Nix

Reproducible development from theory to practice

ners

16 December 2025

https://github.com/ners/nix-lecture

Table of contents

Bibliography

Eelco Dolstra. (2006). *The Purely Functional Software Deployment Model*. https://edolstra.github.io/pubs/phd-thesis.pdf

Eelco Dolstra, Eelco Visser, & Jonge. (2004). *Imposing a Memory Management Discipline on Software Deployment*. https://edolstra.github.io/pubs/immdsd-icse2004-final.pdf

Valentin Gagarin. (2022, July 14). *Nix – taming Unix with functional programming*. https://www.tweag.io/blog/2022-07-14-taming-unix-with-nix/

Introduction

Nix is a solution for getting **computer programs from one machine to another**

— Eelco Dolstra (2006)

Nix is a solution for getting **computer programs from one machine to another** and having them **still work** when they get there.

— Eelco Dolstra (2006)

Motivation

- Software that we want to run has dependencies external to it

Motivation

- Software that we want to run has dependencies external to it
  - The author of the software has all the dependencies installed

Motivation

- Software that we want to run has dependencies external to it
  - The author of the software has all the dependencies installed
  - Most people will not have dependencies installed

Motivation

- Software that we want to run has dependencies external to it
  - The author of the software has all the dependencies installed
  - Most people will not have dependencies installed
- How can the author share their software with other people?

Motivation

- Software that we want to run has dependencies external to it
  - The author of the software has all the dependencies installed
  - Most people will not have dependencies installed
- How can the author share their software with other people?
  - At the very least, specify the dependencies

Motivation

- Software that we want to run has dependencies external to it
  - The author of the software has all the dependencies installed
  - Most people will not have dependencies installed
- How can the author share their software with other people?
  - At the very least, specify the dependencies
  - The specification should be a program

Motivation

- Linux users traditionally install software with package managers

Motivation

- Linux users traditionally install software with package managers
  - different package managers have different availability of software

Motivation

- Linux users traditionally install software with package managers
  - different package managers have different availability of software
  - package managers overwrite existing files when installing new ones

Motivation

- Linux users traditionally install software with package managers
  - different package managers have different availability of software
  - package managers overwrite existing files when installing new ones
    - how do we go back if something stops working?
    - what if we want multiple versions of the same software?

Motivation

- Linux users traditionally install software with package managers
  - different package managers have different availability of software
  - package managers overwrite existing files when installing new ones
    - how do we go back if something stops working?
    - what if we want multiple versions of the same software?
- We can use Podman and Docker to copy filesystem images between computers

Motivation

- Linux users traditionally install software with package managers
  - different package managers have different availability of software
  - package managers overwrite existing files when installing new ones
    - how do we go back if something stops working?
    - what if we want multiple versions of the same software?
- We can use Podman and Docker to copy filesystem images between computers
  - no guarantee that the image came from the recipe

Motivation

- Linux users traditionally install software with package managers
  - different package managers have different availability of software
  - package managers overwrite existing files when installing new ones
    - how do we go back if something stops working?
    - what if we want multiple versions of the same software?
- We can use Podman and Docker to copy filesystem images between computers
  - no guarantee that the image came from the recipe
  - even with the recipe there is no guarantee it is complete or correct

Motivation

- Linux users traditionally install software with package managers
  - different package managers have different availability of software
  - package managers overwrite existing files when installing new ones
    - how do we go back if something stops working?
    - what if we want multiple versions of the same software?
- We can use Podman and Docker to copy filesystem images between computers
  - no guarantee that the image came from the recipe
  - even with the recipe there is no guarantee it is complete or correct
  - these are repeatable, but not reproducible

Motivation

- Linux users traditionally install software with package managers
  - different package managers have different availability of software
  - package managers overwrite existing files when installing new ones
    - how do we go back if something stops working?
    - what if we want multiple versions of the same software?
- We can use Podman and Docker to copy filesystem images between computers
  - no guarantee that the image came from the recipe
  - even with the recipe there is no guarantee it is complete or correct
  - these are repeatable, but not reproducible
  - outside of the scope of this lecture

What is Nix?

- **Reproducible**
  - if a package works on one machine, it will also work on another
  - if it built yesterday it will still build today

What is Nix?

- **Reproducible**
  - if a package works on one machine, it will also work on another
  - if it built yesterday it will still build today
- **Declarative**
  - requirements must be specified completely and correctly
  - specifications can be composed

What is Nix?

- **Reproducible**
  - if a package works on one machine, it will also work on another
  - if it built yesterday it will still build today
- **Declarative**
  - requirements must be specified completely and correctly
  - specifications can be composed
- **Reliable**
  - packages do not interfere with each other
  - atomic: roll back to previous versions

What is Nix?

- How do we ensure that a specification is complete and correct?

What is Nix?

- How do we ensure that a specification is complete and correct?
  - Run the program in an isolated environment

What is Nix?

- How do we ensure that a specification is complete and correct?
  - Run the program in an isolated environment
  - Only give it access to resources declared by the specification

What is Nix?

- How do we ensure that a specification is complete and correct?
  - Run the program in an isolated environment
  - Only give it access to resources declared by the specification
  - Do not give it access to the network or hardware

What is Nix?

- The Nix ecosystem is many things:

What is Nix?

- The Nix ecosystem is many things:
  - Nix: the package manager

What is Nix?

- The Nix ecosystem is many things:
  - Nix: the package manager
  - Nix language: the functional language

What is Nix?

- The Nix ecosystem is many things:
  - Nix: the package manager
  - Nix language: the functional language
  - Nixpkgs: the software repository

What is Nix?

- The Nix ecosystem is many things:
  - Nix: the package manager
  - Nix language: the functional language
  - Nixpkgs: the software repository
  - NixOS: the Linux distribution

Nix store

Nix store

- The **Nix store** is a filesystem tree analogous to heap memory

Nix store

- The **Nix store** is a filesystem tree analogous to heap memory
- **Store objects** are files in the Nix store, analogous to allocated objects in memory

Nix store

- The **Nix store** is a filesystem tree analogous to heap memory
- **Store objects** are files in the Nix store, analogous to allocated objects in memory
- Each store object is referenced by a **store path**
  - filesystem path e.g. `/nix/store/0xk3r0njrijv434qim2lia11j3x9ivkc-hello`
  - analogous to pointers, or memory addresses

Nix store

- The **Nix store** is a filesystem tree analogous to heap memory
- **Store objects** are files in the Nix store, analogous to allocated objects in memory
- Each store object is referenced by a **store path**
  - filesystem path e.g. `/nix/store/0xk3r0njrijv434qim2lia11j3x9ivkc-hello`
  - analogous to pointers, or memory addresses
- Store objects can reference each other via store paths

Nix store

If the Nix store is «heap memory», which program populates and uses this memory?

Nix store

If the Nix store is «heap memory», which program populates and uses this memory?

- The program would be written in a functional language

Nix store

If the Nix store is «heap memory», which program populates and uses this memory?

- The program would be written in a functional language
  - the basic building block is a **function**

Nix store

If the Nix store is «heap memory», which program populates and uses this memory?

- The program would be written in a functional language
  - the basic building block is a **function**
  - functions can have **multiple inputs** and always produce **one output**

Nix store

If the Nix store is «heap memory», which program populates and uses this memory?

- The program would be written in a functional language
  - the basic building block is a **function**
  - functions can have **multiple inputs** and always produce **one output**
  - a function called with the **same inputs** will produce the **same output**

Nix store

If the Nix store is «heap memory», which program populates and uses this memory?

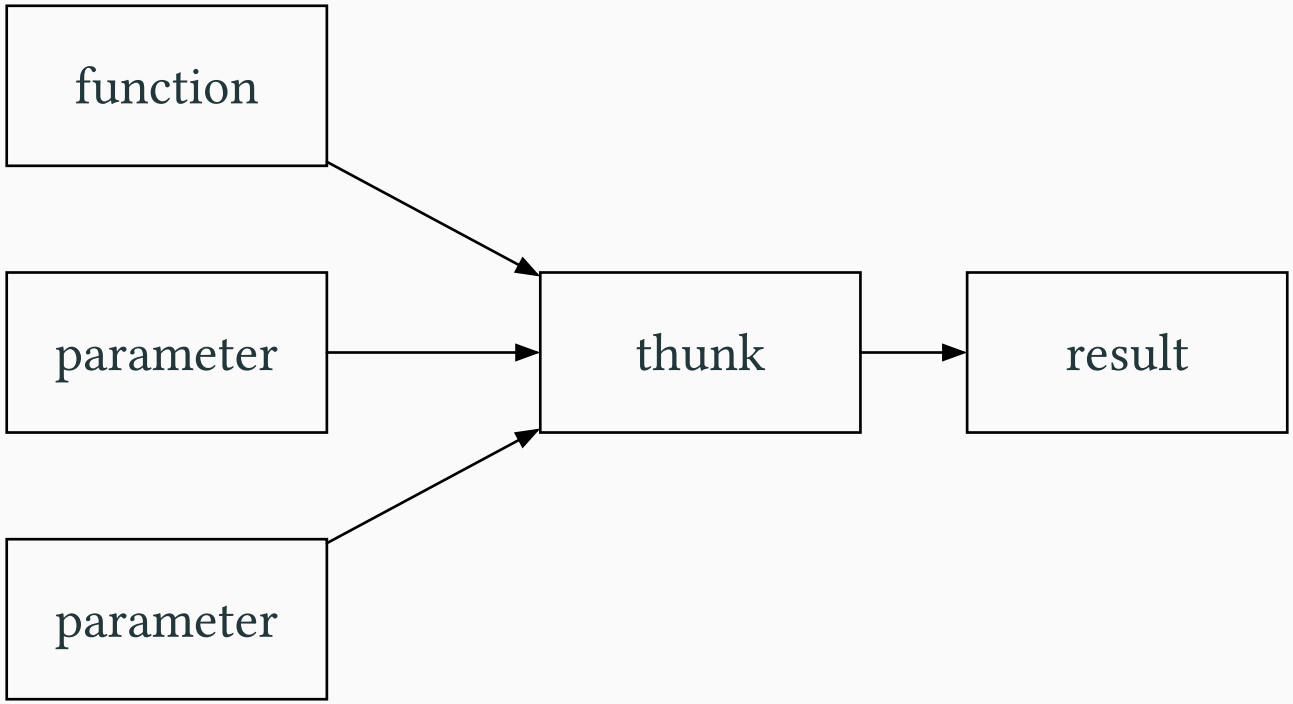
- The program would be written in a functional language
  - the basic building block is a **function**
  - functions can have **multiple inputs** and always produce **one output**
  - a function called with the **same inputs** will produce the **same output**
- The inputs and output are store objects

Nix store

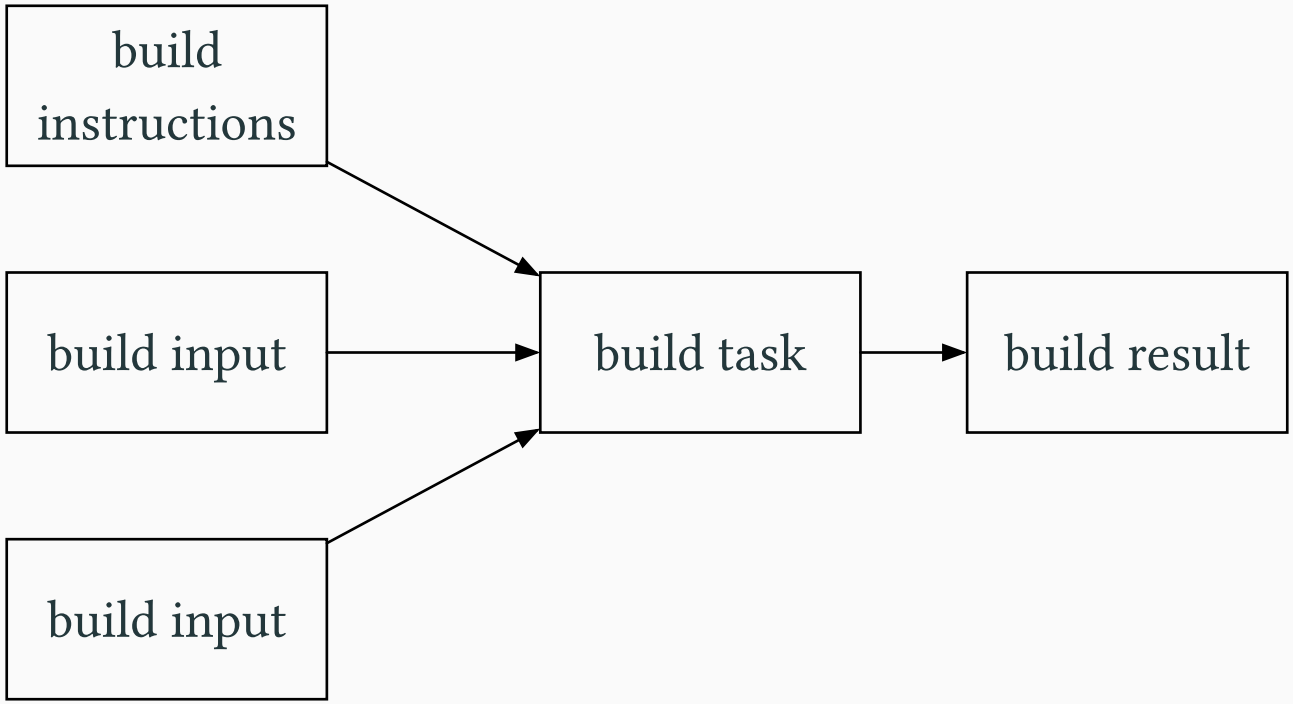
If the Nix store is «heap memory», which program populates and uses this memory?

- The program would be written in a functional language
  - the basic building block is a **function**
  - functions can have **multiple inputs** and always produce **one output**
  - a function called with the **same inputs** will produce the **same output**
- The inputs and output are store objects
- We can only create new store objects, not delete or update them

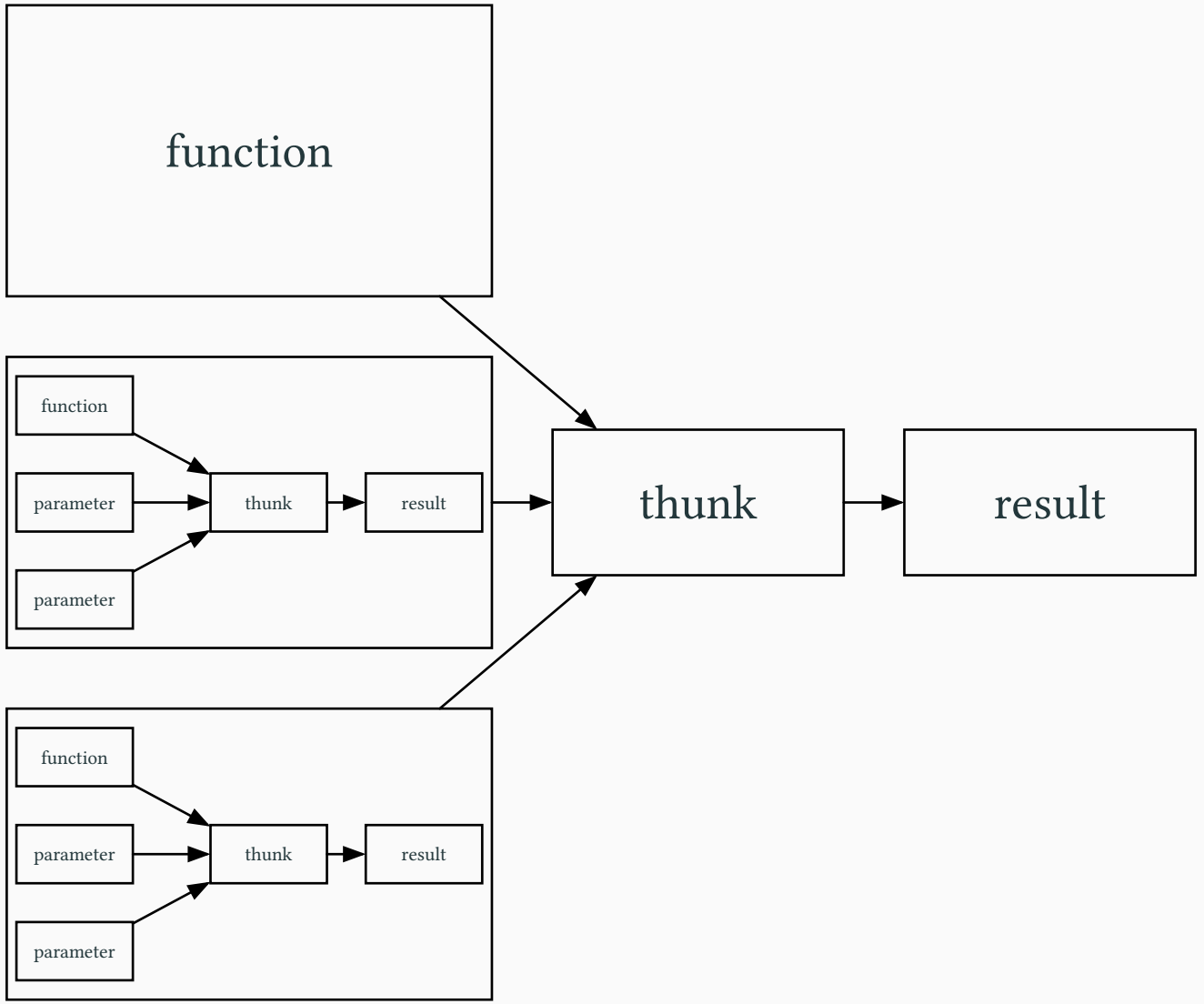
Nix store



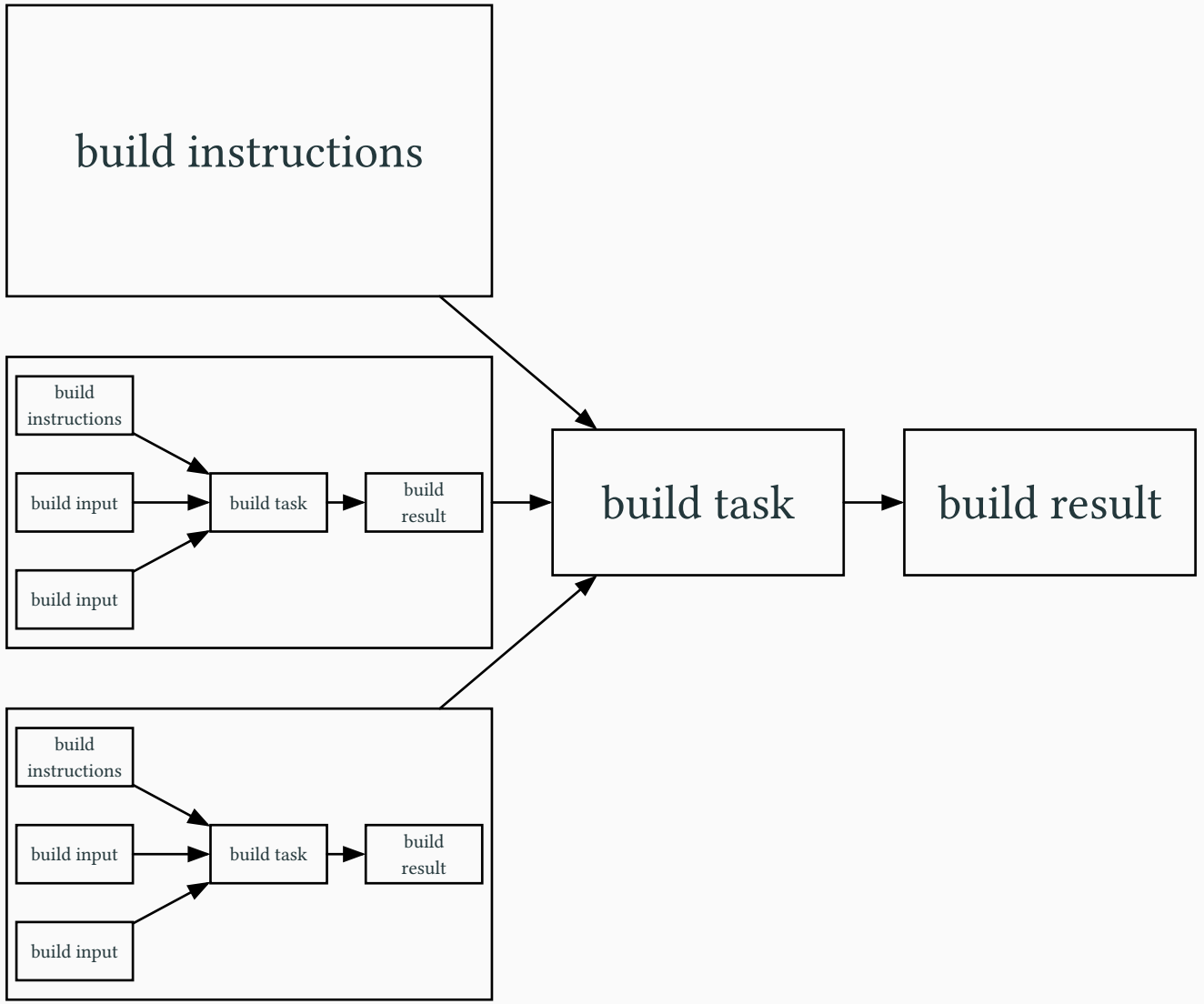
Nix store


build instructions
build input
build input
build task
build result

Nix store


function
function
parameter
parameter
thunk
result
thunk
result
function
parameter
parameter
thunk
result

Nix store


build instructions
build instructions
build input
build task
build result
build input
build task
build instructions
build input
build task
build result
build input

Nix store

What are the constants in our program?

Nix store

What are the constants in our program?

- We can copy files into the Nix store to turn them into store objects

Nix store

What are the functions in our program?

Nix store

What are the functions in our program?

- Unix processes that run **in a sandbox** to behave like pure functions

Nix store

What are the functions in our program?

- Unix processes that run **in a sandbox** to behave like pure functions
  - we only allow read access to the inputs

Nix store

What are the functions in our program?

- Unix processes that run **in a sandbox** to behave like pure functions
  - we only allow read access to the inputs
  - we only allow write access to the output location

Nix store

What are the functions in our program?

- Unix processes that run **in a sandbox** to behave like pure functions
  - we only allow read access to the inputs
  - we only allow write access to the output location
  - the output of the process will become a new store object

Nix store

What are the functions in our program?

- Unix processes that run **in a sandbox** to behave like pure functions
  - we only allow read access to the inputs
  - we only allow write access to the output location
  - the output of the process will become a new store object
- Nix calls these functions **derivations**

Nix store

What are the functions in our program?

- Unix processes that run **in a sandbox** to behave like pure functions
  - we only allow read access to the inputs
  - we only allow write access to the output location
  - the output of the process will become a new store object
- Nix calls these functions **derivations**
- Derivations are also store objects!

Nix store


```
{
  "/nix/store/djn4x0zqf8430kdighzz76wslr8g6alm-hello.drv": {
    "args": [
      "-c",
      "echo hello > $out"
    ],
    "builder": "/nix/store/3zdjy6cy39hyfbfabqi2i949v50s3pcb-sh",
    "inputDrvs": {},
    "inputSrcs": [
      "/nix/store/3zdjy6cy39hyfbfabqi2i949v50s3pcb-sh"
    ],
    "name": "hello",
    "outputs": {
      "out": {
        "path": "/nix/store/dy93f4lj9xkv3gkm6zvnfl6x3vckmgad-hello"
      }
    },
    "system": "x86_64-linux"
  }
}
```

Nix language

Nix language

Why do we need a new language?

Nix language

Why do we need a new language?

- Writing derivations is a complex, mechanical task

Nix language

Why do we need a new language?

- Writing derivations is a complex, mechanical task
- The Nix language automates this process with a concise syntax

Nix language


```
{
  "/nix/store/djn4x0zqf8430kdighzz76wslr8g6alm-hello.drv": {
    "args": [
      "-c",
      "echo hello > $out"
    ],
    "builder": "/nix/store/3zdjy6cy39hyfbfabqi2i949v50s3pcb-sh",
    "inputDrvs": {},
    "inputSrcs": [
      "/nix/store/3zdjy6cy39hyfbfabqi2i949v50s3pcb-sh"
    ],
    "name": "hello",
    "outputs": {
      "out": {
        "path": "/nix/store/dy93f4lj9xkv3gkm6zvnfl6x3vckmgad-hello"
      }
    },
    "system": "x86_64-linux"
  }
}
```

Nix language

```
{
  "/nix/store/djn4x0zqf8430kdighzz76wslr8g6alm-hello.drv": {
    "args": [
      "-c",
      "echo hello > $out"
    ],
    "builder": "/nix/store/3zdjy6cy39hyfbfabqi2i949v50s3pcb-sh",
    "inputDrvs": {},
    "inputSrcs": [
      "/nix/store/3zdjy6cy39hyfbfabqi2i949v50s3pcb-sh"
    ],
    "name": "hello",
    "outputs": {
      "out": {
        "path": "/nix/store/dy93f4lj9xkv3gkm6zvnfl6x3vckmgad-hello"
      }
    },
    "system": "x86_64-linux"
  }
}
```

```
derivation {
  name = "hello";
  builder = ./bash;
  args = [ "-c" "echo hello > $out" ];
  system = builtins.currentSystem;
}
```

Properties of the Nix language

Nix language

Properties of the Nix language

- Domain-specific language

Nix language

Properties of the Nix language

- Domain-specific language
- Purely functional

Nix language

Properties of the Nix language

- Domain-specific language
- Purely functional
- Lazily evaluated

Nix language

Properties of the Nix language

- Domain-specific language
- Purely functional
- Lazily evaluated
- Dynamically typed

Names and values

Names and values

```
let
  b = a + 1;
  a = 1;
in
a + b
```

Names and values

```
let
  b = a + 1;
  a = 1;
in
a + b
```

```
let
  attrset = { x = 1; };
in
attrset.x
```

Names and values

```
let
  b = a + 1;
  a = 1;
in
a + b
```

```
let
  attrset = { x = 1; };
in
attrset.x
```

```
let
  name = "Nix";
in
"Hello ${name}!"
```

Names and values

```
let
  b = a + 1;
  a = 1;
in
a + b
```

```
let
  attrset = { x = 1; };
in
attrset.x
```

```
let
  name = "Nix";
in
"Hello ${name}!"
```

Functions

Names and values

```
let
  b = a + 1;
  a = 1;
in
a + b
```

```
let
  attrset = { x = 1; };
in
attrset.x
```

```
let
  name = "Nix";
in
"Hello ${name}!"
```

Functions

```
let
  f = x: x + 1;
in
f 5
```

Names and values

```
let
  b = a + 1;
  a = 1;
in
a + b
```

```
let
  attrset = { x = 1; };
in
attrset.x
```

```
let
  name = "Nix";
in
"Hello ${name}!"
```

Functions

```
let
  f = x: x + 1;
in
f 5
```

```
let
  f = x: y: x + y;
in
f 2 3
```

Names and values

```
let
  b = a + 1;
  a = 1;
in
a + b
```

```
let
  attrset = { x = 1; };
in
attrset.x
```

```
let
  name = "Nix";
in
"Hello ${name}!"
```

Functions

```
let
  f = x: x + 1;
in
f 5
```

```
let
  f = x: y: x + y;
in
f 2 3
```

```
let
  f = { a, b }: a + b;
in
f { a = 2; b = 3; }
```

Functional programming

Functional programming

```
let
  fib = i:
    if i == 0 then 0
    else if i == 1 then 1
    else fib (i - 1) + fib (i - 2);
in
builtins.map fib [ 0 1 2 3 4 5 6 7 8 9 10 ]
```

Functional programming

```
let
  fib = i:
    if i == 0 then 0
    else if i == 1 then 1
    else fib (i - 1) + fib (i - 2);
in
builtins.map fib [ 0 1 2 3 4 5 6 7 8 9 10 ]
```

```
[ 0 1 1 2 3 5 8 13 21 34 55 ]
```

Nix language

The purpose of the Nix language is to easily **create** and **compose** derivations

Nix language

The purpose of the Nix language is to easily **create** and **compose** derivations

- **First-class filesystem paths**
  - a raw path resolves to a store path

Nix language

The purpose of the Nix language is to easily **create** and **compose** derivations

- **First-class filesystem paths**
  - a raw path resolves to a store path
- **First-class string templating**
  - we can encode snippets of any language in the Nix language
  - we can interpolate Nix expressions

Nix language

The purpose of the Nix language is to easily **create** and **compose** derivations

- **First-class filesystem paths**
  - a raw path resolves to a store path
- **First-class string templating**
  - we can encode snippets of any language in the Nix language
  - we can interpolate Nix expressions
- **String contexts**
  - a Nix string contains text and a set of dependencies
  - a string that refers to store objects contains their dependency closure

Nix language

```
derivation {
  name = "hello";
  builder = ./bash;
  args = [ "-c" "echo hello > $out" ];
  system = builtins.currentSystem;
}
```

```
{
  "/nix/store/djn4x0zqf8430kdighzz76wslr8g6alm-hello.drv": {
    "args": [
      "-c",
      "echo hello > $out"
    ],
    "builder": "/nix/store/3zdjy6cy39hyfbfabqi2i949v50s3pcb-sh",
    "inputDrvs": {},
    "inputSrcs": [
      "/nix/store/3zdjy6cy39hyfbfabqi2i949v50s3pcb-sh"
    ],
    "name": "hello",
    "outputs": {
      "out": {
        "path": "/nix/store/dy93f4lj9xkv3gkm6zvnfl6x3vckmgad-hello"
      }
    },
    "system": "x86_64-linux"
  }
}
```

Nix language

```
derivation {
  name = "hello";
  builder = ./gcc;
  args = [ "-o" "$out" ./hello.c ];
  system = builtins.currentSystem;
}
```

```
{
  "/nix/store/f2a15br5nm0nn7c7sq1rzcibwabisxfx-hello.drv": {
    "args": [
      "-o",
      "$out",
      "/nix/store/pm04fgly1zxmdx64kkqwf3kf1c11smdr-hello.c"
    ],
    "builder": "/nix/store/i31w737kksgxlnnb0ggbaapc5134awrh-gcc",
    "inputDrvs": {},
    "inputSrcs": [
      "/nix/store/i31w737kksgxlnnb0ggbaapc5134awrh-gcc",
      "/nix/store/pm04fgly1zxmdx64kkqwf3kf1c11smdr-hello.c"
    ],
    "name": "hello",
    "outputs": {
      "out": {
        "path": "/nix/store/0a52cvfra32hhv84vy0m4hc4sxdmr3p6-hello"
      }
    },
    "system": "x86_64-linux"
  }
}
```

Nix language

```
let
  buildScript =
    builtins.toFile "builder.sh" ''
      src=${./.}
      gcc=${./gcc}
      for c in $src/*.c; do
        $gcc -I$src -c -o $c.o $c
      done
      $gcc -o $out *.o
    '';
in
derivation {
  name = "hello";
  builder = ./bash;
  args = [ buildScript ];
  system = builtins.currentSystem;
}
```

```
mkdir $out
src=/nix/store/8854jh52mpd8g2rjqk9n2fhifsr20w81-hello-c-multiple
gcc=/nix/store/i31w737kksgxlnnb0ggbaapc5134awrh-gcc
for c in $src/*.c; do
  $gcc -I$src -c -o $c.o $c
done
$gcc -o $out *.o
```

Nix language

- Many real-world C projects are built with some standard utilities

Nix language

- Many real-world C projects are built with some standard utilities
  - Bash, GCC, Make, awk, sed, …

Nix language

- Many real-world C projects are built with some standard utilities
  - Bash, GCC, Make, awk, sed, …
  - These projects are built by running these tools in a sequence

Nix language

- Many real-world C projects are built with some standard utilities
  - Bash, GCC, Make, awk, sed, ...
  - These projects are built by running these tools in a sequence
- We can write a Nix function that builds a derivation with many standard utilities provided

Nix language

- Many real-world C projects are built with some standard utilities
  - Bash, GCC, Make, awk, sed, …
  - These projects are built by running these tools in a sequence
- We can write a Nix function that builds a derivation with many standard utilities provided
  - This function is a template for how to build a platonic C project

- Many real-world C projects are built with some standard utilities
  - Bash, GCC, Make, awk, sed, …
  - These projects are built by running these tools in a sequence
- We can write a Nix function that builds a derivation with many standard utilities provided
  - This function is a template for how to build a platonic C project
  - We fill in the template with function parameters

Nix language

```
let
  make = import ./build-make.nix;
  mkDerivation = { src, ... }:
    builtins.toFile "builder.sh" ''
      cp -r ${src}/* .
      export PATH="${make}/bin:$PATH"
      make
      PREFIX=$out make install
    '';
in
derivation {
  name = "hello";
  builder = ./bash;
  args = [ (mkDerivation { src = ./.; }) ];
  system = builtins.currentSystem;
}
```

Nix language

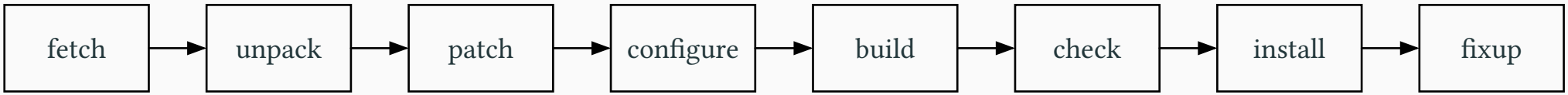
```
let
  make = import ./build-make.nix;
  glibc = import ./build-glibc.nix { inherit make; };
  mkDerivation = { src, buildInputs, ... }:
    let
      libraryPath = builtins.concatStringsSep ":" buildInputs;
    in
    builtins.toFile "builder.sh" ''
      cp -r ${src}/* .
      export PATH="${make}/bin:$PATH"
      export LD_LIBRARY_PATH="${libraryPath}:$LD_LIBRARY_PATH"
      make
      PREFIX=$out make install
    '';
in
derivation {
  name = "hello";
  builder = ./bash;
  args = [ (mkDerivation { src = ./.; buildInputs = [ glibc ]; }) ];
  system = builtins.currentSystem;
}
```

Nixpkgs

Nixpkgs

```
let
  pkgs = import <nixpkgs> {};
in
pkgs.stdenv.mkDerivation {
  pname = "hello";
  version = "0.0.1";
  source = ./.;
  buildInputs = [ ];
  buildPhase = "make";
  installPhase = "make install";
  meta.license = pkgs.lib.licenses.mit;
}
```

All software is built with The Pipeline™:



Nixpkgs

- A human-curated collection of software

Nixpkgs

- A human-curated collection of software
- All of the software has to build together (usually at the latest version)

Nixpkgs

- A human-curated collection of software
- All of the software has to build together (usually at the latest version)
- We can have multiple versions of the same software

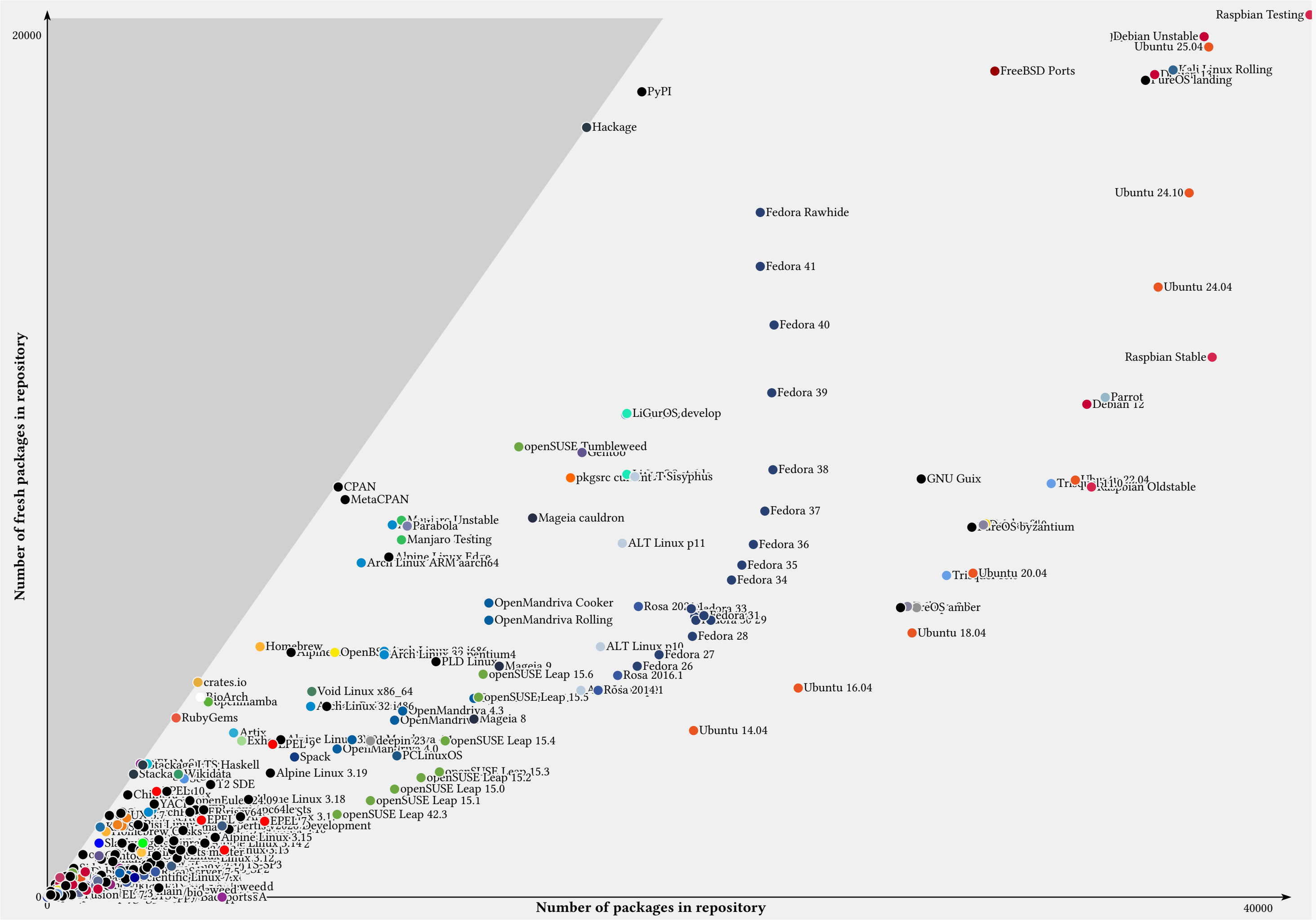
Nixpkgs

- A human-curated collection of software
- All of the software has to build together (usually at the latest version)
- We can have multiple versions of the same software
- The sources of packages are fetched deterministically

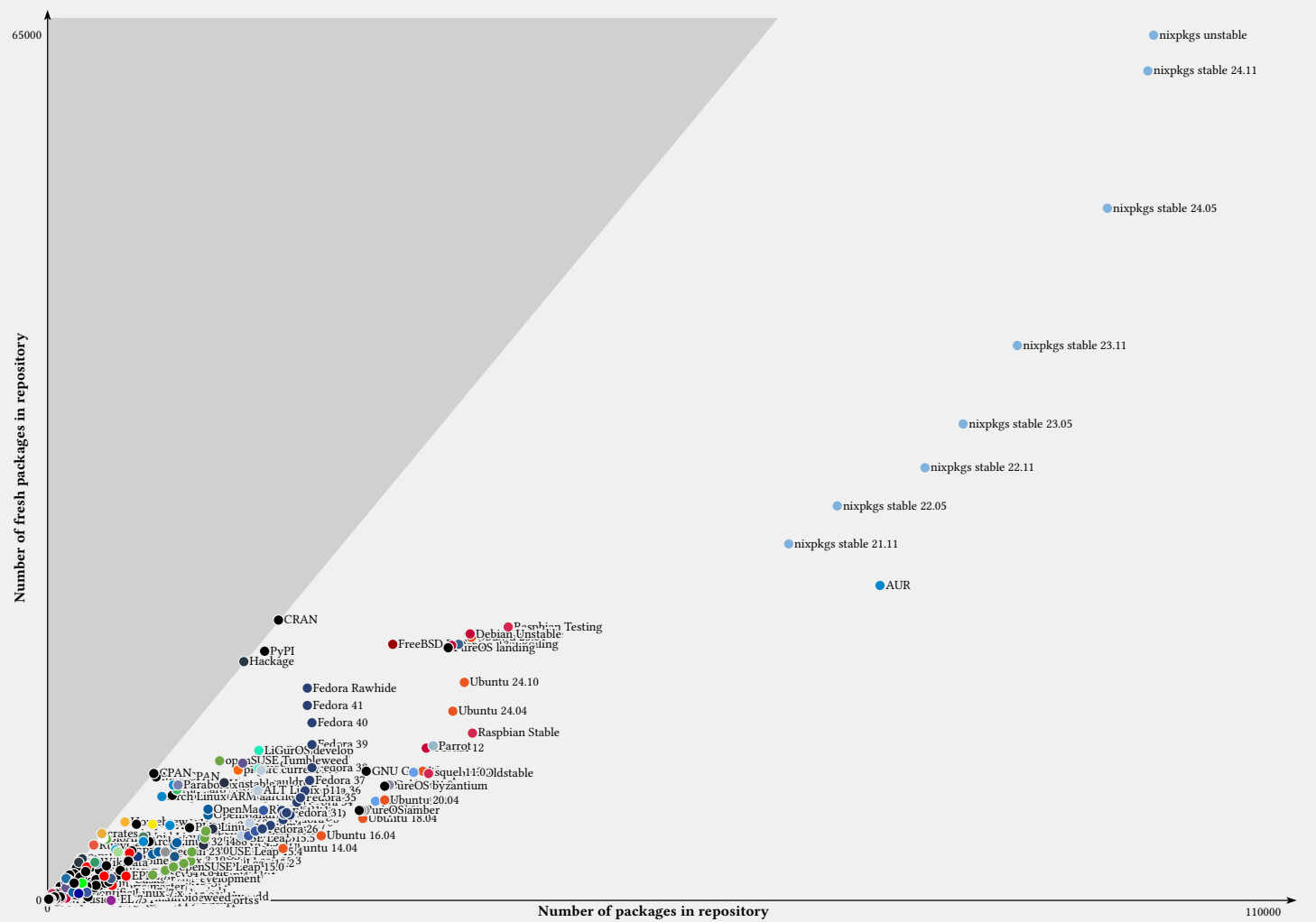
Nixpkgs

- A human-curated collection of software
- All of the software has to build together (usually at the latest version)
- We can have multiple versions of the same software
- The sources of packages are fetched deterministically
- Updating the sources can be automated

Nixpkgs


20000
Number of fresh packages in repository
Number of packages in repository
40000
0
Raspbian Testing
Debian Unstable
Ubuntu 25.04
FreeBSD Ports
Kali Linux Rolling
PureOS landing
PyPI
Hackage
Ubuntu 24.10
Fedora Rawhide
Fedora 41
Ubuntu 24.04
Fedora 40
Raspbian Stable
Fedora 39
Parrot
Debian 12
LiGurOS develop
openSUSE Tumbleweed
Fedora 38
GNU Guix
pkgsrc current
ALT Sisyphus
Raspbian Oldstable
CPAN
MetaCPAN
Mageia cauldron
Fedora 37
Manjaro Unstable
Parabola
Manjaro Testing
ALT Linux p11
Fedora 36
PureOS byzantium
Fedora 35
Arch Linux ARM aarch64
Fedora 34
Ubuntu 20.04
OpenMandriva Cooker
OpenMandriva Rolling
Fedora 31
PureOS amber
Fedora 28
Ubuntu 18.04
Homebrew
ALT Linux p10
Fedora 27
PLD Linux
Mageia 9
Fedora 26
openSUSE Leap 15.6
Rosa 2016.1
crates.io
Void Linux x86_64
Ubuntu 16.04
BioArch
openSUSE Leap 15.5
Rosa 2014.1
OpenMandriva 4.3
RubyGems
Mageia 8
Ubuntu 14.04
Artix
openSUSE Leap 15.4
PCLinuxOS
Spack
Stackage LTS Haskell
Wikidata
Alpine Linux 3.19
T2 SDE
openSUSE Leap 15.3
openSUSE Leap 15.2
openSUSE Leap 15.0
Alpine Linux 3.18
openSUSE Leap 15.1
openSUSE Leap 42.3
Alpine Linux 3.15
Scientific Linux 7.x

Nixpkgs


65000
Number of fresh packages in repository
0
Number of packages in repository
110000
nixpkgs unstable
nixpkgs stable 24.11
nixpkgs stable 24.05
nixpkgs stable 23.11
nixpkgs stable 23.05
nixpkgs stable 22.11
nixpkgs stable 22.05
nixpkgs stable 21.11
AUR
CRAN
PyPI
Hackage
FreeBSD
Debian Unstable
PureOS landing
Raspbian Testing
Ubuntu 24.10
Ubuntu 24.04
Raspbian Stable
Fedora Rawhide
Fedora 41
Fedora 40
Fedora 39
Parrot 12
LiGurOS/develop
openSUSE Tumbleweed
CPAN PAN
Fedora 37
PureOS byzantium
Ubuntu 20.04
Ubuntu 18.04
Ubuntu 16.04
Ubuntu 14.04
openSUSE Leap 15.0

NixOS

- Software can also have environment requirements at run-time

- Software can also have environment requirements at run-time
  - Configuration files

- Software can also have environment requirements at run-time
  - Configuration files
  - Services such as databases

- Software can also have environment requirements at run-time
  - Configuration files
  - Services such as databases
  - Hardware state such as network configuration

- Software can also have environment requirements at run-time
  - Configuration files
  - Services such as databases
  - Hardware state such as network configuration
- How can we ensure that the environment meets the needs of our software?

- Software can also have environment requirements at run-time
  - Configuration files
  - Services such as databases
  - Hardware state such as network configuration
- How can we ensure that the environment meets the needs of our software?
  - Construct the entire environment in a disciplined way

- NixOS is merely a derivation that creates all the files required to run an OS

- NixOS is merely a derivation that creates all the files required to run an OS
  - bootloader, kernel, init process, service manager, configuration files, …

- NixOS is merely a derivation that creates all the files required to run an OS
  - bootloader, kernel, init process, service manager, configuration files, …
- There are multiple choices for each of these components

NixOS

- NixOS is merely a derivation that creates all the files required to run an OS
  - bootloader, kernel, init process, service manager, configuration files, …
- There are multiple choices for each of these components
- How do we make sure they all work together?

NixOS

- NixOS is merely a derivation that creates all the files required to run an OS
  - bootloader, kernel, init process, service manager, configuration files, …
- There are multiple choices for each of these components
- How do we make sure they all work together?
  - Model the choices as constraints, use constraint solving to find a solution

- The module system is a DSL embedded in the Nix language

- The module system is a DSL embedded in the Nix language
- Its purpose is to build a big data structure from multiple interdependent declarations

- Features of the module system:

- Features of the module system:
  - Separate **declaration** and **definition** of configuration options

- Features of the module system:
  - Separate **declaration** and **definition** of configuration options
  - A **type system** that constrains the values of options

- Features of the module system:
  - Separate **declaration** and **definition** of configuration options
  - A **type system** that constrains the values of options
  - Multiple definitions of the same option are **merged** according to the type of that option

- Features of the module system:
  - Separate **declaration** and **definition** of configuration options
  - A **type system** that constrains the values of options
  - Multiple definitions of the same option are **merged** according to the type of that option
- Configuration options can refer to each other via fix-point computation

```
{ lib, ... }:
{
  options = {
    name = lib.mkOption {
      type = lib.types.str;
    };
  };
}
```

```nix
{ lib, ... }:
{
  options = {
    name = lib.mkOption {
      type = lib.types.str;
    };
  };
}
```

```nix
{ ... }:
{
  config = {
    name = "Boaty McBoatface";
  };
}
```

```nix
{ lib, ... }:
{
  options = {
    name = lib.mkOption {
      type = lib.types.str;
    };
  };
}
```

```nix
{ ... }:
{
  config = {
    name = "Boaty McBoatface";
  };
}
```

```nix
let
  lib = import <nixpkgs/lib>;
  result = lib.evalModules {
    modules = [
      ./options.nix
      ./config.nix
    ];
  };
in
result.config
```

- The collection of NixOS modules is a uniform interface for configuring an entire OS

- The collection of NixOS modules is a uniform interface for configuring an entire OS
- The data structure produced by the module system is the final configuration for our OS

Example NixOS configuration

```
{ config, pkgs, ... }:
{
  imports = [
    # Include the results of the hardware scan.
    ./hardware-configuration.nix
  ];

  # Enable the OpenSSH server.
  services.sshd.enable = true;

  # Install GNOME desktop environment
  services.xserver.enable = true;
  services.xserver.displayManager.gdm.enable = true;
  services.xserver.desktopManager.gnome.enable = true;

  # Use nVidia drivers
  nixpkgs.config.allowUnfree = true;
  services.xserver.videoDrivers = [ "nvidia" ];

  # Set up the firewall for HTTP
  networking.firewall.allowedTCPPorts = [ 80 443 ];
}
```

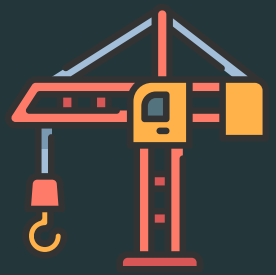
Intermezzo

We'll return after these messages!

Intermezzo

1. Install Nix
   https://nixos.org/download.html
2. Enable flakes
   ```
   mkdir -p ~/.config/nix
   echo "experimental-features = nix-command flakes" > ~/.config/nix/nix.conf
   ```
3. Test it out
   ```
   nix run nixpkgs#hello
   ```

Construction ahead!



Flakes are still a work in progress, and small details of their design may change in the future.

Nix CLI

Nix shell

```
$ git --version
```

Nix shell

```
$ git --version
git version 2.51.2
```

Nix shell

```
$ git --version
git version 2.51.2

$ fortune
```

Nix shell

```
$ git --version
git version 2.51.2

$ fortune
zsh: fortune: command not found
```

Nix shell

```
$ git --version
git version 2.51.2

$ fortune
zsh: fortune: command not found

$ nix run nixpkgs#fortune
```

Nix shell

```
$ git --version
git version 2.51.2

$ fortune
zsh: fortune: command not found

$ nix run nixpkgs#fortune
A banker is a fellow who lends you his umbrella when the sun is shining and
wants it back the minute it begins to rain.
  -- Mark Twain
```

Nix shell

```
$ nix run nixpkgs#fortune | nix run nixpkgs#cowsay
```

Nix shell

```
$ nix run nixpkgs#fortune | nix run nixpkgs#cowsay
 _______________________________________
< Offer void where prohibited by law. >
 ---------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Nix shell

```
$ nix shell nixpkgs#fortune nixpkgs#cowsay
$ fortune | cowsay
```

Nix shell

```
$ nix shell nixpkgs#fortune nixpkgs#cowsay
$ fortune | cowsay
 _________________________________________
/ He who has imagination without learning \
\ has wings but no feet.                  /
 -----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Nix shell

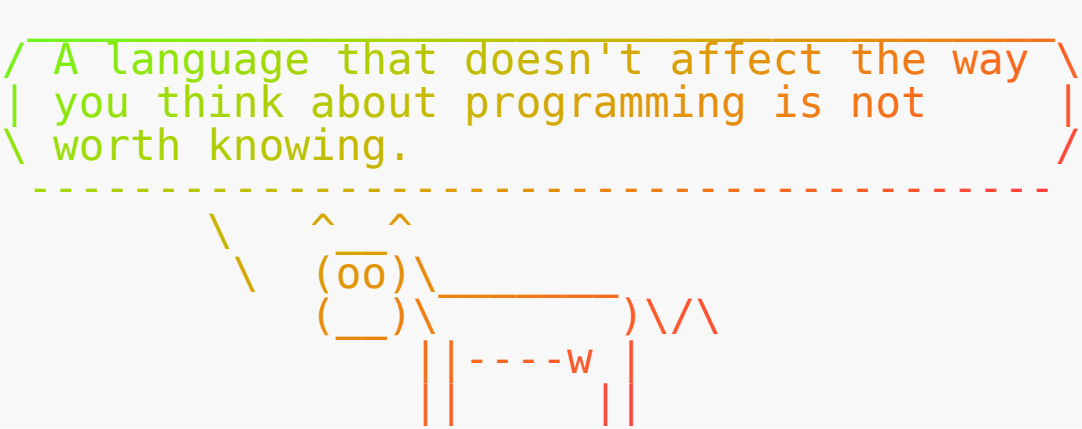
```
$ nix shell nixpkgs#fortune nixpkgs#cowsay
$ fortune | cowsay
 _________________________________________
/ He who has imagination without learning \
\ has wings but no feet.                  /
 -----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
$ exit
```

Nix shell

```
$ nix shell nixpkgs#{fortune,cowsay,lolcat}
$ fortune | cowsay | lolcat
```

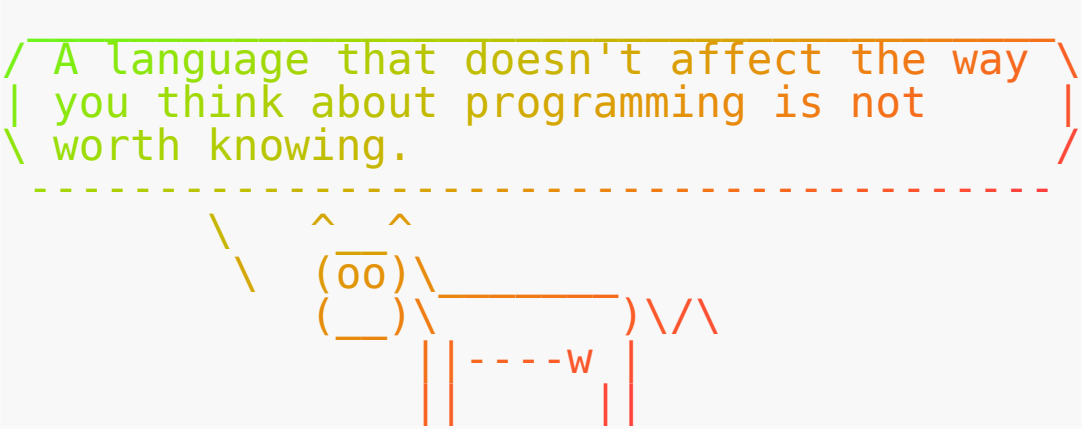
Nix shell

```
$ nix shell nixpkgs#{fortune,cowsay,lolcat}
$ fortune | cowsay | lolcat
```



Nix shell

```
$ nix shell nixpkgs#{fortune,cowsay,lolcat}
$ fortune | cowsay | lolcat
```




```
$ exit
```

Nix projects

Nix files

- A project has Nix powers if it has one or more of these files in its root:
  - `flake.nix` (new: build and shell)
  - `default.nix` (old: just build)
  - `shell.nix` (old: just shell)

Nix files

- A project has Nix powers if it has one or more of these files in its root:
  - `flake.nix` (new: build and shell)
  - `default.nix` (old: just build)
  - `shell.nix` (old: just shell)
- Ideally no other changes to the project are required!

My first flake

```
{
    inputs = {};

    outputs = inputs: {};
}
```

My first flake

```
{
    inputs = {
        nixpkgs = {
            url = "github:nixos/nixpkgs/nixos-25.11";
        };
    };

    outputs = inputs: {};
}
```

My first flake

```
{
    inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

    outputs = inputs: {};
}
```

My first flake

```
{
    inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

    outputs = inputs: {
        hello = "world";
    };
}
```

My first flake

```
$ nix flake show
```

My first flake

```
$ nix flake show
git+file:/tmp/nix-lecture?dir=examples/flake-c
└───hello: unknown
```

My first flake

```
$ nix flake show
git+file:/tmp/nix-lecture?dir=examples/flake-c
└───hello: unknown

$ nix eval .#hello
```

My first flake

```
$ nix flake show
git+file:/tmp/nix-lecture?dir=examples/flake-c
└───hello: unknown

$ nix eval .#hello
"world"
```

My first flake

```
{
    inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

    outputs = inputs:
      let
        system = "x86_64-linux";
      in
      {
      };
}
```

Some other options for `system`:

- `x86_64-darwin`
- `aarch64-darwin`

My first flake

```
{
    inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

    outputs = inputs:
      let
        system = "x86_64-linux";
        pkgs = import inputs.nixpkgs { inherit system; };
      in
      {
      };
}
```

My first flake

```
{
    inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

    outputs = inputs:
      let
        system = "x86_64-linux";
        pkgs = import inputs.nixpkgs { inherit system; };
      in
      {
        devShells.${system}.default = pkgs.mkShell { ... };
      };
}
```

My first flake

```
{
    inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

    outputs = inputs:
      let
        system = "x86_64-linux";
        pkgs = import inputs.nixpkgs { inherit system; };
      in
      {
        devShells.${system}.default = pkgs.mkShell {
          packages = [
            pkgs.fortune
            pkgs.cowsay
            pkgs.lolcat
          ];
        };
      };
}
```

My first flake

```
{
    inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

    outputs = inputs:
      let
        system = "x86_64-linux";
        pkgs = import inputs.nixpkgs { inherit system; };
      in
      {
        devShells.${system}.default = pkgs.mkShell {
          packages = with pkgs; [
            fortune
            cowsay
            lolcat
          ];
        };
      };
}
```

My first flake

- If a project has a `flake.nix` file (new), enter its development shell with:

  ```
  $ nix develop
  ```

My first flake

- If a project has a `flake.nix` file (new), enter its development shell with:

  ```
  $ nix develop
  ```

- If a project has a `shell.nix` file (old), enter its development shell with:

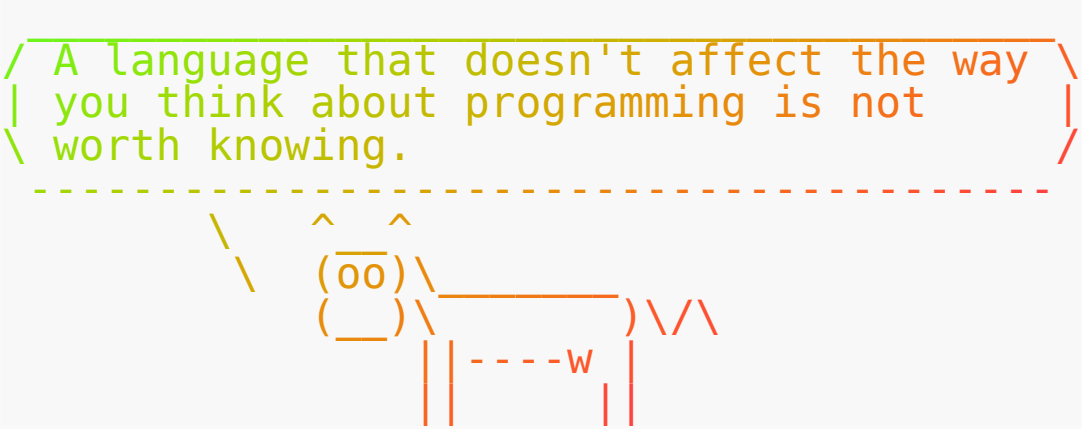
  ```
  $ nix-shell
  ```

My first flake

```
$ nix develop
$ fortune | cowsay | lolcat
```

My first flake

```
$ nix develop
$ fortune | cowsay | lolcat
```



Lockfiles

- Online resources change over time

Lockfiles

- Online resources change over time
- We wish to specify the exact version of flake inputs

Lockfiles

- Online resources change over time
- We wish to specify the exact version of flake inputs
  - But we also still want to easily update them …

Lockfiles

- Online resources change over time
- We wish to specify the exact version of flake inputs
  - But we also still want to easily update them …
- A Nix flake locks its inputs with a lockfile

Lockfiles

- Online resources change over time
- We wish to specify the exact version of flake inputs
  - But we also still want to easily update them …
- A Nix flake locks its inputs with a lockfile
  - Each input gets resolved to its current version and its contents are hashed

Lockfiles

- Online resources change over time
- We wish to specify the exact version of flake inputs
  - But we also still want to easily update them …
- A Nix flake locks its inputs with a lockfile
  - Each input gets resolved to its current version and its contents are hashed
  - The version and hash or each input are written into the lockfile

Lockfiles

- Online resources change over time
- We wish to specify the exact version of flake inputs
  - But we also still want to easily update them …
- A Nix flake locks its inputs with a lockfile
  - Each input gets resolved to its current version and its contents are hashed
  - The version and hash or each input are written into the lockfile
  - Every subsequent interaction with the flake will use the lockfile

Lockfiles


```
{
  "nodes": {
    "nixpkgs": {
      "locked": {
        "lastModified": 1765762245,
        "narHash": "sha256-3iXM/zTqEskWtmZs3gqNiVtRTsEjYAedIaLL0mSBsrk=",
        "owner": "nixos",
        "repo": "nixpkgs",
        "rev": "c8cfcd6ccd422e41cc631a0b73ed4d5a925c393d",
        "type": "github"
      },
      "original": {
        "owner": "nixos",
        "ref": "nixos-25.11",
        "repo": "nixpkgs",
        "type": "github"
      }
    },
    "root": {
      "inputs": {
        "nixpkgs": "nixpkgs"
      }
    }
  },
  "root": "root",
  "version": 7
}
```

Building a C project

```c
#include <stdio.h>

int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

Building a C project

```
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

  outputs = inputs:
    let
      system = "x86_64-linux";
      pkgs = import inputs.nixpkgs { inherit system; };
    in
    {
      packages.${system}.default = pkgs.stdenv.mkDerivation {
        name = "hello";
        src = ./.;
      };
    };
}
```

Building a C project

```
$ nix build
```

Building a C project

```
$ nix build
error: builder for '/nix/store/y4pwjdz73s1s1wmvsc5pnrx9va74d760-hello.drv'
failed to produce output path for output 'out' at '/nix/store/
y4pwjdz73s1s1wmvsc5pnrx9va74d760-hello.drv.chroot/root/nix/store/
wpwvpa6m5gq1fghqzbf7n4s7zbrzafzy-hello'
```

Building a C project

C projects are usually built with a Makefile!

```
build: hello

hello: hello.c
  $(CC) $(CFLAGS) -o hello hello.c

BINDIR ?= $(out)/bin
install: hello
  install -D --mode=755 hello ${BINDIR}/hello
```

Building a C project

```
$ nix build
```

Building a C project

```
$ nix build
$

$ nix run
```

Building a C project

```
$ nix build
$

$ nix run
Hello world!
```

Building a C project with dependencies

Let's add a depedency on Lua

```c
#include <stdio.h>
#include <lualib.h>
#include <lauxlib.h>

const char* s = "print('Hello from Lua!')";

int main() {
  lua_State* L = luaL_newstate();
  luaL_openlibs(L);
  luaL_dostring(L, s);
  lua_close(L);
}
```

Building a C project with dependencies

```
build: hello

hello: hello.c
  $(CC) $(CFLAGS) -llua -o hello hello.c

BINDIR ?= $(out)/bin
install: hello
  install -D --mode=755 hello ${BINDIR}/hello
```

Building a C project with dependencies

```
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

  outputs = inputs:
    let
      system = "x86_64-linux";
      pkgs = import inputs.nixpkgs { inherit system; };
    in
    {
      packages.${system}.default = pkgs.stdenv.mkDerivation {
        name = "hello";
        src = ./.;
      };
    };
}
```

Building a C project with dependencies

```
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

  outputs = inputs:
    let
      system = "x86_64-linux";
      pkgs = import inputs.nixpkgs { inherit system; };
    in
    {
      packages.${system}.default = pkgs.stdenv.mkDerivation {
        name = "hello";
        src = ./.;
        buildInputs = with pkgs; [ lua ];
      };
    };
}
```

Building a C project with dependencies

```
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

  outputs = inputs:
    let
      system = "x86_64-linux";
      pkgs = import inputs.nixpkgs { inherit system; };
    in
    {
      devShells.${system}.default = pkgs.mkShell {
        packages = with pkgs; [ lua ];
      };

      packages.${system}.default = pkgs.stdenv.mkDerivation {
        name = "hello";
        src = ./.;
        buildInputs = with pkgs; [ lua ];
      };
    };
}
```

Building a C project with dependencies

```
$ nix run
```

Building a C project with dependencies

```
$ nix run
Hello from Lua!
```

Python shell

```
import torch

print(torch.__version__)

# Create a scalar tensor
scalar_tensor = torch.tensor(42)
print(scalar_tensor)
```

Python shell

```
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

  outputs = inputs:
    let
      system = "x86_64-linux";
      pkgs = import inputs.nixpkgs { inherit system; };
    in
    {
      devShells.${system}.default = pkgs.mkShell {
        nativeBuildInputs = [
          (pkgs.python3.withPackages (ps: [ ps.torch ]))
        ];
      };
    };
}
```

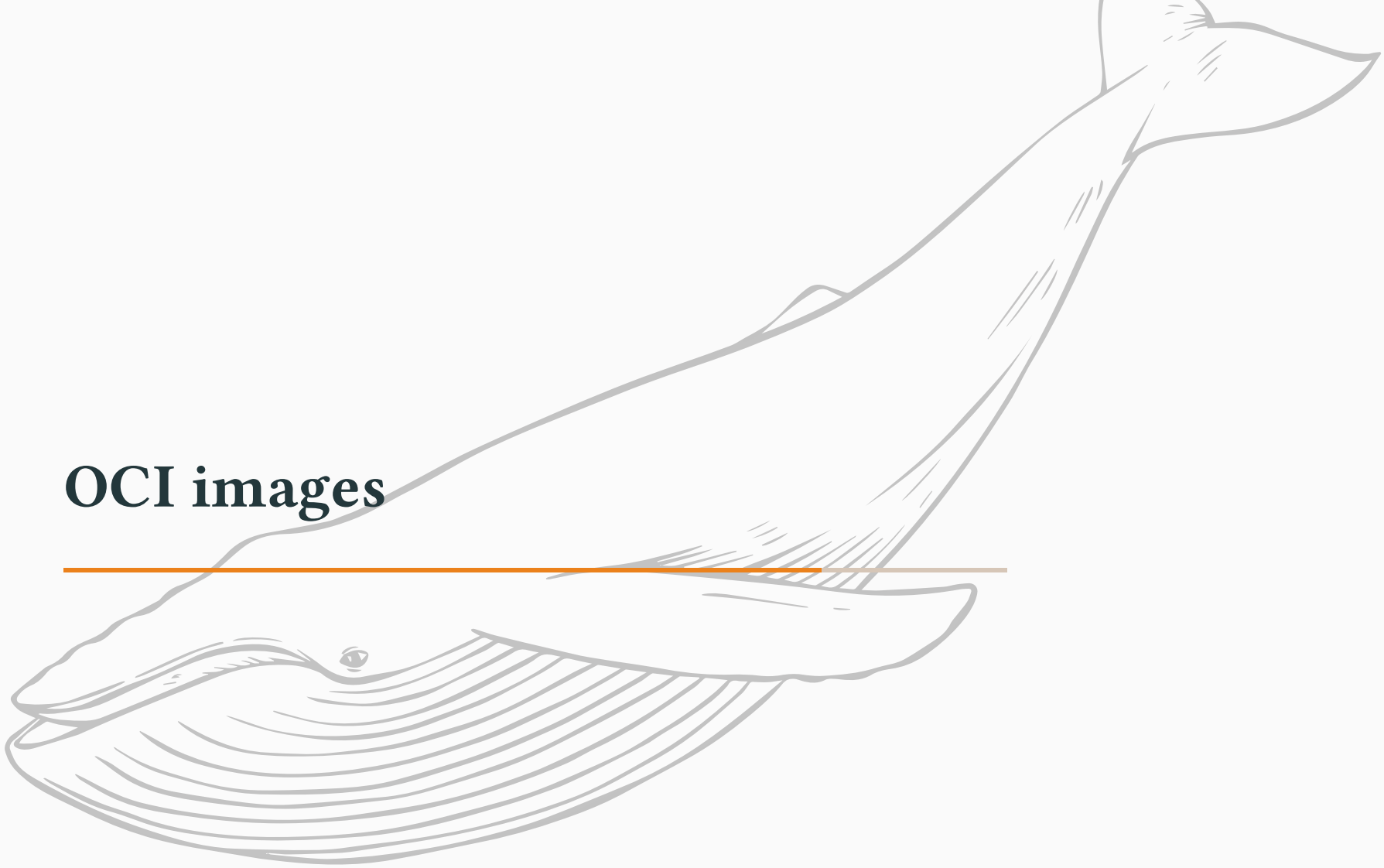
Python shell

```
$ nix develop --command python hello.py
```

Python shell

```
$ nix develop --command python hello.py
2.9.1
tensor(42)
```

OCI images



Building OCI images

- If Nix can build a package, it can also build an OCI image with it!

Building OCI images

- If Nix can build a package, it can also build an OCI image with it!
- Let's package a little Haskell server in Nix and Docker

Building a Haskell project

```
cabal-version: 3.0
name: hello
version: 0.1.0.0

executable hello
  hs-source-dirs: .
  main-is: Main.hs
  build-depends:
    base,
    http-types,
    wai,
    warp,
```

```
{-# LANGUAGE OverloadedStrings #-}

import Network.HTTP.Types
import Network.Wai
import Network.Wai.Handler.Warp

response =
    responseLBS
        status200
        [(hContentType, "text/plain")]
        "Hello Haskell!"

app _ f = f response

main = runSettings defaultSettings app
```

Building a Haskell project

```
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

  outputs = inputs:
    let
      system = "x86_64-linux";
      pkgs = import inputs.nixpkgs { inherit system; };
    in
    {
      packages.${system}.default =
        pkgs.haskellPackages.callCabal2nix "hello" ./. { };
    };
}
```

Building a Haskell project

```
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

  outputs = inputs:
    let
      system = "x86_64-linux";
      pkgs = import inputs.nixpkgs { inherit system; };
      hello = pkgs.haskellPackages.callCabal2nix "hello" ./. { };
    in
    {
      packages.${system}.default = hello;
    };
}
```

Building a Haskell project

```
$ nix run
```

Building a Haskell project

```
$ nix run
```

```
$ curl localhost:3000
```

Building a Haskell project

```
$ nix run
```

```
$ curl localhost:3000
Hello Haskell!
```

Building an OCI image

Let's inspect the runtime dependencies of our executable:

```
$ nix build
$ ldd result/bin/hello
```

Building an OCI image

Let's inspect the runtime dependencies of our executable:

```
$ nix build
$ ldd result/bin/hello
linux-vdso.so.1 (0x00007ff8ce323000)
libm.so.6 => /nix/store/xx7cm72qy2c0643cm1ipngd87aqwkcdp-glibc-2.40-66/lib/libm.so.6 (0x00007ff8ce233000)
libz.so.1 => /nix/store/l7xwm1f6f3zj2x8jwdbi8gdyfbx07sh7-zlib-1.3.1/lib/libz.so.1 (0x00007ff8ce215000)
libgmp.so.10 => /nix/store/54jkwsavi3fdciqfyjmbilq0jhvv4jga-gmp-with-cxx-6.3.0/lib/libgmp.so.10 (0x00007ff8ce16b000)
libc.so.6 => /nix/store/xx7cm72qy2c0643cm1ipngd87aqwkcdp-glibc-2.40-66/lib/libc.so.6 (0x00007ff8cde00000)
librt.so.1 => /nix/store/xx7cm72qy2c0643cm1ipngd87aqwkcdp-glibc-2.40-66/lib/librt.so.1 (0x00007ff8ce164000)
libdl.so.2 => /nix/store/xx7cm72qy2c0643cm1ipngd87aqwkcdp-glibc-2.40-66/lib/libdl.so.2 (0x00007ff8ce15f000)
libffi.so.8 => /nix/store/b9p0zpa93hwvh4d0r1rmgc2500yx2ldn-libffi-3.5.2/lib/libffi.so.8 (0x00007ff8ce14e000)
libelf.so.1 => /nix/store/1nqqjacc6dnj61jlpgz5hk9zdjbfidbr-elfutils-0.194/lib/libelf.so.1 (0x00007ff8ce131000)
libdw.so.1 => /nix/store/1nqqjacc6dnj61jlpgz5hk9zdjbfidbr-elfutils-0.194/lib/libdw.so.1 (0x00007ff8ce080000)
libnuma.so.1 => /nix/store/gdni20c8009xdz8gms6yn1r2hfhmk1jk-numactl-2.0.18/lib/libnuma.so.1 (0x00007ff8ce06f000)
/nix/store/xx7cm72qy2c0643cm1ipngd87aqwkcdp-glibc-2.40-66/lib/ld-linux-x86-64.so.2 => /nix/store/xx7cm72qy2c0643cm1ipngd87aqwkcdp-
glibc-2.40-66/lib64/ld-linux-x86-64.so.2 (0x00007ff8ce325000)
libzstd.so.1 => /nix/store/s7vmxmhkq439cjb7ag9w198p6dk7kl0w-zstd-1.5.7/lib/libzstd.so.1 (0x00007ff8cdd27000)
liblzma.so.5 => /nix/store/q5vlz5jl6p7mv220s2vf6z5pqi1n935z-xz-5.8.1/lib/liblzma.so.5 (0x00007ff8ce03d000)
libbz2.so.1 => /nix/store/xgavznqg1ay2hycpp7yy9ia1n751jcla-bzip2-1.0.8/lib/libbz2.so.1 (0x00007ff8ce029000)
libpthread.so.0 => /nix/store/xx7cm72qy2c0643cm1ipngd87aqwkcdp-glibc-2.40-66/lib/libpthread.so.0 (0x00007ff8ce022000)
libatomic.so.1 => /nix/store/xm08aqdd7pxcdhm0ak6aqb1v7hw5q6ri-gcc-14.3.0-lib/lib/libatomic.so.1 (0x00007ff8ce016000)
```

That's a lot of dependencies! Sure would be a shame if we forgot some...

Building an OCI image

```
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

  outputs = inputs:
    let
      system = "x86_64-linux";
      pkgs = import inputs.nixpkgs { inherit system; };
      hello = pkgs.haskellPackages.callCabal2nix "hello" ./. { };
    in
    {
      packages.${system}.default = hello;
    };
}
```

Building an OCI image

```
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-25.11";

  outputs = inputs:
    let
      system = "x86_64-linux";
      pkgs = import inputs.nixpkgs { inherit system; };
      hello = pkgs.haskellPackages.callCabal2nix "hello" ./. { };
      image = pkgs.dockerTools.buildLayeredImage {
        name = hello.pname;
        config.Cmd = [ hello.meta.mainProgram ];
        contents = [ hello ];
        tag = "latest";
      };
    in
    {
      packages.${system} = {
        default = hello;
        image = image;
      };
    };
}
```

Building an OCI image

```
$ nix build .#image
```

Building an OCI image

```
$ nix build .#image
$ ls -ahl result
```

Building an OCI image

```
$ nix build .#image
$ ls -ahl result
lrwxrwxrwx 1 ners ners 56 Dez 16 08:40 result -> /nix/store/
gljsmlzgwzfpggrxa1jnlj0lhgv1mrix-hello.tar.gz
```

Building an OCI image

```
$ nix build .#image
$ ls -ahl result
lrwxrwxrwx 1 ners ners 56 Dez 16 08:40 result -> /nix/store/
gljsmlzgwzfpggrxa1jnlj0lhgv1mrix-hello.tar.gz
$ podman load -i result
```

Building an OCI image

```
$ nix build .#image
$ ls -ahl result
lrwxrwxrwx 1 ners ners 56 Dez 16 08:40 result -> /nix/store/
gljsmlzgwzfpggrxa1jnlj0lhgv1mrix-hello.tar.gz
$ podman load -i result
$ podman run --publish 3000:3000 hello
```

Building an OCI image

```
$ nix build .#image
$ ls -ahl result
lrwxrwxrwx 1 ners ners 56 Dez 16 08:40 result -> /nix/store/
gljsmlzgwzfpggrxa1jnlj0lhgv1mrix-hello.tar.gz
$ podman load -i result
$ podman run --publish 3000:3000 hello

$ curl localhost:3000
```

Building an OCI image

```
$ nix build .#image
$ ls -ahl result
lrwxrwxrwx 1 ners ners 56 Dez 16 08:40 result -> /nix/store/
gljsmlzgwzfpggrxa1jnlj0lhgv1mrix-hello.tar.gz
$ podman load -i result
$ podman run --publish 3000:3000 hello

$ curl localhost:3000
Hello Haskell!
```

NixOS tests

NixOS tests

- NixOS tests set up a network of NixOS-powered virtual machines

NixOS tests

- NixOS tests set up a network of NixOS-powered virtual machines
- These virtual machines can run any program and communicate with each other over the network

NixOS tests

- NixOS tests set up a network of NixOS-powered virtual machines
- These virtual machines can run any program and communicate with each other over the network
- This is especially useful for client-server tests!

Let's test our Haskell app

```
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-24.11";

  outputs = inputs:
    let
      system = "x86_64-linux";
      pkgs = import inputs.nixpkgs { inherit system; };
      hello = pkgs.haskellPackages.callCabal2nix "hello" ./. { };
    in
    {
      packages.${system}.default = hello;
    };
}
```

NixOS tests

```nix
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-24.11";

  outputs = inputs:
    let
      system = "x86_64-linux";
      pkgs = import inputs.nixpkgs { inherit system; };
      hello = pkgs.haskellPackages.callCabal2nix "hello" ./. { };
    in
    {
      packages.${system}.default = hello;
      checks.${system}.nixosTest = pkgs.testers.nixosTest { ... };
    };
}
```

NixOS tests

```
pkgs.testers.nixosTest {
    name = "hello-test";
    nodes = { ... };
    testScript = "...";
};
```

NixOS tests

```
pkgs.testers.nixosTest {
  name = "hello-test";
  nodes = {
    server = { ... };
    client = { ... };
  };
  testScript = "...";
};
```

NixOS tests

```
pkgs.testers.nixosTest {
  name = "hello-test";
  nodes = {
    server = {
      networking.firewall.allowedTCPPorts = [ 3000 ];
      systemd.services.server = {
        wantedBy = [ "multi-user.target" ];
        path = [ hello ];
        script = hello.meta.mainProgram;
      };
    };
    client = { ... };
  };
  testScript = "...";
};
```

NixOS tests

```
pkgs.testers.nixosTest {
  name = "hello-test";
  nodes = {
    server = {
      networking.firewall.allowedTCPPorts = [ 3000 ];
      systemd.services.server = {
        wantedBy = [ "multi-user.target" ];
        path = [ hello ];
        script = hello.meta.mainProgram;
      };
    };
    client = {
      environment.systemPackages = [ pkgs.curl ];
    };
  };
  testScript = "...";
};
```

NixOS tests

```
pkgs.testers.nixosTest {
  name = "hello-test";
  nodes = {
    server = {
      networking.firewall.allowedTCPPorts = [ 3000 ];
      systemd.services.server = {
        wantedBy = [ "multi-user.target" ];
        path = [ hello ];
        script = hello.meta.mainProgram;
      };
    };
    client = {
      environment.systemPackages = [ pkgs.curl ];
    };
  };
  testScript = ''
    start_all()
    server.wait_for_open_port(3000)
    expected = "Hello Haskell!"
    actual = client.succeed("curl http://server:3000")
    assert expected == actual, "server says hello"
  '';
};
```

NixOS tests

```
$ nix build --print-build-logs .#checks.x86_64-linux.nixosTest
```

NixOS tests

```
$ nix build --print-build-logs .#checks.x86_64-linux.nixosTest
... lots of terminal output ...
```

NixOS tests

```
$ nix build --print-build-logs .#checks.x86_64-linux.nixosTest
... lots of terminal output ...
$
```

NixOS tests

```
$ nix build --print-build-logs .#checks.x86_64-linux.nixosTest
... lots of terminal output ...
$

$ echo $?
```

NixOS tests

```
$ nix build --print-build-logs .#checks.x86_64-linux.nixosTest
... lots of terminal output ...
$

$ echo $?
0
```

That’s all, folks!

Stay in touch

https://zurich.nix.ug