compile to cjs, feels easier right now
i did not really know . i live with framework
I don’t like the bifurcation between Node’s mjs/cjs approach and TypeScript/browser’s native ESM support
I faced usually "cannot use import/export outside the module"; this is constant irritant ;(
I guess the only real pain points these days are when older libraries need significant fiddling with, or a complete rewrite, in order to run as ES modules and the TS issues that arise from those...
I had a case when 2 ESM modules didn't work together from CJS project while using dynamic imports. Needed to rollback one of the modules back to CJS.
I have been using TypeScript which works great, but I remember having to specify the file extensions when importing when I was using `.js` files which was kind of annoying. I also ran into weird issues when creating a library package that worked for ESM users.
I hear that Google says import is slow and they recommend to keep using bundlers for Chrome. I wonder if the same performance overhead exists in Node.js.
I like using typescript, which can be a pain to setup properly. Otherwise using and deploying ESM would be far easier. Also third party documentation often uses CommonJS which is a drag
I may face challenges like tooling support, module interoperability, legacy code compatibility, third-party dependencies, debugging integration, learning curve, and migration strategy.
I mostly work in a Node.js development environment (Node-RED), this uses many complex packages at various levels. Migration from CJS to ESM CANNOT happen quickly. It must be possible to consume ESM from CJS without extensive async rewrites and must be possible to consume CJS modules from ESM.
I never used it

I only use ESM
I remember having a hard time trying to make a single executable app with pkg/nexe from esbuild bundle.
I use build tools and bundles that in many cases default to commons.
I use NestJS (Fastify afapter) for the most of my projects, and the last time I tried to migrate to ESM, some of its plugins such as Swagger Plugin for compile time API Docs generation was not compatible with ESM
I wish there's an ESM equivalent of `if (require.main === module)`   I'm developing a tool that can run in both command line and within another script, and since I'm using ESM's `import` statements, `require` doesn't exist, so `if (require.main === module)` can't be used.   I could use `if (process.argv[1].includes("filename.js"))`, but this condition can be fooled by renaming the other script to the same name, or even the directory name as well. `require.main` probably won't have this problem?   Now I'm using `child_process.execSync()` to ensure I'm dealing with only one way to run the tool, via command line. But this approach definitely can't make the tool distinguish itself as a standalone process, or as a child process to another script. This is assuming no additional command line arguments are given to `child_process.execSync()`. I've tried using `process.pid` and `process.ppid` on both the tool and the other script, but the ids doesn't match.   This is the ideal scenario if `require` exists: ```js // import statements export function main() {   // ... }   if (require.main === module) {   main() } ```
i work with legacy projects that currently does not support ESM
I would like to use ESM, but I have no hope of this working in Node, ever. I see .mjs is still a thing, so I'm not even trying to migrate.
I'd just followed along with the changes, but now I'm learning require() might have been better, and allowed more cool things?
I'm a big advocate for ESM and exclusively use it in personal projects, however, since it's currently impossible to require ESM (v22!) and our tooling (nx) produces bogus ESM code it's not feasible to incrementally adopt ESM. If we didn't have bigger problems I would have solved this one
idk why you guys make every single thing so difficult to use in the js/nodejs world. very single thing. check another ecosystems that done it ruight
If a library is old and made in commonjs, it requires a lot of fiddling to make it possible to use that library when using typescript. If everything was ESM, then I assume this would not be a problem whether I use javascript or typescript
If find myself using ESM when I am developing using react/nextjs. I mostly use commonjs in micro services since most libraries in npmjs registry use this pattern . I fear changing to esm
If you consume one ESM package it feels like the entire application has to be ESM. The tooling from node, to ts-node, to webpack, to typescript, to yarn all it takes is one package not working correctly and the whole thing falls apart. It's a nightmare.
Im still confuse with bundler with too many config and the naming can be closely related.
immense issues -- no clear path.
import file endings
import issues. __dirname __filename features.
import old libraries with common js
import paths changing

Import X from y.js is horrible especially considering that I have to write .js to import from a .TS file
importing cjs modules alongside esm. it’s a bummer to have to rewrite working code just for it to continue to do the same thing
importing from common modules and exporting
Importing modules, custom functions etc...
Importing old packages
importing?
In fact, I only use require () if it's not necessary, because I use Node.js more to develop background API, and require can be imported dynamically.
In Frontend applications, it's still difficult to use `"type": "module"` in `package.json`, because of bundlers, webpack and development tools around it (e.g: Jest), that still struggle to support ESM.   However in backend applications, like using Fastify, with Node.js test runner, I had no problem to use ESM and benefit from it, by using top level await, and updating my dependencies which can only be usable with ESM.
Inability for CJS packages and projects to consume ESM libraries.
Inability to import CJS from ESM. TypeScript configuration confusion.
Inability to require esm modules (which is changing with the new release of Node.js v22!).
Inability to require() ESM, making stepwise migration more difficult.
incompat between require & esm
Incompatibilities and other hoops to jump through when using with TS
Incompatibilities with other packages, the package.json definition
Incompatibility
Incompatibility between CJS and ESM: you cannot import an ESM using a require() in a cjs file. I would like to publish ESM-only packages on NPM. However with this incompatibility, this can impact users.
incompatibility between esm, non esm and complications when using them with typescript
incompatibility with used packages. Some of the services we develop and packages we use are in cjs and other in esm. Often it complicates the build process and is a big pain to get stuff working.
Incompatibility/hoops-jumping with CommonJS, especially when it comes to npm packages
incompatible modules, async imports
incompatible with some modules
Inconsistency
Inconsistency in filepaths while importing. The filename should end with "js/ts" prefix
Inconsistent support by 3rd party modules, packages, libraries, everything is complicated ...
Inconsistent support by third party libraries
Inconsistent transpilers, autocomplete of import paths in vscode
index files Libraries not using ESM Bad / confusing typescript support
Inertia, our current build system works fine.   And vaguely I've heard ESM is still hard to use with Node
Integration with Typescript and most other tooling
Inter-operability with CJS

Inter-opt with CommonJS / require. I'm hopeful that `--experimental-require-module` may largely solve this.
Interaction with npm modules written in CJS and adjusting the Typescript config files accordingly
Interop between CJS and ESM. Bun nails this
interop with commonjs
Interop with old common js packages, package.json exports/file extension name confusion, impossibility of publishing packages that support everything
Interop with older packages using common js has been the biggest pain point as we can't force all things to move all at the same time.
Interop. Libraries that are now only support esm.
Interopability with CommonJS because I use Nest.js extensively.
Interoperability
Interoperability between CJS and ESM – currently they're basically separate ecosystems
Interoperability between ESM and CJS
interoperability between ESM and CJS, since a lot of packages are still in CJS
Interoperability when a dependency upgrades to esm and some packages never get updated
Interoperability wit CJS module
Interoperability with cjs modules
Interoperability with commonjs
interoperability with esm and commonjs
interoperability with existing projects running old versions that don't support it
Interoperability with old packages
Issues with commonjs libraries
Issues with external librairies
It doesn't work, but I'm hoping Joyee's work will fix that.
It is hard to mix cjs with esm.
It needs to integrate seamlessly for typescript user. As a typescript user, I still encounter ESM issues from time to time.
It requires a monolithic upgrade of everything... can't easily do it piecemeal.   Typescript was also painful
It was not yet ready, I use it in new projects
It's a major pain having to figure out that some module I want to import is CommonJS only, and I have to work with some magic bundler property to be able to use it.
It's a old application and my team need to finish some refactoring before we can start using esm
It's confusing to update the tools that I use to all agree on ESM support so I generally just give up and move on.
It’s hard and complex, there is always an issue. It’s get worst with TS and npm packages.
It's never working fine... Tooling nightmare
It's not 100% interopable with Typescript
It's only pain.
It's too complex for library authors to provide *real* ESM modules. I think the problem arises when they want to support both ESM and CJS. Hence, I have stomped with many libraries that market themselves as ESM-compatibles, but they are not. Is an awful DX to install a new library and that the basic "hello world" or tutorial for using it doesn't work because importing problems.

It's very confusing right now. But after a while one can get the hang of it.
its a pain to migrate a cjs ts project to esm since you need to add a `.js` ending to all imports even though the source is a .ts file
Its been hard getting everything to work together. You introduce a new package, mainly build tools, and you spend a day or two trying to figure out how to get it to work with the other shenanigans you had to do to get other build tools to work with ESM. (Note: this has been getting better)
jest
Jest
Jest
Jest
jest
Jest and similar that can’t move from CJS to ESM
Jest support for ESM
Jest support has been lacking.
Jest testing library doesn't have great support.
Jest with TypeScript and ESM.
Jest, fractured ecosystem, TypeScript requiring the extension in the import statements.
json import
Just learning
Just put type:module everywhere, its weird
Just the set up. It should just work, node should figure out the details.
Keeping consistency with legacy CommonJS code.
Laborum saepe delectus qui veniam consequatur
Lack of (easy to find) resources for migrating.
Lack of documentation and complications during migration
Lack of documentation and ESM still undergo experimental changes,
Lack of documentation, which stems from an inability to really prescribe a single path for correct module management, publish, and consumption within Node.js-land, which comes from a significant fragmentation of critical use cases each with a slightly different dependency on some specific Node.js configuration, which stems from (in my non-developer opinion) an overly-unopinionated user interface (package.json, Node executable itself) that is due for a redesign but can’t without incurring huge cost for critical global consumers who would be broken as a result.
Lack of documentation. Missing examples, how-to's, recipes, best practices... Especially how to replace simple commonjs code with url, import.meta, etc.
Lack of ecosystem support, this migration is very painful
Lack of ESM support in CommonJS apps
lack of interoperability from require and import
Lack of interoperability with cjs modules
Lack of interoperability with old libs that still use CJS
lack of library support
Lack of standard (well, up until recently) on how to write and publish libraries. ESM vs CJS vs UMD, and all the various (too many) keys in package.json to fill just to ship an installable library.

lack of support by a lot of packages
Lack of support for certain npm packages that haven't kept up.
Lack of support from libraries for ESM and issues with interop between ESM and CJS.
Lack of support in 3rd party dependencies.
Lack of time. Need to figure out how that works with typescript. Need to figure out what to do with non-esm modules.
Lack of tooling support for native ESM
Large amounts of older libraries that still only provide commonjs bundles
Large scale refactoring needed to convert
leadership
learn more
Learning Curve
Legacy application running on node 15.xxxx
legacy Browser / Webview compatibility
legacy browsers
Legacy CJS modules and presence of `require` calls.
Legacy code
legacy code to mantain
Legacy code written in cjs format and packages we depend upon that are not in esm
Legacy codebase/CJS interop
Legacy company libraries relying on deprecated experimental node.js features
Legacy dependencies that only supports cjs
Legacy libraries on NPM stuck in CJS
Legacy libs support
legacy packages dependencies compatibility with ESM
legacy webpack and babel configurations that are difficult to migrate
lib support
Libraries and build tools that have not updated to ESM; using/renaming to .mjs extensions which make ESM modules not compatible for sharing with front end code.
Libraries doing conditional requires and frameworks (like Jest) not supporting it.
Libraries not updated to ESM.
Libraries not working. Switched to Bun for that reason. Never completed the esm switch, and it took days.
Libraries support
Libraries that are not supporting ESM. Mix and matching require and import / export.
Libraries that are only available in require js. We would need to update all libraries to a version that supports esm. No migration path to gradually do this.
Libraries that do not support ESM
LIBRARIES that doesn't support ESM
Libraries that don’t fully support it, using CommonJS and ESM libraries together
libraries that don't support esm
libraries that force to use "require"
Libraries that pretend to support ESM.
libraries that work exclusively with ESM modules or exclusively with common js modules

Libraries, Bundlers, Testing Framework and Typescript (!!) lacking good support Sheer amount of work to migrate large projects and test them
Library compatability
Library documentations
Library support
library support and documentation
Library support, lack of docunentation, confusion
Long standing issues (in node and v8) preventing jest to support ESM in a stable way. Complex dependency chain in our setup, requiring buttom up migration...
looking for nature .env support
lot of library still use or partially support common js
Maintainers of NestJS, doesn’t see need of migrating into ESM for some reason.
Maintaining multiple build outputs for different consumers
make feeel cray
Making ESM interoperable with existing libraries and tools can present significant challenges.
making it work with cjs
Managing imports was a bit annoying, since some packages were still CJS but some had ESM support. So we had to modify imports on a large number of files just to support the right imports for some libraries.
Many libraries are still CJS only. This is an issue in case we use bundlers such as esbuild since it wraps the require statements in a function leading to the "Dynamic require is not allowed using esm" error.
many libraries use require as default approach
Many packages only support CSM or if you use CSM cant import ESM modules in a easy way
Maturity of how ESM works e load files.
mdb publish
Migrating APIs from `require.resolve`/`__dirname` to `import`/`import.meta`
Migrating existing code
Migrating package.json to type module is painful/long, so for now stick to the mjs ext
Migrating to ES Modules (ESM) in existing Node.js applications can present several challenges and pain points. Some of the most common include: Tool Compatibility: Some development tools may not have full support for ESM or may require additional configurations to work properly. Syntax Differences: The syntax between ESM and CommonJS (CJS) is different, which may necessitate significant refactoring of existing code. TypeScript Integration: Although TypeScript supports ESM, it may be necessary to use some workarounds to integrate it properly. Third-Party Dependencies: Not all third-party libraries may be compatible with ESM, which can lead to integration issues. Transpilation: If you are transpiling your code using Babel or TypeScript, it may be necessary to adjust the configuration to support ESM. Mindset Shift: Developers accustomed to CJS may need time to adapt to the new ESM module system.
Migrating to ESM was quite painless.
Migration
Migration of existing code and interoperability to easily use ESM from CommonJS
Migration, some code still use cjs and hard to migrate.
Mix between ESM and commonJS modules

Mix of third-party libraries that are/are not ESM and challenges using both in parallel - especially once transpilation and TS are involved
mix with require
mixing cjs and esm
mixing CJS modules with ESM
Mixing CommonJS with ESM modules
Mixing ESM and COMMONS
Mixing ESM and non-ESM modules is a nightmare. Having to run code through a transpiler to run reliably feels like an anti-pattern
Mixing require and esm
Mixing require and esm. Also handling of typescript, esm and require. Testing/mocking with jest and esm. Incompatibility of packages with esm and deep imports
mocking from jest
Mocking, APM support
Module differences
Module Loading Hooks and Custom loading
Module Resolution, Dependency Support, Node.js Version Compatibility
modules observability
Most libs are not updated in pure esm.
Most of my projects are ESM. Some toolchains do not support ESM very well, so I have to find alternatives or roll back to CommonJS.   And it's not friendly to multi-module builds. See https://github.com/SBoudrias/Inquirer.js/issues/1159
Most of our applications are large and mainly use commonjs. We need to have a clear and incremental way of moving to esm (we know that work in that direction is underway in latest node).
Mostly out of date runtimes (e.g. Meteor, which spent a while adjusting to the lack of ways to integrate fibers in newer versions of Node), or limited support for ESM (previously AWS Lambda, though that is not a concern anymore). Or, it used a build system that outputted to CommonJS or something like that.
Mostly that it feels like an afterthought and that there's still a lot of libraries that don't support it.
My boss lol. also, I dont know what is the easiest way to migrate.
n/a
N/A
n/a
N/A
N/A
N/A
N/A
N/A
n/a
n/a
n/a
N/A
n/a

Na
NA
named import from cjs
named importing cjs modules is not works in mjs. example `import {createLogger} from 'winston'`. which cause my mjs library fail, while cjs is works as intended. (both are built with rollup)
Native node modules
Native typescript support
Need for specifiing it either with .mjs or in the package.json
Need training and documentation
Needing to change libraries and apps at the same time. The new require esm thing will be great if it works out.
Needing to update import statements, particularly ones referencing `index.js` files.
Needing to use flags, import map support
Nest js
Nestjs currently does not support ESM
NestJS is not ESM.
never converted old version.
niche dependencies still being in cjs
No
No
No
No
no
no
No
No
No
no
no
no
No
no
No blockers
No blockers. Shipping esm
No clear understanding of how to get there without committing to everything and having to fix dozens of dependencies that might not support it yet.
No conditional import/export. top level `await import()` is a substitute but can't be transpiled to CJS for compatibility.
No ESM
No ESM support in nw.js
No I don't know about esm
No pain points
No problem to use ESM in my project.

No problem.
no, except of some of the open source projects those I'm maintainer, there are still developers ask for support CJS format.
Node specific ESM import requirements of file extension vs bundler and other JS runtimes like bun which allow extension less imports.
node supporting ESM, without things like .mjs.
Node.js + TypeScript + ESM is quite messy. Node is trying to strictly follow the ESM spec while TS tries to match what bundlers want which is much more flexible, so it's easy to write TS that compiles to ESM that Node doesn't like.
node's commonjs resolution system
Nodejs has poor typescript support.
Non compatibility with some packages and tool developer push
Non-ESM compatible librairies / tooling. It's not yet the standard way to do things. Community, libs/framework documentations / setups are still pushing CJS too much
non-trivial mocking at module level
none
None
None
none
None
None
None
none
None
none
none
None
none
None
None
none
None
None as of yet
None for my project.
None I use it
None so far
None so far
None so far since I'm a junior developer. Just a few months in to the field
None with Node.js in particular--most of the issues come from adjacent applications (Typescript, bundlers, etc.)
None, using AdonisJS that supports it
None.
None.

None. Have been ESM-only for at least the last three years.   Have very rarely had issues with modules that don't Just Work™ when imported and, usually, I just very quickly find an alternative to those modules :)
None. It generally works these days.
None. We deal with any issues that come up.
Not all external libraries can support ESM. For instance, NestJS + OpenAPI
Not all libraries work
not all modules are compliant yet
Not being able to mix it with CJS easily.
Not being able to mix require and import. We are not going to rewrite millions of lines of code.
Not being able to require es modules. It is hard to upgrade an existing application.
not being able to require modules
not clear what to use
Not compatible packages
Not compatible with both CJM & ESM at once.
Not fully supported by ecosystem
Not in all services it's easy to migrate to ESM. The mix between typescript and jest is the biggest pain point.
not much, I always use ESM because it is much better, but being forced to include file extensions is somewhat annoying sometimes.
not really face it
not support of straightforward use of ESM syntax
Not supporting extensionless imports is a pain
Not sure - in all my projects I use Babel to compile latest JavaScript / TypeScript code to the standards supported by NodeJS, without closely following the exact state of NodeJS support.
Not the default. Some big libraries not yet ported (nestjs)
Note: I already use it, but I have gone through many problems. Now the situation is much better than some months ago.   Outdated and unmaintained transitive dependencies (in part caused by the "opposite" problem: developers who were to eager to bump their packages to ESM-only, making it impossible for others to have an easy transition period)
Nothing
Nothing
nothing really, except for requiring ESM modules in CJS ones. This is now solved - thank you!
NPM commonjs-only dependencies
npm ecosystem variability
NPM packages that don't support ESM.
null
old code
old code still using require and not automatically converting between the two.
old libraries are still based on commonjs require
Old libraries not compatible
Old package dependencies
old packages (commonJS) and mixing/using/creating packages compatible ESM/commonJS

Old packages mainly, that either only supports CJS or only ESM. Most of the time we need to use CJS and downgrade the ESM, where necessary.
Old packages only support CommonJS
Old packages that are miswritten
Old project migration
Old tools not supporting esm and still not planning to do so
Old version of Node in some environments (Glitch for example) Bad esm support from libraries, even if it was used for linting, typecheck, tests, creating migrations, etc.
Old, cjs only libraries
Older libraries
Older libraries that don't support esm in combination with typescript
Older packages
Other developer inertia to leave cjs
Other libraries not playing well and enforcing usage of .cjs for certain things
other libraries that does not support esm
Others using old non-standards, so deprecate it at some point. DON'T use polyfill, compat/legacy support, work-arounds... anyone needs, will use an older branch.
our own technical debt, using old node versions
package incompatibilities and backwards compatibility
packages not supporting ESM
Packages that are not updated
Packages that don't have esm versions
Packages that don't support both ESM and CJS. It is impossible to do a mass conversion for a stable/large code base so incremental migration has to be supported
Packages that make our work easier but aren't well maintained to work with ES6 import
Packages that only export as CJS while our project is ESM
pain points have been the move the project from CJS to ESM: changing imports, changing package.json - especially exports
Pain points in tooling and support in open source libraries
Pain points would be having to specify modules, when ESM should be the default.
Painful , lack of guidance to support both cjs and esm in the librairy i support. Versionning nightmare
Papercuts across dependencies. It's hard to pinpoint a good example... there are so many little edge cases dealing with the long tail of NPM packages.
Performance (CJS is typically faster than ESM) and lack of tooling support (missing loader hooks, etc)
persuading others its the way forward and stop writing code in the format
polyfill
Poor IDE integration and forced async CJS interop. Existing investment on CJS codebases. Newer library versions not supporting interop
Poor interoperability between CJS and ESM. Both when importing/requiring packages, and when importing/requiring files. It's extremely difficult to migrate legacy projects for this reason - I can't just start writing ESM because I can't import it in CJS files.
Prior to node 22, trying to get every dependency on esm was hard.

problems getting it running with webpack
Problems in importing using custom path built using variables
Project conversion to ESM
project that have shared modules, some package of project are commonjs and other esm, shared modules can't be esm because of compatibility of importing esm to cjs.
proper runtime support, syntax, and cumbersome workaround
Quite a few pakcages in the backend have been cjs and that has made using them with Typescript hard.
rando npm modules
Random libraries may not be compatible with esm. I’ve taken to using esbuild to work around issues.
Refactor legacy codebase
refactor time, reused microservices with project overrides
refactoring old require syntax
Releasing npm typescript compiled packages that work both in common js and esm. and the fact that if the same npm module is both transpiled to esm and cjs it might be loaded twice depending who includes it
Require
require ESM lib from cjs app, I have to publish two entrypoint.
require to import migration
Require vs Import is still confusing. Some stuff wants to import now rather than require.
require(ESM)
require(ESM)
Requiring file extensions on imports. Particularly painful when trying to configure with Typescript linting. It's hard to get imports to "just work" out of the box.
Resolving modules programmatically is hard.
Review all the code to change the requires to imports and exports.
Rewriting code
Rewriting the whole code to match the compatibility with modules.
Running ESM is a hard task on it's own & typescript adds comfort which doesn't come in handy when writing ESM directly.
runtime stable support
Setting up packages for esm, cjs, ts etc.
Setup env with ESM is painful.
several dependencies start complaining if you use too modern tooling
Single Executable Applications
Single Executable Applications don't support ESM modules vercel/pkg, nexe - both don't support it
single file executables not supporting esm (built-in sea or vercel's pkg)
Small additional toil configuring projects for CJS + ESM together while using other tools (Jest, Eslint, TS, etc.)
So many packages that didn't work. We even had one project on ESM for a short time, but then had to go back to CJS because of a profiling package that required CJS. I still hope that this will finally be the year of ESM.

Some CJS modules cannot be easily imported using ESM because they do not provide an ESM export method
Some dependencies are ESM only. Other consumers of our shared libraries are CJS only. We're stuck in between.
some dependencies are still not compatible with ESM
Some legacy libraries are not compatible and it's difficult to find an alternative.
Some libraries are ESM only, Some libraries are CJS only. we could not mix these modules, because ESM could not load CJS in Node.js v20.
some libraries are working in local but not in production
Some libraries do not support ESM.
Some libraries do not support it
some libraries still use require syntax so i have to, i wish them to update so i can use all libraries in a single way.
Some libs support one module type only, and cannot be used in another.
Some modules such as path and fs Sony work with es modules.
Some npm packages are not supporting ESM
some npm packages that only support cjs
Some of the existing external packages doesn't work well
Some outdated tooling and dependencies.
Some packages seems to not support ESM
Some people in the industry(backed by heavy marketing and money) encouraging developers to not rewrite their codebase in ESM while ensuring CJS must be there whether is good for the Javascript ecosystem to move forward or not.
Some popular packages that are dependecies of other packages, or i directly use these popular packages refuse to update from cjs to esm, or, the switch is partial and does not work well.
some required libs do not support ESM
Some third party libs integration and extra configuration in the project
some tools don't support ESM
Some unmaintained libraries may not support ESM in projects planned/ongoing the migration to ESM.
Something horrible is when you have (`import`/`export`) and "require" (with jest for exemple) so you cannot import functions in your tests
Sometimes I can't only change type to module, because some packages don't have esm version.
Sometimes i cannot find the ESM on npm for a package i'm looking for
Sometimes I meet the package providing only ESM, which doesn't have compatibility with other packages.
Sometimes incompatibility with old libraries. Had difficulties to change the way I think about the dependency tree
Sometimes is a pain in the ass mix ESM and CJS dependencies in the same project.
Specific npm dependencies
Specifying "type": "module" (barely a pinch) and Jest's alpha support for mocking ESM modules in Node. I wish there were better options for mocking ESM modules without having to `await import()` at the last second; I want to do import statements at the top of the file.

Static types
still many libraries on repository require commonjs, like unzipper
subpackages that are not ESM compatible
support for commonjs
Support for mocking in test frameworks / particularly Sinon.   We are currently doing a massive test rewrite to remove that dependency so we can convert to esm
support for running typescript files directly, packages support, default values
Support from libraries like Prisma
Supporting both CommonJS and ESM is a significant challenge.
Supporting libraries
Supporting other libraries/tools that don't support ESM.
testing
Testing, especially mocking part
Testing. Jest requires you to run node behind an experimental flag, which has proven to be a rough edge. It would be amazing if esm was fully supported without the need for running behind a flag.
Tests runner support
that import.meta.resolve is still partially experimental. Also that it can't resolve a package's directory or, equivalently, it's package.json file.
That it's not the default.
The .mjs extension or the need to specify it in package.json
The biggest pain has been being forced to point directly to a file rather than being able to point to a folder that will look for the corresponding "index.js" file. If you are using ESM via the package.json "type: 'module'" configuration, you can't import from a directory you must import from a "<directory>/<file>". Now people will have "<folder>/index.{js/ts}" all over the place because the ESM support does not resolve default "index" files. This creates a lot of confusion, as well. I don't understand why NodeJS can't just do what everyone else is doing with ESM support.
The biggest pain with ESM is that it took Node over 10 years to support it properly, and that tools like Babel and Webpack had to step in and fill the gap, and by extension becoming the defacto standards, unfortunately these tools did not implement ESM correctly and invented a bunch of problems, like importing modules without file extension. This also means that we now have a bunch of tooling like JEST etc. that relies on a broken version of ESM, and we are stuck in this horrible state of not being able to move ahead with deprecating CJS. If we do anything in the next 10 years, it should be to make sure we do not repeat the complete disaster that was ESM and Michael Jackson Scripts.   ESM is the standard, remove support for CJS in Node 30. Tell developers to just deal with it. This wishy-washy approach where we get the worst of both worlds to not upset anyone is upsetting everyone.
The boilerplate/setup that goes on to be able to have a project properly use ESM import/export in an environment that is mostly built around CJM is a pain. Happy to see CJM is slowly, but surely coming to an end, it's a real set back.
the chain dependencies on CJS modules
The ecosystem catching up

the ecosystem has no clear path. package authors suffer with exports. Some packages are only ESM so my CJS-only workflow does not work , etc
The inability to load (older) CommonJs dependencies consistently and hassle-free. ESM and CommonJs are incompatible and require a lot of configuration and a deep understanding of the inner workings of all tools involved. Basically, people want to write modern code consistently without being tooling experts.
The incompatible behavior with the Web ecosystem, that requires some hit-and-miss configuration, specially when using alongside Typescript.
The integration with TypeScript, especially for the bundle tools like rollup and esbuild, are not working properly against the old packages which are still use the ES5 way of import/export. We'd prefer to have an official powerful tool to help on the bundle behaviour.
The lack of native TypeScript support and poor interoperability between ESM and CJS modules. This is partly to blame on library authors: the very early ESM adopters where libraries onlt worked in best-case scenarios, abandoned libraries stuck on CJS, and library authors that don't have the resources or the desire to refactor. However, node also carries much of the blame, since in Node.js ESM support and loader hooks were stuck in an experimental and unstable phase for for years, hurting adoption and confidence.
The lack of test cases in legacy code
The largest pain has been Node diverging from bundlers and the larger ecosystem with how it adopted ESM. Not being able to just require() ESM has caused so many issues for us and our users.
The migration is very confusing. This applies as both a consumer and package author. The reason our core product is still not using ESM is because of a major package that we depend on not being ready.
the old packages which use require、amd and so on
The one-way nature of the migration.
The overwhelming infrastructure of existing CJS packages, and trying to maintain support without overly complex build tooling.
The pain of support of critical dependencies
the platform too old to support ESM or the experience using external scripts written in NodeModules
The required file extension in import paths, and making it work with TypeScript and testing frameworks like Jest
The smaller ecosystem of packages supporting ESM. I used ESM and was unable to use very primary / necessary packages in my project.
the stack may be incompatible with the latest version of the node
The top-level project in our tree of internal modules is massive and has many dependencies on cjs semantics, meaning all subprojects that could be esm cannot be
The typescript integration, and the npm ecosystem make it difficult to work with ESM only packages
The whole ecosystem is broken. Almost every package out there is shipping invalid ESM with broken types. ESM is almost unusable without transpiling first.

The whole ESM/CJS debacle. At times it can be an absolute pain trying to import a library if it is only using CJS as we use ESM (why wouldn't we). For example, __dirname/import.meta.url is the absolute bane of my existence
There are still many packages and tools that do not support ESM
there is no business value in transforming existing projects to esm
There is no way to unload a imported module in ESM. That can lead to memory leaks when a persistent service need to update some logics.
There main pain is that I cannot import ESM modules with .js extension without explicitly write this extension. For example: I have ESM file user.js an I need to import it from a different module. So I have to write "import user from './user.js'". I wish I can import like this: "import user from './user'"
There's a lot of stuff I'd have to migrate, and I don't have time to do it. Also, since so much is "transpiled" down to CJS, CJS seems like a better path to success.
Third party libraries that does not support ESM
Third party node_modules that don't export as esm
time
time and having a huge amount of files to migrate to ESM
Time and regression testing
Time disparities between feature support and library support of the the feature
Time taken to figure out how to use import/export.
Time to go back and rewrite old code
To refactor large codebases in CJS into ESM, going full ESM to prevent incompatibilities because of mixim import/require. Some tooling to aid that refactor?
too complicated.
Too many issues to list. All sorts of things go weird with the tooling. In most cases I've switched to Bun. Bun did it right. It allows devs to not care about CJS vs ESM -- things just work. Node should do the same.
Too many settings Too complicated Low compatibility
Too many to list. Porting large existing codebases is hard as lots of tooling always seems to break in unpredictable complicated ways (Karma, Oclif, old Webpack, TS issues) and changing to ESM in one place is viral so requires updates to huge chains of packages & code that are otherwise totally working and haven't been touched in years. New ESM changes look very promising if it will support incremental migration (mixed ESM/CJS projects - realistically this will be the reality for many years/decades to come). This has been a *huge* pain point.
Too many variations of the import syntax between EMCA versions as well as Typescript
Too old big monorepo and small team
Toolchain issues
Tooling
Tooling and libraries comparability
Tooling and observability
Tooling designed with CommonJS doesn't always support ESM.
tooling support, e.g. testing frameworks primarily (mocking)
Tooling, like nx. I also had problems with jest, but I think it was related to nx
Tools

Tools/libraries that I'm used to, not working properly or requiring extra effort.
Top Level Await in ESM
Transcompiling
transpile
Transpiliation, mixing of ESM and CommonJS depenencies, module isolation.
Transpiling from typescript
Transpiling typescript is my main pain point, specially when integrating with test tools (jest, etc)
transpilling the code and matching with typescript
Troubles of importing ESM <> CJS
Tsc build can't transpile ESM to ESM
typescript
typescript
TypeScript
Typescript
typescript
Typescript / CommonJS / ESM compatibility, especially with external libraries that ship ESM only
TypeScript and Jest
typescript and the fact that we have to choose between cjs and ejs and having both im the same application is a pain
typescript and tooling in general
Typescript does not support imports like "*.ts" without hacks, and I need to import ts files like "*.js" that is confusing
TypeScript has become the standard for front-end development. Developing Node.js applications inevitably involves a compilation process. The ESM standard requires that package imports require a file suffix. In development, there is usually no suffix. Build tools don't support this situation very well untill now.
Typescript in combination with ESM
TypeScript integration
Typescript integration
TypeScript problems, dual format package building/path resolving
typescript support
Typescript support was clunky and major libraries not supporting ESM. Would be easier if Node.js had first class support for Typescript
typescript tooling and editor integration of ESM in mono repos. Ie 90% of the time I can't just `ts-node src/dev.ts` in package b that depends on package a as it will never compile package a.
TypeScript, compatibility
Un-supported libraries. Would love to see Node JS support features in a light-weight easier way in house.
Unable to use CommonJS modules.
uncompatibility of packages
Unnecessary setup steps. ESM should be default.
Unstable APIs, special steps, changing requirements - it should just work automatically and by default.   Also, when authoring packages, the different requirements for importing ES modules

forces me to dual-export to both ESM and CommonJS, which complicates my building and deploying.
Unsupported third party packages.
Unsure how to implement it.
Unusual errors tend to crop up, but it's difficult to tell if those are nodejs specific, package specific, or something else
Upgrading frameworks.
use additional tools and bundlers.
Using a mix of libraries that support esm and those that do not
using ESM and non-ESM 3rd-party libraries together
Using JS/CommonJS dendencies.
Using libraries still in CommonJS in ESM applications.
Using libs that do not support ESM
Using mix of packages that can be commonjs or esm
Using non ESM compatible libraries (NestJS)
Using none esm packages
Using old and new packages
Using older node version currently
Utilizing build tools such as esbuild there are few, if any pain points
VB.
Very few famous packages (xlsx) still using CJS to export so that when you import it in some application, it won't work, unless you write it as: async function foo() {   const bar = await import('./module.js');   bar.doSomething(); }   It's just not clean.
VS Code extensions cannot use ESM yet.
Wanted to use Electron, didn’t work with ESM.
We bundle for production but for development Jest can't load native ESM properly so we still use CJS only because of that)':
we din't try this
We have a huge CJS code base. We need an approach that allows incremental change and both CJS and ESM to be able to require each other.
we have not have any problem
We have other priorities
We just haven't had time
We unfortunately use older versions of node that doesn’t support ESM but we like to use them during development
We use node to test and transpile a large suite of react webapps. Janky ESM support in test packages like Jest continues to be a thorn in our side, but that's not on you.
We use path aliases to shorten our import urls. But there isn't any library that can cater to both converting urls to relative imports and also take care of .js at the end of file name. If there was a way that I can utilize path aliases without having to write .js in my ts files but on building, it would automatically resolve imports full i.e. relative path and put js then.
We’ve now migrated to Vitest, but lack of stable Jest support was an issue. Also, the lack of a stable "require-in-the-middle" for ESM (important for observability) due to unstable Node APIs does not help.

Webpack does not stably support esm output. Reqct does not have an official esm build.
Well, some of it does, some of it doesn't. The backend stuff does. The code we ship to the browser does not. That just requires a better understanding of bundling on our end.
What will and won't work across the matrix of runtimes, TypeScript, and bundlers is really confusing.
When a library drop the support for CJS to use ESM, everything breaks, some libraries call this lib using `require` and others using `import` which cause a lot of issues, we can ask the library mainaners to provide both, but some of them doesn't want add a build process, and write only js using ESM syntax, so if node can support both syntax at once (I don't want to say "like bun") it would be great,   another issue is the difference in named and default export between CJS and ESM, because of that, when I try to compile a TS code that uses default and named exports, the default export will different between CJS and ESM usage (CJS: `const module = require('module').default;`, ESM `import module from 'module';), I know that this is the nature of using mixed exports in your module, but it would be great if you figure out a way to solve it (you have smarter people 😉)
when spawning a node app package.json is not read, so I have to pass a flag --experimental-default-type=module
When you want to bypass the package.json exports for any reason, you are obliged to import with a relative path to node_modules libraries. This can be a pain.
wiht {json}
With bundlers and other transpilation steps, the configuration to achieve proper ESM module support feels daunting.
Writing standards compliant ESM import syntax in the REPL. This is painful enough that I literally use Deno as my temporary REPL during development.
yes
Yes
Yes
yes defaut npm init is commonjs
Yes i do.

Q18 Have you encountered any issues writing TypeScript where the production code used ES module syntax? If so, please include as much detail as possible such as tech stack.

i haven't
- No support for absolute paths (had to write custom loader) - TSConfig for ESM is very confusing to get right - Adding the .js import is manual (should have a codemod)
- TS prefers no extensions in imports while they are required by ESM

1 using vite or rollup, typescript `import {createLogger} from 'winston'` will fail on mjs, but not in cjs while executing (so, I forced to write another plugin to convert it to `import winston from 'winston'; const createLogger = winston`)   2 talking abount ts, Im REALLY want a non-js type checking. original typescript type checking is super slow on large projects. its slightly better with something like fork-ts-checker-webpack-plugin, but its still slow and this fork checker includes all project ts files, event if its not required/imported at runtime   3 I know, its about typescript only, but its will be good to have something like a "loaders" for typescript typeckeck. rn, I forced to transpile and reformat my vue files to typescript files via weird ts files during ts-check job in my ci pipeline
A lot, but everything was fixable
A year ago Vercel and Next.js didn't support ESM very well, preventing me from moving forward with my project to ESM.
Absolutely. We wanted to consume a module that was written in native ESM, but we couldn't because it mandated that our application was in ESM. But then when we tried typescript demanded we did line endings with extensions, but most of our folders were imported as "#root/someFolder" and that folder had an index.ts file which barreled the content out, so then we had to update every import to "#root/someFolder/index.ts" but then that didn't work because it had to be "index.js"... it was constant debugging for zero tangible gain and we threw it away to use cjs.
adding .js suffix to all relative module imports.
all the troubles relates to the case when you need to have cjs and esm packages both
All typescript
Any
apk
at first yes, but switched to a framework nestjs and just used their documentation and stuff works.
At work, we use ts-node, and it is configured to compile to commonjs, this is so that there were no massive changes on the codebase regarding libs compatibility.
building and consuming private npm packages while having a mixture of nodejs versions and package.json settings among all applications
Can’t get our universal app to work with type=module. Fe (vite) doesn’t need it (or want it). I’m supposed to change all out imports by including file extensions or whatever. Just can’t get it to work so we’re transpiling to conmonjs modules for the Node app.   Stack: Fe: Vite+React+TS Be: node+epxress+ts-node(dev)+TS
Can't recall all the details, but tried to flip everything on a backend with 10k-50k lines of code to ES modules and ran into issues getting it to run. Didn't have time to dive down the rabbit hole and just left them as CJS.
chalk
compatible issue. upgrade to esm from cjs is painful
Complicated config compare to Bunjs
Conflict between modules
const foo: typeof import(“yada/lib/index.js”) = await import(require.resolve(“yada”, path: [dependencyLayer])); is a lot rougher than just calling require.
Convert vanila js to typescript for better future use case

couldn't take data from mjs files
Different Typescript require/import/etc syntax not always fully compatible
Don't remember
don't use TS
Don't use TS
don’t use ts
don't use ts at the moment
Don't use TypeScript currently, but may happen in the future.
Dont use TypeScript
During Session Management of a User where some Tokens are to be injected and there we have to manually configure the ts.config file .
Dynamic Path of modules
Encountered no issues.
Enums
ES
ES is extremely customizable and configurable so there are sometimes confusing issues getting it to work consistently because different developers make different assumptions.
Everything I’ve encountered boils down to dependency hell, which is not a Node.js specific problem, but rather a math problem. Most issues can be resolved and in my opinion are resolved by the time you need them to be resolved. If I’m using some <1,000 user dependency, that’s my own (sometimes unavoidable) problem. As adoption of TypeScript and post-ES6 JS engines has improved, so has typing since the days where you had to pray somebody put @types/foo in DefinitelyTyped where foo is some <10,000 or even <100,000 user thing that you use. The problem is when dependency hell prevents you from benefiting from the good new thing you need. Two trivial problems (they had to be trivial, I doubt anybody who has production code significant enough to matter (e.g. my teams) encountering significant enough ES Module typing issues to incur even a 10% loss in economic value (through unsafe workarounds that lead to more costly breakages) would still be using Node.js. There’s a reason why there are freaks online who still proclaim .NET to be the best, and I am not one of them, but what makes my liberal attitude any better than that of the people I mock who write production code in Python? I digress. For example when first adopting Svelte (prior to sveltekit) and attempting to write TS-embedded svelte with a serverful Express (also TS-written) backend, it just simply was not going to work (not worth proving it can work). Thanks to Svelte devs and Vite dev (singular lol) becoming good at Node and TypeScript, SvelteKit with its community of adapters tends to work ok — in no small part thanks to the thankless souls on Microsoft’s content writing team who somehow have better documentation on how to configure and deploy a Node.js application in any given deployment environment than you guys do… Another trivial example is silly things like, what if you’re using Node.js basically as an IDE for TypeScript code? Ok the solution there is easy: you need Node to *actually* run because you’re depending on it for ESLint and some of your deployment workflow tools (such as transpiling lol) and you need TypeScript compiler to actually compile it to say real ES6 that runs in let’s say Apple’s JavaScriptCore. You accept the lack of true symbolic / prototype chain type safety because when transpiling to some other environment with its own custom stuff you know the runtime env loses the prototype chain across because everything is across “frame/window”. You also accept that certain String

operations and Error constructors have existed since ES6 and are supported in every major browser / engine, but Node does not support it until ES2021, so TypeScript keeps giving you type errors. You accept it, include “ES2021.Strings”, and move on. Anyway, this is not how anybody uses Node.js, so it doesn’t matter. And by the time somebody needs to use Svelte, it works in Node. I’ll ask my friends who are still at Google Search how that TypeScript migration is going and get back to you.
executing typescript files as scripts is burdensome
Fastify, TypeORM
Hard to Find data types for inbuilt function and libraries
Hard to maintain tsconfig
hard to setup at first
Haven't used esm on prod due to above issue.
https://github.com/microsoft/TypeScript/issues/16577 and related
I can't share the exact details, but I described the issue above.
I definitely have but I completely forget
I did at first but added configuration files and then that seemed to fix the issue but I'm only on my 2nd year in my role as a dev when something doesn't work will let you know thanks for the great job you really have already done amazing job hope all is well.
I didn’t use to Typescript, I worked with javascript
I do not use Typescript
I do not use TypeScript, it feels like the wrong tool for the job. I wouldn't mind having optional type capabilities in CJS/EMCAScript, but if I wanted to write C#, I'd just write C#.
I don't code in typescript but I'm looking forward to code in typescript.
I don't use TS
I don't use TypeScript
I don't use TypeScript
I don’t use typescript so far. Planing to switch soon
I don't use typescript, even if I did I prefer Java :D
I don't use typescript.
I don't write TypeScript
I dont do TS. It’s a cancer that just needs to die already!
I dont use typescript
I faced usually "cannot use import/export outside the module"; this is constant irritant ;(
I had an issue few months ago as I am new to typescript. I wanted my compiled modules in ESM as well, but due to some error I have no idea about, I had to change compilation to CJS.
I had no problems.
I hate having to always install @types/node A native integration of atleast the type defs (not full TS support in node) would be nice
i hate how esbuild is needed to bundle ES-module files
I hate TypeScript
I have but I don't understand what's always going on. At a certain level, I want things to "just work" and various modes of compilation feel required. Between `esbuild` and others, I'm never sure what the right way is to run my typescript with esm.

I have not encountered these issues because I use a bundler for BOTH the back-end and front-end because I am keenly aware that I can't run that kind of code natively.
I have not. I try my best to completely separate the two technologies; I hardly ever use them in tandem. When I have, the frustrations have been minimal.
I have to add `.js` to the imports for node.js applications. When using ts-loader with Webpack for the frontend it's not required.
I just want straightforward way to run Typescript applications in NodeJS. Current workflow is really messed up. NodeJS is not a browser, it should be possible to run typescript projects out of the box with NodeJS. Just take the typescript config it uses bundle/transpile/compile transparently to the user.
I love what TypeScript does for “intellisense”/auto-complete, but I dislike reading it and never write it. I prefer to avoid transpilation.
I never write typescript, only .d.ts files for my packages in npm
I purposefully avoid using TypeScript as I do not want a build stage for our projects. I use JSDoc and dev-time editor intelligence to get almost all of the benefits of TypeScript without any of the shackles it imposes on a dynamic language like JavaScript. (Kitten – https://codeberg.org/kitten/app – does have a very quick – under 500ms – build process that uses esbuild for bundling.)
i still think there is a room for development on TS it still seems like struggling on this part
I use nestjs heavily, It takes of js to ts conversion. But it confuses me sometimes
I use no TypeScript.
I use TypeScript and `"type": "module"` in `package.json` (ESM), and have no issues.
I've had trouble finding the types for core libs -for instance const server = http.createServer((req: http.IncomingMessage, res: http.ServerResponse) => {   reading the type declaration of createServer, it wasn't clear to me at first that the types were IncomingMessage and ServerResponse not Request and Response   Also - getting a command line setup for typescript is much harder than it should be.
If I use `module: nodenext` and `moduleResolution: nodenext`. TypeScript needs me to write file suffixes in the development environment.
If you want to use tsc as a TypeScript compiler for Node.js, you have to add .js extension when you import. e.g. `import { something } from "foo.js"` You have to import from foo.js although it does not exist and there is only foo.ts. It was very confusing when I was not familiar with that spec. Fortunately, I recently noticed that I can set `    "allowImportingTsExtensions": true` in tsconfig.json to allow `import { something } from "foo.ts"` when I use TypeScript compilers other than tsc (e.g. esbuild and swc), so I wonder if you can officially recommend to enable allowImportingTsExtensions as a best practice.
Ill use TypeScript when it is faster than native.
Im struggled with tsconfig file, so many setting for the newbie to understand and when my monorepo has FE and BE use same base tsconfig is terrible for satisfy typescript warning import of es need .js
Import to SvelteKit required to write `.default.default` and similar annoyances.
importing CJS and/or NodeNext libraries from React 18 SSR codebases (Remix, Vite) breaks not possible/too onerous

In every single case when I do ts-node / node index.ts the first thing I get is "can't call import in non module file bla bla bla"
in general, module resolution in node is a pain
In node 20, ts-node and other Typescript related tools that allows you to itterate fast and still ensure compilation sucess stopped working out of the box if you're writting ESM. Some workaround exists, like running node with --loader flag and --import with a custom js file to still get the compilation but it's not a straightforward and easy to setup, as you often need to dig into actual node.js github issues to figure out what's happening instead of the official documentation.
In some infrastructures there is a problem to occurs down time in services that caused by some es module syntax is failed to compute
inability to compile if the module doesn't support commonjs.
interop with commonjs
Interop. Libraries that are now only support esm.
Is extremely hard to setup and configure a project. The need to use ts-node or tsx and the inconsistencies with how things can be configured. Especially in a monorepo. TYPESCRIPT should be a language directly understood by nodejs like Deno does
is hard to start a Project in typescript
It is not always easier configuring Typescript to use more modern Javascript version, while having to fine tune module generation settings as well, requires quite a bit of web searching when hiting import errors.
It takes a bit of time and trial to setup the modules and we are always in a hurry.
It's confusing to setup ts when you have esm enabled.
It's hard to choose right tool to bundle TS into JS. It's way easier on the frontend
It's just all very confusing in general. I can almost always get things working but most recently I spent about 30 minutes trying to figure out how to get a deprecation warning about --module-loader to go away when using ts-node.
It's very frustrating when I need to import a `utils.ts`, I have to write `import * from './util.js'`, such a confusing suffix, instead of not needing a suffix like CJS.
its mainly painful to set up, runs fine in production
Just recently, I was working with the project next-export-optimize-images which require()s its configuration file from the current working directory. I use ESM wherever possible which causes problems. Ended up forking to make it build ESM.
lack of __filename/__dirname, imports need to use .js extension, which feels strange as there are no js files in ci, just the final bundle file.
Linting Errors: Developers may encounter linting errors when using custom TypeScript modules and namespaces, as ES2015 module syntax is generally preferred. This can be resolved by modifying the code to use import and export instead of namespace. Import Errors: When integrating certain dependencies that require Pure ESM, developers might face errors stating “Must use import to load ES Module” if their tsconfig or package.json was initially set up for CommonJS. Adjustments in the configuration files are necessary to resolve this. Transpilation Issues: There might be challenges when transpiling TypeScript code to be compatible with Node.js’s native ES module support. Tools like Babel may be required for a smooth transition. Syntax Compatibility: Custom TypeScript modules and namespaces can lead to issues since they lack the benefits and standardization of ES2015 module syntax. Refactoring to the newer

syntax can help avoid global namespace pollution and improve module interoperability. File Extension Conflicts: There have been reports of .cts files emitting .cjs files with ES module syntax, which can cause conflicts and require careful handling of file extensions.
Long ago, yes. But I got back to compiling my TypeScript to CommonJS since that.
Main issues have been when libraries/packages have gone "full esm" and then cannot be easily consumed in projects that still rely on CommonJS requires.
Main issues occur with typescript occur with misconfigured tsconfig.json where the esm was configured for bundler consumption which works for UI meta frameworks like Next.js and Remix and JS runtimes like bun but not Node.js which has its own specific expectations for ESM.
Mainly issues that came up were with a Vite project I wanted to leverage some package that was commonjs only and I needed to modify some vite.config setting which drills down into rollup.
Managing source maps, when connecting external services, e.g. Sentry
Many libraries are still CJS only. This is an issue in case we use bundlers such as esbuild since it wraps the require statements in a function leading to the "Dynamic require is not allowed using esm" error.
mdb publish
Minimal, usually tsc and vue confusing types.
Missing file extensions in imports + Conflicting module systems
Mixing ESM with CJS
Most commercial typescript projects can't even manage to set up a typescript build correctly, so esmodules are so far off it's not even funny.
Most developers using TypeScript are using ESM syntax which transpiles to common js syntax. It is a big hassle for typescript developers to use ESM without some major changes on the tsconfig.json or even changing the module type of the entire project. The latter one is going to break a lot of things.
Most problems are related to tests migration (jest) and not in the nodejs itself
Mostly had issues setting up tests (Jest) to properly use ESM imports (TypeScript).
Multi-threading implementation in typescript, unable to set up
N/A
N/A
n/a
n/a
N/A
N/A
N/A
n/a
n/a
n/a
N/A
N/A
n/a
N/A
n/a

N/A We use typescript .d.ts classes that export namespaces to be used with jsDoc
Na
NA
NA
Native support for TypeScript would be great, having to deal with multiple tools and configs could get out of hand
NestJS issue mentioned in the previous answer
never
Never
Never done it
Never wrote TS
Never.
new syntax is good
Next.js Edge environment
Nk
no
no
No
No
no
No
No
no
No
No
No
no
No
No
No
No
No
No
No
No
No
No
No
No
no
No
No
no
No

No
No
No
no
no
NO
No
no
No
No
No
No
No
No
No
No
No
No
No
no
no
no
No
No
no
No
No
No
No
No
no
No
no
No
no
no
No
No
no
No
no
no
No
no

no
No
No
No
No
no
No
No but CJS interop is an issue
No but we also don't let production code use ES modules, we use webpack to transpile all of it. We have a wide browser support requirement so it's hard for us to change that decision at the best practices level. 1
No I only use javascript but in future I will go through typescript
No issue. It's just more cumbersome that it needs to be, look bun.js
no issues
No issues at the moment
No issues.
no just types
No notable issues.
No problem
No problem.
No problems with the resulting output of TS -> JS code with ESM. TS itself will often misunderstand default exports, though I don't have any specific examples.
no problems with TypeScript
No problems writing TS
No TypeScript.
No, I don't live TypeScript. I expect that somedays we have the official type definitions within ES core, and Node.js implemented that specs.
No, I use JS, not TS
no, ihavent
No, the main pain point is the development, especially in dockers. Development requires a huge tool set to make it work, unless you are willing to use a framework (nest as an example)
No, TypeScript-generated ES modules have worked great in Azure.
No: we tried migrating to ESM but it caused too much pain so we never got to production with it. See above answer for more details.
No.
No.
no.
No.
No.
No.
No.
No.
no. but i dont use typescript
no(t yet)

node doesnt support it natively
Node js
Node v18.17.0   Using __dirname with import.meta.dirname throws error as Node 18 don’t support this case
NodeNext and local library package reference needing to be .js
non use correct information.
None
None
None
none
none
None
none
None
none
None
None - our production code is still (sadly) using commonjs for now
None other than it not working 1-to-1 with EMCA imports
None so far
None so far.
none, i'm not using TS, i'm prefer JS
None.
None.
nope
nope
Nope
nope
Nope
Nope!
Not as related to TypeScript other than the timing mentioned
Not at all.
Not more than normal interop, Jest is very painful. Using TypeScript and Node together is more painful than it needs to be, see Bun and Deno. TypeScript is the de-facto language for type-safe JS, so Node.js support would be nice.
Not really
Not really
not really face it
not really, TS and ESM in node has been great so far.
Not since the latest releases of ts
Not that I recall
Not that I recall, but if I could skip that build step and node index.ts that would be very apreciated
not using TS
not using ts

not using typescript
Not when using ES module syntax, atleast not while running the code. In case of development there has been a lot of battling tsconfig in order to make everything match. Also the need for different tsconfigs depending on if I am developing towards web or node in a monorepo.
Noth I'd like to talk about.
null
Oh yes, much banging of head on keyboard. Dynamic/synchronous require()
Only on front-end applications running in the browser with certain libraries using Angular.
Only use JS + typescript doc via VS Code IDE
Only when using loaders such as ts-node or tsx. I stopped using those, and now I no longer have these problems.
Partially related: https://github.com/nodejs/node/issues/47747
Plain, vanilla TS with ESM. Couldn't convince verbatimModuleSyntax to play nice.
Plenty of libraries don't expose their exports properly, such that they cannot be loaded from an ESM context. This requires fiddling with the * or default export to get it to spit out the right property. There's also issues with loader hooks when using ts-node with node 18, 20+ (but not the earlier versions of 19). There is more, such as with knex.js with a knexfile written in TypeScript, but one never has the time to dive deeply into those issues and record what's really happening.
Poor type support from packages' authors.
Rarely, when I tried to import old packages. It was a big refactoring project from Node JS 16 written with JavaScript to Node 20 with TypeScript
Really annoying to change all imports to include file extensions. And to get path aliases in TypeScript to work in production
Recent Typescript versions make it a nightmare to package both ESM and commonjs. I don't have much detail to provide other than it basically without moving away from tsc and the typescript language server completely I currently see no way to to make ESM feel good to work with in typescript. I have had to shoehorn crappy workaround that make the dx very annoying basically just compiling every dependency of a package before hand and losing every bit of debugging tooling for anything but the package I am running in directly, all other workspace TS packages are basically just js esm and my tooling can't map back to the ts source, also for any of the type info to make to the downstream package I have to compile the upstream. Maybe IntelliJ/Webstorm's tooling is just bad but when the compiling and ts language server itself can't find type info without such work-arounds in my use-case ESM is just a nightmare to use.
same as above
Same as above, mostly related to getting build tools to work.
Same point as before, if a package only exports ESM, it won't get picked up correctly, unless TypeScript (e.g. ts-node) is configured correctly.
Setting up TS is a pain in Node JS projects. On top of that, path aliasing can be tricky sometimes too. Testing ESM TS code is painful to set up. It is not well supported by test libraries like Jest and Mocha, and you have to install other packages, spend time Googling, and hack around just to get tests to work. I think I'll move to node:test
Setup might be not so easy to do, lot of configuration is needed to correctly deploy a ts code as Node ESM

so many
So many issues with ESM. - Has to specify `.js` extension when writing TS, which points to nonexistent files - The new moduleResolution option breaks a lot of projects
Some libraries are not properly typed and require hacks.
Some third-party modules/packages written in CJS have problems with import/exports. Also transpilation
sometime IDE works and runtime broken and sometim runtime works IDE doesnt pickup types.
Sometimes the strict is soo Tedious
Source maps are a pain to get working
synta
Tears :')   It's been a while since I tried... but it was confusing that I had to import a .js file despite being used to writing .ts (or no extension) when using the other import styles.   The import field in package.json also never seemed to work   I'm aware they've made this a little more ergonomic since.
The biggest problems I had with ESM and TS were related to Jest (+plugins), ESLint (+plugins), and other dev tools that make too many assumptions about code structure and drag tons of outdated transitive dependencies. I ditched all of them.
The developer experience of TypeScript is terrible due to the mandatory compilation step. We desperately need the "Types as Comments" proposal to become a reality to make native types available and improve the language.
The issue with typescript is that I also have to use file extension in imports.
The main issue is to setup bundler with ts + esm. Too hard to configure
The main problems with TypeScript and ES modules that I have encountered are with testing and specifically mocking.   TypeScript give many spurious errors when mocking functions.
The node types are lacking in some areas.
The only way to have node run typescript is with experimental loaders.
The requirement of imports to have `.js` extension. This works poorly with Typescript. One either have to hardcode `.js` everywhere even though there are not .js files in the first place or use a bundler to rewrite imports or bundle into a single file.
The setup, especially with Typescript
there are some minor fixes which needs to be done but still i can say we are oj right path
There are still many types that just dont work with ESM, generally there's no easy way to make sure .js extensions exist, when you have .js extensions it is usually broken after building with a bundler like esbuild
These days the problems I've encountered are mostly around understanding, I've seen plenty of .cjs.js and .esm.js rather than .cjs and .mjs, I've seen export conditions used incorrectly, I've seen __dirname in ESM etc...   Earlier there were some tradeoffs I've made due to unstable apis changing but those are forgotten times now.   What would be nice is if TypeScript supported producing CJS and ESM code in one build step...
This is hard question for a survey. But yes, there are plenty of issues.
tried to use typescript, but configuring jest for testing was real pain, gave up.
Troubleshoot when there is some error on code, because the error it is on the js file, not in the ts file, there is sometimes that I have to search in the builded file.

Trying to get extensionless imports like `import { format } from './util'`. Will either complain in build but not in typescript. I just want this to work in both.
TS allergic 🤧
TS on 3rd party or deps is ok, but must not be forced to JS developer.
ts sucks
ts-node does not support ES versions. It is a mordern day requirement to have TS node work without transpiling to Node
ts-node/loader need to have type "module" defined in the package.json
TS's preference to omit file extensions vs. ESM requirements for them is an occasional pain.
tsc automatically transform `await import` -> `require` when target is cjs In fact, I am using the interoperability with `await import` between cjs and esm
Tsc build can't transpile ESM to ESM
tsconfig options are a lot, I need to teach these options to our project members.
TypeScript and Node is a topic that can be improved a lot. Maybe some inspiration from Deno and Bun would be great, to support TypeScript out of the box.
Typescript compiling is too slow.
Typescript does not understand some cjs library that also support esm. Which sometimes the code seems to work in dev but it brokes in production.
Typescript in my opinion is generally a big overhead of extra complexity
TypeScript is a PITA on its own, not Node.js specific
Typescript is a virus created by MS to force people to use their development tooling. Javascript with sensible type management like jsdocs is clean and flexible but jsdoc syntax is too verbose. Type declaration should only ever be used to clarify ambiguity so I prefer to go without and instead do better testing.
TypeScript is pointless. Plain, vanilla javascript is all that is needed. It's understandable, does not require using node or anything node related, does not require compiling assembling or anything else, runs fast and is easy, readable, understandable.
typescript is sxxt
TypeScript it's a huge mistake, that's the only issue
TypeScript latest
Typescript needing to import .js Typescript can't generate .cjs or .mjs on demand Parcel.js can't find the .ts when requiring .js
TypeScript not being able to accurately determine whether a "default export" of a CJS module will be, at runtime, available directly or via the "default" property.
TypeScript requiring the extension in the import statements is a non-starter for us.
Typescript slowdowns my dev speed.
Typescript will generate invalid import path. For example, `import 'abc/efg'` is a valid path by bundler, but not for node runtime. And typescript doesn't convert the code automatically when target to node ESM.
Typescript’s autocomplete messes module imports up.
Upstream libraries converting to ESM, but forgetting to change the TS module to node16 or node next causing missing type issues. Usually they can be fixed with a quick upstream PR, but it is a reoccurring issues with libraries that migrate.
Urg, would never use TypeScript.

using cjs in production, so no.
Using ESM with Typescript is a total mess. Node’s exports field in package.json is not transferable to other systems such as frontend bundlers and browsers, so writing a package that works correctly in all places is very hard. Trying to distribute a package that uses both commonjs and ESM in a hybrid approach feels nearly impossible to get correct. Typescripts documentation in this regard is vastly vastly superior to Node’s. Layering on Nx for monorepo management adds a new layer of difficulty because their tooling makes a lot of incorrect choices about how to structure ESM modules, and also relies on tsconfig paths pointing at relative module imports in order to do local package linking, which isn’t possible to do in Node ESM (but is possible in bundlers etc)
Using Express, Drizzle and postgres.js together.
using TSC is a pain with including workers with ESM one time you need type module and the other you don't, and setting the file to .mjs / .cjs crashes for some obcure reason.. i wish i never gone that route again
using webpack, I couldn't import pure ESM libs (had to await import())
Usually I transpile my TS code with tsc or any other bundler. If I target NodeJS, I go full CJS.
wayyyy too many times to list here
we do not write TS
We don't use TS, we see no benefit if you have a good plan. Testing is better than this uncomplete solution.
We dosen't use TS
We have a cobbled together build pipeline with webpack, esbuild-loader, and ts-loader that we'd love to upgrade, but we started production in 2016 and it's proven difficult to modernize.
We have not yet started using TypeScript, only JSDocs
We haven't been able to upgrade to ESM in production yet, despite the fact we use import/export syntax everywhere.
We used typescript, but then we moved to JSDoc, because we do not need to compile the code.
web3.js@1.x.x
Website may only using please help that
When combining local packages, tsconfig and such
When transpiling to ESM, TypeScript enforces odd requirements that aren't enforced for CommonJS, such as explicit file extensions in imports. This often messes with other tools, such as testing frameworks. Specifically, I recently ran into an unsolvable issue with mocha (ts-mocha) that prevented me from moving to ESM because it could not interpret the explicit ".js" extensions.
When working on a large code built around CommonJS require syntax, makes us unable to use ESM because refactoring everything to ESM import/export is very time consuming. Relying on third-party packages that don't properly support ESM. If dependencies are CommonJS only, it can block the use of ESM. And also mixing CommonJS and ESM can sometimes cause unexpected errors or behaviours.
with nx library don't works well
With vite I can write TS and emit OK esm code but it is another complication

Working on Typescript with Nodejs and Express. Its very laggy while compiling typescript and running server.   I am using this commands. "dev": "concurrently \"tsc --watch\" \"node --watch --env-file=.env.local dist/app.js\""

working with bpm workspaces and typescript tooling

Writing code is not an issue, but bundling is really a pain. Especially when we try to bundle the package and release to AWS Lambda via Serverless, there's always some packages that do not support TypeScript ESM bundling, it's such a pain to us.

Writting TypeScript is not a problem at all. The problem is to transpile the code and you don't knoe if the final user is going to use ESM or CJS. So you need extra non-official packages to create both transpilations. It is a pity this situation because there is a standard, ESM, and it is 10 years old!!!

Yeah, I had issues with ESM modules due to other tools. Honestly, I just don't like the whole idea around 2 types modules: ESM and CJS. That's just a mess rn.

yes

Yes

Yes

yes

Yes

Yes

yes

Yes

yes because there were no Types possible.

Yes but I don't remember the specifics

Yes but the issues were with the code intended to be bundled and/or run in browsers. Node is OK now.

yes,

Yes, authoring libraries in ts that supports both cjs and esm is incredibly hard.

yes, basically every typescript project that doesn't have moduleresolution node16

Yes, because TS can resolve extensionless imports. When transpiled, we need to use a build step (package called tsc-alias) to convert the extensionless imports to fully qualified ones. Stack: - Node.js v20 - Typescript 5.4 - tsc-alias

Yes, but I see TS as a pointless bloat on top of what used to be an amazingly fun and dynamic language! The only major thing I've experienced in adding types to JS is that a [literally] 3 second change can now take up to 3 hours/days to make, due to issues created by, "types", which are, more often than not, ridiculous and/or incorrect assumptions on what is/could be happening!   Code now has to be forced into being less optimal too, thanks to TS's crappy "interpretation" of what is or isn't type-guarded! For example. A very quick, small and meaningless example: ```typescript function test(foo: string[] | string = `bar`) {   const type = Object.prototype.toString.call(foo);   switch (type) {   case `[object Array]`:   /* Even though we have a valid check for an Array, we still get: TS2339: Property 'map' does not exist on type 'string | string[]'.   Property 'map' does not exist on type 'string'.   */   return foo.map(val => test(val))   .join(`\n`);   case `[object String]`:   return `foo ` + foo + `.`;   }   throw new TypeError(`TestFunctionError Invalid type for "foo": expecting: "[object Array]" or "[object String]", but instead received: "${type}".`); }   ```

Yes, but it's mostly because the configuration can get quite deep. Frankly I'd rather get rid of TypeScript or have some sort of "jsi" file that describes the types used in my JS files in terms of TypeScript syntax, drives type checking of corresponding modules (unlike .d.ts files), but otherwise doesn't add another thing to transpile.
Yes, but mostly about import statement in which I need to import a `.js` file extension from a Typescript file
Yes, but that only applied to third-party libraries. When one library supports only ECM and another only CIS
yes, docker
Yes, especially tsc, when I have multiple tsconfig (extends) and some config that is relative or not (all files I put output:"./dist", like that but seems don't apply so I have to create 20-30 files with that)
Yes, for old project that I no longer maintain, there's some weird issue that when being deployed into production, the program either failed or can't find said module, however I haven't encountered it on my recent projects
Yes, I use the NestJS and when I tried to migrate to ESM it didn't work well, I use other libs that didn't played well with esm. Maybe this will change in the future and then I can migrate.
Yes, I've encountered issues when writing TypeScript with ES module syntax in a Node.js and React tech stack. Node.js does not natively support ES module syntax for TypeScript files, leading to compatibility challenges.
Yes, it was quite painful until "module: node16" became available.
Yes, it’s quite difficult to get started with this due to lack of clear documentation on how to properly do this, where I might (or might not) need to rely on tools like bundles etc.   I particularly want to use ESM for libraries I publish but there’s no simple toolchain in the Node ecosystem to produce “correct” packages from TS which support both ESM and CJS.
Yes, Jest errors can be very generic in some cases using TypeScript. It was also hard to configure.
Yes, js module resolution
Yes, sometimes it is difficult to find the right typing from imports, when the @types don't exist. I have faced situations where I just used the @ts-nocheck notation to a perfectly working code, just to remove the type errors.
Yes, the imports are required to have file extensions on them.
yes, the issues are primarily related to parts of the project written in js, other in ts
yes, there were some issue with not proper filextension, then with packages that used commonjs
Yes, they don't seem to integrate well when you still have CJS Libs
Yes, using the standard tsc with type: "module" has generated issues in the past.
Yes, very difficult to understand the ESM configuration options in the documentation and in almost all the forums
Yes, with esm only packages like zx with cjs
Yes, with the --require node option.   Using `node -r observability.js dist/main.js` it gave error if using import/export, I had to use require.
Yes, writing paths is harder in ESM than CommonJS, where you could've just used __dirname to get the current directory.

yes; using tsc or babel to transpile was not very easy
Yes: disagreements between types and reality on how things are imported, challenges with assumptions vs requirement to use esModuleInterop (as library maintainer, can you assume this downstream? if an upstream library enables this, will it cause me problems?). Confusion when in an ESM project importing an ESM library importing CJS, giving type conflicts that forced skipLibCheck. Hard to pin down the details, but lots of seemingly non-local effects.
Yes.
Yes.
Yes.   I have many many examples and usually just end up downgrading to a dependency version that still supports CommonJS
Yes. Couldn't import into an esm application, problem with esm immutability regrind mocking and testing.
yes. for example with jest also internal package that I had to declare a module to be able to import the default class
Yes. I build a lot of backend APIs with Express and Firebase. It has proved difficult to transition without ESM docs for these major libraries and the setup required
Yes. I had to specify module resolution bundler to match library code used on front end with the same code used on cli. I wish node supported typescript directly
Yes. In many cases, TS allow named exports but node.js runtime only support default exports for commonjs packages.
yes. Nest.js (10.x) and TypeScript 4-5.x. I have to use await import(...) but for some libraries it is complicated as well
Yes. Nestjs
Yes. Using ts, ts-node, npm workspace, nodemon, jasmine and vscode. Getting these to mesh in a nice way is very tricky.
Yes. Yarn v4, VSCode. I did not realize that I was using the wrong command when compiling typescript so I couldn't compile my ESM code. The issue was I omitted the project flag, so it was pointing to my VSCode config instead of my local config, even though VSCode already knew I wanted to use the version specified in my package.json. I suppose I shouldn't assume that `tsc` would look in my dir for `tsconfig.json`, but I did and I lost hours!
you really need to know what you are doing to make typescript and node to work together with esm. Basically no simple defaults
希望有中文文档(missing Chinese documents)

Q19 - Are you using the following experimental features of Node.js? - Other

--experimental-code-coverage
--frozen-intrinsics
.exe creator
fs.glob, --run
fs/promises API
I don't invest in expermental features so I neither use nor care about them.
I don't use any experimental features
Import Maps
Lastest
lldep
mdb publish
No
No
node test
node:fs glob and task runner
none
none, didn't know they existed
Not at the moment but I've played with them
Other (please specify)
Plenty of new shortcut APIs instead of third party modules
Regarding import attributes, I am using them, unfortunately, because I have to. I’m not entirely convinced they warrant the additional complexity. At the same time, it’s not a hill I wish to die on.
require(esm)
Source maps
Still on v20. Interested in watch, dotenv and wbesocket client when we update
test coverage
The one for use local certs (mkcert)
They all seem not important to me.
top level await

Q20 - Are you using the following new stable features?

--env-file
apk
Axios
Don't know yet
fs/promises, node: protocol
glob (node:fs), styleText (node:utils)
I like test, but not using it. Some missing functionality that would bring parity with larger testing frameworks.

Mocks in tests
No
No
no
no
no
node —run
Not using the test runner because symbols look bad on Windows native CLI. Could use some polishing along with links to packages which extend the reporter
Workers
Yes

Q21 Do you encounter any recurring issues when using Node.js that you would like to share?

- assert API is missing some easy features compared to alternatives from NPM. DeepEqual sometimes is too strict and defects a difference despite there is none
- just kill (or actually deprecate) cjs already please - we have to move on for our own mental sake   - if you have the resources - put some love into ts-node (especially esm version)   - maybe have a *full* LTS changelog when release drops (like "Node.js 22 just became LTS, here is what you need to know about it")? it's always cool to see new features added, but a bit painful to dig through 20 medium blog-posts (some of which are by individuals and therefore paywalled) to understand what actually shipped and/or never left discussion stage and/or was deprecated and/or was a breaking change
- NPM as Package Manager, fat node_modules directory... - I miss a global async...
- The fact that CJS still exists. What's the point? - Native typescript, my dream: `node script.ts`.
- the docs are not complete enough. I think we should have complete examples in the learn section and the api in the docs. Also `--loader` is not document
- This survey should have had more multiple choice - Esm / cjs imports/sharing -> kinda fixed now - Startup time compared to bun - More speed compared to bun
.cjs and .mjs file suffixes are annoying
1. Difficulties to update .node modules after major Node.js updates. There should be a better abstraction between Node.js versions and .node files.   2. .node files can only run from the location they have been installed because they use absolute path references. Therefor I cannot share/move around .node files or custom load them cleanly.
1. mainFields and `exports` in `package.json` is so hard to learn 2. Phantom dependence and install time is so long, `node_modules` is too big 3. devDependencies and dependencies are easy to get Phantom dependence problem
Add built-in support for TypeScript
Add normal fetch from box
Always install nodemon and ts-node

An error occurred when the dependency package was downloaded
apk
APM instrumentation quality for non-library code (outside of libraries directly supported by instrumentation agent) is inferior to PHP and Java.
As someone who supports a number of developers in my organization, I am keenly aware of the impact of breaking changes. It'd be great if those could be highlighted more prominently before they take effect, that would be lovely.   I'd love to see a migration guide standard with every breaking change.
Async tracing. Not just following a promise chain, but also things like: node crashed because of emit('error', ...) for a minor error that should be ignored. I can get the error trace, but I can't trace the emit, which is what actually causes the crash. Similarly debugging errors that are eventually thrown on nextTick with no meaningful stacktrace. Async hooks may help here, but it would be great to see more built-in error handling use this.
Backwards compatibility. It happens rarely, but more compatibility the better. New features are great! Keep them coming!
Better tools/details to find memory leaks and performance issues at runtime would be really helpful.
Blocking the event loop is too easy to do and too hard to track down.
Bundling is a pain. Every project has its own tools and strategies. I feel like it should be part of node with great default to ship to the browser, the server or any target.
c8 node --test is broken in Node.js 16, but not in newer Node.js versions. I know Node.js 16 is EOL, but I don't want to drop support in my packages just to fix tests.
Callback Hell + Dependency Management + Error Handling
Can't run typescript directly, and have to struggle with ts-node / tsx... it's extremely convenient that deno / bun can do this out-of-the-box.
changes to the V8 aren't as transparent whenever Node.js release comes along. V8 still not supporting proper tail calls
CJS vs ESM still seems like a nightmare. Lots of weird errors when trying to migrate that are a huge pain to fix. Many core ecosystem tools still don't support things properly.
CJS, Please recommend people to ship more ESM
CLI apps are hard to make. Arguments aren't passed well and interactive components are hard to make.
Coding tutorials never show how to code well for handling errors.
Cold starts in lambda, a modern standard library (utility packages like execa), ESM and CJS interoperability (soon)
Complexity due to there being too many ways to do the same thing. An example being configuring package.json, there's so many settings and ways to configure it it's hard to know what is the current best practice.
Conflicting node versions for developer environments.
Corepack and its documentation needs to be improved. I'm using it daily and it's still not clear to me what command does what.   You are doing an amazing work. I love NodeJS
Could be interesting an built in es-lint / prettier in node.js like that node js code could have the same standars everywhere
CPU and memory management

crashes (SIGSEGV), slow execution, high memory usage, annoying incompatibilities between versions and operating systems
Debugger can sometimes be unreliable / slow, though it's gotten much better than it was a few years back.
debugging out of memory problems
Debugging TypeScript is always difficult as sourcemaps are often out of date. It's not clear to me how to keep them in sync.
dependencies issues
dependencies to libraries other than C++. In some environments it is not allowed to download anything needed.
Despite small annoyances dealing with the ESM/CJS "thing" periodically no! I think Node.js is pretty great.
Diagnosis of event loop lags is not an easy task on a large production application. A new use-case, not correctly handled can quickly have huge impact on the whole running application. Some more advanced tooling might be integrated in Node: maybe some alerting / logs that fire when a function is too blocking for the event loop that helps fixing it early
Difficulties using import statement because is a pain to import using string concatenation based on path specified in variables
Difficulty in managing different versions of Node.js on Windows.
Docker images and @types/node are always lagging behind releases, with days or weeks delay. It makes it hard to know when the ecosystem is ready for the upgrade.   I'm aware the Docker delay is due to the Docker team not merging the PRs right away, but I think @types/node could be supported officially by the Node.js team (even if in another package).
Don't know yet but I think I will have issues
Error handling
Error handling for promises, and anything involving running scripts or building stuff for my node project that involves having a non-JS build pipeline e.g. build-essentials on Ubuntu or xcode tools on Mac. Naturally this is more part of the NPM ecosystem, but I digress.
Error messages are sometimes unclear and not pointing to the right place of the issue.
Erros while using fetch (undici) with aws lambda
ESM only packages can't be used in our microservices. Because all of them are CJS.
ESM support, documentation. Almost all alrticles, docs are in CommonJS.   Simple example apps aren't easily available. New learners feel it overwhelming to have to decide how to do everything on their own.   Like dependency injection, project template using express and connecting 10 resources - how do you bootstrap that ?
ESM/CJS and TypeScript are by far the biggest pain points. As a user, I would love it if ESM and TypeScript "just worked" without the need for additional tooling (like Deno and Bun).   As a library author, publishing TS packages is a pain. I'd love there to be some toolchain (ideally built into Node or NPM) or even some documentation (provided by Node or NPM) which allowed me to simplify the process of publishing packages which support both ESM and CJS.
esm/cjs incompatibilities (before the newest feature allowing you to require/import across module types). if a package upgrades to esm, you didn't have an easy way to keep using it from cjs
Esm/cjs is the biggest right now. Love the release cadence though.

Experimental code coverage does not work with typescript files and tsx (have to use nyc instead)
Explicit resource management support.   URLPattern support.   Bad document.   No way to listen to a key combination in raw mode. It is simple that I cannot listen to the ctrl-c keystroke in raw mode.   Cannot properly shutdown
fetch() is annoying in Node.js, I always tend to use fetch-retry or axios
file descriptor limits in windows when building nuxt
Find memory leaks
For an Api server application, more and more we're starting to run into noisy neighbor problems with CPU intensive code.
for deployment i dont have good idea how we can maintain- server crashing, monitoring, logging
Forcing of use new import type,Its so painfull to put additional lines in npm package.json just to run temporary code. It slows rapid development.
Fully typescript projects don’t run code natively
having to compile TS
Having to use other tools to just run typescript code is a headache. Wish node could replace tsx, ts-node, etc
Hell to use node-gyp and native compilation for package argon2, bcrypt
Honestly, on. For now, it is being so good I do not want to stop using Node.js for its features and incredible performance
horrible typedefs in the core modules such as events and streams
How to effectively use typescript with node. The build steps hurt adoption.
how we ship our backend app it would be nice if we compile it into a single js file and get rid of node modules folder in production
I am trying to keep an eye on other existing runtimes such as Deno and Bun. What I like about Deno is the fact that it can transpile TypeScript on the fly and it can generate single file binaries which is awesome if you plan to build CLI tools.
I can't use both Corepack and globally installed pnpm/yarn at the same time. Note that not all my projects are using Corepack.
I do not besides I wish I could do more to help out the core team!
I do not like inconsistent API, for example: - node:timers/promises.setInterval - node:events.once and .on vs EventEmitter.once and .on - import/export
I don't encounter any recurring issues but I do hope more easy workaround with typescript
I don't like the assumptive install of NPM in the native installer. I don't use it, if I use a package manager (which is not often for security reasons) I use others. Ive had too many problems with NPM, NPx.
I find it very hard to identify memory leaks location. In general I find the tools for Performance lacking
I hate package managers.
I know It's not node but bundling is still difficult
I like Nodejs Echo system, had some roadblocks due to lack of nodejs knowledge.
I love it. But startup times could be better.
I love node. Honestly, the biggest issue is with NPM, ESM, and how web, backend, and full stack engineers can possibly know everything they need to to use Node safely, securely, and

effectively! But that isn’t as easy as one would like it to be. Onboarding and new projects can be overwhelming.
I often see warnings about deprecated features from libraries I'm using, which I do not have any control of.
I really like the out of the box typescript support from other javascript runtimes. That is probably the only reason i'm sometimes using other runtimes.
i require a sound type system and find tsx to be a pain to use. this is why i switched to bun.sh for the most part
I still have some hard time to execute TypeScript using a module loader. Also the DX of `node:test` output is not so good...
I think is great, but I miss JS decorators (method arguments decorators are included) pattern matching and immutable structures from tc39
I think Node js is slow
I think node should stay node. A good standard lib in js with a rich module ecosystem. NodeJs should _not_ include test runners, mocking functionality etc.
I think that the most recurring issue I have encounter is the observability. Coming from PHP, where a simple agent installed from a Dockerfile is enough to trace everything properly, I feel blind.
I tried to contribute to the node project and it was very difficult to help guidance in its slack.
I tried to use the node:test runner, but gave up due to missing test file pattern matching - especially when writing tests in TS files. If I pick it up again, I will probably use a wrapper like “borp” but I am hopeful to use the node:test runner without any wrapper in the future.
I want to seeing the node.js as a fastest   and efficient and extremely scalable runtime .
I wasnt even aware of the new features, I love this survey just for that. Screenshoted the list in question 19 and 20 and sharing with the team.
I wish mocha handled ESM with —watch. Switching all the tests to node native seems overkill
I wish node have an built in SqlLite/Other local DB   (that does not depend on node gyp like most SQLite’s on npm)
I wish that typescript was better integrated into node. I don't understand why I need to be so aware of my code being transpiled rather than node just dealing with it. It feels like most people in the community wouldn't recommend starting a new node based project without typescript and yet a lot of node tooling feels like it wants to create separation from typescript. As an outsider, I don't understand but perhaps there are good reasons.
I wish to see the TypeScript-as-Comments proposal advance to stable soon.
I would like to have cancellable promise implemented natively
I would like to have opportunity use Typescript built-in without lost performance
I would like to see more Web Standards in node Streaming files using next.js caused some issues when converting node streams from fs to Web streams. The usecase is not well documented
I'd like have better feedback on the Web APIs and how the node.js different versions support them
I'd love the REPL to support ESM

I'm really interested in projects like https://github.com/stdlib-js/stdlibt that aim to bring more mathematical computing into the javascript world. I'd be very interested in seeing the JS/Node ecosystem match the Python/NumPy world.
I've found that errors can often be undocumented—which causes problems with things like the node:fs library which can throw an error for any of a thousand unpredictable reasons.
I've tried to get core-pack going twice before on an existing project going from yarn v1 to yarn v4, and gave up both times after multiple days each time. I put this blame mainly on yarn since I was following their migration documentation, but at the same time yarn v1 & npm just don't seem to be that easy to leave from. In saying this, I have not tried it for any new projects and would probably have a much better time if I did. But the pain point is still there.
Import issue from windows to linux because of case of file or folder name
Import statements in the REPL.   No way to turn off backtrace (a lá Emacs Lisp's `debug-on-error`) on error.   This survey including East Germany yet not Taiwan in the list of residence locations.
Importing moudule with query parameters is an important feature in some cases. Although Node.js has been supporting this feature for a long time, it cannot be intelligently recognized in VSCode.
Improve DX
In large monorepos, POSIX/Bash shell scripts don't cut it because it's difficult to maintain them. JavaScript, although much better, isn't easy to maintain either because of its super-dynamic nature.   So I write TypeScript... and if I want to use it with NodeJS, it introduces an extra build step (I don't use ts-node or similar hacks because they are very problematic with ESM code). For this reason, I stopped using NodeJS to run my monorepo automation scripts, and I switched to Bun (I prefer Deno for its permissions model, but it does not support "worskpace:*" version specifiers in package.json files.
In the present AI world. I think we need some ML library to deal with data processing. Though we have stream, still this is little bit lower level to deal with data. I wish we can leverage our developing boundary by using ML/DL module.
In the repl, pressing Enter (to execute the code) should only take the current input NOT the suggestion (right arrow or tab should be used, as normal, to accept the predictive suggestion)... Otherwise the only way I've found to execute what I've written (not the suggestion) is to add a semi-colon or space
Incompatibility between CJS, ESM, TypeScript
Inconsistencies between browser APIs and Node APIs, as well as having Promise-based APIs instead of nodeback-style APIs. Both have been improving over the past couple of years. I've also repeatedly had issues with watching in development setups on Windows where the process can no longer be terminated with a command line signal.
Incorrectly configured workspaces/monorepos and typescript builds.
Install scripts need a better path to use binaries. Bettersqlite is a great example of the current mess things are in. Seems like platform specific napi binding files should be an out of box option without resorting to the multi-package strategy
Instead of having to set the maximum memory allowed at the start, Node.js should use the system free memory value as the limit. That or allow to change the maximum memory allowed during runtime. This is an issue for games running in a Node.js instance in the client machine.

ipv4 and ipv6 issue "net.setDefaultAutoSelectFamily" it has to be set to false as it's resulting in node socket error and timeouts of Task Queue
Issue with running Typescript. Probably because loaders have been a moving target.
It is best if Node JS supports Interfaces in-house than depending on TypeScript. Have most of the advantages of using Typescript, or other libraries with Node Js itself. Depending on many libraries makes the project unnecessarily complex for basic things
It is hard to discern the documentation for native fetch and node-fetch. People get confused about this and don't know there's a difference.
it is not downloading and working on vs how it will be
It will be good to embed some async runtime like python trio. EventEmiters pattern in userspace code bad idea, IMO. This leads to distributed FSM of many files and many event handlers which dificult to understand.
It would be good to see event loop blocking detection because it isn't always easy to find where the event loop is blocked.
It would be really nice to have Typescript by default like bun/deno.   Hot module reloading in NodeJS seems either unsupported or unreasonably difficult. Being used to this on the client side it feels very absent on the server (although built in watch is a great step towards this -- thanks).   NPM very obviously should not run install scripts by default.   I strongly believe Node needs to be opinionated about ESM only by deprecating CJS, providing tooling to help with the switch. Running things with the TSX module seems to do a good job here.   In general I wish NodeJS was more opinionated & willing to do a one-off "breaking changes" release: - choose between CJS and ESM = a fragmented disaster. We need Node to lead the community here   - TypeScript in the same way as deno/bun. So many pain points have to do with build steps/bundlers/stack traces. It would be great if the code we write feels more like the code that runs   - disable NPM post-install scripts   Great work on building in .env and watch.
It's incredibly hard to track down and resolve deprecation issues like the following on which happens since node 21:   (node:8) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead. (Use `node --trace-deprecation ...` to show where the warning was created)   1) Why not display the causing module in the console message? 2) Why not provide a link to the issue/PR where the deprecation was discussed and first included (https://github.com/nodejs/node/pull/47202)? 3) Even better, provide a central place that collects the deprecation notices
It's not exactly a problem, but a better algorithm within npm to get the attention of users using other existing package managers would be welcome.
It's unacceptable to have semver-breaking releases of Node.js.   I've started phasing out my usage of Node after I ended up in a situation where I couldn't deploy a project, because the oldest working version of Node.js was not compatible with an old dependency I used, and a newer version of the dependency required extensive code changes.   I want my old Node projects to continue running forever, while allowing me to update the Node runtime without code rewrites. Rust can do it thanks to Editions. Node can't.   Insecurity of old dependencies is an issue, but that's another major broken thing about Node: CVEs are full of junk, and CVSS is crying wolf by design. It's impractical to pay any serious attention to the 'vulnerability' reports when they're clogged by ReDoS and are based on naive coarse-grained checks that don't check whether I actually use the vulnerable functionality.

Its really painful to have to compile typescript before running it. Almost everything uses typescript nowadays, from infra tools such as CDK/CDK8S/CDKTF to framework and bundler config files, and especially, scripting tools. TypeScript support (without typechecking) would improve DX a lot.
just add an easy and better way to use ESM, and thank you for all your work
Just having trouble loading files using .load
Just make it faster so it can replace other old languages
Just the lack of good ESM support and the need for tooling to fix the lack of proper module import support in NodeJS.
Lack of dependency injection
lack of HTTP/3 server and HTTP/2 request client
Lack of interop between ESM and CJS modules is a rapidly increasing problem that may impact our ability to use Node.js.
Lack of native typescript support and node-gyp errors
Lack of typescript
Let is be honest, Node.js is wonderful. I have issues, but mostly because of TypeScript behavior and ES/CJS compatibility.
Long live Node.js!
Lots of configuration for typescript for dev/test/prod that requires a lot of tinkering and loads of different properties. Should be as easy as in bun.js
Mainly the way errors are handled isn’t really enforced.
make require(esm) stable and unflagged as fast as possible and backport it as far as possible
make something with versions... i dont want change always versions! nvm n fvm volta etc.... why is so hard to just make easy to use for every one
managing dependencies with mutiple versions of node.
Managing versions for different projects on different versions of node. It's also not available out of the box via package managers like apt.
Massive memory consumption due to memory leaks caused by jest-node-v8 interactions in a project with 10k+ tests
mdb publish
Memory / heap size control.
memory leaks in fetch. still really hard to run in production when server gets to many request it goes into super slow mode instead of stop accepting new requests
Memory leaks, and slow responses because of event loop delays -> because of too much back pressure
Missing android IOS support
Missing guidelines and good practices on how to write server side JS. It’s always either written as Java or as front end cose
Module resolution and integration with Typescript is tricky to migrate existing projects
Modules that throw errors outside the normal thread of execution. The moral equivalent of: setImmediate(() => {   throw new Error(); });The uncaughtException event is a big hammer and inappropriate for most uses.
Modules vs commonjs
More memory consumption

Most common issues I face are either solved quickly or there is usually an easy workaround.
Mostly observability. When deploying on a server, the developer has to bring their own logging and manually configure certain settings like tracing and stackframes.
My main pain point with Node.js is the extra complexity that comes with using TypeScript. I often wish that Node.js ran TS natively so the entire developing/building/deploying/running process could be much simpler. I know that's a tall order. I also know there are tools like tsx and ts-node, but those are an additional dependency and are often slower. In general, I would like to see closer cooperation with TypeScript in the future.
N/A
N/A
N/A
n/a
n/a
N/A
NA
NA
native fetch (unidici) has had problematic bugs keeping me stuck on node-fetch, which I'd rather drop
native modules always seem to break most often
Native support for typescript files would be nice like bun has
native typescript on runtime
native typescript support
Native typescript support would be great!
Nay, all good ☺
Need native typescript support
njm
no
no
No
No
No
no
no
no
no
No
No
No
no
No
No
No
No
No

No
No
No
no
no
No
No
No
No
no
no
no
No
No
No
No
no
no
no
No
No
NO
No
No
No
No
No
No
No
no
No
No
No
no
No
No
No
No
No `Set.prototype.intersection()`   )`:
No I don't find any type of issues using the node js lts version
no issue
No particular issue. In fact is very very stable. But bun.js (that still has some incompatibility) is easier to work with but node is catching up. Please take inspiration from it and there will be no need for other runtimes

no single contributor should be able to tank a PR. consensus-seeking doesn't scale, and Node.js should experiment with different technical governance. maybe that looks more hierarchical and less democratic, or team-based (CODEOWNERS style).   TSC members should probably have term limits. there should be a limit to the number of TSC members, and they should be countable on one hand. it might need to look more like a republic, where each team sends a representative to the TSC.   I don't know. but the project should experiment with different ideas, and the current governance makes experimentation difficult!
no, cuz its just perfect
no, i dont
No, I haven't.
No, I love Node, it is the best thing that has ever happened to web development. I look forward to many more years of next level productivity.
No, it's fine
No, node is good :)
No, not yet.
no, regarding stability, Node.js is better than Deno and Bun
No.
No.
No.
No. Good Job !
Node and all of its tools should be able to support typescript out of box without any additional configuration.
Node doesn't ship with it's own version manager and a means to update itself from the cli, yet it ships with npm out of the box. Sure, npm is there for historic reasons, but maybe it shouldn't be and maybe a node version manager should be?
Node is awesome and I believe heading in the right direction but please please improve NPM. Overall experience of NPM can be vastly improve if given proper attention.
Node is getting better over time, now I mostly use defaults things are nicer than 10 years ago.
Node is simply the best 👌, I can't even think of any issues
Node should be able to interpret typescript files out of the box
Node significantly. Start-up times can be an issue in serverless.
node-gpy出现问题且基本无法解决(Node-gpy has encountered problems and is basically unable to be resolved)
Node.js 22 breaks a lot of stuff.
Node.js itself is rarely ever the pain point of issues I encounter. Normally it comes down to some part of a pipeline updated to support a newer thing and every other part is not updated or updated in some way that brakes ever other tools that have updated...
Node.js not designed to be run as PID 1 results in needing an init container to handle process signals correctly in Kubernetes.
Node.js version management provided by tools like `nvm`, `volta`, `fnm` and others, all have the problem they don't fully work under certain conditions. - `nvm` breaks when using `pnpm` because it doesn't work with some parameters `pnpm` sets.   - `volta` sets the Node.js version globally, so you cannot work with one Node.js version on one terminal and with another version on another terminal. - `fnm` is the best so far, but it also has problems related to `pnpm` usage.

The `pnpm` used and updated is not associated with current Node.js version set.   I'd like better ecosystem tooling by default when installing Node.js, without having to use any external tools. A good package manager, like `pnpm` should be the default, no more `npm`. And a good version manager should also be included with Node.js.
Node.js' Fetch never seems reliable enough to use in production. No issue particularly stands out but I usually resort to using node-fetch or axios
Node.js's consistent excellence is truly inspiring, as it continues to demonstrate its awesomeness time and time again.
Nodejs does not fully support Typescript compared to Bun when config. I'm trying Bun and it is much easier than Nodejs when working with Typescript
NodeJS is marvelous but it has some painpoints related to setup a modern (or better said current) JS project that competitors resolved very well: built-in support to Typescript, ESM is first-class citizen, etc...
nodejs need a better cross thread memory sharing solution. Atomics is too weak for the real world. It even doesn't support floats.
None
None
none
none
None
None
None
None - extremely happy with platform, toolset, deployment.
None besides out-of-memory errors when running bundlers like Rollup in large-scale projects...but that's not really Node.js's fault I guess
None since finding the tsx package. Lovely stuff, your work and that package <3
none so far
None.
None.
Nope
Nope
Nope
nope
Nope
Nope. Keep up the good work.
Nope. Keep up the great work everyone! I really appreciate you all.
Not
not at the moment! Works great for me :-)
Not especially, but sometimes new major releases break things.
Not from Node but from npm as a package manager. It duplicates a lot of space and is really slow on it's operations
not really

Not really Node's fault, but ECMAScript lagging in terms of making new language features stage 4, and consequently affecting many tools who use that as a threshold for implementation/support (E.G. eslint)
Not really. It's a joy to work with and I'm so glad that – unlike other alternatives that shall go unmentioned – Node is stable and reliable and isn't trying to go serverless or working towards a venture-capital-funded exit. In other words, I am very happy with the total lack of ens********n in Node.js and I hope it'll stay that way :)
Not really... I wish the ES transition happened earlier, but I generally think the Node.js project is doing a pretty good job.   I just found out about some of the abilities in v22 (especially built-in `--watch`!) and I am excited to give them a try!
Not yet
Not yet
Not yet.
Not. Nothing.
npm i should save exact dependency by default
npm install issue sometimes,
npm libraries shipping separate .d.ts file and .js file is just crazy. Why don't just support typescript out of the box and save us from this debugging hell.
NPM Package is relatively bigger than the universe. Make it small.
npm shipped with node.js 20.x.x is unstable (almost unusable) on Windows.
nvm doesn't work. no time to evaluate the 10 competitors. I tried to use corepack but didn't work and the doc is so poor I couldn't understand anything
One of the strongest point of node, it is one weak point of him. This is the variety of packages that community does, when someone just have arrived at node doesn't know what to use. Could be good some recommendations of npm packages.
Open SSL Error is the most common and frequent node.js error to me when ever I try to install packages
Out of memory errors (have to use --max-old-space-size, etc.)
Outdated examples. Use clean async style and ES6
packageManager field expects exact version number instead of version range !! This is really annoying in practice now that the ecosystem is starting to adopt it (netlify, pnpm, etc.)
packaging application. for ubuntu (rewrite two times). Unstable behaviour after heavy load (does not respond to request)
performance and memory consumption is still very high, especially compared to bun or deno
Performance and memory foot print
Performance could be better. Support for newer ECMAScript features could be better. APIs that don't match the web standard version.
Performance of asynchronous code, particularly in asynchronous GraphQL resolvers and when using async hooks (AsyncLocalStorage) for observability/tracing. To be clear, the bottleneck is _not_ I/O such as DB queries, but rather the overhead of async code itself, as described e.g. in https://www.softwareatscale.dev/p/the-hidden-performance-cost-of-nodejs and https://github.com/nodejs/benchmarking/issues/181
Performance tends to be abysmal. Memory leaks with known reproductions occasionally get ignored for long periods of time

Performance with promise-caused garbage collection.   In `node —test`, when you mark a single test within describes with `.only`, you need to also mark the parent describes with `.only`, which is cumbersome.
Please add a better documentation. It's quite hard to navigate. First thing would be to index the doc. Then maybe add a LLM to answer questions 😎
Please come to an agreement with Bun and Deno. Please no drama like with io.js.
Please deprecate `ts-node`. It's really out of date but a lot of newbies are still using it.
Please keep the API of Node.js as simple and straight forwards as possible. This war for all the years the good part. Nothing was abstracted away and was only usable after initializing dozens of objects. Keep it this simple. Thank you.
please make a easy one shot tutorial or guide so that i can learn node from bottom to hero with a best level of understanding and working on commands
Please make nodejs faster . type safe
Please Please consider a GoLang/Deno type setup with Default linting, testing, formatting and error guidance like the above languages/runtimes. (Can be done in the same way Go/Deno does it with a VS Code Extension.)
Please relax esmodules so they're more flexible
Pretty happy with it. Thank you!
probably nothing that couldn't be solved with turbopack and better project structure.
Production server taking more memory. And going down when more requests are coming
Profiling async code. Due to its nature, the chrome Dev tools are a poor fit. 3rd party APM tools do the job but usually cost a pretty amount. Writing instrumentation code yourself isn't hard but visualisation is
Punycode depreciation warning
Running TypeScript natively is a pain
secure
selecting between npm, yarn, pnpm, etc.
Setting up a typescript monorepo is hard especially if is both front end and backend. We need typescript natively supported and more standard JS APIs we find in the browser, as Deno is doing
Setting up and managing nodejs with Typescript and deploying them is still too tricky. Do one use tsx or ts-node? What about jest and typescript? Nodejs is not isolated, it's at the center of the ecosystem.
Setting up TS is tedious. I would appreciate if the experience was better.
Setting up TypeScript is always a pain.
Setup to run TS is a pain. Having a project be ESM while importing a package that is CJS.
Single file build scripts fail to run due to esm/cjs conflicts when importing dependencies from other workspaces in larger project.
Slow adoption of next-generation features. As a whole, NodeJS is doing fine, but projects keep popping up because people expect the adoption of key technologies for a long time. The fragmentation of the community is getting out of control.
Slow cold starts
Slow on Windows

So many ways to install so many ways to use it, I would have loved a more clear way to setup a project from beginning to end. But having used it now a few years I don't have to actively think about how to setup a project things work.
Some memory related issues. The memory usage is a bit high. In particular npm memory usage, for simple tasks, tend to be too high
Some stuff are enabled only by experimental flags, but these flags aren't consistent across Node versions.   like --experimental-loader=, --es-module-specifier-resolution   which makes it difficult to migrate project to newer version of Node.
Sometime, the socket release is not happening, so the sockets usage will be reach to the top limit after a few minutes, especially using on Lambda, some packages will create socket connections but never try to close them, then we have to reboot Lambda instance every few minutes, by monitoring the socket counts.
Sometimes I wish I could write typescript without needing some sort of compile process. I guess maybe that means I should try demo or bun.
Sometimes when im trying to add more secure API's, Node.js just doesn't work until I restarted it.
Somewhat bloated install size, especially npm with its own gigantic node_modules
Speed of rebuild (switched to bun)
Speeeeeeeeeeeeeeed. Too slow.
Still have periodic crash due to the heap memory allocation and have not found the source of the issue yet neither with debugger nor with Clinic.js. so whatever can be done to extend information when crash occurs would be great.
Still haven't encountered any serious issues.
stream/web vs stream/node should be compatible when it comes to typescript.
Suffer from package dependencies
test runner globs don't work code coverage for all files does not work
Testing app with the famous esm/cjs is really a pain sometimes. We believe in your job guys!
the biggest issue is JS, s**t language with not existing stdlib working terribly slow but world seems to be happy about it.
The complexity of setting up a project for Node.js. When to use TypeScript + NestJS.   I would love to use the standard HTTP server features of Node.js without the extra dependencies of frameworks. But such functionality takes too long to develop. And there are ready-made solutions.   I would like an alternative from the Node.js platform that replaced Express, Fastify or NestJS.
The documentation sucks 😅 you have to jump around all the time, default values and behaviors are not always super clear, and there are very few examples.   Nowadays, for documentation I use 70% AI tools and 30% official doc
The error handling when running in non optimal environments (Low on Disk, Low on Memory, Overloaded CPI) can become frustrating as node seem to have a habit of shooting itself in the head rather than waiting for resources to become available.
The ESM module system and "exports" continues to change far too quickly. node should prioritize perf, robustness, security, and avoiding breaking changes, rather than prioritizing new features.
The ESM vs CJS schism is something I really hope we can put behind us ASAP.

The event-based environment is killing me because there's too much of them. I wish Node would become something more like Python.
The insecure and unstable npm package ecosystem
The lack of native TypeScript support, stable ESM/CJS interoperability and assorted tooling has been very painful over the years. While everything works on paper, deviation from the beaten path or an encounter with an old/misbehaving library during node upgrades often leads to unpredictable, frustrating and completely unnecessary loss of time as we're forced to climb the usual Jenga tower of dependencies and config files to get the hydra to dance to the beat. When it works it works, but it's a constant reminder of how little the Node.js project seems to care about developer experience beyond the comfort of their own codebase and their own issues with third party dependencies. Your inability to properly coordinate with Microsoft, the authors of top libraries and the rest of the community to make rapid, effective strides beyond the stagnant and ossified state of Node.js development - despite constant pleas from the community as should be evident from the many surveys - will effect a slow and quiet decline of the runtime as people flee to Bun and other competitors. We want to run TypeScript without loaders or intermediate steps, we want to mix ESM and CJS modules and dependencies at will and we want everything to be faster.
The main recurring issue with NodeJS is code rot over time. A project built 1 or 2 years ago has a chance to just not build today. I'd like to see some stability where older projects can just be built even on newer versions of NodeJS.
The need to use a Node version manager on every machine. Without this, uninstalling or upgrading Node is a mess, especially on MacOS. Also, Node as a single file would be a good addition.
The node community reinvent the wheel so often that's hard to keep up with the development of node itself.
The only issue I really have had was with the US politics on the webpage. That made me consider ditching node for deno at the time.
the persistent insistance that every dir and env / cli and or general js project or any of the above that utilize javascript at all want all the modules if any at all causing unneccesary bloating and catastrophic interoperability issues, so unless you specify you don't want it and are aware its weesled it's way in there quietly in the background in the first place it creates resource intensive and time consuming troubleshooting situations
The platform is great for helping me get my job done.
The setup and understanding of ALL the tooling (webpack, eslint, prettier, typescript, jest, ...) is sometimes hard (not to talking about containers, git, ci, cd, databases, ...) for me as a senior but junors are lost. NodeJs is not hard, but the ecosystem.
The stream package is easy to get wrong. Hard to debug unhandled exceptions and unhandled rejections.
The whole ecosystem around node is a complete and utter mess changing every couple of months pulling in hundreds of practically un-auditable packages, nothing has any kind of standards body defining interop between ecosystems, everything has to be written new and from scratch with yet another way of configuring stuff without enough effort put into just porting work into existing projects. The worst part is now Bun being on the scene and again 0 coordination between node and Bun to make sure stuff like macros and the shell api stays

interoperable between runtimes, really just how the rest of the JS ecosystem behaves too. There is a desperate need for maintainers to come together at a table and actually define standards like Ecmascript does it
There are bugs in Undici and few other places here and there. (all reported)
There is no problem with node. We the developrs have skill issues.
There should be some way to obfuscate the code for use in electron js
There were no noticeable bugs for me in these two years of usage.
They're all my fault.... I mean, no.
this is one of the few amazing products that has only gotten better with time like a fine wine thank you!!
Too tightly bundled with npm
Top level await
type definitions in @types/node are not updated at the same time as node, thus preventing me for using new features I want
Typescript
TypeScript + Node.js is still not as friendly as I like to see it. TypeScript support out of the box would be really nice.
Typescript configuration is too complicated when initializing the project because it does not directly support typescript from the beginning like bun.
TypeScript interop out of the box would be nice, it's the main pain point. ESM - CJS interop needs to be improved (e.g. require(esm) is a great step), but also need a clear vision from Node.js to soft-deprecated CommonJS.
typescript on native
Typing differences with web, making portable library typings a problem (setTimeout not returning number as the silliest and worst example)
Unclear syntax for async context creation. Body parser compatibility.
Unhandled Exceptions in Streams: Errors can occur at any time during a stream’s lifecycle, and if not properly handled, they can crash the application. Callback Hell: Also known as “Pyramid of Doom,” this is a situation where callbacks are nested within other callbacks several levels deep, making the code hard to read and maintain. Asynchronous Operations: Managing asynchronous operations can be tricky, especially when dealing with recursion or when a certain sequence of operations is required. Memory Leaks: Inefficient code or not properly managing memory can lead to memory leaks, which can degrade the performance of the application over time. Blocking the Event Loop: Performing heavy computation or synchronous operations in callbacks can block the Node.js event loop, leading to performance issues. Error Handling: Proper error handling is crucial in Node.js applications. Without it, a single uncaught exception can stop the entire server. Version Mismatch: Compatibility issues may arise when different parts of a project use different versions of Node.js or when deploying to an environment with a different Node.js version. Package Management: Dependency management can become complex, especially when dealing with a large number of packages or when packages are updated frequently. Security Vulnerabilities: Node.js applications are not immune to security risks, and developers must be vigilant about securing their applications against common vulnerabilities.
Update my preferences

Upgrading is a giant pain. It's the way that dependencies are so granular, which results in a sprawling dependency tree. We have code running Node 12 which will be retired before it will be upgraded to a newer Node version.
version issues
version issues
Version maintenance is hard, I would like to keep longer and more stable versions instead of having to update every year
watch mode didn't work for me, it ran the script but it didn't behave the same way as the normal script running.
watch mode takes more time as compared to nodemon
we all have issues, but rarely do I see one with so little issues as node.js great work thanks for the help and making my life easier.
We need native typescript support in node.
We spend more time than other teams at our company maintaining our build system (frontend and backend) and package dependencies but we are very happy with consistent improvements in node.js/Typescript. Our anonymous internal surveys show that our developers feel the most productive and happy about the tech stack.
We target serverless infrastructure, so we have little to no control over command line options that enable more or less experimental features. We also cannot use the latest version of Nodejs when it comes out, we have to wait for the infrastructure to start supporting it. This compounds to very long wait times for new features. I would love for Nodejs to focus on a stable core that actually finished features rather than releasing them behind flags for four major versions, and to move library features (like fetch, test, websocket client) out of the core so we can enjoy them as we see fit rather than waiting for our infrastructure to support them.
We would like to add node canvas feature for graphic visibiliy like python matplotlib .. plot data ..
We would really like to spawn processes using a lesser permission user.
We're currently experiencing slow startup times of our monolithic application, which is difficult to assess/profile.
Web server built using Node.JS is a little bit slow...
web streams are extremely slow on node.js
Well, that's basically not a serious problem, btw I've tried bun, if bun make this form, maybe I can answer it
when i tried to upgrade my node version to v20 then I'm getting many errors which I don't understand how to understand and resolve
When installing with PNPM, it always says "listener numbers exceeded, may be a leak"
When installing, that npm or node just is not in %PATH%, even when that setting is enabled in the installer.
When something goes wrong at development, logs shows errors belong to node.js itself instead telling what happened exactly.   ``` const fs = require("fs"); const readStream = fs.createReadStream("nonexistent-file.txt");   stream.js:60   throw er;   ^ Error: socket hang up   at createHangUpError (_http_client.js:200:15)   at Socket.socketOnEnd (_http_client.js:285:23) ``` For example at above, at least i would like to know which part of my code is causing this error, instead seeing that internal details about lib itself. Like this: Access

to fetch at 'http://example.com/api/test' from origin 'http://localhost:3000' has been blocked by CORS policy
When using starter projects on Github it's less of an issue, but when building a project from scratch, the build tools, their config, and the project settings can very easily conflict in unintuitive ways.   Specifically with build tools, there are so many string values for properties that I just don't even know what they do.
Windows watch mode bugs
With node:test running locally or in CI, coverage reporting is unreliable. Experimental coverage reports were missing files while c8 frequently just crashes the test process. Result: CI runs without coverage report. Coverage needs manual review.
Would like to see the adoption of types as comments ES proposal to allow the native consumption of typescript files without a transpilation step like Bun provides.
Y.
Yes
yes
yes
Yes
Yes
yes i share
yes unable to use the powershell in the Visual studio code, And unable to use newman library package in the cmd. Unable to api collections.
Yes whatever this is keeps popping up in all my apps. Currently using node 22.2.0 up from 20 (node:9440) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead. (Use `node --trace-deprecation ...` to show where the warning was created)
Yes!
You are doing a great job, thank you for your work!