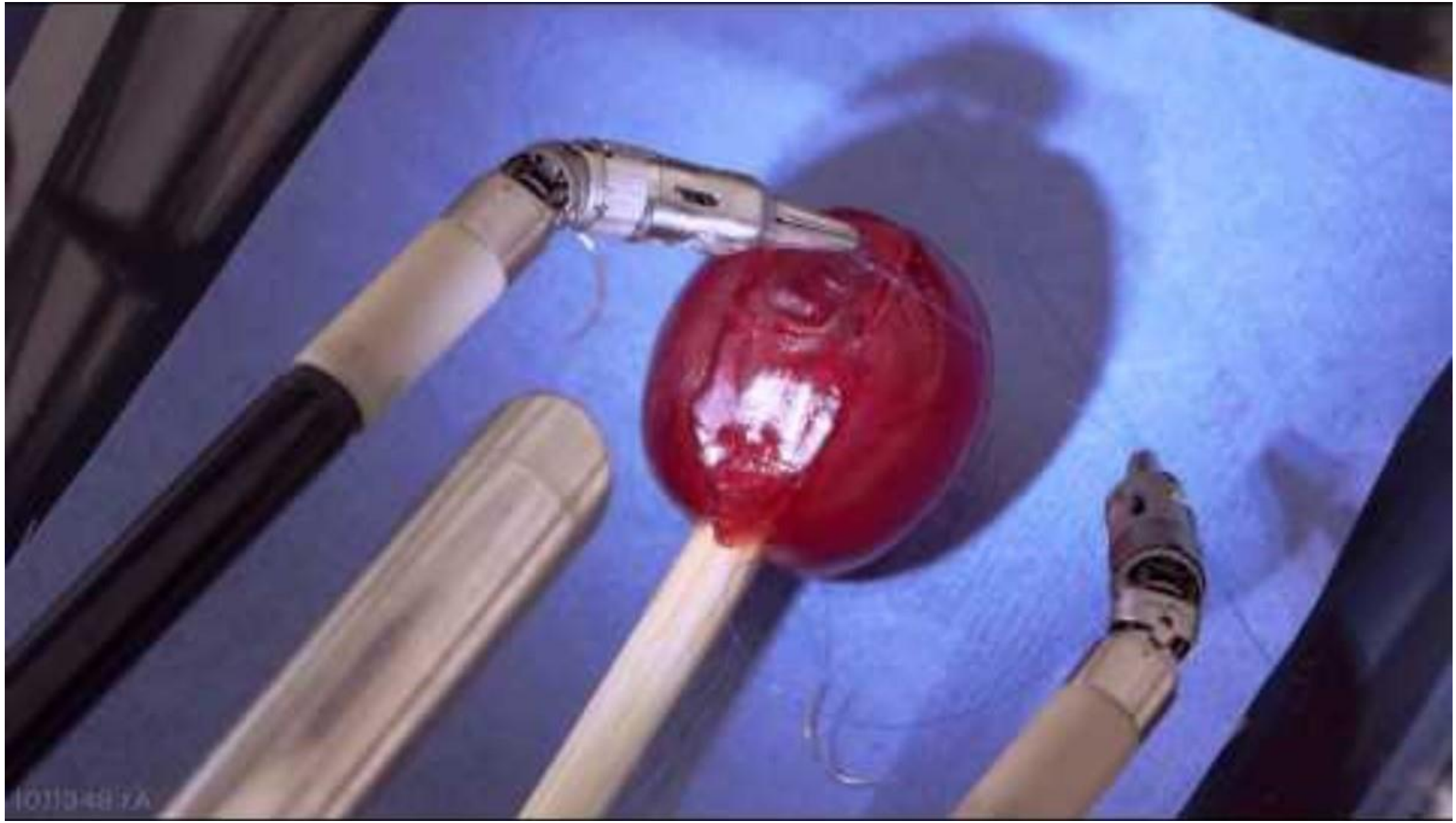
Building Robots with Nix and Bazel

Badi' Abdul-Wahid

Build engineer at Intuitive Surgical

NixCon North America 2024-03


DAVINCI



https://youtu.be/0XdC1HUp-rU

Complexities

- Regulated Entity
- Lots of targets (embedded and conventional)
- Long term support but internet bits rot
- Giant monorepo (~200 GB checkout ~40 GB worktrees)
- Lots of churn and testing
- Large brittle software blobs
- Specific toolchain requirements

Why Nix and Bazel?

- Requirements
  - Speed – build fast to move fast (without breaking things!)
  - Correctness – b/c surgery
  - Reproducible and consistent – indexable by commit

- Specifically:
  - Pinning/vendoring first layer of dependencies.
    - Important for reporting to regulatory bodies
  - Implicit transitive dependencies
  - Ability to reach deep into the dependency graph as needed
  - Ability to rebuild the world (blessing AND curse)
  - Usual caching, hermetic builds, etc

This talk

- Assumes (tries to) little / no nix background. Ask questions!
- Focuses primarily on our nix usage (but has a tiny bit of bazel too)
- Everything is awesome*

Calibration: core nix concepts we'll need

- Functions
- Overrides
- Interpolation
- Paths
- Derivations
- Overlays
- CallPackage

Nix AND Bazel?

NIX

- Build ANYTHING
  - Autotools + friends
  - Cmake
  - Shell scripts
  - Etc etc etc
- Cross compilation support
- Multilevel caching

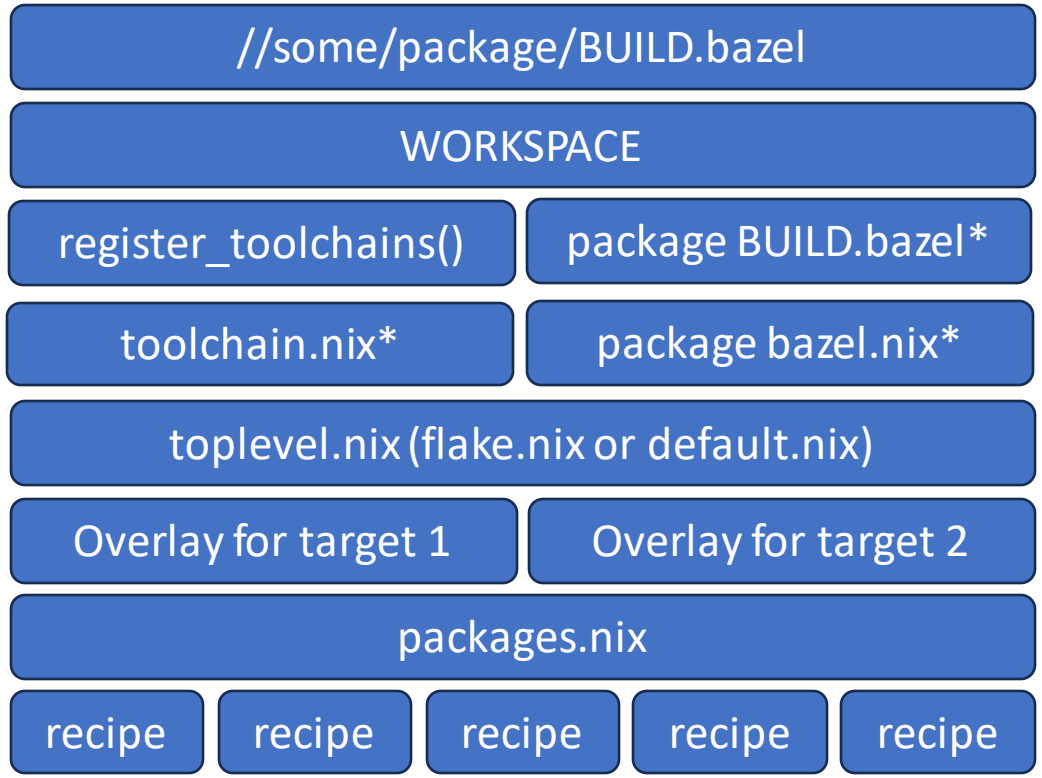
BAZEL

- Cross comp+caching
- *Finely granular*
- Lesser support for arbitrary packages

Idea

- Nix: provides tooling, toolchains, and external dependencies
- Bazel: build all our things!

Unwrapping the nix cake…



What is this?

```
{ stdenv }:

stdenv.mkDerivation {
  pname = "hello";
  version = "0.1";
  buildPhase = ''gcc ${./hello.c} -o hello'';
  installPhase = ''mkdir -p $out/bin ; mv hello $out/bin'';
}
```

Recipe :: Attrset -> Derivation

Sometimes called "package", but that doesn't quite capture what we're trying to communicate

```
{ stdenv }:

stdenv.mkDerivation {
  pname = "hello";
  version = "0.1";
  buildPhase = ''gcc ${./hello.c} -o hello'';
  installPhase = ''mkdir -p $out/bin ; mv hello $out/bin'';
}
```

What about this? Why?

```
#packages.nix
final: prev: let callPackage = final.callPackage; in
{
  foo_v1 = callPackage ./foo/v1 {};
  bar_v1 = callPackage ./bar/v1 {};
  bar_v2 = callPackage ./bar/v2 {};
  zot_v9 = … etc etc etc
}
```

Packages :: AttrSet*

*defined overlay

Analogous all-packages.nix in nixpkgs: collection of packages exposable to bazel.

Why? Support target-specific versions and config/build options

```
#packages.nix
final: prev: let callPackage = final.callPackage; in
{
  foo_v1 = callPackage ./foo/v1 {};
  bar_v1 = callPackage ./bar/v1 {};
  bar_v2 = callPackage ./bar/v2 {};
  zot_v9 = … etc etc etc
}
```

Uses package bindings in the overlay scope

Lazy evaluation has tradeoffs

What is this?

```
final: prev: {
  a = final.callPackage ./a {};
  b = prev.b.override { systemd = null; withSSL = true; };
}
```

```
# incomplete recipe for b
{ stdenv, lib, a, systemd, openssl withSSL ? false }:
stdenv.mkDerivation {
  configureFlags = lib.optional withSSL [ "--with-ssl" ];
  buildInputs = [ a systemd ] ++ lib.optional withSSL [openssl];
}
```

Overlay :: AttrSet -> AttrSet -> AttrSet

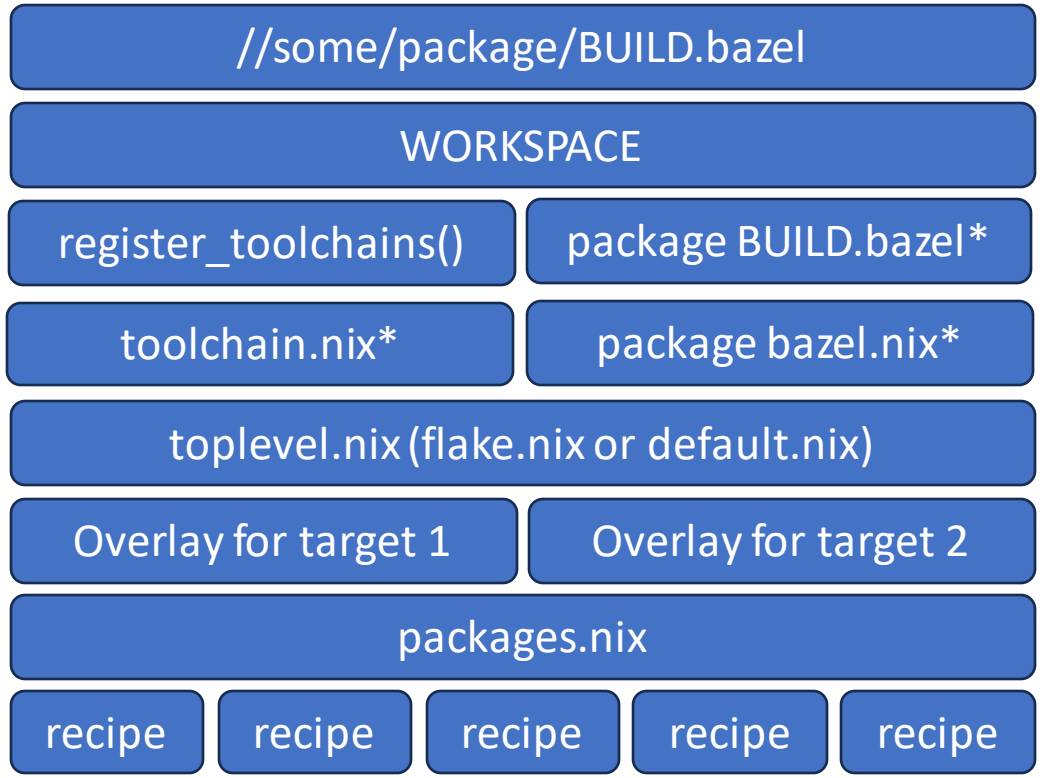
- Allows injection into nixpkgs

```
# example.nix
with import <nixpkgs> { overlays = [ (import ./overlay.nix) ]; }

# shell
$ nix build –f example.nix --argstr crossSystem asdf a hello

# overlay.nix
final: prev: {
  a = final.callPackage ./a {};
  b = prev.b.override { systemd = null; };
}
```

More is needed to make this work, but out of scope.

But: overlays can be used as an allowlist

Unwrapping the nix cake…

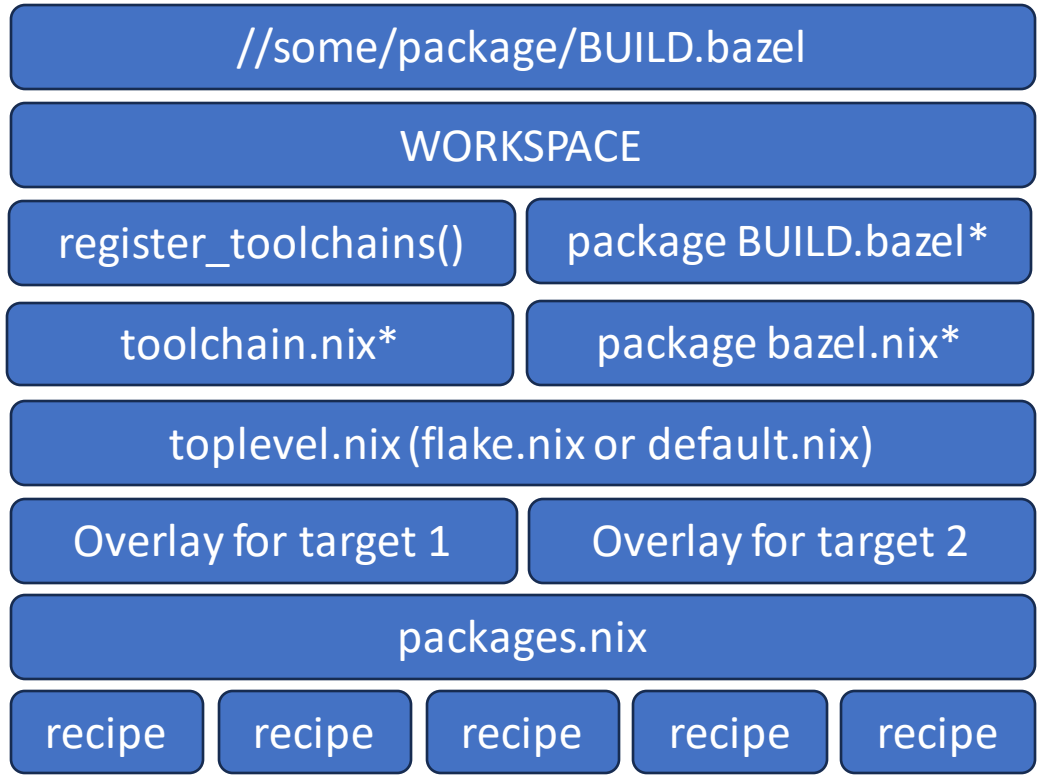



```
#target 1
final: prev: {
  foo = packages.foo_v1;
  bar = packages.bar_v2;
}
```

```
#target 2
final: prev: {
  bar = packages.bar_v1
    .override { … };
}
```

```
#packages.nix
foo_v1 = callPackage ./foo/v1 {}
bar_v1 = callPackage ./bar/v1 {}
Bar_v2 = callPackage ./bar/v2 {}

#recipe
{stdenv, …}: stdenv.mkDerivation { … }
```

Unwrapping the nix cake…



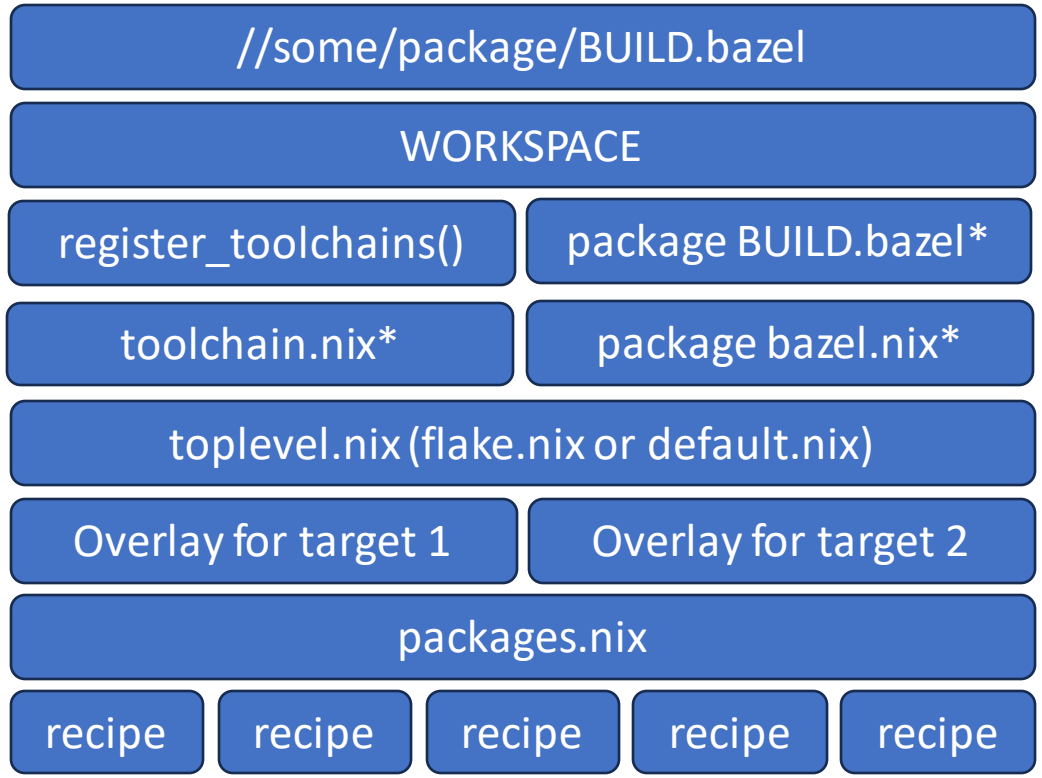

```
#package bazel.nix
{ foo, lib }: lib.wrapForBazel foo
```

```
    #target 1
    final: prev: {
      foo = packages.foo_v1;
      bar = packages.bar_v2;
    }
```

```
    #target 2
    final: prev: {
      bar = packages.bar_v1
        .override { … };
    }
```

```
                #packages.nix
                foo_v1 = callPackage ./foo/v1 {}
                bar_v1 = callPackage ./bar/v1 {}
                Bar_v2 = callPackage ./bar/v2 {}

            #recipe
            {stdenv, …}: stdenv.mkDerivation { … }
```

Unwrapping the nix cake…




```
#package BUILD.bazel
cc_library(
  name = "libfoo",
  srcs = ["lib/libfoo.so"],
)
```

```
#WORKSPACE using tweag rules_nixpkgs
nixpkgs_packge(
  name "nix.libfoo",
  attribute = "foo",
)
```

https://github.com/tweag/rules_nixpkgs

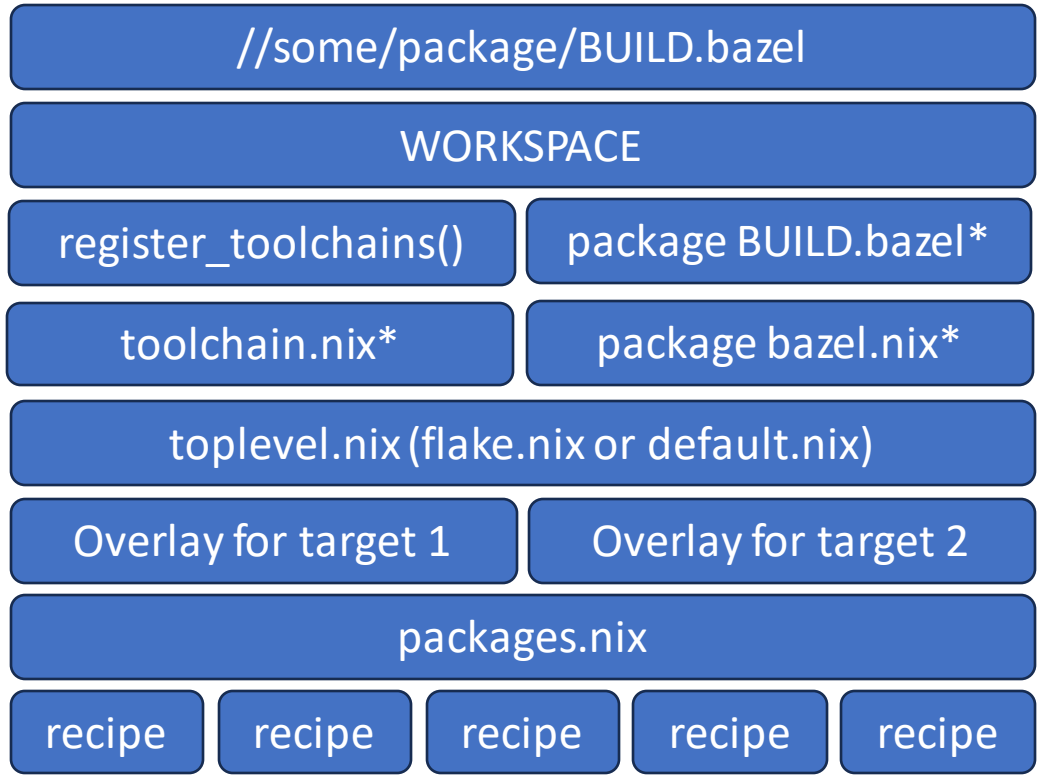
```
#package bazel.nix
{ foo, lib }: lib.wrapForBazel foo
```

```
#target 1
final: prev: {
  foo = packages.foo_v1;
  bar = packages.bar_v2;
}
```

```
#target 2
final: prev: {
  bar = packages.bar_v1
    .override { … };
}
```

```
#packages.nix
foo_v1 = callPackage ./foo/v1 {}
bar_v1 = callPackage ./bar/v1 {}
Bar_v2 = callPackage ./bar/v2 {}

#recipe
{stdenv, …}: stdenv.mkDerivation { … }
```

Unwrapping the nix cake…




```
# $repo/some/package/BUILD.bazel
cc_library(
  name = "package",
  deps = ["@nix.libfoo//:libfoo"],
)

#WORKSPACE using tweag rules_nixpkgs
nixpkgs_packge(
  name "nix.libfoo",
  attribute = "foo",
)
```

https://github.com/tweag/rules_nixpkgs

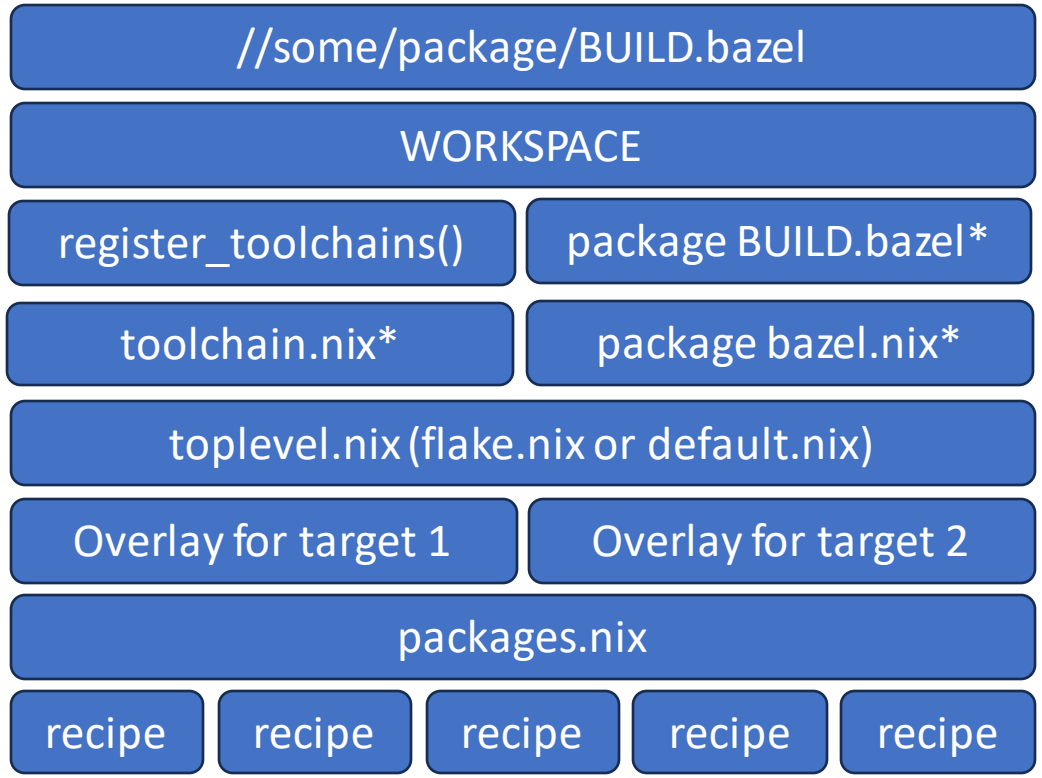
```
#package BUILD.bazel
cc_library(
  name = "libfoo",
  srcs = ["lib/libfoo.so"],
)

#package bazel.nix
{ foo, lib }: lib.wrapForBazel foo

    #target 1
    final: prev: {
      foo = packages.foo_v1;
      bar = packages.bar_v2;
    }

    #target 2
    final: prev: {
      bar = packages.bar_v1
        .override { … };
    }

            #packages.nix
            foo_v1 = callPackage ./foo/v1 {}
            bar_v1 = callPackage ./bar/v1 {}
            Bar_v2 = callPackage ./bar/v2 {}

          #recipe
          {stdenv, …}: stdenv.mkDerivation { … }
```

Developer's perspective

```
# $repo/some/package/BUILD.bazel
cc_library(
  name = "package",
  deps = ["@nix.libfoo//:libfoo"],
)
```

```
$ bazel build //some/package --platforms=a
   ---> nix-build --argStr crossSystem a –A libfoo
$ bazel build //some/package --platforms=b
$ bazel build //some/package --platforms=c
```

How did we end up here?




```
# $repo/some/package/BUILD.bazel
cc_library(
  name = "package",
  deps = ["@nix.libfoo//:libfoo"],
)

#WORKSPACE using tweag rules_nixpkgs
nixpkgs_packge(
  name "nix.libfoo",
  attribute = "foo",
)
```

https://github.com/tweag/rules_nixpkgs

```
#package BUILD.bazel
cc_library(
  name = "libfoo",
  srcs = ["lib/libfoo.so"],
)

#package bazel.nix
{ foo, lib }: lib.wrapForBazel foo

    #target 1
    final: prev: {
      foo = packages.foo_v1;
      bar = packages.bar_v2;
    }

    #target 2
    final: prev: {
      bar = packages.bar_v1
        .override { … };
    }

            #packages.nix
            foo_v1 = callPackage ./foo/v1 {}
            bar_v1 = callPackage ./bar/v1 {}
            Bar_v2 = callPackage ./bar/v2 {}

          #recipe
          {stdenv, …}: stdenv.mkDerivation { … }
```

https://easy-peasy.ai/ai-image-generator/images/majestic-unicorn-colorful-environment

What's worked well?

- breakpointHook <3 <3 <3
- Having direct access to the layers (for debugging)
- Custom installer
- Ontology adjustments: Recipe, package set, wrapping
- Multilayered caching*
- Sandboxing catching bugs*

What hasn't worked well…

Multiple outputs*

- Nix: good to split into `outputs = [ "out" "lib" "dev" "bin" "doc" ]`
- Bazel: wtf I can't find the files
- Solution: wrapForBazel === symlinkJoin all outputs

Cross compilation

- Cross compilations is simple...until you need to add custom compilers and toolchains
- replaceStdenv isn't sufficient
- What about specific glibc (or using alternatives)?
- How to correctly set up splicing?
- Nix has great cross infra, but requires in-depth knowledge

Onboarding new users to nix

- Ontology

  We're settling onto something like:

  Nix lang; functions; recipes; packages; package set; overriding; extending

Friction....UI

- nix build
- nix-build
- Verbosity and progress

nix "infects" the output binaries

- Need to ship binaries to places without /nix
- Buildtime / runtime interpreters
  - Eg /nix/<…>/ld-linux.so vs /lib/ld-linux.so

Giant blobs of proprietary software

python

error: infinite recursion encountered

Friction: summary

- Lack of docs around custom toolchains
  - How do you override the libc? Use a different gcc or non-gcc-based compiler?
  - How to fit into the nixpkgs booting and splicing?
  - replaceStdenv vs overrides
- Need ontology.
  - Nix lang -> functions -> recipes -> packages -> package set
- Nixpkgs the infra coupled with the package set
  - e.g.heavy assumption on systemd
  - e.g. given glibc
- Git lfs support
- De-"nix"-ifying outputs
- Giant brittle proprietary blobs
- Nix-to-bazel interface
- Infinite recursion

Summary

- Failure can be catastrophic: there's a real person trusting the surgical team – including robot – with their life
- Multiple target platforms, overlays provide package sets and allow-lists
- Multiversion support, hermeticity, sandboxing, etc is critical

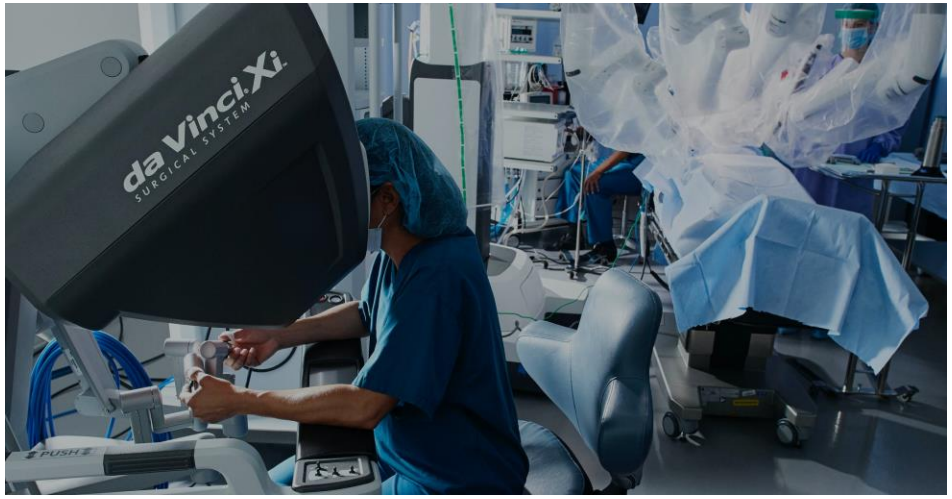
Questions?

- We're hiring! Please talk to me!

Me: badi.abdul-wahid@intusurg.com
Team lead: oleksiy.pikalo@intusurg.com

https://careers.intuitive.com/en/jobs/743999968121601/JOB6254/sr-embedded-software-engineer-build-systems/

More context



https://www.intuitive.com/en-us/patients/da-vinci-robotic-surgery

https://www.intuitive.com/en-us/products-and-services/ion/how-ion-works

https://www.intuitive.com/en-us/patients/da-vinci-robotic-surgery/about-the-systems

nixpkgs infra vs packages

- More modularity
  - nixpkgs the package set vs the tooling to build those packages

```
import ./root {
  pkgs = ./nixpkgs-checkout;
  stdenv = pkgs: pkgs.mk-stdenv ./gcc-9000;
  libc = pkgs: pkgs.mk-lib ./glibc-2.92.29;
  pkg-set = pkgs: pkgs.load-pkg-set ./all-packages
  system = …;
}
```