Flakes feature freeze

The core team has decided to freeze the Flake feature set and semantics at its current point, excluding bugfixes. No new features will be added to Flakes for the time being, including (but not limited to) new input types, new features for existing fetchers and input types, or flake-centered changes to the evaluator.

Our long-term plan is to move Flakes out of the Lix code base into a separate plugin, all while cleaning up the structural foundations and building a new platform for any third party to build flake-equivalent functionality upon. Until then, we intend to continue our support of Flakes in their current form. They will remain an experimental feature and explicitly *not* be stabilized.

The main reason for the freeze is that the Flakes implementation code is *extremely* brittle, causing significant maintenance overhead. With our stated goal of refactoring the Lix code base and moving Flakes out of it, any changes to Flakes themselves would make them a moving target, and greatly increase the risk of breakage or regressions. Moreover, Flakes provide virtually no versioning mechanism to safely introduce major changes, and while Flakes are *stated* to be "experimental", their current wide adoption thwarts our ability to rapidly iterate.

Design issues of Flakes

As many other people have exhaustively written about, Flakes are sprinkled with design deficiencies. Many of those issues are related to versioning and future-proofing, making it significantly harder to fix Flakes in an iterative way without causing disruption. The list presented here is not meant to be exhaustive and represents the main issues motivating the freeze of Flakes in Lix.

- The Flake schema is effectively unversioned and thus impossible to evolve without breaking existing users. This necessitates extensive compatibility checks in the implementation, most of which very quickly becomes effectively untested and thus a regression hazard.
- The Flake lock file schema is versioned, but its version never changes even when its semantics change. Adding new input types or how existing input types are interpreted should imply a change to the lock file schema to ensure that older version do not misinterpret a lock file or fail with cryptic errors.
  - The design of flake locking is unsound: It compares the lock entry to the current lock entry, even if the thing in the lock file is fetchable but just not what *this* nix implementation would have generated for the same original input.
  - This means that *any* divergence in libfetchers input format or lock file format causes incompatibility between nix implementations. There are already divergence bugs, e.g. https://git.lix.systems/lix-project/lix/issues/841, https://git.lix.systems/lix-project/lix/issues/520.
  - In the current system, the only way to fully prevent all compatibility bugs is to have all implementation agree to a common specification. For a multitude of reasons, this

is not going to happen. There is a (stalled) effort of standardizing at least `fetchTree` here: https://github.com/nix-community/fetchTree-spec.

- Using the Nix language to specify Flake inputs requires very invasive changes to the evaluator to ensure that Flake input definitions are "trivial", i.e. pure attribute sets that do not contain computations. Instead, inputs should be defined in a format that does not require evaluating code, much like the lock file.
- The Flake interface, especially the devshells, are tied very strongly to nixpkgs. while `nix-shell` has always been tied to nixpkgs to an extent (`nix-shell -p thing` internally runs `{...}@args: with import <nixpkgs> args; (pkgs.runCommandCC or pkgs.runCommand) "shell" { buildInputs = [ (thing) ]; } ""`), this binding was not so strong that it could not be used freely by non-nixpkgs package sources. `nix develop` on the other hand hard-codes nixpkgs-specific internals in many places and is all but useless on derivations that do not use nixpkgs' `stdenv`.
- Cross-compilation support is completely absent and cannot be added without changing the Flake schema, which is (as noted above) all but impossible without introducing new attributes (which would be admonished by older `nix flake check`) or outright breakage.
- Flakes do not compose well. Reusing an expensive nixpkgs instantiation can be impossible because flakes do not allow passing parameters to the implicit instantiation created by `legacyPackages`, and even when reusing an instantiation is possible the final graph of a flake-built package may be inconsistent if multiple versions of nixpkgs were used in the flake input graph.
- Using the same source in multiple versions is not only easy to do by accident, it is hard to *not* do even with intent. The mechanism for this, flake input follows, are spooky action at a distance that allow a flake to override inputs of its dependencies even if those dependencies do not expect this, possibly causing silent breakage of packages.
  - c.f. https://github.com/mercuryTechnologies/unused-flake-input

---

Revision #1
Created 2025-10-05 19:25:38 UTC by piegames
Updated 2026-02-01 13:14:00 UTC by piegames