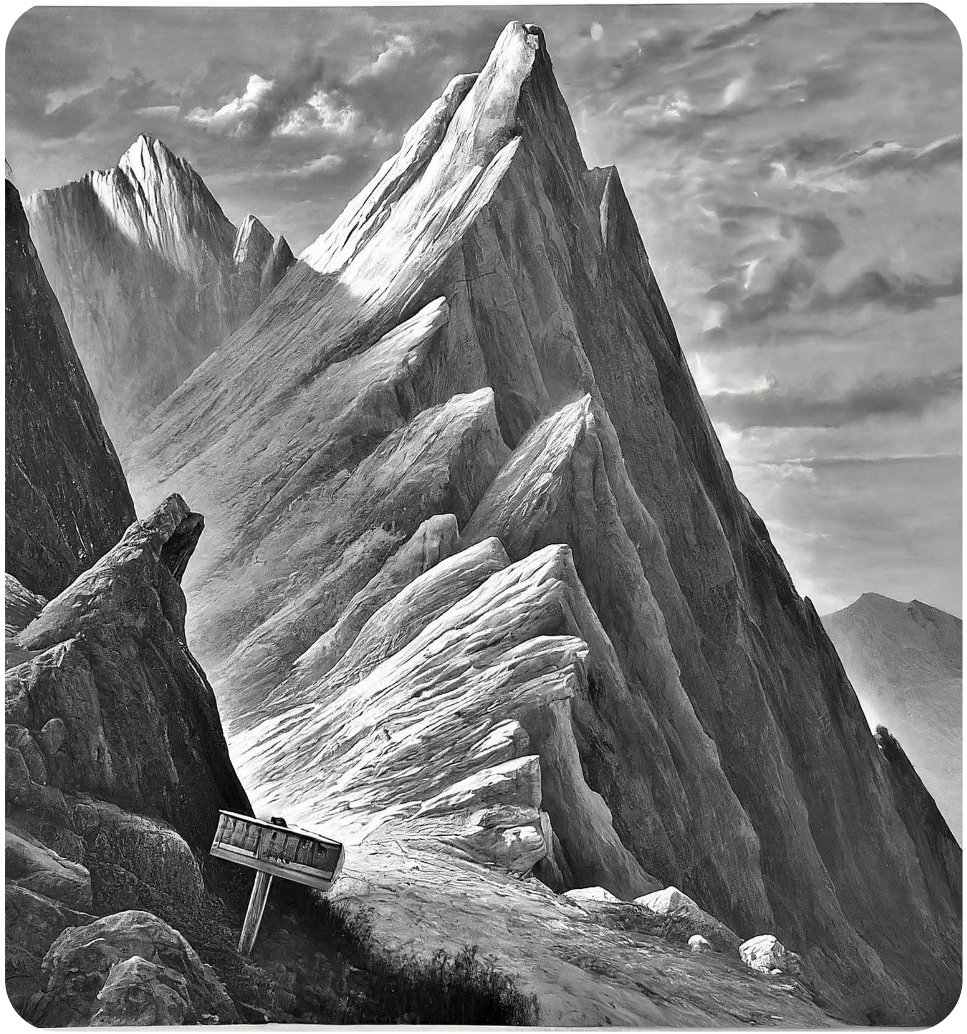
[High|Low] Lights of Adopting Nix at Looker

Disclaimers
The Fine Print

- Nothing here is unimaginable
- Our opinions are our own and not those of Google
- Our goal is to be fair but overall tell a positive story



Farid Zakaria
fmzakari@google.com
Engineering Manager

Micah Catlin
micahc@google.com
Staff Software Engineer

Looker: Business Intelligence

JRuby

Ruby

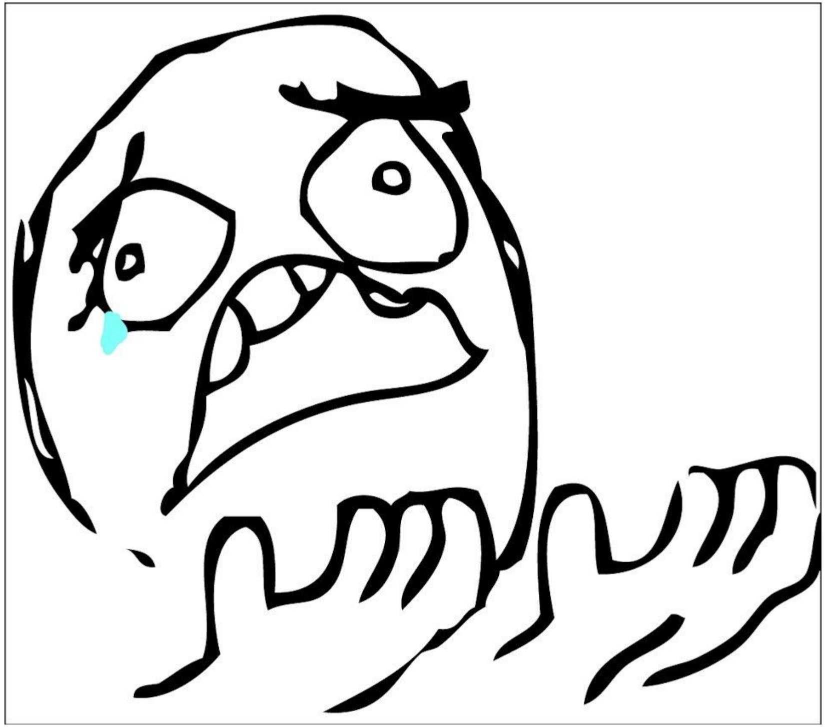
node JS

Kotlin

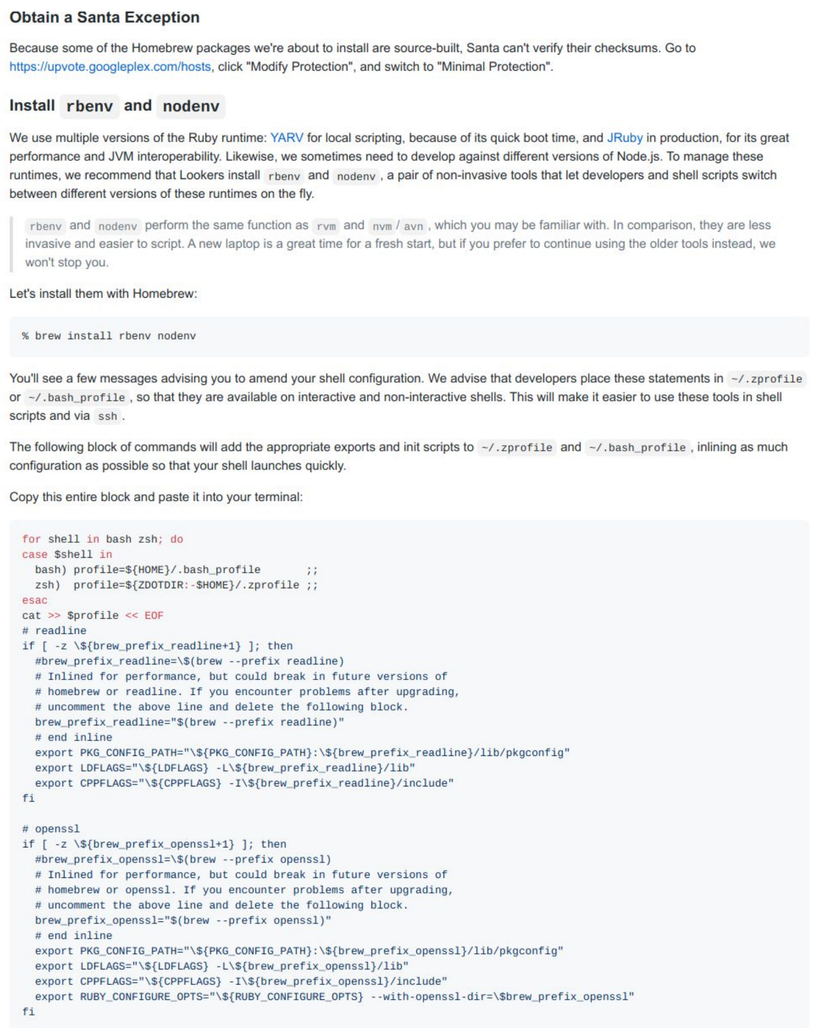
JS

MySQL

Mac OS



Can we do better?



Obtain a Santa Exception

Because some of the Homebrew packages we're about to install are source-built, Santa can't verify their checksums. Go to https://upvote.googleplex.com/hosts, click "Modify Protection", and switch to "Minimal Protection".

Install `rbenv` and `nodenv`

We use multiple versions of the Ruby runtime: YARV for local scripting, because of its quick boot time, and JRuby in production, for its great performance and JVM interoperability. Likewise, we sometimes need to develop against different versions of Node.js. To manage these runtimes, we recommend that Lookers install `rbenv` and `nodenv`, a pair of non-invasive tools that let developers and shell scripts switch between different versions of these runtimes on the fly.

> `rbenv` and `nodenv` perform the same function as `rvm` and `nvm` / `avn`, which you may be familiar with. In comparison, they are less invasive and easier to script. A new laptop is a great time for a fresh start, but if you prefer to continue using the older tools instead, we won't stop you.

Let's install them with Homebrew:

```
% brew install rbenv nodenv
```

You'll see a few messages advising you to amend your shell configuration. We advise that developers place these statements in `~/.zprofile` or `~/.bash_profile`, so that they are available on interactive and non-interactive shells. This will make it easier to use these tools in shell scripts and via `ssh`.

The following block of commands will add the appropriate exports and init scripts to `~/.zprofile` and `~/.bash_profile`, inlining as much configuration as possible so that your shell launches quickly.

Copy this entire block and paste it into your terminal:

```
for shell in bash zsh; do
case $shell in
  bash) profile=${HOME}/.bash_profile       ;;
  zsh)  profile=${ZDOTDIR:-$HOME}/.zprofile ;;
esac
cat >> $profile << EOF
# readline
if [ -z \${brew_prefix_readline+1} ]; then
  #brew_prefix_readline=\$(brew --prefix readline)
  # Inlined for performance, but could break in future versions of
  # homebrew or readline. If you encounter problems after upgrading,
  # uncomment the above line and delete the following block.
  brew_prefix_readline="$(brew --prefix readline)"
  # end inline
  export PKG_CONFIG_PATH="\${PKG_CONFIG_PATH}:\${brew_prefix_readline}/lib/pkgconfig"
  export LDFLAGS="\${LDFLAGS} -L\${brew_prefix_readline}/lib"
  export CPPFLAGS="\${CPPFLAGS} -I\${brew_prefix_readline}/include"
fi

# openssl
if [ -z \${brew_prefix_openssl+1} ]; then
  #brew_prefix_openssl=\$(brew --prefix openssl)
  # Inlined for performance, but could break in future versions of
  # homebrew or openssl. If you encounter problems after upgrading,
  # uncomment the above line and delete the following block.
  brew_prefix_openssl="$(brew --prefix openssl)"
  # end inline
  export PKG_CONFIG_PATH="\${PKG_CONFIG_PATH}:\${brew_prefix_openssl}/lib/pkgconfig"
  export LDFLAGS="\${LDFLAGS} -L\${brew_prefix_openssl}/lib"
  export CPPFLAGS="\${CPPFLAGS} -I\${brew_prefix_openssl}/include"
  export RUBY_CONFIGURE_OPTS="\${RUBY_CONFIGURE_OPTS} --with-openssl-dir=\$brew_prefix_openssl"
fi
```

Goals

- ✅ Consistent developer environment with our CI jobs
- ✅ Speed up developer setup (new devs, new machines)
- ❌ Consistent developer environment with production

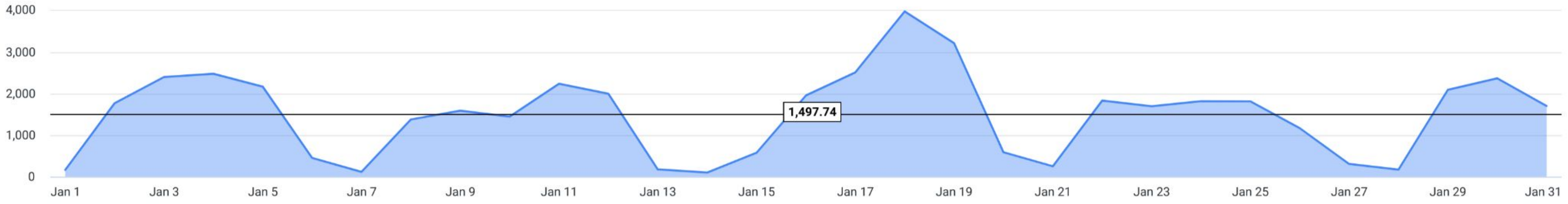
Goals (accomplished?)

- ✅ Consistent developer environment with our CI jobs
  a. For humans, Used `shell.nix` + `nix-direnv`
  b. For CI, prepended build scripts with this boilerplate

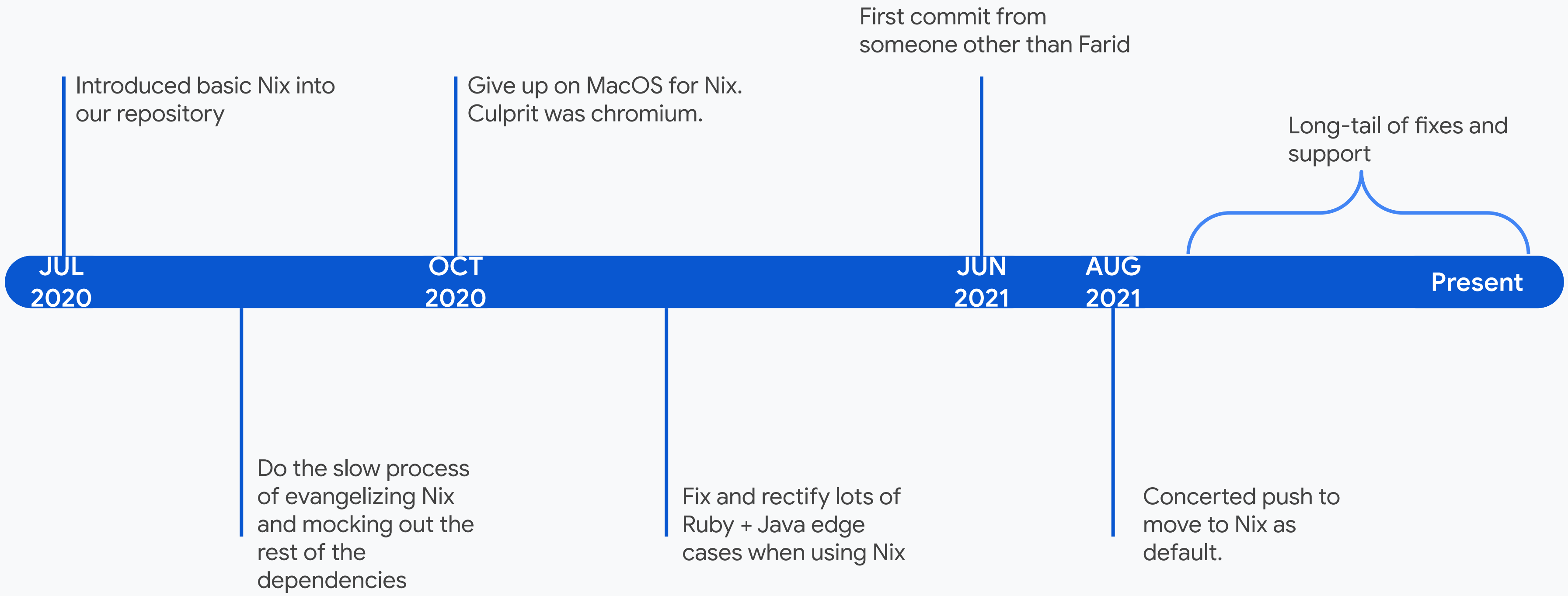
```
if [ -z "${IN_NIX_SHELL}" ]; then
  this_script=$(realpath "${BASH_SOURCE[1]}")
  cd "${nix_root}" && \
    exec nix-shell --run "${this_script} $*;"
fi
```

- ✅ Speed up developer setup (new devs, new machines)
  a. Replaced monster README.md with a few reliable scripts leveraging nix

- ❌ Consistent developer environment with production
  a. We don't (yet) invoke the product builder with nix, but the legacy build runs inside "`nix-shell`"


4,000
3,000
2,000
1,000
0
1,497.74
Jan 1
Jan 3
Jan 5
Jan 7
Jan 9
Jan 11
Jan 13
Jan 15
Jan 17
Jan 19
Jan 21
Jan 23
Jan 25
Jan 27
Jan 29
Jan 31

Timeline



JUL 2020 — Introduced basic Nix into our repository

Do the slow process of evangelizing Nix and mocking out the rest of the dependencies

OCT 2020 — Give up on MacOS for Nix. Culprit was chromium.

Fix and rectify lots of Ruby + Java edge cases when using Nix

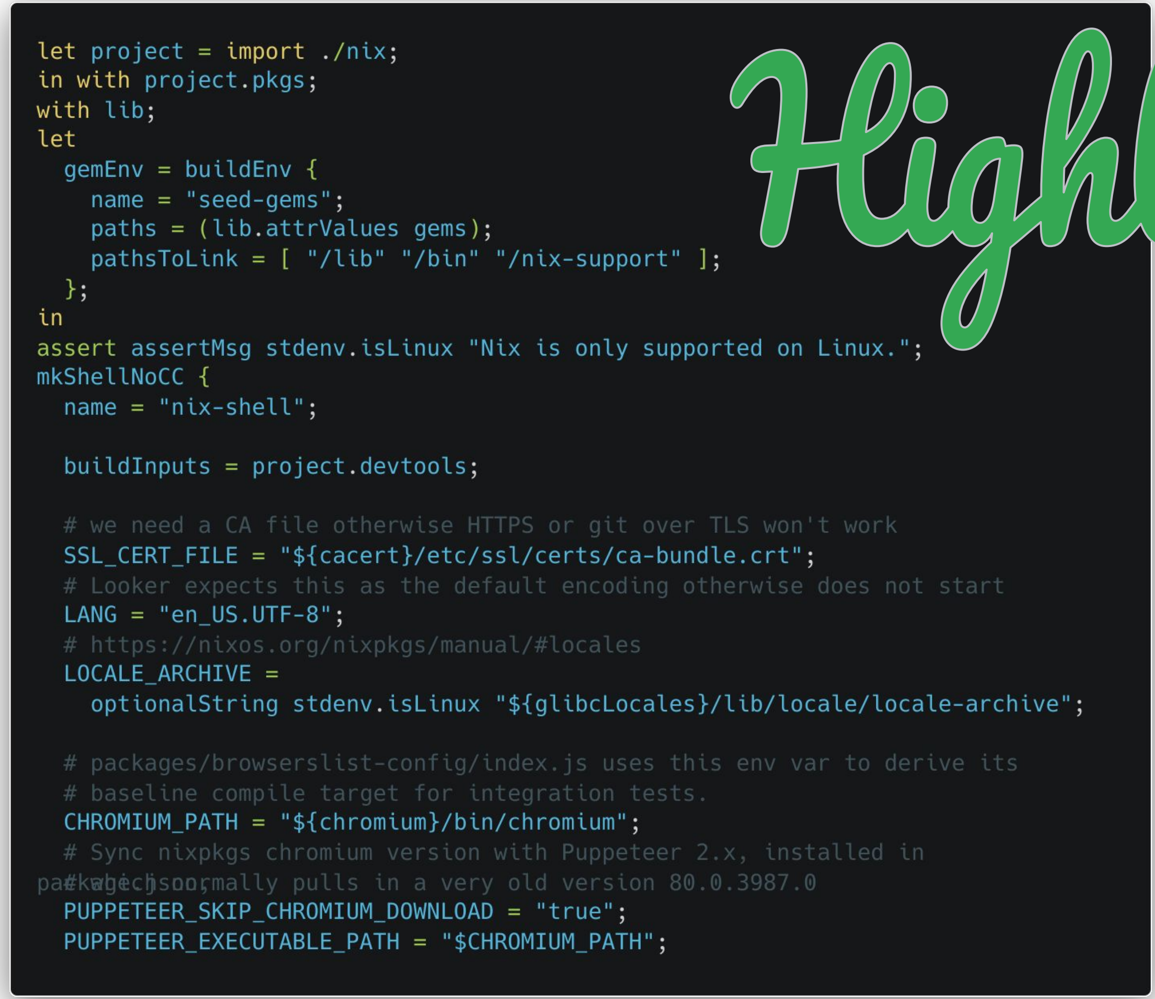
JUN 2021 — First commit from someone other than Farid

AUG 2021 — Concerted push to move to Nix as default.

Long-tail of fixes and support

Present

Highlight



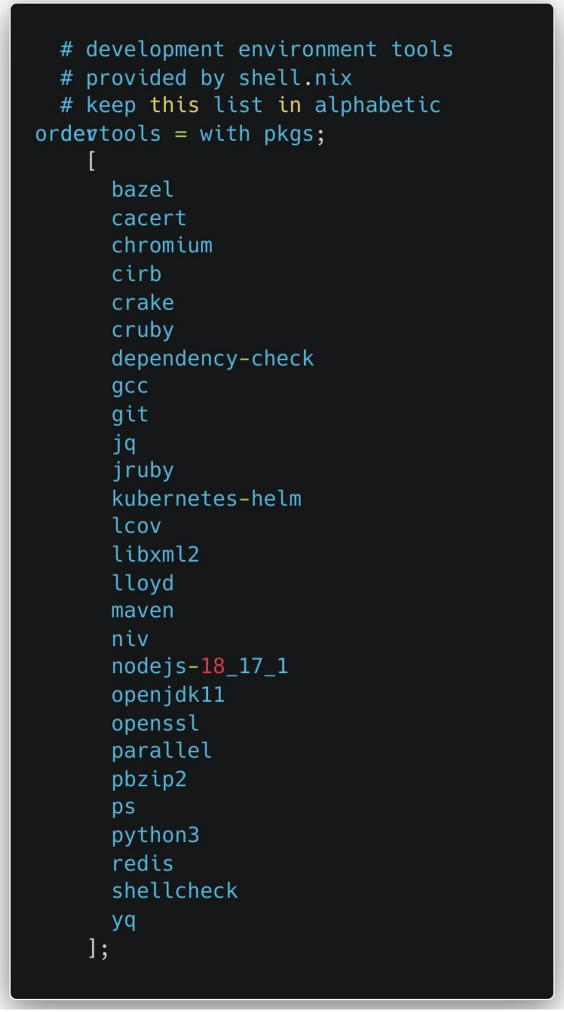
```
let project = import ./nix;
in with project.pkgs;
with lib;
let
  gemEnv = buildEnv {
    name = "seed-gems";
    paths = (lib.attrValues gems);
    pathsToLink = [ "/lib" "/bin" "/nix-support" ];
  };
in
assert assertMsg stdenv.isLinux "Nix is only supported on Linux.";
mkShellNoCC {
  name = "nix-shell";

  buildInputs = project.devtools;

  # we need a CA file otherwise HTTPS or git over TLS won't work
  SSL_CERT_FILE = "${cacert}/etc/ssl/certs/ca-bundle.crt";
  # Looker expects this as the default encoding otherwise does not start
  LANG = "en_US.UTF-8";
  # https://nixos.org/nixpkgs/manual/#locales
  LOCALE_ARCHIVE =
    optionalString stdenv.isLinux "${glibcLocales}/lib/locale/locale-archive";

  # packages/browserslist-config/index.js uses this env var to derive its
  # baseline compile target for integration tests.
  CHROMIUM_PATH = "${chromium}/bin/chromium";
  # Sync nixpkgs chromium version with Puppeteer 2.x, installed in
package.json;
  # which normally pulls in a very old version 80.0.3987.0
  PUPPETEER_SKIP_CHROMIUM_DOWNLOAD = "true";
  PUPPETEER_EXECUTABLE_PATH = "$CHROMIUM_PATH";
```

Google

Highlight

Avoid all the Nix Flakes Discussion

Lowlight

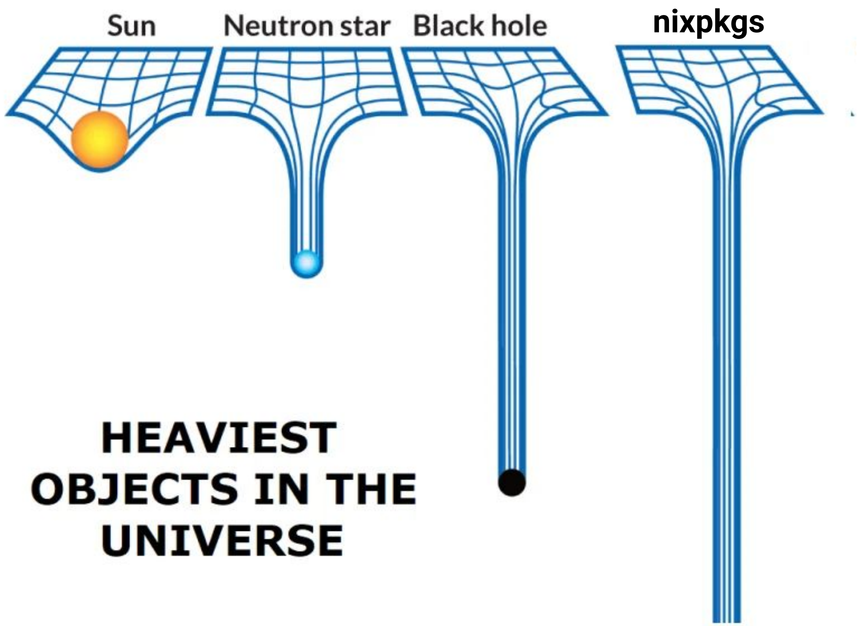


```
  # development environment tools
  # provided by shell.nix
  # keep this list in alphabetic
order
  devtools = with pkgs;
    [
      bazel
      cacert
      chromium
      cirb
      crake
      cruby
      dependency-check
      gcc
      git
      jq
      jruby
      kubernetes-helm
      lcov
      libxml2
      lloyd
      maven
      niv
      nodejs-18_17_1
      openjdk11
      openssl
      parallel
      pbzip2
      ps
      python3
      redis
      shellcheck
      yq
    ];
```

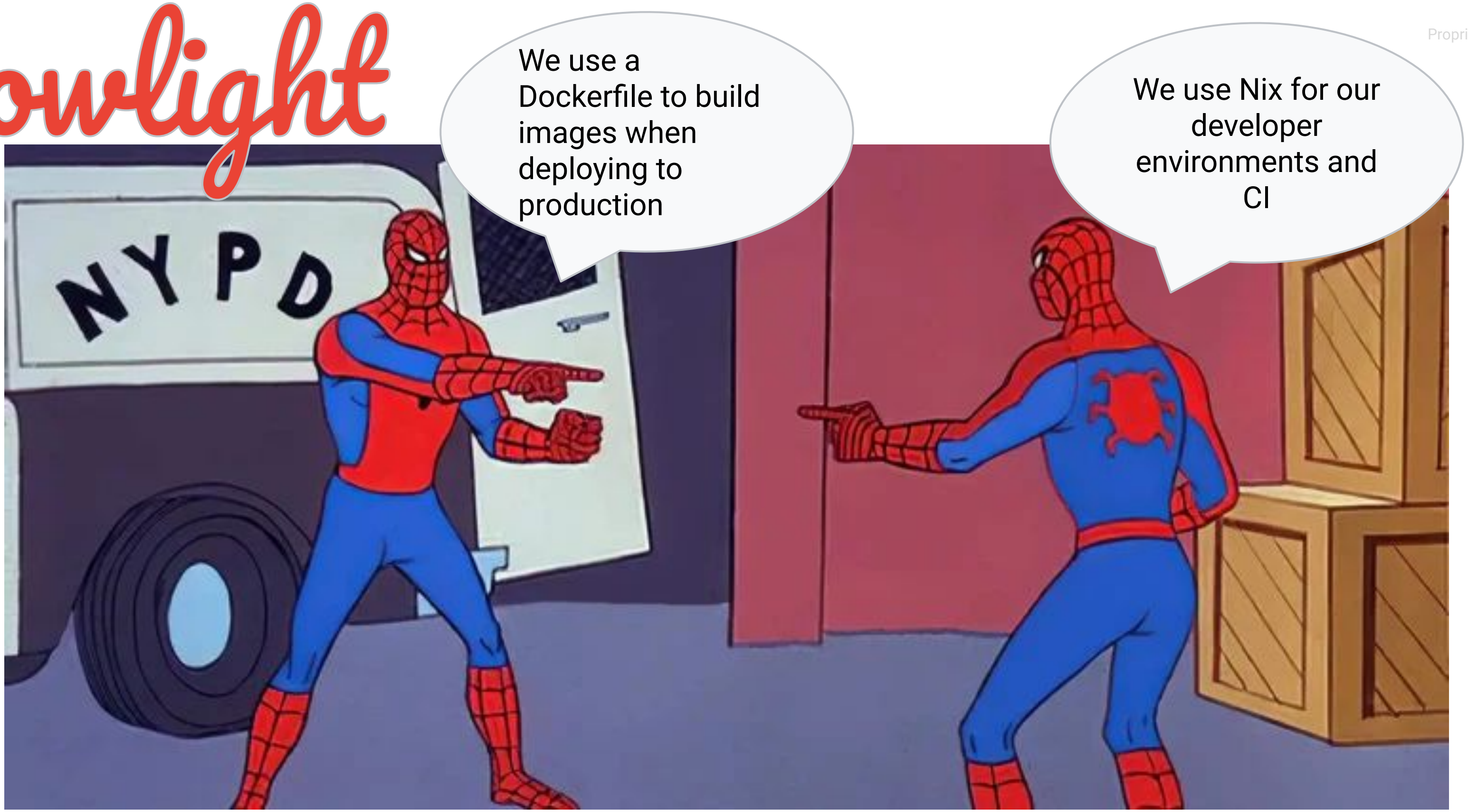
Lowlight

I want multiple versions of Node.js at an exact version with a patch applied!

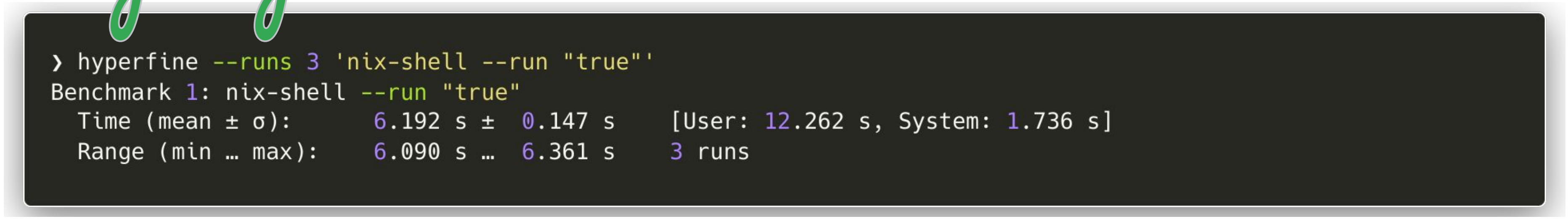
Lowlight


Sun
Neutron star
Black hole
nixpkgs
HEAVIEST OBJECTS IN THE UNIVERSE

Lowlight

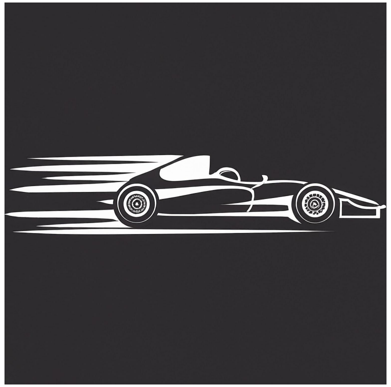

We use a Dockerfile to build images when deploying to production
We use Nix for our developer environments and CI
NYPD

Highlight



```
❯ hyperfine --runs 3 'nix-shell --run "true"'
Benchmark 1: nix-shell --run "true"
  Time (mean ± σ):      6.192 s ±  0.147 s    [User: 12.262 s, System: 1.736 s]
  Range (min … max):    6.090 s …  6.361 s    3 runs
```

+ direnv

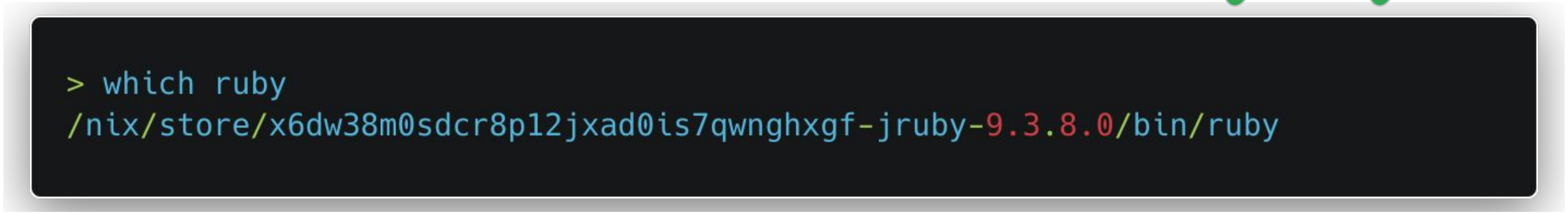


SOMETHING WRONG WITH YOUR NIX SETUP
RUN REMOVE_NIX.SH AND RESTART
imgflip.com

Highlight

Google

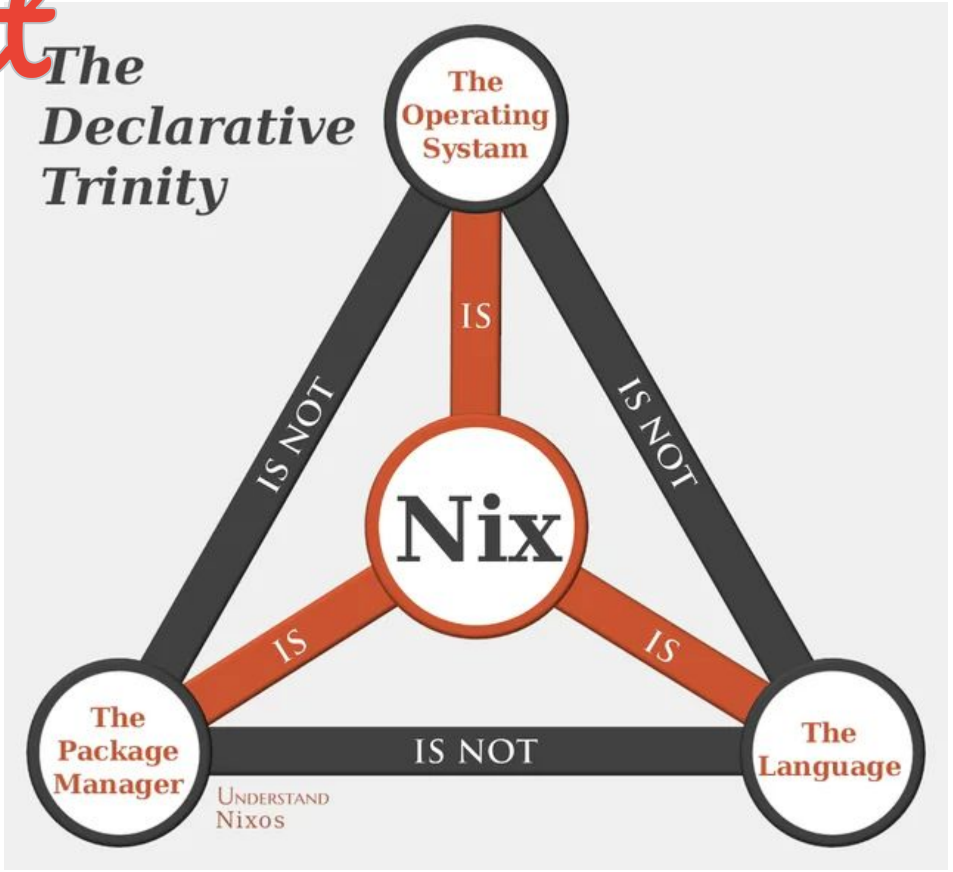
Highlight



```
> which ruby
/nix/store/x6dw38m0sdcr8p12jxad0is7qwnghxgf-jruby-9.3.8.0/bin/ruby
```

Highlight

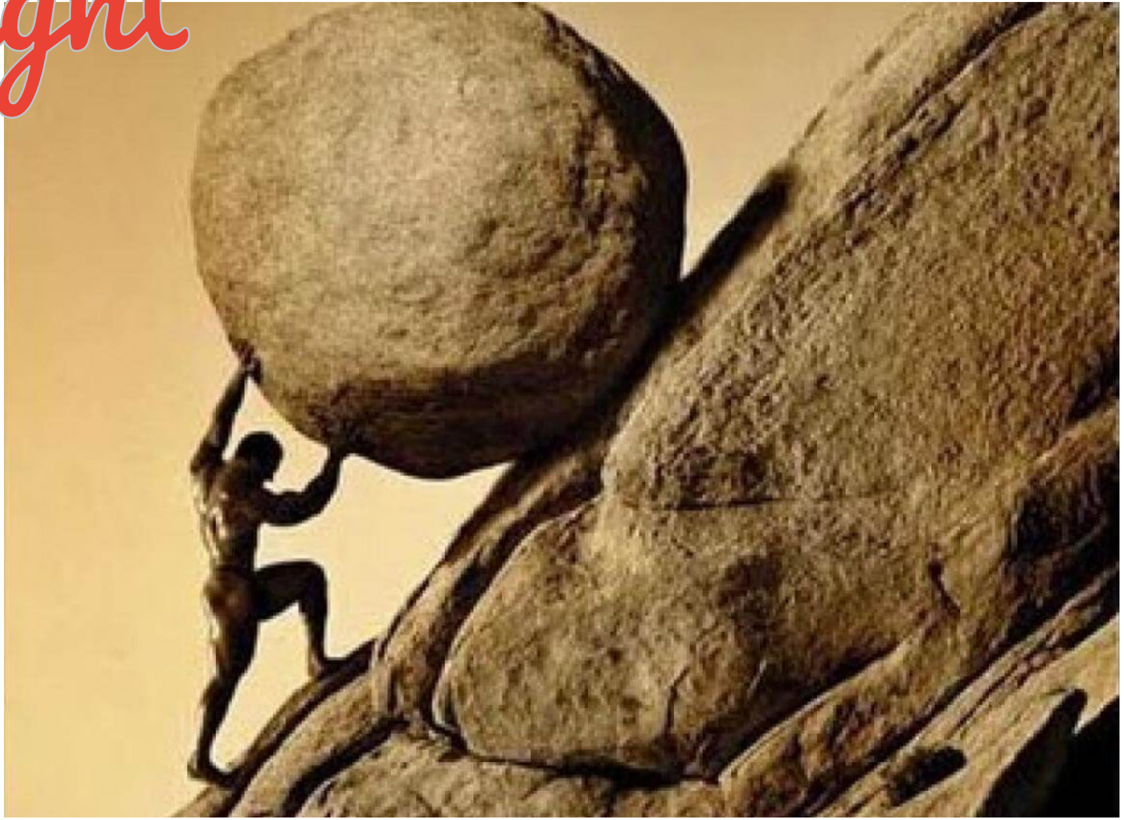
Lowlight


The Declarative Trinity
The Operating System
The Package Manager
The Language
Nix
IS
IS
IS
IS NOT
IS NOT
IS NOT
Understand Nixos

Lowlight

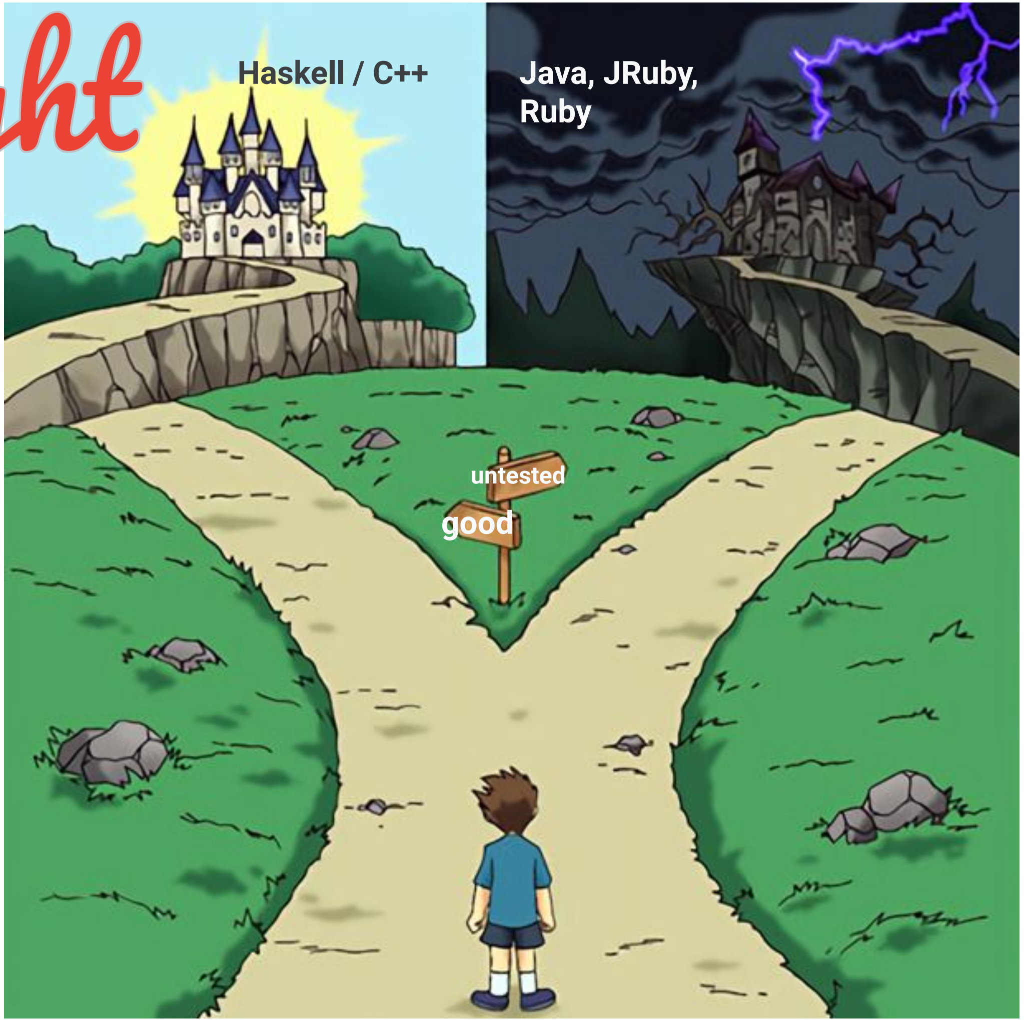
Reproducible but on whose machine?

Lowlight



Highlight

Lowlight



TODOs

- Updating the base layer for our CI
    - Each CI job pulls a pretty large amount of files from our local cache
- Using Nix to generate the container images for our deployment and in CI
- Training others how to use Nix
    - How to upgrade a package
        - “easy” upgrade is updating nixpkgs which can break a ton of things
        - “medium” upgrade is upgrading a custom derivation
        - “hard” you give up and use the same version
- More granular dependency management than just *unstable-nixpkgs*
- Integrate Bazel & Nix “properly”
    - Bazel still using it’s own toolchain rather than what Nix provides
- Remove our LD_PRELOAD hacks
    - Largely for *libnss* support in our version of Nixpkgs;
    - Changes across our branches stopped working

and despite all this, adding Nix was beneficial

Questions & Answers

Summary

- History of how we ran our tools
- Timeline
  - When was Nix first introduced
  - When did we cut over to it officially?
- Statistics?
  - # of builds we run in CI (order of magnitude)
- Benefits:
  - Switching versions of tooling has been easy for branches. (release-18, release-20, etc w/ different JVM)
  - Direnv has really made it smooth the integration of the toolchain
  - Wrapping tools has been straightforward such as bazel to include secret key management
  - We had a more heterogeneous setup: Mac, Linux
- Negatives:
  - We've upticked the
  - Some friction w/ overlapping Ruby Gem dependencies of the product itself and of the developer environment (like remote debugging gem)
    - Persisted processes like IDE that pickup session environment can cause problems across changes to the nix-shell
    - IDE must be launched from within the nix-shell to pickup the tooling
  - Config management of users sometimes caused issues.
    - Our solution: Just blow them away
    - Override the NIX_CONFIG directory to one we control
- What's the problem we set to achieve with Nix?
  - It wasn't really deployment to customers but only actually to unify the CI and developer workstations
- Prepare_shell:
  - Blowing away everything and restarting has been a straightforward approach to solving issues when they come
  - Issues have been mostly around: direnv and unfound impurities
- Java:
  - I fixed a couple of Java impurities. Mileage of Nix varies depending on how widely used that ecosystem is.
- S3 cache
  - Couldn't use public cache and hammer it from our CI
  - Started more as a requirement. We've even had to cache the nixpkgs tarball since GitHub would rate limit us
- Build everything from source not hydrated from public cache; wth mixed results
  - Chromium specifically was a nightmare to build and bring in
- We did not use flake
- Overlapping functionality of bazel + nix
  - Bazel used to build product
  - Nix only used to bring tooling
- Organizational challenge is we couldn't see Nix all the way through even to CI
- https://docs.google.com/spreadsheets/d/1l0s1uAUQ6SuxqNPGCAYued7o7N2-qw4lssOMDAzkFqk/edit?resourcekey=0-oiUiFhkbrwok2nxrDk3gxw#gid=819240326
- https://docs.google.com/document/d/13bUrMsNEiZDqAIOpYq9SC9qdKuSfUGUBil_jPd5b7qs/edit?resourcekey=0-jYFnpqileNQruJrDba_LPw&tab=t.0
- https://flox.dev/blog/nitw-looker