
Nix & NixOS 101
nixcademy.com

Section: Nix & NixOS 101 Training

About You 🙋

About the Lecturers 👨‍💻

The Nix & NixOS 101 Class 🏫

About You 🙋

Your Starting Point and Expectations

What technology stacks and languages are you **experienced** with already?

What are your **feelings** about Nix(OS)?

What are your **expectations** from this training?

About the Lecturers 👨‍🏫

👨‍💻 M.Sc. EEng, Braunschweig, Germany

Years of Contribution to the Nixpkgs Project,
NixOS Test Driver Maintainer

💪 Strengths: SW Arch., Complexity Control, C++, Haskell

2023 NIXCADEMY EXPERT EDUCATION

2018 FH MÜNSTER University of Applied Sciences

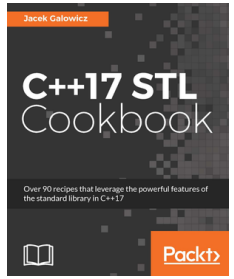
2017 CYBERUS TECHNOLOGY

2014 FIREEYE™

2013 RWTH AACHEN UNIVERSITY

2012 intel

Author



Editor

Personal Path to Nix(OS)

Electrical Engineering

↓

Operating Systems & Micro Kernels

↓

C++

↓

C++ Template Metaprogramming

↓

Haskell

↓

Nix

↓

NixOS

👨‍💻 M.Sc. Computer Science, University of applied sciences Münster, Germany

Travelling the world as digital nomad

💪 Strengths: Software Architecture, DevOps, CI/CD, Functional Programming, Haskell

2024 NIXCADEMY EXPERT EDUCATION

2021 CYBERUS TECHNOLOGY

2014 FH MÜNSTER University of Applied Sciences

Student Job in Java EE

↓

Software Quality lecture at University by Jacek

↓

Master Thesis
@ Cyberus Technology GmbH

↓

Got hooked on Nix

👨‍💻 Informatics Degree, Software Engineer

Nix/NixOS user and contributer since 2019

💪 Interests/Experience: Regression-free Development, Complexity Control, Haskell, Rust, C++

2024 Supercede

2023 NIXCADEMY EXPERT EDUCATION

2022 Platonic.Systems

2020 CYBERUS TECHNOLOGY

2017 - 2019 KIT Karlsruher Institut für Technologie, FZI

🎨 Art Degree, Software Engineer

Nix/NixOS user & Nixpkgs contributer since 2020

🍵 Interests/Experience

Free software, functional programming, front-end dev, blogging, □□□ + **ลาว**, cooking, gridiron football

Professional software development since 2010 (internship ⟶ corporation ⟶ ad firm ⟶ startups ⟶ remote ⟶ contracting)

Advertising • Cryptocurrencies • Data processing • Healthcare • Real estate • User media management

Art
↓
Interactive media
↓
Web dev / front-end → ECMAScript
↓
FP: Elm / PureScript / etc.
↓
Nix / NixOS
↑
Arch
↑
Debian

Nix user & contributor since 2021

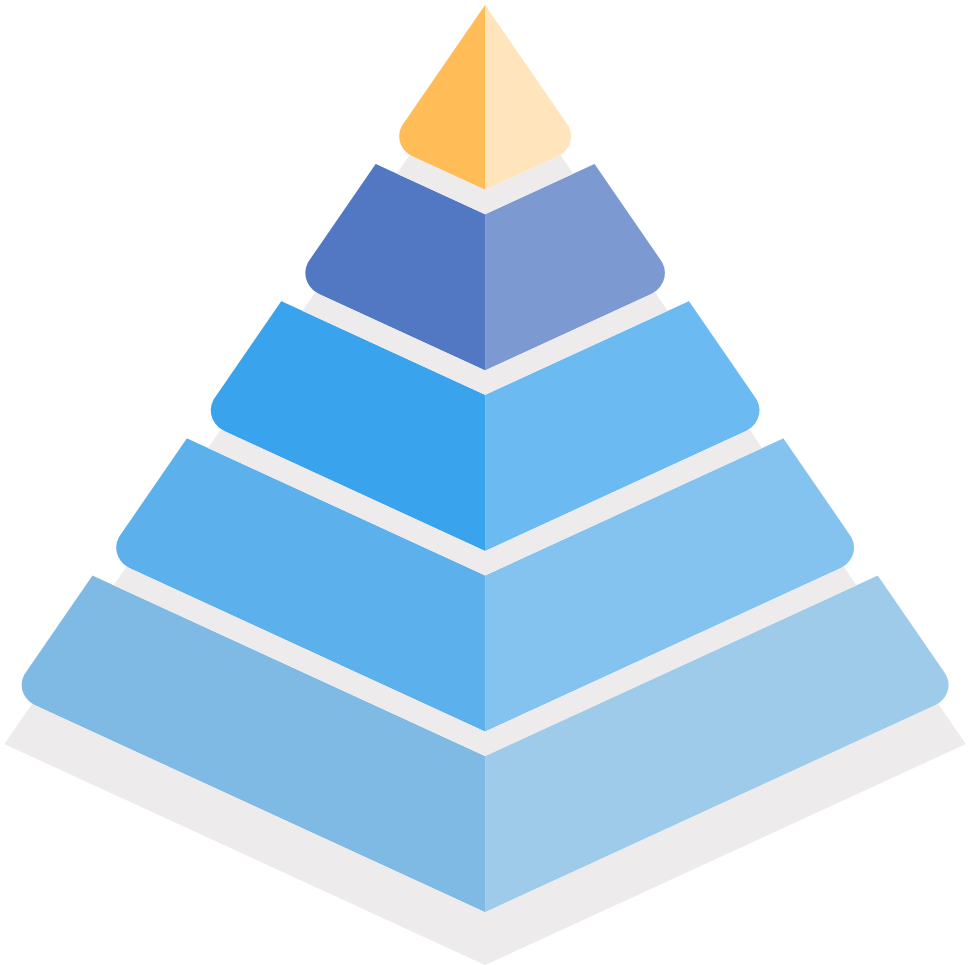
Member of the NixOS Infrastructure Team

Leader of the Nix Formatting Team

2018 - 2024	honor	*Python, TypeScript/React, Docker, k8s, asdf, Salt*
2015 - 2018	WORLD CUBE ASSOCIATION	*Java, Ruby, Chef*
2010 - 2015	ARISTA	*TACC, C++, Python*

The Nix & NixOS 101 Class 🏫

Class Goals



05 **Full Control over Nix**

You master deadlines by working with and not against nix.

04 **Package Variation**

Overrides, Overlays, Package variants

03 **Sandbox & Packaging**

Understanding how Nix makes builds reproducible

02 **Nix Language**

Language Syntax, brief functional programming intro

01 **Nix Basics**

Basic Commands & Package Management

User Perspective:

- Install, Modify, & Use Packages
- Profit from the Biggest and Freshest Package Repository
- … Without Tainting the System's State

Developer Perspective:

- Define Declarative Development Environments
- Create Reproducibly Buildable Packages
- Manage, Manipulate & Pin Dependencies
- Separate and Manage Patch Sets
- Extend Nixpkgs with Complex Overlays
- Prepare Projects and Nix Expressions for Cross-Compilation

User Perspective:

- Install & Configure NixOS on Laptops, Servers, Cloud
- Define Containers, VMs, Custom Installer Images
- Seamlessly Update & Roll Back Systems
- Playful Evolution of Complex System Configs

Developer Perspective:

- Create NixOS Modules for Custom Services
- Simplify System Image Generation & Deployment
- Define Declarative Integration Tests

What's Not Included in *This* Class

- Dependency Management for *All* the Languages
- FHS UserEnvironments
- macOS Specialties
- AppImage, SNAP, systemd Portable Services, etc.
- nix ops, krops, etc.
- Setting up Binary Caches and Remote Builders

Don't rush. 🏃
We have time. ⏳
It's not a race. 🏎️

It's also an exercise in how
to read the manuals. 📚

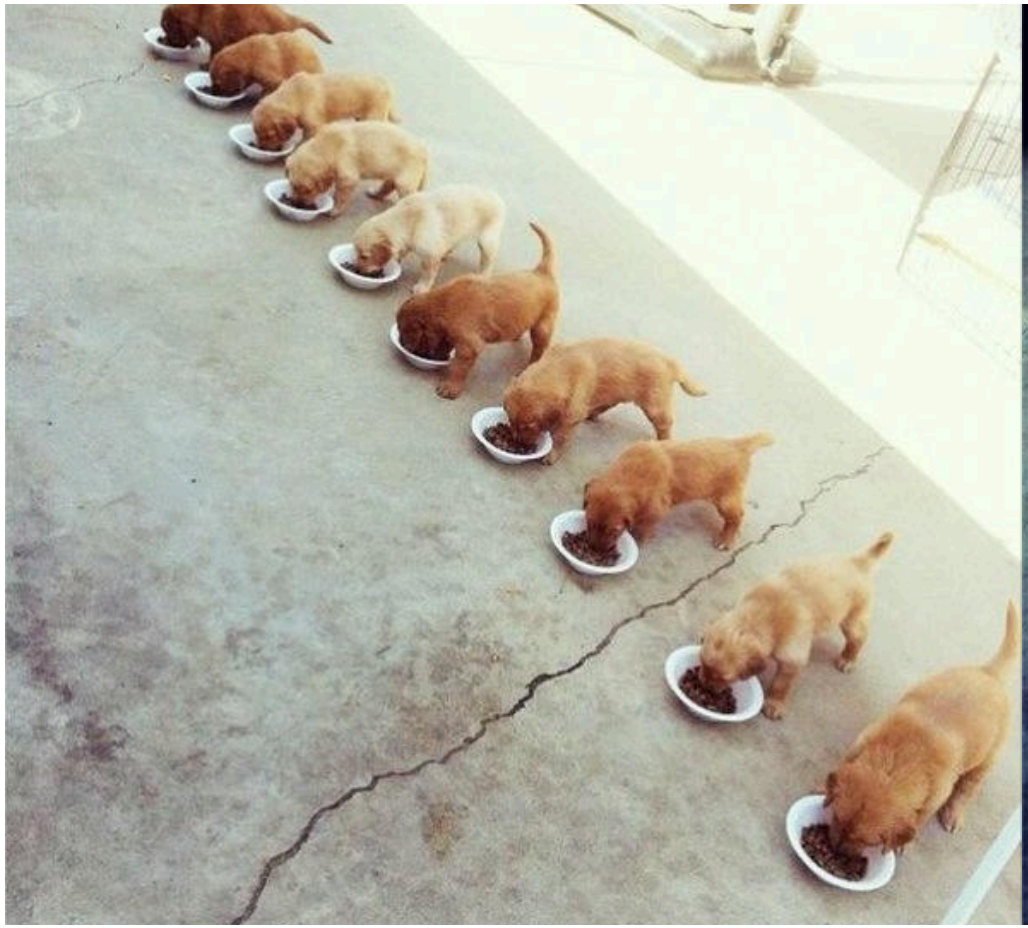
You'll get help. 🚸
We're not in school. 🏫
You can work together. 🤝

How to draw an owl

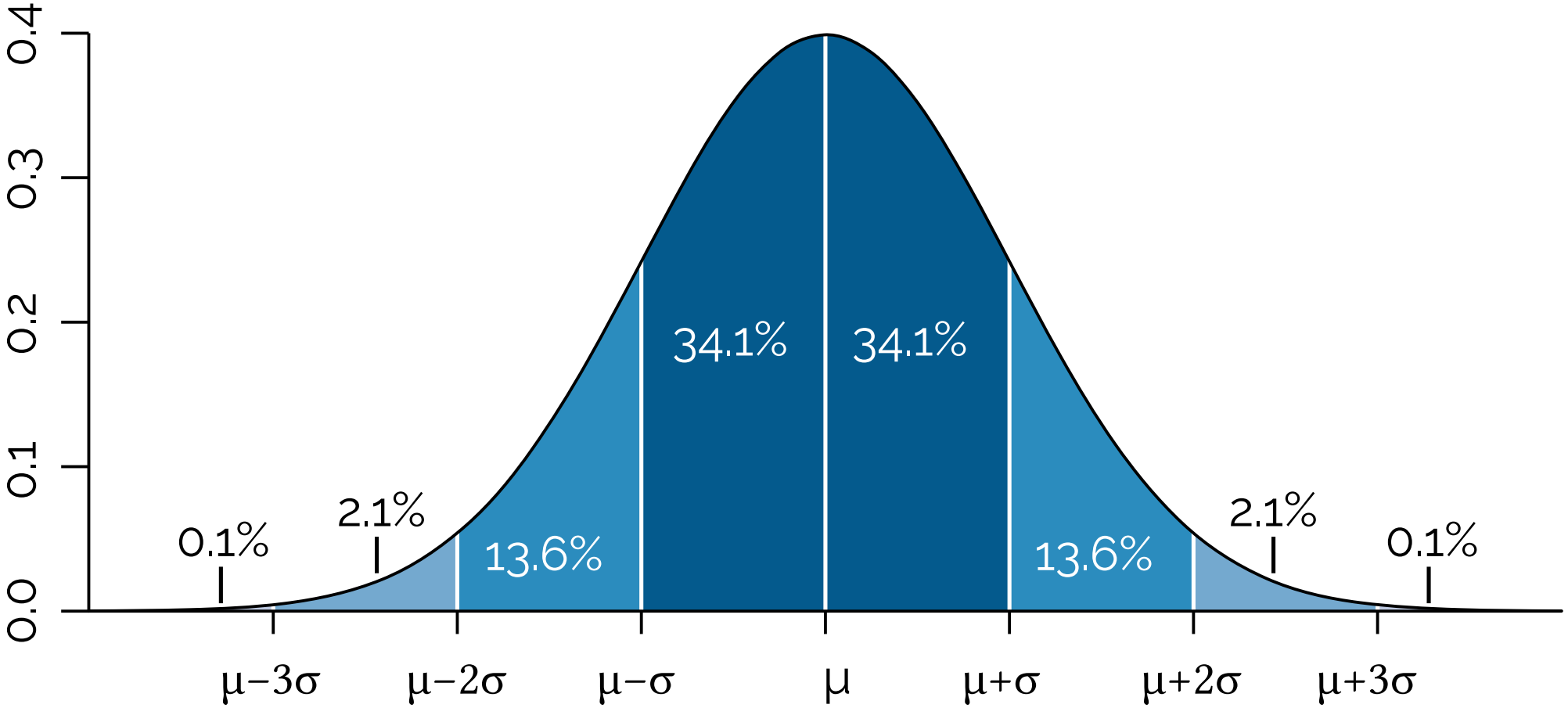
Fig 1. Draw two circles

Fig 2. Draw the rest of the damn owl

THEORY



PRACTICE


0.0
0.1
0.2
0.3
0.4
0.1%
2.1%
13.6%
34.1%
34.1%
13.6%
2.1%
0.1%
μ−3σ
μ−2σ
μ−σ
μ
μ+σ
μ+2σ
μ+3σ

Let's plan this week's days like this:

🕘	9:00	9 AM	Begin of training
🕛	12:00	12 PM	Lunch 🍝
🕐	13:00	1 PM	Resumption of training
🕔	17:00	5 PM	End of the day 🍺 & 🍸

☕ & 🚽 Pause: 10 Minutes every ~1.5h

Nix Part:

- Any Linux/macOS Laptop 💻 or VM with `sudo` rights and internet
- Windows WSL might work
- Please don't use Docker containers

NixOS Part:

- In this part, we also need KVM
- Nested Virtualization might not work satisfactorily

Section: Nix

About Nix

Using Nix 🛠️

Nix Language 🈵

Packaging Basics 📦

Pinning 📌

Project Patterns 🎨

Advanced Packaging Topics 🚀

Cross compilation 🔀

Overlays 🔁

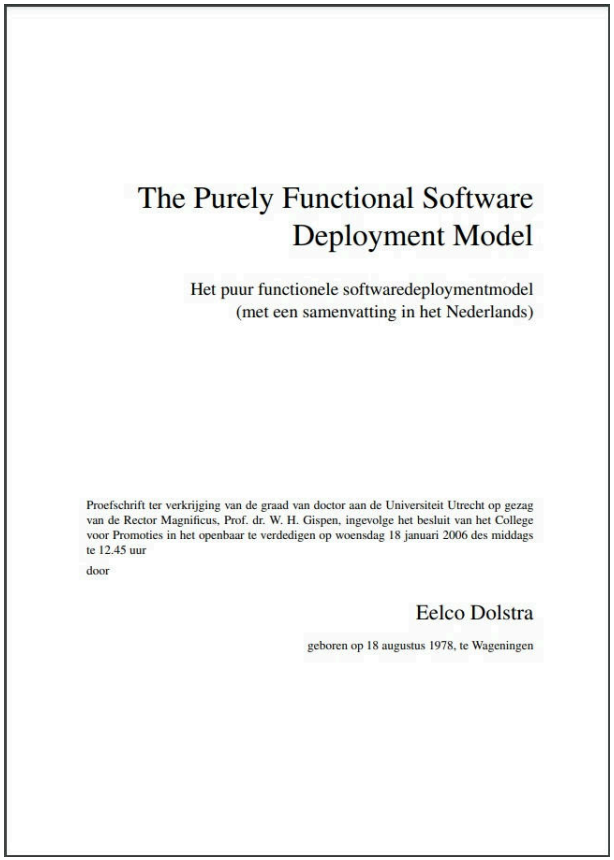
Flakes ❄️

About Nix

Nix: A PhD Thesis that Escalated 👩‍🎓

Nix started in 2003 as an academic project at Universiteit Utrecht (EU, Netherlands):

"This thesis is about getting computer programs from one machine to another—and having them still work when they get there."



The Purely Functional Software Deployment Model

Het puur functionele softwaredeploymentmodel
(met een samenvatting in het Nederlands)

Proefschrift ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de Rector Magnificus, Prof. dr. W. H. Gispen, ingevolge het besluit van het College voor Promoties in het openbaar te verdedigen op woensdag 18 januari 2006 des middags te 12.45 uur

door

Eelco Dolstra

geboren op 18 augustus 1978, te Wageningen

Eelco Dolstra's original PhD thesis: https://edolstra.github.io/pubs/phd-thesis.pdf

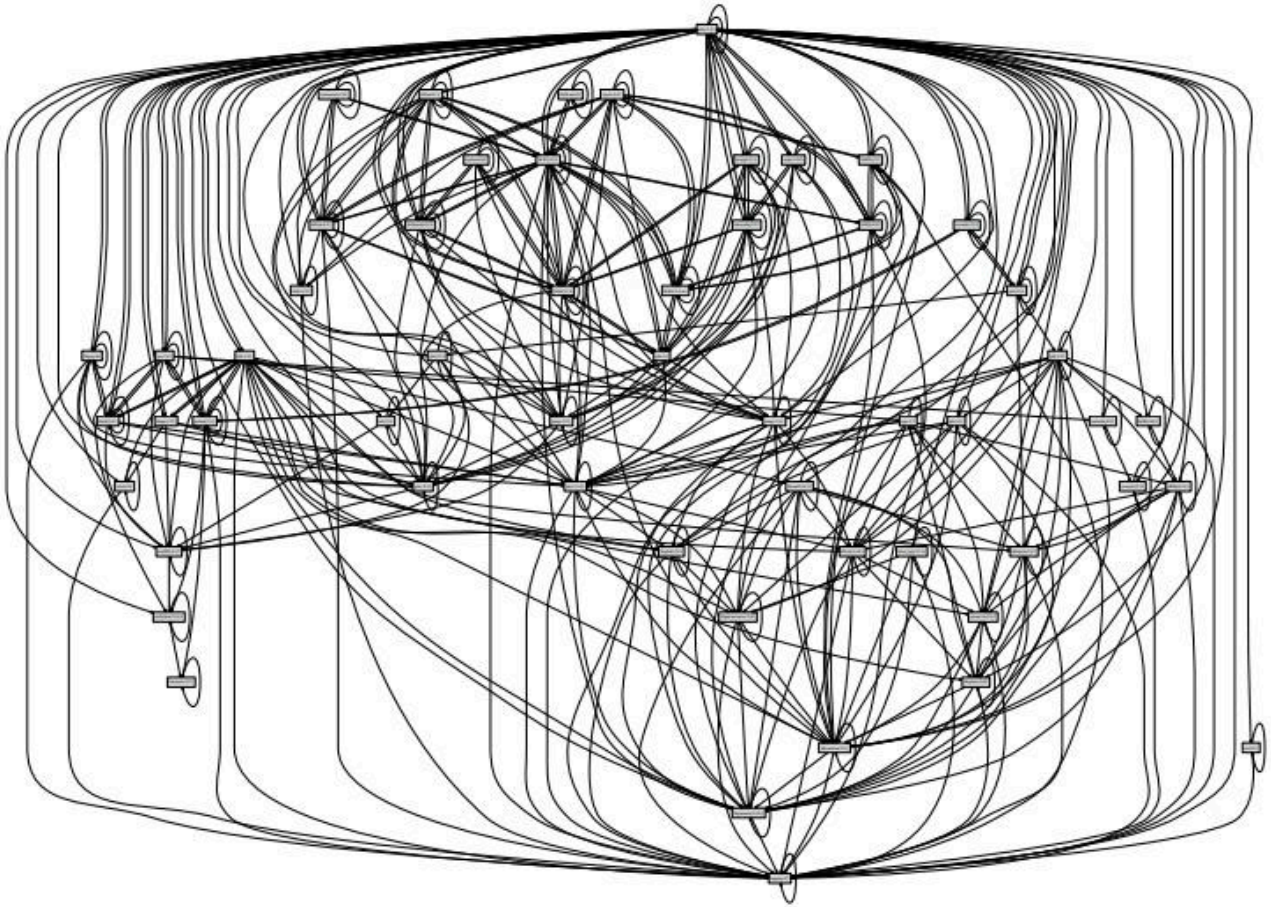
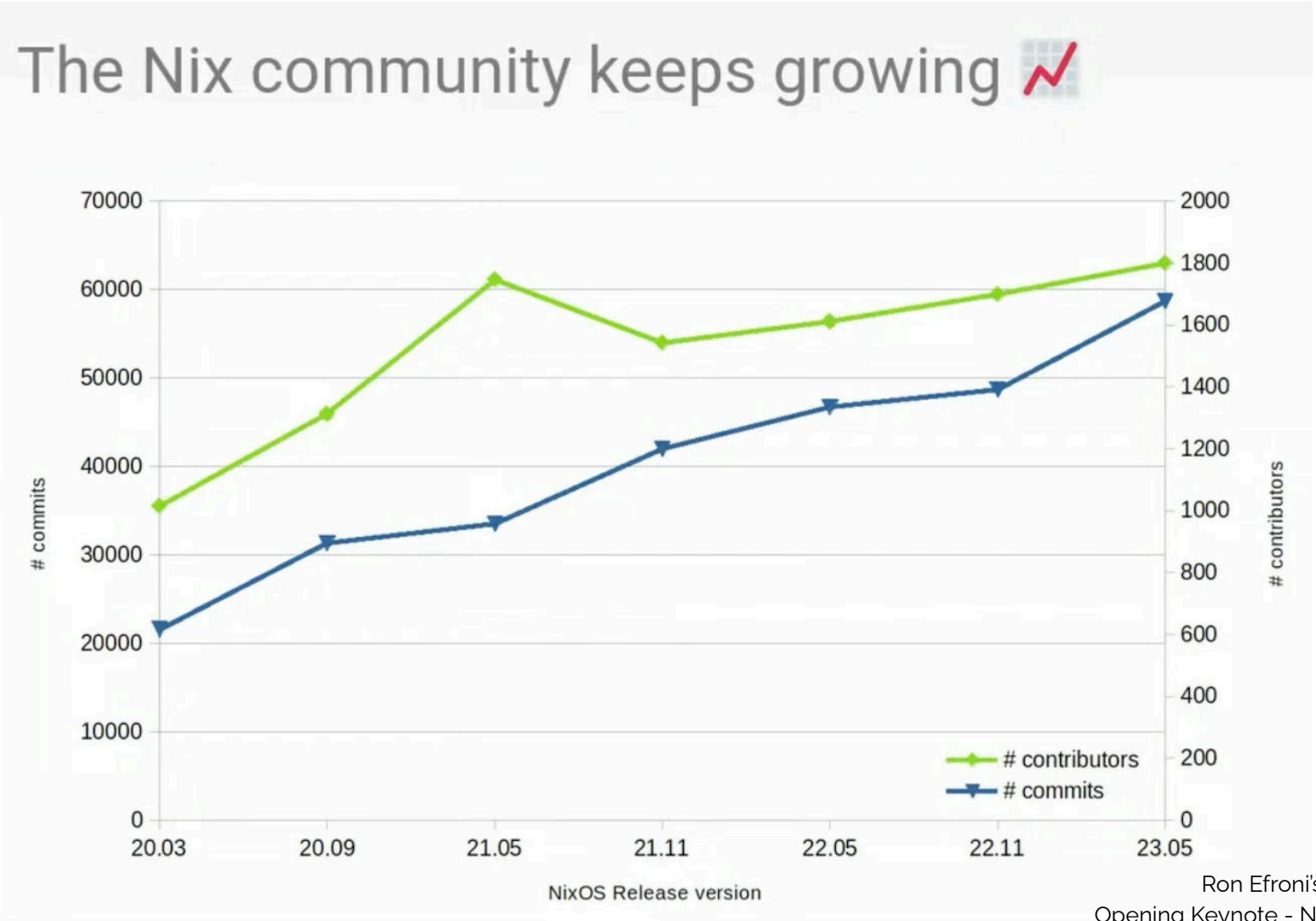


Figure 1.5.: The *dependency hell*: the runtime dependency graph of Mozilla Firefox

Source:
Nix PhD Thesis

Nixpkgs Commits and Contributors Growth



Ron Efroni's NixCon Berlin 2024
Opening Keynote - Nix State of the Union
https://www.youtube.com/watch?v=oAZkHpSWhaI

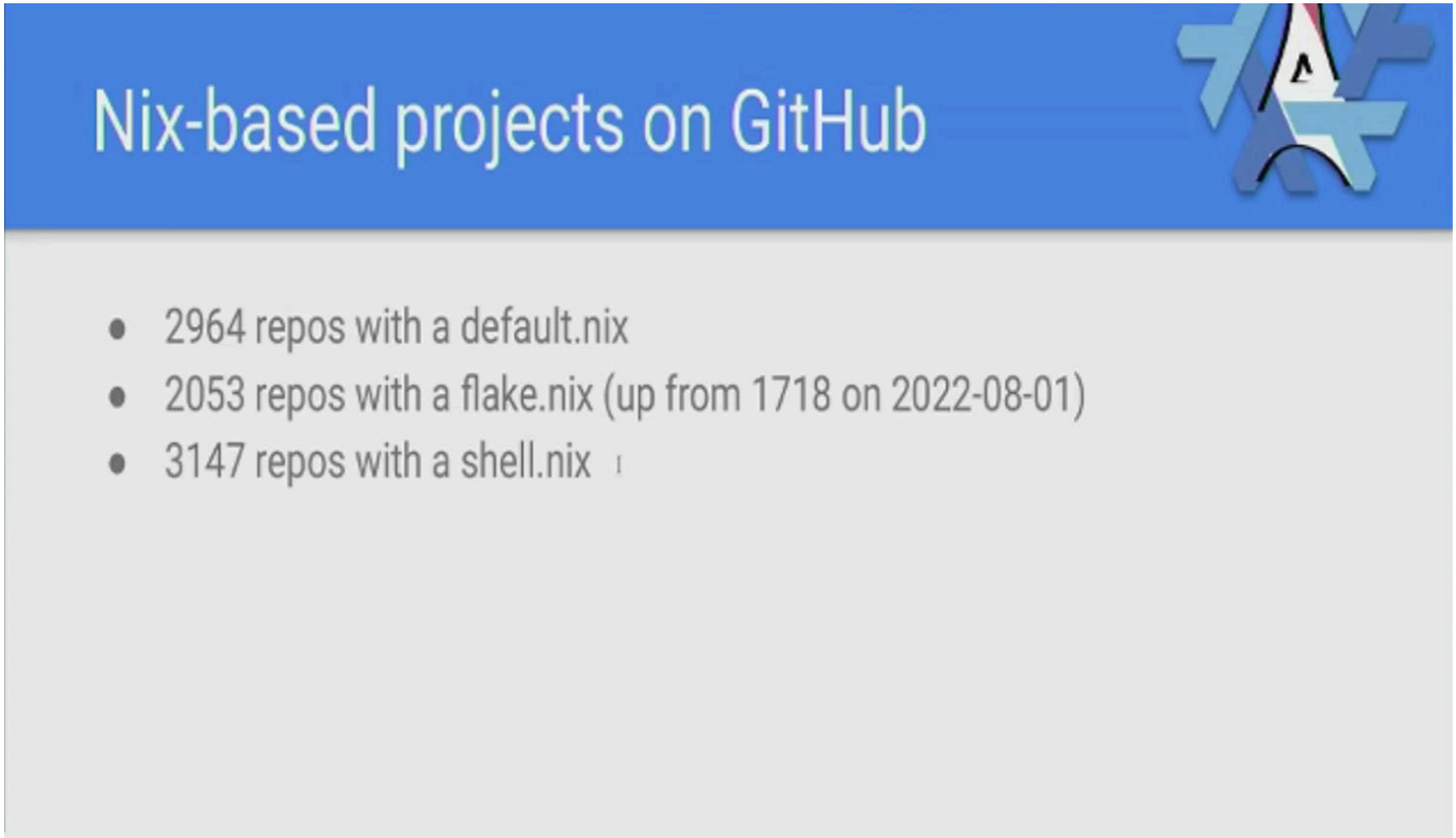
tbnsn@anduril.com

Eelco Dolstra's NixCon Paris 2022 Opening Keynote - Nix State of the Union
https://www.youtube.com/watch?v=s2fkgkN55vk

cache.nixos.org - Cache Size

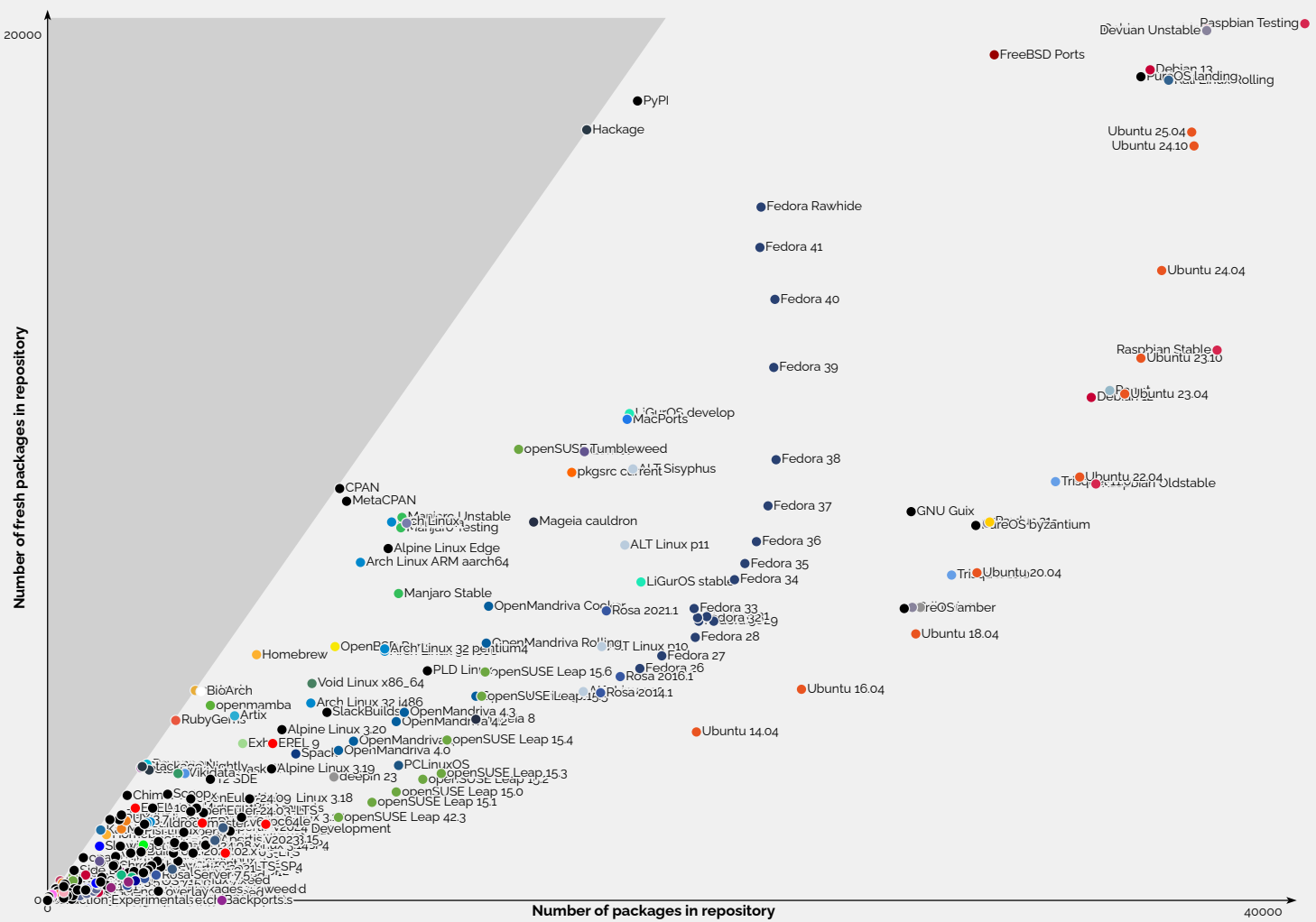
- **468.5 TiB** Infrequent Access, **101.3 TiB** Standard
  (Sept 2023: **346.5 TiB** Infrequent Access, **114.1 TiB** Standard)
- **892M** objects
  (Sept 2023: **717M** objects)
- **2.27 PB** traffic served by Fastly in the last month

Ron Efroni's NixCon Berlin 2024 Opening Keynote - Nix State of the Union
https://www.youtube.com/watch?v=oAzkHpSWhaI



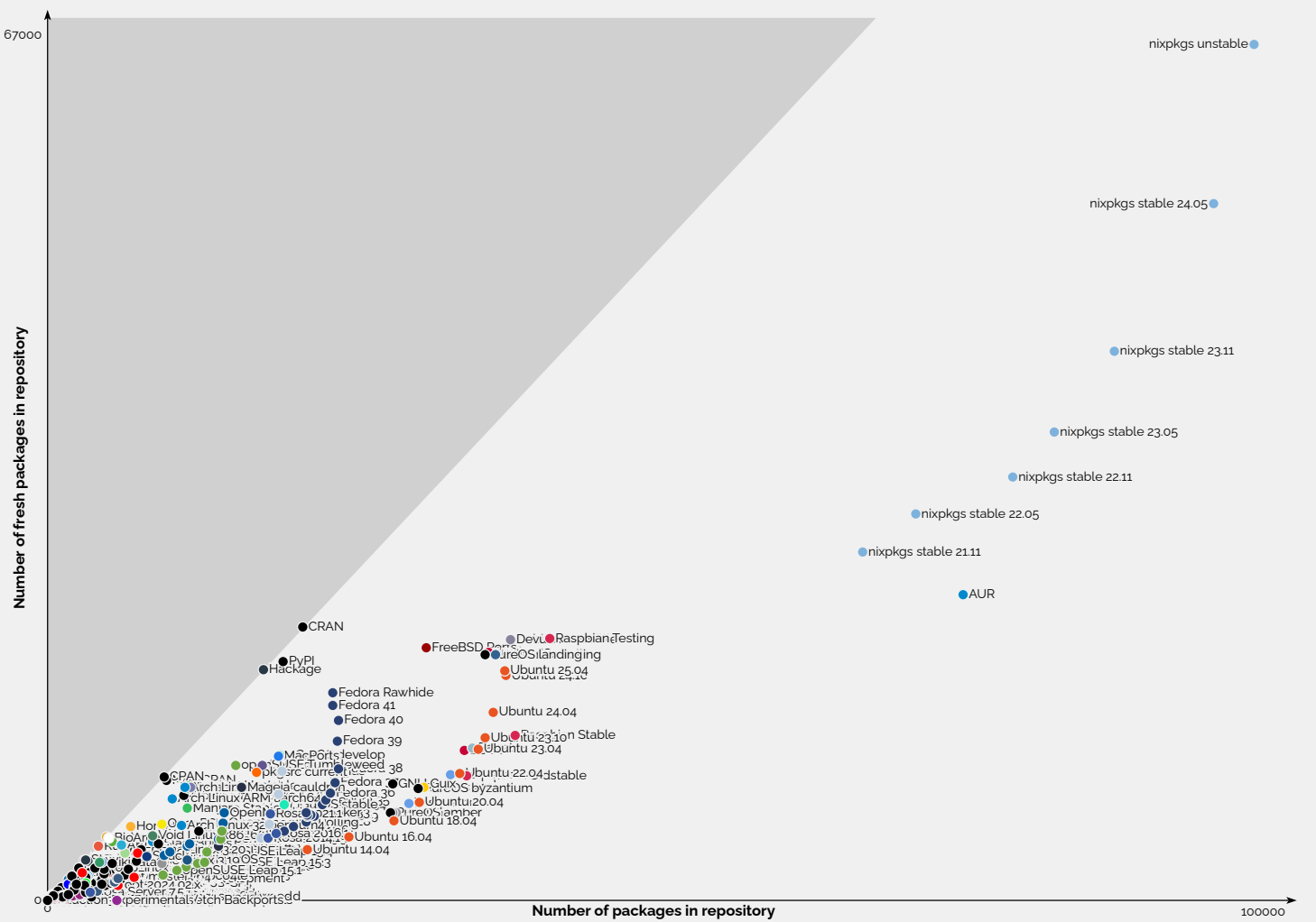
Eelco Dolstra's NixCon Paris 2022 Opening Keynote - Nix State of the Union
https://www.youtube.com/watch?v=s2fkgkN55vk

Nixpkgs: The Biggest Package Repository (1/2)



https://repology.org/repositories/graphs as of 2024-11

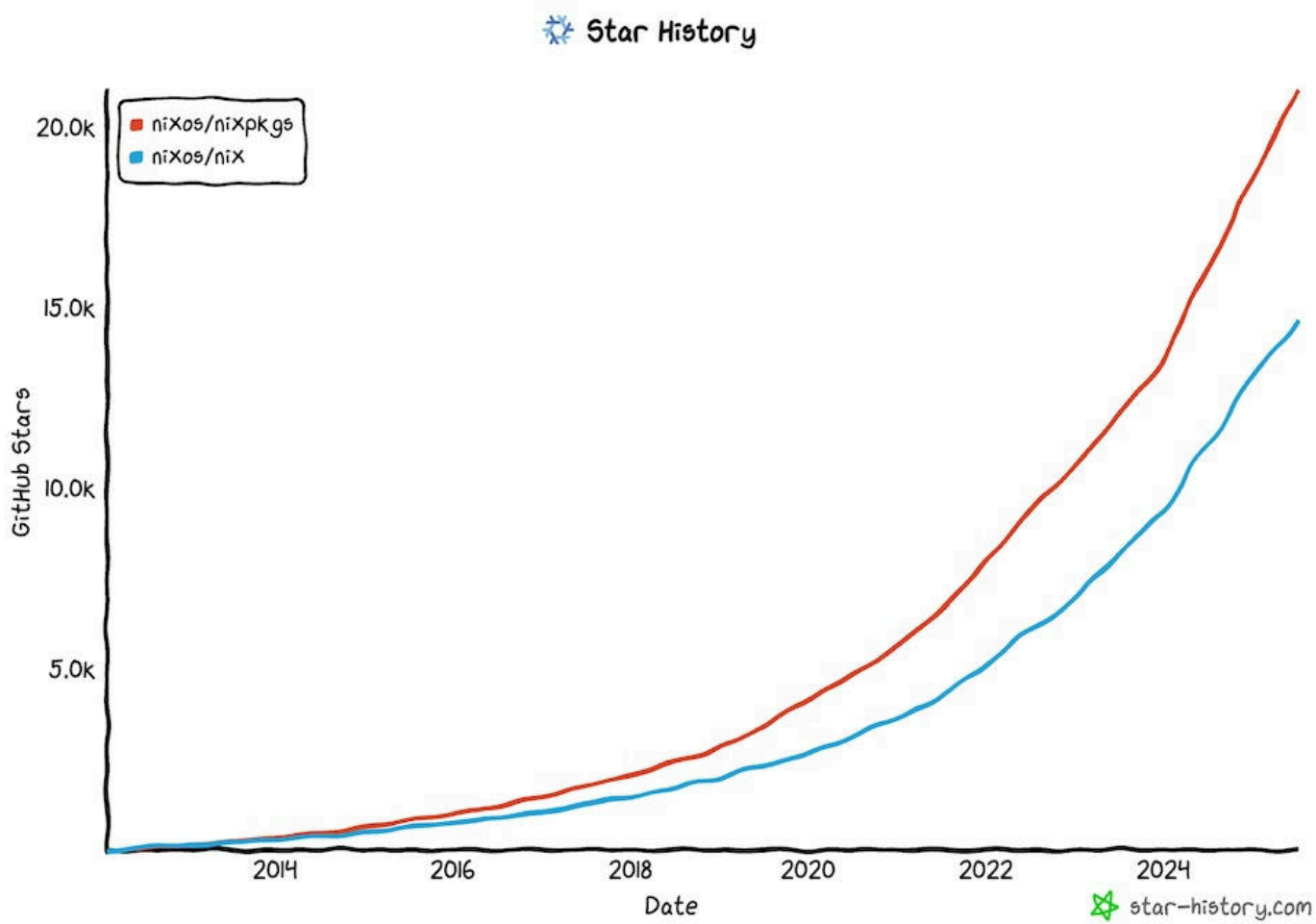
Nixpkgs: The Biggest Package Repository (2/2)



https://repology.org/repositories/graphs

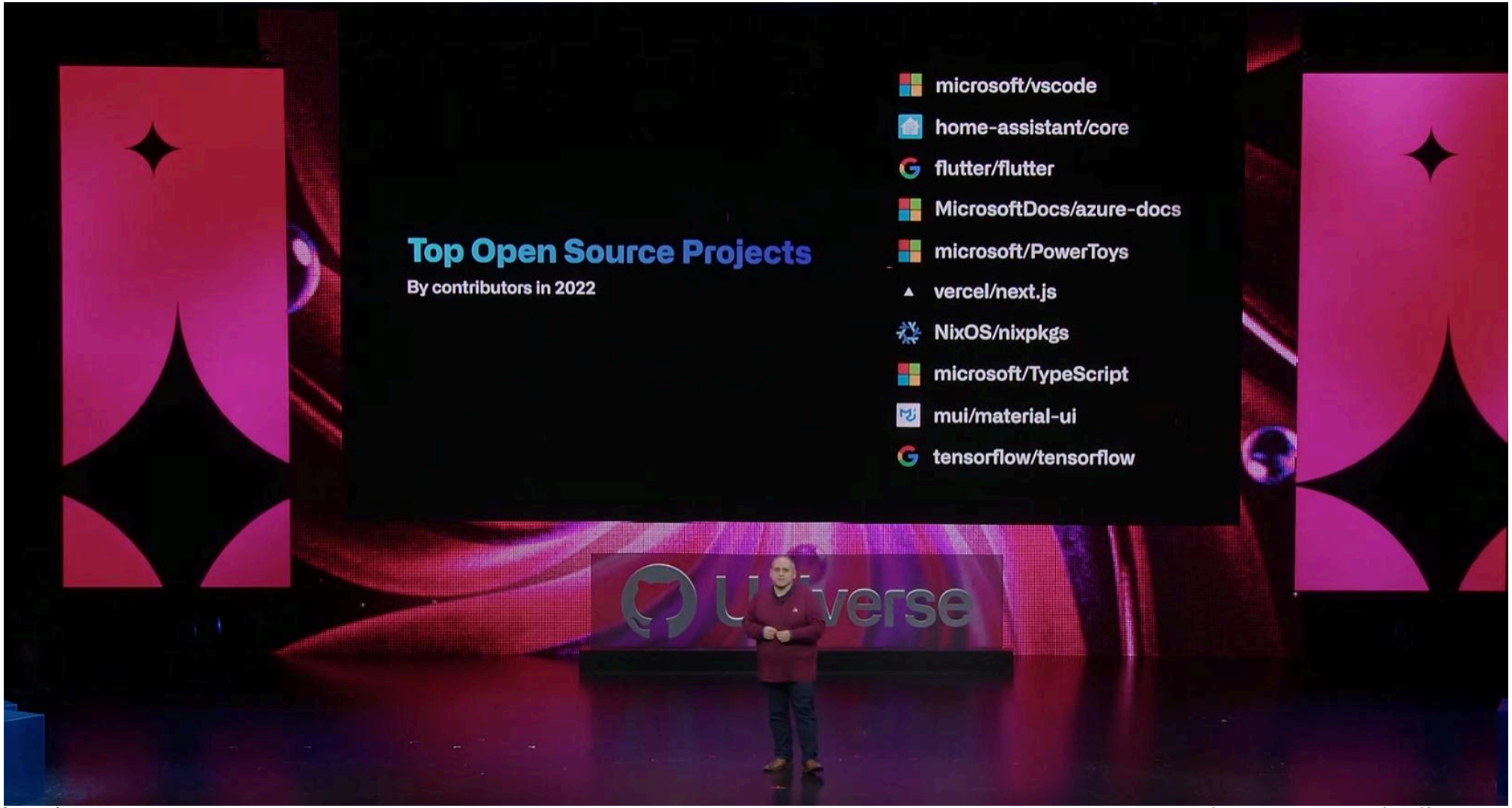
as of 2024-11

Nixpkgs: GitHub Star History

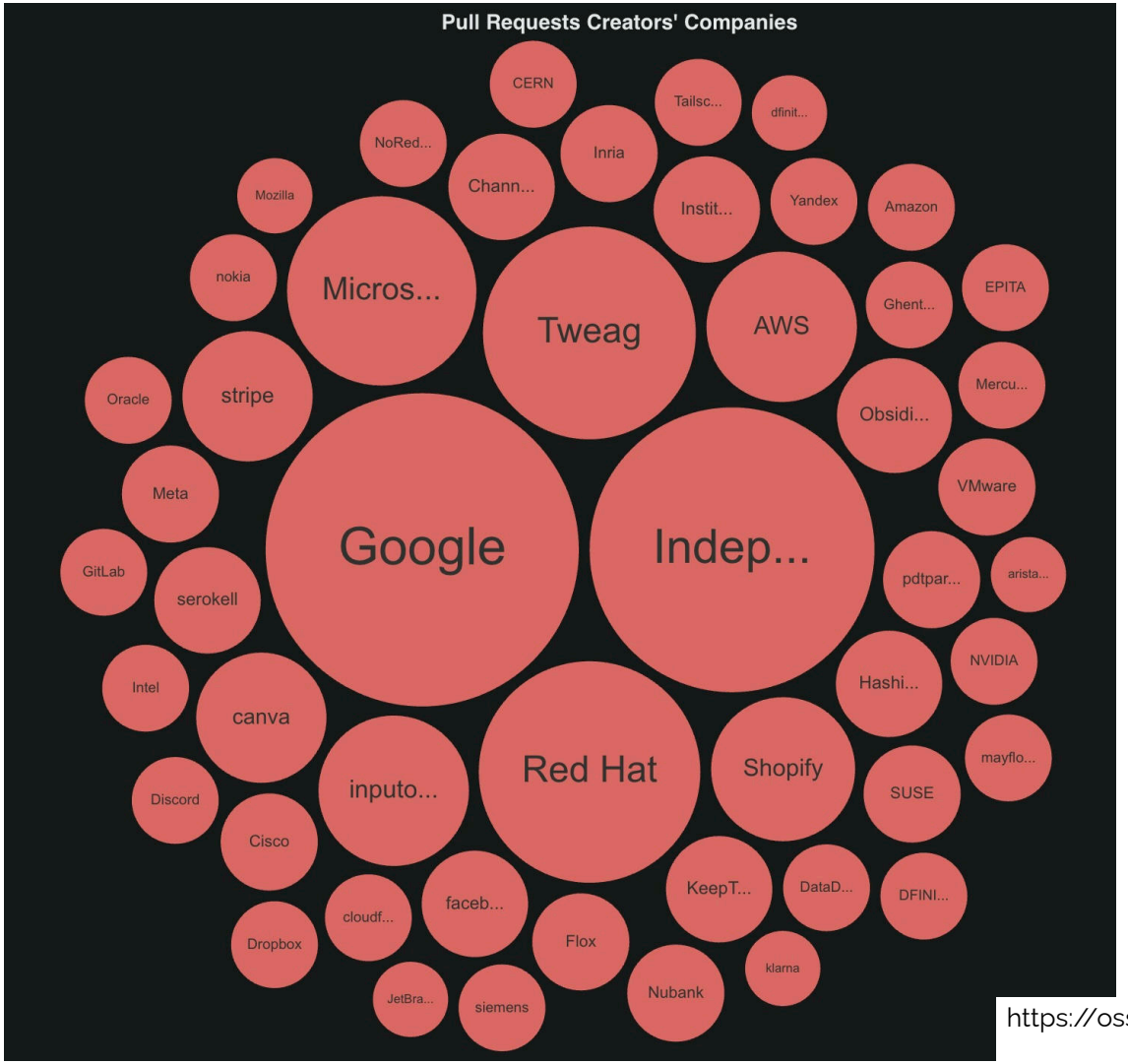


https://star-history.com
as of 2025-07

Nixpkgs: GitHub Top 10



Nixpkgs: Pull Requests by Company



https://ossinsight.io/analyze/NixOS/nixpkgs
as of 2025-07

Our Steps to Master Nix

1. Nix Commands
    - User perspective
2. Nix Language
    - Functional Programming
3. The `builtins.derivation` primitive
    - Sandboxing
4. `stdenv.mkDerivation`: Generic package builds
    - Advanced composable sandbox construction
5. Patterns: `callPackage`, `override`/`overrideAttrs`, overlays
    - Composable build products
6. Flakes
    - The new interface for distribution to users

Using Nix 🛠️

One-liner installation command from https://nixos.org/download:

```
sh <(curl --proto '=https' --tlsv1.2 -L https://nixos.org/nix/install)
```

If asked for multi-user vs single-user installation: Select multi-user!

Installation: Avoid distro packages

Prefer the official installer from https://nixos.org/download!

Only Debian/Ubuntu and Arch Linux packages are *tested*
(Link to automated tests)

If you used some other distro's packages or in doubt

👉 Uninstall and reinstall with official installer.

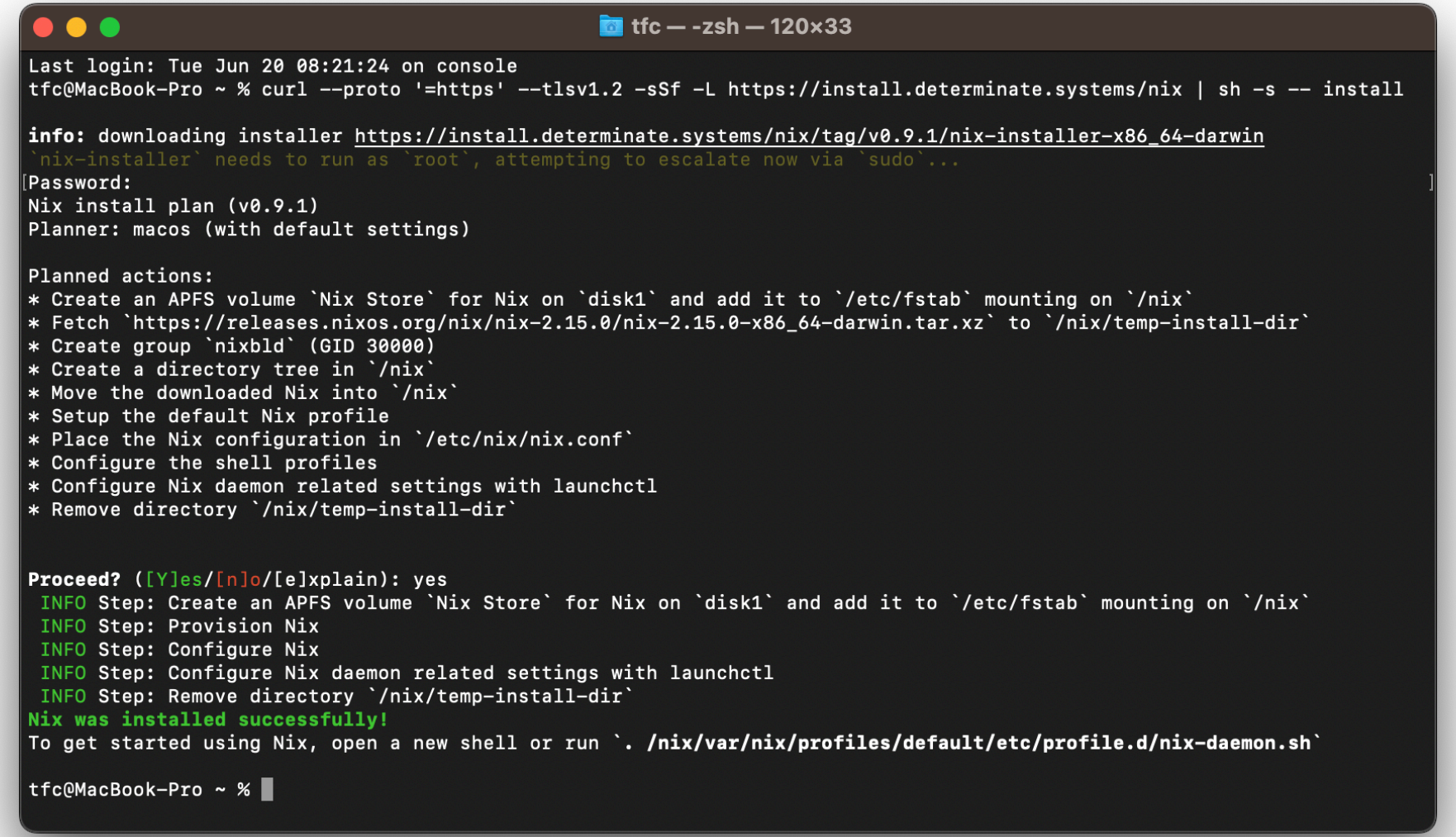
Determinate Systems provides an alternative installer with improved UX, auto-enabled flakes, and improved macOS compatibility:

https://github.com/DeterminateSystems/nix-installer

```
curl --proto '=https' --tlsv1.2 -sSf -L https://install.determinate.systems/nix | \
  sh -s -- install
```

Please note that it nowadays installs `determinate-nix` by default.

Installation: Alternative Determinate Systems Installer



Non-NixOS

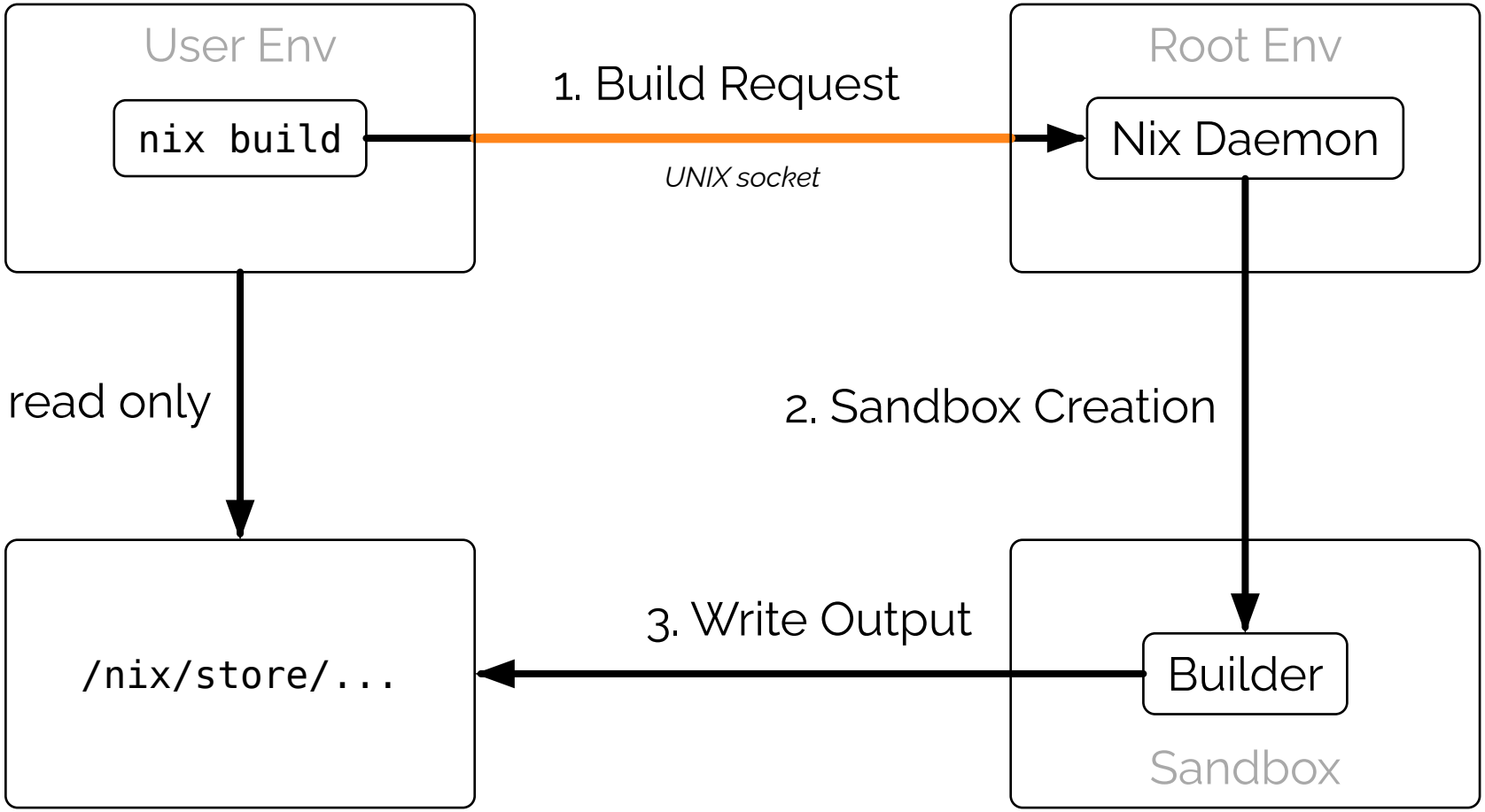
Add this line to `~/.config/nix/nix.conf` or `/etc/nix/nix.conf`:

```
experimental-features = nix-command flakes
```

NixOS

Add this to your NixOS configuration:

```
{
  nix.settings.experimental-features = [
    "nix-command" "flakes"
  ];
}
```


User Env
nix build
1. Build Request
UNIX socket
Root Env
Nix Daemon
read only
2. Sandbox Creation
/nix/store/...
3. Write Output
Builder
Sandbox

Basic Nix Commands

Command	Effect
`nix profile`	Manipulate Nix User Environment 📦
`nix shell` / `nix develop` (`nix-shell`)	Create a shell with a specific environment 🐚
`nix-collect-garbage`	Delete unreferenced store paths ♻️
`nix repl --file '<nixpkgs>'`	open a REPL with `pkgs` in scope 🔄
`nix build` (`nix-build`)	One-step evaluation and realization 🏭
`nix-instantiate`	Create derivation from nix expression
`nix derivation show`	Show a derivation 🔍
`nix-store -r`	Realize a derivation

Nix / NixOS Cheatsheet

Most Important Documentation Links

Nix Docs (Tool, Language)	https://nixos.org/manual/nix
Nixpkgs (Packaging, prog langs, library)	https://nixos.org/manual/nixpkgs
NixOS	https://nixos.org/manual/nixos
Nixpkgs package and NixOS module search engine	https://search.nixos.org

Imperative Package Management

Update package list	`apt update`	*happens automatically*
Search	`apt search <pkgname>`	`nix search nixpkgs <pkgname>`
Install	`apt install <pkgname>`	`nix profile install nixpkgs#<pkgname>`
Upgrade installed	`apt upgrade`	`nix profile upgrade '.*'`
List installed	`dpkg -l`	`nix profile list`
Remove	`apt remove <pkgname>`	`nix profile remove <list-number>`
Rollback	-	`nix profile rollback`

Per-Project Shells

Ad-hoc shell with packages	`nix shell nixpkgs#pkg1` or `nix shell nixpkgs#(pkg1,pkg2)`
Project-shell with flake	`nix develop`
Project-shell with `shell.nix` or `default.nix` file	`nix-shell`

Building Packages

Build `default.nix` or default pkg from flake	`nix build`
Build specific attributes (flakes)	`nix build .#pkg1 .#pkg2`

Input Management

Flakes

Init flake project	`nix flake init`
Init flake-parts project	`nix flake init -t github:hercules-ci/flake-parts`
Update flake inputs	`nix flake update`
Update and commit lock file	`nix flake update --commit-lock-file`
Update specific input	`nix flake lock --update-input <name>`

Niv (Pre Flakes)

Install Niv Files	`niv init`
Add GitHub Repository	`niv add <github-owner>/<reponame>`
Update inputs	`niv update`
Update specific input	`niv update <name>`
Switch branch/tag of input	`niv update <name> -b <gitref>`

Flake References

Flake in current Directory	`.`
Local path	`[path:]/path/to/repo`
HTTPS URL to flake tarball	`https://host/flake.tar.gz`
Git Repo via HTTPS	`git+https://host/repo`
Git Repo via SSH	`git+ssh://git@host/repo`
GitHub Repo	`github:owner/repo`
Specific Branch/Tag	`git+ssh://git@host/repo?ref=abc123`

Nix REPL

Start Nix REPL	`nix repl`
Load local Flake	`:lf .`
Build derivation	`:b attribute.with.derivation`
Build & Install derivation	`:i attribute.with.derivation`
Fully print expression	`:p some.expression`
Show documentation of builtin function	`:doc builtins.listToAttrs`
Show REPL help	`:?`

NixOS System Rebuild

Rebuild system and activate	`nixos-rebuild switch`
Rebuild system and activate for now without updating bootloader	`nixos-rebuild test`
Rebuild w/o activating, but update bootloader	`nixos-rebuild boot`
Rollback	`nixos-rebuild switch --rollback`
Build on host a, deploy to host b, authorize with sudo	`nixos-rebuild switch --build-host a --target-host b --use-remote-sudo`

Note: nixos-rebuild expects a flake `/etc/nixos/flake.nix` to exist, and within that flake, a `nixosConfiguration` attribute with the hostname of the current system.

Garbage Collection

Collect unreferenced nix store paths	`nix-collect-garbage`
Also collect old profile/system generations	`nix-collect-garbage -d`
Only delete up to 50GB	`nix-collect-garbage --max-freed 50G`
Find and link identical files	`nix-store --optimise`
Print all GC roots	`nix-store --gc --print-roots`

NIXCADEMY EXPERT EDUCATION

Professional Nix & NixOS Trainings
https://nixcademy.com/

https://nixcademy.com/cheatsheet

Nix	https://nixos.org/nix/manual
Nixpkgs	https://nixos.org/nixpkgs/manual
NixOS	https://nixos.org/nixos/manual

Nix commands, flakes, Nix language → Nix manual
Anything within `pkgs` → Nixpkgs manual
NixOS installation, configuration, modules, integration testing → NixOS manual

The Nixpkgs and NixOS manuals are not indexed by search engines because they are too large single-page documents. → Keep them handy with bookmarks! 📑

Nix Profile

```
$ nix profile install nixpkgs#hello
installing 'hello'

$ hello
Hello World

$ nix profile remove hello
uninstalling 'hello'
```

If a derivation's output contains binaries in its `$out/bin` folder, it's an installable package.

How Nix Profiles Work

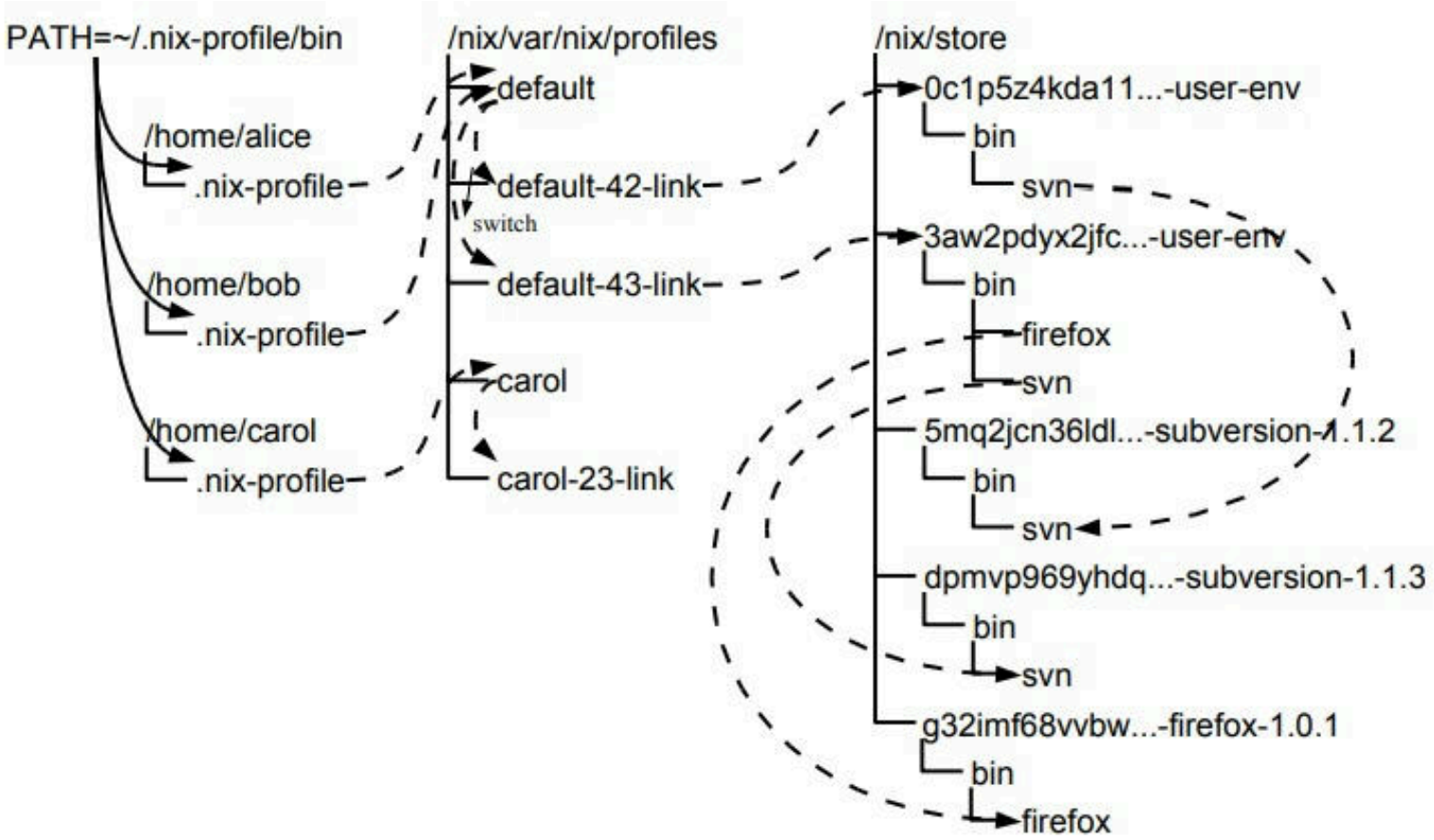


Figure 2.11.: User environments

Source: Nix PhD thesis

Listing & Upgrading

```
$ nix profile list
...

$ nix profile upgrade --all
```

Nix has its own garbage collection for the nix store.

```
$ nix-collect-garbage -d
...
removing old generations of profile /nix/var/nix/profiles/per-user/tfc/profile
removing generation 94
...
deleting '/nix/store/jbchpg8...ac8m56vxs6j67-rpn_calc'
deleting '/nix/store/qpavjqr...9hgcpv1j1g2zi-rpn_calc'
deleting '/nix/store/trash'
deleting unused links...
note: currently hard linking saves 3683.55 MiB
2642 store paths deleted, 3129.22 MiB freed
```

This deletes all derivation outputs in the nix store that are not directly or indirectly referenced on the system.

Nix Language 得

Why is Nix not just YAML?

Nix is like *JSON with functions*.

This language is very powerful and we will see why over the rest of this week.


JSON + functions
lambda calculus
EXPLAINING NIX LANG
imgflip.com
JAKE-CLARK.TUMBLR

Play with code in the REPL

```
$ nix repl --file '<nixpkgs>'
Added 20817 variables.
nix-repl> :b pkgs.hello

This derivation produced the following outputs:
  out -> /nix/store/rnxji3jf6fb0nx2v0svdqpj9ml53gyqh-hello-2.12.1
[1 copied (0.2 MiB), 0.0 MiB DL]
nix-repl>
```

- `:b x` builds derivation x and prints its output path(s)
- `:?` lists all colon-commands
- Note that the REPL has tab-completion
- (`nix repl nixpkgs` also works but is not the same)

Attribute sets are like Python dicts

```
nix-repl> x = 1

nix-repl> s = { a = 1; b = x; }

nix-repl> s
{ a = 1; b = 1; }

nix-repl> t = { inherit x; inherit (s) a b; }

nix-repl> t
{ a = 1; b = 1; x = 1; }
```

Forget what you knew about “inheritance” from OO languages

Beware: `or` is not `||`

```
nix-repl> true && (false || true)
true

nix-repl> set = { a = 1; b = 2; }

nix-repl> set ? a
true

nix-repl> set ? c
false

nix-repl> set.c or 10
10
```

NIXCADEMY EXPERT EDUCATION

Currying

```
nix-repl> plusOne = x: x + 1

nix-repl> plusOne 10
11

nix-repl> (x: x + 1) 10
11

nix-repl> f = a: b: a + b

nix-repl> f 1 2
3

nix-repl> plusOne' = f 1

nix-repl> plusOne' 10
11
```

Python:

```
> class Plus:
  def __init__(self, a):
    self.a = a

  def __call__(self, b):
    return self.a + b

> Plus(1)(2)
3
```

Nix:

```
> plus = a: b: a + b

> plus 1 2
3
```

Currying is just another way to bind local/member variables.

Nix Syntax: Attribute Set Parameters

```
nix-repl> f = { a, b }: a + b

nix-repl> f { a = 1; b = 2; }
3

nix-repl> g = { a ? 100, c ? 200, ... }: a + c

nix-repl> g { a = 1; b = 2; }
201

nix-repl> h = s@{ a, ... }: a + s.b

nix-repl> h { a = 1; b = 2; }
3
```

Nix Syntax: Variable Scopes

```
nix-repl> let x = 1; y = 2; in x + y
3

nix-repl> s = { a = 1; b = 2; }

nix-repl> with s; [ a b ]
[ 1 2 ]

nix-repl> let
            s = { a = 1; b = 2; c = 3; };
          in
            with s; [ a c ]
[ 1 3 ]
```

Nested `with` scopes

```
nix-repl> s1 = { a = 1; }

nix-repl> s2 = { a = 2; }

nix-repl> with s1; with s2; a
2

nix-repl> let
            a = 3;
          in
            with s1; with s2; a
3
```

This behavior can be surprising - use `with` sparingly!

```
nix-repl> x = 1
nix-repl> bla = "bla"

nix-repl> "x = ${builtins.toString x}"
"x = 1"

nix-repl> ''
            interpolated: ${bla}
            not interpolated: ''${bla}
          ''
"interpolated: bla\nnot interpolated: \${bla}\n"
```

Unions of Sets and Recursive Updates

```
nix-repl> { x = 1; } // { y = 2; }
{ x = 1; y = 2; }

nix-repl> { x = 1; z = 1; } // { y = 2; z = 3; }
{ x = 1; y = 2; z = 3; }

nix-repl> :p { x = { y = 1; }; } // { x = { z = 2; }; }
{ x = { z = 2; }; }

nix-repl> :p pkgs.lib.recursiveUpdate
                { x = { y = 1; }; }
                { x = { z = 2; }; }
{ x = { y = 1; z = 2; }; }
```

Nix Syntax: Import function

File: `x.nix`

```
1 + 1
```

File: `y.nix`

```
x: 1 + x
```

Nix REPL

```
nix-repl> import ./x.nix
2

nix-repl> import ./y.nix
«lambda @ /path/to/y.nix:1:1»

nix-repl> import ./y.nix 10
11

nix-repl> f = import ./y.nix

nix-repl> f 10
11
```

Trace values at evaluation time

```
nix-repl> x = 1

nix-repl> builtins.trace
            "value of x: ${builtins.toString x}"
            (x + x)
trace: value of x: 1
2
```

Why doesn't this fail?

```
let
  x = 1 / 0;
  y = [ (1 / 0) (2 / 0) ];
in
  if builtins.length y > 1
    then "all good!"
    else x
```

See also "Weak head normal form". (Look it up)

Let's play with this one in the REPL.

Head vs. Tail

```
nix-repl> mylist = [ 1 2 3 4 5 ]

nix-repl> builtins.length mylist
5

nix-repl> builtins.head mylist
1

nix-repl> builtins.tail mylist
[ 2 3 4 5 ]
```

What are the *types* of `head` and `tail`?

File: `fibonacci.nix`

```
let
  fib = x:
    if x == 0 then 0
    else if x == 1 then 1
    else fib (x - 1) + fib (x - 2);
in
  fib
```

Let's play with this one in the REPL.

To master the Nix syntax and iteration in purely functional programming once and forever, let's do **2** quick language exercises.

After that, we are done with learning this seemingly academic language and start doing very practical things.

Exercise Time: Write a Sum Function

NIXCADEMY EXPERT EDUCATION

Task:

- Write a function `sum` that accepts a list of numbers and returns the *sum*
- Put the implementation into a file, use it in the REPL
- Test cases:
  - `sum [ 1 2 3 ] = 6`
  - `sum [] = 0`

Notes:

- Functions from `builtins.*` that will be useful: `length`, `head`, `tail`.
- Solution: https://nixcademy.com/downloads/nix-functions-solutions.zip

Exercise Time: Write a Take-N Function

NIXCADEMY EXPERT EDUCATION

Task:

- Write a function `takeN` that returns only the first N items of a list
- Put the implementation into a file, use it in the REPL
- Test cases:
  - `takeN 3 [ 1 2 3 4 5 ] = [ 1 2 3 ]`
  - `takeN 3 [ 1 2 ] = [ 1 2 ]`
  - `takeN 3 [] = []`

Notes:

- Functions from `builtins.*` that will be useful: `length`, `head`, `tail`.
- Concatenate lists with `[ 1 ] ++ [ 2 ] = [ 1 2 ]`
- Solution: https://nixcademy.com/downloads/nix-functions-solutions.zip

Where Do Functions Come From?

Nix is a real language that is able to solve a big share of real-world algorithmic problems.

`builtins.*` Documented in the Nix Manual
`pkgs.lib.*` Documented in the Nixpkgs Manual

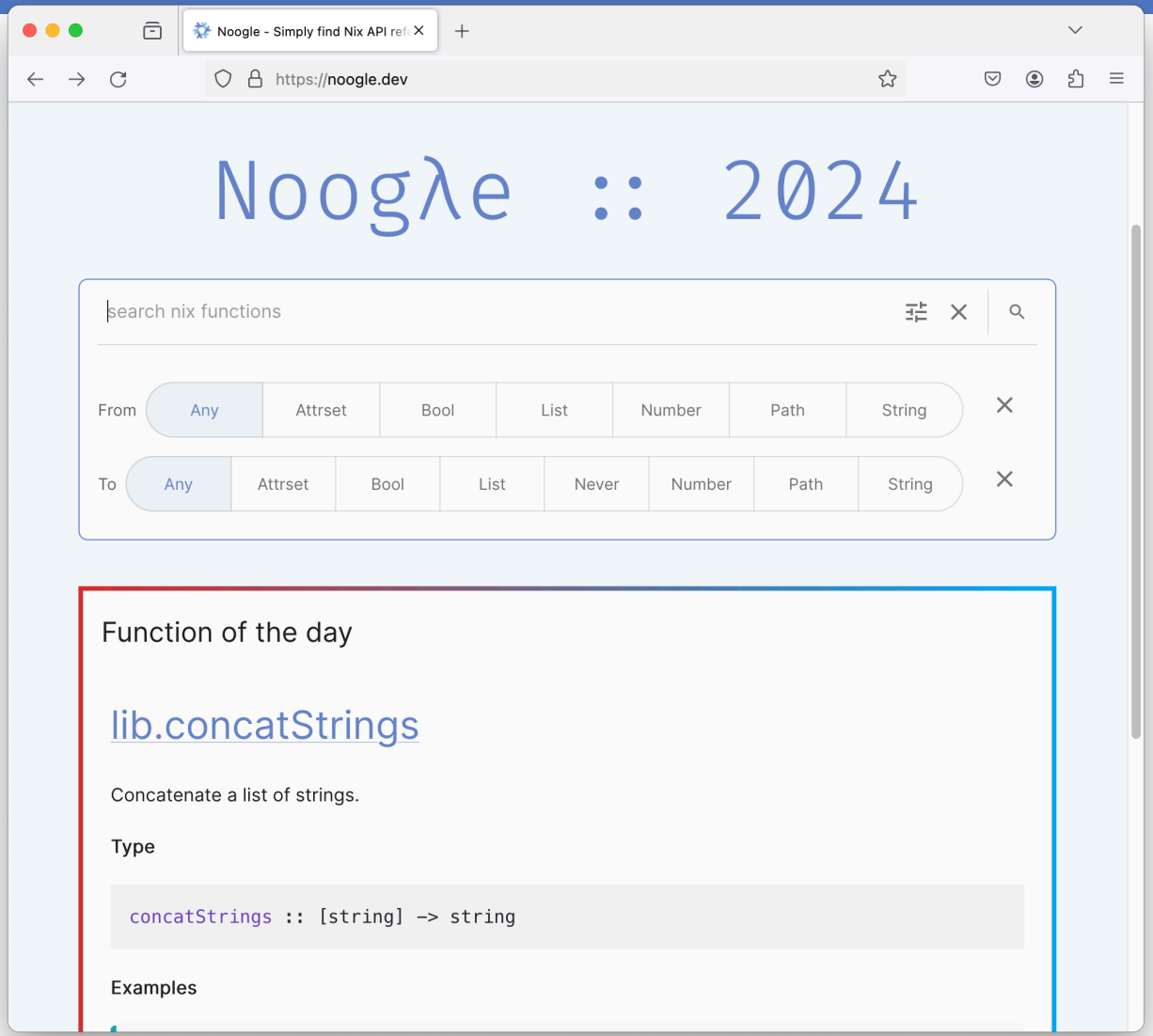
Nix developers should read at least once through these files:

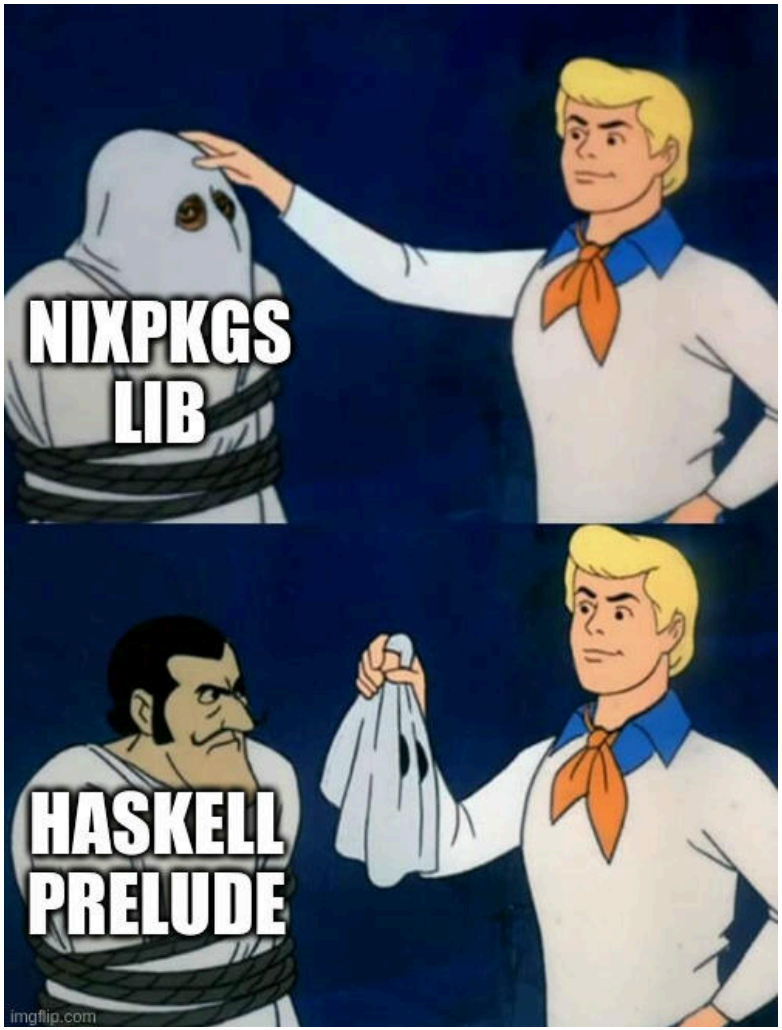
- `nixpkgs/lib/attrsets.nix`
- `nixpkgs/lib/lists.nix`
- `nixpkgs/lib/customisation.nix`

Many people don't and end up reimplementing algorithms. Nearly everything you need is there already!

Search for functions in `builtins.*` and `pkgs.lib.*`

- by name
- by type




NIXPKGS
LIB
HASKELL
PRELUDE
imgflip.com

Ways to import Nixpkgs:

```
pkgs = import ./path/to/nixpkgs { };
```

or

```
pkgs = import <nixpkgs> { };
```

or

```
pkgs = import (builtins.getFlake "nixpkgs") { };
```

NIXCADEMY EXPERT EDUCATION

Options for importing Nixpkgs

```
pkgs = import ./path/to/nixpkgs {
  # optional parameters:
  config = {
    allowBroken = true;
    allowUnfree = true;
    allowUnsupportedSystem = true;
    # e.g. a package contains meta.knownVulnerabilities ["CVE-XXXX-XXXX"]
    permittedInsecurePackages = [
      "openssl-1.2.3"
    ];
  };
};
```

See also Nixpkgs Manual, Chapter 2.6: Config Options Reference

Packaging Basics 📦

- Evaluation translates `.nix` expressions to `.drv` *derivations*
- A derivation is a machine-readable recipe without dynamic parts
- A *realized* derivation results in a read-only output path with build products: `/nix/store/<hash>-<packagename>`

Evaluation & realization phases

`.nix` ⟶ `.drv` ⟶ `/nix/store/<hash>-<packagename>`

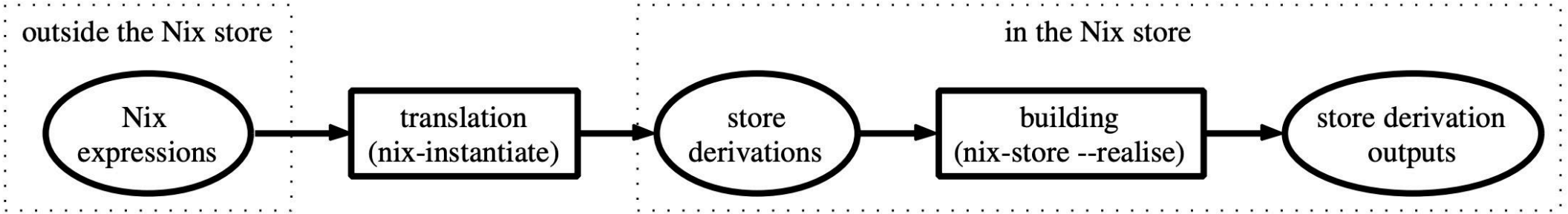
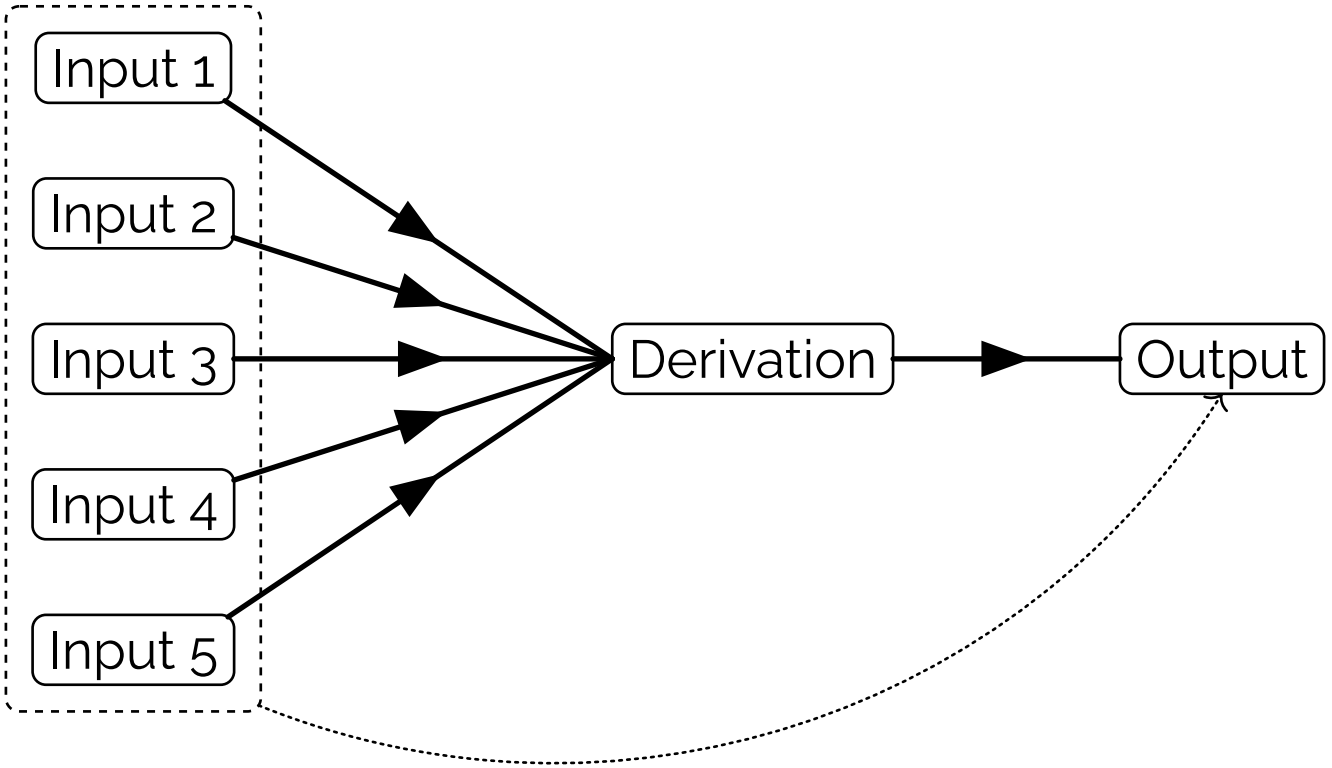


Figure 2.12.: Two-stage building of Nix expressions

from Nix PhD thesis


Input 1
Input 2
Input 3
Input 4
Input 5
Derivation
Output
Hash parameters for output path

A Minimal Nix Derivation

File: `builder.sh`

```
echo "foobar" > $out
```

File: `default.nix`

```
let pkgs = import <nixpkgs> { };
in
derivation {
  name = "myDerivation";
  builder = "${pkgs.bash}/bin/bash";
  args = [ ./builder.sh ];
  system = builtins.currentSystem;
}
```

Derive: `nix-instantiate default.nix`

Build: `nix-build default.nix`

Exercise Time: Make it Build!

Download 💾

https://nixcademy.com/downloads/mini-hello.zip

Task:

- Use `builtins.derivation` to package `mini-hello` (don't use `stdenv.mkDerivation`)
- Inspect `nix derivation show $(nix-instantiate)`
- Provide the binary in the `$out/bin/` folder and install it with `nix profile install <output path>`

Notes:

- Needed packages are: `gcc`, `gnumake`, `coreutils`
- All derivation attributes become **environment vars** at build time
- Solution: https://nixcademy.com/downloads/mini-hello-solution.zip

Let's have a look at the dependency tree of our app!

Print the dependency tree

```
$ nix-store --query --tree <store path>
```

Explain the difference between these two variants:

Run Time vs Compile Time Dependencies

```
$ nix-store --query --tree $(nix-build ...)
$ nix-store --query --tree $(nix-instantiate ...)
```

Also: Check out the command `nix why-depends <package> <dep>`

Copy Closure via SSH

```
$ nix copy --to ssh://server $(nix-build ...)
```

Copy *build* or *runtime* closure to other host and build or run the package there.

Pinning 📌

Pinning: Don't Flakes do this for us already?

Yes, but here we understand *how* it works, what the caveats are, and it's still relevant for individual packages.

`<nixpkgs>` is expanded with a path that is looked up from `NIX_PATH`

Content of the `NIX_PATH`

```
$ echo $NIX_PATH
nixpkgs=/nix/var/nix/profiles/per-user/root/channels/nixos:nixos-config=/
etc/nixos/configuration.nix:/nix/var/nix/profiles/per-user/root/channels

$ nix repl

nix-repl> <nixpkgs>
/nix/var/nix/profiles/per-user/root/channels/nixos/nixpkgs
```

- The nixpkgs from here are simply a git clone of the original repo
- See also Nix Manual, Chapter 18: Common Environment

Pinning expressions 📌

To make nix expressions build *forever*, it's possible to *pin* all the inputs.

```
$ cat nixpkgs.nix
builtins.fetchTarball {
  url = "https://github.com/nixos/nixpkgs/archive/bd4e35e14a0130268c8f7b253
a15e09d76ec95b7.tar.gz";
  sha256 = "0k9ffj84mq21w17jfdh0rcr9pcvnbmaq9y3r7yd9k1mvry708cq5";
}

$ nix repl
nix-repl> nixpkgsSrc = import ./nixpkgs.nix
nix-repl> pkgs = import nixpkgsSrc {}
```

- The hash references a specific Git commit
- If the commit is so old that no binaries are available on `cache.nixos.org`, Nix will simply rebuild the packages

Pinning app `npins` in Action

```
$ nix-shell -p npins
$ npins init
$ npins add github nixos nixpkgs -b nixos-unstable --name nixpkgs
$ npins add github nixos nixpkgs -b nixos-21.11 --name nixpkgs-old
$ nix repl

nix-repl> sources = import ./npins

nix-repl> pkgs = import sources.nixpkgs { }
nix-repl> pkgs.gcc.version
"14.3.0"

nix-repl> pkgsOld = import sources.nixpkgs-old { }
nix-repl> pkgsOld.gcc.version
"11.2.0"
```

Flakes in Action

```
$ nix flake init

$EDITOR flake.nix # need to edit manually

$ nix repl

nix-repl> :lf .
Added 9 variables.
nix-repl> inputs.nixpkgs.legacyPackages.x86_64-linux.gcc.version
"13.3.0"
nix-repl> inputs.nixpkgs-old.legacyPackages.x86_64-linux.gcc.version
"12.2.0"
```

Building Software - The Typical Dance 👯

unpack, configure, build, test, install

```
wget https://company.com/product-1.2.3.tar.gz
tar xf product-1.2.3.tar.gz
cd product-1.2.3
./configure
make
make test
make install
```

It's always similar, no matter the language, OS, etc.

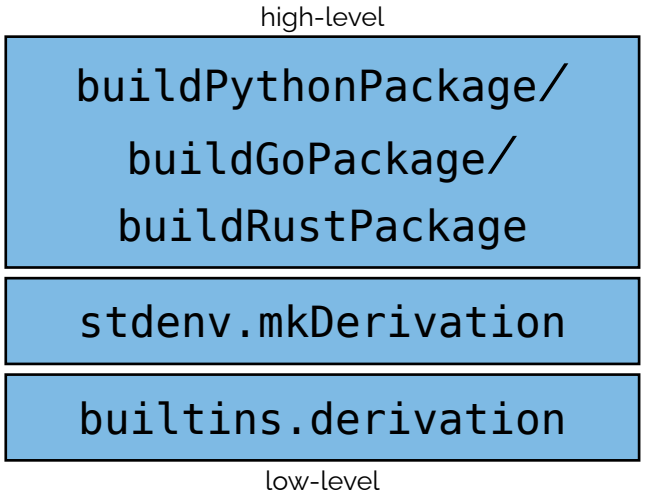
A Standard Derivation with mkDerivation

```
pkgs.stdenv.mkDerivation {
  name = "myPackage";
  src = ./.;
  # ...

  installPhase = ''
    # only necessary if there is no install target
    mkdir -p $out/bin
    cp myApp $out/bin/
  '';
}
```

See also Nixpkgs Manual, Section 6.1: Using stdenv

I need Go/Rust/Python builds, why are we looking at mkDerivation and C projects?

All the `buildGoModule`, `buildRustPackage`, `buildPythonPackage` functionality builds on top of `mkDerivation`. To master it (especially in the cases when we need to debug things), we need to understand `mkDerivation`'s flow.



Exercise Time - Build GNU hello 👋

Download 💾

https://gnu.mirror.constant.com/hello/hello-2.9.tar.gz

Goals:

- Create a `default.nix` and get it to build with `nix-build`
- Use `pkgs.stdenv.mkDerivation`
- Keep this folder, we will revisit it

The `stdenv` comes with the following dependencies by default:

- Bash
- C Compiler, configured with C and C++ support, (GNU on Linux, Clang on macOS)
- coreutils
- findutils
- diffutils, `patch`
- `patchelf`
- `awk`, `grep`, `sed`
- `tar`, `bzip2`, `gzip`, `xz`
- Make

All tools come as the GNU variants.

Dependencies are listed as parameters to `mkDerivation` in the following attributes:

`nativeBuildInputs`	compile time deps
`buildInputs`	run time deps
`propagatedBuildInputs`	run time deps of scripts
`checkInputs`	test deps

These are the most important ones.
See also Nixpkgs Manual, Section 6.3.1. Variables specifying dependencies

`mkDerivation` uses *phases*:

1. `unpackPhase`
2. `patchPhase`
3. `configurePhase`
4. `buildPhase`
5. `checkPhase`
6. `installPhase`
7. `fixupPhase`
8. `installCheckPhase`
9. `distPhase`

Each phase …

- … acts as a *hook* with a most-of-the-time useful default implementation
- … can be overriden if the default behavior doesn't apply for the use case
- … can be augmented with additional behavior using `pre<phase>` and `post<phase>`

We will learn how to customize or override these.

Enabling/Disabling Phases

- `dontUnpack = true`
- `dontPatch`
- `dontConfigure`
- `dontBuild`
- `doCheck`
- `dontInstall`
- `dontFixup`
- `doInstallCheck`
- `doDist`

The phases of…

`do*` `dont*`

…attributes are…

disabled enabled

❌ ✅

…by default

Fetching Code with `fetchurl`

```
stdenv.mkDerivation {
  name = "myPackage";

  src = pkgs.fetchurl {
    url = "ftp://alpha.gnu.org/gnu/sed/sed-4.2.2-pre.tar.bz2";
    sha256 = "11nq06d131y4wmf3drm0yk502d2xc6n5qy82cg88rb9nqd2lj41k";
  };

  # ...
}
```

The `src` attribute is a **fixed output derivation**

Let's update an existing package

```
stdenv.mkDerivation {
  name = "sed";

  src = pkgs.fetchurl {
-   url = "ftp://alpha.gnu.org/gnu/sed/sed-4.2.2-pre.tar.bz2";
+   url = "ftp://alpha.gnu.org/gnu/sed/sed-5.0.0.tar.bz2";
    sha256 = "11nq06d131y4wmf3drm0yk502d2xc6n5qy82cg88rb9nqd2lj41k";
  };

  # ...
}
```

What could go wrong?

Let's update an existing package

```
stdenv.mkDerivation {
  name = "sed";

  src = pkgs.fetchurl {
-   url = "ftp://alpha.gnu.org/gnu/sed/sed-4.2.2-pre.tar.bz2";
+   url = "ftp://alpha.gnu.org/gnu/sed/sed-5.0.0.tar.bz2";
    sha256 = "11nq06d131y4wmf3drm0yk502d2xc6n5qy82cg88rb9nqd2lj41k";
  };

  # ...
}
```

What could go wrong? → Yes, we need to change the **hash**, too!

(See also `pkgs.lib.fakeHash`)

Obtaining the Correct Hash

Trust on first use (TOFU) with `lib.fakeHash`

```
unpacking source archive /build/hello-2.9.tar.gz
error: hash mismatch in fixed-output derivation '/nix/store/....drv':
         specified: sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
            got:    sha256-1aFSTdB6F9qR8hch6QTZzQm5rgEhh1SbNIfFs61FsK8=
error: 1 dependencies of derivation '/nix/store/....drv' failed to build
```

Avoiding TOFU: Hash local file before upload

```
$ nix hash file /.../myfile.tar.gz
sha256-1aFSTdB6F9qR8hch6QTZzQm5rgEhh1SbNIfFs61FsK8=
```

There are many more fetchers:

- `fetchzip` - *auto unpack (any tarball format)*
- `fetchpatch` - *normalizes patches*
- `fetchgit` - *supports submodules*
- `fetchFromGitHub`
- `fetchFromGitLab`
- …

See also Nixpkgs Manual, Section 11: Fetchers

`pkgs.nix-prefetch-scripts` contains a number of scripts to help get hashes for many types of fetchers. More advanced prefetchers echo JSON to `stdout`.

```
$ nix-prefetch-git https://git.savannah.gnu.org/git/hello.git
…
{
  "url": "https://git.savannah.gnu.org/git/hello.git",
  "rev": "1a04646423406df85b4c41609e55eb8482806247",
  "date": "2025-07-25T12:56:49+01:00",
  "path": "/nix/store/xhkn46xbc4ci9rv4ww1if0q71ga3clyl-hello",
  "sha256": "1wz74gb0xsjlwfd7a783hbzp7m0zhx5yy8wjfr7jj3izj1fw5i95",
  "hash": "sha256-JcXCXZA/DilPdpIj70uHH9Rz/4IDHXWa41TqDtYj5/M=",
  "fetchLFS": false,
  "fetchSubmodules": false,
  "deepClone": false,
  "fetchTags": false,
  "leaveDotGit": false,
  "rootDir": ""
}
```

Exercise Time - Extend the GNU hello derivation 👋

Download 💾

https://gnu.mirror.constant.com/hello/hello-2.9.tar.gz

Goals:

- Write the configure, build, check and install phases yourself
- Use `mkDerivation`'s `patches = [ ... ];` parameter to customize this app.

Notes:

- You might need the `--prefix` parameter at configure time
- `make check` runs the unit tests
- use `diff -ru folder1 folder2 > my.patch` creates a patch file
- Solution: https://nixcademy.com/downloads/hello-solution.zip

Debugging Failing Builds

- `nix-build --keep-failed`
- `nix-build --debug`
- Set `NIX_DEBUG` in the derivation's attrs (see nixpkgs manual 6.4.1.1)
- Add `pkgs.breakpointHook` to `nativeBuildInputs` (see nixpkgs manual 16.4.)

Nixpkgs comes with 50 language-specific builders/frameworks:

Agda, Android, Astal, BEAM Languages (Erlang, Elixir & LFE), Bower, CHICKEN, Coq and coq packages, COSMIC, Crystal, CUDA, Cue (Cuelang), Dart, Dhall, D (Dlang), Dotnet, Emscripten, Factor, GNOME, Go, Gradle, Hare, Haskell, Hy, Idris, Idris2, iOS, Java, Javascript, Julia, lisp-modules, Lua, Maven, Nim, OCaml, Octave, Perl, PHP, pkg-config, Python, Qt, R, Ruby, Rust, Scheme, Swift, Tcl, TeX Live, Typst, Vim, Neovim

If you work with one of these languages/techstacks, *read* the whole chapter about it. It helps **a lot**. 📚

Manual: https://nixos.org/manual/nixpkgs/stable/#chap-language-support

Python Builder Example

```
pkgs.python3Packages.buildPythonPackage rec {
  pname = "mypackage";
  version = "1.0.0";

  pyproject = true;
  build-system = [ setuptools ];

  src = pkgs.python3Packages.fetchPypi {
    inherit pname version;
    sha256 = "08f...ef0";
  };

  doCheck = false;
}
```

See also Nixpkgs Manual, Python Specific Builder

Go Builder Example

```
pkgs.buildGoModule {
  pname = "mypackage";
  version = "1.2.3";

  src = pkgs.fetchFromGitHub {
    owner = "my-org";
    repo = "my-repo";
    rev = "v1.2.3";
    hash = "sha256-abc123...";
  };

  vendorHash = "sha256-def456...";
  # set to `null` for vendored deps
}
```

See also Nixpkgs Manual, Go Specific Builder

```
pkgs.rustPlatform.buildRustPackage {
  pname = "myproject";
  version = "1.0.0";

  src = ./.;
  cargoLock.lockFile = ./Cargo.lock;

  buildInputs = [ ... ];
}
```

See also Nixpkgs Manual, Rust Specific Builder

https://crane.dev

- Source fetching: Automatically done using a Cargo.lock file
- Pre-compiles Dependencies for Nix store
- Composable: Splits builds and tests into granular steps.
- Enables pure `clippy` use cases and much more

Usage:

Import `default.nix` via npins or `lib` attribute via flake.

Crane project function is similar to nixpkgs' `buildRustPackage`.

"Build Helpers" (formerly "trivial builders") are wrapper functions around `mkDerivation` that are helpful for recurring tasks like:

- Create a preconfigured nix shell
- Running simple scripts to generate package outputs
- Creating prepackaged standalone scripts
- Creating text files
- Batch-Symlinking of scattered paths

See also Nixpkgs Manual, Chapter 12: Build Helpers, and `nixpkgs/pkgs/build-support/trivial-builders/default.nix`

Create a development shell

```
pkgs.mkShell {
  packages = with pkgs; [
    hello
    jq
  ];
  inputsFrom = with pkgs; [
    somePackage
  ];
  shellHooks = ''
    export FOOBAR=123
  '';
}
```

Python Shell with Individual Python Versions

File: `shell.nix`

```nix
let
  pkgs = import <nixpkgs> { };
  myPython = pkgs.python313.withPackages (ps: with ps; [
    flask numpy requests
  ]);
in
  pkgs.mkShell {
    packages = [
      myPython                        # each of these can
      pkgs.python312.pkgs.black       # use a different
      pkgs.python314.pkgs.mypy        # python version.
    ];                                # -> no clash!
  }
```

The runCommand helper

```
let
  pkgs = import <nixpkgs> {};
  env = {
   nativeBuildInputs = [ pkgs.cowsay ];
  };
in
pkgs.runCommand "cowsay-output" env ''
  find "${pkgs.cowsay}/share/cowsay/cows" \
   -name "*.cow" \
   -exec cowsay -f {} "Hello Nix Users" >> $out \;
''
```

Package shell scripts

```
pkgs.writeShellApplication {
  name = "show-nixos-org";

  runtimeInputs = with pkgs; [ curl w3m ];

  text = ''
    curl -s 'https://nixos.org' | w3m -dump -T text/html
  '';
}
```

Most notable differences to `writeScript` et al:

- Runs `shellcheck` at build time
- `$out/bin/show-nixos-org` is *installable*

Join folder structures

```
pkgs.symlinkJoin {
  name = "myFavoriteApps";
  paths = with pkgs; [ hello cowsay fortune ];
}
```

Result

```
$ ls result/bin/
cowsay  cowthink  fortune  hello  rot  strfile  unstr
```

Synthesize folder structures

```
pkgs.symlinkJoin {
  name = "myLinkFarm";
  paths = [
    derivationWithTarballs1
    derivationWithTarballs2
    ...
  ];
  postBuild = ''
    sha256sum $out/* | sed 's|/nix/store/.*/||' > $out/hashes.txt
  '';
}
```

This derivation could now for example be served as-is by a webserver.

Project Patterns 🎨

Project Structure: Example Project

```
|
+-- a/
|   +- default.nix
|   +- ...
+-- b/
|   +- default.nix
|   +- ...
+-- c/
|   +- default.nix
|   +- ...
+-- nix/
|   +- overlay.nix
|   +- sources.nix
|   +- ...
+-- flake.nix
+-- default.nix
+-- release.nix
+-- shell.nix
```

- Either `flake.nix` and/or `release.nix` (or both)
- Each module in their own folder
  - Minimizes closures
  - Facilitates structure
- Sane default shell and default files
- Release/Flake file gives access to all build targets
  - All variants of binaries
  - Integration tests
  - Automatic linting, format checkers, etc.
  - Documentation

NIXCADEMY EXPERT EDUCATION

Example `a/default.nix` *package function*

```
{ stdenv , boost, openssl, staticBuild ?
false }:

stdenv.mkDerivation {
  name = "my-app";
  src = ./.;
  buildInputs = [
    boost
    openssl
  ];
  # ...
}
```

Usage: `a = pkgs.callPackage ./a/default.nix { };`

Quick & Dirty: `nix-build -E "(import <nixpkgs> {}).callPackage ./default.nix {}"`

How does `callPackage` even work?

Let's play with `callPackage`

```
nix-repl> f = { x, y, z }: x + y + z

nix-repl> myCallPackage = pkgs.lib.callPackageWith { x = 1; y = 10; }

nix-repl> myCallPackage f { z = 100; }
111

nix-repl> myCallPackage f { z = 100; x = 2; }
112
```

One more thing…

Let's redecide things after the fact

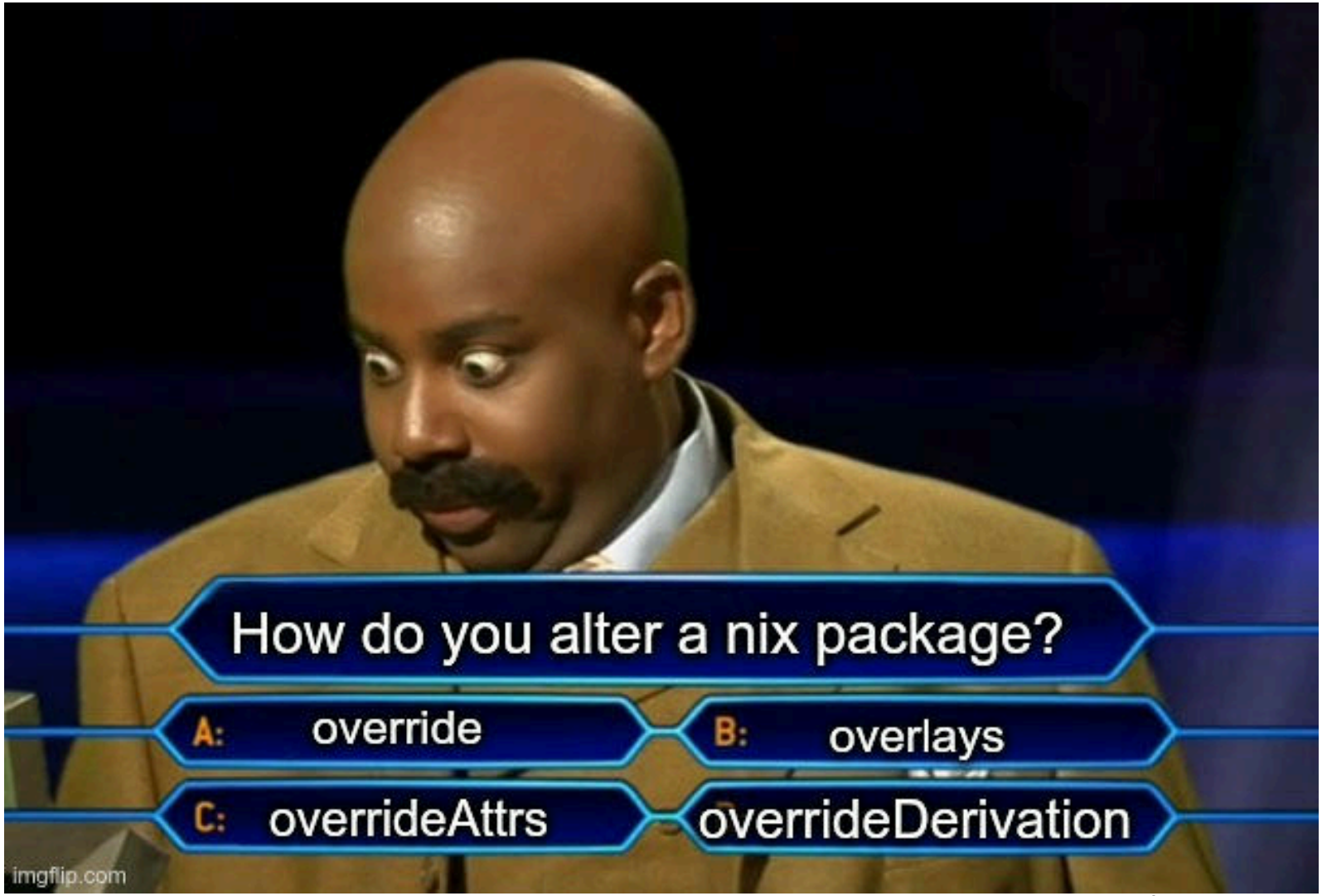
```
nix-repl> f = { x, y, z }: { result = x + y + z; }

nix-repl> myCallPackage = pkgs.lib.callPackageWith { x = 1; y = 10; }

nix-repl> foo = myCallPackage f { z = 100; }

nix-repl> foo.result
111

nix-repl> (foo.override { x = 2; }).result
112
```

File: `release.nix`

```
let pkgs = import <nixpkgs> {};
    a = pkgs.callPackage ./a/default.nix { };
in
{
  inherit a;

  a-static = a.override {
    staticBuild = true;
    openssl = pkgs.openssl.override { static = true; };
  };

  a-patchedversion = a.overrideAttrs (oldAttrs: {
    patches = oldAttrs.patches or [] ++ [
      ./patches/specialstuff.patch
    ];
  });
}
```

`callPackage` and `override` unlock huge potential!

When to use `override`/`overrideAttrs`?



Override

```
drv.override {
  x = y;
}
```

overrideAttrs

```
drv.overrideAttrs (oldAttrs: {
  patches = oldAttrs.patches or []
    ++ [ ./my.patch ];
})
```

There are many legitimate use cases for `overrideAttrs`.

Still, design your build functions so that callers don't need `overrideAttrs` for *everything*.

```
{ stdenv , packageA , packageB }:
```
← override changes these

```
stdenv.mkDerivation
{
  name = "my-app";
  src = ./.;
  buildInputs = [
    packageA
    packageB
  ];
  # ...
}
```
← overrideAttrs changes these

release.nix - Selective Builds

File: `release.nix`

```
{
  package1 = ...;
  package2 = ...;
  package3 = ...;
}
```

`nix-build release.nix`	Builds all derivations in parallel
`nix-build release.nix -A package2`	Builds only package2

Download 💾

https://gnu.mirror.constant.com/hello/hello-2.9.tar.gz

Goals:

- Rewrite your `default.nix` to the *normal* form
- Write a `release.nix` that provides different versions of `hello`:

`default`	Default variant, no patches
`hello-clang`	Built with Clang
`hello-patched`	Patched to print “Hello, Nix!”

Notes:

- Use `pkgs.clangStdenv` (Linux), `pkgs.gccStdenv` (macOS) or play with `pkgs.overrideCC`
- Solution: https://nixcademy.com/downloads/hello-release-solution.zip

The `with` Anti-Pattern

NIXCADEMY EXPERT EDUCATION

bad example

```
with import <nixpkgs> { };
with lib;

stdenv.mkDerivation {
  # ...
  buildInputs = foo [ a b c ];
}
```

good example

```
let
  pkgs = import <nixpkgs> { };
  inherit (pkgs) stdenv a b c;
  inherit (pkgs.lib) foo;
in
stdenv.mkDerivation {
  # ...
  buildInputs = foo [ a b c ];
}
```

- Unclear scope, clash probability
- Where do symbols come from?
- Evaluation performance penalty

bad example

```
rec {
  x = 1;
  y = 2;
  z = x + y;
}
```

good example

```
let
  x = 1;
  y = 2;
in
{
  inherit x y;
  z = x + y;
}
```

Imagine this in large nix expressions

bad example

```
stdenv.mkDerivation rec {
  pname = "bla";
  version = "1.2.3";

  src = someFetcher {
    url = "...${version}...";
    hash = "...";
  };
}
```

good example

```
stdenv.mkDerivation (finalAttrs: {
  pname = "bla";
  version = "1.2.3";

  src = someFetcher {
    url =
      "...${finalAttrs.version}...";
    hash = "...";
  };
})
```

Accidental implicit singleton pattern

bad example

```
# file: foo.nix
{ pkgs ? import <nixpkgs> {}
, other
, parameters
}:

...
```

bad example

```
# File: `release.nix`
let
  pkgs = import ... {
    config.overlays = ...;
  };
  x = import ./foo.nix { };
in
{
  inherit (x) a b c;
}
```

Debugging experience: “Why does it always pick the wrong version of my inputs? I mean, I overrode it in an overlay, right?”

The callPackage on Non-Derivations Anti-Pattern

bad example

```nix
# File: `foo.nix`
{ foo
, bar
, callPackage
}:
{
  x = callPackage ./x { };
  y = callPackage ./y { };
  ...
}
```

bad example

```nix
# File: `release.nix`
let
  pkgs = import ... { };
in
{
  # next line: bad
  foo = pkgs.callPackage ./foo.nix { };
}
```

Only use `callPackage` on *actual derivations*

See also `pkgs.callPackages`

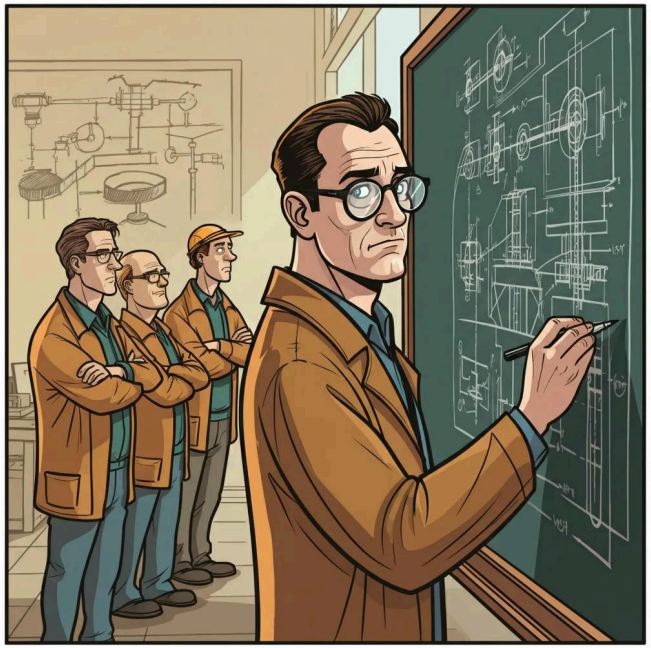
File: `generate-nix-expr.nix`

```
{ runCommand }:
runCommand "gen-nix-expr" {} ''
  mkdir $out
  sleep 5 # simulate slow generation
  cat << EOF > $out/hello.nix
  { stdenv, hello }:
  stdenv.mkDerivation {
    name = "hello";
    src = hello.src;
  }
  EOF
''
```

File: `default.nix`

```
let
  pkgs = import <nixpkgs> {};

  hello = pkgs.callPackage
    ./generate-nix-expr.nix
    {};
in

pkgs.callPackage
  "${hello}/hello.nix"
  {}
```

Change the `sleep` duration and watch the weird caching behavior.



Nix IFD: A Ticking Time Bomb in Your Build Pipeline?
https://nixcademy.com/posts/what-is-ifd-ups-and-downs/

Advanced Packaging Topics 🚀

What is wrong here?

bad example

```
stdenv.mkDerivation {
  pname = "bla";

  src = ./.;

  # ...
}
```

Bad Practise: Unfiltered Source Folders

Everything is copied into the nix store:

- … all the Nix expressions
- … README files
- … build artifacts from local work
- … VCS metadata directories (for example: `.git/`)
- … editor state/backup files
- … `result` symlinks from previous builds (!!)
- The name of your folder becomes part of the nix store copy, too

Bad Practise: Unfiltered Source Folders

Everything is copied into the nix store:

- … all the Nix expressions
- … README files
- … build artifacts from local work
- … VCS metadata directories (for example: `.git/`)
- … editor state/backup files
- … `result` symlinks from previous builds (!!)
- The name of your folder becomes part of the nix store copy, too

With source filters, we get:

- More cache hits
- Fewer rebuilds
- Fewer and smaller source copies into the nix store

Source Filtering: cleanSource

`pkgs.lib.cleanSource`

Drops VCS files, `*.o` artifacts, editor backup/swap files, `result` symlinks

Example Usage

```
src = lib.cleanSource ./.;
```

The `pkgs.lib.fileset` library

Example: https://github.com/tfc/pprintpp

```
src = lib.fileset.toSource {
  root = ./.;
  fileset = lib.fileset.unions [
    ./include
    ./test
    ./example
    ./CMakeLists.txt
    ./pkg-config.pc.cmake
  ];
};
```

Only copy source files with certain file types

Example: `unions` + `hasExt`

```
let
  fs = lib.fileset;
  extensionOf = extensions: file:
    builtins.any file.hasExt extensions;
in
fs.toSource {
  root = ./.;
  fileset = fs.unions [
    ./.latexmkrc
    (fs.fileFilter (extensionOf [ "jpg" "pdf" "png" "tex" ]) ./.)
  ];
}
```

Subtract one file list from the other

Example: Composing filters

```
let
  fs = lib.fileset;
  nixFiles = fs.fileFilter (file: file.hasExt "nix") ./.;
in
fs.toSource {
  root = ./.;
  fileset = fs.difference (fs.gitTracked ./.) nixFiles;
}
```

Source filters can be *traced*:

Example: Composing filters

```
src = let
  fs = lib.fileset;
  nixFiles = fs.fileFilter (file: file.hasExt "nix") ./.;
in
fs.toSource {
  root = ./.;
  fileset = fs.traceVal (fs.difference ./. nixFiles);
}
```

Remember: Complex source filtering *may* be indicator of clumsy project structure.

Nixpkgs Manual: `lib.fileset` file set functions

Implementation: `<nixpkgs/lib/fileset>`

Cross compilation 🔀

Importing nixpkgs with crossSystem for aarch64 builds

```
pkgs = import <nixpkgs> { };
systems = pkgs.lib.systems.examples;

aarch64Pkgs = import <nixpkgs> {
  crossSystem = systems.aarch64-multiplatform;
};
```

NIXCADEMY EXPERT EDUCATION

```
{ stdenv, cmake, openssl }:

stdenv.mkDerivation {
  name = "myproject";
  src = ...;

  nativeBuildInputs = [ cmake ];
  buildInputs = [ boost openssl ];

  # ...
}
```

Runs on build host, compiles for target

Runs on build host

Compatible with target

strictDeps: Disable legacy mode compatibility

- Many packages exist that mix up `nativeBuildInputs` and `buildInputs`.
  They don't cross-compile well.
- For this reason, `mkDerivation` puts both of them in `PATH`
- Use `strictDeps = true;` inside your `mkDerivation` call to disable this
- To enforce strict behavior on all packages, use the nixpkgs configuration option `strictDepsByDefault` (link to docs)

Just use `pkgsCross`

```
pkgs = import <nixpkgs> { };

windowsPkgs = pkgs.pkgsCross.mingwW64
```

pkgs exist for all attribute sets from `pkgs.lib.systems.examples`

Nixpkgs's crossSystem Parameter: Static Linking

NIXCADEMY EXPERT EDUCATION

Importing nixpkgs with isStatic

```
pkgs = import <nixpkgs> { };

staticPkgs = import <nixpkgs> {
  crossSystem = {
    isStatic = true;
    # ... Select musl ...
  };
};
```

Just Use pkgsStatic

```
pkgs = import <nixpkgs> { };

staticPkgs = pkgs.pkgsStatic
```

How Do pkgsCross/pkgsStatic Work?

Stdenv runs with different settings:

- Cross-Compiler
- compiler default flags (`NIX_CFLAGS`)
- configure default flags for autoconf, cmake, meson, etc.
- `glibc` substitute
- stdenv carries flags like `isStatic`

Most of the time these settings work. Some packages need tweaks.

See also

`nixpkgs/pkgs/top-level/stage.nix`

`nixpkgs/pkgs/stdenv/adapters.nix`

Many build system descriptions are hacked to work on specific systems but sacrifice *portability*.

General Advice:

- Do not mix dependency management with build management. Typical smells of non-portable projects:
  - Build system downloads and installs deps by itself
  - Build system looks for hardcoded compiler paths/names/versions
  - Subshells call build systems of git submodules
  - Build system sets static flags itself instead of providing mechanisms for both dynamic and static builds
- Know your tools, write portable build descriptions
- Keep your projects modular

Blog: More About Cross Compilation

Separation of Concerns in Cross-Compilation
https://nixcademy.com/posts/cross-compilation-with-nix/

Example - This stops the "Java Duke" icon from jumping in macOS when using `plantuml` in the terminal:

`makeWrapper` example

```
let
  env = { nativeBuildInputs = [ pkgs.makeWrapper ]; };
  headless-jre = pkgs.runCommand "headless-jre" env ''
    mkdir -p $out/bin
    makeWrapper ${pkgs.jre}/bin/java $out/bin/java \
      --add-flags "-Djava.awt.headless=true"
  '';
in
pkgs.plantuml.override { jre = headless-jre; }
```

The best documentation of this tool is in its source code:
`nixpkgs/pkgs/build-support/setup-hooks/make-wrapper.sh`

There are some advanced `sed/awk`-like facilities:

substitute script

```
export myVar=123

substitute ./in.txt ./out.txt \
  --replace-fail "foo bar" "baz qux" \
  --subst-var myVar \
  --subst-var-by otherVar lol
```

File: `in.txt`

```
foo bar
@myVar@
@otherVar@
```

File: `out.txt`

```
baz qux
123
lol
```

More Substitution Helpers

```
substituteInPlace a.txt b.txt c.txt \
  --replace-fail a b

# replace @x@ and @y@
export x=1
export y=2

substituteAll in.txt out.txt

substituteAllInPlace a.txt b.txt c.txt
```

There is no `/usr/bin/{env,bash,python,...}` in the Nix sandbox.

All *executable* package outputs are patched automatically in the install phase. If you need to execute scripts in earlier phases, use:

Patching Shebang Lines Example

```
preBuild = ''
  patchShebangs ./scripts/
  ./scripts/foo.sh
'';
```

Patching `/usr/bin/env xxx` works if `xxx` is part of the build inputs.

More Examples Online 1/3

https://galowicz.de/2023/01/23/mixed-cpp-monorepo-project/

- Organizing multiple C++ modules in one big project (CMake)

https://github.com/tfc/meson_cpp_example

- Organizing multiple C++ modules in one big project (Meson)

https://github.com/tfc/pprintpp

- The pprintpp C++ library, tested with multiple compilers

More Examples Online 2/3

https://galowicz.de/2023/01/16/cpp-qt-qml-nix-setup/

- A QtQuick C++ Project with Nix

https://galowicz.de/2019/04/17/tutorial_nix_cpp_setup/

- Building a multi-dependency project with the cartesian product of compilers and input library versions

https://github.com/cyberus-technology/hedron

- OS kernel implemented in C++, built with multiple compilers

https://github.com/tfc/pandoc-drawio-filter

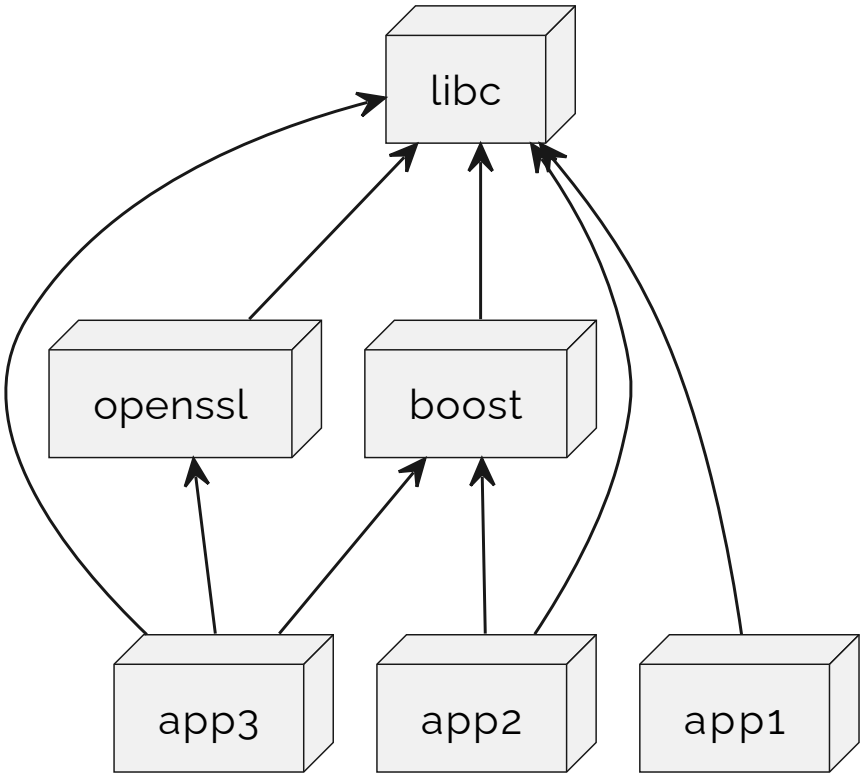
- Pandoc filter for draw.io images, includes nice self-test

https://github.com/tfc/electron-purescript-example

- Node.js + PureScript + Electron Example

Overlays 🔁

How to Patch Deps of multiple Packages?



`override` manipulates the inputs of one package.

How can we override a package for *all* downstream consumers?

What can overlays do?

- Add packages
- Update/Change packages
- All changes *can* affect pre-existing packages all along

Example Overlay file `overlay.nix`

```
final: prev: {
  app = prev.callPackage ./app { };
}
```

NIXCADEMY EXPERT EDUCATION

Example Overlay file `overlay.nix`

```
final: prev: {
  app = prev.callPackage ./app { };
}
```

Example Evaluation

```
nix-repl> pkgs = import <nixpkgs> {
  overlays = [ (import ./overlay.nix) ];
}

nix-repl> :b pkgs.app
```

Example Overlay file `overlay.nix`

```
final: prev: {
  opencv4 = prev.opencv4.override {
    enableGtk3 = true;
    enableTesseract = true;
  };
}
```

All packages that depend on `pkgs.opencv4` are now affected by this change.

Example Overlay file `overlay.nix`

```
final: prev: {
  opencv4-xxx = final.opencv4.override {
    enableGtk3 = true;
    enableTesseract = true;
  };
  myApp = prev.myApp.override {
    opencv4 = final.opencv4-xxx;
  };
}
```

Only `myApp` is affected. `opencv4-xxx` is also prepared for *further* overrides.

Note `prev` vs. `final` in this example!

How does it work? Enter Fix Point Recursion

Definition of `pkgs.lib.fix`

```
fix = f: let x = f x; in x;
```

How does it work? Enter Fix Point Recursion

Definition of `pkgs.lib.fix`

```
fix = f: let x = f x; in x;
```

```
nix-repl> f = self: {
  x = 1;
  y = self.x + 1;
  z = self.y + 1;
}
```

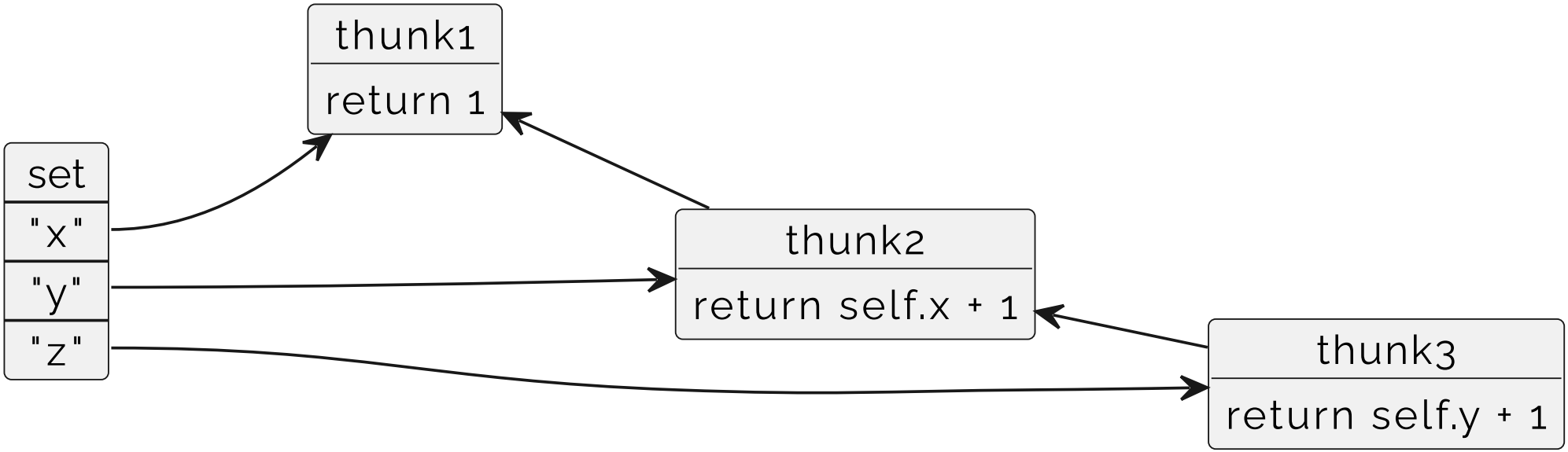
How does it work? Enter Fix Point Recursion

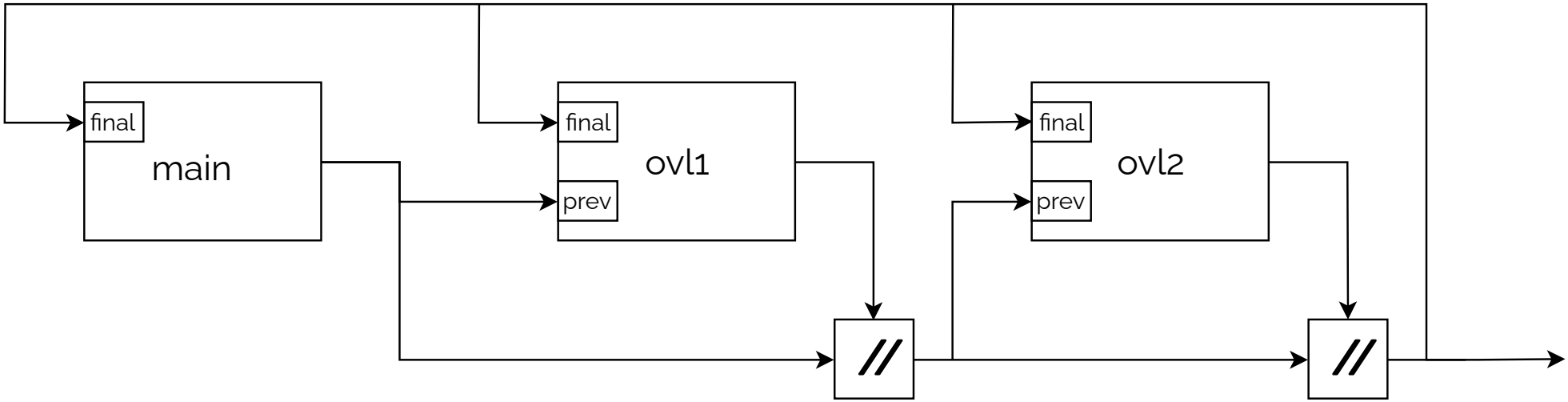
NIXCADEMY EXPERT EDUCATION

Definition of `pkgs.lib.fix`

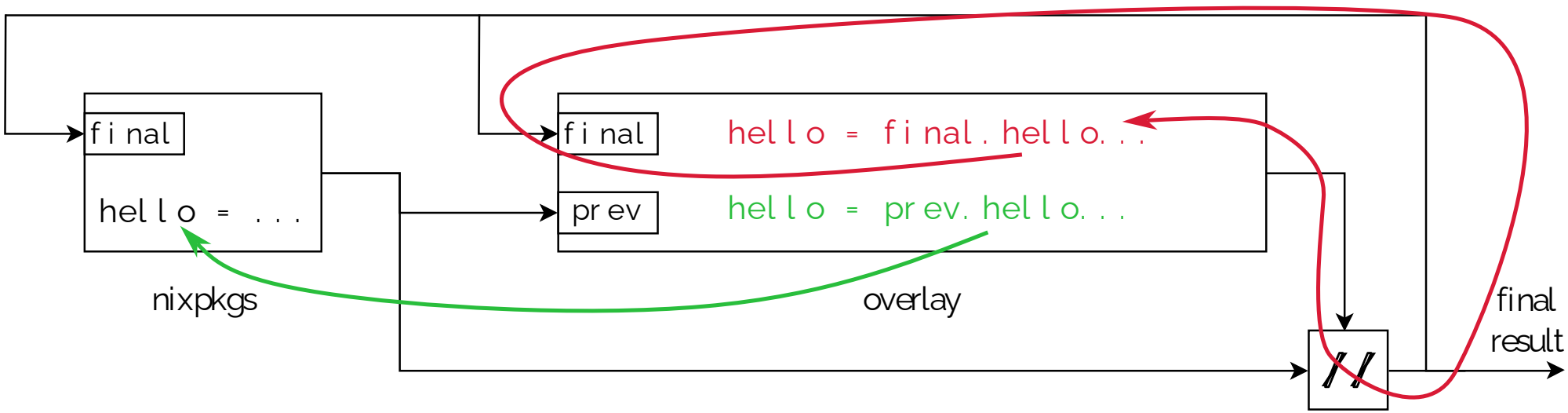
```
fix = f: let x = f x; in x;
```

```
nix-repl> f = self: {
  x = 1;
  y = self.x + 1;
  z = self.y + 1;
}

nix-repl> pkgs.lib.fix f
{ x = 1; y = 2; z = 3; }
```


thunk1
return 1
set
"x"
"y"
"z"
thunk2
return self.x + 1
thunk3
return self.y + 1


final
main
final
ovl1
prev
final
ovl2
prev
//
//

Infinite Recursion - How does it happen in overlays?



The Graft-Preserving Rules Flavor:

- Use `prev` by default.
- Use `final` only if you reference a package/derivation from some other package.

The Everything-Is-Overridable Rules Flavor:

- Use `final` by default.
- Use `prev` if you override a symbol to avoid infinite recursion.

Either ruleset is fine. Be consistent within your project/org.

Composing overlays

```
ovl1 = final: prev: { ... };
ovl2 = final: prev: { ... };

combined1 = pkgs.lib.composeExtensions ovl1 ovl2;

combined2 = pkgs.lib.composeManyExtensions
              [ ovl1 ovl2 ];
```

See also `nixpkgs/lib/fixed-points.nix`

Get the Git repo 💾

```
$ git clone  https://github.com/nixcademy/override-overlay-exercise
```

Tasks:

1. Use `override` & `overrideAttrs` to patch “abcd” to “xyzw”
   (only change `release.nix`)
2. Create an overlay file `overlay-patched.nix`, include it in the nixpkgs import, achieve the same effect with it.

Notes

- You can use `substituteInPlace` to patch return values of functions.
- Solutions are in the git branches `solution-1` and `solution-2`

More about Overlays Online

(Thread on x.com)

Nicolas is the inventor of nixpkgs overlays

https://nixcademy.com/posts/mastering-nixpkgs-overlays-techniques-and-best-practice/

Lazy Evaluation in Fixpoint Recursion
https://nixcademy.com/posts/what-you-need-to-know-about-laziness/

Flakes ❄

- 🚀 Cached Evaluation
- Concept of *Evaluation Closure*
- Hermetic *Evaluation* (no `NIX_PATH`, no `builtins.currentSystem`, …)
- Registry instead of Channels
- Built-in Pinning and overriding of Evaluation Inputs
- Nice tooling for attribute discovery
- Introduction of (opinionated) structure enforcement

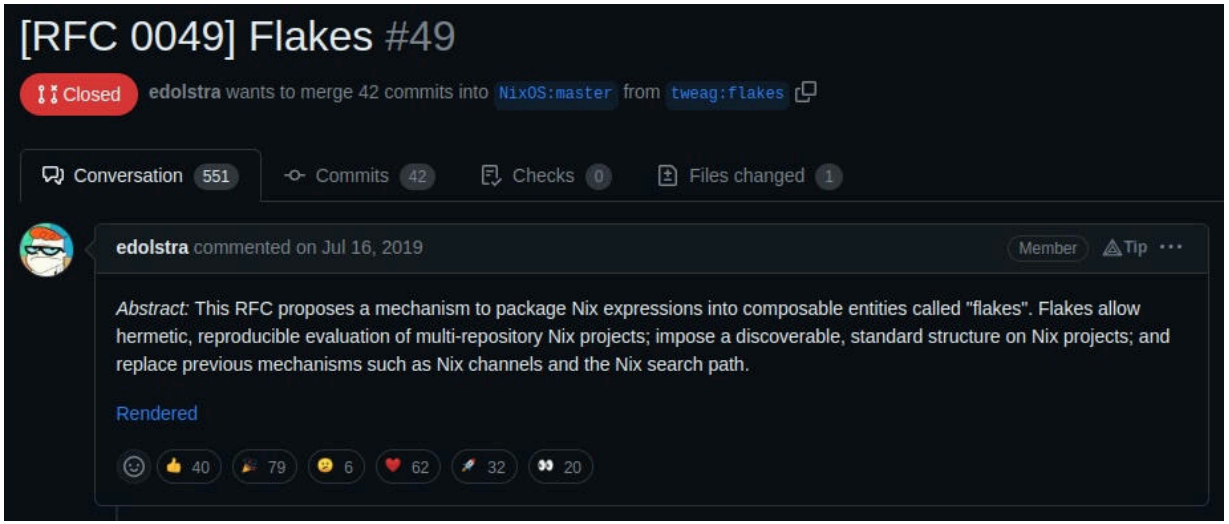
Example: Composable nix shells

```
nix shell nixpkgs#{nodejs,ruby}

nix shell \
    nixpkgs#ruby \
    github:nixos/nixpkgs/nixos-20.03#nodejs
```

Example: Run apps from the internet w/o checkout

```
nix run github:nix-community/nixos-anywhere
```

Experimental State? It's Complicated... 🙈 🙉 🙊



Lunch at NixCon October *2022* in Paris 🥐 🥖 :
Jacek asked Eelco "When will Flakes be stable and enabled by default?"
Eelco: "Probably something around 3 months." 🤷

However, Flakes are very usable and bring some nice benefits.

Non-NixOS

Add this line to `~/.config/nix/nix.conf` or `/etc/nix/nix.conf`:

```
experimental-features = nix-command flakes
```

NixOS

Add this to your NixOS configuration:

```
{
  nix.settings.experimental-features = [
    "nix-command" "flakes"
  ];
}
```

A Minimal Example Flake

File: `flake.nix`

```
{
  description = "A flake for building Hello World";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-23.11";

  outputs = { self, nixpkgs }: let
    pkgs = nixpkgs.legacyPackages.x86_64-linux;
  in {
    packages.x86_64-linux.default = pkgs.hello;

    devShells.x86_64-linux.default = pkgs.mkShell {
      packages = [ pkgs.something ];
    };
  };
}
```

File: `flake.nix`

```
{
  description = "Flake utils demo";

  inputs.flake-utils.url = "github:numtide/flake-utils";

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let pkgs = nixpkgs.legacyPackages.${system}; in
      {
        packages.default = pkgs.hello;
      }
    );
}
```

Potential anti-pattern: Flake-utils

Flake-utils can be considered an anti-pattern for mostly these reasons:

- It's easy to implement yourself
- Flakes end up depending on 10 different versions of flake-utils transitively
- They "hide" which platforms are supported

1000 Instances of flake-utils
https://nixcademy.com/posts/1000-instances-of-flake-utils/

File: `flake.nix`

```
{
  description = "Flake.parts example";

  inputs.flake-parts.url = "github:hercules-ci/flake-parts";

  outputs = inputs@{ flake-parts, ... }:
    flake-parts.lib.mkFlake { inherit inputs; } {
      systems = [
        "x86_64-linux" "aarch64-linux" "aarch64-darwin" "x86_64-darwin"
      ];
      perSystem = { config, pkgs, system, ... }: {
        packages.default = pkgs.hello;
      };
    };
}
```

```
inputs = {
  nixpkgs.url = "github:NixOS/nixpkgs/nixos-23.11";

  some-flake.url = "git+https://some.git.com/repo";
  some-flake.inputs.nixpkgs.follows = "nixpkgs";

  other-flake.url = "git+ssh://git@github.com/....git";
  nixpkgs-other.follows = "other-flake/nixpkgs";
};
```

`nix flake update` updates inputs and recreates the `flake.lock` lock file 🔒

no extra `npins` tooling needed 📌

See also Nix Manual, Chapter 7.5.1 nix flake

Flakes Export Different Kinds of Outputs

`packages`	Buildable Packages: `nix build flake#myPkg`
`apps`	Runnable apps: `nix run flake#myApp`
`nixosModules`	Importable NixOS modules
`nixosConfigurations`	Deployable NixOS configs: `nixos-rebuild --flake flake#myConfig --target root@host`
`checks`	Tests and checks: `nix flake check`
`overlays`	Overlay functions
`devShells`	Development shells: `nix develop flake#myShell`

Get the Git repo 💾

```
$ git clone  https://github.com/nixcademy/override-overlay-exercise
```

Task:

- Let's port `release.nix` together to `flake.nix`

Notes:

- In git repos, nix flake commands don't see files without prior `git add`
- apps attrs look like this:
  `apps.x86_64-linux.myapp = { type = "app"; program = "/exec/path"; }`
- Solutions in branches `solution-flake`

Get the Git repo 💾

```
$ git clone  https://github.com/tfc/nixery
```

- Flakified version of https://github.com/tazjin/nixery
- Original author is not a fan of flakes
- Works on GNU/Linux and macOS

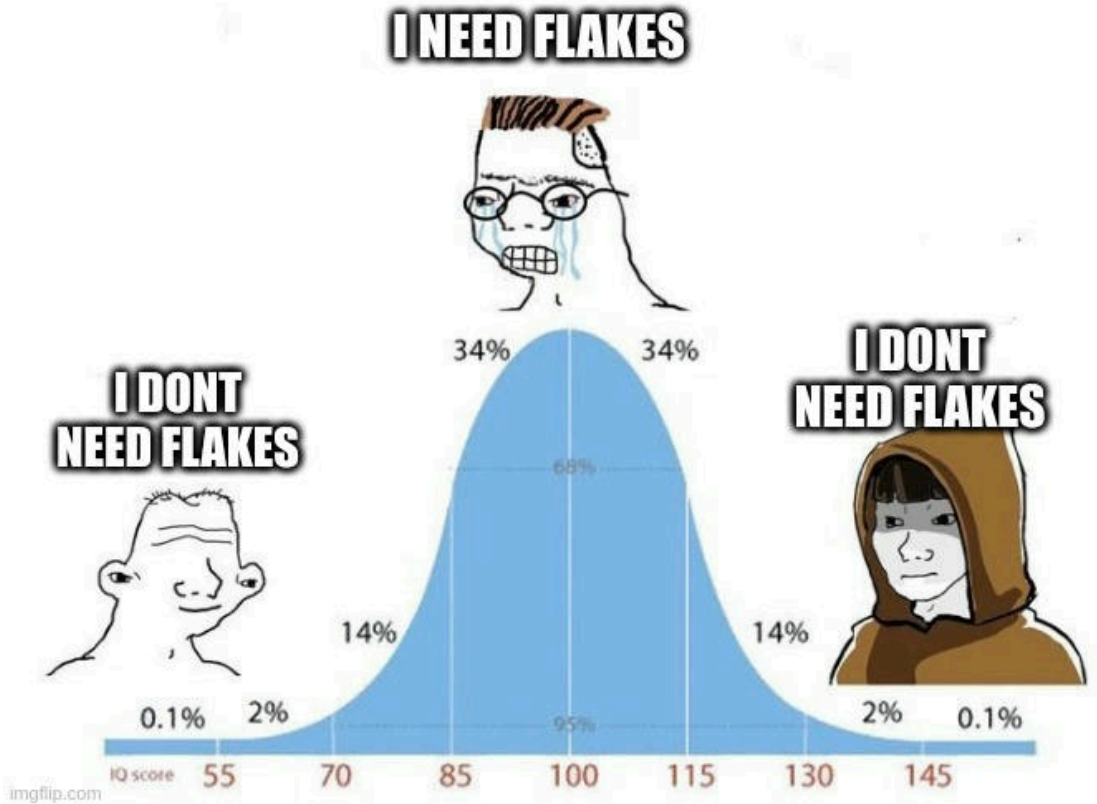
Get the Git repo 💾

```
$ git clone  https://github.com/tfc/rust_async_http_code_experiment
```

- Uses https://crane.dev
- Cached dependencies (*)
- Optional static linking
- Cross-compilation
- Pre-Commit checks
- Automatic dependency security audit
- Documentation Package
- Works on GNU/Linux and macOS

Flake tradeoffs & disadvantages

- Flakes are slow in huge monorepos
- Flakes are Git-aware (`+/-`)
- Flakes cannot be parameterized
- Flakes hardcode supported architectures
- `inputs.xxx.follows` is weaker than overlays



Section: NixOS

About NixOS

Installing NixOS

NixOS Modules

NixOS Configuration Examples

The `nixos-rebuild` Command

NixOS Containers

NixOS Patterns

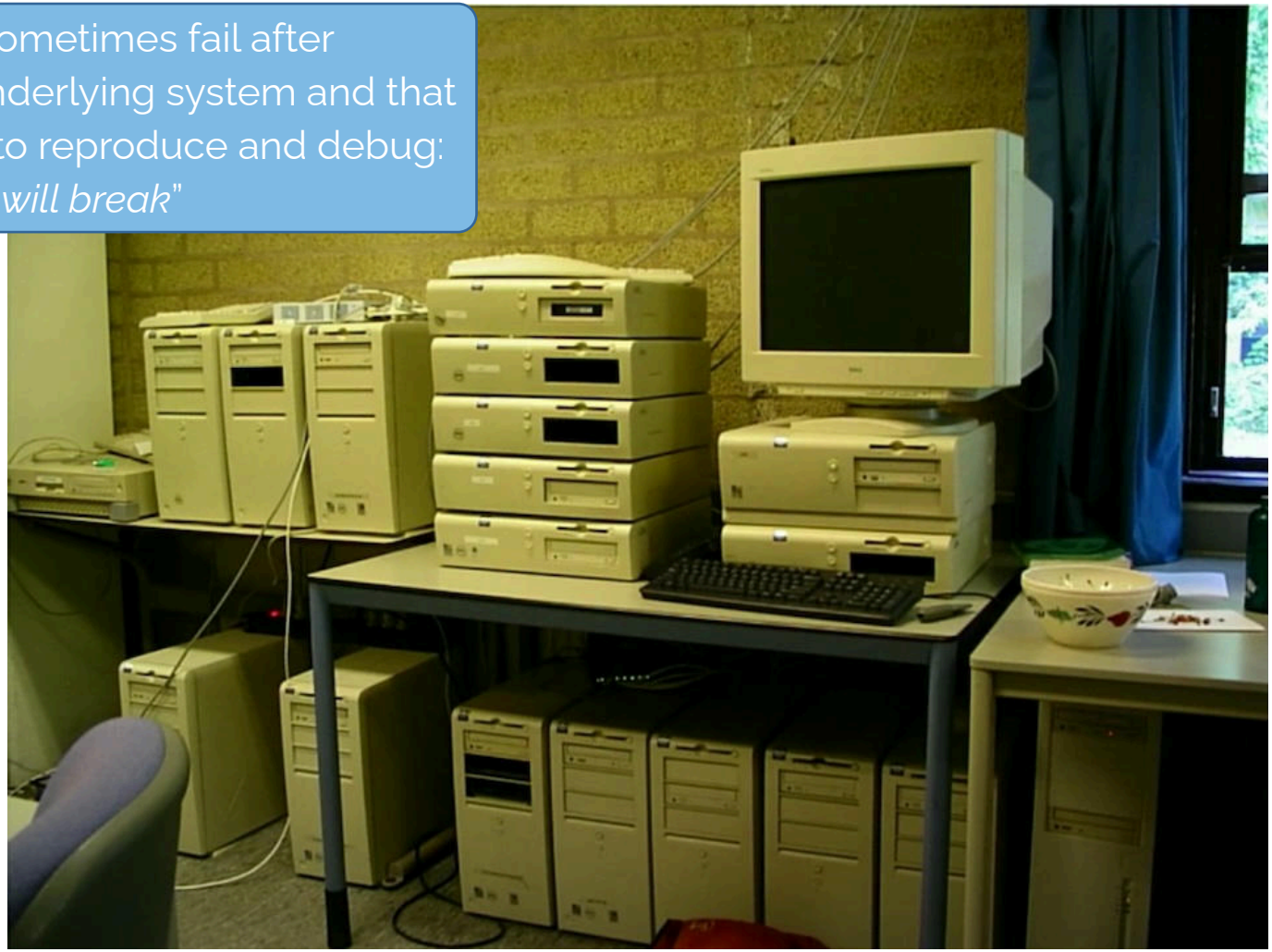
NixOS Integration Tests

About NixOS

NixOS: A Master Thesis that Escalated 👩‍🎓

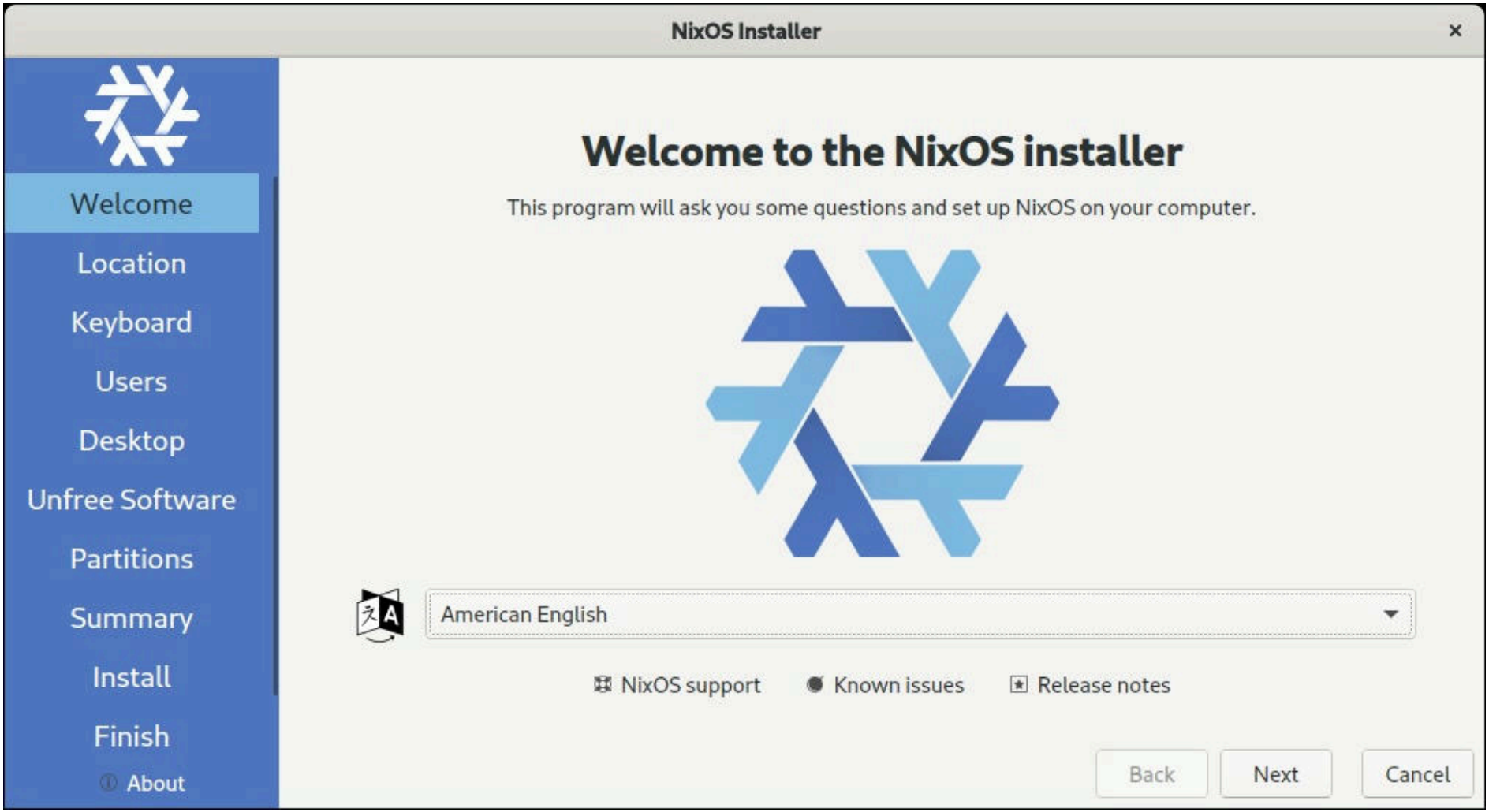
NIXCADEMY EXPERT EDUCATION

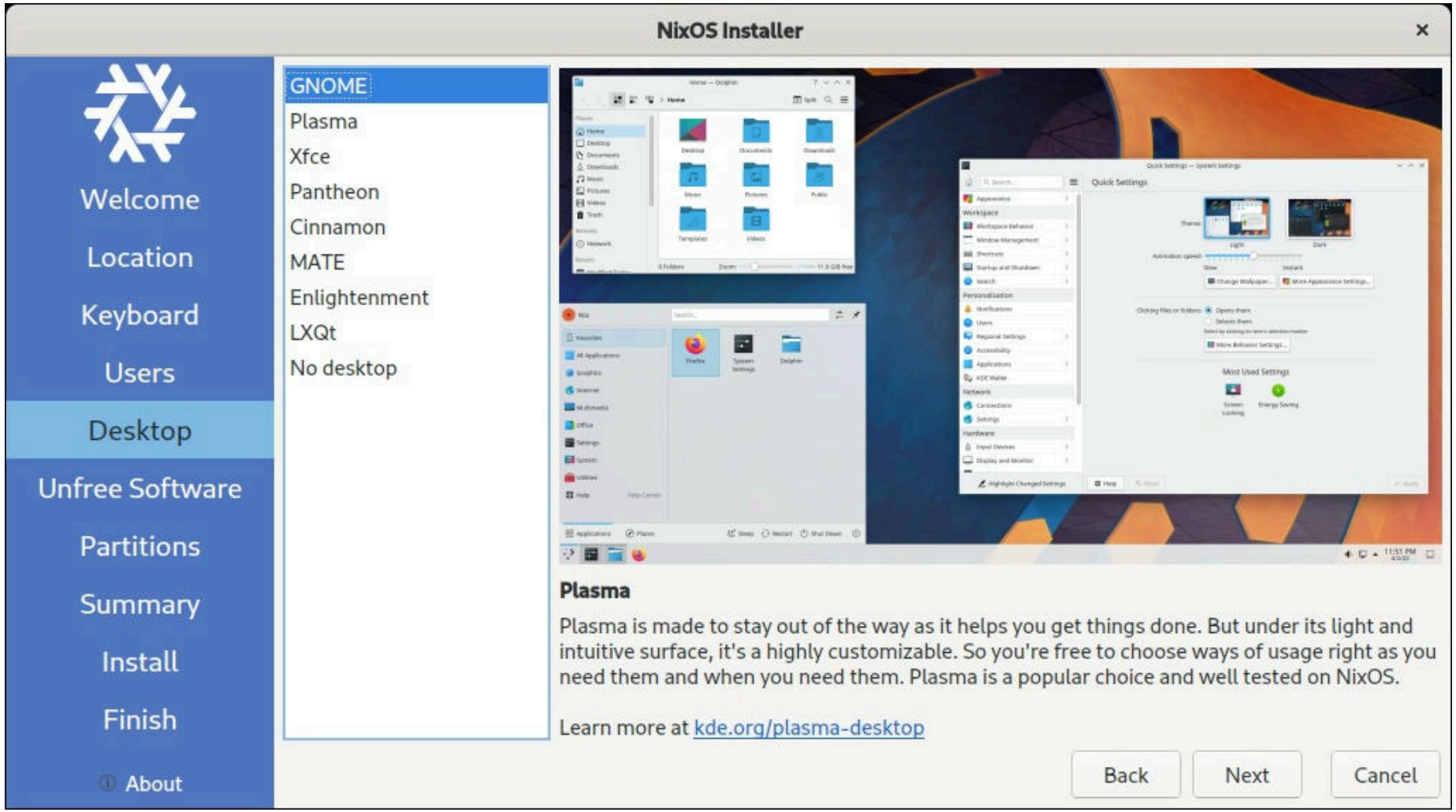
"Builds would sometimes fail after updating the underlying system and that would be hard to reproduce and debug: *Don't touch or it will break*"

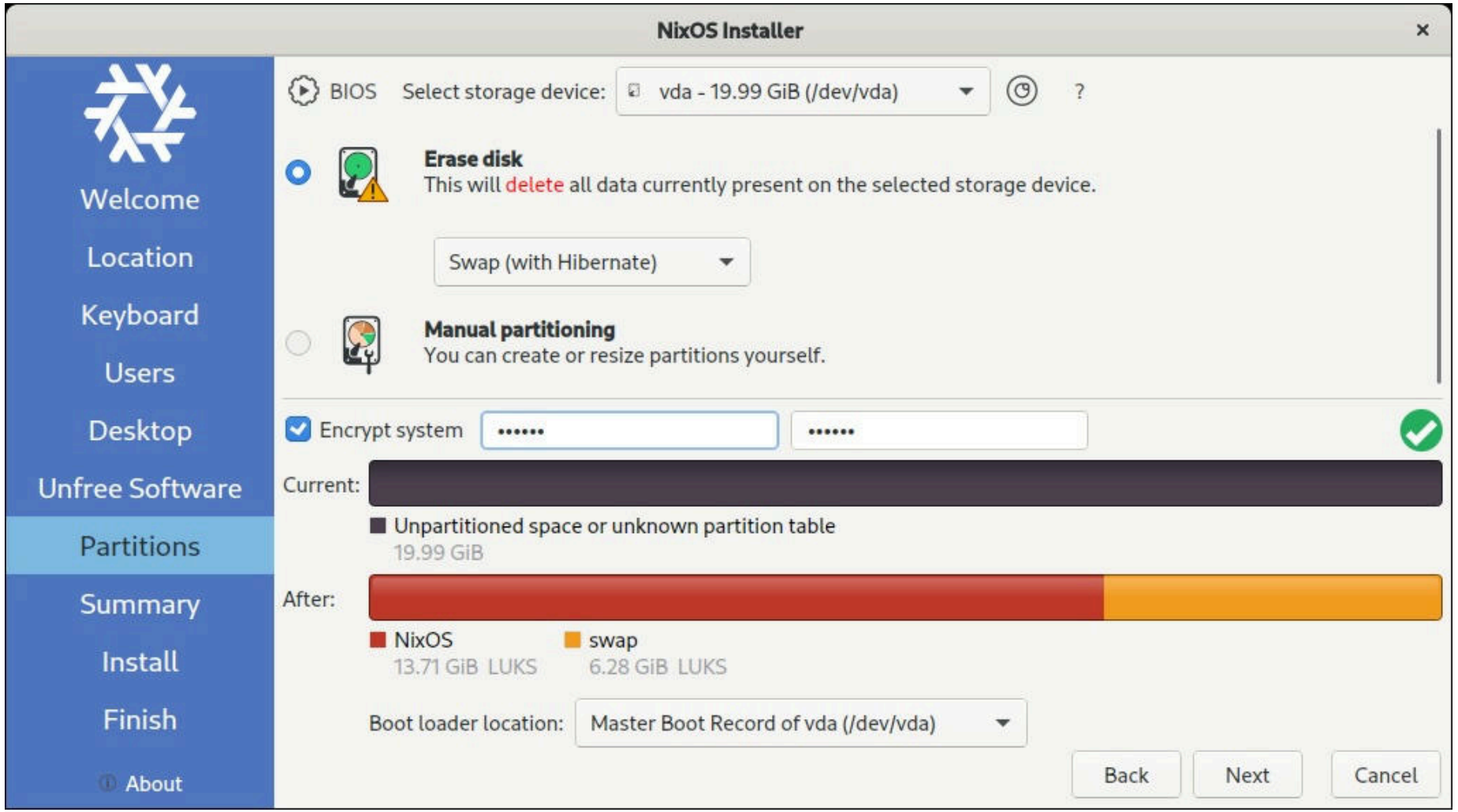


tjaalton@ubuntu.com

Installing NixOS


NixOS Installer
Welcome
Location
Keyboard
Users
Desktop
Unfree Software
Partitions
Summary
Install
Finish
About
Welcome to the NixOS installer
This program will ask you some questions and set up NixOS on your computer.
American English
NixOS support
Known issues
Release notes
Back
Next
Cancel


NixOS Installer
Welcome
Location
Keyboard
Users
Desktop
Unfree Software
Partitions
Summary
Install
Finish
About
GNOME
Plasma
Xfce
Pantheon
Cinnamon
MATE
Enlightenment
LXQt
No desktop
Plasma
Plasma is made to stay out of the way as it helps you get things done. But under its light and intuitive surface, it's a highly customizable. So you're free to choose ways of usage right as you need them and when you need them. Plasma is a popular choice and well tested on NixOS.
Learn more at kde.org/plasma-desktop
Back
Next
Cancel


NixOS Installer
Welcome
Location
Keyboard
Users
Desktop
Unfree Software
Partitions
Summary
Install
Finish
About
BIOS
Select storage device:
vda - 19.99 GiB (/dev/vda)
?
Erase disk
This will delete all data currently present on the selected storage device.
Swap (with Hibernate)
Manual partitioning
You can create or resize partitions yourself.
Encrypt system
Current:
Unpartitioned space or unknown partition table
19.99 GiB
After:
NixOS
13.71 GiB LUKS
swap
6.28 GiB LUKS
Boot loader location:
Master Boot Record of vda (/dev/vda)
Back
Next
Cancel

The official download page provides multiple ways to install or try NixOS:
https://nixos.org/download.html#nixos-iso

- ISO image: Graphical and Minimal Variant
- VM as Virtualbox `.ova` file
- Amazon EC2 Image for Launch in Cloud

We will see different ways to install NixOS. Let's see what they all have in common.

1. **Boot Live System**
   - Set up network
2. **Partition, Format & Mount Disk**
   - Use your favorite tools
3. **Create Initial System Config**
   - `nixos-generate-config`
4. **Install**
   - `nixos-install`
   - Generates *Top-Level Closure*
   - Runs Install Script of Top-Level Closure

See also NixOS Manual, Chapter 2 Installing NixOS

NixOS Minimal Installer Manual

https://nixos.org/manual/nixos/stable/index.html#sec-installation-manual-partitioning

Tasks:

- Let's create a NixOS VM from scratch to learn about the UX better
- Let's change the config and run `sudo nixos-rebuild switch`
- Rebuild into *older* generations

Notes:

- Only follow the UEFI *or* MBR/BIOS steps, according to your VM
- Mind the disk space: Choose e.g. at least 20GB
- You can skip/shrink the suggested 8GB swap partitioning to save space

NixOS Modules

Let's first have a look at configurations in general.

Then, we write our own.

A NixOS Configuration: configuration.nix

NIXCADEMY EXPERT EDUCATION

```
{ config, pkgs, ... }:
{
  imports = [
    ./hardware-configuration.nix
  ];

  networking.hostName = "jongepad";
  networking.networkmanager.enable = true;

  environment.systemPackages = with pkgs; [
    git
    vim
    wget
  ];

  # ...

  system.stateVersion = "22.05"; # This needs some discussion
}
```

A NixOS Configuration: hardware-configuration.nix

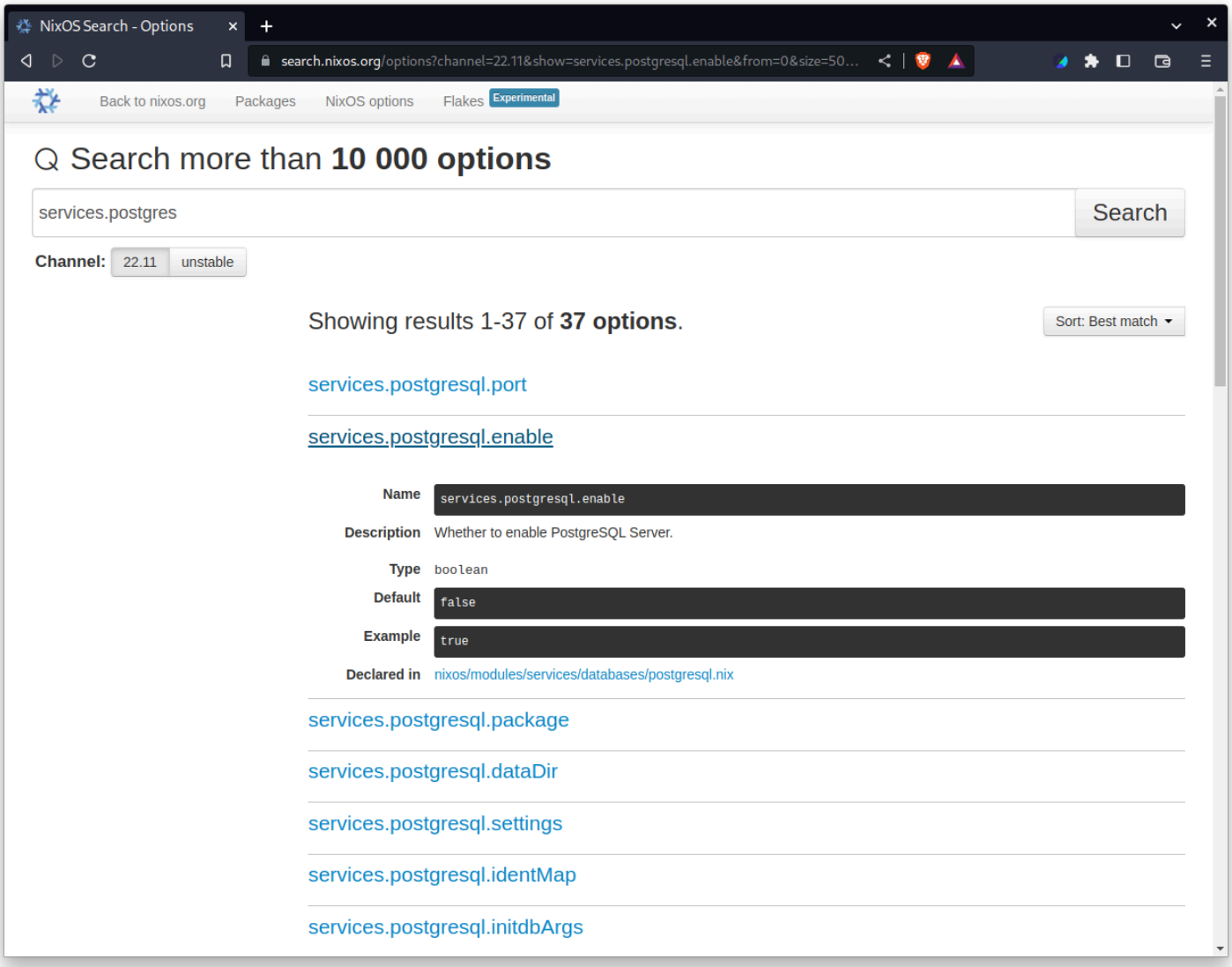

```
{ lib, modulesPath, ... }:
{
  imports = [ (modulesPath + "/installer/scan/not-detected.nix") ];

  boot.initrd.availableKernelModules =
    [ "xhci_pci" "nvme" "usb_storage" "sd_mod" "sdhci_pci" ];
  boot.initrd.kernelModules = [ "dm-snapshot" ];
  boot.kernelModules = [ "kvm-intel" ];

  fileSystems."/" = {
    device = "/dev/disk/by-uuid/6fdeb208-6446-4943-9a7d-e8a5d78ade50";
    fsType = "ext4";
  };
  fileSystems."/boot" = {
    device = "/dev/disk/by-uuid/091D-4B27";
    fsType = "vfat";
  };

  powerManagement.cpuFreqGovernor = lib.mkDefault "ondemand";
}
```

All NixOS Modules Always

```
{ config, pkgs, ... }:
{
  imports = [
    # paths of other modules
  ];

  attribute.path = this;
  other = {
    path = that;
  };
}
```

Where do we know these *magic* attribute paths from?

Shopping for NixOS Settings: search.nixos.org



NixOS Configuration Examples

Immutable User Accounts

```
{
  users.users.alice = {
    isNormalUser = true;  # default
    home = "/home/alice"; # default
    description = "Alice Foobar";
    extraGroups = [
      "networkmanager"
      "wheel"
    ];
    openssh.authorizedKeys.keys = [
      "ssh-dss AAAAB3Nza... alice@foobar"
    ];
  };
}
```

```
{
  boot.binfmt.emulatedSystems = [
    "aarch64-linux"
    "wasm32-wasi"
    "x86_64-windows"
  ];
}
```

Run foreign binaries in your shell!

NixOS did not invent this feature - but it's the only distro where it's so easy to use!

```
{
  boot.kernelPatches = [
    {
      name = "custom-config";
      patch = ./my-patch-file.patch;
      extraConfig = ''
        DEBUG_INFO y
        ENABLE_MY_FUNCTION y
      '';
    }
  ];
}
```

Note how patch-specific config snippets are organized with the patch.

Composition: Kernel version and more patches can be set in other modules.

More Examples Online

https://galowicz.de/2023/03/13/quick-vms-with-nixos/

- How to quickly set up a VM and extend it

https://galowicz.de/2023/04/05/single-command-server-bootstrap/

- Automate partitioning, formatting, mounting, and installing NixOS on foreign resources

https://nixcademy.com/2023/09/26/nixos-openstreetmap/

- Run OpenStreetMap on NixOS

https://nixcademy.com/2023/09/04/nixos-multi-php-version-container/

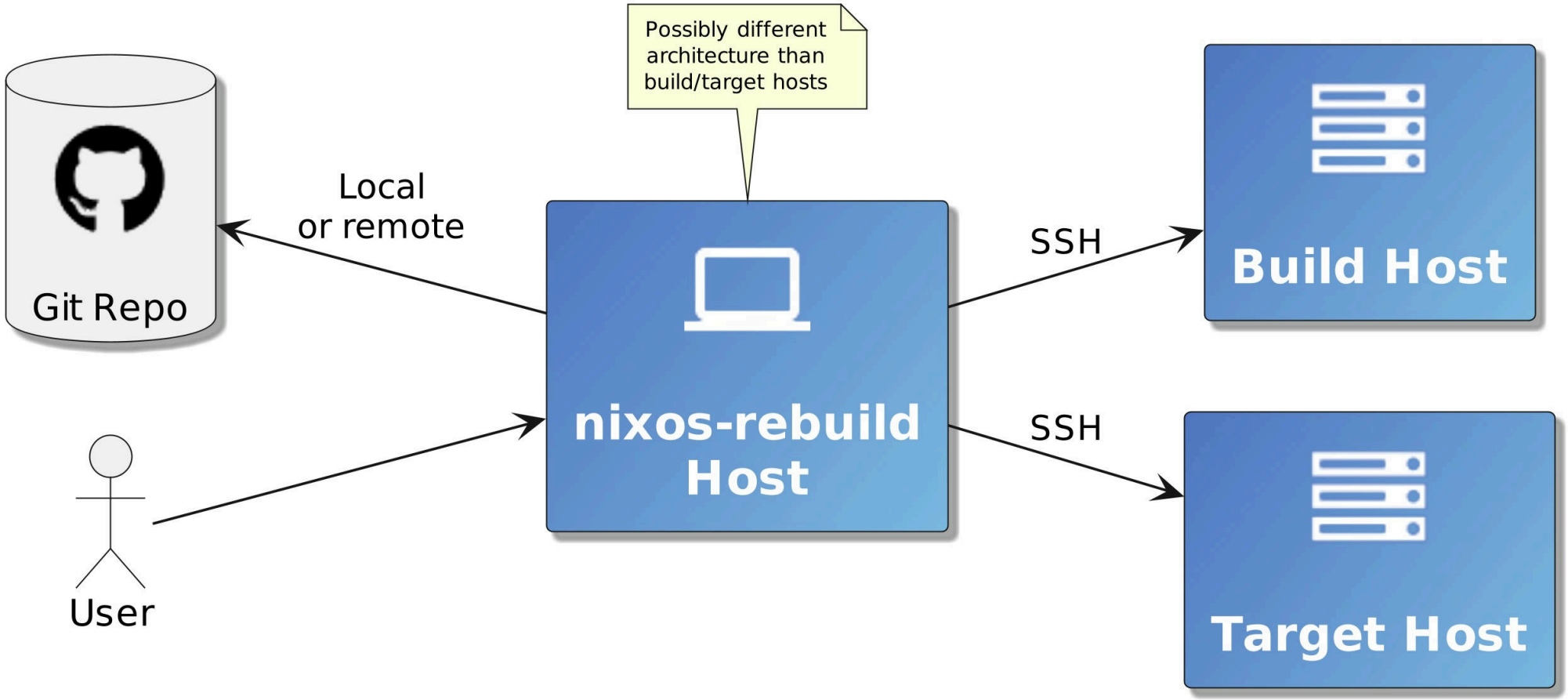
- Run Nginx + multiple PHP versions on NixOS

The `nixos-rebuild` Command

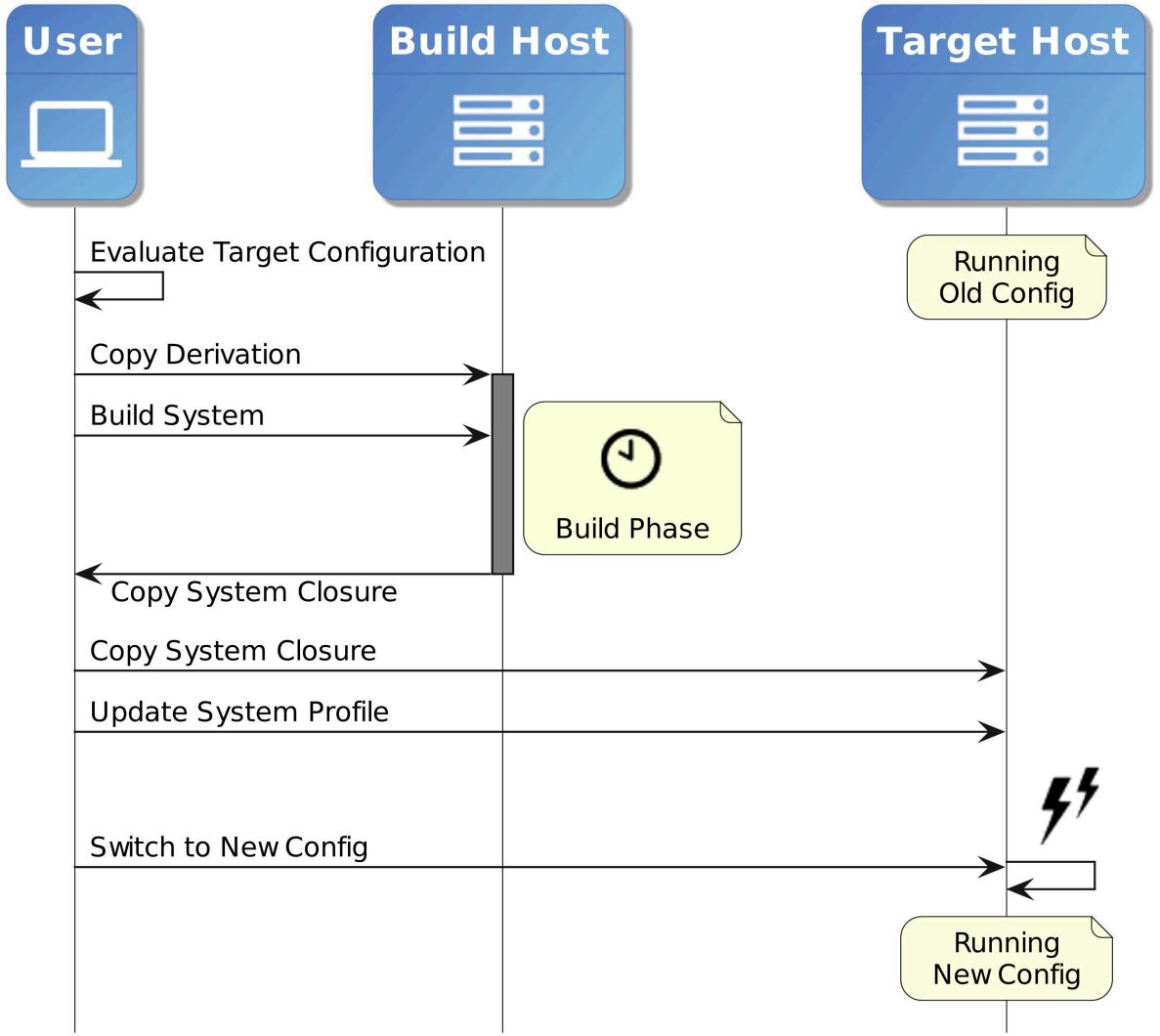
Build and/or deploy on/to different host

nixos-rebuild example

```
nixos-rebuild \
  --flake .#mySystem \
  --build-host builduser@buildhost \
  --target-host deployuser@deployhost \
  --sudo \
  --use-substitutes \
  switch
```


Possibly different architecture than build/target hosts
Git Repo
Local or remote
nixos-rebuild Host
SSH
Build Host
User
SSH
Target Host

Remote Deploy Sequence with `--target-host`



1. `nix build .#nixosConfigurations.mySystem.config.system.build.toplevel`
2. Copy the result to the target host with `nix copy`
3. `nix-env -p /nix/var/nix/profiles/system --set <toplevel storepath>`
   (on the target host)
4. `sudo <toplevel storepath>/bin/switch-to-configuration switch`

NixOS Containers

NIXCADEMY EXPERT EDUCATION

Not Containerized

```
{ config, pkgs, lib, ... }:
{
  imports = [
    ./some-service.nix
  ];
}
```

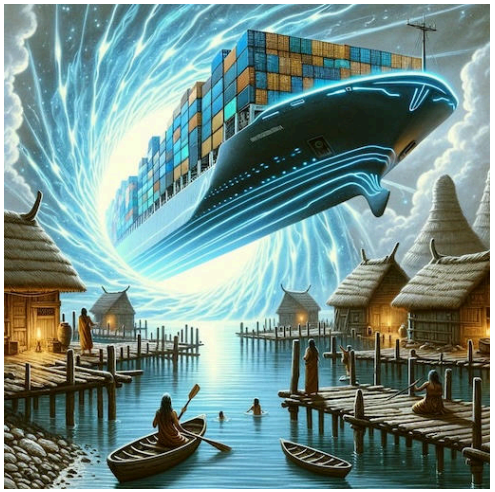
Containerized

```
{ config, pkgs, lib, ... }:
{
  containers.webserver = {
    autoStart = true;
    privateNetwork = false;
    config = { config, pkgs, ... }: {
      imports = [
        ./some-service.nix
      ];
    };
  };
}
```

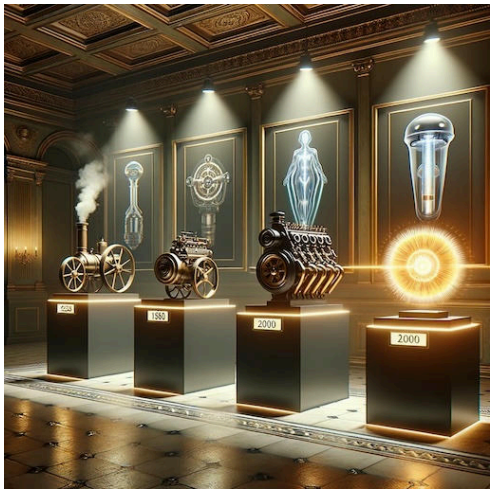
Imperatively start and maintain NixOS containers

```
$ nixos-container create foo --config '
  services.openssh.enable = true;
  users.users.root.openssh.authorizedKeys.keys = ["ssh-dss AAAAB3N…"];
'
$ nixos-container create test --config-file test-container.nix \
  --local-address 10.235.1.2 --host-address 10.235.1.1

$ nixos-container start foo
$ systemctl status container@foo
$ nixos-container root-login foo
$ nixos-container destroy foo
```

Blog: systemd containers



Running NixOS from any Linux Distro in systemd-nspawn Containers
https://nixcademy.com/posts/nixos-nspawn/



Multi-Version PHP with Nginx on NixOS: Containerized and Not
https://nixcademy.com/posts/nixos-multi-php-version-container/

NixOS Patterns

Extended Module Structure

```
{ config, pkgs, ... }:
{
  imports = [
    # paths of other modules
  ];

  options = {
    # option declarations
  };

  config = {
    # option definitions
  };
}
```

See also NixOS Manual, Chapter 67 Writing NixOS Modules

Option Structure

```
options = {
  name = mkOption {
    type = # type specification
           # one of pkgs.lib.types.*
    default = # default value
    example = # example value
    description = "cool explanation";
  };
};
```

Automatic docs use these descriptions, too

See also:
NixOS Manual, Chapter 67.1 Option Declarations
NixOS Manual, Chapter 67.2 Options Types

`mkEnable` Helper

```nix
lib.mkEnableOption "the ultimate service"

# is equivalent to:

lib.mkOption {
  type = lib.types.bool;
  default = false;
  example = true;
  description = "Whether to enable the ultimate service";
}
```

See also 67.1.1. Utility functions for common option patterns

File: `module-a.nix`

```
{ ... }:
{
  a.b.c = [ 1 ];
}
```

File: `module-b.nix`

```
{ ... }:
{
  a.b.c = [ 2 ];
}
```

File: `top-module.nix`

```
{ ... }:
{
  imports = [
    ./module-a.nix
    ./module-b.nix
  ];
  a.b.c = [ 3 ];
}
```

Resulting Configuration

NixOS Module Options are Monoids

File: `module-a.nix`

```
{ ... }:
{
  a.b.c = [ 1 ];
}
```

File: `module-b.nix`

```
{ ... }:
{
  a.b.c = [ 2 ];
}
```

File: `top-module.nix`

```
{ ... }:
{
  imports = [
    ./module-a.nix
    ./module-b.nix
  ];
  a.b.c = [ 3 ];
}
```

Resulting Configuration

```
{
  config.a.b.c = [ 1 2 3 ];
}
```

Our First Own NixOS Module

```
{ config, pkgs, lib, ... }:
let cfg = config.services.hello; in
{
  options.services.hello = {
    enable = lib.mkEnableOption "GNU-Hello as a Service";
    port = lib.mkOption {
      type = lib.types.port;
      default = 8081;
      description = "Port to listen on";
    };
  };
  config = lib.mkIf cfg.enable {
    systemd.services.hello = {
      wantedBy = [ "multi-user.target" ];
      serviceConfig.ExecStart = ''${pkgs.socat}/bin/socat \
        TCP4-LISTEN:${builtins.toString cfg.port},reuseaddr,fork \
        EXEC:${pkgs.hello}/bin/hello
      '';
    };
  };
}
```

Exercise Time: Run & Extend the Hello aaS 👋

Download 💾

https://nixcademy.com/downloads/hello-as-a-service.zip

Tasks:

- Run the VM: `$(nix-build)/bin/run-nixos-vm`
- Inspect the shell script, answer this question:

  👉 Why is rebuilding the whole VM after config changes so quick?
- Enable the firewall
- Add an option: `services.hello.openFirewall`

Notes:

- Look up `networking.firewall` on https://search.nixos.org
- Set `virtualisation.graphics = false;` to run on training-machine.nixcademy.com
- Solution: https://nixcademy.com/downloads/hello-as-a-service-solution.zip

The nixos Function

```
let
  pkgs = ...;
  nixos = pkgs.nixos [
    module1 module2 module3 # ...
  ];
in {
  inherit (nixos.config.system.build)
    toplevel # always available
    isoImage # modulesPath + "/installer/cd-dvd/iso-image.nix"
    vm       # modulesPath + "/virtualisation/qemu-vm.nix"
    # whatever the nixos config exports
    ;
}
```

Endless Possibilities:

Images for Amazon, Azure, Google Cloud Engine, VirtualBox, kexec, netboot, KubeVirt, LXC, Proxmox, CloudStack, many more, and *your own*!

⚠️ NixOS Module Rules 1/2 ⚠️

- Modules must not know the paths of other modules
  - Exception: *Internal* module imports of modules that are split up in sub-modules for structure
- The import of a module must always be free of side-effects
  - i.e. modules only export their options by default
  - Modules with side-effects are called *profiles*
- Do not add overlays or nixpkgs configuration changes in a module
  - Consider providing `services.myservice.package` attrs
- The final NixOS configuration should pick all the imports and nixpkgs overrides
  - Composition over inheritance ☝️

These rules are for **module writers**.
Consumers may do whatever they want in final NixOS configs.

NIXCADEMY EXPERT EDUCATION

🚫 The most important and dangerous antipattern regarding modules 🤦

bad example

```nix
# File: `my-module.nix`
some: parameters:
# bad: taking "pre-parameters"
#      in NixOS modules

{ config, pkgs, lib, ... }:
{
  # using "some" and "parameters"
}
```

bad example

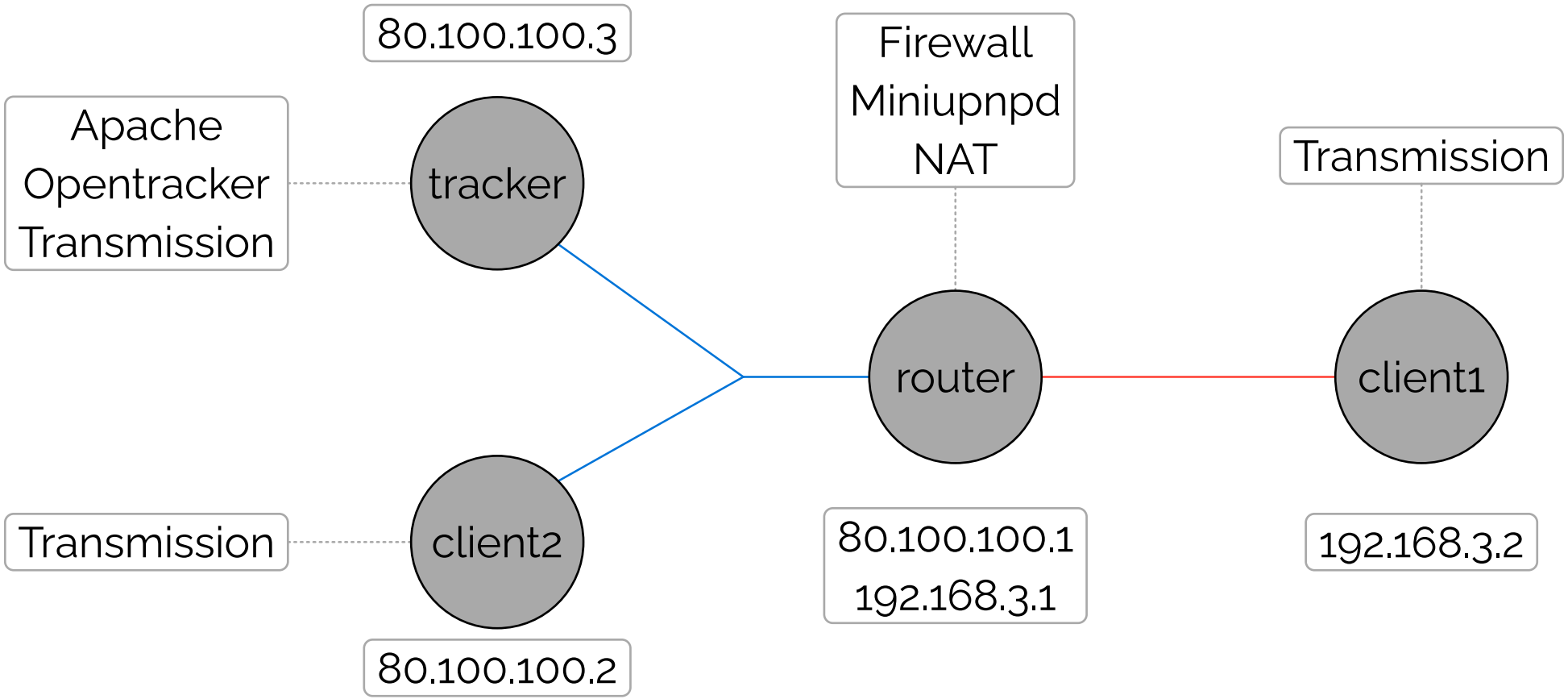
```nix
# File: configuration.nix`
{ config, pkgs, lib, ... }:
{
  imports = [
    (import ./my-module.nix
      somePkg
      parameterExpr
    ) # bad antipattern
  ];
}
```

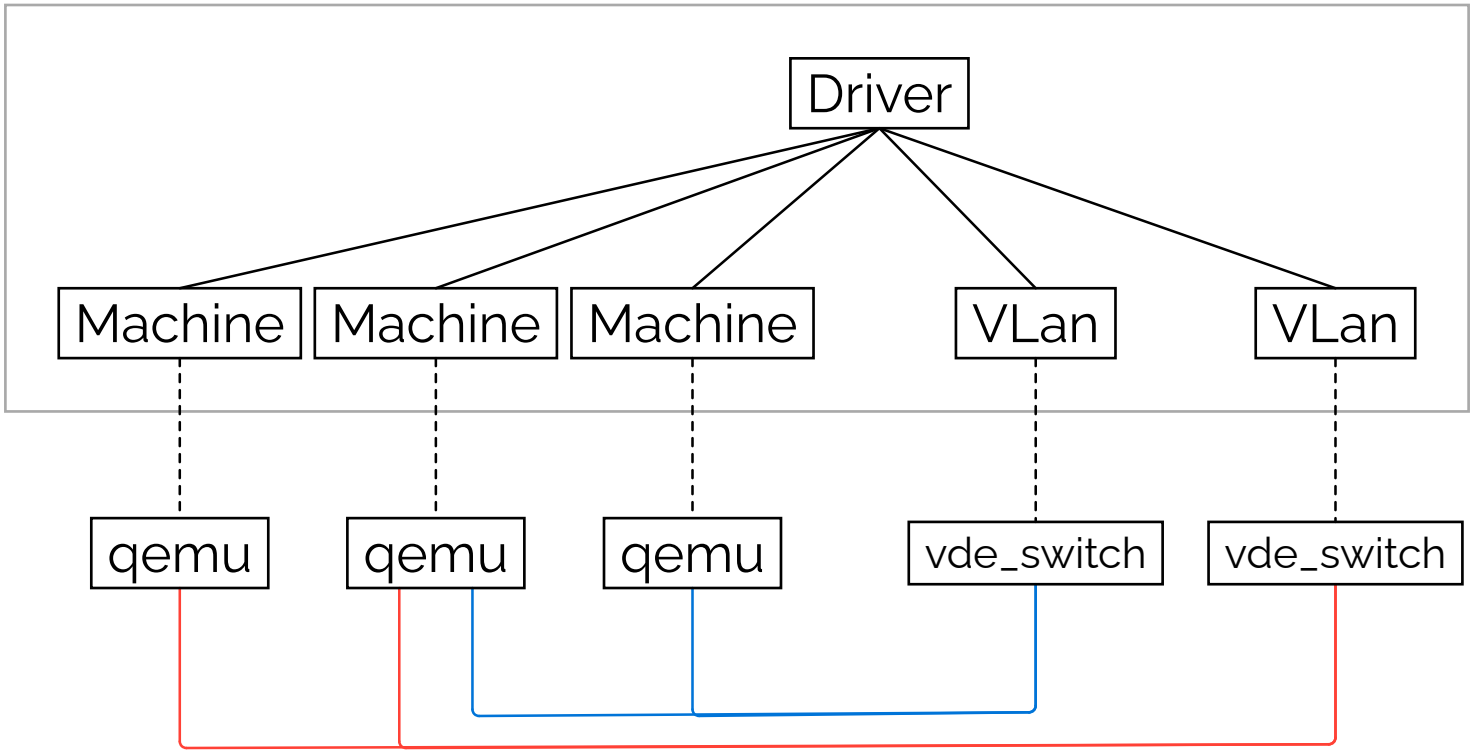
Instead, do:

- Inject values via new module options ✅
- Use `nixpkgs.overlays` for package level overrides ✅
- Use composition instead of inheritance for module additions ✅

NixOS Integration Tests

Complex Network & Service Definitions in ~50 LOC




Driver
Machine
Machine
Machine
VLan
VLan
qemu
qemu
qemu
vde_switch
vde_switch

What are the VMs built of?

```
node.config.system.build.vm
```

We already know why it's so **quick** to rebuild and rerun 4 different VMs!

Structure of a NixOS Integration Test

NIXCADEMY EXPERT EDUCATION

File: `old-way.nix`

```nix
let pkgs = import <nixpkgs> { };
in pkgs.nixosTest ./test.nix
```

File: `new-way.nix`

```nix
let pkgs = import <nixpkgs> { };
in
pkgs.testers.runNixOSTest {
  imports = [
    ./test.nix
  ];

  defaults = {
    networking.firewall.enable = false;
  };
}
```

File: `test.nix`

```nix
{
  name = "My NixOS Test";

  nodes = {
    machine1 = { ... }: { ... };
    machine2 = { ... }: { ... };
  };

  testScript = ''
    start_all()
    machine1.succeed(
      "ping -c 1 machine2"
    )
    machine2.succeed(
      "ping -c 1 machine1"
    )
  '';
}
```

See also NixOS Manual, Chapter 71 NixOS Tests

More Interesting `runTest` Parameters

```
pkgs.testers.runNixOSTest {
  imports = [ ... ];
  defaults = { ... };

  skipLint = true;
  skipTypeCheck = true;
  extraPythonPackages = p: [ p.numpy ];
}
```

All options are defined in module files in `nixpkgs/nixos/lib/testing/*`

See also NixOS Manual, Chapter 71.1.6 Test Options Reference

The `Machine` class can do many things

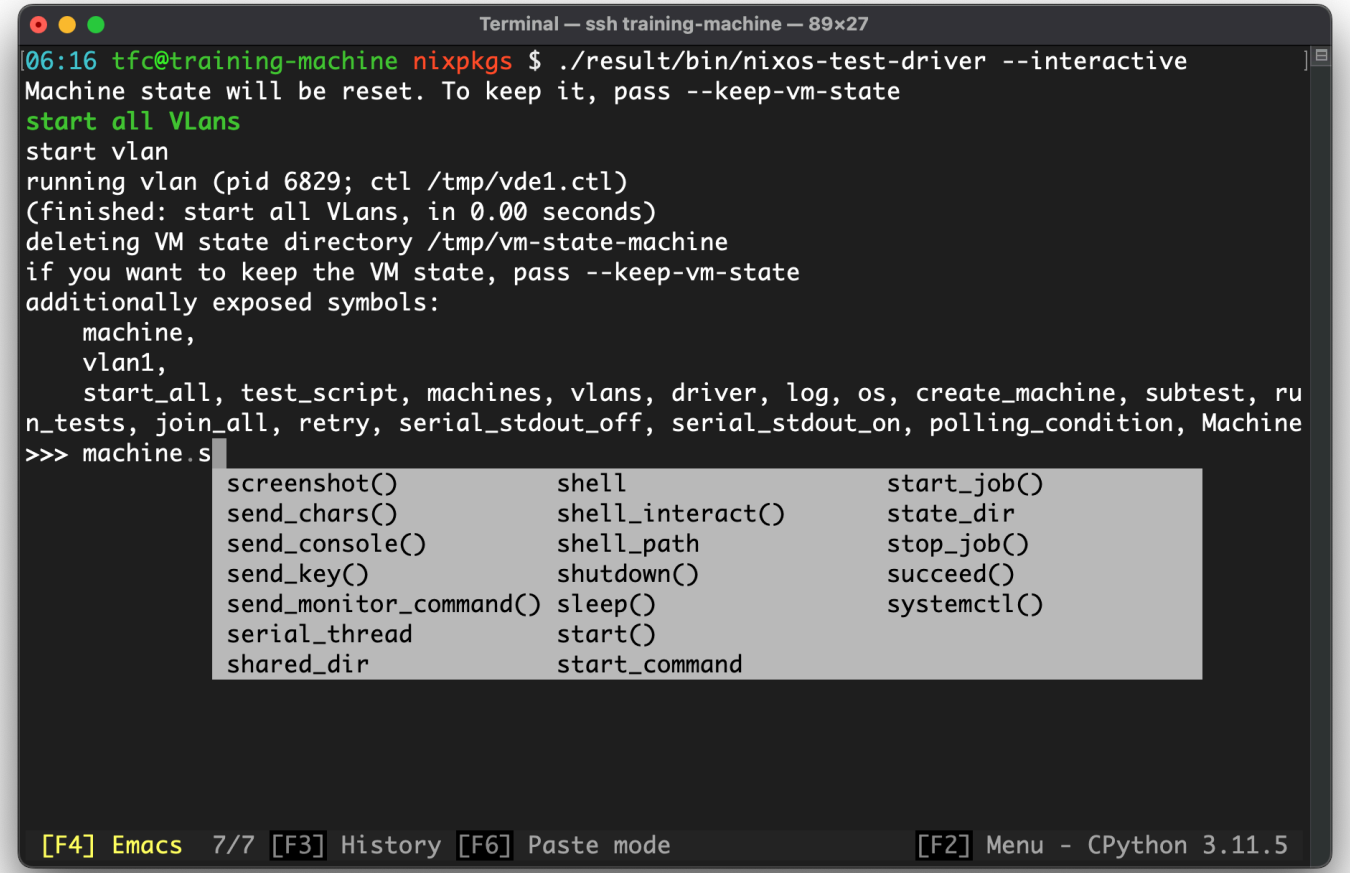
```
machine.start()
machine.wait_for_unit("my-systemd-unit.service")
machine.wait_for_open_port(123)

output = machine.succeed("command that shall succeed")
assert "foobar" in output, "command outputs the right things"
machine.fail("command that shall fail")

@polling_condition
def my_service_running():
    machine.succeed("pgrep -x my-service-process")

with my_service_running:
    machine.succeed("this takes time and may crash the service")
    machine.succeed("that might make it go OOM")
```

All these methods and *many more* are implemented in `nixpkgs/nixos/lib/test-driver/src/test_driver/machine/`

See also NixOS Manual, Chapter 71.1.3 Machine Objects

Interactive Mode

```
$ nix build .#myTest.driverInteractive
$ ./result/bin/nixos-test-driver
```



Settings for interactive mode

```
# test.nix ...

interactive.nodes.machine1 = {
  services.openssh = {
    enable = true;
    settings = {
      PermitRootLogin = "yes";
      PermitEmptyPasswords = "yes";
    };
  };
  security.pam.services.sshd.allowNullPassword = true;
  virtualisation.forwardPorts = [
    { from = "host"; host.port = 2222; guest.port = 22; }
  ];
};

# ...
```

Existing NixOS Integration Tests

>1000 integration tests exist in `nixpkgs/nixos/tests`.

These test partitioning, booting, apps on the desktop, take screenshots, perform OCR, run distributed services with sharding and redundancy, etc., etc…

Find the code in `nixpkgs/nixos/tests` and run from within the `pkgs.nixosTests` on different platforms.

Make sure to have a look at existing tests to see how others use this complex allrounder test framework.

🚨 Important Notice for fast tests 🚨

The NixOS test driver needs write access to `/dev/kvm`:

```
$ lsmod | grep kvm
kvm                   1355776  1 kvm_amd
...
$ ls -lsa /dev/kvm
0 crw-rw-rw- 1 root kvm 10, 232 Aug  8 16:31 /dev/kvm
```

If the `nixbldXX` users don't have write access to `/dev/kvm`, virtualisation will still work but it will be **slow**!

Quick fix (not persistent over reboots):

```
$ sudo chmod 777 /dev/kvm
```

Exercise Time: Test Hello-aaS

Download 💾

https://nixcademy.com/downloads/nixos-test.zip

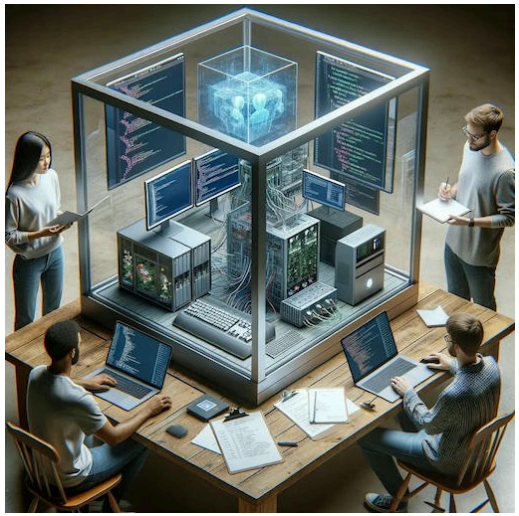
Task:

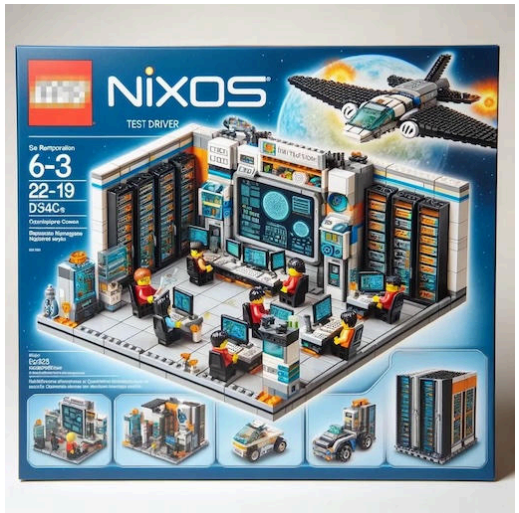
- Run the test using `nix-build`
- Read and understand the configuration files
- Copy `hello-as-a-service.nix` and `configuration.nix` from last exercise and use it
- Test the service meaningfully

Notes:

- Wait for a systemd service: `machine.wait_for_unit("bla.service")`
- Wait for a port to open: `machine.wait_for_open_port(123)`
- Solution: https://nixcademy.com/downloads/nixos-test-solution.zip

Blog: More about Integration Tests





Unveiling the Power of the NixOS Integration Test Driver (Part 1)
https://nixcademy.com/posts/nixos-integration-tests/

Unveiling the Power of the NixOS Integration Test Driver (Part 2)
https://nixcademy.com/posts/nixos-integration-tests-part-2/

Blog: NixOS Integration Tests on macOS

Build and Deploy Linux Systems from macOS
https://nixcademy.com/posts/macos-linux-builder/

Run NixOS Integration Tests on macOS
https://nixcademy.com/posts/running-nixos-integration-tests-on-macos/

Section: Additional resources

Exercise Time: Find and Fix Anti-Patterns 💪

Download 💾

https://nixcademy.com/downloads/anti-patterns.zip

Tasks:

- Study the code: it's full of anti-patterns!
- Discuss the disadvantages in a group
- Fix the anti-patterns
- Look into `hints.md` when you don't see any more anti-patterns

Notes:

- Solution: https://nixcademy.com/downloads/anti-patterns-solution.zip

The 3 Manuals:

Nix	https://nixos.org/nix/manual
Nixpkgs	https://nixos.org/nixpkgs/manual
NixOS	https://nixos.org/nixos/manual

Additional Resources:

Nix pills articles	https://nixos.org/guides/nix-pills
Official tutorials	https://nix.dev
NixOS Wiki	https://wiki.nixos.org
NixOS Forum	https://discourse.nixos.org
Chat	https://matrix.to/#/#community:nixos.org


NIXCADEMY
EXPERT EDUCATION

Monthly links + summaries of the Nix(OS) Blogosphere.

The *one* newsletter you need to subscribe to in order to stay up to date in the NixOS sphere.

https://fulltimenix.com