CASE FILE · OPS-02

A Plain-English Field Guide

XOPS RELEASE ENGINEERING & PLAYBOOKS

FOR DUMMIES

FIX · BUILD · TEST · PUBLISH · PUSH

"npm publish can't be undone — so the pack-check that blocks a secret from shipping runs every single time, even when your ignore files already look right."

A friendly, technical walk-through of how **xops** releases packages: the fix → install → build → test → publish → push lifecycle, the optional package graph, the safety net that blocks secrets from ever reaching a tarball, and the operational playbooks that turn one-off scripts into repeatable, risk-classified commands.

`xops.json` `publish safety` `playbooks` `exit codes`

FILED: JULY 2026

STATUS: FOR INTERNAL USE

Case File Directory

WHAT'S INSIDE

READ ME FIRST

CHAPTER 0

HOW TO USE THIS GUIDE

What this book covers, what it assumes you already know, and where that other stuff lives.

xops is the operating layer for Node.js repositories — one command surface for the work around npm, git, tests, and release. This book is the deep dive into the half of xops that matters when you're actually shipping something: the release lifecycle, the optional package graph, the safety net around publishing, and the operational playbooks that turn ad-hoc release scripts into repeatable, risk-classified commands.

This book assumes you already know the basics of the xops command surface, `xops ask`, and everyday flags like `--build`/`--test`/`--filter` — that's the companion book, *xops for Dummies: the Daily CLI*. This book does not re-explain those; it goes straight into what happens when you run `xops release` or `xops --full-flow` for real.

SOURCE MATERIAL	WHAT IT GIVES YOU
The Release Lifecycle	Operation order, dependency-aware sequencing, and what happens to git.
The Optional Package Graph	What `xops.json` is for, and why most repos don't need one.
Publish Safety	The pack-check that blocks secrets, exit codes, and failure behavior.
Publish Scripts	Tiered monorepo release scripts and the ordered publish runbook.
Operational Playbooks	Risk-classified, parameterized templates for repeated repo actions.

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 daily-CLI companion book.

Who this is for

You're the developer who runs the release, the person who set up a monorepo's publish scripts, or the code agent operator who needs `xops release` to fail loudly and safely instead of silently shipping a half-built package. You don't need to memorize every JSON key in `xops.json`; you do need to know the order things happen in, what the pack-check actually blocks, and which exit code means "your problem" versus "the environment's problem."

CHAPTER 1

THE RELEASE LIFECYCLE

Fix, install, build, test, publish, push — in that order, and why the order is not negotiable.

Operation order

For each package, xops runs: **[fix] → install → build → test → publish → push**. Only the requested operations run. `fix` runs by default only for `--publish-flow` and `--full-flow`, or when `--fix` is passed explicitly; `--no-fix` skips it.

THE TWO FULL-FLOW SHORTCUTS

```
xops --publish-flow   # fix, install, publish, and push, with a final report
xops --full-flow      # fix, install, build, test, publish, and push, with a final report
```

REMEMBER

The default fix scope only refreshes in-house `@x12i/*` and `@exellix/*` packages. Public toolchain bumps (vite, react, and similar) need the separate `--fix-public` flag — a release flow will not silently bump your framework version.

Dependency-aware execution order

Local package relationships are mapped automatically. If `@x12i/api` depends on `@x12i/core`, then `@x12i/core` is installed, built, tested, and published first. Dependency cycles are detected and reported before any command runs.

Releasing from a subfolder gets an extra safety check: when you run lifecycle commands from a package's own folder, xops walks to the git root, finds sibling packages, and compares their local version to what you declared. If a sibling is ahead of your declared range or not yet published:

COMMAND	BEHAVIOR
`install` / `build` / `test`	Proactive alert — fast check, does not block
`validate`	Reports `sibling-release-first` as an error
`release` / `publish`	Asks permission to release the upstream package first (default yes)

Git behavior: monorepo vs. multi-repo

In a **monorepo**, all npm lifecycle steps run across every package first, then git add/commit/push runs once at the repository root. In **multi-repo** mode, each package/repo completes its full lifecycle — including its own push — before xops moves to the next repo. If there are no git changes to commit, xops reports that and continues rather than failing.

TIP

With `--push`, if `git push` fails because the remote has commits you don't have locally, xops shows the missing commits and offers to run `git pull --rebase origin <branch>` automatically before retrying — pass `--yes` to approve that recovery non-interactively.

FIELD NOTE

The natural-language resolver behind `xops ask "publish everything in the right order"` and the approval prompt it shows before a dangerous operation are covered in *xops for Dummies: the Daily CLI*, Chapter 4.

CHAPTER 2

THE OPTIONAL PACKAGE GRAPH

xops.json documents publish order and release profiles — and nothing changes until you opt in.

Discovery doesn't need it

xops finds packages from `package.json` files without any extra configuration. By default, discovery is scoped: inside a git repo, to that repo's root; inside an npm workspace, to the workspace root; anywhere else, to the current folder only. Pass `--all` to scan every `package.json` recursively, including nested git repos.

What xops.json is for

Nothing changes unless you opt in. `xops.json` is a machine-readable package graph — not a replacement for `package.json`, but a complement documenting discovery layout, publish sequences, local dependency edges, known cycles, and release profiles. Add it only when a repo benefits from explicit publish order, documented dev/test cycles, or tiered releases.

GENERATING A STARTER GRAPH

```
xops map                       # writes xops.json (backs up existing to xops.json.bak)
xops map-to ./custom-graph.json # writes to a custom file
xops map-out                   # prints JSON to stdout
```

WATCH OUT

If `xops.json` already exists, `xops map` renames it to `xops.json.bak` first — and an existing `.bak` gets overwritten. Don't run `xops map` twice in a row expecting to keep two backups.

Once `xops.json` exists at the repo root, xops picks it up for `list`, `validate`, and lifecycle commands: publish order comes from its `sequences`, relationships still come from each `package.json` by default (`operations.graphSource` defaults to `package-json`), private roots are excluded via `operations.privatePackages`, and documented dev cycles are respected when `operations.allowDocumentedDevDependencyCycles` is set.

REMEMBER

`xops list` and `validate` warn when `xops.json` dependency metadata is out of sync with `package.json` — re-run `xops map` from the repo root to refresh it. Without `xops.json` at all, behavior is completely unchanged: xops just infers relationships directly from each `package.json`.

Fix configuration lives here too

Once you have `xops.json`, you can pin toolchain versions or disable automatic fix for lifecycle flows:

DISABLING AUTOFIX AND EXCLUDING PACKAGES FROM IT

```
{
  "operations": {
    "autoFix": false,
    "fix": {
      "dependencies": true,
      "devDependencies": false,
      "peerDependencies": false,
      "exclude": ["typescript", "@types/node"]
    }
  }
}
```

CHAPTER 3

PUBLISH SAFETY

The pack-check that runs no matter what your ignore files say, and the exit codes that tell you whose problem it is.

What happens before every publish

Before publishing any package, xops runs a fixed sequence: validate that `.gitignore` covers `.env*` and `.npmrc` when `--push` is also requested; validate that `.npmignore` covers the same (creating one if it's missing); run `npm pack --dry-run --json` and fail if any `.env` or `.npmrc` file would be included; bump the minor version unless `--no-version-bump` is set; run a second pack-check on the final package state; publish with `npm publish --access public`; then refresh dependent packages to `@latest` for anything published earlier in the same run.

REMEMBER

The pack-check is the real safety net — it runs even when the ignore files already look correct, because `package.json`'s `files` field can override them and quietly include something it shouldn't.

What the CLI enforces regardless of configuration

- `.gitignore` must exclude `.env*` and `.npmrc`
- `.npmignore` must exclude the same (auto-created if missing)
- `npm pack --dry-run --json` always runs before publish — publishing is blocked if a `.env` or `.npmrc` would ship
- Tokens and token-like values are redacted from all output and reports

WATCH OUT

`--fast` skips some slower checks (tarball inspection details, gitignore/npmignore enforcement niceties, post-publish verify, install-health) — but the pack-check itself still runs on publish and cannot be disabled. There is no flag that lets a secret ship.

Exit codes: whose problem is it

CODE	MEANING
`0`	Success
`1`	Failed — package, registry, validation, or command error
`2`	Partial success — warnings only, e.g. publish succeeded but git push failed
`3`	Environment/permissions — root-owned cache, global install dir, or `node_modules`; not a package bug
`4`	Approval required — `xops ask` matched a command needing confirmation but none was given non-interactively

Failure behavior

xops stops on the first critical failure. A package is never published if install, build, or test failed for it, and in a dependency chain, if one package fails, its dependents are never processed. On multi-package `--all install` runs, failures include an install summary — which packages succeeded, failed, were skipped, or were never attempted — so a partial run is obvious rather than silently incomplete.

WATCH OUT

`npm publish` cannot be rolled back. If publish succeeds but the git push fails and rebase recovery was declined or unavailable, the run reports partial success (exit code `2`) — you still need to manually `git pull --rebase origin main && git push origin main`, or retry with `--push --yes`. The npm side of that release already happened.

FIELD NOTE

The npmrc-resolution order (package folder → invocation root → git root → system config) and the isolated npm cache at `~/.cache/xops/npm` are shared with everyday install — see *xops for Dummies: the Daily CLI*, Chapter 2, for the cache fallback and troubleshooting story.

CHAPTER 4

PUBLISH SCRIPTS AND THE MONOREPO WORKFLOW

Generate tiered scripts once, then run them by name instead of re-typing the flags every release.

Tiered scripts

`xops scripts init` writes `scripts/publish-*.sh`, wires them into root `package.json`, and merges into `xops.json` if it's present:

SCRIPT NAME	SHELL FILE	UNDERLYING COMMAND
`preflight`	`scripts/publish-preflight.sh`	`xops validate`
`core`	`scripts/publish-core.sh`	`xops --build --test --publish --report`
`all`	`scripts/publish-all.sh`	`xops release --report`

TYPICAL MONOREPO FLOW

```
xops map                    # optional: write xops.json graph
xops scripts init           # create publish scripts
xops --all install          # bootstrap all packages
xops scripts run preflight  # validate
xops scripts run all        # release when ready
```

The ordered runbook, for agents or manual publishing

`xops scripts init runbook` discovers packages, computes dependency order, and writes both a step-by-step markdown runbook and an executable version of the same flow:

FILE	PURPOSE
`scripts/publish-runbook.md`	Step-by-step guide: npm preflight, build + publish per package, verify loop
`scripts/publish-ordered.sh`	Executable version of the same flow

SCOPING THE RUNBOOK

```
xops scripts init runbook --filter "@x12i/memorix-*"   # publish subset; verify all packages
xops scripts init runbook --dry-run                    # preview markdown only
```

TIP

Script definitions are read from `xops.json`'s `scripts` key (populated by `xops map`) or inferred from root `package.json`'s `publish:*` entries — and they always run from the git repo root with that root's `.npmrc` applied automatically. No wrapper script needed for cache or npmrc.

REMEMBER

Forward extra args to any script with `--`: `xops scripts run all -- --dry-run` previews the full release playbook without publishing anything.

CHAPTER 5

OPERATIONAL PLAYBOOKS

Parameterized, risk-classified templates for repository actions you do more than once.

The built-in catalog

xops has first-class operational playbooks for repeated repository actions that need parameters, planning, risk metadata, approval behavior, and run reports. The built-in catalog currently includes `kill-port`, `dev-stack`, `publish-chain`, and `publish-all`.

WORKING WITH PLAYBOOKS

```
xops playbooks list
xops playbooks status --json
xops playbooks init kill-port
xops playbooks plan kill-port --port 3000 --json
xops playbooks run kill-port --port 3000 --dry-run
xops playbooks run dev-stack --stack jobs --port 3000
xops playbooks run publish-chain --package @scope/app
xops playbooks run publish-all --group core --bump minor
xops playbooks explain dev-stack
xops playbooks validate
xops playbooks doctor
```

Shorter aliases exist for the common case: `xops play kill-port --port 3000` and `xops run-playbook kill-port --port 3000` both work the same way.

Where playbooks live

Local playbooks live in `.xops/playbooks/`. `xops playbooks init <name>` writes a `.mjs` definition using `defineXopsPlaybook(...)` and updates `.xops/playbooks/playbooks.json`; `xops playbooks adopt scripts/foo.mjs --name foo --yes` copies an existing ad-hoc script into the playbook catalog instead of making you rewrite it from scratch. Run artifacts land in `.xops/playbooks/runs/<run-id>.json` and `.xops/playbooks/runs/<run-id>.md`.

Plans are explicit about risk

A playbook plan includes resolved inputs, packages, env files, ports, commands, files, required tools, risks, approval requirements, expected artifacts, and steps. Risk classes are explicit, not implied: `read-only`, `local-write`, `local-process`, `network-local`, `network-external`, `package-install`, `package-publish`, `git-mutation`, `system-mutation`, and `risk-sensitive`.

> **REMEMBER**
>
> A `package-publish` or `git-mutation` risk class means the playbook behaves like any other dangerous xops operation — it's plan-first, and it asks for approval before doing anything irreversible.

Reaching playbooks from the SDK and MCP

Playbook SDK functions are exported from `@x12i/xops`: `listPlaybooks`, `detectPlaybooks`, `loadPlaybook`, `planPlaybook`, `runPlaybook`, `validatePlaybook`, `generatePlaybook`, `explainPlaybook`, and `adoptPlaybook`. Authoring helpers ship separately from `@x12i/xops/playbooks`. MCP exposes the same surface as `xops.playbooks.list`,

`.load`, `.plan`, `.run`, `.validate`, `.generate`, `.explain`, and `.adopt`.

FIELD NOTE

The three official interfaces — CLI, SDK, MCP — and what each one is for are introduced in *xops for Dummies: the Daily CLI*, Chapter 1. This chapter only covers the playbook-specific corner of that surface.

APPENDIX

QUICK REFERENCE

Operation order, exit codes, the publish safety checklist, and the built-in playbooks, on one page.

Operation order

`[fix] → install → build → test → publish → push`

Exit codes

CODE	MEANING
`0`	Success
`1`	Failed
`2`	Partial success (warnings only)
`3`	Environment/permissions
`4`	Approval required (`xops ask`, non-interactive)

Publish safety checklist (runs every time, no opt-out)

CHECK	OUTCOME IF IT FAILS
`.gitignore` covers `.env*` / `.npmrc`	Validation error (with `--push`)
`.npmignore` covers `.env*` / `.npmrc`	Auto-created if missing
`npm pack --dry-run --json` (pre-bump and post-bump)	Publish blocked if a secret file would ship
Version already on npm registry (no `--no-version-bump`)	Package skipped unless `--force-republish`

Built-in playbooks

NAME	PURPOSE
`kill-port`	Free a local dev port
`dev-stack`	Start a named local dev stack
`publish-chain`	Publish one package and its local dependents
`publish-all`	Publish a group of packages together

The one-sentence summary

Every xops release runs the same fixed order, every publish gets pack-checked for secrets whether or not you asked for it, and repeated repo actions belong in a risk-classified playbook instead of a script nobody remembers the flags for.