ChatGPT

Monolithic vs. Modular Nix Flakes – Best Practices & WSL Use Cases

Monolithic Flake Use Cases (Has it been done?)

Yes – many Nix projects start as **single "monolithic" flakes** that bundle everything. For example, one developer described how they began with a single `flake.nix` to install an entire simulation framework plus dev tools and tests. This **all-in-one flake** worked fine during early development when everything lived in one repo [1]. The convenience of a single flake is appealing because:

- **One-Stop Configuration:** Everything (packages, devShells, NixOS config, etc.) is defined in one file/repo, so you don't have to jump between multiple flakes or files.
- **Simple Usage:** Users run one flake for the full environment (e.g. `nix develop` or `nixos-rebuild` on that flake) and get all needed components.
- **Less Overhead (initially):** No need to set up multiple repositories or Nix inputs – a monolithic flake can vendor all pieces in one place.

Why do this? Often projects begin monolithic for *convenience and speed*. If you're the only consumer of the flake, keeping it all together reduces cognitive load. In the example above, it served "well in all manner of use cases" during initial development [1]. Small personal configs or single-purpose projects (like a dev environment for one language) are commonly one flake.

However, monolithic flakes can become **"god objects"** – large and unwieldy as they grow. Your current flake (~3000 lines) is a good example: it contains **packages**, **overlays**, **devShells**, **Docker/Podman setup**, and **NixOS modules** all in one. This violates the *single responsibility principle*, making the flake hard to maintain. As one user put it, *"the baggage from having used the monolithic flake for a while... is preventing me from seeing decent solutions."* [2] In practice, once a flake expands to support multiple distinct use-cases or deliverables, it's a sign to consider modularizing.

Modularization vs. Multiple Flakes (Speed and Best Practices)

Is it best practice to split into multiple flakes? Generally, large Nix configurations benefit from modular structure – but that can mean **splitting one flake into multiple files/modules** *or* even **splitting into multiple flake projects**. The goal is to improve maintainability, clarity, and sometimes performance.

Forms of modularization:

- *Within one flake:* Break up the single `flake.nix` into separate Nix files (e.g. `packages.nix`, `devshells.nix`, `overlays.nix`, `nixosModules.nix`) and import them in `flake.nix`. This keeps one flake, but code is organized by concern.

- *Multiple flake outputs:* Use one flake but define multiple outputs for different use cases (e.g. several `devShells` for different scenarios, multiple `packages` or `apps`). Flakes support this natively (multiple `devShells.<name>` or `packages.<name>` outputs) [3] [4].
- *Truly multiple flakes:* Factor out parts of your project into independent flakes (perhaps even separate repos) and make them inputs to a top-level flake. For example, you could have a `ros2-packages` flake that provides ROS2 derivations, and a separate flake for your NixOS configuration that uses those as inputs.

Does splitting slow down Nix? – Not inherently. In fact, splitting can **speed up evaluation** in large projects. Nix flakes evaluate in a hermetic way with caching of evaluation results. If you have multiple flakes, each is cached by its content hash, so re-evaluating one doesn't redo the others unless they changed. In a massive monorepo scenario, developers found that focusing Nix on a sub-flake dramatically improved performance (90s eval down to 8s) [5]. Essentially, a smaller flake context means less work per evaluation. Another NixOS community member noted that finer-grained derivations can improve build caching, but with diminishing returns if not done thoughtfully [6]. The key is that **Nix's eval cache** makes multiple flakes feasible without heavy overhead, and in some cases "copying the whole tree is definitely slower" than evaluating a focused flake [5].

That said, using many small flakes *can* introduce some overhead in complexity: - You'll need to manage multiple repositories or at least multiple flake inputs, which is extra work. - Each flake input (if remote) is another fetch. If all are in one repo as path inputs, this is less of an issue. - There's a balance: don't split *every tiny thing* into its own flake unless they are truly reused independently.

Community best practices favor **modularity** for maintainability: - It's easier to test or build components in isolation. - Collaboration is smoother (different team members can own different files or flakes). - Code reuse is improved (e.g. you could reuse your `ROS2` package definitions in another project by moving them to a flake or making them an overlay).

Several tools exist to help structure large flakes. For example, *flake.parts* allows writing flakes in a modular, NixOS module-like style [7], and projects like **drv-parts** or **haumea** provide patterns for splitting Nix code into directories [7]. These indicate a common pain point: large monolithic flakes are hard to manage, so community solutions focus on breaking them into pieces.

Bottom line: Splitting your big flake into logically separate parts is considered good practice. Whether that means multiple files in one flake or multiple flake projects depends on your needs: - If all components are only used together, one flake with internal modules is simpler. - If some components (e.g. your custom packages or devShell) could be useful standalone, you might publish them as independent flakes.

Crucially, doing this **should not significantly slow down builds** – Nix will only (re)evaluate what's necessary. If anything, you may see *faster* command times on a leaner flake, as noted above [5].

Nix Flakes for WSL Distro Creation

Yes, you can absolutely use Nix (and flakes) to create a Windows Subsystem for Linux distro. **Use case:** Providing a NixOS-based environment on Windows without a VM. The prime example is **NixOS-WSL**, an official community project that packages NixOS for WSL2. This project uses NixOS modules and a flake to build a special `.wsl` image (essentially a tarball with a NixOS rootfs and metadata). Users can download this and import it as a WSL distro. For example, the quick start is simply:

1. Enable WSL and download the `nixos.wsl` release file.
2. Double-click it (or run `wsl --import` with that file) to install NixOS on WSL [8].
3. Launch WSL with the new NixOS distribution (`wsl -d NixOS`) and you have a fully declarative NixOS running on Windows [9] [10].

This means your entire NixOS configuration (including flakes, Home Manager, etc.) can run under WSL2. In fact, one engineer reported using NixOS-on-WSL to get "**Flakes, Home Manager, and full declarative configs**" seamlessly inside Windows [11]. The `.wsl` image is built using Nix – essentially `nixos-rebuild` produces an output that can be imported by WSL.

Repositories & examples: Aside from the main nix-community/NixOS-WSL repo [8] (2.7k stars, actively maintained), there are user configurations demonstrating flakes on WSL. For instance: - **kenlasko/nixos-wsl** – a personal config repository for NixOS on Windows 11 WSL that uses flakes and Home Manager [12] [13]. It outlines how to rebuild the OS with a flake and even automate WSL setup. - Others have shared their flake configs for WSL on forums (e.g. a user on Reddit shared a flake for WSL and a blog post about setting up NixOS on WSL with existing configs [14] [15]).

These show that using a flake to create or manage a WSL distro is not only possible, but fairly common for NixOS users: - You can **generate a WSL-compatible NixOS image** with `nixos-rebuild build --flake ...` and then import it in WSL. - Tools like **nixos-generators** can output various image formats (including installers, ISO, VM images; WSL is essentially an ext4 filesystem image which NixOS-WSL provides). - The benefit is consistency: your WSL instance is built from the same flake that might also deploy bare-metal NixOS or containers, etc., ensuring parity across environments.

In summary, **Nix flakes can indeed produce WSL distros**. The use case is typically for developers who want a reproducible dev environment on Windows without leaving Windows. The existence of projects like NixOS-WSL [8] is strong evidence of the approach. If you need to provision a WSL rootfs from scratch via Nix, you would create a NixOS configuration (flake-based) and build a WSL image from it – many have done this successfully.

Audit Recommendation – Single vs Multiple Flakes Path Forward

Your audit suggested refactoring the single giant flake into "multiple flakes". The core idea is to **tame complexity**. Based on current best practices and the analysis above, here's how you might proceed:

- **Start with Modularization:** Break the monolith into multiple files or modules **within one flake**. For example, have separate files for ROS2 package definitions, general packages, devShells, NixOS modules, etc., and have `flake.nix` import them. This will immediately make the codebase easier to navigate and maintain, without introducing the overhead of managing multiple flake projects.
- **Leverage Flake Outputs:** Use multiple outputs in your flake for different concerns. You likely already have several devShells (default, full, minimal, CI). Make sure each is clearly defined. You can also provide multiple `packages` outputs if needed (e.g. `packages.x86_64-linux.rosWrapper` vs `packages.x86_64-linux.tools`).

- **Consider Separate Flakes if Reuse or Distribution is needed:** If parts of your project could live on their own (say an overlay/flake for ROS2 that others could use independently, or a flake template for new projects), then splitting those into distinct flakes is worthwhile. In the discourse example, the author wanted to provide: (1) the library itself as a package, (2) a dev env for developing that library, (3) a dev env for *using* that library in other projects, etc. [16]. They reasoned that some of those might need to be separate flakes to avoid interference [17]. Similarly, ask if your project has deliverables that benefit from being isolated. If not – if it's all one product – separate flakes might be overkill.

- **Performance Impact:** Don't worry about flakes slowing things down. As discussed, splitting the flake *should not* hurt build performance in any notable way. Nix will cache evaluations per flake. In fact, if you structure it such that, say, your ROS2 package set is one flake input (maybe pointing to a stable ROS overlay), and your config is another, you might get better incremental speeds since changes in one don't force reevaluation of the other.

- **Developer Workflow:** Using multiple flakes means you'll reference them via inputs. For local development, you can use `path:` inputs to point one flake at another on disk (helpful if you split into multiple repos or directories). This is how monorepos use sub-flakes to improve dev speed [5]. It adds a bit of complexity (you have to update inputs or run `nix flake update` if using Git), but it's manageable and often worth it for large systems.

Is it the best path forward? Given the size and scope of your project, **yes**, adopting a more modular flake architecture is likely the right move. It aligns with what many Nix users do once a configuration grows beyond a few hundred lines. You'll gain in maintainability and clarity more than you'll lose in any minor tooling overhead. As a middle ground, you might first reorganize into one flake with multiple files (for sanity), and only split into multiple separate flakes if you identify clear boundaries where independent versioning makes sense.

In conclusion, the audit's suggestion is well-founded: a single 3000-line flake is hard to maintain. Breaking it into multiple pieces (conceptually and/or into distinct flakes) follows Nix community best practices and does not hurt performance. In fact, it can improve your development speed and pave the way for easier scaling (and possible reuse) of your Nix code. Consider this an investment in the **long-term health** of your project's Nix infrastructure – modular design will pay off as the project evolves.

Sources:

- NixOS Discourse – *"Breaking up a monolithic flake"* (developer discussion on splitting a large flake) [1] [2]
- NixOS Discourse – *Monorepo Flake Performance Tips* (noting 90s eval reduced to 8s by focusing on sub-flake) [5]
- NixOS Discourse – *"Is there advantage to multiple flake packages?"* (discussion of when to split derivations/outputs) [6]
- NixOS Discourse – *Suggestions for flake modularization tools (flake.parts, etc.)* [7]
- **NixOS-WSL** project (GitHub) – Running NixOS on Windows Subsystem for Linux [8]
- **kenlasko/nixos-wsl** (GitHub) – Example of using flakes for a WSL-based NixOS dev environment [8]
- Reddit r/NixOS – user experiences with flakes in WSL and building WSL images [14].

---

① ② ③ ④ ⑦ ⑯ ⑰ Breaking up a monolithic flake - Help - NixOS Discourse
https://discourse.nixos.org/t/breaking-up-a-monolithic-flake/30475

⑤ Improve Flake Performance in absurdly bloated Monorepo - Help - NixOS Discourse
https://discourse.nixos.org/t/improve-flake-performance-in-absurdly-bloated-monorepo/21282

⑥ Is there advantages to multiple Nix Flake Packages? - Guides - NixOS Discourse
https://discourse.nixos.org/t/is-there-advantages-to-multiple-nix-flake-packages/14344

⑧ ⑫ ⑬ GitHub - kenlasko/nixos-wsl: Configuration repository for NixOS running in Windows 11 WSL
https://github.com/kenlasko/nixos-wsl

⑨ ⑩ ⑪ NixOS on WSL: The Best of Both Worlds | by Piyush Kumar Singh | Medium
https://medium.com/@piyushkumarsingh.nmims/nixos-on-wsl-the-best-of-both-worlds-bdcaba54ee58

⑭ ⑮ Using current flake setup with WSL2 : r/NixOS
https://www.reddit.com/r/NixOS/comments/12po37r/using_current_flake_setup_with_wsl2/