Next Steps for Reproducible Testing and Deployment

"The attempt to rewrite the foundations of mathematics in terms of category theory is evil and wrong." –Walt Pohl[1]

Ananth Venkatesh

MIT Arcturus Autonomy

April 11, 2025

[1] Attribution: (Pohl 2005)

Outline

1.1 Nix as the Purely Functional, Monadic Build Tool

- Software directly controls hardware

1.1 Nix as the Purely Functional, Monadic Build Tool

- Software directly controls hardware

(or so we think)

- In reality, software *can't execute itself*

1.1 Nix as the Purely Functional, Monadic Build Tool

- Software directly controls hardware

(or so we think)

- In reality, software *can't execute itself*
- This is the purpose of a **build tool**

1.1 Nix as the Purely Functional, Monadic Build Tool

- Software directly controls hardware

(or so we think)

- In reality, software *can't execute itself*
- This is the purpose of a **build tool**

**We want a way to tell computers how to run software,
and we want software to run the same way on any system, regardless of its internal state.**

1.1 Nix as the Purely Functional, Monadic Build Tool

- This is a famously non-trivial problem

1.1 Nix as the Purely Functional, Monadic Build Tool

- This is a famously non-trivial problem
- Software has dependencies (for compilation, development, and in production), configurations, targets (supported systems), feasibility relations, and a bunch of other stuff we don't know about ("the world")

1.1 Nix as the Purely Functional, Monadic Build Tool

- This is a famously non-trivial problem
  - Software has dependencies (for compilation, development, and in production), configurations, targets (supported systems), feasibility relations, and a bunch of other stuff we don't know about ("the world")
- Dealing with dependencies is a very serious problem (packages will require conflicting versions of the same package)

Eelco Dolstra, original Nix theorist

1.1 Nix as the Purely Functional, Monadic Build Tool

- We want a function f that takes a set of dependencies X and maps it to a fully working implementation FY

1.1 Nix as the Purely Functional, Monadic Build Tool

- We want a function f that takes a set of dependencies X and maps it to a fully working implementation FY
  - X represents any arbitrary list of packages, and F is a transformation on the type Y of package that includes "the world" generated in the process of trying to build this package

1.1 Nix as the Purely Functional, Monadic Build Tool

- We want a function f that takes a set of dependencies X and maps it to a fully working implementation FY
  - X represents any arbitrary list of packages, and F is a transformation on the type Y of package that includes "the world" generated in the process of trying to build this package
  - FY contains both a resulting package and a bunch of metadata/configuration/dependencies created along with that package

1.1 Nix as the Purely Functional, Monadic Build Tool

- We want a function f that takes a set of dependencies X and maps it to a fully working implementation FY
  - X represents any arbitrary list of packages, and F is a transformation on the type Y of package that includes "the world" generated in the process of trying to build this package
  - FY contains both a resulting package and a bunch of metadata/configuration/dependencies created along with that package
- We have a new package Z that depends on Y but doesn't care about all this random junk produced in the process of building Y

1.1 Nix as the Purely Functional, Monadic Build Tool

- We want a function f that takes a set of dependencies X and maps it to a fully working implementation FY
  - X represents any arbitrary list of packages, and F is a transformation on the type Y of package that includes "the world" generated in the process of trying to build this package
  - FY contains both a resulting package and a bunch of metadata/configuration/dependencies created along with that package
- We have a new package Z that depends on Y but doesn't care about all this random junk produced in the process of building Y
  - It has a build function that looks like $g : Y \to FZ$

1.1 Nix as the Purely Functional, Monadic Build Tool

- There is an indirect relationship between Z and X, and we want to find $h : X \to FZ$

1.1 Nix as the Purely Functional, Monadic Build Tool

- There is an indirect relationship between Z and X, and we want to find $h : X \to FZ$
  - $h := f \gg= g$

1.1 Nix as the Purely Functional, Monadic Build Tool

- There is an indirect relationship between Z and X, and we want to find $h : X \to FZ$
  - $h := f \gg= g$
  - The so-called “monadic bind”

1.1 Nix as the Purely Functional, Monadic Build Tool

- There is an indirect relationship between Z and X, and we want to find $h : X \to FZ$
  - $h := f \gg= g$
  - The so-called "monadic bind"
- We want a **monadic** build system ("The Postmodern Build System" 2023)

1.1 Nix as the Purely Functional, Monadic Build Tool

- There is an indirect relationship between Z and X, and we want to find $h : X \to FZ$
  - $h := f \gg= g$
  - The so-called "monadic bind"
- We want a **monadic** build system ("The Postmodern Build System" 2023)
- We want **optimized incremental** builds (i.e. the bind operation should be fast and builds should be **pure**)
  - Same input, same output

1.1 Nix as the Purely Functional, Monadic Build Tool

- Some nice properties of **pure builds**
  (the formalism we've been developing for talking about software deployment):

1.1 Nix as the Purely Functional, Monadic Build Tool

- Some nice properties of **pure builds**
  (the formalism we've been developing for talking about software deployment):
  - Minimal recomputation
    - only the specific dependencies that are changed need to be rebuilt

1.1 Nix as the Purely Functional, Monadic Build Tool

- Some nice properties of **pure builds**
  (the formalism we've been developing for talking about software deployment):
  - Minimal recomputation
    - only the specific dependencies that are changed need to be rebuilt
  - Distributed builds
    - all systems produce the same binaries

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

- Nix is not a dogmatic philosophy but the result of logical deliberation over the optimal dependency management scheme

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

- Nix is not a dogmatic philosophy but the result of logical deliberation over the optimal dependency management scheme
- Nix is the unique existing build system satisfying both a **reproducibility criterion** and having **high interpretability**

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

Figure 1: This is absolutely not true

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

1.2.1 Reproducibility

- We like

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

1.2.2 Reproducibility

- We like
- Build twice. Any system, anywhere, any time, any [insert here].
  Same result.

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

1.2.3 Reproducibility

- We like
- Build twice. Any system, anywhere, any time, any [insert here].
  Same result.
- Everything can be **built from source**
  - Critical for OPSEC

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

1.2.4 Reproducibility

- We like
- Build twice. Any system, anywhere, any time, any [insert here].
  Same result.
- Everything can be **built from source**
  - Critical for OPSEC
- Stuff just works.™

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

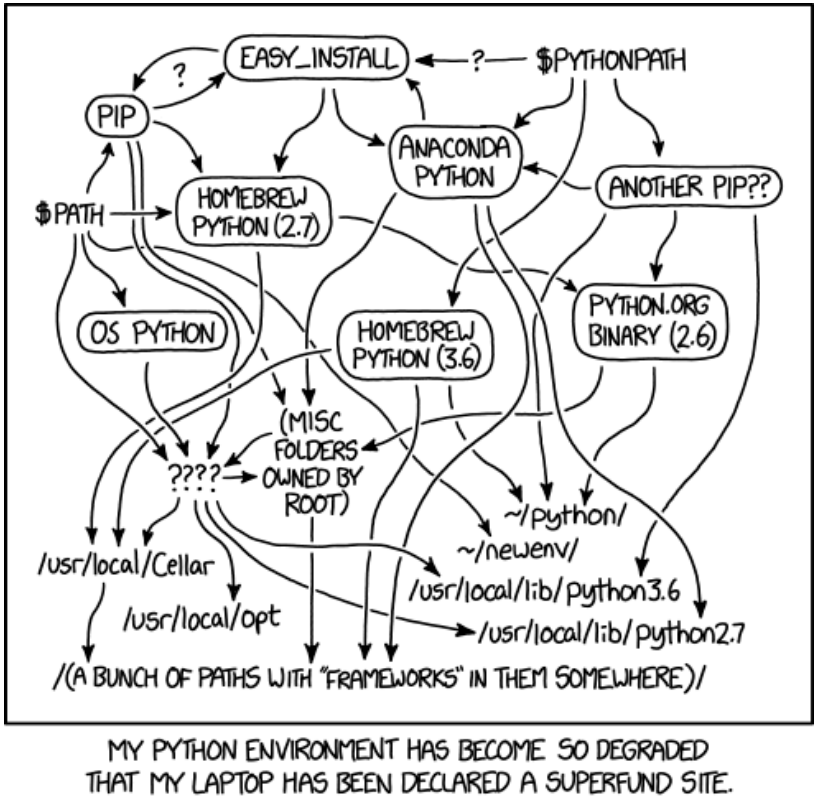

Figure 2: Other so-called "package managers"

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

1.2.5 *Docker enjoyers may object to the previous discussion* (Crane 2022)

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

1.2.6 *Docker enjoyers may object to the previous discussion* (Crane 2022)

There are some important differences, however (this is the **interpretability criterion**)

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

1.2.7 *Docker enjoyers may object to the previous discussion* (Crane 2022)

There are some important differences, however (this is the **interpretability criterion**)

- Docker is **slow**, Nix is **fast**
  - About 5x faster compared to the ROS development environment (Purvis and Wanders 2022)

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

1.2.8 *Docker enjoyers may object to the previous discussion* (Crane 2022)

There are some important differences, however (this is the **interpretability criterion**)

- Docker is **slow**, Nix is **fast**
  - About 5x faster compared to the ROS development environment (Purvis and Wanders 2022)
- Docker comes with supply-side **bloat**, Nix comes with a **minimum package set**

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

1.2.9 *Docker enjoyers may object to the previous discussion* (Crane 2022)

There are some important differences, however (this is the **interpretability criterion**)

- Docker is **slow**, Nix is **fast**
  - About 5x faster compared to the ROS development environment (Purvis and Wanders 2022)
- Docker comes with supply-side **bloat**, Nix comes with a **minimum package set**
- Docker is **stateful**, Nix is **stateless**

1.2.10 Docker

Reproduce, exactly, the entire system **state** of a working environment

1.2.11 Docker

Reproduce, exactly, the entire system **state** of a working environment

For Arcturus, this isn't even the same as our deployment environment! This is **very bad**!

1.2.12 Docker

Reproduce, exactly, the entire system **state** of a working environment

For Arcturus, this isn't even the same as our deployment environment! This is **very bad**!

We don't know **why or how stuff works**, only that it works on a given system. Equivalently, any change to the system state (Docker configuration) could lead to critical build failures.

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

1.2.13 Nix

Reproduce, exactly, the minimum set of **components** (packages and related configurations) that define a working environment

1.2 Nix as the Fully Reproducible, Interpretable Build Tool

1.2.14 Nix

Reproduce, exactly, the minimum set of **components** (packages and related configurations) that define a working environment

We can ask questions like *which packages does this module depend on?* or *why did this new build fail?*

1.3 Nix as as the Universal Build Tool

"The category of food, FÜD, is a full subcategory of the category of stuff, STÜF, whose objects are the stuff you can eat. A morphism of this category is known as a recipe."

—Ed Morehouse (Morehouse 2015)

1.3 Nix as as the Universal Build Tool

"The category of food, FÜD, is a full subcategory of the category of stuff, STÜF, whose objects are the stuff you can eat. A morphism of this category is known as a recipe."

—Ed Morehouse (Morehouse 2015)

Nix is often described as a bad implementation of a good theory, mainly because it has actually been implemented and no longer exists in the minds of category theorists

1.3 Nix as as the Universal Build Tool

1.3.1 Nix can run anywhere natively!

(Except Windows, but it can run on WSL)

- Any project, any dependency, any version

1.3 Nix as as the Universal Build Tool

1.3.2 Nix can run anywhere natively!

(Except Windows, but it can run on WSL)

- Any project, any dependency, any version
- Nix is extensible enough to describe build recipes for a wide variety of projects, and ROS is no exception

1.3 Nix as as the Universal Build Tool

1.3.3 Nix can run anywhere natively!

(Except Windows, but it can run on WSL)

- Any project, any dependency, any version
- Nix is extensible enough to describe build recipes for a wide variety of projects, and ROS is no exception
- However, ROS uses infamously bad dependency specifications and runtime environments

Outline

2.1 Non-invasive, isolated, and verifiable

Due to the severe shock that would result from hacking on Nix code in the primary codebase (as is best practice), the Nix build system has been developed completely independently.

2.1 Non-invasive, isolated, and verifiable

Due to the severe shock that would result from hacking on Nix code in the primary codebase (as is best practice), the Nix build system has been developed completely independently.

- **Non-invasive** (using Nix does not interfere with existing working build environments)

2.1 Non-invasive, isolated, and verifiable

Due to the severe shock that would result from hacking on Nix code in the primary codebase (as is best practice), the Nix build system has been developed completely independently.

- **Non-invasive** (using Nix does not interfere with existing working build environments)
- **Isolated** (self-contained, use it only when you want)

2.1 Non-invasive, isolated, and verifiable

Due to the severe shock that would result from hacking on Nix code in the primary codebase (as is best practice), the Nix build system has been developed completely independently.

- **Non-invasive** (using Nix does not interfere with existing working build environments)
- **Isolated** (self-contained, use it only when you want)
- **Verifiable** (benefit from safe binary caching and reproduce tests on other systems)

2.1 Non-invasive, isolated, and verifiable

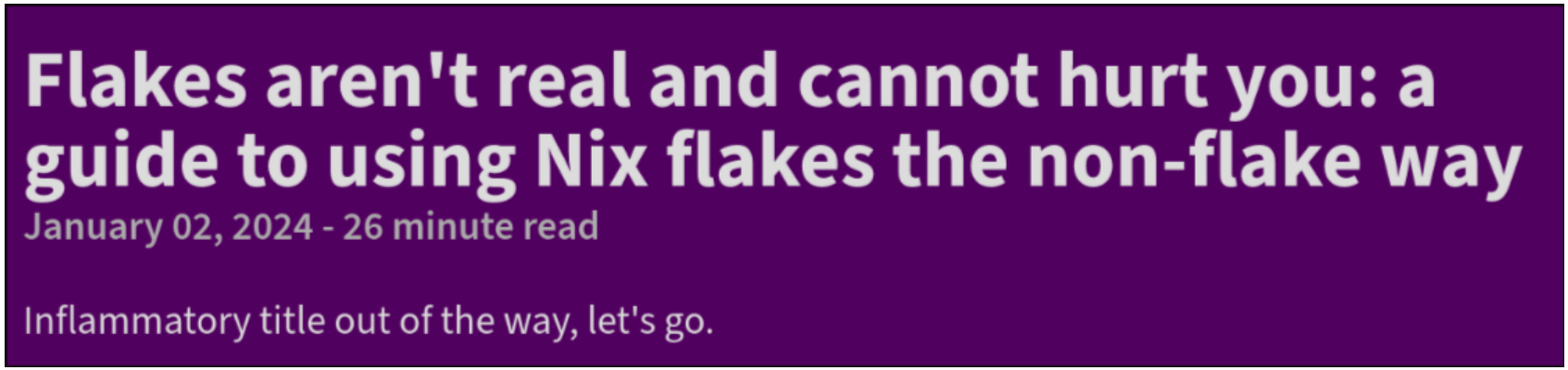

Figure 1: tl;dr: you should be using Nix already

2.2 Where we are in relation to the literature

2.2.1 Where we are:

- Clear dependency specifications for each module in the main codebase

2.2 Where we are in relation to the literature

2.2.2 Where we are:

- Clear dependency specifications for each module in the main codebase
- A reproducible testing environment for individual nodes

2.2 Where we are in relation to the literature

2.2.3 Where we are:

- Clear dependency specifications for each module in the main codebase
- A reproducible testing environment for individual nodes
- Preliminary implementation of a binary cache

2.2 Where we are in relation to the literature

2.2.4 The literature:

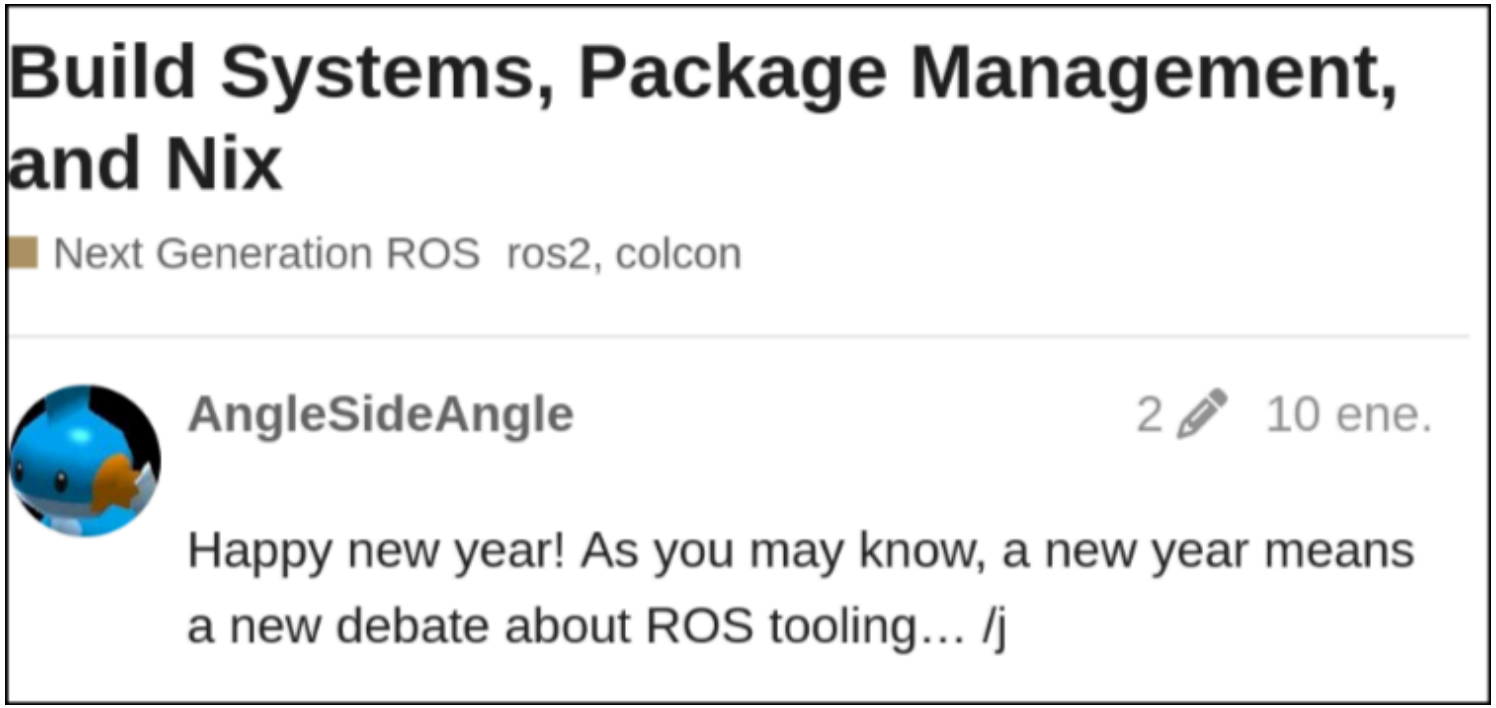

Figure 2: Grade A level trolling by Motorsports

2.2 Where we are in relation to the literature

Nix is not (only) being promoted by category theorists (and other students of abstract nonsense)

2.2 Where we are in relation to the literature

Nix is not (only) being promoted by category theorists (and other students of abstract nonsense)

- Presented at ROSCon 2022 (Purvis and Wanders 2022)

2.2 Where we are in relation to the literature

Nix is not (only) being promoted by category theorists (and other students of abstract nonsense)

- Presented at ROSCon 2022 (Purvis and Wanders 2022)
- Work undergoing to rewrite ros2nix in Motorsports

2.2 Where we are in relation to the literature

Nix is not (only) being promoted by category theorists (and other students of abstract nonsense)

- Presented at ROSCon 2022 (Purvis and Wanders 2022)
- Work undergoing to rewrite ros2nix in Motorsports
- Failure to keep pace with these developments reduces our research competitiveness, puts our build system in jeopardy of deprecation, and leaves our OPSEC vulnerable to upstream security issues

Outline

3.1 The work ahead of us

- Fully realize Nix speed benefits by **caching everything**

3.1 The work ahead of us

- Fully realize Nix speed benefits by **caching everything**
  - Locally hosted, on a machine running NixOS

3.1 The work ahead of us

- Fully realize Nix speed benefits by **caching everything**
  - Locally hosted, on a machine running NixOS
- Automatic module build tests with Nix (#107)

3.1 The work ahead of us

- Fully realize Nix speed benefits by **caching everything**
  - Locally hosted, on a machine running NixOS
- Automatic module build tests with Nix (#107)
  - Will add unit testing and formal verification

3.1 The work ahead of us

- Fully realize Nix speed benefits by **caching everything**
  - Locally hosted, on a machine running NixOS
- Automatic module build tests with Nix (#107)
  - Will add unit testing and formal verification
- Gazebo/VRX simulation natively in Nix

3.2 The nPOV

3.2.1 Isn't there work to do outside of Nix nonsense?

Yes, here is a sampling of some interesting research directions that align with the nPOV:

3.2 The nPOV

3.2.2 Isn't there work to do outside of Nix nonsense?

Yes, here is a sampling of some interesting research directions that align with the nPOV:

- Work on adversarial robustness

3.2 The nPOV

3.2.3 Isn't there work to do outside of Nix nonsense?

Yes, here is a sampling of some interesting research directions that align with the nPOV:

- Work on adversarial robustness
  - Requires an analysis and reworking of current perception algorithms
  - Docking will be a good case study

3.2 The nPOV

3.2.4 Isn't there work to do outside of Nix nonsense?

Yes, here is a sampling of some interesting research directions that align with the nPOV:

- Work on adversarial robustness
  - Requires an analysis and reworking of current perception algorithms
  - Docking will be a good case study
- Formal verification with Deal

3.2 The nPOV

3.2.5 Isn't there work to do outside of Nix nonsense?

Yes, here is a sampling of some interesting research directions that align with the nPOV:

- Work on adversarial robustness
  - Requires an analysis and reworking of current perception algorithms
  - Docking will be a good case study
- Formal verification with Deal
  - Use Haskell or another functional language for rapid theorizing and testing, independent of main codebase

3.2 The nPOV

3.2.6 Isn't there work to do outside of Nix nonsense?

Yes, here is a sampling of some interesting research directions that align with the nPOV:

- Work on adversarial robustness
  - Requires an analysis and reworking of current perception algorithms
  - Docking will be a good case study
- Formal verification with Deal
  - Use Haskell or another functional language for rapid theorizing and testing, independent of main codebase
- Fully vision-based autonomous systems

3.2 The nPOV

3.2.7 Isn't there work to do outside of Nix nonsense?

Yes, here is a sampling of some interesting research directions that align with the nPOV:

- Work on adversarial robustness
  - Requires an analysis and reworking of current perception algorithms
  - Docking will be a good case study
- Formal verification with Deal
  - Use Haskell or another functional language for rapid theorizing and testing, independent of main codebase
- Fully vision-based autonomous systems
  - Station keeping would make a good R&D study of the feasibility of this

R&D subteam?

3.3 Further Reading

More to come on this in the future...

3.3 Further Reading

More to come on this in the future...

This report is entirely open-access and open-source. You are welcome to contribute.
https://github.com/arcturusNavigation/tdr

4. References

Crane, Dana. 2022. “Trust, Security and the Reproducibility Crisis in Software”. February 2022. https://www.activestate.com/blog/trust-security-and-the-reproducibility-crisis-in-software/.

Morehouse, Ed. 2015. *Burritos for the Hungry Mathematician*. Pittsburgh, PA. https://edwardmorehouse.github.io/silliness/burrito_monads.pdf.

Pohl, Walt. 2005. “Being Square”. June 2005. https://examinedlife.typepad.com/johnbelle/2005/06/being_square.html?cid=6115545#comment-6a00d83451601c69e200e55022e42a8833.

4. References

Purvis, Mike, and Ivor Wanders. 2022. “Better ROS Builds with Nix”. Kyoto, Japan. 2022. http://download.ros.org/downloads/roscon/2022/Better%20ROS%20Builds%20with%20Nix.pdf.

“Flakes Aren’t Real and Cannot Hurt You.” 2024. January 2024. https://jade.fyi/blog/flakes-arent-real/.

“The Postmodern Build System.” 2023. December 2023. https://jade.fyi/blog/the-postmodern-build-system/.