CASE FILE · OPS-01

A Plain-English Field Guide

XOPS

FOR DUMMIES

INSTALL · ASK · RUN · SHIP

"xops doesn't replace npm or git — it decides which one should handle what you just typed, in plain English if you'd rather not remember the flags."

A friendly, technical walk-through of **xops**, the operating layer for Node.js repositories: how it discovers your packages, how it routes commands to npm, pnpm, or git, how `xops ask` resolves plain English deterministically, and the everyday flags that make it worth using instead of memorizing three tools' worth of subcommands.

`xops ask` `passthrough` `xgit` `doctor`

FILED: JULY 2026

STATUS: FOR INTERNAL USE

Case File Directory

WHAT'S INSIDE

READ ME FIRST

CHAPTER 0

HOW TO USE THIS GUIDE

What xops is, who reaches for it, and where the release/publish deep dive lives instead.

xops is the operating layer for Node.js repositories: one command surface for the work that happens around npm, git, tests, scanners, release, setup, and code agents. It does not replace npm or git — it detects repository state, routes work to the right native tool, plans risky actions, executes approved workflows, and reports what happened.

This book covers the part of xops you touch every day: installing it, understanding the command surface, letting `xops ask` turn plain English into a real command, and the flags that show up in ordinary work. It deliberately does not go deep on publishing, release playbooks, or the package graph — that's *xops Release Engineering & Playbooks for Dummies*, the companion book in this series.

SOURCE MATERIAL	WHAT IT GIVES YOU
Install & Setup	Getting xops on a machine, troubleshooting PATH/EACCES, and the doctor command.
The Command Surface	Reserved xops commands vs. passthrough to npm/pnpm/git, and how xops tells them apart.
xops ask	How plain-English requests resolve deterministically to a real command, with approval gates.
Everyday Flags	The flags you'll actually reach for day to day, and how package selection works.

The stamps in the margins

TIP

Practical advice — a shortcut, a convention, or a habit that saves you a debugging session later.

REMEMBER

A rule that's easy to forget but expensive to get wrong.

WATCH OUT

A trap. Something that looks fine, compiles fine, and is still wrong.

TECHNICAL STUFF

Deeper detail you can skip on a first read and come back to when you need it.

FIELD NOTE

A cross-reference, an edge case, or extra context worth a second glance — often a pointer to the release/playbooks companion book.

Who this is for

You're a developer working in a Node.js project or monorepo who wants one command surface instead of memorizing npm, pnpm, and git's separate subcommand vocabularies — or you're a code agent operator who needs a deterministic, approval-gated way to resolve "publish everything" into an actual argv list. You don't need to memorize every flag; you do need to know how xops decides what to run, and how to ask it in your own words when you don't remember the flag.

TIP

Everything in this book applies whether you invoke the binary as `xops`, or its compatibility aliases `xnpm`, `xgit`, `x12i-xops`, `x12i-npm`, or `x12i-git` — same engine, same package, different day-to-day ergonomics.

CHAPTER 1

WHAT XOPS IS (AND ISN'T)

One operating layer, three interfaces, and a hard boundary around what it refuses to own.

The operating layer, not another package manager

`npm` is the native package manager. xops is the operating layer for Node.js repositories: it detects the tools already present in a repo (npm or pnpm, git, GitButler, test runners, linters, security scanners, dependency bots, CI), reports what's operable, recommends missing capabilities only when repo evidence supports them, and applies non-seamless setup only after explicit approval. Every native tool keeps owning its own domain — xops owns the operating contract between them.

> **REMEMBER**
>
> xops does not implement its own package manager or dependency resolution, and it does not use an LLM for `xops ask`. Resolution is deterministic catalog matching, every time.

Three official interfaces

`@x12i/xops` exposes the same operating layer three ways: the **CLI** (`xops`) for humans, a **TypeScript SDK** (typed functions like `detectRepository`, `runCheck`, `runRelease`) for scripts and dashboards, and an **MCP server** (`xops mcp --stdio` or `xops agent mcp --stdio`) for agent runtimes and IDE agents. SDK functions return structured `XopsRunResult`-style objects with issues, tool runs, recommendations, and stable status values — JSON an agent can consume without scraping terminal text.

PRODUCT NAMING, AT A GLANCE

```
Product name          xops
npm package           @x12i/xops
Daily command         xops
Agent MCP command     xops mcp --stdio  (or: xops agent mcp --stdio)
Compatibility aliases  xnpm, xgit, x12i-xops, x12i-npm, x12i-git
```

xgit: the same engine, git-first

The same package also ships `xgit` / `x12i-git`: the identical engine with git-first ergonomics. Where `xops` defaults to npm-first phrasing (`xops status` still runs `git status`, but the ask catalog checks xops workflows, then npm, then git), `xgit` checks the git catalog first, then xops, then npm — useful if your daily habit is "git verbs first."

> **FIELD NOTE**
>
> Everything about publishing, release playbooks, and the package graph (`xops.json`) is covered in depth in *xops Release Engineering & Playbooks for Dummies*. This book only covers enough of `xops release` / `publish` to recognize them as reserved commands.

What xops deliberately does not do

xops does not replace the underlying package-manager or git binaries — it orchestrates and passes through to them. It does not manage npm login or store credentials interactively, does not create git tags, and does not run operations in parallel.

WATCH OUT

xops does not cover every possible package-manager or git phrase in `xops ask`. An unmatched request fails safely with suggested examples — it never falls back to guessing or running an LLM-generated shell command.

CHAPTER 2

INSTALLING AND SETTING UP

Global install, the one gotcha with sudo, and what to run when something looks broken.

Global install (the recommended path)

On Windows, install without `sudo`. On macOS and Linux, `sudo` is needed because npm's default global directory (`/usr/local/lib/node_modules`) is not writable by a normal user.

GLOBAL INSTALL

```
# Windows
npm install -g @x12i/xops

# macOS / Linux
sudo npm install -g @x12i/xops

# confirm it's on PATH
xops --version
```

Global installation is the primary workflow. You do not need to install `@x12i/xops` into every package you manage — once it's global, `xops` is available from any package folder, packages root, monorepo root, or workspace root.

REMEMBER

`@x12i/xops` does not install Node.js, npm, or git for you. The station needs Node.js >= 20, npm on PATH, and git already installed if you plan to use `--push` or `--create-git`.

One-off and CI usage

You don't have to install globally to try xops on a CI runner or a temporary machine:

NO GLOBAL INSTALL NEEDED

```
npx @x12i/xops --build --test
npx @x12i/xops@latest --build --test --publish --push
```

Upgrading

`xops upgrade` (or `xops --upgrade`) checks npm for the latest release and reinstalls globally for you. It also refreshes already-installed seamless companion tools, such as GitButler's `but` CLI — but it does not install new companion tools during an upgrade just because they're recommended.

WATCH OUT

If the global install directory isn't writable, xops exits with code `3` (environment/permissions, not a package bug) and prints both a `sudo chown…` fix and a `sudo npm install -g…` fix. That exit code means "fix your environment," not "xops is broken."

When something looks wrong

Two built-in commands cover most setup problems: `xops doctor` checks Node/npm/git/xops setup and auto-fixes common PATH issues on Windows; `xops troubleshooting` prints the install/PATH/cache/platform guide directly in your terminal.

DIAGNOSE WITHOUT INSTALLING ANYTHING NEW

```
xops doctor
xops troubleshooting
npx @x12i/xops doctor
```

TIP

xops uses an isolated npm cache at `~/.cache/xops/npm` so installs keep working even when `~/.npm` has permission issues from a past `sudo npm` run. If that cache itself becomes root-owned, xops automatically falls back to `~/.cache/xops/npm-clean` and prints a one-line `chown` fix — you never need a wrapper script that exports `NPM_CONFIG_CACHE`.

CHAPTER 3

THE COMMAND SURFACE

Reserved xops commands, native passthrough, and how xops tells the two apart when a word could mean either.

The commands xops always keeps for itself

A short list of commands is reserved — they always stay on the xops side no matter what: `ask`, `doctor`, `help`, `list` / `ls`, `map`, `map-to`, `map-out`, `scripts`, `troubleshooting`, and `validate`.

COMMAND	DESCRIPTION
`install`, `i`	Fix dependency specs to npm latest, then install and run any requested lifecycle steps.
`list`, `ls`	List discovered packages, versions, and local dependency relationships.
`validate`	Check local dependency integrity and npm registry versions — read-only.
`doctor`	Check Node/npm/git/xops setup; auto-fix common PATH issues on Windows.
`ask <text>`	Resolve plain-English text to xops, npm, or git — deterministic, no LLM.
`npm <args>`, `git <args>`	Run native npm or git directly in the current folder.

Passthrough: xops fronting npm, pnpm, and git

For everyday project work, recognized project and git subcommands run natively through xops, so you don't need separate binaries open in another terminal tab:

IMPLICIT PASSTHROUGH

```
xops run build     # npm run build / pnpm run build
xops test          # npm test / pnpm test
xops status        # git status
xops push          # git push
xops pull          # git pull
```

Project passthrough uses pnpm automatically when it detects `pnpm-lock.yaml`, `pnpm-workspace.yaml`, or `packageManager: "pnpm@..."` in the repo. Registry and publish commands always stay on npm regardless.

How xops disambiguates a word that could mean two things

`xops install` alone and `xops install lodash` are not the same operation — the first is xops automation (refresh in-house scoped packages, install dependencies), the second is a plain `npm install lodash` passthrough. xops tells them apart by whether you named a package.

YOU TYPE	ROUTED TO
`xops install`	xops automation (refreshes `@x12i/*` / `@exellix/*` at latest)
`xops install lodash`	`npm install lodash` / `pnpm add lodash`
`xops install -D typescript`	`npm install -D typescript` / `pnpm add -D typescript`
`xops list` / `xops ls`	xops package discovery (reserved)
`xops npm list`	`npm list`

REMEMBER

If you want to force native npm or git regardless of ambiguity, use explicit passthrough: `xops npm run build`, `xops npm outdated`, `xops git status`, `xops git diff` always work.

WATCH OUT

`xops --publish` and `xops publish` both end up at the xops lifecycle publish workflow (`npm publish` under the hood) — but `xops --test` is xops's own lifecycle test step, while bare `xops test` is a native `npm test` / `pnpm test` passthrough. The leading `--` is what tells them apart; skim past it and you'll run the wrong one.

CHAPTER 4

TALKING TO XOPS IN PLAIN ENGLISH

xops ask is a deterministic phrase resolver, not a chatbot — here's exactly how it decides what to run.

Not an LLM, on purpose

`xops ask` lets you describe what you want and resolves that text to a normal command — xops automation flags, native package-manager commands, or git. There is no model dependency and no local AI setup. Resolution is deterministic: input is normalized and matched against phrase catalogs by `@x12i/ask-cli`, then turned into a structured plan that's validated before anything runs. Resolution order is xops catalog first, then npm, then git.

EXAMPLES

```
xops ask "install all packages"
xops ask "publish all @x12i packages and push"
xops ask "show outdated packages"
xops ask "show git status"
xops ask "pull latest changes"
```

How it works, step by step

When you run `xops ask "<text>"`, the CLI normalizes the text (lowercase, trim, collapse spaces, expand shorthands like "everything" → "all packages"), matches it against a phrase catalog, extracts known variables (scope, package name, git provider), builds a structured argv list — it never executes the raw English string — validates the plan with the same rules as a normal command, shows you the resolved command and its risks, asks for approval if it's dangerous, and only then executes.

TECHNICAL STUFF

Phrase catalogs ship as JSON at `src/catalog/xops-ask.catalog.json`, `npm-ask.catalog.json`, and `git-ask.catalog.json` (built to `dist/catalog/`). Matching and normalization come from `@x12i/ask-cli`; xops owns validation, approval, and execution.

Approval for dangerous operations

Safe operations (install, build, test, status, diff) run immediately once resolved. Dangerous ones — publish, push, pull, commit, first-publish, git creation — show a confirmation prompt first:

WHAT A DANGEROUS-OPERATION PROMPT LOOKS LIKE

```
I understood this as:

  xops --filter "@x12i/*" --build --test --publish --push --report

This will:
  - Select packages matching the requested scope.
  - Run install.
  - Run build and test.
  - Publish packages in dependency-aware order.
  - Push git changes.

Risks:
  - This may publish new npm versions.
  - This may push commits to git remotes.

Execute? [y/N]
```

Only `y` or `yes` proceeds. Anything else cancels — there's no "press enter to continue" trap.

Code-agent-safe ask

`xops ask` never blocks on stdin when the caller is a code agent or CI. `--plan` resolves the request to a command, argv, risks, and explanation, and exits without running it; add `--json` for machine-readable output. `--agent`, the env var `XOPS_AGENT=1`, `CI=true`, or a non-TTY stdin all mean "never wait for an interactive answer" — in any of those contexts, a dangerous operation without `--yes` exits with code `4` (approval required) instead of hanging on a prompt nobody will ever answer.

RESOLVING INTENT WITHOUT RUNNING ANYTHING

```
xops ask "release the stack" --plan --json
xops ask "release the stack" --agent
xops ask "release the stack" --yes
```

> **REMEMBER**
>
> TTY behavior for humans is unchanged — `--agent` / `CI=true` /non-TTY detection only changes what happens when approval would otherwise be required.

When nothing matches

If no catalog entry matches your text, `xops ask` fails safely and suggests known examples instead of guessing:

UNMATCHED REQUEST

```
I could not match that request to a known xops, npm, or git command.

Try one of these:
  xops ask "install all packages"
  xops ask "run build"
  xops ask "show git status"
```

> **FIELD NOTE**
>
> Some phrases resolve straight into release/publish workflows — e.g. `xops ask "publish everything in the right order"` resolves to a full build-test-publish-report run. *xops Release Engineering & Playbooks for Dummies* covers what that run actually does under the hood.

CHAPTER 5

FLAGS YOU'LL ACTUALLY USE

The everyday flags, and how package selection works when a repo has more than one package.

The core lifecycle flags

FLAG	DESCRIPTION
`--build`	Run `npm run build` / `pnpm run build` after install.
`--test`	Run `npm test` / `pnpm test` after build.
`--dry-run`	Show planned execution without modifying anything.
`--report`	Suppress noisy native output, show a high-level summary only.
`--no-install`	Skip npm install steps — useful right after a local `npm install`.

TIP

Without `--report`, xops streams native npm/git output directly, exactly as if you'd run the commands yourself — reach for `--report` when you want a clean summary instead of a wall of npm log lines.

Selecting packages in a multi-package repo

`--package <name>` selects by exact `package.json` name and is repeatable. `--filter <glob>` matches package names using glob syntax — not folder paths. Multiple values of either flag are OR-based, and the two flags are OR-based together.

FILTER EXAMPLES

```
--filter "@x12i/*"        matches every package under the @x12i scope
--filter "@x12i/ai-*"     matches only @x12i packages starting with ai-
--filter "*gateway*"      matches any package name containing gateway
--filter "@x12i/*" --filter "@exellix/*"   matches either scope
```

WATCH OUT

If no package matches a given `--filter` glob, xops fails with a clear error rather than silently doing nothing. Package-name and folder-path matching are never mixed in the same flag.

Discovery scope: `--all`

By default xops scopes discovery to avoid crawling unrelated sibling repos: inside a git repo, discovery is scoped to that repo's root; inside an npm workspace, to the workspace root; anywhere else, to the current folder only (nested `.git` directories are skipped). Pass `--all` to scan every `package.json` recursively, including nested git repos — useful from a shared parent folder holding multiple independent projects.

SCOPE IN PRACTICE

```
cd ~/projects/x12i
xops install              # nothing, unless a package lives directly here
xops install --all        # every package under x12i
```

Fix flags

`--fix` refreshes in-house `@x12i/*` and `@exellix/*` packages (and workspace locals) to the npm-latest range. Bumping public/third-party dependencies (vite, react, tailwindcss, and similar) needs the separate `--fix-public` flag, or naming the package explicitly (`xops install --fix vite`). `--no-fix` skips automatic fix entirely on install and the release flows.

REMEMBER

This split exists on purpose: refreshing your own in-house packages is low-risk and happens by default on release flows, but bumping public toolchain dependencies is a separate, deliberate decision.

FIELD NOTE

`--publish`, `--push`, `--publish-flow`, `--full-flow`, `--stack`, and the whole package-graph/`xops.json` flag set are covered in depth in *xops Release Engineering & Playbooks for Dummies* — this book only needed the flags you reach for on an ordinary day that doesn't end in a publish.

APPENDIX

QUICK REFERENCE

Every reserved command, every everyday flag, and the exit codes, on one page.

Reserved commands (always xops)

COMMAND	NOTES
`ask`	Plain-English resolver, deterministic
`doctor`	Node/npm/git/xops setup check
`list` / `ls`	Package discovery and dependency relationships
`validate`	Read-only integrity + registry check
`troubleshooting`	Install/PATH/cache guide
`help` / `-h` / `--help`	Usage, flags, examples

Everyday flags

FLAG	EFFECT
`--build`	Run build after install
`--test`	Run tests after build
`--report`	Summary output instead of raw npm/git streaming
`--dry-run`	Plan only, no changes
`--all`	Scan every package recursively, including nested repos
`--package <name>`	Select by exact package name (repeatable)
`--filter <glob>`	Select by name glob (repeatable, OR-based)
`--fix` / `--fix-public` / `--no-fix`	In-house vs. public dependency refresh
`--plan` / `--json` / `--agent` / `--yes`	`xops ask`-only: resolve, machine output, non-interactive, approve

Exit codes worth recognizing

CODE	MEANING
`0`	Success
`1`	Failed — package, registry, validation, or command error
`2`	Partial success — warnings only
`3`	Environment/permissions — not a package bug
`4`	`xops ask`: approval required, none given in a non-interactive context

The one-sentence summary

xops is one command surface that detects what your repo already has, routes ordinary work straight to npm, pnpm, or git, and resolves plain-English requests into a validated, approval-gated plan instead of guessing.