Your Infrastructure is a DAG

Manage it with Nix

Ethan Carter Edwards
ethan@ethancedwards.com

2026-08-08

Slides, source: https://ethancedwards.com/blog/2026-def-con-nix-vegas

2026 DEF CON 34 Nix Vegas

Disclaimers

- Thank you to exa.ai for sponsoring my presentation and making it possible for me to attend DEF CON 34 Nix Vegas!
- The views and opinions expressed here are my own and do not necessarily reflect the official policy, position, or opinions of my employer, university, or any other affiliated entity.
- Slides: https://ethancedwards.com/blog/2026-def-con-nix-vegas

About Me

- FLOSS lover
- Nix user, Nixpkgs contributor for almost 6 years
- Member of NGI, CUDA, and darwin-maintainers teams
- Summer of Nix 2025 fellow
- Infra, Nix @ `exa.ai` (we're hiring!! come find me please)
- CS+Philosophy @ Harvard '29
- `ethancedwards8` on GitHub, Matrix

Outline

What is Nix?

What is Nix?

That's a great question...

I used to explain Nix to people like this

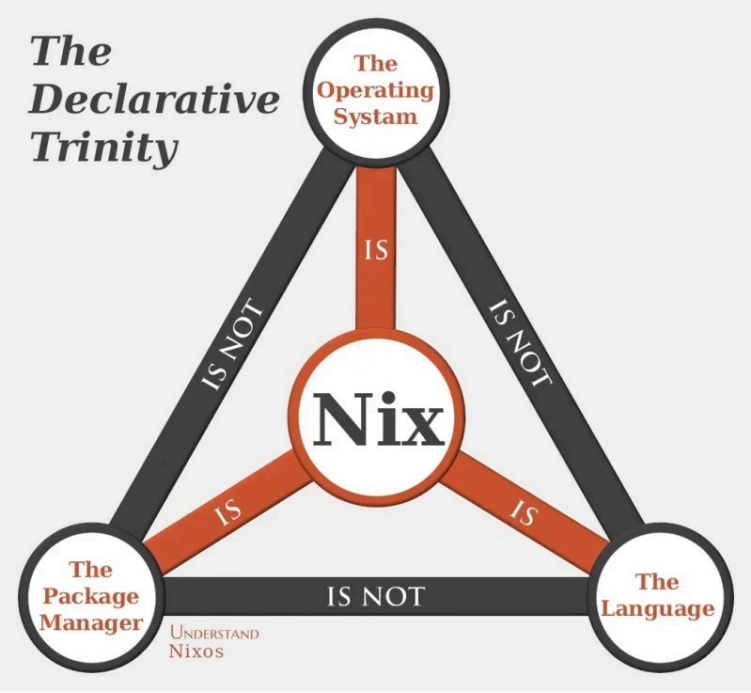


Figure 1: dgt.is

This is a good start, but I've recently started explaining Nix as a really sophisticated graph manager that is often used for building and managing software.

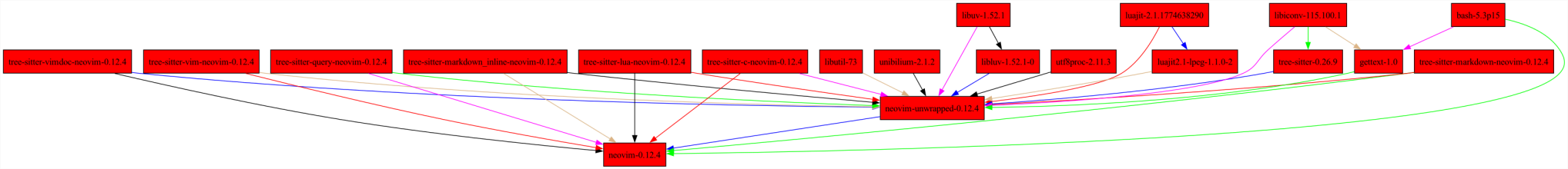


Figure 2: NB: these are the runtime deps and not buildtime (which go back to the root hex0-seed)

It's really good at this! But I think we can use it for more.

To explain, we need a little bit of context...

Nix @ Exa

Why did Exa start using Nix?

Started using Nix a bit more than a year ago because we needed a build system with features like:

- Deterministic, reproducible builds
- Fine-tuned caching (looking at you Pytorch)
- Fixing the “works on my machine problem”
- Integrated well with OCI images + Kubernetes deployments
- Having really flexible CI primitives
- Having unified dev<>production dependencies and environments
- Dependency pinning
- Declarative configuration

Like most companies, Exa adopted Nix incrementally.

We started with devshells + direnv, then builds and CI, then container images, and eventually the entire company converged to using it in every part of the software lifecycle.

More:

- We're big believers in the monorepo
- We're heavy users of agentic tooling. This influences some of our design decisions
  - We love direnv + envrc
- We revere IaC and declarative configuration and shun clickops

The first experiment

Eventually, we converged to something that looks like this:

```
1  ├ flake.{nix,lock}
2  ├ .envrc
3  ├── vectordb
4  │   ├ flake.{nix,lock}
5  │   ├ Cargo.{toml,lock}
6  │   └── src/**.rs
7  ├── dashboard
8  │   ├ flake.{nix,lock}
9  │   ├ package{,-lock}.json
10 │   └── app/**.ts
11 ├── base
12 │   └ flake.{nix,lock}
```

- We have a `flake.nix` in every service directory
  - Exposes `packages.{default,docker,typecheck,lint,unitTest,integrationTest,deploy}`
  - Exposes `devShells`
  - Defines `inputs.{nixpkgs,base}` and other libraries/projects
- Why?
  - We needed each project to be its own self-contained build
    - Run `nix build` from any project and it works!
  - We needed each project to have its own version of `nixpkgs`
  - The `nix3/nix-command` CLI is so much better than `nix2`

dashboard/flake.nix

```nix
{
  inputs = {
    base.url = "path:../base";
    nixpkgs.url = "cache.xyz/github/....";
  };
  outputs = { base, nixpkgs, ... }: let
    project = system: base.lib.makeXProject {
      pkgs = import nixpkgs { inherit system; };
      src = ./.;
      dockerSpec = { ... };
      extraBuildAttrs = { postInstall = "..."; };
    };
  in {
    inherit (forAllSystems: (systems: project))
     packages devShell checks;
  };
}
```

base/flake.nix

```nix
{
  inputs.nixpkgs.url = "cache.xyz/github/...";

  outputs = { nixpkgs, ...}: {
    lib.makeRustProject = { ... }@args: { ... };
    lib.makeGoProject = { ... }@args: { ... };
    lib.makeNodeProject = { ... }@args: { ... };
    lib.makePythonProject = { ... }@args: { ... };
    lib.makeBunProject = { ... }@args: { ... };
    lib.makeDockerImage = { ... }@args: { ... };
    lib.deployConfig = { ... }@args: { ... };
    lib.makeIntegrationTest = { ... }@args: { ... };
    lib.makeUnitTest = { ... }@args: { ... };
    lib.makeE2eTest = { ... }@args: { ... };
    ...
  };
}
```

Ideal per-project flake.nix files

```
{
  inputs = {
    base.url = "path:../base";
    nixpkgs.url = "cache.xyz/github/...";
    rust-crate-abc.url = "path:../rust-crate-abc";
    rust-crate-xyz.url = "path:../rust-crate-xyz";
  };
  outputs = { base, nixpkgs, ... }@inputs: let
    project = system: base.lib.makeRustProject {
      pkgs = import nixpkgs { inherit system; };
      dependencies = with inputs; [ rust-crate-xyz
rust-crate-abc ];
    };
  in { ... };
}
```

Actual per-project flake.nix files

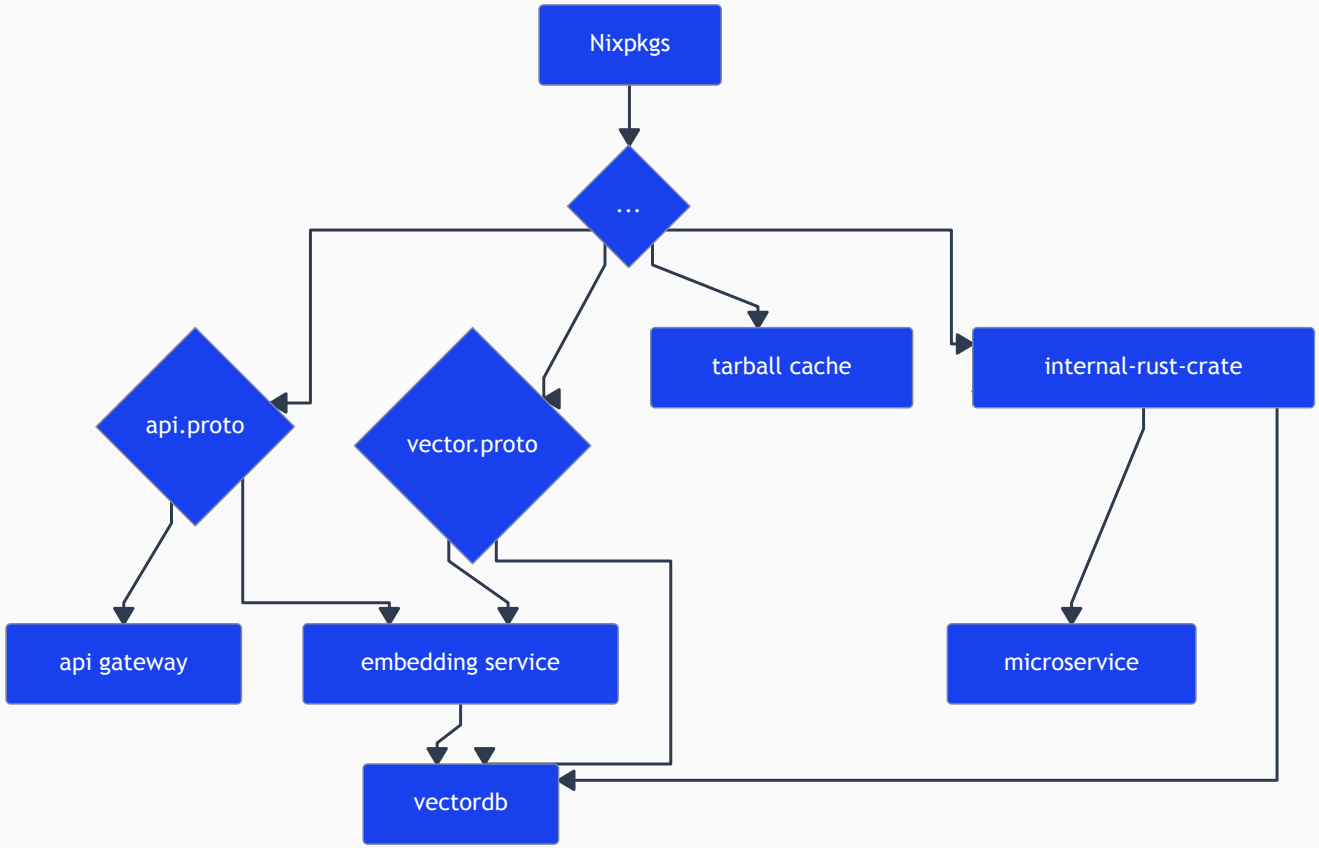
```
inputs = {
  nixpkgs.url = "cache.xyz/github/...";
};

outputs = {
  legacyPackages = forAllSystems (system: {
    default = rustPlatform.buildRustPackage {
      ...
      doCheck = false;
      ...
      src = ./.;
    }
  });
};
```

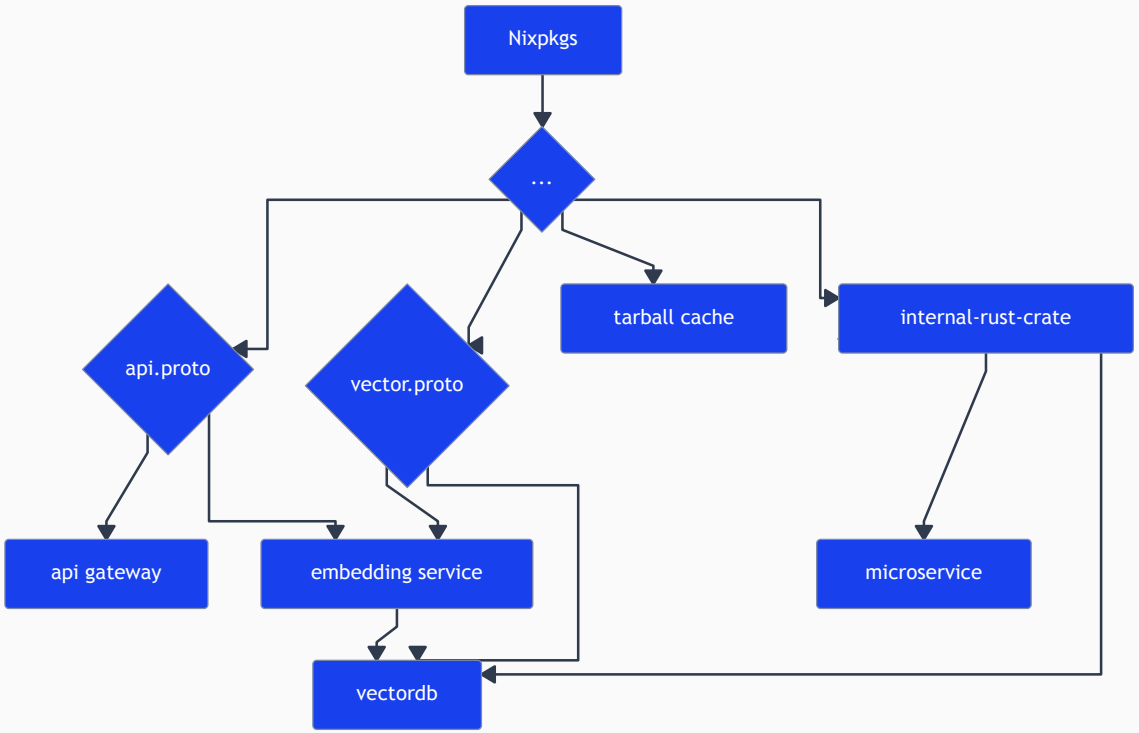
Scaling Problems

It worked, but had problems:

- The merge conflicts from locks was a nightmare
- Reverse lookup is HARD
- ~~Evaluating a flake is slow~~
  - ALL eval is slow
- Lots of sloppy, incosistent boilerplate
- Enforcing clean idioms was really hard
- Bespoke, unfamiliar
- This slide's DAG is a lie



Reverse Dependency Lookup



What is a reverse dependency?

- All the things that depend on something
- `vectordb` depends on `vector.proto`
- If we change `vector.proto`, we should always rebuild `vectordb` and `embedding service`

Key question:

- How do we know what derivations need to be rebuilt given a set of changes?

It's really hard.

Actually, it's not that hard. It's just really hard to do fast.

`nix-dep-graph`: really weird internal tool that we kinda just slopped together:

1. Evaluated every flake in the monorepo
2. Used a custom Nix command we added to our fork to list all of the paths a flake depended on (jank, only worked for transitive dependencies like 30% of the time...)
3. Created a large mapping of flakes to files
4. When a PR is opened, run `nix-dep-graph` and only rebuild the flakes affected

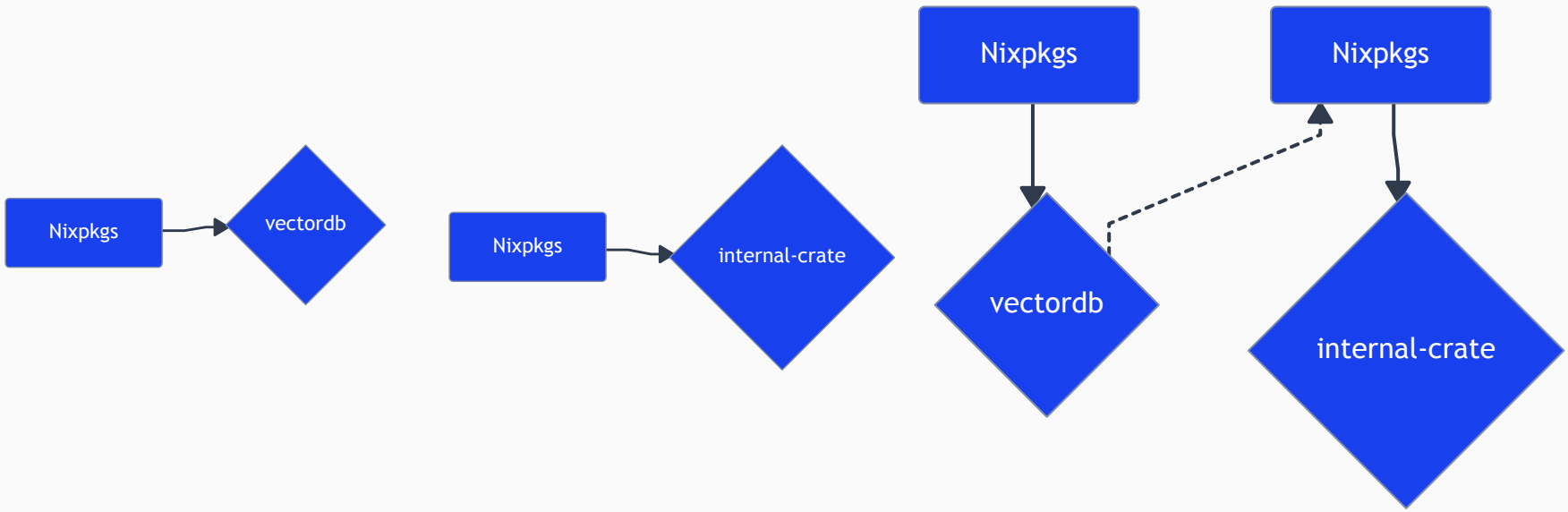
`vectordb` paths:

- `src/*.rs`
- `../../../{vector,api}.proto`
- `../../internal-crate-xyz/`

Now multiply this by like 500 for every single project/service we have.

Over time, there were optimizations and caching, but it was fundamentally the wrong solution.

Instead of creating one coherent DAG, we were essentially creating 500-odd DAGs and grafting them all together:



Exapkgs

exapkgs

The solution to all of our Nix problems

I spent about a month thinking: “If I could redo Nix @ Exa, what would I change?... “

Once I had my plan and design, the question become: “How can I convince this company to let me, an intern, rewrite their build systems from scratch?...”

Spoiler: they said yes. I’m persistent ;).

Flakes begone...

The very first design decision I made was to remove all of the flakes from the project directories and replace them with normal `project.nix` files.

- Flakes suck. `flake.nix` files are not regular Nix files and cannot be used like regular nix files.
- Create a large package set (fixed-point combinator) with:
- Hoist all of the projects to the top-level scope to run `nix build .#project`, like Nixpkgs
- Host their tests to the root flake `checks` attr for `nix flake check` functionality

```nix
let
  inherit (pkgs) lib;
  exapkgs = lib.makeScope pkgs.newScope (final: with final; {
    ... = callPackage ./.../project.nix { };
  });
in exapkgs
```

+ can use `by-name-overlay` recursive directory walk from Nixpkgs to avoid `callPackage` entries!

This reminds me of Nixpkgs...

- `project.nix` per service
- Construct a package set of all projects
- Recursively discover all `project.nix` files
- Create our own abstractions to expose:
  - the package
  - development shell
  - docker image
  - deployment script!!
  - unit, e2e, integration tests
  - lints, typechecks, formatting
  - more via passthru!

`project.nix`

```
{
  exalib,
  makeRustProject,
}: // simple example
makeRustProject {
  workspace = ./src;
  integrationTest = { ... };
  deployConfig = { ... };
  passthru.extraTest = ...;
  meta.owners = exalib.teams.infra;
}
```

But wait, what about per-project independence?

If you recall, one of the major reasons why we chose flakes to begin with was having per-project Nixpkgs versions and being able to `nix build` from any directory. That way, we can avoid having to always build from the root.

To maintain the interface, I decided to add a very simple, input-less `flake` stub alongside each `project.nix` that hoists the root flake attributes to the project.

```nix
{
  # Makes plain `nix build`/`nix run`/`nix flake check` work
  outputs = { ... }: import ../../../exalib/callRootFlake.nix ./.;
}
```

What about per-project Nixpkgs?

```
makeRustProject {
  nixpkgs = builtins.fetchurl { ... };
}
```

The `nixpkgs` argument is then imported and passed to the rest of the function.

We're able to avoid paying eval-time import costs per-project by memoizing the results of the imports and passing them down the callstack.

Directory layout:

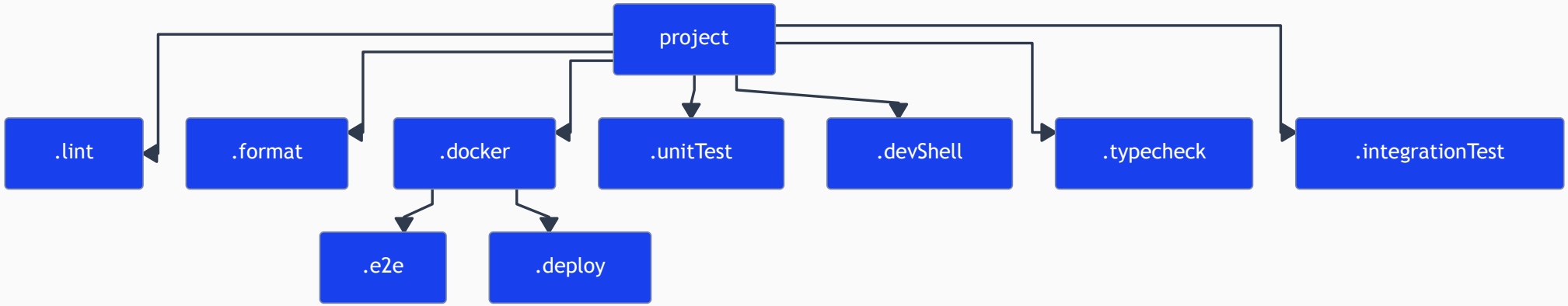
```
vectordb/
- flake.lock
  flake.nix
+ project.nix
```

Each of the `make{Rust,Go,Node,Python}Project` is just a wrapper around the respective `gobuild.nix`, `uv2nix`, `crate2nix`, etc.

They also define the logic to create sub-attribute derivations. This helps reduce boilerplate across the repo. These are all highly composable for use in things like Maturin or Tauri Projects as well.

The Project DAG

Each project will produce outputs that roughly look like this in DAG form:



Each node is a `derivation` that is cached. If we recursively build the project and its sub-derivations in CI, we can cache them!

If we solve reverse dependency lookup, we can also only run them when we need to!

Reverse Dependency Lookup Solved

In order to solve reverse dependency lookup and kill `nix-dep-graph`, we need to draw inspiration from prior art: Nixpkgs.

During Nixpkgs CI, reverse dependency lookup is solved by:

1. Evaluating the entire PR tree and creating a map of attributes to `drvPaths`
2. Evaluating it again at the PR merge commit to create another map
3. Diff'ing the two maps to get a list of attributes that have changed

Diffing

```
[
  "independent-project": /nix/store/i-will-not-change.drv
  "cool-crate": /nix/store/yay-nix-rocks.drv
  "consumer-crate": /nix/store/we-love-nix.drv
]
```

```
[
  "independent-project": /nix/store/i-will-not-change.drv
- "cool-crate": /nix/store/yay-nix-rocks.drv
+ "cool-crate": /nix/store/oh-no-i-changed.drv
- "consumer-crate": /nix/store/we-love-nix.drv
+ "consumer-crate": /nix/store/we-love-nix-vegas.drv
]
```

How do we actually evalaute the tree?

Again, great artists steal, and we draw inspiration from Nixpkgs:

1. Recursively walk the package tree
2. Create nested attr set of tree
3. Flatten tree
4. Export to JSON for diff

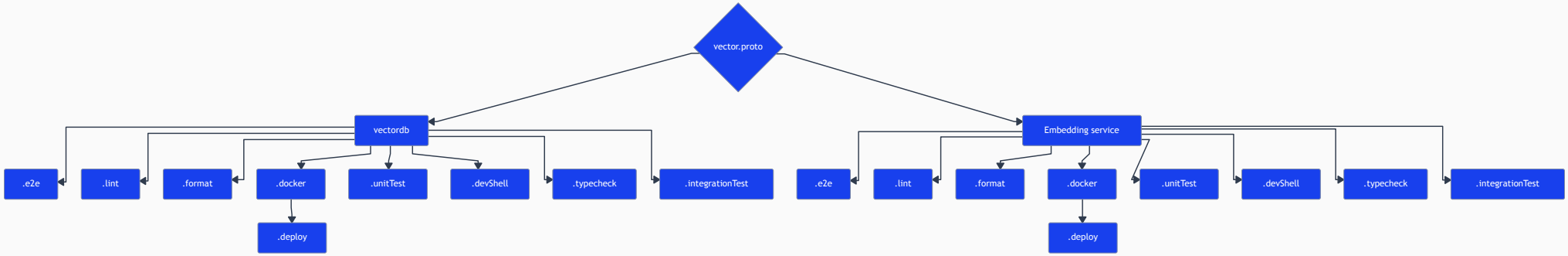
Once you have the JSON export for a merge commit and a branch HEAD commit, diffing them is easy with `jq`

```
{
  "added": [],
  "changed": [
    "python313Packages.tesserocr.x86_64-linux",
    "python314Packages.tesserocr.x86_64-linux",
    "scantpaper.x86_64-linux",
    "tests.config-nix-unit.x86_64-linux",
    "weblate.x86_64-linux"
  ],
  "rebuilds": [
    "python313Packages.tesserocr.x86_64-linux",
    "python314Packages.tesserocr.x86_64-linux",
    "release-checks",
    "tests.config-nix-unit.x86_64-linux",
    "weblate.x86_64-linux"
  ],
  "removed": []
}
```

Back to Infra as DAG

Derivations and DAGs are powerful

We can draw clear lines between dependencies

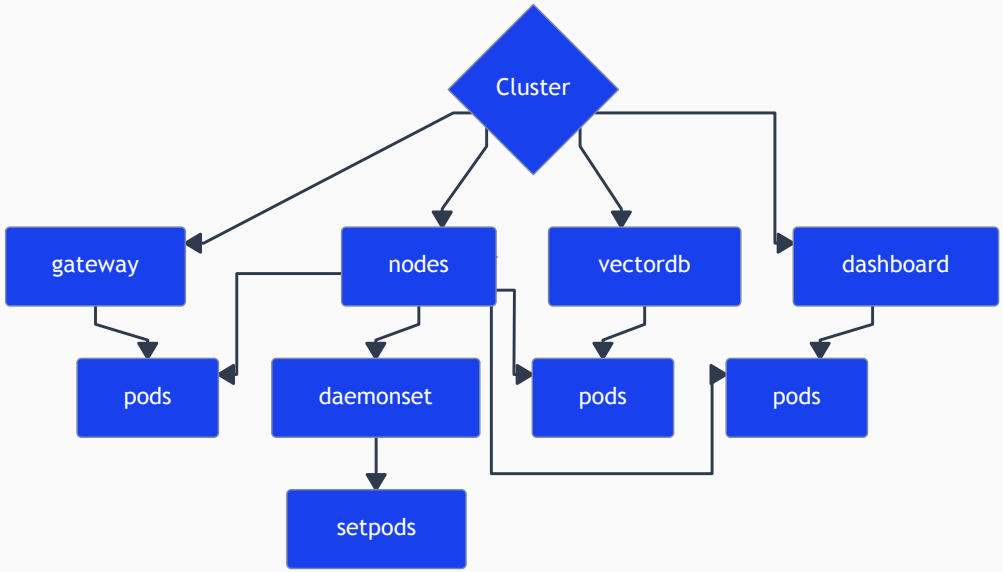


Modeling actual infrastructure

Everything is a DAG

The `.proto` examples is kind of contrived and uninspired. What if we model IaC with Nix Derivations? Using technologies like Pulumi, Terraform/Tofu, Ansible, etc.

Using Pulumi, we can create and model things like Kubernetes Clusters, Deployments, Resources, etc. This isn't a Pulumi/k8s talk, but this is a good way to think about things:



How do we put infra resources in a derivation?

Admittedly, modeling this is hard. It doesn't really fit within the bounds of a normal Nix derivation. Sandboxing makes accessing APIs to deploy and check resources pretty much impossible. Impurity is an option, but it's not a good one. Maybe there's a way to do this with FODs? Perhaps this would also be cleaner with TofuNix?

What we have to do is essentially model them as meta-derivations in a script:

```
1  deploy = runCommand "deploy-service" { nativeBuildInputs = [ exa-deploy ]; } ''
2    exa-deploy ${docker} deploy.ts
3  '';
```

We include the previous derivations (docker here) to form the DAG. When the derivation changes, we build the build script and then run it.

One of the inherent problems of IaC utilities is the constant drift between the state of the world/reality and the state of our IaC. For some static resources like DNS records, this isn't a big deal and is easy to manage/audit.

But for things like production Kubernetes deployments, managing this is really important. Ideally, the state of reality should be as close to the state of our IaC+Nix as possible.

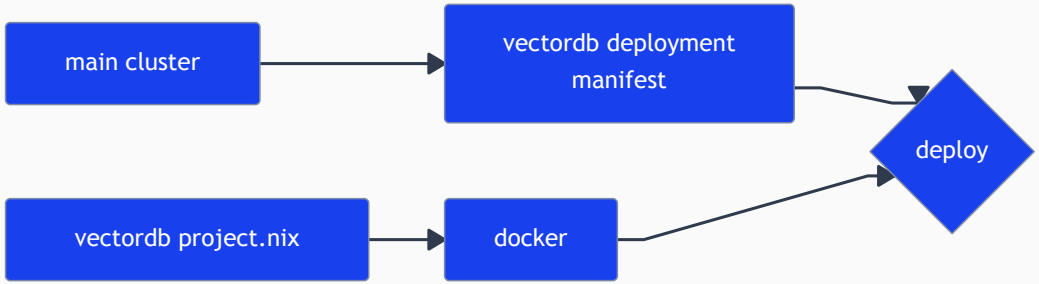
Modeling some kinds of resources as derivations is non-sensical: nodes, pods, etc. These often autoscale and are ephemeral, and we don't really care what happens to them. These are often managed by Kubernetes and friends (Karpenter).

However, clusters and persistent deployments are definitely useful.

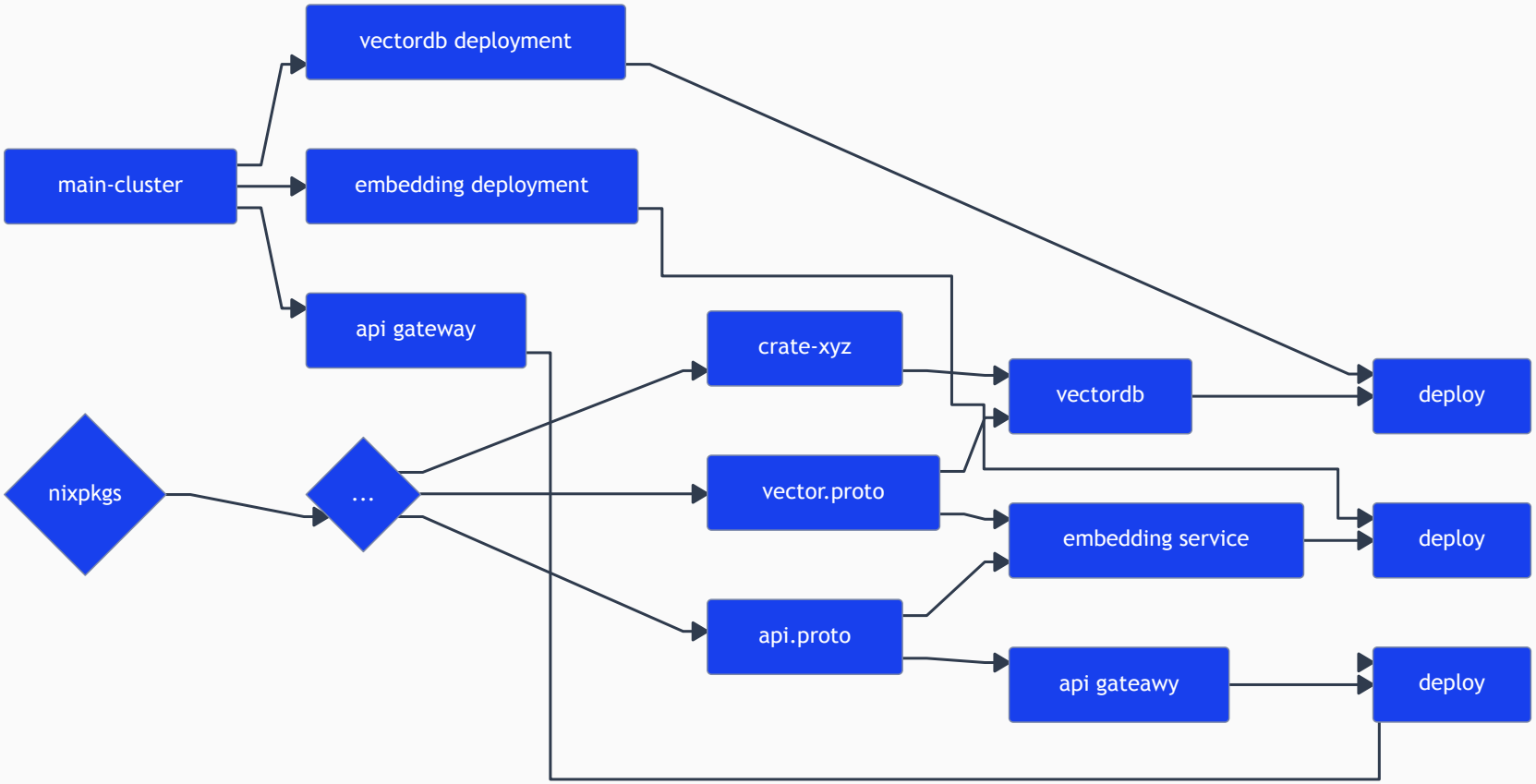
Creating the Infra DAG

```
infra/main-cluster:
  cluster.ts # define pulumi
  project.nix # define
derivations

infra/staging-cluster:
  cluster.ts
  project.nix
```



Fully Constructed DAG



When a PR is opened that will cause rebuilds, we split the derivations into two categories:

1. For regular/build derivations, we can simply `nix build` and push the result to cache
2. For infra meta-derivations that are just a deploy script, we will deploy them with `nix run` to our preview environment for the PR

Pros:

- Because we have one large DAG, we run every single test in the monorepo via `nix flake check` in under 2 minutes on every merge, ensuring our repo is never in a broken state.
- Visualizing dependencies between services via a DAG is really useful
  - Helps agents, humans understand production deployments + infra
  - Gives agents a way to check themselves
- We're actually rebuilding everything in CI now and ensuring that we ship breakages

Cons:

- We're actually rebuilding everything in CI now
- Eval is SO SLOWWWW

Exa is hiring across Infra (we use Nix!!), Security, Full-Stack, Product, Research, etc. please come find me if you're interested (we have swag 🐐)!

We push Nix to its limits and are operating at beyond web scale.

Want the slides? Visit my website: https://ethancedwards.com/blog/2026-def-con-nix-vegas

Q/A about the talk?