# Prepare for the Talk:

## Install Nix!

Nix runs perfectly on any Linux Distro
(and on macOS and WSL)

Everybody can install Nix and actively participate :)

- Visit https://determinate.systems/nix-installer/
- Follow instructions for your platform
- You can easily uninstall later

You can also use the NixOS Live ISO

- Visit https://nixos.org/download
  - NixOS : the Linux distribution (Choose GNOME / Plasma / Minimal)

# Nix/NixOS

## Reproducible Builds, Functional Packages

Robin Finkelmann

OpenColloq Informatik @uniwue

2025-05-21 18:00:00

# Outline

# 1. Introduction

# 1.1 Motivation

Imagine…

- You use a specific version of a software (e.g. KiCAD v8)
- You need to use a different version too (e.g. a KiCAD v9 project)
- But you don’t want to update your old projects yet

# 1.1 Motivation

Imagine…

- You want to switch your Linux Distro's Desktop / WM
- (Or, even worse, your audio backend)

# 1.1 Motivation

Imagine…

- Your laptop SSD just broke.
- You have a backup of your home dir
- Maybe even a full backup image
- But now you need to install to a smaller SSD…
- Or reinstall and reconfigure everything…

# 1.1 Motivation

Imagine...

- You wish to try out a piece of software
- You install it
- You try it out
- You forget it
- It floats around forever, possibly breaking future system updates

# 1.1 Motivation

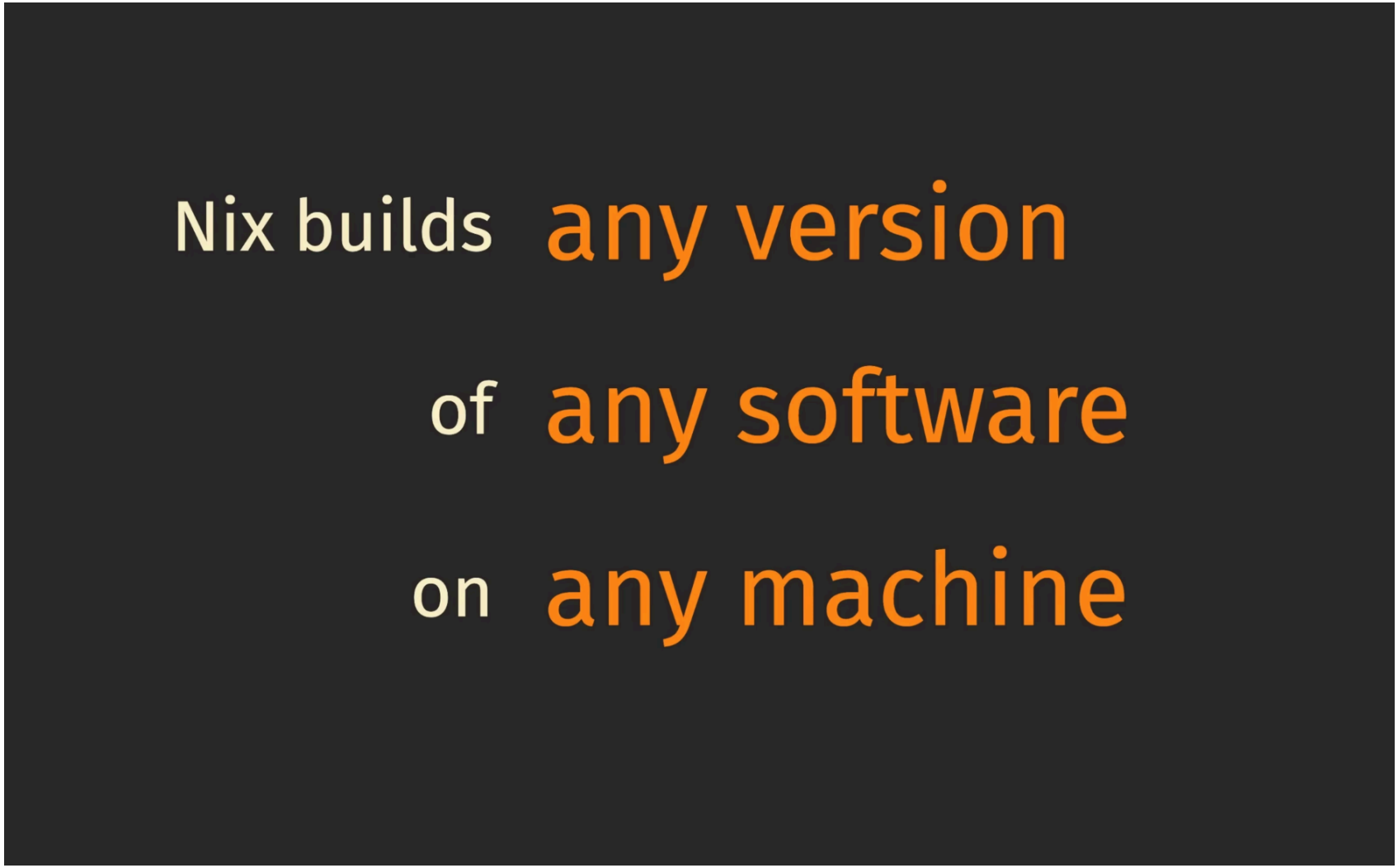
Imagine…

- You have a specific problem
- You stumble upon a Git Repo that solves your problem
- It is many years old
- It uses old python packages
- It wants to install system-wide pip packages
- It has a ~~cursed~~ **special** install script that copies files to /usr/bin
- The README is written for Debian, but not even Debian allows this anymore (without tweaking)

# 1.2 Why Nix?

- Developing, Building, Deploying
- Reproducible, versioned Builds (write once, deploy anywhere)
- Functional Programming Language
- 1 Language for everything
- Install multiple Versions of the same Package
- Distro-Independent, even on WSL and Darwin
- Largest Package Repo of all Distros

# 1.2 Why Nix?



# 1.3 Why NixOS?

- Declarative Configuration
- Atomic Upgrades
- Rolling/Unstable and 6 Month Staged/Stable

# 1.4 Why not Nix?

- ***TERRIBLE*** Documentation
  - Many different Formats: Nix, Nix Commands, Flakes, Nixpks, NixOS
  - Multiple different Tools for the “same” job
  - Many experimental Features are “the norm”
  - Many community projects are “the norm”
  - Aims to build everything (Eierlegende Wollmilchsau)
- Not FHS-compliant
  - Most dynamic Binaries will not work out-of-the-box
- Not 100% stable and secure
- No LTS

# 1.5 Nix vs Nixpkgs vs NixOS

- Nix
  - Language / Expressions
    - Functional Language
    - Features for Building Packages (Derivation, Realisation)
    - Builtins
  - Commands
    - Nix Store
    - Nix Profiles
    - Nix Commands
    - Shells
    - Building
  - Available on all Linux Distros (and macOS)

# 1.5 Nix vs Nixpkgs vs NixOS

- Nixpkgs
  - Package and Option Collection for Nix / NixOS
  - Includes Wrappers for most common Programming Languages
  - Stdenv, Lib
- NixOS
  - Linux Distro built upon Nix
  - Packages and Options from Nixpkgs
  - Manages System through Options and Modules
- Home Manager
  - Community Project
  - Manages your Dotfiles and User Environment declaratively
  - Also available on all Linux Distros (and macOS)

# 1.6 Stable vs Experimental Nix

## Stable Nix Commands

- `nix-<command>`, e.g. `nix-shell`
- Some are somewhat outdated
- Still occasionally used

## Experimental Nix Commands

- `nix <>`, e.g. `nix shell`
- More up-to-date
- Often better
- Widely used
- No feature parity

# 1.7 Ad-hoc vs Declarative Nix

- Ad-hoc refers to using Nix ‘on the fly’, i.e. in a shell environment
  - Ad-hoc use is optimal for experimenting
- Declarative refers to writing files that specify your actions
  - Declarative use is optimal for reuse

# 1.8 Installation of Nix / Lix

Nix runs perfectly on any Linux Distro (and on macOS and WSL)

Everybody can install Nix and actively participate :)

I recommend the Nix Installer from Determinate Systems

- More deterministic, easy uninstall, Experimental Features enabled
- Visit https://determinate.systems/nix-installer/

Or if you ~~are a bit edgy~~ want a faster community fork of Nix:

- Visit https://lix.systems/install/

You can also use the NixOS Live ISO (and then directly install NixOS)

- Visit https://nixos.org/download
  - NixOS : the Linux distribution (Choose GNOME / Plasma / Minimal)

# 2. Nix: The Package Manager

# 2.1 nix-shell

Warning: `nix-shell` is a stable nix command.

Start a shell with GNU Hello in your environment.

```
1 nix-shell -p hello
```
Shell

You are now dropped in a Bash Shell and can run hello.

```
1 hello
```
Shell

CTRL+D quits the shell again.

You can also specify a command to run (instead of interactive bash):

```
1 nix-shell -p hello --run hello
```
Shell

For some more fun, I recommend this little script:

```
1 nix-shell -p lolcat cowsay --run \
2 "cowsay Hello, Nix! | lolcat"
```
Shell

# 2.2 nix shell

Warning: `nix shell` is an experimental nix command.

Start a shell with GNU Hello in your environment.

```
1 nix shell nixpkgs#hello
```
Shell

This seems inconvenient for just using nixpkgs, but the syntax is more versatile!

Like using Repos (Flakes, more later):

```
1 nix shell github:nixos/nixpkgs#hello
2 hello
```
Shell

# 2.3 shell.nix

Warning: `shell.nix` is a stable nix convention.

Write shells declaratively!

Nix

```nix
{ pkgs ? import <nixpkgs> {} }:
pkgs.mkShell {
  packages = [ pkgs.lolcat pkgs.cowsay ];
  inputsFrom = [];
  shellHook = ''
    echo Hello Shell!
  '';
}
```

# 2.4 nix run

Warning: `nix run` is an experimental nix command.

And to run directly:

```
1 nix run github:nixos/nixpkgs#hello
```
Shell

Add branch:

```
1 nix shell github:nixos/nixpkgs/nixos-24.11#hello
```
Shell

More infos about this in later Chapter about Flakes!

# 2.5 direnv

Automatically loads a shell env when entering a directory

- Install nix-direnv via Home Manager or NixOS
- Create a file called `.envrc` in the directory
- Create a `shell.nix` in the directory

-Used with default.nix, shell.nix, or a Flake (more details later)

```
1 use nix
```
envrc

```
1 use flake
```
envrc

https://github.com/nix-community/nix-direnv

# 2.6 Install packages: nix-env

Warning: Usage not encouraged!

Consider using Shells/Direnv, Home Manager or NixOS instead!

- Installs packages into a profile in user's home directory

```shell
nix-env -qaP fastfetch  # search for package
nix-env -iA fastfetch   # install package
nix-env -e fastfetch    # remove package
nix-env -uA fastfetch   # upgrade a package
nix-env -u              # upgrade all packages
```

https://howarddo2208.github.io/posts/02-nix-beginner-guide/

# 2.7 Install packages: Home Manager

- Simply add packages to your user's environment with Home Manager
- Home Manager is also available independent of NixOS
- More about Home Manager later

```nix
home.packages = [ pkgs.fastfetch ];
```

Nix

# 2.8 Install Packages: NixOS

- Simply add packages to your system environment with NixOS
- Services must be installed using options
  - More about that later

```
1    environment.systemPackages = [ pkgs.fastfetch ];    Nix
```

# 3. Inner Workings

# 3.1 Inspiration

Strongly inspired by:

- https://youtu.be/5D3nUU1OVx8

More info at

- https://nix.dev/tutorials/nix-language.html
- https://nixos.org/guides/nix-pills/

# 3.2 Language

First things first: Nix can be interactively evaluated using:

```
1 nix repl
```

Shell

# 3.2 Language

Comments:

```nix
# Comment
/*
  Comment
*/
```

# 3.2 Language

Numbers:

```nix
123     # Integer
123.4   # Float
```

# 3.2 Language

Strings:

```nix
"abcdef"

''abc
def''

"${pkgs.echo}/bin/echo Hello world!"
```

# 3.2 Language

List:

```nix
[ 123 "abc" ]
```

# 3.2 Language

Attribute Set:

```nix
{
  name = "Nix";
  fun = 42;
}
```

# 3.2 Language

Functions:

```nix
# Defining
# f = ...
a: a + 1
# g = ...
{x, y ? 0}: x + y

# Calling (pretending 'f' and 'g' exists)
f 2
g {1,2}
```

# 3.2 Language

Let in:

```nix
let
  g = {x, y ? 0}: x + y;
  f = a: g a;
in
  f 20
```

# 3.2 Language

With (discouraged):

```nix
let
  attrs = {a = 1; b = 2; c = 3;};
in
  with attrs;
  a + b + c
```

# 3.2 Language

Inherit:

Nix

```nix
let
  attrs = {a = 1; b = 2; c = 3;};
  inherit (attrs) a b c;
in
  a + b + c
```

# 3.2 Language

Builtins: e.g. `builtins.attrNames` and `buildins.functionArgs`

# 3.2 Language

Imports:

```nix
import ./a.nix
import "./b.nix"
```

# 3.3 Nix Store

- Content-addressable, immutable
- `/nix/store/<hash>-<name>-<version>/...`
- Every nix derivation and realization lives here
- Everything can be added:
  - `nix store add-file`
  - `nix store add-path`
- Everything is symlinked into the store
  - Results, Profiles, Binaries, Libraries, …

# 3.4 Derivations

- Declarative building instructions
- Native Nix language feature
- Produces intermediate representation `.drv`

# 3.4 Derivations

File called `my-derivation.nix`

Nix

```nix
derivation {
  name = "my-program";
  system = "x86_64-linux";
  builder = "/bin/bash";
  src = ./main.c;
  args = [ "-c" ''
    /usr/bin/clang $src
  ''];
}
```

# 3.4 Derivations

```shell
nix-instantiate my-derivation.nix
nix derivation show <store-path>

nix-store --realize <store-path>
```

# 3.4 Derivations

File called `my-derivation.nix`

Nix

```nix
derivation {
  name = "my-program";
  system = "x86_64-linux";
  builder = "/bin/bash";
  src = ./main.c;
  args = [ "-c" ''
    /usr/bin/clang $src -o $out
  ''];
}
```

# 3.4 Derivations

```shell
nix-build my-derivation.nix
```

Shell

Depending on environment, either succeeds or fails.

Heavily impure!

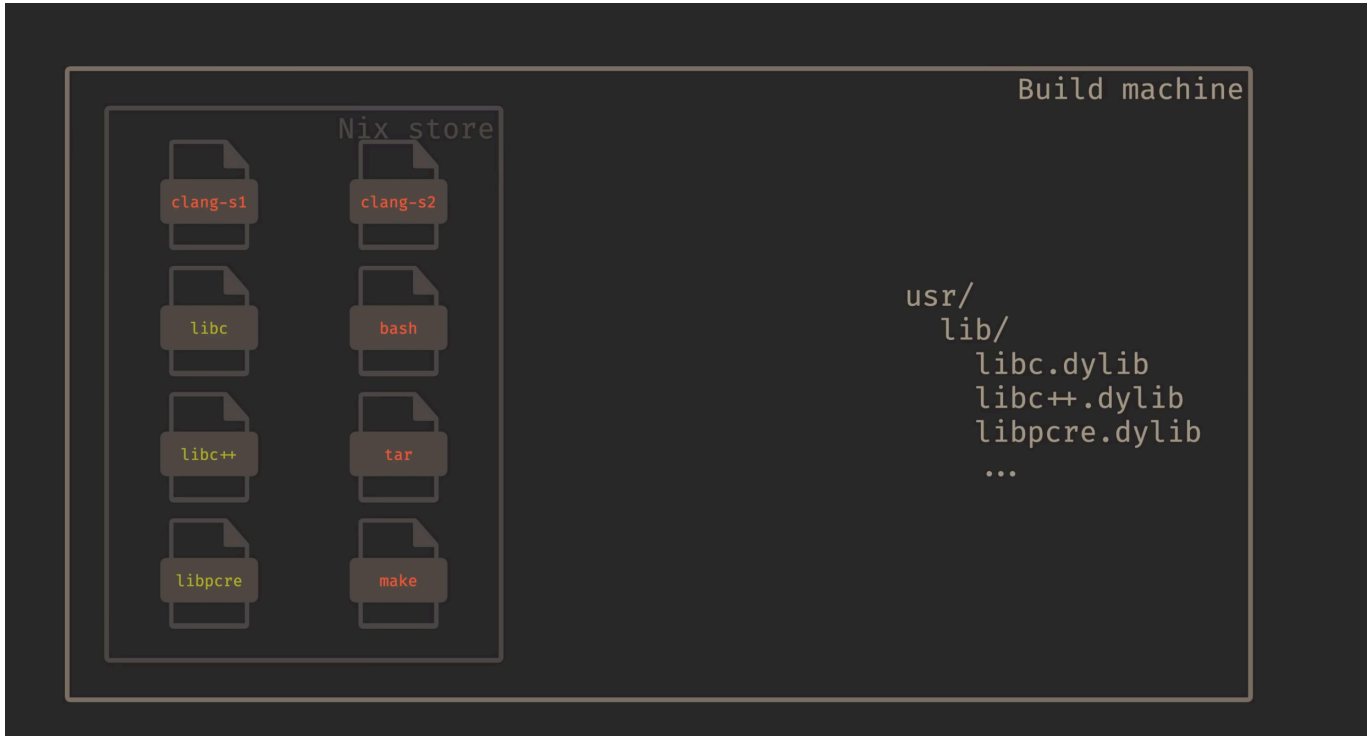



Figure 1: Proper bootstrapping of all build tools into Nix

This is where Nixpkgs and Stdenv come into play!

# 3.4 Derivations

Nix

```nix
{
  pkgs ? import (fetchTarball "https://github.com/NixOS/nixpkgs/archive/06278c77b5d162e62df170fec307e83f1812d94b.tar.gz") {}
}:
derivation {
  name = "my-program";
  system = "x86_64-linux";
  builder = "${pkgs.bash}/bin/bash";
  src = ./main.c;
  args = [ "-c" ''
```

# 3.4 Derivations

```
10      ${pkgs.clang}/bin/clang $src -o $out
11    ''];
12  }
```

# 3.4 Derivations

Nix

```nix
{
  pkgs ? import (fetchTarball "https://github.com/NixOS/nixpkgs/archive/06278c77b5d162e62df170fec307e83f1812d94b.tar.gz") {}
}:
pkgs.stdenv.mkDerivation {
  name = "my-program";
  system = "x86_64-linux";
  nativeBuildInputs = [];   # Build-Time
  buildInputs = [];         # Runtime
  dontUnpack = true;
```

```
10    buildPhase = ''
11      clang $src -o my-program
12    '';
13    installPhase = ''
14      mkdir -p $out/bin
15      cp my-program $out/bin$
16    '';
17  }
```

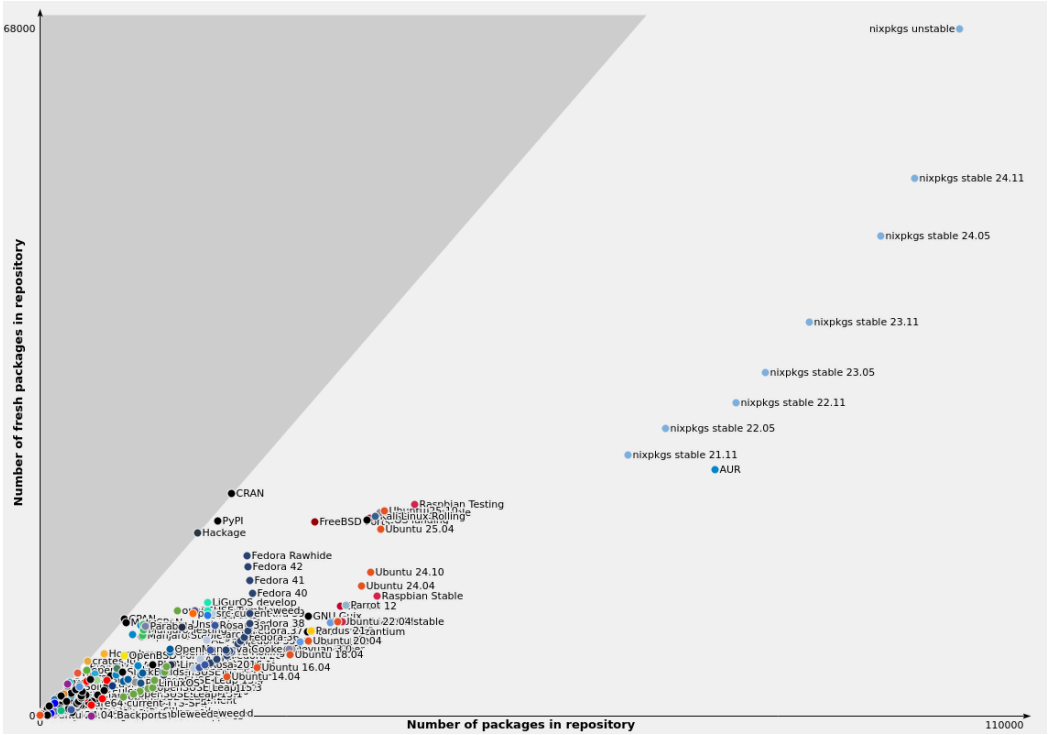



Figure 2: Linux Distro's Repo's Package Counts

- https://repology.org/repositories/graphs

# 3.5 Nixpkgs

- https://github.com/NixOS/nixpkgs

# 3.6 Channels

Warning: Channels are a stable nix feature.

- Provide an atomic version of nixpkgs

```shell
nix-channel --list
nix-channel --add https://nixos.org/channels/nixpkgs-unstable nixpkgs
nix-channel --add https://nixos.org/channels/nixos-unstable nixos
nix-channel --add https://nixos.org/channels/nixos-25.05 nixos
nix-channel --update
```

# 3.7 NARs

Nix Archives

https://nix.dev/manual/nix/2.22/protocols/nix-archive

# 3.8 Profiles, GC

Profiles: Atomic collection of symlinks into the Nix Store

# 3.8 Profiles, GC

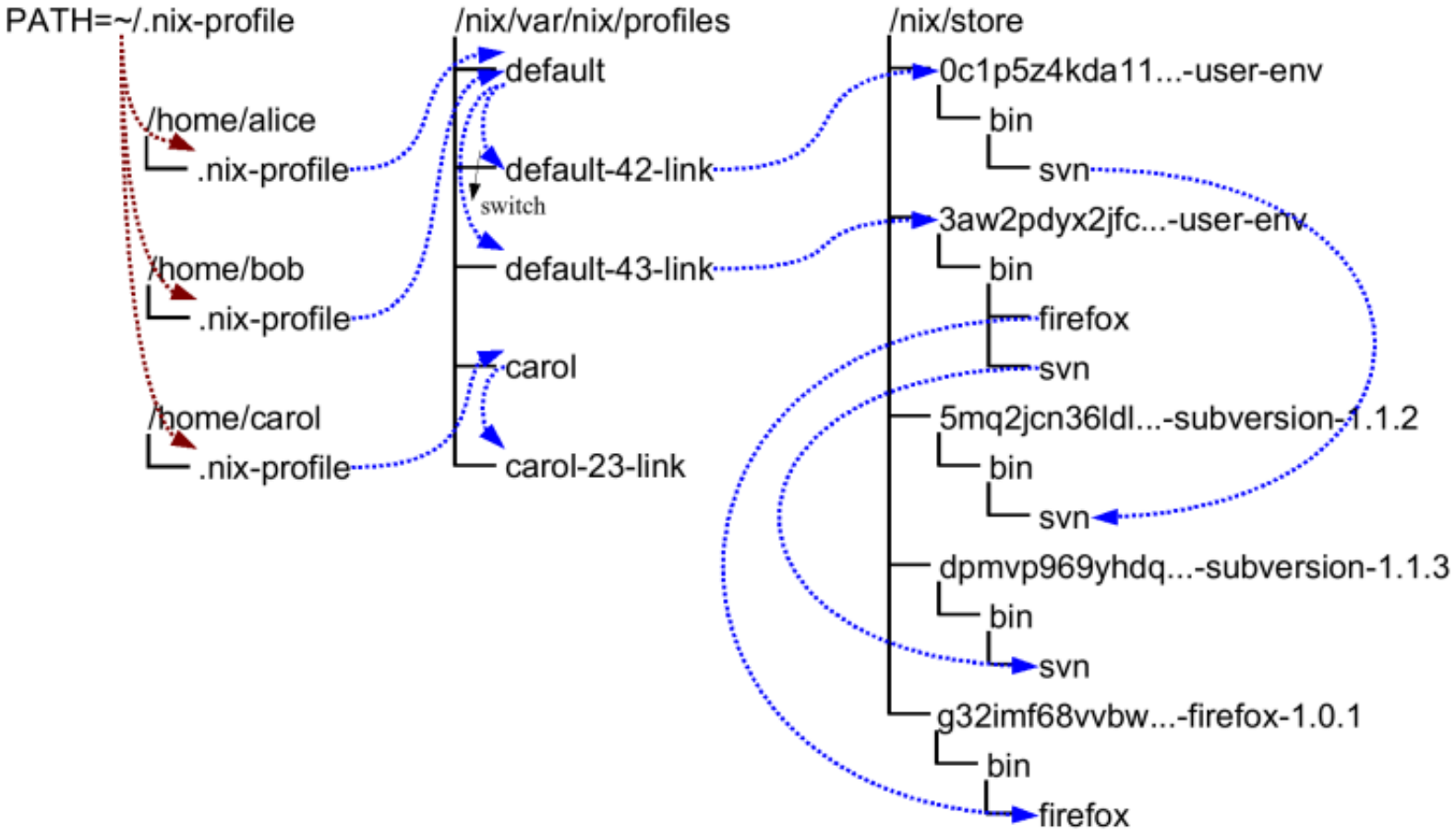



Figure 3: https://nix.dev/manual/nix/2.22/protocols/nix-archive

# 3.8 Profiles, GC

- `nix-env` manipulates user-profiles
- NixOS manipulates system profiles

```shell
ls -la /nix/var/nix/profiles/
```

# 3.8 Profiles, GC

Store paths can be marked as Garbage Collection roots.

- GC Roots will never be deleted
- Dependencies will never be deleted

Shell

```
nix-env --delete-generations old
nix-store --gc
nix-collect-garbage -d
nix-collect-garbage --delete-older-than 30d

nix profile ...
nix gc ...
```

# 4. Nix: The Build System

# 4.1 nix-build (ad-hoc)

Warning: `nix-build` is a stable nix command.

Shell

```
nix-build -E "with import <nixpkgs> { }; hello"
./result/bin/hello
rm result
```

# 4.2 nix-build (default.nix)

Warning: `default.nix` is a stable nix convention.

Create a file named `default.nix` with the following content:

Nix
```nix
let
  pkgs = import <nixpkgs> {};
in
pkgs.hello
```

Build this expression using `nix-build`. A result symlink will appear.

Shell
```shell
nix-build
./result/bin/hello
```

# 4.2 nix-build (default.nix)

```
3  rm result
```

# 4.3 nix vs nix flakes

Both `default.nix` and `shell.nix` are the stable Nix way of doing things. The experimental successor is Flakes.

I try to use flakes whenever possible, only using ad-hoc, `default.nix` and `shell.nix` for quick'n'dirty usecases.

# 4.4 Flakes

Flakes! Not scary at all!

However, a lot of confusion what flakes actually are

First, take a look at a flake:

```shell
nix flake init
cat flake.nix
nix flake show
cat flake.lock
nix flake update
```

# 4.4 Flakes

Nix

```nix
{
  description = "A very basic flake";
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs?ref=nixos-unstable";
  };
  outputs = { self, nixpkgs }: {
    packages.x86_64-linux.hello = nixpkgs.legacyPackages.x86_64-linux.hello;
    packages.x86_64-linux.default = self.packages.x86_64-linux.hello;
```

```
9     };
10  }
```

# 4.4 Flakes

So, a flake is only another way to write nix expressions

- Specify inputs
- Specify outputs
- Less implied context like default.nix, shell.nix or nixpkgs

# 4.4 Flakes

Ease of use:

- Combine all possible outputs into one file `flake.nix`
  - We know `shell.nix` and `default.nix`, but also NixOS Configs etc.
- Flakes can easily be used as inputs for other flakes
  - Typically, `nixpkgs` is used as a flake input

Reproducability out of the box:

- Locks inputs in a `flake.lock` file (just like Rust's `cargo.lock`)
  - In most cases a Link (e.g. Git Revision) and a Hash
- Exactly specify which architecture a package is for
  - i.e. `x86_64-linux`

But also, added complexity (no sane person can write flakes by memory)

# 4.4 Flakes

Now, look at a comlpete flake template

Nix

```nix
{
  description = "A very basic flake";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs?ref=nixos-unstable";
  };

  outputs = { self, ... }@inputs:
  {
```

# 4.4 Flakes

```
10      # Executed by `nix flake check`
11      checks."<system>"."<name>" = derivation;
12      # Executed by `nix build .#<name>`
13      packages."<system>"."<name>" = derivation;
14      # Executed by `nix build .`
15      packages."<system>".default = derivation;
16      # Executed by `nix run .#<name>`
17      apps."<system>"."<name>" = {
18        type = "app";
19        program = "<store-path>";
20      };
```

```
21      # Executed by `nix run . -- <args?>`
22      apps."<system>".default = { type = "app"; program =
"..."; };
23
24      # Formatter (alejandra, nixfmt or nixpkgs-fmt)
25      formatter."<system>" = derivation;
26      # Used for nixpkgs packages, also accessible via
`nix build .#<name>`
27      legacyPackages."<system>"."<name>" = derivation;
28      # Overlay, consumed by other flakes
29      overlays."<name>" = final: prev: { };
```

# 4.4 Flakes

```
30      # Default overlay
31      overlays.default = final: prev: { };
32      # Nixos module, consumed by other flakes
33      nixosModules."<name>" = { config, ... }: { options = {}; config = {}; };
34      # Default module
35      nixosModules.default = { config, ... }: { options = {}; config = {}; };
36      # Used with `nixos-rebuild switch --flake .#<hostname>`
```

# 4.4 Flakes

```
      #
37  nixosConfigurations."<hostname>".config.system.build.topl
    must be a derivation
38      nixosConfigurations."<hostname>" = {};
39      # Used by `nix develop .#<name>`
40      devShells."<system>"."<name>" = derivation;
41      # Used by `nix develop`
42      devShells."<system>".default = derivation;
43      # Hydra build jobs
44      hydraJobs."<attr>"."<system>" = derivation;
45      # Used by `nix flake init -t <flake>#<name>`
```

```
46      templates."<name>" = {
47        path = "<store-path>";
48        description = "template description goes here?";
49      };
50      # Used by `nix flake init -t <flake>`
51      templates.default = { path = "<store-path>";
    description = ""; };
52    }
53  }
```

https://wiki.nixos.org/wiki/Flakes

# 4.4 Flakes

Some other important tools related to flakes:

- flake-compat: Interface between `default.nix`, `shell.nix` and `flake.nix`
  - https://github.com/edolstra/flake-compat
- flake-utils: easily use multiple systems
  - https://github.com/numtide/flake-utils

# 4.4 Flakes

Let's have a look at more Flakes!

- Behold, my NixOS Config!
- Nixpkgs
- 4rth wall break

# 4.5 nix develop

Enters a development environment for a given package. Useful for debugging the build process.

```Shell
nix develop nixpkgs#hello
unpackPhase
cd <name>
configurePhase
mkdir build && cd build
buildPhase
checkPhase
installPhase
installCheckPhase
```

# 4.5 nix develop

```
10  ../outputs/out/bin/hello
```

CTRL+D quits again.

# 4.6 Shells / nix shell

Now you better understand how Nix works, how does this work:

```
1  nix shell nixpkgs#hello
```

Shell

# 4.6 Shells / nix shell

Shells can be put inside a Flake.

```
1 nix flake init -t "github:determinatesystems/
  zero-to-nix#cpp-dev"
```

Shell

```
1 nix flake init -t "github:determinatesystems/
  zero-to-nix#rust-dev"
```

Shell

```
1 nix shell
```

Shell

# 4.7 Packages / nix build

Packages can be put inside a Flake.

```shell
nix flake init -t "github:determinatesystems/zero-to-nix#cpp-pkg"
```

Shell

```shell
nix flake init -t "github:determinatesystems/zero-to-nix#rust-pkg"
```

Shell

```shell
nix build
```

Shell

# 4.8 A “better” Nix Command

`nix-output-monitor` makes the building process prettier

- Drop-in wrapper for
  - `nix-build` and `nix build`
  - `nix-shell` and `nix shell`
  - `nix develop`
- Just replace `nix` with `nom`

Test it out like this:

```
1 nix shell nixpkgs#nix-output-monitor
2 nom shell nixpkgs#linux --no-substitute
```

Shell

# 5. Other Stuff

# 5.1 NixOS

- https://nixos.org/manual/nixos/stable/

## 5.1.1 configuration.nix

configuration.nix

## 5.1.2 hardware-configuration.nix

hardware-configuration.nix

## 5.1.3 The Module System

modules

## 5.1.4 Options

Options

### 5.1.5 nh

- https://github.com/nix-community/nh

### 5.1.6 nix-tree

- https://github.com/utdemir/nix-tree

# 5.2 Home Manager

Home Manager declaratively manages your home's dotfiles.

Nix

```nix
programs.git = {
  enable = true;
  userName  = "my_git_username";
  userEmail = "my_git_username@gmail.com";
};
programs.direnv = {
  enable = true;
  nix-direnv.enable = true;
};
programs.fish.enable = true;
```

# 5.2 Home Manager

Nix

```nix
programs.git = {
  enable = true;
  userName  = "my_git_username";
  userEmail = "my_git_username@gmail.com";
};
programs.direnv = {
  enable = true;
  nix-direnv.enable = true;
};
```

# 5.3 Fix dynamically linked Binaries

For running most binaries, add `pkgs.autoPatchelfHook` to your env, either ad-hoc or in the nativeBuildInputs of a package.

```
1  nix shell nixpkgs#autoPatchelfHook
```

Shell

Or write a FHS env. Or do whatever this is:

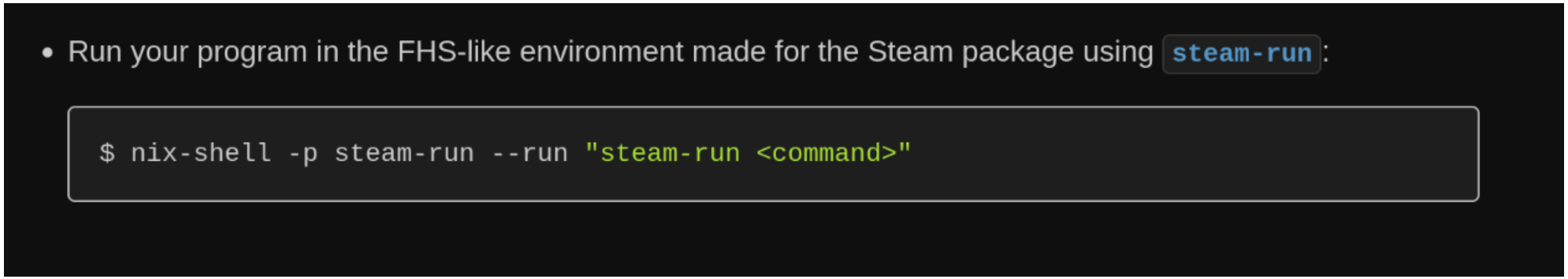



Figure 4: Official FAQ from https://nix.dev/guides/faq

# 5.4 FHS

Nixpkgs provides a `pkgs.buildFHSEnv` function, calling `.env` on it drops you in its shell.

```nix
{ pkgs ? import <nixpkgs> {} }:
  (pkgs.buildFHSEnv {
    name = "buildroot-fhs-env";
    multiPkgs = pkgs: (with pkgs; [ hello ]);
    runScript = "fish";
  }).env
```

https://nixos.org/manual/nixpkgs/stable/#sec-fhs-environments

# 5.5 Cross Compilation

Cross compilation can be achieved by using `pkgsCross.<architecture>` instead of `pkgs`, e.g by executing

```
1 nix build nixpkgs#pkgsCross.riscv64.hello
```
Shell

# 5.6 CUDA

## CUDA

# 5.7 Binary Caching, Cachix

Binary Caching, Cachix

# 5.8 Sharing Nix Stores

## Sharing Nix Stores

# 5.9 Distributed Builds

Distributed Builds

https://nix.dev/tutorials/nixos/distributed-builds-setup.html

# 5.10 Hydra

Hydra

https://github.com/NixOS/hydra https://wiki.nixos.org/wiki/Hydra
https://hydra.nixos.org/

# 5.11 Darwin

Darwin

https://github.com/nix-darwin/nix-darwin

# 5.12 Disko


Figure 5: Official `Disko` Logo

`Disko` enables declarative Disk Partitioning for NixOS

- `https://github.com/nix-community/disko`

# 5.13 SOPS / Age-Nix

- https://github.com/ryantm/agenix (recommended)
- https://github.com/Mic92/sops-nix (also possible)

# 6. Appendix

# 6.1 Important Websites

Nix: Package Management

- `https://search.nixos.org/`
  - Official Package and Option Index
- `https://github.com/nixos/nixpks`
  - Repo for Nixpkgs

# 6.1 Important Websites

Nix: Development

- `https://nix.dev/`
  - Good Reference for using Nix productively, especially for Devs
- `https://noogle.dev/`
  - Nix Function Search, Function of the day
- `https://zero-to-nix.com/`
  - Very good tutorials!
- `https://nixos.org/manual/nixpkgs/stable/`
  - Official Manual for Nixpkgs

# 6.1 Important Websites

NixOS

- `https://wiki.nixos.org/`
  - Official Wiki, good for NixOS users
  - Warning: The older Community Wiki `https://nixos.wiki` is still online and often pops up when googling!
- `https://nixos.org/manual/nixos/stable/`
  - Official Manual for NixOS
- `https://github.com/nix-community/nixos-hardware`
  - Repo for NixOS modules for specific hardware
- `https://www.youtube.com/@vimjoyer`
  - Good, quick tutorials

# 6.1 Important Websites

Just great:

- `https://github.com/nix-community/awesome-nix`
  - List of many community projects
- `https://nixos.org/guides/nix-pills/`
  - Explaining Nix from the ground up (tho somewhat dated)
- `https://edolstra.github.io/pubs/phd-thesis.pdf`
  - The PHD of Eelco Dolstra about developing Nix

# 6.2 Other stuff

Customers Who Bought This Item Also Bought

# 6.2 Other stuff

`https://guix.gnu.org/`

- Guile Scheme as Language
- No non-FOSS packages
- Hurd Kernel sometime?

https://luj.fr/blog/how-nixos-could-have-detected-xz.html

https://guix.gnu.org/en/blog/2023/the-full-source-bootstrap-building-from-source-all-the-way-down/