devenv 2.0: A Fresh Interface to Nix

You type `nix develop`. The terminal fills with a single cryptic line: copying path, 47 of 312, 28.3 MiB, something something NAR. Five seconds. Ten. Is it evaluating? Downloading? Both? You change one line in your config and wait again. When it finally drops you into a shell, you switch to another branch and direnv hijacks your prompt for a rebuild you didn't ask for. You switch back, and Nix evaluates everything from scratch, even though nothing changed.

Nix gives you reproducibility that nothing else can match. But the moment to moment experience of *using* it has never matched the power underneath.

devenv 2.0 polishes Nix developer experience. Keeps the power, removes frictions. Here's what that looks like.

Interactive

To fully leverage what's going on in your development shell, we've made it fully interactive.

Terminal UI

Every devenv command now shows a live terminal interface. Instead of scrolling Nix build logs, you see structured progress: what Nix is evaluating, how many derivations need to be built and downloaded, task execution with dependency hierarchy, and error details that expand automatically on failure.

Native shell reloading

You save a file, direnv fires, your prompt locks up for thirty seconds while Nix rebuilds, and you sit there staring at a frozen terminal.

With native shell, you save a file, devenv rebuilds in the background, a status line at the bottom of your terminal shows progress, and you press `Ctrl+Alt+R` when you're ready to apply the new environment. Your shell stays interactive the entire time. If the rebuild fails, the error appears in the status line without disrupting your session.

An example empty environment with only `joe` package:

devenv.nix

```
{ pkgs, ... }:

{
  packages = [ pkgs.joe ];
}
```

Shell reloading is currently supported for bash, with fish and zsh coming soon (#2487).

direnv isn't needed anymore with `devenv shell` but it's still supported for automatic activation when switching directories; see the direnv integration

Native process manager

devenv 2.0 ships a built in Rust process manager that replaces process-compose.

Dependency ordering, restart policies, readiness probes (exec, HTTP, and systemd notify), systemd socket activation, watchdog heartbeats, file watching, and port allocation. All declarative, all in one place. Dependencies use `@ready` by default (wait for the probe to pass) or `@completed` (wait for the process to exit). You can freely mix processes and tasks in the same dependency chains.

devenv.nix

```
{ pkgs, config, ... }:

{
  services.postgres.enable = true;

  processes = {
    api = {
      exec = "${pkgs.python3}/bin/python -m http.server ${toString
config.processes.api.ports.http.value}";
      after = [ "devenv:processes:postgres" ];
      ports.http.allocate = 8080;
      ready.http.get = {
```

```
        port = config.processes.api.ports.http.value;
        path = "/";
      };
    };

    worker = {
      exec = ''
        echo "Worker connected to API on port ${toString
config.processes.api.ports.http.value}"
        exec sleep infinity
      '';
      after = [ "devenv:processes:api" ];
    };
  };
}
```

This foundation opens the door to a fully integrated development loop: running processes in the background directly from your shell session, and automatically restarting them when the shell reloads.

process-compose is still available via `process.manager.implementation = "process-compose"`. If something is missing from the native manager, let us know.

Instant

Run `devenv shell`. Wait a few seconds while Nix evaluates your configuration and builds what's needed. Now run it again.

This time it takes milliseconds.

Most of the performance gain comes from replacing multiple `nix` CLI invocations with a C FFI backend built on nix-bindings-rust. Instead of spawning five or more separate Nix processes per command, devenv 2.0 calls the Nix evaluator and store directly through the C API, evaluating one attribute at a time. This also gives us better error messages and real time progress in the TUI. We currently carry patches against Nix to extend the C FFI interface, but these are fully upstreamable and we plan to contribute them back. Thanks to Robert Hensing for creating nix-bindings-rust and making this possible.

This makes the evaluation cache incremental. Each evaluated attribute is cached individually along with the files and environment variables it touched. When you change one thing, only the attributes that depend on that change are re-evaluated; everything else is served from cache. A single evaluation now covers `devenv shell`, `devenv test`, `devenv build`, and every other command. When nothing changed (verified by content hash), the cached result is returned immediately without invoking Nix at all.

The cache invalidates when:

- Any source file that was read during evaluation changes
- Environment variables that were accessed during evaluation change
- The devenv version, system, or configuration options change

You can force a refresh with `--refresh-eval-cache` or disable caching with `--no-eval-cache`.

Polyrepo support

Most teams don't live in a single repo. You have a backend in one repository, a frontend in another, shared libraries in a third.

Referencing outputs from another devenv project was the third most upvoted issue. Now you can reference any option or output from another project through `inputs.<name>.devenv.config`:

devenv.yaml

```yaml
inputs:
  my-service:
    url: github:myorg/my-service
    flake: false
```

devenv.nix

```nix
{ inputs, ... }:
let
  my-service = inputs.my-service.devenv.config.outputs.my-service;
in {
  packages = [ my-service ];
  processes.my-service.exec = "${my-service}/bin/my-service";
}
```

This builds on the existing monorepo support and extends it to multi-repository workflows. See the polyrepo guide for full documentation.

Out of tree devenvs

Not every project has a `devenv.nix` checked in, and sometimes you want one configuration to serve multiple repositories. This was the fourth most upvoted issue. devenv 2.0 adds `--from`:

```
$ devenv shell --from github:myorg/devenv-configs?dir=rust-web
$ devenv shell --from path:../shared-config
```

Works with `devenv shell`, `devenv test`, and `devenv build`. Currently `--from` only works with projects that use `devenv.nix` alone; projects that also rely on `devenv.yaml` for extra inputs aren't supported yet.

For coding agents

A coding agent spins up your project in the background. It starts the dev server. Port 8080 is already taken by another agent running the same project. The process crashes. The agent

retries, hits the same port, crashes again.

Meanwhile, that agent has full read access to every `.env` file in your project. Your API keys, database credentials, third party tokens. It never asks permission. It never tells you what it read.

devenv 2.0 fixes both problems.

Automatic port allocation

Define named ports and devenv finds free ones automatically:

devenv.nix

```
{ config, ... }:

{
  processes.server = {
    ports.http.allocate = 8080;
    exec = "python -m http.server ${toString
config.processes.server.ports.http.value}";
  };
}
```

If port 8080 is taken, devenv tries 8081, 8082, and so on. Ports are held during evaluation to prevent races, then released just before the process starts. Use `devenv up --strict-ports` to fail instead of searching.

Secret isolation with SecretSpec

devenv 2.0 ships with SecretSpec 0.7.2 for declarative, provider-agnostic secrets management. Declare what secrets your project needs in `secretspec.toml`, and each developer provides them from their preferred backend: keyring, dotenv, 1Password, or environment variables.

Here's the thing: because password managers prompt for credentials before giving them out, secrets are never silently leaked to agents running in the background. This is a fundamental difference from `.env` files that any process can read.

Let's declare some secrets:

secretspec.toml

```
[project]
name = "myapp"
revision = "1.0"
```

```
[profiles.default]
DATABASE_URL = { description = "PostgreSQL connection string", required = true }
STRIPE_KEY = { description = "Stripe API secret key", required = true }
SENTRY_DSN = { description = "Sentry error tracking DSN", required = false }
```

And see how devenv asks for them and starts:

MCP server

The devenv MCP server exposes package and option search over stdio and HTTP:

```
$ devenv mcp --http 8080
```

We host a public instance at `mcp.devenv.sh` that any MCP compatible tool can query without needing a local devenv installation.

devenv.new is a coding agent powered by the same package and option search that generates `devenv.nix` files for you.

And more

Language servers for your code. Most language modules now have `lsp.enable` and `lsp.package` options, giving you a language server for your project out of the box.

Language server for `devenv.nix`. Get completion and diagnostics while editing your devenv configuration:

```
$ devenv lsp
```

`devenv eval`. Evaluate any attribute in `devenv.nix` and return JSON:

```
$ devenv eval languages.rust.channel services.postgres.enable
{
  "languages.rust.channel": "stable",
  "services.postgres.enable": true
}
```

`devenv build` returns JSON. `devenv build` now outputs structured JSON mapping attribute names to store paths:

```
$ devenv build languages.rust.channel services.postgres.enable
{
  "languages.rust.channel": "/nix/store/...-stable",
  "services.postgres.enable": "/nix/store/...-postgresql-16.6"
}
```

NIXPKGS_CONFIG. devenv now sets a global `NIXPKGS_CONFIG` environment variable, ensuring that nixpkgs configuration (like `allowUnfree`, CUDA settings) is consistently applied across all Nix operations within the environment.

Breaking changes

For a step by step upgrade guide, see Migrating to devenv 2.0.

- The `git-hooks` input is no longer included by default. If you use `git-hooks.hooks`, add it to your `devenv.yaml`.
- `devenv container --copy <name>` has been removed. Use `devenv container copy <name>`.
- `devenv build` now outputs JSON instead of plain store paths. Update any scripts that parse the output.
- The native process manager is now the default. Set `process.manager.implementation = "process-compose"` if you need the old behavior.

Deprecation of devenv 0.x

devenv 0.x is now deprecated. Support will be dropped entirely in devenv 3.

Final words

Over the next few weeks we will be focused on fixing bugs and stabilizing the release. If you run into any issues, please open a report and we will prioritize it. Join the devenv Discord community to share feedback!

Domen

---

devenv

Fast, Declarative, Reproducible, and Composable Developer Environments using Nix

Documentation

Getting Started

Examples

Languages

Services

Options Reference

Subscribe to updates

name@example.com Subscribe
