React for Enterprise

Timeless Architecture for Enterprise Apps

By Przemyslaw Nowak,
with the Nx Core Team

theNowak.dev

Introduction

Who is this book for?

It’s for engineers working in large organizations - where you're not just writing code, you’re navigating architecture decisions made by dozens of teams over the years. Where every "small" change comes hard. Where the same problems - inconsistent modules, duplicate business logic, conflicting code styles - show up again and again, just with different logos.

It's targeted especially for teams managing long-lived React products - apps that need to stay fast and maintainable as they grow. But it’s just as relevant for teams starting from scratch. Greenfield projects can benefit significantly by starting with the right foundations. The payoff may not be immediate, but in six months, when most teams are reworking architecture or duplicating efforts, you'll have a consistent stack ready to move fast without friction.

It’s for tech leads, principal engineers, and architects who are actively shaping how React is used across their organization. You’re likely building internal frameworks, enforcing standards, managing migrations, or aligning multiple teams around a consistent way of building apps - all while trying not to slow people down. You’ve probably already introduced tools like Nx, or you’re seriously considering it.

But this book is also for individual developers - even if you're not “the architect” today. If you’re trying to understand how your company's architecture choices shape your daily work, or why certain patterns keep emerging in large systems, this book will help. You’ll learn how to think about React beyond components and hooks - how to approach it as a scalable, structured ecosystem that can evolve over years.

Whether you’re maintaining a legacy mess, cleaning up after "move fast and break things," or laying the foundation for a new product - this book gives you a principled, tool-supported approach to React in the enterprise.

Why React Needs Opinionated Architecture

React gives you a component model and a hook system. That’s it.

It doesn’t tell you how to structure your application, where to put your business logic, how to handle server state, or when a component should be shared. That’s by design - React was built to be flexible. But flexibility without guardrails leads to chaos at scale.

In small teams, that chaos can be managed. Everyone knows the quirks. You copy patterns from each other. But as teams grow, and code lives longer, you start to feel the slowdown - duplicated logic, diverging patterns, and growing inconsistency. Two similar features are built differently. Data is fetched in three different ways. A new hire has to learn five styles of state management just to fix a bug.

React doesn’t fall apart in large codebases because it’s bad - it falls apart because it lacks structure. And that structure has to come from you.

An opinionated architecture is how you align teams, reduce cognitive load, and avoid reinventing the wheel every quarter. It gives developers a clear direction for how to work and make decisions in the codebase. And it sets you up for consistency - which is the only way to scale continuously.

That doesn’t mean you need to build a React monolith. It means you need conventions. Module structure. Clear rules for data flow and code reuse. Enforced layering. Standard generators. Predictable names. Shared mental models.

Angular gives you most of it out of the box. React makes you build it - and that’s not a disadvantage. That’s actually React’s strength.

With React, you can start faster, prototype quicker, and evolve naturally. It fits how modern teams work. The ecosystem is more vibrant, AI tooling is better aligned with React patterns, and there are simply more developers in the market familiar with it. If you're using V0, Bolt, or any AI-assisted dev tools, React is what you'll get. And with the right architecture, you can go from a throwaway prototype to a scalable product with minimal effort and without needing a full rebuild.

That’s the beauty of React. You can start with a single component and get something on the screen. Then, when you're ready, you refactor it into the structure you've defined. With the right conventions and a tool-assisted architecture - like the one in this book - even AI can do that transformation for you.

So yes, Angular gives you batteries-included. But React gives you leverage. And this book gives you the architecture to turn that leverage into long-term velocity.

Benefits for Teams, Developers, and Orgs

This architecture isn't about esthetic - it's about leverage at every level.

For teams

- You spend less time debating how to structure code and more time delivering value.
- Features can be built and owned independently, but still feel part of a single, coherent system.
- You onboard once. If a developer joins one project, they're productive across all of them - because the structure stays consistent everywhere.
- Migrations are smoother. Standards are predictable. Tooling and codemods actually work.

For developers

- You work in a codebase that's structured and familiar, so you can focus on solving problems instead of fighting architecture.
- You don't waste time guessing how something was built - you already know.
- AI tools become practical assistants. They understand your patterns and generate code that fits.
- You can switch teams or products with minimal friction.

For orgs

- Consistency scales. You don't need a central team of code police.
- Quality improves not because people are perfect, but because the system guides better decisions.
- AI automation becomes possible - structure makes the codebase machine-readable.
- Teams stay productive as the organization grows.

Principles this book is built on

This book is grounded in principles that make React codebases scalable, maintainable, and ready for automation. These are not abstract ideals - they come from the real-world work of building framework like Brainly Gene inside large organization using Nx.

1. Define and document clear rules

Set standards for naming, code structure, file layout, and common patterns - and make them easy to find. This isn't just for humans. The more predictable your code is, the better AI tools and new team members can understand and work with it.

2. Enforce rules automatically

Use Nx and tools like ESLint, Prettier, and Stylelint - tightly configured. The goal is to remove debates and inconsistency at the root. If something matters, it should be enforced before it hits the main branch.

3. Decouple logic with layered architecture

Separate UI from business logic and data access. Build clear boundaries using Nx tagging and constraints - not just mental models. Keep core logic isolated from frameworks like Next.js. The less your code depends on a specific stack, the easier it is to evolve.

4. Generate structure, don't repeat it

If something is repeated, automate it. Nx generators should scaffold every standard - modules, services, components - so every new feature starts clean and consistent. No one should be reinventing boilerplate.

5. Use CI to enforce consistency at every level

Your CI pipeline should reflect how your code runs - and how it's reviewed. With Nx, you get a consistent and fast build process that runs the same way locally and in CI. That means no surprises between environments.

Start by using Nx to run affected builds, tests, and linting. Add custom checks to validate code structure, file placement, naming conventions, and deeper rules using AST-level analysis. These checks should reflect not just layout, but also typical standards, boundaries, and architectural decisions.

GitHub Actions can help enforce these and additional rules (using AST to check if React specific code style has been kept) automatically during pull requests.

6. Use TypeScript for clarity and predictability

Type safety isn’t just about catching bugs. It helps humans and AI alike understand your system. Clearly defined types make your codebase easier to navigate, refactor, and extend.

How to Use This Book

You don’t need to drop everything and rebuild your entire app tomorrow.

This book is structured for gradual adoption. You can introduce one rule, one generator, one boundary at a time - and still get value. Some teams will follow this system fully. Others will borrow pieces that solve specific pain points.

You might start with code structure and naming rules. Later, you can introduce CI checks or Nx generators. Or you might already have some standards in place - and use this book to formalize and automate them across projects.

It’s also perfect for greenfield projects. You don’t need to learn by failing. Start with a clear architecture. Build fast - without cutting corners that come back to affect you in six months.

If you're not the architect yet, this book still helps. Use it to guide reviews, influence direction, or advocate for improvements that actually scale.

Throughout the book, you’ll also see references to **Brainly Gene** - the open source framework I've built together with FE Infra team at Brainly and battle-tested in production. Gene follows the exact principles this book is built on. You won’t find Gene’s details in every chapter, but it serves as a real-world example of what’s possible. The appendix includes a brief overview of its structure.

You don’t need to recreate Gene (as it is open sourced). But you can build something similar - with your team, in your context - using the same foundation.

Part I: Foundations

1. Nx and the Role of Tooling

1.1 Why Nx is the Best Companion for Enterprise React

If you're serious about scaling React in an enterprise, you'll quickly realize that React's minimalism is both a strength and a gap. It gives you freedom, but that freedom becomes chaos without constraints. Nx is how you bring intelligent structure, automation, and consistency to your codebase - without sacrificing flexibility.

This chapter lays the groundwork by showing why Nx is not just a nice-to-have, but a critical layer in any large-scale React architecture.

Why Tooling Matters in Enterprise

In a 5-person team, you can agree on patterns over lunch. In a 500-person org, you need systems - ones that are enforceable, observable, and repeatable. That's what Nx enables:

- **Code scaffolding**: `nx generate` gives you a standardized starting point - enforcing repository structure, naming conventions, and boilerplate setup that aligns with your standards. We'll explore how to define and extend these generators later.
- **Project graph**: `nx graph` lets you understand your project structure and dependencies via live graphs. This is essential when working with hundreds of libraries and apps - you can spot architectural violations, understand critical paths, debug circular dependencies, and onboard new developers faster by showing them how the system fits together visually.
- **Enforced boundaries**: Want to block UI libraries from importing data-access code? Nx can stop that at lint time - using static code analysis and tag-based constraints. This is more than just a lint rule: it enforces architectural boundaries and protects your workspace from becoming an unmanageable mess of circular dependencies and spaghetti code. With well-defined, cohesive units and declarative rules, even large teams can scale safely without accidental cross-wiring of responsibilities.
- **Build intelligence**: In a growing monorepo, why rebuild 300 projects when you only changed two? Nx scopes builds to affected apps/libs only - thanks to a powerful integrated task runner that understands project dependencies. This means faster CI pipelines, shorter feedback loops, and fewer wasted cycles. It also brings consistency: the same logic used for build orchestration in CI can be used locally by any engineer, without writing custom scripts or glue code. Nx gives you precise, dependency-aware execution - parallelized, cacheable, and optimized out of the box.

- **Maintenance over time:** Nx provides a powerful migration system that not only automates dependency and config upgrades, but also applies codemods to update your actual code when APIs change. In an integrated monorepo with a single version policy, this kind of automation can save weeks of manual work - eliminating the need for large-scale coordinated refactors and significantly reducing upgrade risk and effort.

In short, Nx takes enterprise headaches - unclear ownership, inconsistent code, slow CI, and risky refactors - and solves them with tooling, not policies.

Nx vs. DIY (Do it yourself): Why It Wins

Yes, you can technically roll your own monorepo setup. But here's what you'd be reinventing:

Capability	DIY Cost	Nx Support
Project Graph Visualizations	High	Built-in `nx graph`
Affected Builds & Tests	Very High	`nx affected:*`
Custom Code Generators	Complex	First-class `workspace generators`
Dependency Constraints	Fragile	Enforced via ESLint rules
Single Version Policy	Manual & brittle	Standard in Nx. **You can have also multiple version policy and Nx supports it

Nx is framework-agnostic and works beautifully with React. What makes it shine in enterprise is how **opinionated the tooling is** while letting you define the **actual architecture**.

Monorepo Strategy Done Right

A well-executed monorepo isn't just about placing all your code in one place - it's about creating an environment where **shared code, atomic changes, and developer mobility** are not only possible but frictionless. Nx gives you the structure and automation needed to manage complexity as your codebase and teams scale.

- **Shared code via libraries**, with clear boundaries by tag types (`component` , `service` , `module` , `utility`) and scopes (`domain:<X>` , `platform:<Y>`). These tags are fully customizable to reflect your architecture.
- **Tag-based dependency rules** enforce architectural boundaries, ensuring domain-specific modules don't depend on each other accidentally. This helps you build a clean, maintainable dependency graph and avoid issues like circular imports or spaghetti code.
- **Atomic changes across the stack** become easy - update a shared component or backend API and all affected consumers in a single commit.
- **Consistent development experience** - all projects use the same standards, tooling, commands, and workflows.

Together, this enables a modular, testable, and maintainable architecture. You can isolate domains, share platform layers, and enable developer mobility without relying on undocumented, team-specific know-how.

Do I Need One Large Monorepo in My Enterprise?

If you're working with a few React apps that don't require separation due to legal, organizational, or scaling reasons - **just go with a single monorepo**. It simplifies everything: CI pipelines, one set of tools, shared code, and consistent workflows out of the box.

Monorepos offer huge advantages, but that doesn't mean every line of code in your company must live in a single repo. **You can - and often should - have multiple monorepos**, especially when legal, organizational, or architectural boundaries demand it.

The key is to **avoid code collocation without coordination**. Nx enables monorepo-style development - including code generation, testing standards, and dependency rules - that scales across multiple repos as long as you share conventions.

Keep in mind: multiple monorepos introduce another layer of shared tooling and standards that you'll need to manage well - it takes more effort to do it right than managing a single repo. But in large enterprises, this complexity is often necessary and unavoidable. That's where the idea of **multiple harmonized monorepos** comes into play.

To preserve developer mobility and automation between repos:

- Use shared Nx plugins and generators
- Align on code ownership rules and tagging conventions
- Keep tooling consistent even across separate codebases

The technical constraints of managing a large monorepo are largely solved by Nx. The remaining challenge is organizational alignment. If your teams can agree on core principles, a unified monorepo (or multiple harmonized ones) becomes a productivity engine instead of a bottleneck.

Conclusion: Why Nx, Why Now

If you're adopting a monorepo, aiming for code reuse, enforcing standards, or building internal platforms - Nx is the multiplier your enterprise architecture needs.

For React enterprises, it fills the gap between flexible framework and structured system. It's not about replacing your setup - it's about making it consistent, observable, enforceable, and scalable.

And when your architecture is predictable, your automation - including AI tools - becomes 10x more effective.

1.2 Workspace Setup, Generators, Constraints, and CI Integration

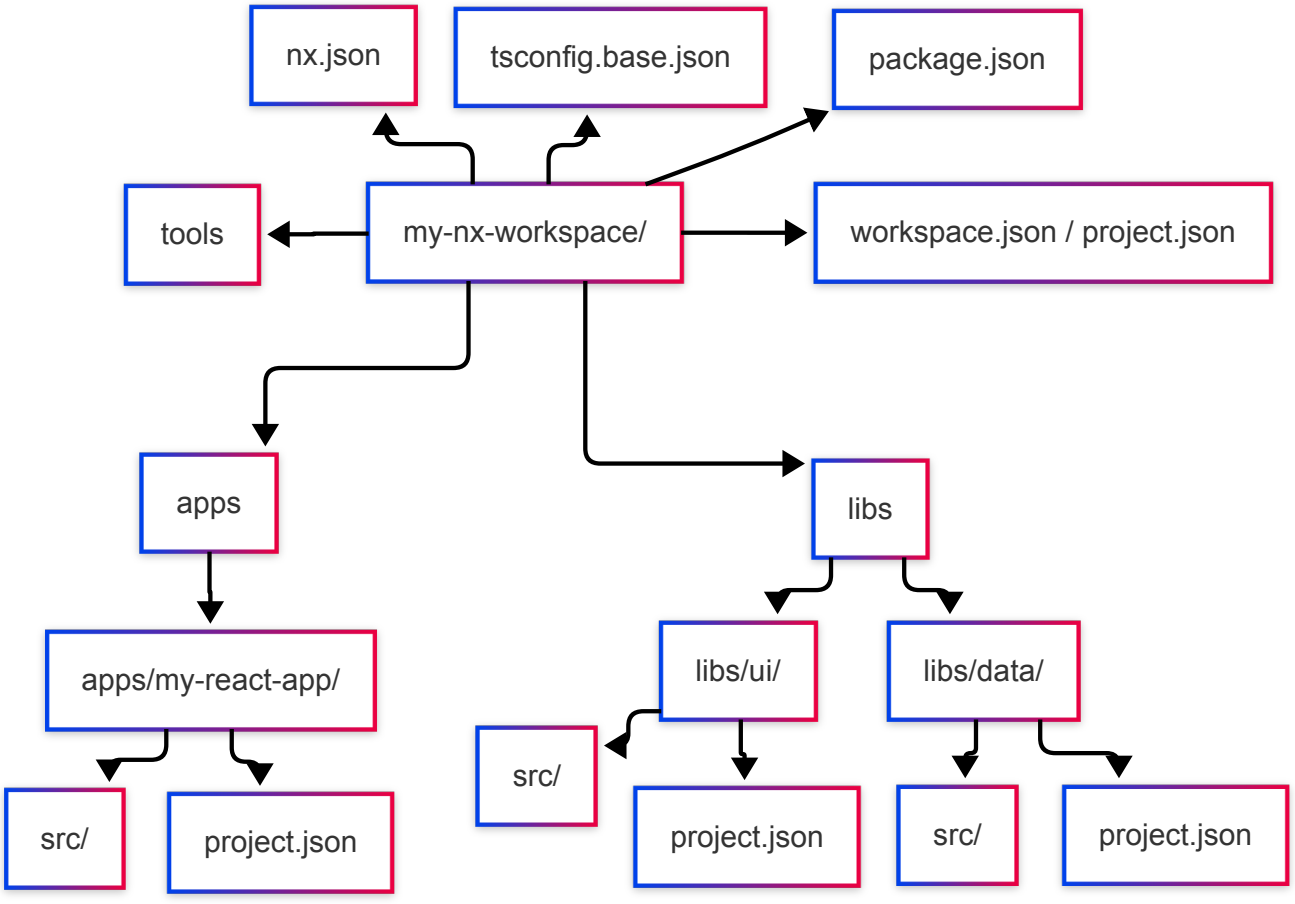
Starting a New Nx Workspace

To create a new Nx workspace from scratch, you can run:

```
npx create-nx-workspace@latest
```

You'll be prompted to select a preset. Choose the **React** preset (or **empty** if you want full control). Nx sets up the workspace with sensible defaults, a working React app, and some recommended libraries.

Official guide: https://nx.dev/getting-started/installation



Adding Nx to an Existing Project

Nx also works great as an incremental tool. You can convert an existing React app to Nx by running:

```
npx nx@latest init
```

Nx will detect your existing setup, convert it to a compatible structure, and scaffold its configuration. You can then start using generators, project graphs, dependency constraints, and affected builds without rewriting your app.

Official guide: https://nx.dev/getting-started/installation#installing-nx-into-an-existing-repository

This is the simplest way to get all the benefits of monorepo tooling - **without moving your code to a new repo**.

Designing the Nx Workspace for React at Scale

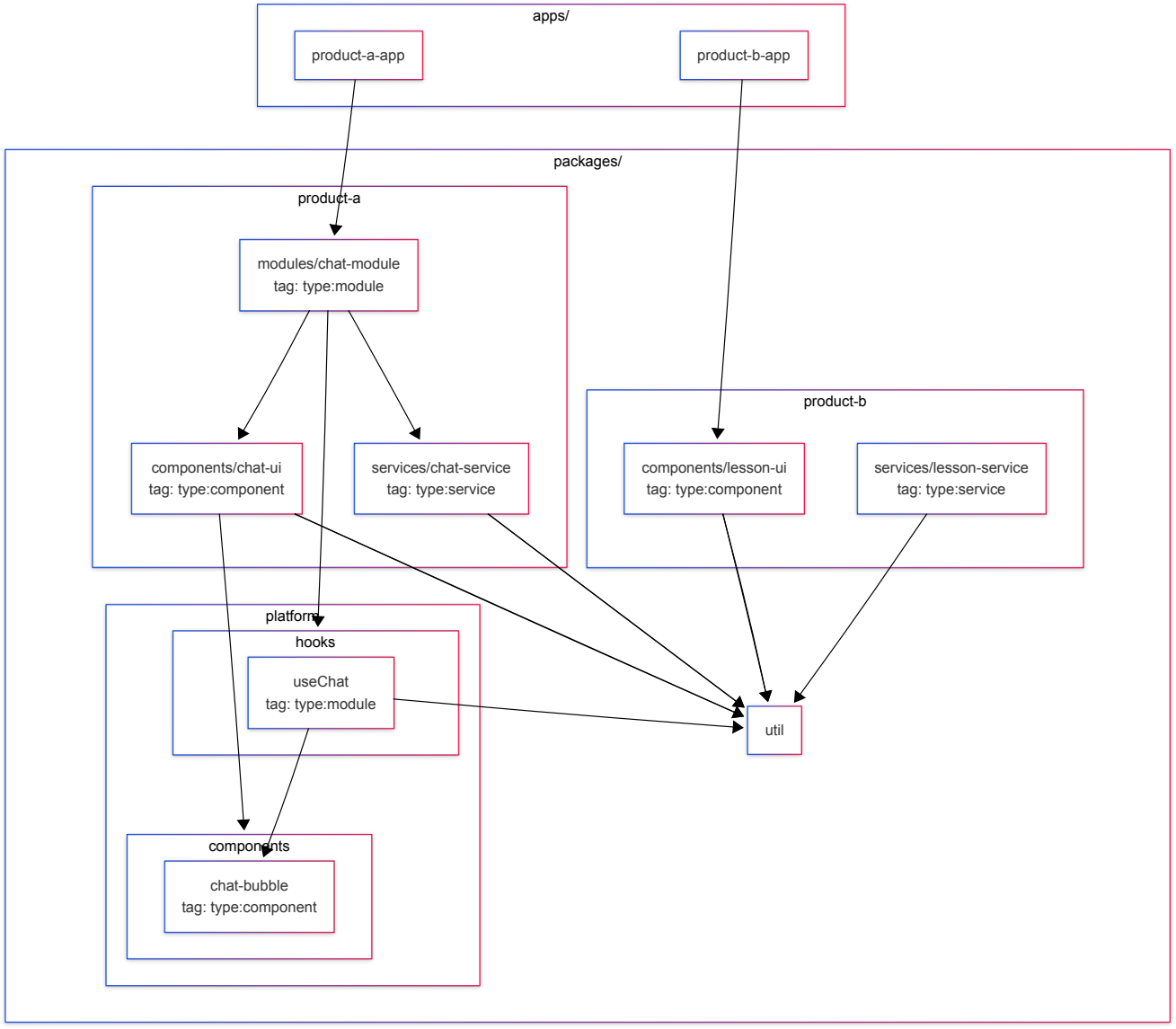
A great Nx setup doesn't start with apps - it starts with structure. In enterprise React projects, your folder layout defines the boundaries, visibility, and velocity of your engineering org.

At the core:

- `apps/` : Shells only. No logic inside. They wire modules and handle routing.
- `packages/` : The real architecture. Split by domain, platform, or shared utilities
- Each package has a `package.json` with tags like `type:module` , `type:component` , `type:service` , `platform:*` , `domain:*` - used later for constraints and dependency logic.

In most new Nx workspaces, `package.json` is enough and recommended. Only when you hit the limits of `package.json` -based scripts should you add a `project.json` for advanced configuration.

If you want to understand the mechanics in detail (including how TS project references optimize build performance), read this blog post https://nx.dev/blog/new-nx-experience-for-typescript-monorepos



Using Nx Generators to Scaffold with Confidence

Nx comes with a rich set of built-in generators that make it easy to scaffold apps, libraries, components, and configurations in a consistent way. These generators:

- Create the right folder structure and files automatically
- Add metadata like tags into `package.json` (or into `project.json` if you need advanced targets beyond what `package.json` can handle)
- Respect naming conventions and boundaries

Because this ebook promotes a **Single Version Policy**, relying on a unified root `package.json` for dependencies is critical. This ensures every package in the monorepo uses the same versions - a foundation for stability, codemods, and automation (covered in the next chapter).

You can start a new library or app with a single command and be confident that it follows Nx conventions - no manual wiring or setup required. This gives you immediate value out of the box, without needing to write your own tools.

Here are a few practical examples of using Nx generators in a real-world workspace layout:

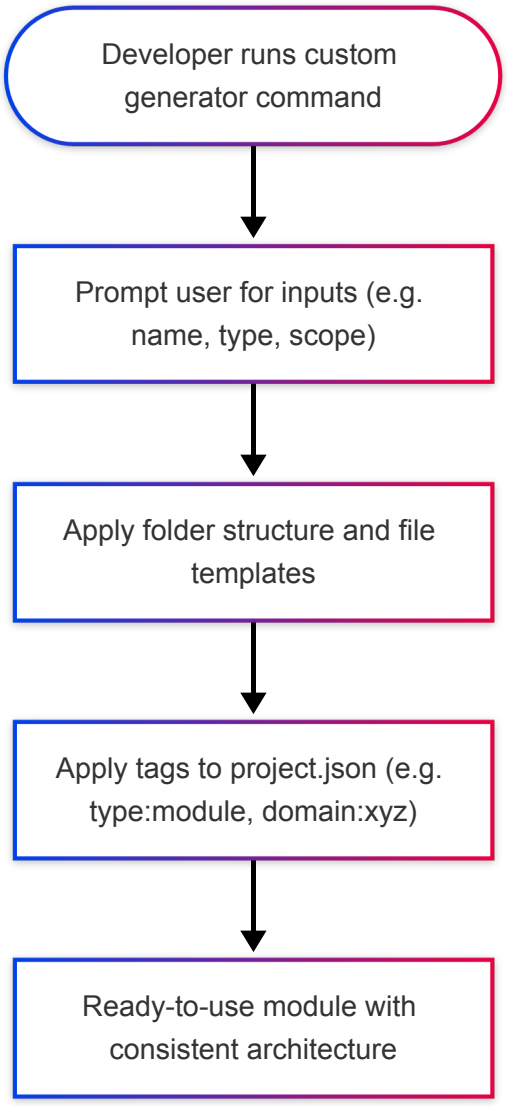
```
# Generate a presentational component in product-a domain
nx g @nx/react:component chat-ui \
  --project=product-a-components \
  --directory=packages/product-a/components/chat-ui \
  --export

# Generate a service library tagged for type:service
nx g @nx/js:library chat-service \
  --directory=packages/product-a/services/chat-service \
  --tags=type:service,domain:product-a

# Generate a feature module for chat experience
nx g @nx/js:library chat-module \
  --directory=packages/product-a/modules/chat-module \
  --tags=type:module,domain:product-a
```

These commands reflect a structure like:

- `packages/product-a/components/chat-ui` – reusable UI pieces
- `packages/product-a/services/chat-service` – data access or business logic
- `packages/product-a/modules/chat-module` – orchestrates services and components

These generators give you the ability to structure and scale confidently - right out of the box, with Nx.

We'll dive deeper into **custom generators** and how to extend them to match your opinionated architecture in the next chapters.



Enforcing Architecture with Tag Constraints

With great freedom comes great spaghetti - unless you draw boundaries. Nx allows you to enforce those boundaries with tag-based rules using static analysis via ESLint and the `@nx/enforce-module-boundaries` plugin.

Recommended rules include:

- UI libraries (`type:component`) must not import service libraries and modules (`type:service` , `type:module`)
- `platform:*` libraries can only depend on utilities (`type:util`)
- `domain:<X>` libraries must remain isolated from other `domain:<Y>` libraries

> ⚠️ These constraints are not enforced by default. You must define them explicitly in your ESLint configuration.

You can configure these rules in the `.eslintrc.base.json` (or project-specific `.eslintrc.json`) file under the `overrides` section:

`// eslintrc.json`

```
{
  "overrides": [
    {
      "files": ["*.ts", "*.tsx"],
      "rules": {
        "@nx/enforce-module-boundaries": [
          "error",
          {
            "enforceBuildableLibDependency": true,
            "allow": [],
            "depConstraints": [
              {
                "sourceTag": "type:component",
                "onlyDependOnLibsWithTags": ["type:component",
"type:util", "type:platform"]
              },
              {
                "sourceTag": "type:module",
                "onlyDependOnLibsWithTags": ["type:platform",
"type:component", "type:service", "type:util"]
              },
              {
                "sourceTag": "type:service",
                "onlyDependOnLibsWithTags": ["type:platform", "type:util"]
              },
              {
                "sourceTag": "platform:*",
                "onlyDependOnLibsWithTags": ["type:util"]
              },
              {
                "sourceTag": "domain:*",
                "onlyDependOnLibsWithTags": ["type:component",
"type:service", "type:module", "type:util", "platform:*"]
              }
            ]
          }
        ]
      }
    }
  ]
}
```

This setup ensures architectural discipline from day one. Teams can work independently and safely - with Nx acting as a protective layer against accidental cross-layer imports or architectural drift.

CI Integration: Only Build What You Break

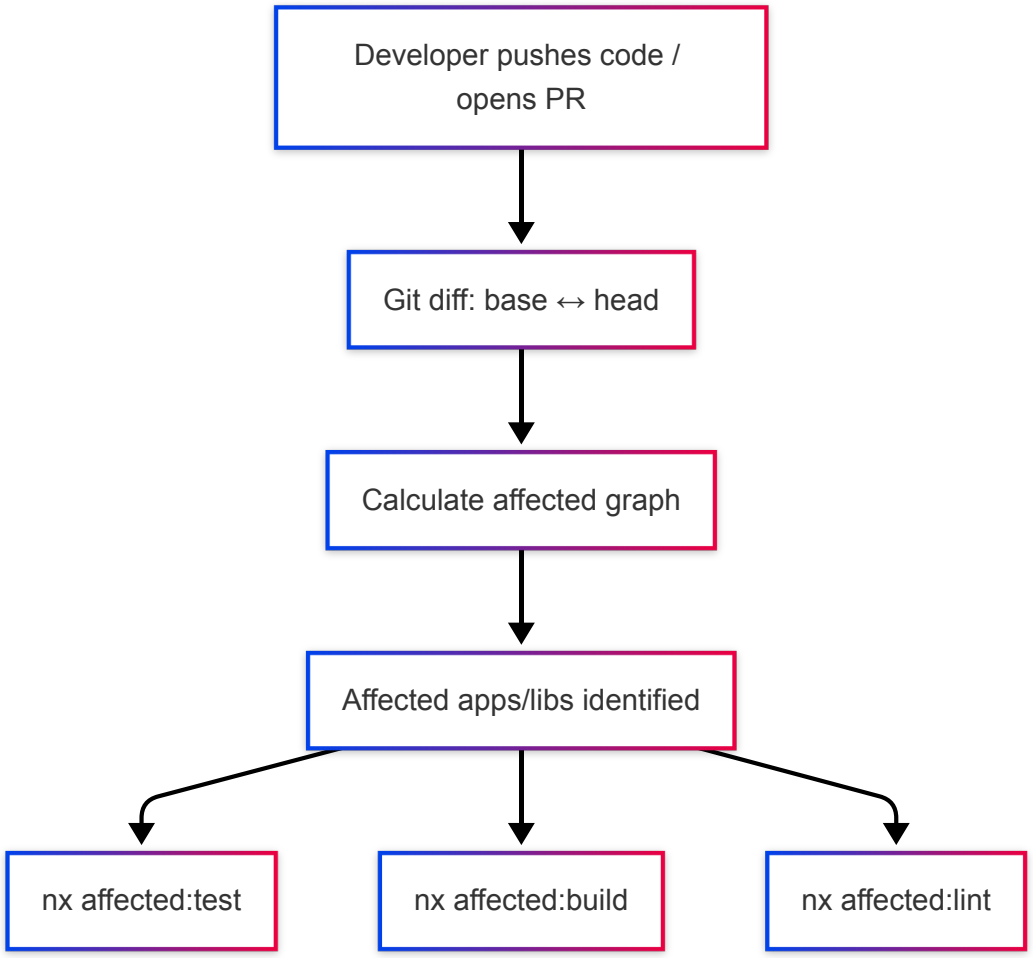
In most enterprises, CI becomes the bottleneck. Nx flips that by integrating `affected:*` logic into your pipelines:

```
nx affected:test --base=origin/main --head=HEAD
nx affected:build --base=origin/main --head=HEAD
```

This means:

- PRs only run tests for impacted projects
- Builds are scoped, parallelized, and cacheable
- CI behaves exactly like local dev

And with **Nx Cloud**, you get distributed caching and remote execution - so even your heaviest projects build in seconds on team machines.



Strangler Fig Pattern: Gradual Modernization with Nx

The Strangler Fig Pattern offers a powerful mental model for modernizing legacy projects. Instead of rewriting everything at once - which is risky, expensive, and often infeasible - you grow new structure alongside the old system.

This means you can start adding Nx to your project in a targeted way:

1. **Start small**: Introduce Nx with a simple feature or utility library. Let it coexist with legacy code.
2. **Expand with purpose**: Move more logic and features into Nx-managed libraries over time. Add tags and boundaries to enforce growing structure.
3. **Automate and migrate**: Replace legacy parts progressively. Use affected builds and shared constraints to ensure predictability.

Like the fig, Nx slowly overtakes the old tree - until the modern system becomes dominant, and the legacy structure fades away.

The way you set up your workspace defines what’s possible. Nx lets you turn architectural intent into enforceable systems:

- Custom generators define how new code is created
- Tag-based rules define what can talk to what
- CI integration makes it scale

Done right, you don’t just get a monorepo - you get an internal platform that accelerates everyone.

1.3 Single Version Policy: The Bedrock for Consistency and Automation

In monorepo architecture, especially at enterprise scale, consistency isn't a nice-to-have - it's a survival strategy. At the heart of this consistency lies one principle: **Single Version Policy** (SVP).

This means every dependency - React, React Query, ESLint, Tailwind, your internal libraries - is used in a single version across the whole workspace. One set of versions, consistently applied across the repo. In many setups this means a single root `package.json`, but with today's npm/pnpm workspaces-based you're free to structure things differently. What matters is alignment of versions, not the file layout.

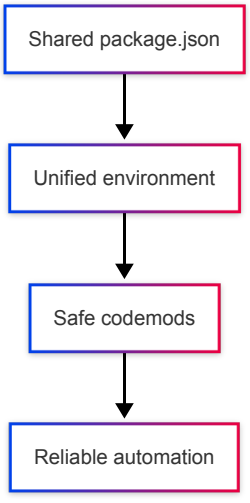
Why It Matters

Without a Single Version Policy:

- Teams drift in tooling and behavior
- Codemods become risky or impossible
- AI tooling (e.g. codegen, summarization) fails due to inconsistencies
- Upgrades are chaotic, with multiple breaking changes across apps

With a Single Version Policy:

- You upgrade once - everywhere
- Nx migrations and plugin updates work smoothly
- Linters and formatters behave the same for everyone
- Teams can collaborate across codebases instantly
- Your architecture becomes **AI-augmented** and **automation-friendly**



Nx + SVP: Enforcing It in Practice (and When to Break It)

Nx encourages SVP by design:

- Nx supports both a **single root** `package.json` and npm/pnpm workspaces-based setups. While teams are free to choose either model, keeping dependency versions aligned across packages preserves the benefits of the Single Version Policy.
- Every app and lib inherits from the same versioned base
- Nx plugins (e.g. for React, ESLint, Storybook) enforce version-aware migrations

Your dependency landscape becomes predictable and traceable. Upgrading React or Tailwind? You do it once - and Nx can automate it across the board.

Automation Becomes Possible

- Want to upgrade all `useQuery()` usage to the new syntax? Possible - and safe.
- Want to codemod all `styled()` calls to `twMerge`? Go for it.
- Want to migrate all Storybook 6 configs to v8? Nx plugins + SVP make it a non-event.

This is the foundation of **internal DX platforms**: consistency → predictability → automation.

Not Just About Dependencies

SVP applies to more than your `package.json`. It's about alignment:

- Code style (Prettier, ESLint)
- Test runners (Vitest, Jest)
- Lint configs and TypeScript settings

Your entire engineering culture becomes codified and shared. This is what allows codemods, AI agents, and automated migrations to thrive.

With npm/pnpm workspaces, SVP is no longer a hard requirement enforced by Nx. You can technically diverge versions between packages, and some organizations do. But the more you deviate, the more you lose the compounding benefits of consistency and automation.

Valid Exceptions to Single Version Policy

While SVP is the ideal, there are valid cases where **multiple versions** might temporarily make sense - especially in transitional or legacy-heavy environments.

1. Legacy Integration

Bringing legacy applications into a new Nx monorepo often means working with older dependency versions (e.g., React 17). In this case:

- Start your Nx setup using the same versions as the legacy system.
- Align tooling (e.g., ESLint, Jest) as much as possible.
- Use this shared foundation to enable gradual migration - without breaking what's already stable.

In some cases, you're not planning to migrate the legacy system at all - just to maintain it. Then it might make sense to:

- Keep the app isolated in the monorepo
- Freeze its package versions
- Avoid sharing code if dependencies aren't compatible

This approach gives operational convenience but little architectural benefit.

2. Piloting New Tooling or Frameworks

You may want to experiment with a new major version of React, Vite, or other tooling:

- A team can try React 19 in isolation
- You avoid friction from organization-wide upgrades

This makes sense **only as a short-term evaluation phase**. Once validated, standardize and migrate other apps to match - don't let divergence continue unchecked or become permanent.

> ⚠️ Piloting does **not** justify starting a new product line with newer dependencies unless the rest of the codebase is being updated. Divergence here leads to inconsistent architecture and harder automation.

Summary

Multiple versions are technically allowed by Nx, but should be used with caution:

- Treat them as **migration scaffolding**
- Keep boundaries hard and clear
- Avoid code sharing across mismatched versions
- Have a plan to converge toward SVP when feasible

2. Why React Needs a Framework

React gives you building blocks, not a blueprint. That’s its superpower - and its biggest problem at scale.

2.1 React Is Powerful, but Unopinionated

React is a library, not a framework. It handles the view layer - rendering, hooks, JSX. But everything else?

- Routing
- Data fetching
- State management
- File structure
- Code sharing
- Testing conventions
- Build pipelines

All of that is up to you.

This works beautifully in small apps or teams led by experienced engineers. But in large organizations, this freedom leads to:

- Divergent patterns between teams
- Inconsistent testing and tooling setups
- Fragmented approaches to data and state
- Reinventing architecture every time

The result? More overhead, more bugs, and less velocity.

The Cost of Scaling Without Structure

Without a common framework, teams solve the same problems again and again:

- "How do we structure features?"
- "Where should server state live?"
- "How do we avoid circular imports?"
- "What’s our storybook and test setup?"
- "What’s our shared UI strategy?"

These decisions are made over and over in isolated silos - or worse, not made at all.

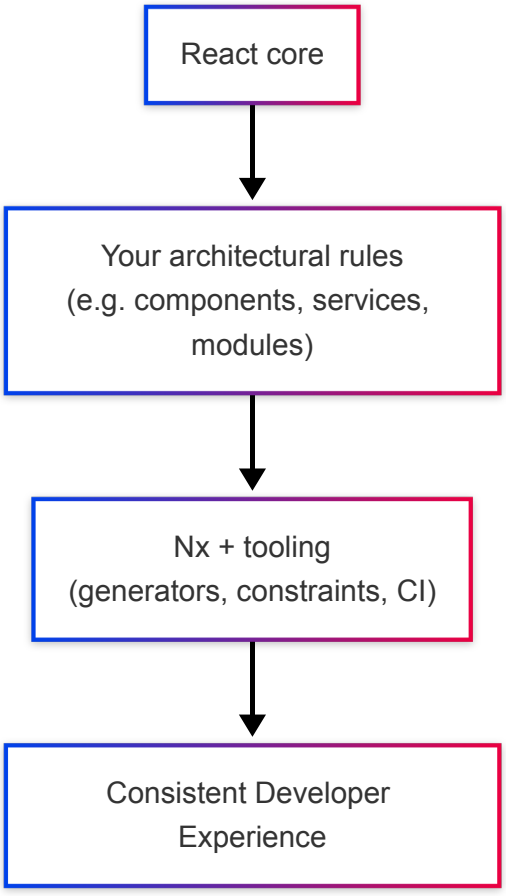
You lose:

- Developer mobility
- Fast onboarding
- Tooling compatibility
- Codemod safety

What Enterprise React Needs

Enterprise-grade React isn't about picking the flashiest libraries. It's about having **a consistent internal framework** built on top of React:

- ✅ Clear folder and library boundaries
- ✅ A consistent mental model (components, services, modules)
- ✅ Rules for code reuse (platform vs. domain, public APIs)
- ✅ Tag constraints and lint enforcement
- ✅ Generators for scaffolding with confidence
- ✅ Unified tooling, CI flows, and testing



3. The Core Architectural Model

3.1 Domains and Platforms

Most people want to start from the platform. But in our architecture, **platform is the outcome - not the starting point**. We begin with domains - which map to entire products - and allow shared abstractions to emerge organically as platforms.

To build consistent, scalable applications in React at enterprise scale, you need clear boundaries between concerns. Our architecture introduces two foundational layers:

- **Platform** - Reusable foundations and shared abstractions
- **Domain** - Product-specific business logic and features

This structure enforces separation of responsibilities, isolates scope, and enables developer mobility.

Domain Layer

Each domain maps directly to a **product** within your organization. Domains encapsulate all logic, modules, and components for that product.

Examples from a healthcare enterprise:

- `patient-portal`
- `telemedicine`
- `clinical-assistant`
- `practice-management`

Within a domain, we follow a strict structure:

- `type:component` - pure UI building blocks
- `type:service` - server state, backend integration
- `type:module` - glue logic that transforms data and connects UI and services
- `type:app` - the **exposure layer**, responsible for rendering and orchestration. It handles routing, shell layout, server-side rendering, and preloading. This layer consumes modules from the domain, but **does not include any business logic**.

How the Architecture Evolves

Let’s walk through the evolution of this structure in a healthcare enterprise.

Step 1: Start from a Domain

You begin with a single domain - say, `patient-portal`.

Inside this domain, you create a module like `appointment-module`. It contains its own logic and internal UI components.

Step 2: Reuse in the Same Domain

Next, you add another module - `lab-results-module` - that also uses a calendar component. Rather than duplicate it, you extract it to:

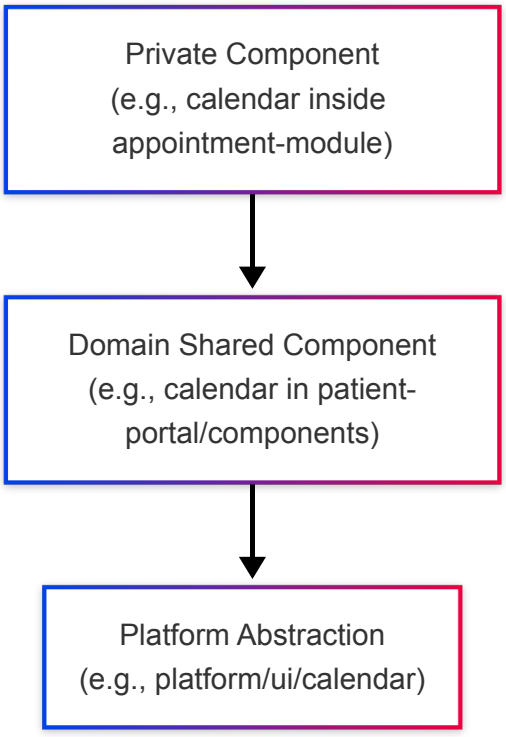
```
packages/patient-portal/components/calendar
```

Now both modules can use the same component cleanly.

Step 3: Abstract into Platform

Later, your `telemedicine` domain also needs the same calendar or auth logic. That's your signal to extract to platform:

```
packages/platform/auth/useAuthSession
packages/platform/calendar/calendar
packages/platform/video/useVideoCall
```



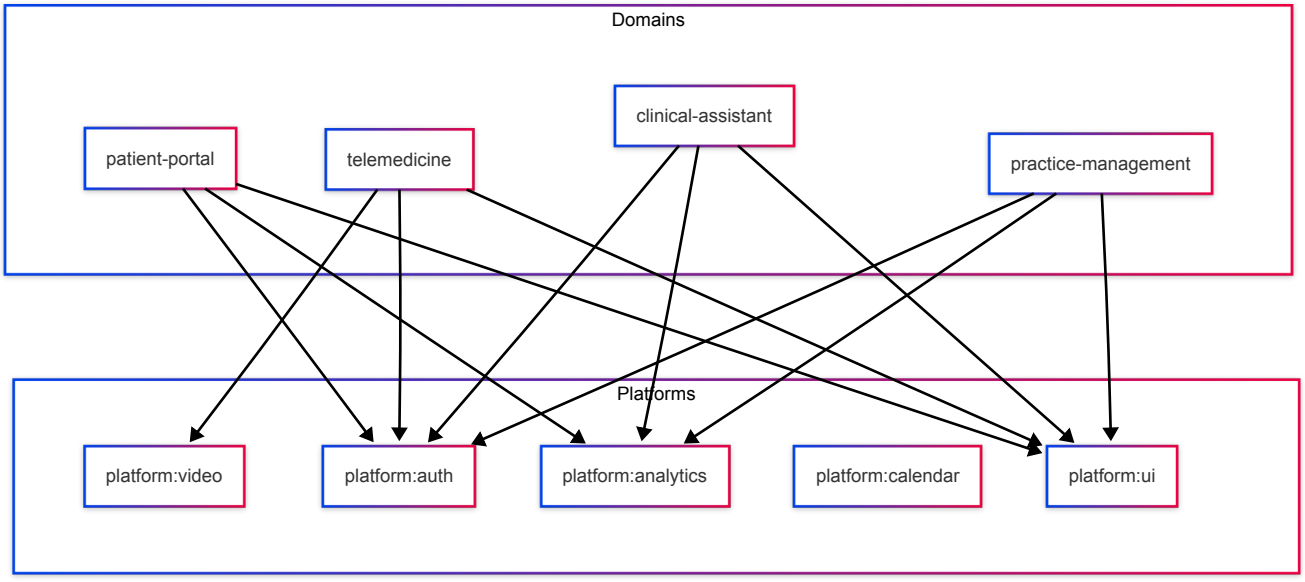
Step 4: Add More Domains

You now introduce new domains:

- `telemedicine`
- `clinical-assistant`
- `practice-management`

Each domain owns its modules and internal structure but can pull from platform utilities like:

- `platform:auth`
- `platform:analytics`
- `platform:ui`



Platform Layer

The platform layer contains libraries that are reused across multiple domains - only after patterns emerge. Platforms must be demand-driven.

Examples from a healthcare enterprise:

- `platform:auth` - central authentication logic
- `platform:analytics` - shared event tracking and GDPR compliance
- `platform:video` - video SDK abstraction for virtual consultations
- `platform:ui` - design tokens, inputs, buttons, and UI primitives

Platform can depend only on:

- Other platforms
- Utils (type:util)

3.2 Library Types within the Domain

To scale a domain effectively, we must understand how the parts compose together. Each domain follows a layered internal structure with clear responsibilities:

- `type:component` - UI-only components, visual only, no business logic
- `type:service` - server-state, backend communication, data caching
- `type:module` - the only place with business logic; glues components and services together
- `type:app` - renders modules; handles routing, layout, SSR, hydration

---

Component (UI Layer)

This is your raw visual layer - dumb components, purely driven by props. It lives inside:

```
packages/patient-portal/components/calendar
```

These components:

- Do not fetch data
- Do not contain any domain or business logic
- Are fully reusable and testable

Service (Server-State Layer)

This layer is responsible for connecting to the backend or local cache. Example:

```
packages/patient-portal/services/useAppointmentList.ts
```

It uses React Query or SWR and may wrap fetch or axios. React Query is recommended, as it keeps around 90% of data manageable within a simple and reliable cache (context-based by default).

Module (Business Logic Layer)

This is the glue and adapter layer. It is where you:

- Combine data from services
- Transform and adapt shape
- Inject it into components

Example:

```
packages/patient-portal/modules/appointment-calendar
```

The module imports:

- `useAppointmentList()` from service
- `<Calendar />` from component

App (Exposure Layer)

This is where the modules are rendered. It contains:

- Page-level routing
- Layout shell
- SSR (e.g. Next.js or Remix loaders) if needed
- Hydration if needed

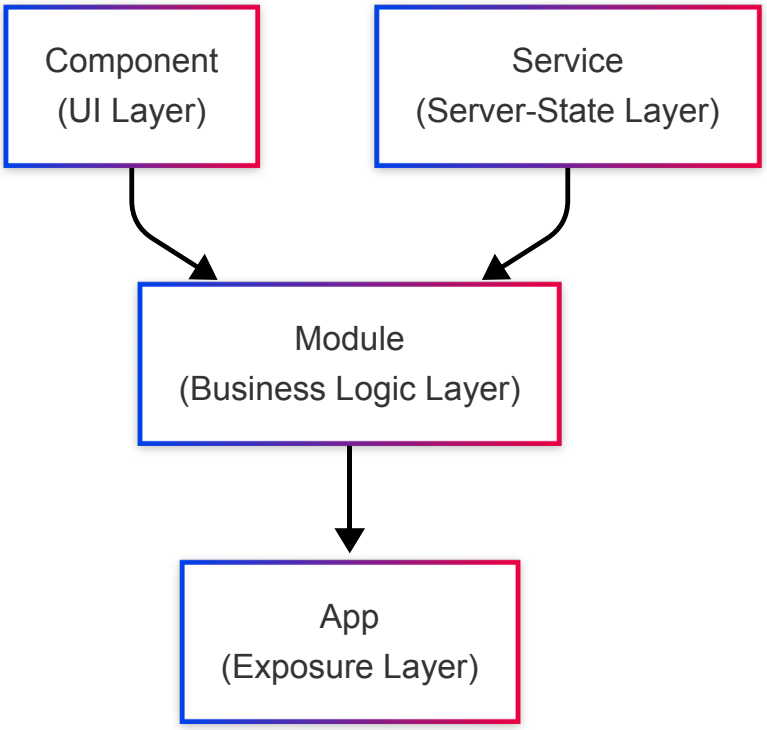
NextJS Example:

```
apps/patient-web/pages/appointments.tsx
```

```
import { AppointmentCalendar } from '@org/patient-
portal/modules/appointment-calendar';

export default function AppointmentsPage() {
  return <AppointmentCalendar />;
}
```

The beauty of this architecture is that it doesn't tie you to a specific framework like Next.js or Remix. Instead, it aligns your organization with React itself - a powerful foundation for building browser-renderable UI blocks. This flexibility is React's true strength: with a layered, constraint-driven architecture, you can decide how you want to build and serve your apps.

Frameworks like Next.js or Remix can absolutely make sense, though - they solve application-level concerns like routing, code splitting, and SSR, providing batteries-included tooling for building scalable frontends. At the same time, you're free to render your modules in a standalone TypeScript-built file, push it to S3, and serve it directly to users.



Why It Works

- You enforce separation of concerns: no app-specific logic in platform, no platform leaks into domain
- You control what can depend on what using tags and constraints
- You gain testability and deployability at the module/domain level
- You prepare your codebase for automation, codemods, and AI tooling

Part II: Building with Structure

4. Modules: The Feature Backbone

4.1 What is a Module?

In our architecture, a **module** is not a routing unit. It's a self-contained feature implementation - the place where UI and service layers meet.

The module is responsible for:

- Combining services and components into a functional unit
- Owning **business logic** (e.g., filtering, sorting, computed states)
- Serving as the entry point for rendering a feature in the app
- Being **framework-agnostic** - it doesn't care if it's rendered inside a Next.js app, a standalone test runner, or a Storybook-like demo shell

In other words, **the module is where features become real**.

Feature Context: Prescription Renewal

Let's say we're working on the **patient portal** for a healthcare product. One of the new features in the roadmap is letting users view and renew their prescriptions.

Requirements:

- Show the patient's current medications
- Only allow renewal for eligible prescriptions (not expired, not already renewed)
- Let the user trigger a renewal
- Show renewal status (pending, approved, rejected)

This functionality will be encapsulated in the module: `packages/patient-portal/modules/prescription-renewal`

Module Anatomy

```
packages/
└── patient-portal/
    └── modules/
        └── prescription-renewal/
            ├── src/
            │   ├── prescription-renewal.module.tsx      <-- Main
composition entry
            │   ├── prescription-renewal.module.spec.tsx
            │   ├── prescription-renewal.module.stories.tsx
            │   ├── hooks/
            │   │   └── usePrescriptionLogic.ts          <-- Extracting
logic to custom hooks
            │   └── index.ts                             <-- Barrel export
```

We start with a single `usePrescriptionLogic.ts` file, but as logic grows, we can extract smaller hooks like `useRenewalLogic`, `useFilteredPrescriptions`. This promotes composability and readability while fully leveraging React's custom hooks pattern.

At the top of the module is the **composition file**: `prescription-renewal.module.tsx`. Here, we bring everything together:

```tsx
// prescription-renewal.module.tsx
import { usePrescriptionLogic } from './hooks/usePrescriptionLogic';
import { PrescriptionList } from '@org/patient-
portal/components/prescription-list';

export function PrescriptionRenewalModule() {
  const { prescriptions, isLoading, onRenewClick } =
usePrescriptionLogic();

  return (
    <PrescriptionList
      items={prescriptions}
      isLoading={isLoading}
      onRenew={onRenewClick}
    />
  );
}
```

Note: the component is passed **data** and **actions**, no business logic leaks into the UI layer.

Business logic: usePrescriptionLogic

```
// hooks/usePrescriptionLogic.ts
import {
  usePrescriptionListQuery,
  useRenewPrescriptionMutation,
} from '@org/patient-portal/services/prescription';
import { Prescription } from '@org/types';

export function usePrescriptionLogic() {
  const { data, isLoading } = usePrescriptionListQuery();
  const renewMutation = useRenewPrescriptionMutation();

  const prescriptions = (data ?? []).map(prescription => ({
    ...prescription,
    canRenew: !prescription.isExpired && !prescription.isRenewed,
  }));

  function onRenewClick(prescriptionId: Prescription['id']) {
    renewMutation.mutate({ id: prescriptionId });
  }

  return { prescriptions, isLoading, onRenewClick };
}
```

> Custom hooks enable logic extraction as the module scales. This keeps modules maintainable and testable as real-world complexity grows.

4.2 Testing and developing the Module

We follow strict principles:

- We do **not** mock internal hooks like `usePrescriptionListQuery`
- We **only** mock external dependencies (like HTTP via MSW)
- We test modules as **integration units**, preferably in Storybook or through E2E-style component tests

```
// prescription-renewal.module.spec.tsx
import { render, screen } from '@testing-library/react';
import { PrescriptionRenewalModule } from './prescription-renewal.module';
import { setupServer } from 'msw/node';
import { rest } from 'msw';

const server = setupServer(
  rest.get('/api/prescriptions', (_, res, ctx) => {
    return res(
      ctx.json([
        { id: '1', name: 'Ibuprofen', isExpired: false, isRenewed: false
},
      ])
    );
  })
);

beforeAll(() => server.listen());
afterEach(() => server.resetHandlers());
afterAll(() => server.close());

test('renders prescriptions and allows renewal', async () => {
  render(<PrescriptionRenewalModule />);
  expect(await screen.findByText(/Ibuprofen/i)).toBeInTheDocument();
  expect(screen.getByRole('button', { name: /Renew/i })).toBeEnabled();
});
```

Developing

Every module should have its own **Storybook story** that mirrors the logic and mocks HTTP APIs:

```tsx
// prescription-renewal.module.stories.tsx
import { PrescriptionRenewalModule } from './prescription-renewal.module';
import { rest } from 'msw';
import type { Meta, StoryObj } from '@storybook/react';

const meta: Meta<typeof PrescriptionRenewalModule> = {
  component: PrescriptionRenewalModule,
  parameters: {
    msw: {
      handlers: [
        rest.get('/api/prescriptions', (_, res, ctx) => {
          return res(
            ctx.json([
              {
                id: '1',
                name: 'Ibuprofen',
                isExpired: false,
                isRenewed: false,
              },
            ])
          );
        }),
      ],
    },
  },
};

export default meta;
export const Default: StoryObj<typeof PrescriptionRenewalModule> = {
  render: () => <PrescriptionRenewalModule />,
};
```

Storybook enables isolated development and experimentation without requiring you to bootstrap the entire (often large) application. In fact, Storybook becomes the standard for module sandboxes when using generators. That means that whenever you scaffold a new module, your sandbox is already wired up and ready. While Storybook is the most popular choice, it’s not the only option - any lightweight sandbox app can be used to render and develop a module in isolation with fast feedback and high testability.

4.3 A/B Testing with Module Variants

This is where custom hooks shine.

Let’s say we want to experiment with a new animated UI experience. Instead of changing the main module, we create a new one:

```
prescription-renewal.module.tsx
prescription-renewal-with-new-ui.module.tsx
```

The core logic remains untouched. The new module reuses `usePrescriptionLogic`, and introduces new behavior by **composing a separate hook**, e.g. `useAnimatedListUX`. This hook can manage animation state, delays, transitions, etc.

We do **not** modify `usePrescriptionLogic`. If we did, we’d risk introducing bugs into the primary experience. By duplicating logic or composing side-by-side hooks, we isolate the blast radius of change.

This approach is intentionally WET (Write Everything Twice) because clarity and testability are more valuable here than clever DRY abstractions.

Of course, you could always make your hooks accept parameters and branch logic within them -but for A/B testing, this explicit branching pattern is often more readable and reliable.

As your product matures, introducing A/B tests at the module level becomes not only possible - it becomes trivial.

```
// prescription-renewal-with-new-ui.module.tsx
import { usePrescriptionLogic } from './hooks/usePrescriptionLogic';
import { useAnimatedListUX } from './hooks/useAnimatedListUX';
import { NewPrescriptionList } from '@org/patient-portal/components/new-
prescription-list';

export function PrescriptionRenewalWithNewUIModule() {
  const { prescriptions, isLoading, onRenewClick } =
usePrescriptionLogic();
  const { animationProps } = useAnimatedListUX();

  return (
    <NewPrescriptionList
      items={prescriptions}
      loading={isLoading}
      onRenew={onRenewClick}
      animation={animationProps}
    />
  );
}
```

Because business logic lives in composable hooks, building and testing multiple variants is effortless; you mix, match, and evolve with full confidence.

You can create a **separate Storybook file** for this new module variant (`prescription-renewal-with-new-ui.module.stories.tsx`), or keep all variants in a single stories file depending on your needs. The development experience remains smooth - setup is simple and MSW integration works the same way. There’s no architectural penalty for experimenting. You ship fast without technical debt.

This also sets the stage for adopting platform-level abstractions.

Rendering in the App Layer (e.g., Next.js)

On the app layer, it's your job to decide which module variant to render based on user segmentation. This could happen on the server side using SSR context or cookies, or via a platform-level experimentation hook.

```tsx
// pages/prescriptions.tsx (Next.js pages router example)

import { useRouter } from 'next/navigation';
import { RouterAdapterProvider } from '@org/platform/router';
import { getUserSegment } from '@/platform/experiments/getUserSegment';
import { PrescriptionRenewalModule } from '@org/patient-
portal/modules/prescription-renewal';
import { PrescriptionRenewalWithNewUIModule } from '@org/patient-
portal/modules/prescription-renewal-with-new-ui';

export default function PrescriptionPage({ userSegment }) {
  const router = useRouter();
  const routerAdapter = {
    navigate: router.push,
    replace: router.replace,
  };

  const Component = userSegment === 'experiment-group-A'
    ? PrescriptionRenewalWithNewUIModule
    : PrescriptionRenewalModule;

  return (
    <RouterAdapterProvider value={routerAdapter}>
      <Component />
    </RouterAdapterProvider>
  );
}

export async function getServerSideProps(context) {
  const userSegment = getUserSegment(context.req);
  return { props: { userSegment } };
}
```

This keeps A/B logic out of your module - your modules don’t know they’re being tested. App-level orchestration gives you full control over rendering while keeping modules focused and reusable.

4.4 Using App Layer Internals Like Routing

One important question you may have is: *how does a fully decoupled module trigger navigation or use application-level internals like routing or framework utilities?*

The answer is: we define a simple **interface**, and then **inject it at the app layer** using a React context.

For example, we define a contract for routing:

```
// packages/platform/router/types.ts
export interface RouterAdapter {
  navigate: (url: string) => void;
  replace: (url: string) => void;
}
```

We implement this in the app layer using the framework's router (e.g., Next.js):

```
import { useRouter } from 'next/navigation';
import { RouterAdapterProvider } from '@org/platform/router';
import { PrescriptionRenewalModule } from '@org/patient-
portal/modules/prescription-renewal';

export default function PrescriptionsPage() {
  const nextRouter = useRouter();

  const routerAdapter = {
    navigate: nextRouter.push,
    replace: nextRouter.replace,
  };

  return (
    <RouterAdapterProvider value={routerAdapter}>
      <PrescriptionRenewalModule />
    </RouterAdapterProvider>
  );
}
```

Then inside the module, we consume it without caring how it's implemented:

```
// inside usePrescriptionLogic.ts or a sub-hook
import { useRouterAdapter } from '@org/platform/router';

const router = useRouterAdapter();
router.navigate('/prescriptions/123');
```

This pattern keeps modules decoupled from framework-specific logic while still enabling them to participate in application behavior.

From Modules to Platforms

Once you start seeing similar logic reused across modules or even across domains - you have the freedom to elevate it.

For example, the prescription logic in `usePrescriptionLogic` might become relevant not only in `patient-portal`, but also in a future `clinical-assistant` app. That's when you move it to:

```
packages/platform/medications/PrescriptionModule/hooks/usePrescriptionLogi
c.ts
```

And because your hooks follow a consistent pattern, this move doesn't require rewrites - just imports. You're simply shuffling well-structured, decoupled logic across your workspace.

This is the payoff of composable design: you don't redesign, you refactor with confidence. You don't repeat core logic, you promote it. And your entire system benefits from structural consistency.

5. Services: Managing Server State

5.1 Why Do We Need Services?

At first glance, it may seem like we can simply use `useQuery()` from React Query directly inside a module or a feature hook. And in small-scale applications, that might be acceptable. But as the codebase and business needs grow, this approach leads to tight coupling, reduced testability, and zero separation between business logic and data access.

The **Service layer** is our answer to this - a thin, reusable abstraction for managing server state, communicating with external APIs, and exposing query/mutation logic to modules and hooks.

This layer becomes even more critical when:

- You need to **reuse server state** between pages and modules (React Query cache)
- You want to **switch data sources** (e.g. migrating from GraphQL to REST)
- You want to combine data from **multiple APIs**
- You want to **hydrate data from SSR**, or persist it during back-forward navigation
- You want to abstract access to allow things like **Apollo**, **React Query**, or any custom fetch strategy

Rules of the Service Layer

- Services **may contain domain/business logic** (e.g. combining APIs, transforming backend formats)
- Services **must not contain feature-specific logic** (e.g. sorting for a UI tab, adapting for a specific screen)
- Services expose only queries and mutations
- Services return predictable, consistent shape - typically `{ data, isLoading, error, ... }`
- They are the only layer responsible for data fetching

5.2 Service Interface and Adapter Pattern

In real-world enterprise systems, you may have legacy code using **Apollo Client**, newer modules using **React Query**, and yet other code reaching out to raw `fetch()` or external vendor SDKs. The service layer acts as an adapter boundary between the module and the underlying transport mechanism.

The best practice is to standardize the return signature of a service hook to a common shape:

```
// packages/platform/server-state/types/query-result.ts
export interface QueryResult<T> {
  data: T | undefined;
  isLoading: boolean;
  error: unknown;
  // In a real implementation, this type could also include helpers like:
  // refetch, invalidate, status, updatedAt, etc.
}
```

This common type should live in a shared `packages/platform/server-state` package and be used across all services.

> Note: This is a simplified version of what the service return type could look like. In production systems, you'd likely want a richer interface that mirrors more of what React Query, Apollo, or your chosen data layer offersv; such as `status`, `refetch`, `isStale`, and other metadata. Keeping a common wrapper lets you switch or evolve your data layer over time without affecting consuming modules.

Pro Tip: If React Query is your long-term choice for data handling, consider re-exporting its type definitions directly:

```
export type QueryResult<T> = UseQueryResult<T, unknown>; // from React
Query
```

This way, even when using legacy clients like Apollo or Redux, you can adapt their responses to this shape. That way, modules consuming services don’t need to know or care what library powers the cache they only deal with a consistent interface. This isn’t about copying caches - it’s about adapting interfaces for stability and progressive migration. Regardless of whether the implementation uses `useQuery`, Apollo, or any other data layer.

This enables modules to consume services without knowing or caring what the backend is, or how it's being queried. You can switch from Apollo to React Query, or wrap `fetch()` without changing any consuming module.

Instead of hardcoding logic or rewriting the feature, we can abstract the fetch logic in services and expose a unified contract:

```
// packages/patient-
portal/services/prescription/usePrescriptionListQuery.ts
export function usePrescriptionListQuery() {
  const restQuery = useQuery(['prescriptions'],
fetchPrescriptionsFromRest);

  return {
    data: restQuery.data,
    isLoading: restQuery.isLoading,
    error: restQuery.error,
  };
}
```

Or if using Apollo Client:

```
export function usePrescriptionListQuery(): QueryResult<Prescription[]> {
  const { data, loading, error } = useQuery(PRESCRIPTIONS_QUERY); // This
useQuery is from Apollo Client

  return {
    data: data?.prescriptions,
    isLoading: loading,
    error,
  };
}
```

No matter the source - REST, GraphQL, fetch - your hook always calls `usePrescriptionListQuery()` and gets the same structure. This standardization allows your modules to stay timeless and unopinionated about the underlying stack.

5.3 Composing API Calls in Services

Sometimes, a service needs to **combine multiple API calls** and **create a new server cache object** that the application can rely on consistently.

```
export function usePrescriptionQuery(): QueryResult<Prescription[]> {
  return useQuery(['prescriptions'], async () => {
    const internal = await fetchFromInternal();
    const vendor = await fetchFromVendor();

    return mergePrescriptions(internal, vendor);
  });
}
```

This is different from just calling two hooks and returning merged values. What we're doing here is:

- Composing two or more backend data sources
- Aggregating them into a new structure
- **Storing that structure as a cache entity in React Query**, which can now be mutated, refetched, or invalidated independently

This approach allows your mutation logic to safely update or invalidate this cache. And because this is done in the service layer, every module or page consuming `usePrescriptionQuery()` will benefit from the same cache key and hydration behavior.

This also enables the service to own all logic related to data shape and API coordination. For example, if `fetchFromInternal()` and `fetchFromVendor()` return different types, the logic to merge and normalize them into a single prescription type stays internal to the service - keeping your module free from low-level integration details. The React Query cache object you create here becomes the single source of truth for all future mutations, cache invalidation, or rehydration strategies tied to prescriptions.

SSR, Hydration, and App Context

Service hooks are also where you gain the power of React Query:

- **Hydrate from SSR** with `dehydrate()` / `hydrate()`
- **Retain data across pages**
- **Trigger optimistic updates**
- **Clear cache when needed**

6. Components - UI Layer

Component Rules and Philosophy

Components are the pure UI layer - the visual atoms and molecules that make up your application. Their job is to render and visually respond to props. Nothing more.

The Core Rules:

- A component **must be pure UI** - it cannot contain business logic.
- A component **must not talk to the app layer** - no direct routing, no analytics dispatch, no framework-coupled behaviors.
  - Instead, components expose callbacks like `onClick`, `onChange`, `onSubmit`, and let modules decide what happens next.
  - The only exception: if the component needs to perform **internal behavior** first (like animation or visual state transition), it can wrap that before calling back.
- Every component **must have a Storybook file**. We build and validate in isolation first.
- Every component **must have a unit test (** `**.spec.tsx**` **)** to validate core render behavior and interactivity.

Private Components:

If a component is only a visual subunit of another component (e.g., a row inside a list), you can extract it into its own file **without storybook or tests** - this is allowed under the "delegation" pattern.

But:

- It must **not be exported** or shared outside its parent file tree.
- It must not start to grow business logic or side effects.
- It is a local helper, not a public component.

Component Folder Structure

For our continuing example (`PrescriptionList`):

The public `index.ts` file should re-export only the main component, never any private helpers. Private visual units should live under a dedicated `/private` directory and remain internal only. This is not strictly required by Nx or Node resolution rules, but it creates a strong, enforceable convention in the codebase.

```
packages/patient-portal/components/prescription-list/
├── prescription-list.tsx              # The component
├── prescription-list.spec.tsx        # Unit tests
├── prescription-list.stories.tsx     # Storybook file
├── index.ts                           # Public re-exports
└── private/
    └── row.tsx                        # Private subcomponent
```

Why a dedicated `/private` folder? Technically, you could just avoid exporting internal helpers from `index.ts`. But in large workspaces, being explicit in the folder structure has major benefits:

- **Clarity for humans** - new joiners immediately see what is public API vs internal delegate. No need to scan exports.
- **Clarity for machines** - AST rules, linters, and custom scripts can exclude `/private` at a directory level, making automated enforcement (tests, Storybook presence, coverage checks) far simpler.
- **Consistency** - ensures internal “delegate” components don’t accidentally creep into the public API surface.
- **Future-proofing** - keeps the door open for AI assistants or low-code tools to understand visibility at a glance.

Why This Matters

Keeping components pure allows:

- Clean separation of UI and behavior
- Easy migration of components to other domains
- High Storybook reusability
- Better AI-generated or low-code integrations (because props are deterministic)

This minimal contract maximizes flexibility downstream.

Storybook is the default sandbox. It’s deeply integrated and enables automated visual testing, per-component dev, and visual QA by design or QA teams. If you prefer another sandboxing method (e.g. internal playground apps), that’s fine but Storybook remains the easiest and most scalable path.

Unit tests ensure your visual contract stays intact. You don’t test business outcomes here - only rendering logic and UI feedback.

Part III: Scaling the Codebase

7. Consistency through Tooling

7.1 Custom Generators

In large teams, manual setup is your enemy. The fastest way to ensure consistency is to generate it.

We use **Nx workspace generators** to scaffold components, services, and modules following strict rules and structure. This guarantees every developer starts with the right file structure, naming conventions, tags, and test files - without thinking.

Local Generator Setup Example

If you don't already have a local plugin:

```
nx add @nx/plugin
nx g @nx/plugin:plugin tools/my-plugin
```

Then create a generator inside it:

```
nx generate @nx/plugin:generator tools/my-plugin/src/generators/component
```

This sets up your generator like:

```
tools/my-plugin/
├── src/generators/component/
│   ├── generator.ts
│   ├── generator.spec.ts
│   ├── schema.d.ts
│   └── schema.json
```

Example: Component Generator with Specs and Stories

Let’s walk through what a real component generator might look like for this React Enterprise Framework.

schema.json

```
{
  "cli": "nx",
  "id": "component-generator",
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "description": "Component name",
      "$default": {
        "$source": "argv",
        "index": 0
      }
    },
    "directory": {
      "type": "string",
      "description": "Target library or module directory (e.g.,
packages/domain/components)"
    }
  },
  "required": ["name", "directory"]
}
```

generator.ts

```
import {
  Tree,
  formatFiles,
  generateFiles,
  names,
  joinPathFragments,
} from '@nx/devkit';
import * as path from 'path';

export default async function (tree: Tree, schema: any) {
  const componentNames = names(schema.name);
  const targetDir = joinPathFragments(schema.directory,
componentNames.fileName);

  generateFiles(
    tree,
    path.join(__dirname, 'files'),
    targetDir,
    {
      tmpl: '',
      name: componentNames.fileName,
      className: componentNames.className,
    }
  );

  await formatFiles(tree);
}
```

files/prescription-list.tsx__tmpl__

```
export function <%= className %>() {
  return <div><%= className %> works!</div>;
}
```

files/prescription-list.spec.tsx__tmpl__

```
import { render } from '@testing-library/react';
import { <%= className %> } from './<%= name %>';

describe('<%= className %>', () => {
  it('should render successfully', () => {
    const { baseElement } = render(<%= className %>());
    expect(baseElement).toBeTruthy();
  });
});
```

files/prescription-list.stories.tsx__tmpl__

```
import { <%= className %> } from './<%= name %>';

export default {
  component: <%= className %>,
  title: 'Components/<%= className %>',
};

export const Default = {
  render: () => <<%= className %> />,
};
```

Using the custom generator

This setup gives your workspace the ability to run:

```
nx g @myorg/my-plugin:component prescription-list --
directory=packages/patient-portal/components
```

And it will generate:

```
packages/patient-portal/components/prescription-list/
├── prescription-list.tsx
├── prescription-list.spec.tsx
├── prescription-list.stories.tsx
├── index.ts
```

Following this example, you can extend the generator for services, modules, queries, mutations, and more.

To see more real-world examples of generators in the React Enterprise Framework, check out Brainly's open-source toolkit: https://github.com/brainly/gene/tree/master/packages/gene-tools

Official Nx documentation: https://nx.dev/extending-nx/recipes/local-generators

7.2 Architecture Enforcement via Linting, AST Rules, and GitHub CI Checks

Generators help you start right - **AST-based lint rules and CI-based validations** help you stay right and scale safely.

We use `@nx/eslint-plugin` and custom ESLint + AST rule sets to enforce architectural constraints across every feature.

ESLint Plugin Setup Example

In `.eslintrc.json`, include:

```
{
  "overrides":[
    {
      "files":[
        "*.ts",
        "*.tsx"
      ],
      "rules":{
        "@nx/enforce-module-boundaries":[
          "error",
          {
            "enforceBuildableLibDependency":true,
            "allow":[

            ],
            "depConstraints":[
              {
                "sourceTag":"type:component",
                "onlyDependOnLibsWithTags":[
                  "type:component",
                  "type:util",
                  "type:platform"
                ]
              },
```

```
            {
              "sourceTag":"type:module",
              "onlyDependOnLibsWithTags":[
                "type:platform",
                "type:component",
                "type:service",
                "type:util"
              ]
            },
            {
              "sourceTag":"type:service",
              "onlyDependOnLibsWithTags":[
                "type:platform",
                "type:util"
              ]
            },
            {
              "sourceTag":"platform:*",
              "onlyDependOnLibsWithTags":[
                "type:util"
              ]
            },
            {
              "sourceTag":"domain:*",
              "onlyDependOnLibsWithTags":[
                "type:component",
                "type:service",
                "type:module",
                "type:util",
                "platform:*"
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

This configuration enforces boundaries such as:

- Components can only depend on other components, utility libraries, or platform code
- Services must not import components or any presentational logic
- Modules must only connect platform, service, and UI libraries not bypass boundaries
- Domains cannot leak into each other

PR-Level Structural Checks with GitHub Actions

On top of linting, you can enforce **procedural checks** to validate if a feature was scaffolded properly.

GitHub Action Example: `check-component-structure.yml`

```yaml
name: Check Component Structure

on:
  pull_request:
    paths:
      - 'packages/**/components/**'
      - 'apps/**/components/**'

jobs:
  validate-component:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node
        uses: actions/setup-node@v3
        with:
          node-version: 18
      - run: npm ci
      - name: Run structure validation
        run: node ./scripts/validate-component-structure.js
```

Then in `scripts/validate-component-structure.js`, you can validate structure using real AST parsing. Here's a simplified version:

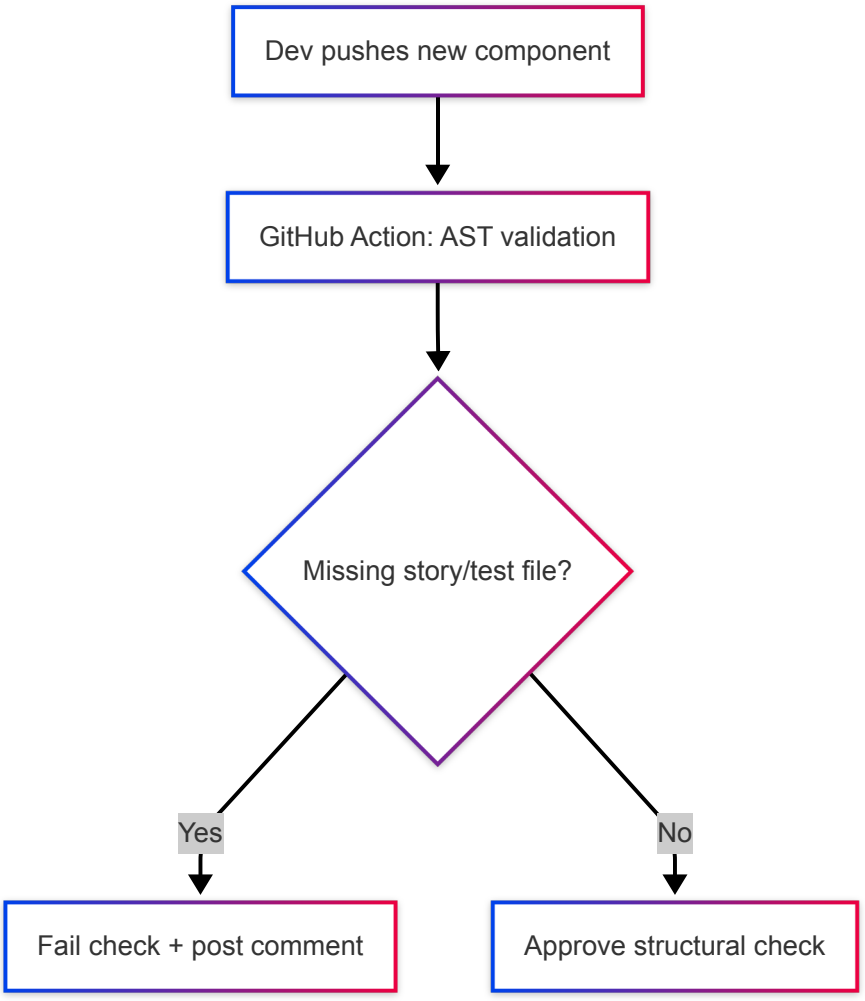
```
const { getComponentsLibsPaths } = require('./getComponentPaths');
const fs = require('fs');
const path = require('path');
const ts = require('typescript');

const componentFolders = getComponentsLibsPaths();

componentFolders.forEach(folder => {
  const files = fs.readdirSync(folder);

  const hasSpec = files.some(file => file.endsWith('.spec.tsx'));
  const hasStory = files.some(file => file.endsWith('.stories.tsx'));
  const hasIndex = files.includes('index.ts');

  if (!hasSpec || !hasStory || !hasIndex) {
    console.error(`❌ ${folder} is missing required files:`);
    if (!hasSpec) console.error(' - Missing spec');
    if (!hasStory) console.error(' - Missing story');
    if (!hasIndex) console.error(' - Missing index');
    process.exitCode = 1;
  }

  // Validate that no file from /private is exported in the index.ts file
  const indexFile = path.join(folder, 'index.ts');
  const privatePath = path.join(folder, 'private');

  if (fs.existsSync(indexFile) && fs.existsSync(privatePath)) {
    const indexContent = fs.readFileSync(indexFile, 'utf8');

    const privateFiles = fs
      .readdirSync(privatePath)
      .map(file => path.basename(file, path.extname(file)));

    privateFiles.forEach(privateComponent => {
      const exportRegex = new RegExp(`export.*
['\"]\.\/private\/${privateComponent}['\"]`);
      if (exportRegex.test(indexContent)) {
        console.error(`❌ ${privateComponent} from /private is exported in
index.ts (folder: ${folder}).`);
        process.exitCode = 1;
      }
    });
  }
    }
  }
});
```

You can also optimize this check by validating only the components affected in the current PR.

Here's an extended example that uses Nx to get all component paths, and GitHub's API to determine which files were changed:

```
// @ts-check
const github = require('@actions/github');
const core = require('@actions/core');
const { readWorkspaceConfig } = require('@nx/workspace');
const fs = require('fs');

function getComponentsLibsPaths() {
  const { projects } = readWorkspaceConfig({ format: 'nx' });
  const projectKeys = Object.keys(projects);

  const componentsLibs = projectKeys
    .filter(project => (projects[project].tags ||
[]).includes('type:component'))
    .map(project => projects[project].root);

  const componentsInsideModules = projectKeys
    .filter(project => {
      const tags = projects[project].tags || [];
      return (
        tags.includes('type:module') ||
        tags.includes('type:core-module') ||
        tags.includes('type:application-module-library')
      );
    })
    .map(project => {
      const projectRoot = projects[project].root;
      const libPath = `${projectRoot}/src/lib`;
      if (!fs.existsSync(libPath)) return [];

      return fs
        .readdirSync(libPath)
        .filter(directory => !/.+\.(tsx?|jsx?)/.test(directory))
        .map(directory =>
`${projectRoot}/src/lib/${directory}/components`)
        .filter(directory => fs.existsSync(directory));
    });
```

```
  const componentsInsideApps = projectKeys
    .filter(project => (projects[project].tags ||
[]).includes('type:app'))
    .map(project => `${projects[project].root}/components`)
    .filter(directory => fs.existsSync(directory));

  return [
    ...componentsLibs,
    ...componentsInsideModules.flat(),
    ...componentsInsideApps,
  ];
}

async function getChangedComponentFiles(token) {
  const client = github.getOctokit(token);
  const context = github.context;
  const baseSha = context.payload.pull_request.base.sha;
  const headSha = context.payload.pull_request.head.sha;

  const response = await client.rest.repos.compareCommits({
    base: baseSha,
    head: headSha,
    owner: context.repo.owner,
    repo: context.repo.repo,
  });

  if (response.status !== 200) {
    core.setFailed(`compareCommits request failed with code
${response.status}`);
  }

  const changedFiles = response.data.files
    ?.filter(file => ['added', 'modified',
'renamed'].includes(file.status))
    .map(file => file.filename);

  const componentsLibsPaths = getComponentsLibsPaths();

  return changedFiles.filter(filePath =>
    componentsLibsPaths.some(componentRoot =>
filePath.startsWith(componentRoot))
  );
}

module.exports = { getComponentsLibsPaths, getChangedComponentFiles };
```



This refined script ensures you're only analyzing the component folders actually affected in a pull request making your CI faster and more targeted.

This validates that:

- `.spec.tsx`, `.stories.tsx`, and `index.ts` exist
- Private components exist only inside a `private/` folder
- No file inside `/private` uses `export` declarations

You can scale this pattern to validate naming, folder nesting, prop structure, or anything else using AST traversal. Above script is only simple demo of what is possible.

This enables rules like:

> If a component is added or modified, it must follow all structural and visibility constraints.

We can use `@nx/eslint-plugin` and custom AST rules to enforce architectural boundaries and prevent regressions.

All of this is automatable. Combine ESLint for in-editor enforcement, CI GitHub Actions for file structure guarantees, and Nx tagging to gate architectural boundaries.

Optional: Nx Conformance (Powerpack/Enterprise)

If you prefer **declarative, workspace-level policy** over bespoke CI scripts, Nx provides **Conformance** which lets you codify standards as rules and run them centrally.

- **Scope:** complements ESLint - enforce repo-wide policies (ownership, boundaries, required files), not just import graphs.
- **Authoring:** rules written in TypeScript; can read your workspace graph and file system, so the same logic from the demo script can live as a rule.
- **Execution:**
  - Local: `nx conformance` (with auto-fix generators, if configured)
  - CI: `nx conformance:check` (fail PRs on violations)

Minimal `nx.json` snippet (illustrative):

```
{
  "conformance": {
    "rules": [
      {
        "rule": "@nx/conformance/enforce-project-boundaries",
        "options": {
          "depConstraints": [
            {
              "sourceTag": "scope:shared",
              "onlyDependOnProjectsWithTags": ["scope:shared"]
            }
          ]
        }
      },
      {
        "rule": "@nx/conformance/ensure-owners",
        "projects": ["!experimental-app"]
      },
      {
        "rule": "./tools/local-conformance-rule.ts"
      }
    ]
  }
}
```

CI (illustrative)

```
- name: Enforce conformance rules
  run: npx nx conformance:check
```

> Conformance requires **Nx Powerpack/Enterprise**. To keep this chapter vendor-neutral, details live in **Appendix C** (optional Nx Cloud integration). Your OSS path above (ESLint + AST + GH Actions) remains fully valid.

AI-Aware Tooling

The stronger your structure, the better AI tooling can work.

When modules, services, and components follow repeatable shape:

- IDE agents (like Cursor) can suggest consistent fixes
- GitHub bots can autofix or annotate PRs that violate boundaries
- AI can reliably scaffold new features without human input

This architecture is built for automation - whether human or AI-driven.

How Enforcement Works in Practice

Depending on your workflow, you can:

- **Block PRs** entirely when checks fail (via GitHub branch rules)
- **Reject commits** on pre-commit if ESLint fails
- **Allow warnings**, but report in CI dashboards

Enforcement isn’t about punishment - it’s about **scaling quality** without handholding.

8. CI, Testing, and Affected Builds

As enterprise codebases grow, testing everything becomes hard. Speed and safety at scale require intelligent tooling to:

- Run only what changed
- Test at the right layer**Note - Caching beats “affected”:** Affected selects the *right* work. **Caching** skips work entirely. With caching enabled, many tasks return instantly - even if you run more than the minimum. We’ll rely on **local caching** (open source) and outline how to set up **DIY remote caching** architecturally. If you prefer a managed route, see **Appendix C: Nx Cloud**.
- Build selectively

This chapter describes the patterns and tools used to keep feedback loops fast without compromising confidence or stability.

8.1 Affected Commands and CI

In more complex pipelines we start by using the `nx-set-shas` action to determine the commit range (`NX_BASE` / `NX_HEAD`) for the `nx affected` command. Then we run `nx show projects --affected` or `nx run-many`. This lets you split work across CI runners or GitHub matrix jobs.

Tip: The `nx g ci-workflow` generator scaffolds a GitHub Actions template that includes `nx-set-shas` and the `nx affected` steps, saving setup effort.

In monorepos with hundreds of libraries, testing and building everything on every commit is inefficient. **Nx provides affected commands** that intelligently detect changes and run only the necessary tasks.

Note - Caching beats “affected”: Affected selects the *right* work. **Caching** skips work entirely. With caching enabled, many tasks return instantly - even if you run more than the minimum. We’ll rely on **local caching** (open source) and outline how to set up **DIY remote caching** architecturally. If you prefer a managed route, see **Appendix C: Nx Cloud**.

How the affected works:

```
nx affected:test
nx affected:build
nx affected:e2e
```

These commands compare the base and head of the commit range (automatically in CI) and identify which projects (apps, modules, services, components) are impacted.

Example CI step using `nx affected`

```
- name: Test Affected Projects
  run: nx affected --target=test --parallel=3
```

Example CI step using `nx run-many`

This approach gives more flexibility for advanced CI setups.

```
- name: Get affected projects
  id: affected
  run: echo "projects=$(nx show projects --affected --target=test --
select=projects)" >> $GITHUB_OUTPUT

- name: Run tests for affected projects
  run: |
    IFS=',' read -ra projects <<< "${{ steps.affected.outputs.projects }}"
    nx run-many --target=test --projects="${projects[*]}" --parallel=8
```

Or even use matrix strategy:

```
- name: Set up project matrix
  id: set-matrix
  run: |
    echo "::set-output name=matrix::{ \"include\": [ { \"project\": \"app-
a\" }, { \"project\": \"lib-b\" } ] }"

- name: Test project in matrix
  run: nx run-many --target=test --projects=${{ matrix.project }} --
parallel=1
```

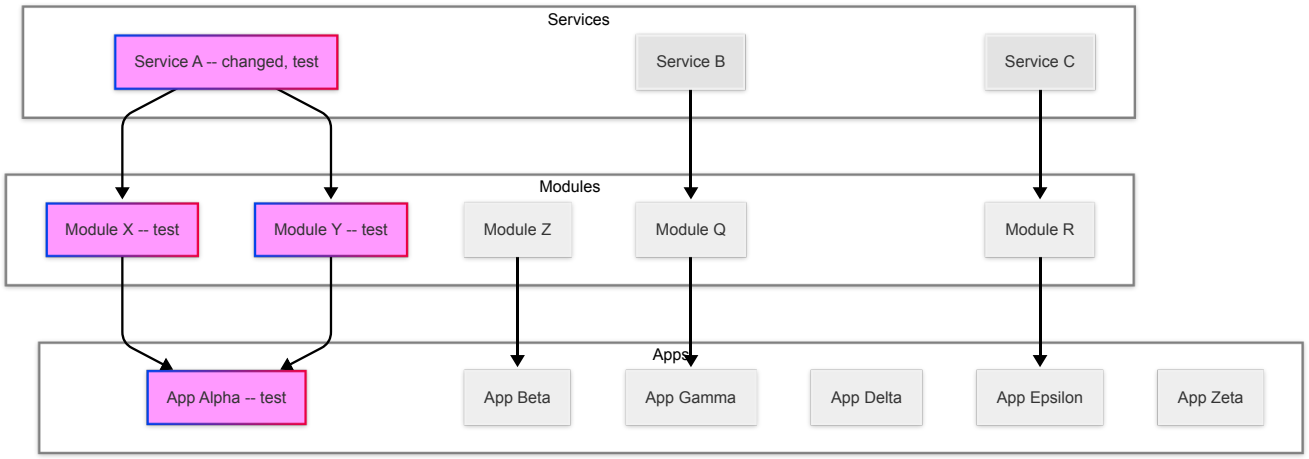
Both methods leverage the same dependency graph - one is just more configurable for scaling CI workflows across runners or jobs.

You can use any target - `test`, `build`, `lint`, `e2e` - depending on your stage in the pipeline.

What is a Matrix Strategy?

Matrix strategy lets you dynamically split CI jobs across a list of values (e.g. projects), running each job in parallel. This is helpful when testing many projects concurrently to speed up total execution time.

Visualizing Affected Graph Execution



In this diagram:

- Only **Service A** changed.
- This affects **Module X** and **Module Y**, and transitively, only **App Alpha**.
- All other modules and apps remain untouched.
- Nx detects this through the dependency graph and triggers tests only for the affected chain.

Local Caching (Open Source) - Your primary speedup

Nx fingerprints each task (sources, deps, env, inputs) and **replays outputs** when nothing relevant changed. This works **out of the box** on developer machines and CI runners.

- **Cacheable targets:** typically `build`, `test`, `lint`, `e2e`.
- **Effect:** unchanged tasks complete in seconds with no work.
- **Practice:** it's often faster - and safer - to run **more** with cache than over-optimize selection.

Remote Caching - DIY architecture vs managed

You can extend cache reuse **across machines and runs** in two ways:

A) DIY Remote Cache (no Cloud) - Architecture
Think of the Nx cache as a content-addressed store with two layers:

- **Keying:** Each task key is a deterministic hash of inputs (files, deps, env).
- **Artifacts:** A task's outputs (files, terminal output metadata) stored under that key.
- **Transport:** Choose a transport that lets runners **restore** cache entries before work and **persist** new ones after work. Common patterns:
  - **CI cache mechanism** (e.g., pipeline cache/artifact store) - simple to wire, good for short-lived reuse.
  - **Object storage** (e.g., any blob store) - durable, cross-runner, cross-PR reuse.
  - **Shared volume** (e.g., network disk) - fast in tightly controlled infra.

Operational guidelines (vendor-neutral):

- **Scope intentionally:** segment by repo/branch or by "release trains" to avoid cross-contamination.
- **Eviction & TTLs:** configure lifecycle to cap growth and prune stale keys.
- **Atomicity:** writes should be "all-or-nothing" per key to avoid partial artifacts.
- **Integrity:** rely on the task hash (and optionally checksums) to prevent cache poisoning.
- **Cold-start strategy:** warm from a stable branch baseline when possible.
- **Observability:** track hit/miss rates; a falling hit rate usually signals input drift.

B) Managed Remote Cache (Nx Cloud)
If you want remote caching, task distribution, and CI self-healing **without building the transport + ops**, use Nx Cloud. We keep the details out of the main text - see **Appendix C: Nx Cloud** for the deep dive and copy-paste CI examples.

Both methods leverage the same dependency graph. With **local caching** they're already fast; a **DIY remote cache** makes results reusable across runners. For managed remote caching, task distribution, and self-healing CI, see **Appendix C: Nx Cloud**.

8.2 Layered Testing Strategy

We align our testing strategy with our architectural layering:

Modules

- **Tested in isolation** using `@testing-library/react` + MSW to mock API services.
- These tests cover integration of UI and service logic (e.g. the full prescription renewal flow).
- Do not require bootstrapping the app.

Components

- **Unit tested** with `jest` and `@testing-library/react`.

Services

- **Pure unit tests** or mocked fetch layer (e.g. using `msw` or `axios-mock`).
- Should not contain feature-specific logic (covered in modules).

Applications

- **End-to-end (E2E) tested** using Cypress or Playwright.
- These tests verify routing, integration across modules, SSR, auth, and overall sanity.

The goal is speed without losing coverage. Here's how Nx helps:

- A commit that updates a component will only trigger:
  - Its `.spec.tsx`
  - Any modules that use that component
  - The app, if component is rendered there (tracked by graph)
- A commit that changes shared service logic will run:
  - All modules using that service
  - Tests for that service only
  - Affected apps (if included in build graph)
- A commit that touches only docs or markdown files? No tests at all.

9. Developer Mobility and productivity

In a well-structured monorepo, code isn't owned by gatekeepers - it's readable, learnable, and buildable by any engineer. The moment you lock down conventions and structure, you unlock a new kind of team flexibility: **developer mobility**.

Mobility means a developer working in `telemedicine` app today can ship a bugfix or feature in `clinical-assistant` app tomorrow - without onboarding hell. They understand the file structure, recognize patterns, reuse shared components, and don't have to learn new mental models just to contribute.

Mobility turns every engineer into a force multiplier.

The Opposite of Mobility Is Reinvention

In unstructured codebases, every product team becomes its own startup:

- Different folder structures
- Custom state management hacks
- Business logic scattered across UI
- Fetching patterns copied and mutated N+1 times

What happens? Devs stay siloed. Fixes are delayed. Engineers avoid touching "foreign" codebases. You hear things like "Oh that's Team B's thing, I don't know how it works."

Mobility kills this bottleneck.

What Enables Mobility?

We achieve mobility through:

- **Consistent layering**: `component`, `service`, `module`, and `app` mean the same thing in every domain. A developer can jump between `telemedicine` and `patient-portal` and immediately know where to start.
- **Predictable tags**: `type:component` never contains business logic. `type:service` is only for server state. `type:module` is always the glue and the only place business logic lives.
- **Clear platform boundaries**: If a developer touches `platform:auth`, they know it's shared, DRY, and core - not to be changed lightly. Domains consume, but don't mutate, platform logic.
- **Shared CI and linting rules**: No surprises between teams. No one "forgets" to write tests or violates structure because enforcement is automatic.

Concrete Example: From Patient Portal to Telemedicine

Let's say a developer worked on `patient-portal/modules/prescription-renewal`.

Tomorrow, they jump into `telemedicine/modules/appointment-calendar`.

They already know:

- Where to find reusable components: `packages/telemedicine/components/calendar`
- How server state is handled: `packages/telemedicine/services/useAppointmentListQuery.ts`
- Where the business logic lives: in `modules/appointment-calendar`, not inside UI
- How routing works: through the `app/telemedicine-app/pages/index.tsx` exposure layer
- What rules are enforced: same linting, same generator setup, same Storybook conventions

They don't need onboarding. They don't need to "ask around." They're productive **day one**.

That's what good structure buys you.

9.1 Rules Aren't Just for Humans - They're for Machines Too

Most engineers think rules exist to prevent junior mistakes or align team behavior. But that's only part of it.

The real reason we care about structure, constraints, tags, and file layouts is this:

> **Machines can't guess. They need structure to help you.**

Consistency Is the API Between Developer and Machine

AI tools, generators, codemods - they all depend on predictable patterns. Not just syntax patterns, but **semantic ones**:

- What does a `module` mean in your codebase?
- Where is data fetched from?
- What's reusable vs what's private?
- Where should new logic go?

If you don't define those clearly, tools can't help you. They can't scaffold new features. They can't safely refactor. They can't generate useful tests. They'll hallucinate or break things.

But when you do define them, something powerful happens.

Human-Defined Rules → Machine-Augmented Work

Because you define library tags (`type:component` , `type:service` , `platform:calendar`), you can write AST rules to enforce them. But more than that - now AI tools like Cursor or GitHub Copilot can:

- Understand what lives where
- Generate files in the right places
- Suggest changes that align with conventions
- Help onboard new developers with confidence

You’re building a system where both humans and machines can contribute without stepping on each other.

Structure becomes a shared contract - between your team, your tools, and your AI agents.

10. AI-Assisted Development - From Structure to Intelligence

Why Consistency is Key to Codemods and AI

- **Structure Enables Understanding**: Consistency is the foundation for code parsing and safe transformation. AI tools are not magic - they rely on patterns. If you’ve tagged your libraries, if you follow layering principles, if your services don’t mix UI logic, then an AI model (or a codemod engine) can actually reason about your code.
- **Codemods Need Predictability**: Every codemod script assumes certain patterns. Without consistent structure, you need a human to review every edge case. With structure, codemods become safe, repeatable, and scalable across hundreds of apps.
- **AI Gets Smarter in Predictable Environments**: Tools like Copilot or Cursor can generate better suggestions when the codebase is stable, layered, and semantically clear. For example, knowing that `modules` are the only place with business logic helps Copilot propose correct `useEffect` or `mutation` logic.

Enabling Safe Automation and Generation

- **Generators That Scale**: Once structure is enforced, you can scaffold new components, services, or modules with total safety. Nx generators, custom templates, and even GPT-generated scaffolds all become viable when they have a predictable destination.
- **Testable, CI-Enforced Changes**: When structure is codified, CI can enforce it - and AI can integrate with it. Want to enforce that every component has a test and story? Your GitHub Action and your AI linter can check and even produce that together.

- **Zero-Risk Refactors**: With consistent naming, tagging, and layering, codemods can rename props, move files, or refactor logic without breaking consumers. AI-backed refactors rely on these same guarantees.

The Future of Developer Experience

- **From Manual to Intent-Driven Work**: Developers won't write every line of code. Instead, they'll describe what they want - and AI agents will scaffold the foundation. But this only works if your codebase gives the AI clear rules to follow.
- **Codified Context, Shared Language**: With platform libraries, strict boundaries, and enforced structure, every developer - and every AI agent - speaks the same language. You don't need tribal knowledge to build features.
- **Collaboration Between Human and AI**: The best developer experience isn't just faster - it's safer and smarter. You describe the intent, the AI suggests a patch, CI enforces boundaries, and you review the result. That's the loop.

This is not science fiction. It's happening now - and your structure is the entry ticket.

10.1 AI Assisted development

Shared Rule Enforcement Across Environments

One of the most powerful effects of structured architecture is that your rules aren't tied to one tool - they become universal. Whether your AI assistant runs inside the developer's IDE (like Cursor) or on GitHub (as part of a pull request review with GitHub Copilot), they can follow the same rules.

Example: GitHub Copilot Code Review Rules

GitHub Copilot can use natural language coding guidelines defined in your repo settings:

```
{
  "coding_guidelines": [
    "Use React Query for all server-state hooks",
    "Business logic must live in modules, not services",
    "All exported components must have unit tests"
  ]
}
```

These guidelines are interpreted by the Copilot Code Review agent when analyzing pull requests.

Example: Cursor IDE Rules

Cursor uses `.cursor/rules/*.mdc` files to define per-project behavior:

```
# file: .cursor/rules/enforce-structure.mdc

- Only `type:module` libraries can contain business logic.
- Components must be pure UI and live in `type:component` libraries.
- Use `useQuery` and `useMutation` only inside services.
```

These rules guide the AI suggestions made inside the IDE - ensuring the code it proposes matches what your system expects.

By storing both GitHub and Cursor rules in your repository, you ensure that whether a developer is working locally or reviewing a PR, the feedback and suggestions are aligned. And it applies to any other tool too. Nx itself has started baking these ideas directly into the developer workflow. For example, Nx's **MCP (Model Context Protocol) integration** allows AI agents to query workspace rules and dependency graphs directly, instead of guessing. And features like **self-healing CI** use these rules automatically to detect and fix common issues during builds. These are not just "nice-to-have" extras, they show how a well-structured workspace can unlock machine assistance at the tooling level, not just through external agents. For teams that want to go deeper, Nx Cloud extends these concepts with advanced **self-healing CI** capabilities. We'll cover those in detail in **Appendix C: Nx Cloud**.

Nx MCP (Model Context Protocol) - what it gives your AI tools

When enabled, the Nx MCP server exposes your monorepo's **project graph, tasks, generators, ownership, tech stacks, and docs** as first-class tools the AI can call. Instead of guessing, the assistant can ask Nx for "projects that depend on X", "run task Y in project Z", "open graph", or "where is the generator for …" – using a local MCP server hosted by Nx Console (or stdio, depending on your setup). You can read the up-to-date documentation here: https://nx.dev/getting-started/ai-integration

High-Level Architecture: Custom AI Tooling Across Local + CI

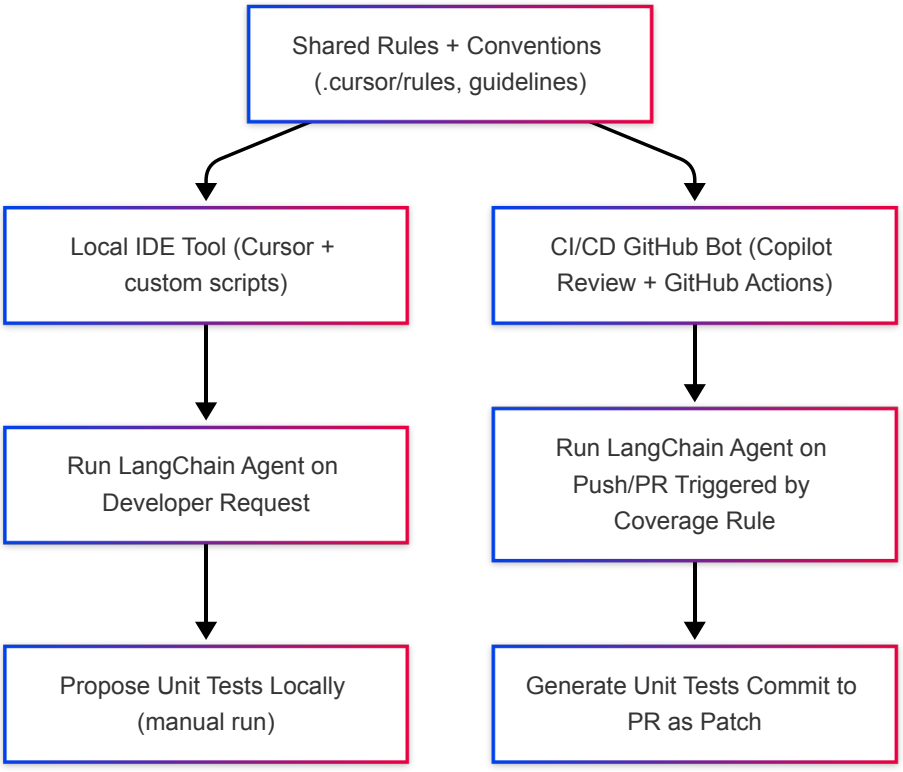
Here's how you can design your own AI-based tooling system that works across environments - from local development to automated CI reviews.

Practical Example: Custom Test Generator

Imagine you're building a tool to auto-generate unit tests for `type:component` and `type:module` libraries. Your setup might include:

- **LangChain Agent**: Accepts a component file and generates tests.
- **Shared Rules File**: Ensures business logic isn't tested inside the component.
- **Cursor Integration**: Lets a dev right-click a file and run `Generate Tests`.
- **GitHub Action Integration**: When CI notices a module lacks test coverage, it auto-triggers the same LangChain agent and proposes a PR patch with the missing tests.

Thanks to shared structure and tagged libraries, the same tooling works everywhere.



Key Takeaway

This is how AI development becomes predictable, scalable, and safe - without sacrificing speed or creativity. In practice, this means you don't have to build everything yourself. Nx already comes with a **minimal set of AI-ready rules** preconfigured in new workspaces, and its ecosystem is evolving quickly to expose project structure and constraints to AI assistants. Treat this as your baseline: a workspace that is both human-friendly and machine-readable from day one.

Appendices

Appendix A: Getting Started with Brainly Gene Framework

Gene is Brainly's internal React framework - open-sourced and built on the exact principles this ebook lays out. It's designed for enterprise speed: feature development in isolation, modules that render anywhere, enforced consistency, and everything scaffolded by generators.

If you've read the previous chapters, this will feel familiar. That's the point. Gene isn't just an implementation - it's a real-world, production-ready proof of concept. In this appendix, you'll go from zero to a working feature module in minutes.

Let's build your first Gene app.

Step 1: Create a Workspace

Start with the Gene CLI:

```
npm install nx -g
npm init @brainly-gene acme
cd acme
```

This sets up a full Nx workspace, pre-wired for Dependency Injection, testing, linting, Storybook and tags - the whole stack.

Step 2: Generate Your App

Gene works with any React setup. But the fastest path is using its built-in generator for Next.js apps:

```
npx nx g @brainly-gene/tools:nextjs-app --name my-first-app --directory
apps/domain-a/my-first-app --tags=domain:domain-a --no-interactive
```

This creates a `domain-a` Next.js app with:

- A DI container already registered
- App layer (exposure layer) ready to render modules
- Tagged app libraries for future constraints

If you want to use React Router instead of Next.js, that works too, just generate a plain React app via `@nrwl/react:app` and set up DI bindings manually.

To run the app:

```
npx nx serve my-first-app
```

And open `localhost:4200` - app is up and running and ready for development.

Step 3: Create a Feature Module

Let's see Gene's module pattern in action. Generate a module:

```
npx nx g @brainly-gene/tools:module
```

Choose a name (e.g. `Simple`) and pick an app to generate module for (my-first-app will be the only one to pick) - it will generate module library too in first run.

Notice in `project.json` that it will already pick correct domain as you already pointed app you are creating module for: `["domain:domain-a", "type:module", "type:application-module-library"]
This is typical model of development first for app, if something will become core - then we will move it core module / platform level.

> 💡 **Protip**: Use https://nx.dev/recipes/nx-console in VSCode or WebStorm for a GUI experience.

Now spin up Storybook for isolated development:

```
npx nx run my-first-app-app-modules:storybook
```

You can now develop and test the module in total isolation - no need to wire anything into your app yet. It's a standalone feature, just open `http://localhost:4400/ and code.

Step 4: Add a Component

Let’s add a UI component to your module:

```
npx nx g @brainly-gene/tools:component
```

- On the question about component type, pick: `I would like to create a basic component with defaults for props, events, copy, styles.`
- Then pick `my-first-app-app-modules` as an library you want to generate component in and pick `simple-module`
- Component name: e.g ItemsList

This generates a **private** component (i.e., only usable in this module). Notice that entire structure of the component has been created and kept, together with stories, specs and events files. Thanks to having story file - you can open storybook we run earlier and see component on the list which allows for development in isolation.

Want to share it? Generate a new shared UI library instead and run above component generator to generate components to the new library, just follow same steps as above.

```
npx nx g @brainly-gene/tools:components-library
```

Step 5: Build Your Component

Let's keep it simple. Here's a minimal `ItemListComponent` that takes a list of items and dispatches an event when an item is clicked:

```
import React from 'react';
import { dispatch } from '@brainly-gene/core';
import {
  ItemsListEventsType,
  ItemsListOnClickType,
} from './ItemsListEventsType';
import { Button, Flex, Box } from 'brainly-style-guide';

export type PropsType = Readonly<{
  items: Array<{
    id: string;
    name: string;
  }>;
}>;

const ItemsList = ({ items }: PropsType) => {
  const handleOnClick = React.useCallback((e: React.MouseEvent, id:
string) => {
    dispatch<ItemsListOnClickType>(e.target,
[ItemsListEventsType.ON_CLICK, { id }]);
  }, []);

  return (
    <div data-testid="items_list">
      {items.map((item) => (
        <Box border key={item.id}>
          <Flex justifyContent="space-between">
          {item.name}
          <Button onClick={e => handleOnClick(e, item.id)}>Visit</Button>
          </Flex>
        </Box>
      ))}
    </div>
  );
};

export default React.memo<PropsType>(ItemsList);
```

Notice:

- It accepts data via props
- It uses `dispatch()` to dispatch a custom event.
- It's fully decoupled — no app dependencies, no internal logic

Gene enforces this pattern for component isolation.

Step 6: Add a Service (with Real Data)

Let's fetch some data. Generate a service:

```
npx nx g @brainly-gene/tools:service --name=list-data --directory=domain-a/services --serviceType=react-query --tags=domain:domain-a
```

Update the generated API URL and types:

```
const apiUrl = 'https://api.sampleapis.com/beers/ale';
```

And type example:

```
type Beer = {
  id: number;
  name: string;
  price: string;
  rating: { average: number; reviews: number };
  image: string;
};
```

Now your service returns predictable data via React Query — with `loading`, `error`, `refetch`, etc.

Step 7: Glue It Together in a Module

Now connect everything inside your module, let's create a custom hook to fetch and transform the data for the view, `hooks/useSimpleListData.ts`

```
import { useListDatas } from '@acme/packages/domain-a/services/list-data-service';

export const useSimpleListData = () => {
  const { data, isLoading, error } = useListDatas({});

  return {
    data: data
      ? data?.map((item) => ({ id: item.id, name: item.name }))
      : undefined,
    isLoading,
    error,
  };
};
```

and glue it together with component inside the module:

```
import React from 'react';
import { useRef } from 'react';
import { createGeneModule, useMediator } from '@brainly-gene/core';
import { useSimpleListData } from './hooks/useSimpleListData';
import ItemsList from './components/ItemsList/ItemsList';
import { ItemsListEventsType, ItemsListOnClickType } from
'./components/ItemsList/ItemsListEventsType';

function RawSimpleModule() {
  const ref = useRef(null);
  const { data } = useSimpleListData()

  useMediator<ItemsListOnClickType>(ItemsListEventsType.ON_CLICK,
(payload) => {
    // fire on ItemsList click
    console.log(payload); // do something with the payload
  }, ref);

  return (
    <div ref={ref} data-testid="simple_module_id">
      {data && <ItemsList items={data} />}
    </div>
  );
}

const {
  declarations,
  identifiers,
  module: SimpleModule,
  container,
} = createGeneModule({
  module: RawSimpleModule,

  declarations: {},
});

export { SimpleModule, declarations, identifiers, container };
```

Pattern recap:

- Data is fetched via a service (`useListData`)
- Events are mediated (`useMediator`)

- App-specific functionality like navigation is injected via DI

This is pure **module-level logic**, exactly as described in Part II of the book.

Step 8: Expose via the App Layer

Last step - render the module in your app:

```
// apps/domain-a/my-first-app/pages/index.tsx
// just import
import { SimpleModule } from '@acme/packages/domain-a/my-first-app/app-modules';

// and render
export default function HomePage() {
  return <SimpleModule />;
}
```

This is the exposure layer. It's only responsible for wiring - no logic here. Everything else lives in modules, services, and components.

What Just Happened?

You just built a feature module using:

- Code generation for 100% of the boilerplate
- Strict layering: app → module → service + component
- DI to keep framework logic outside of your business logic
- Storybook-first development for fast feedback
- Type-safe React Query for API access
- Event mediation for clean communication

In short: this is what enterprise React architecture looks like in practice.

Everything you've read in this book? Gene applies it, codifies it, and runs it in production, serving over 15 million users a day.

One Last Note

This appendix is just a quick intro. Gene handles much more:

- Branching and module variations
- Core modules and shared logic
- Optimistic updates and cache invalidation
- DI adapters for platform-level wiring
- Storybook + integration test layering

We've iterated on these patterns for years. What you see here is just the foundation — but that's what matters most. Build the foundation right, and everything scales.

If you want to explore further, check the open-source repo: https://github.com/brainly/gene

Appendix B: Nx Cloud

Throughout this book, we've explored how Nx helps you build and scale monorepos locally. But as your monorepo grows, your CI pipeline can quickly become a bottleneck without the right approach.

Nx Cloud extends Nx's capabilities to your CI environment. While Nx itself is MIT-licensed and open source, Nx Cloud is the commercial offering that provides intelligent caching, distributed task execution, and AI-powered automation to keep your pipelines fast and reliable. A generous hobby plan is available to explore its features.

In this appendix, we'll explore the key features of Nx Cloud and how they work together to optimize your CI pipeline.

Optimizing Time To Green

The main premise of Nx Cloud is to reduce your Time To Green (TTG): the total time from when code is written until the PR is green and ready to merge.

Nx Cloud tackles this through three complementary approaches:

Making CI runs fast: This comes in layers, starting with remote caching as the foundation. You can then add distributed task execution across multiple machines, and finally incorporate task atomization for long-running tests. Each layer builds on the previous one to progressively reduce execution time.

Eliminating delays: Self-healing CI uses AI to automatically identify and fix broken PRs, while flakiness detection and automatic retries handle intermittent test failures. These features keep your PR moving forward without requiring constant attention.

Improving reliability: Nx Cloud uses a task-based execution model rather than a traditional VM-based approach. Instead of assigning fixed work to specific machines, any agent can pick up any task. If an agent fails or is slow, others can take over the work, dramatically improving overall pipeline reliability

Connect to Nx Cloud

Starting a new Nx workspace with Nx Cloud

Visit https://cloud.nx.app/get-started to create a fully configured Nx workspace with Nx Cloud enabled. Connect via GitHub, and Nx Cloud will create a PR in your repository with all necessary configuration. Select your workspace and organization, and you'll have distributed caching set up in less than 5 minutes.

Connecting an existing Nx workspace

If you already have an Nx workspace, run:

```
npx nx@latest connect
```

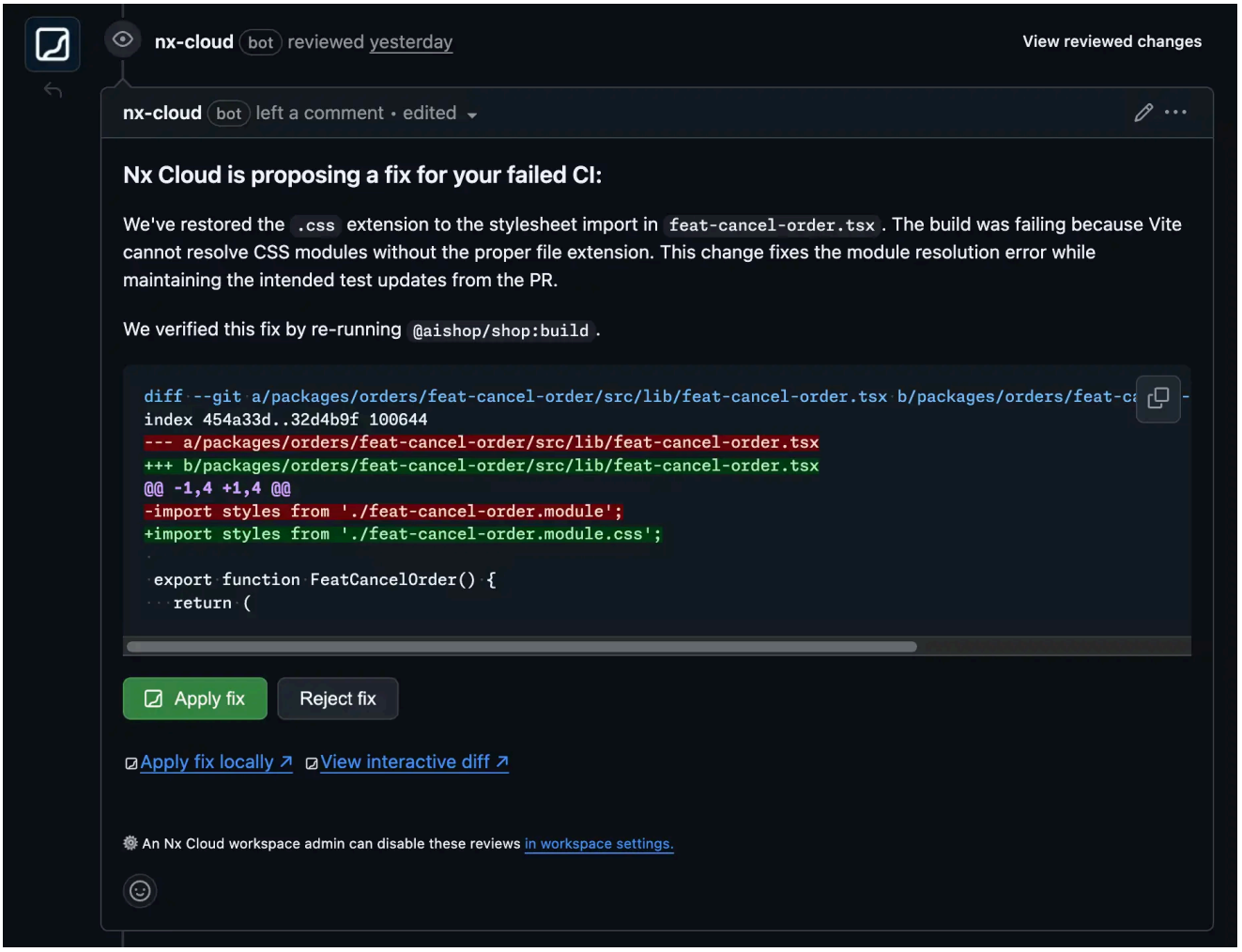
This guides you through setup and configures your workspace to use Nx Cloud. The connection is secure by default and requires no additional configuration.

> **Note:** While Nx Cloud is offered as a managed service, enterprise options are also available including on-premise installations and single-tenant deployments. These options give you complete control over where your data is stored and how it's managed. Visit nx.dev/enterprise (https://nx.dev/enterprise) for more information.

Self-Healing CI

Self-Healing CI is an AI-powered system that automatically detects, analyzes, and proposes fixes for CI failures. When a task fails, Nx Cloud starts an AI agent that examines the error logs, understands your codebase structure through the Nx project graph, and identifies the root cause. The agent then creates a fix and presents it via GitHub PR comments or directly in your editor through Nx Console.

This speeds up Time To Green by eliminating the delay between CI failure and fix. The AI agent handles failures automatically while you stay focused on building features.



How to configure Self-Healing CI

Self-Healing CI requires GitHub VCS integration and an Nx Cloud workspace connection. Add the following to your CI workflow:

```yaml
# .github/workflows/ci.yml
jobs:
  main:
    steps:
      - run: npx nx affected -t lint test build
      - run: npx nx fix-ci
        if: always()  # IMPORTANT: Must run even when previous steps fail
```

By default, proposed fixes require manual approval. You can enable auto-fixing for specific tasks:

```
- run: npx nx start-ci-run --auto-apply-fixes="*format*,*lint*" --no-
distribution
```

You can also specify which tasks should be eligible for self-healing:

```
# Only self-heal specific tasks
- run: npx nx start-ci-run --fix-tasks="*lint*" --no-distribution

# Exclude certain tasks
- run: npx nx start-ci-run --fix-tasks="!*deploy,!*test" --no-distribution
```

Note, the `--no-distribution` is just needed if you don't want to use Nx Agents. For more details and advanced configuration options, see the Self-Healing CI documentation (https://nx.dev/features/ci-features/self-healing-ci).

Remote Caching

Once your workspace is connected to Nx Cloud, remote caching (also known as Nx Replay) works automatically. Nx Cloud intelligently intercepts all `nx` commands and checks whether the task has been executed before. If it has, Nx Cloud restores the cached results: terminal output and all generated files, making the task complete instantly.

Here's a typical CI configuration:

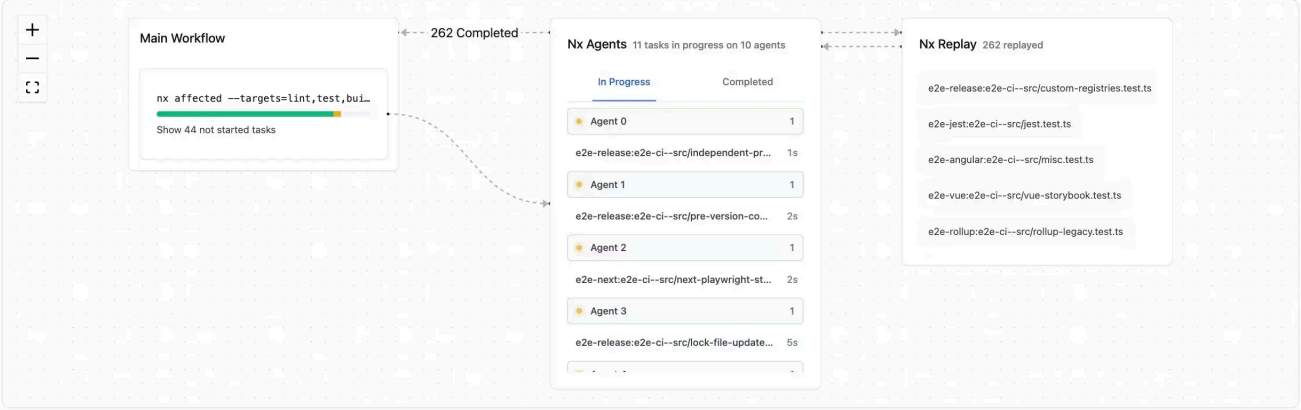
```
# .github/workflows/ci.yml
jobs:
  main:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: 'npm'

      - run: npm install
      - run: npx nx affected -t lint test build
```

That's it. No special configuration is needed. Nx Cloud automatically handles caching behind the scenes. When tasks run in your PR's initial CI run, subsequent runs reuse those cached results, making builds and tests complete instantly if already executed.

Remote caching forms the foundation for all other Nx Cloud features. It not only speeds up CI by avoiding redundant work but also acts as a transport mechanism for distributing work across multiple machines, which we'll explore in the next section.

Distributed Task Execution with Nx Agents

When remote caching isn't enough, you need to parallelize work across multiple machines. Many teams use matrix strategies to manually split work across jobs. This requires maintenance as your monorepo evolves and can't adapt to individual PRs: small and large changes use the same resources.



Nx Agents uses a task-based model where interchangeable agents dynamically pick up tasks based on historical run times and dependencies. Distribution adjusts automatically as your workspace changes. You can even scale agents based on PR size, using fewer machines for small changes and more for large ones.

Think of it this way: instead of assigning specific team members unique tasks (where work stops if someone gets stuck), you put all work in the middle of the room and let anyone pick up any task.

How to configure Nx Agents

Enabling distributed task execution requires adding a single line to your CI configuration:

```
# .github/workflows/ci.yml
jobs:
  main:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - run: npx nx-cloud start-ci-run --distribute-on="3 linux-medium-js"
--stop-agents-after="build"

      - uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: 'npm'

      - run: npm install
      - run: npx nx affected -t lint test build
```

The `start-ci-run` command tells Nx Cloud to create agents and distribute your tasks. After tasks complete on the agents, all artifacts and logs are played back to the main job as if the tasks ran locally. This means your existing post-processing steps (like uploading coverage reports or test results) continue to work without modification. The distribution is completely transparent to the rest of your CI pipeline.

Declarative and Reliable by Design

The configuration is declarative: you simply specify what needs to be accomplished (e.g., "use 3 linux-medium-js agents"), not how to split the work. Because agents are interchangeable and can pick up any task, the system dramatically improves pipeline reliability. If an agent is slow or fails, another agent automatically picks up the work. The system can tolerate failures and automatically rebalance work across available resources, making your CI more resilient than traditional approaches where each machine has a unique, predetermined job.

Atomizing

Even with distributed task execution, long-running tasks can create bottlenecks. Consider a Playwright E2E test suite that takes 10 minutes. This monolithic task can only run on one machine, becoming the slowest link in your CI pipeline regardless of how you distribute other tasks.

The Atomizer automatically splits large tasks into smaller units (typically one task per test file). That 10-minute E2E suite becomes five 2-minute tasks that run in parallel across different agents. The splitting happens automatically; you don't need to manually divide your tests.

How to enable the Atomizer

Ensure your workspace is connected to Nx Cloud, then add the appropriate plugin for your test framework:

```
nx add @nx/playwright # for Playwright

nx add @nx/cypress # for Cypress

nx add @nx/jest # for Jest
```

The Nx plugin automatically infers tasks for each spec file so they can run independently on CI. For an E2E project, you'll see:

- `e2e` - The base task for local development (runs all tests together)
- `e2e-ci` - The CI variant that orchestrates all split tasks
- `e2e-ci--src/e2e/app.spec.ts` - Individual task for one test file
- `e2e-ci--src/e2e/login.spec.ts` - Individual task for another test file

Using atomized tasks in CI

Update your CI configuration to use the `e2e-ci` variant:

```
# .github/workflows/ci.yml

- run: npx nx-cloud start-ci-run --distribute-on="3 linux-medium-js" --
stop-agents-after="e2e-ci"

- run: npx nx affected -t lint test build e2e-ci
```

For local development, continue using the base task (e.g., `nx e2e my-app-e2e`), which runs all tests together and is more efficient on a single machine. The `e2e-ci` variants are specifically designed for distributed execution in CI environments.

Flakiness Detection

Flaky tasks (those that fail intermittently in certain environments) are notoriously time-consuming to identify and debug. Behind the scenes, Nx Cloud automatically detects and handles these flaky tasks without you even noticing.

Here's how it works: Nx Cloud leverages its caching mechanism to track task outcomes. When a task runs, Nx Cloud computes a hash of all inputs (source code, dependencies, configuration). If a task ever fails with a particular set of inputs and then later succeeds with the exact same inputs, Nx Cloud identifies it as flaky. This detection is completely automatic and requires no configuration.

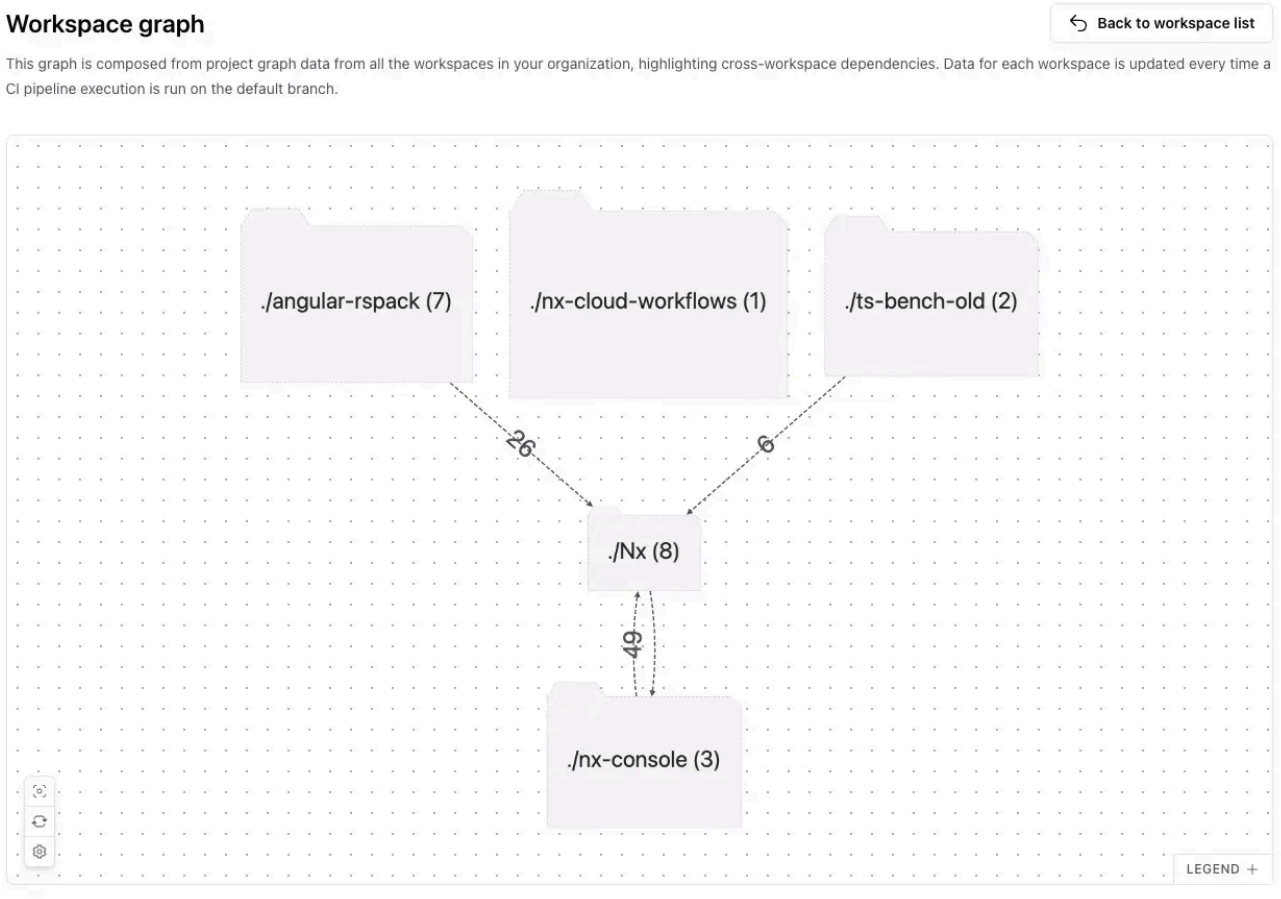
Once a task is flagged as flaky, Nx Cloud takes action. When using distributed task execution with Nx Agents and a flaky task fails, it's automatically sent to a different agent and retried.

You don't need to manually identify flaky tests or configure retry logic. Nx Cloud handles it automatically, and if a task hasn't exhibited flaky behavior for 2 weeks, the flaky flag is removed.

Flakiness detection is enabled by default when you use Nx Agents with no additional configuration, keeping your CI reliable and your PRs moving forward.

Beyond CI

While the features above focus on optimizing CI pipelines, Nx Cloud's Enterprise offering extends beyond CI to help large organizations manage multiple repositories. Organizations with separate monorepos or polyrepos can gain monorepo-like advantages (visibility, consistency, governance) without consolidating into a single repository.



Workspace Graph visualizes dependencies across all your repositories, even those not using Nx. This gives teams a complete view of how their code connects across the organization, making it easier to understand the impact of changes to shared libraries or design systems. **Conformance** enables platform teams to define and enforce organizational standards across all repositories through automated rules. **Custom Workflows** run scheduled compliance checks and security audits across selected repositories, with automated notifications for failures.

These features work with both Nx and non-Nx repositories through metadata-only workspaces, allowing you to gain organizational visibility without requiring teams to migrate their existing build systems. For more details, see the Polygraph documentation (https://nx.dev/docs/enterprise/polygraph) and the Conformance guide (https://nx.dev/blog/nx-cloud-conformance-automate-consistency).