

Nix Shells: Supercharging Your Projects with Reproducibility, Reliability & Declarative Control



By

Vivekanandan KS

<https://linksta.cc/@vivekanandanks>

Attendees type & vibe check:

- 1) NixOS users?
- 2) Nix users?
- 3) Linux users?
- 4) DevOps people?
- 5) Scripting people?
- 6) Just curious people 😊?

Before we start - The Prerequisites:

- Any of the below systems with Nix Installed:
 - a) Linux(any distro of your choice),
 - b) Mac,
 - c) Windows (WSL- Windows subsystem Linux)

Nix installer link: (Determinate Nix Installer)

<https://determinate.systems/posts/determinate-nix-installer/>

OR

- NixOS inside a VM (with flakes enabled)

Ready for Nix?







Let's Start 🔥

Arc 1: The Imperative Curse & The Quest for Determinism







Decades of fundamental problems in the IT world

“

I used to keep a README  full of setup steps.
Then I had a Makefile . Then a Dockerfile .
Then a setup.sh, a requirements.txt , and a ci.yml .
And still... things broke. 

“

Someone would clone the repo and ask:
"What version of Python does this need? "
"Do I install that with Brew  or Pip ?"
"Why doesn't the Docker build work on my machine? "

Decades of fundamental problems in the IT world

In today's fragmented tooling landscape , projects rely on multiple moving parts:

- Language-specific package managers

- System setup scripts

- CI configurations, etc

The culprit: IMPERATIVE (Creates Divergent system)

Imagine giving the same ingredients and recipe to many and somehow the food tastes different.

But, this difference is a pain in the IT world. We've become accustomed to this behavior through repeated troubleshooting and temporary fixes. And the more the tool stacks increase, hell seeps through these small cracks.


Even identical systems diverge over time, if started and maintained the same way.

Adjustments and midway script fail rollbacks become frequent habits.

Dependency Hell:

Traditional PMs- Global dependencies and assumes only one right version

Lang specific solution - venv (for python), etc. But cross language environment support is a pain.

VMs solved, but with the tinkering overhead on u.
Containers solved the overhead of VMs resources
and more. Containers are great  but a bit overkill
and heavy for scripting and other local workflows
because of base image, reliance on conventional
package manager and imperative dockerfile etc.

Let's have a very basic
dockerfile example:

```
FROM ubuntu:latest
RUN apt-get update && \
    apt-get install -y curl wget
WORKDIR /app
COPY . .
CMD ["curl" "--version"]
```

Dockerfile Problems:

- 1) Repeatable, not reproducible
- 2) LTS end version change
- 3) App removed from repo
- 4) Running same dockerfile after months or years will yield different result

Same Old Story, Different Surface:

Earlier: You made imperative steps(scripts) to set & patch things up in VMs. 😊

Now: You make imperative steps(dockerfile) to set & patch things up in Docker. 😊

Both creating the imperative cracks 💔.

Good Old Meme



Are we doomed to continuously tinker stuffs and pray that it works?



Are we cursed to continuously sync dev, test and prod environments in CI/CD?



One Solution: DECLARATIVE

Instead of instructing how to achieve a result,
You precisely declare what you want 🎯.

Advantages:

Portable 

Atomic 

Reproducible 

Tracked Changes 

And what if still that's not enough?

How to actually solve 🤔:

Imperative cracks? 💣🧱

Diverging environments? ↔️🌍

Repetitive script failures? ↺️❌📜

Inconsistent builds across systems? ≠ ⚙️💻

Arc 2: Nix Unleashed: The Declarative Revolution Begins



Nix - The Revolutionary Declarative system for
Determinism, Reliability and Precise control.



Nix is popularly synonymous to these:

(PF = Purely Functional -> Declarative -> Atomic -> Reliable)

- 1) Nix - The PF Programming Language, Build Tool
- 2) Nixpkgs - PF Package Manager and repo
- 3) NixOS - PF Linux Distro
- 4) Specific Nix projects - PF [Shell, Containers, VMs, Custom ISOs etc] (Anything as Code)

Nix takes declarative a step ahead through purely functional approach.

Nix is a universal build bootstrapping tool, which brings a language agnostic way to build, cache and store artifacts.

Artifacts can be - Shells, packages, containers, VMs, Custom ISOs etc

Language Agnostic = Not tied to a single ecosystem like python, rust, etc

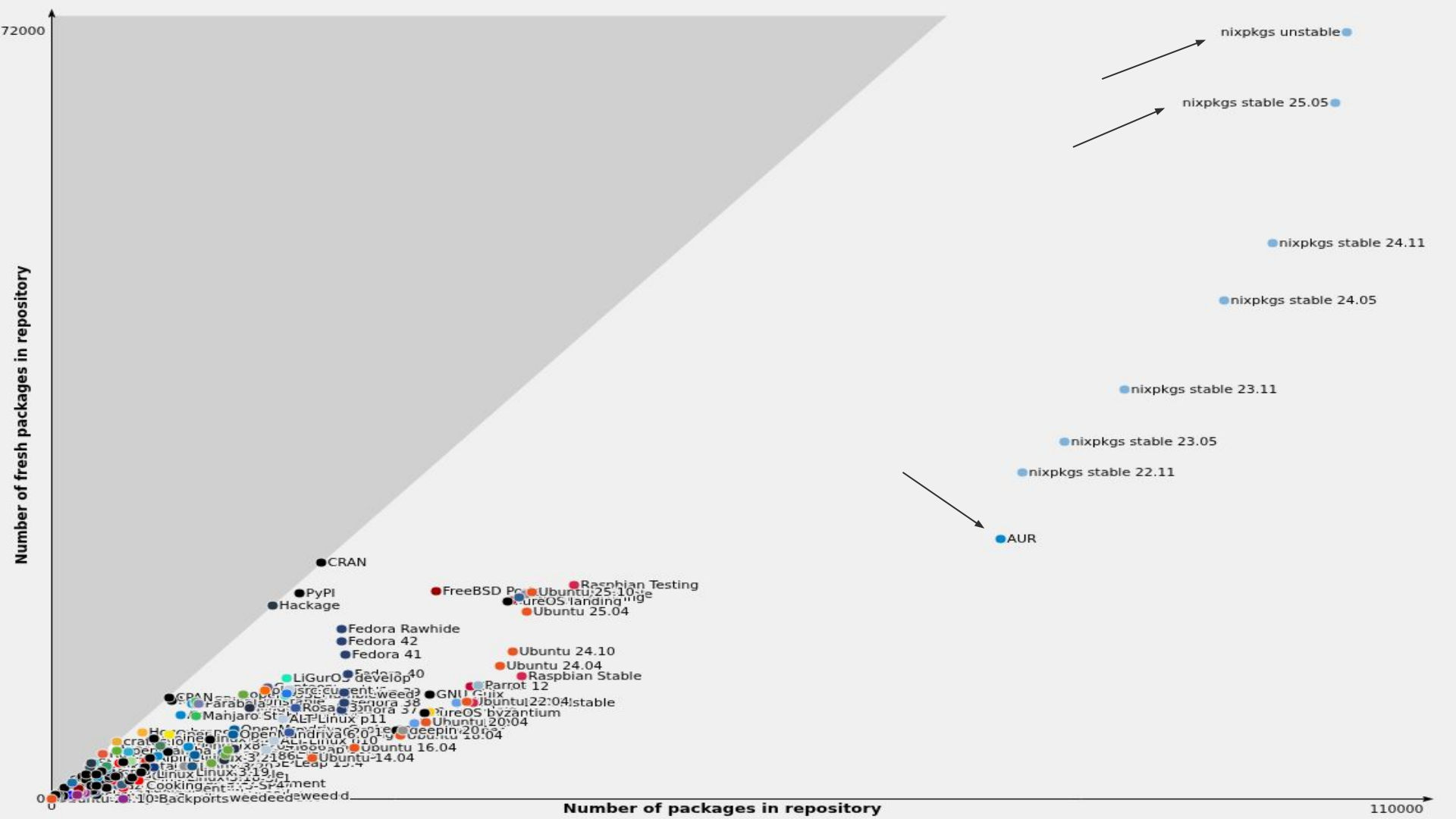
Nix Store : The Immutable Read Only Store

How it's different from traditional PMs - Hashing, Symlinks.

a) Hashing - Thus coexistence of any number of versions of same package 🔥.

Eg:

`/nix/store/b6gvzjyb2pg0kjfwrjmg1vfhh54ad73z-firefox-33.1`



Live graph:

<https://repology.org/repositories/graphs>

Nixpkgs search: <https://search.nixos.org/packages>
(120,000 packages while preparing this PPT 🔥)

Guess the size of the whole nixpkgs repo? 🤔

How many TBs or 100s of GBs?

Answer ->

It's Just 5.05 GB 🔥 🔥 🔥 🔥 🔥

Including all revisions and releases from the past

Because everything are just bootstrapped nix files, all text ,
with all revisions of older packages too. It's super efficient.

nixpkgs use source repo model instead of binary repo model
with optional hashed binary caching substituters. Eg:
`cache.nixos.org`

Caching - Reusing the artifacts

- a) Public cache - cache.nixos.org, etc
- b) Local cache are stored in Nix store `/nix/store` along with other derivations.

Derivations = Artifact produced by nix, could be anything from text, file, OS etc.

[Back to nixos.org](#)[Packages](#)[NixOS options](#)[Flakes](#)[Experimental](#)

Q Search more than **120 000** packages

Channel: **24.11** **Deprecated** **25.05** **unstable**

Please help us improve the search by [reporting issues](#).

♥ Elasticsearch instance graciously provided by [Bonsai](#). Thank you! ♥

Find all packages
here:

<https://search.nixos.org/packages>

That's a simple bird's-eye view of Nix so far. Nix solves stuffs in various layer.

And for this workshop let's focus more on....

Arc 3: The Scripting Saga: Conquering the Chaos



Problems around modern scripting

To Attendees:

List out some problems/difficulties in scripting.




Question for u all 🤔:


Why organisation restrict their employees from installing external packages in their work system?

Scripting hardships:

- 1) “It works on my machine”
- 2) Installing dependencies pollute the user space.
- 3) Even when u install dependencies in a pre-Script phase, the post-Script phase cleaning process should be careful to avoid polluting the user space again.
- 4) Imagine midway script failure during any of these phases 🦴. Manual Rollbacks and tinkering becomes a maintenance overhead.

One Solution:

Create an image from a DockerFile , source the local directory as working directory inside it  and then mount a volume to persist the stateful changes .

 Still with all the mentioned problems with DockerFile.

Reproducible Scripts with Docker. Easy right?



Imagine doing that in CI/CD pipelines to sync dev, test & prod 😊





Your Savior is Here!

Nix Shells



Nix Shells



Temporarily expose the package from `/nix/store` to the environment.

Temporary Expose = Adding the package binaries to the `PATH` temporarily

A very neat and simple trick right? 😎

Needless to say about the advantage of multi version coexistence of the same package of nixpkgs in the `/nix/store`.

Today we gonna learn few different types of usage of Nix shells. So that u can start using them directly in ur everyday workflows.

Nix shells usage types:

- 1) Ad-Hoc Shells & One-shot method
- 2) In a script file
- 3) Scripting inside flakes(portable)

Arc 4: Nix Shells: Ad Hoc Powers Activated!



Nix Ad hoc shell environments:

Let's do some simple commands:

```
echo "hello world" | cowsay
```

But cowsay doesn't exist. So let's enter a nix shell with the cowsay package,

```
nix-shell -p cowsay
```

Now try the command again:

```
echo "hello world" | cowsay
```

What's happening?

- 1) Nix built the packages mentioned and
- 2) Nix created a temporary shell environment with the built packages' binaries added to the PATH.

Now exit the shell by typing exit and try the hello world command again.

U can see that Nix removed the binaries from PATH without polluting user environment.

U can also enter a shell, run a command and exit in one shot 🔥 like this:

```
nix-shell -p cowsay --run 'echo "hello,  
world" | cowsay'
```

```
nix-shell -p cowsay --command 'echo "hello,  
world" | cowsay'
```

- `--command cmd`

In the environment of the derivation, run the shell command *cmd*. This command is executed in an interactive shell. (Use `--run` to use a non-interactive shell instead.) However, a call to `exit` is implicitly added to the command, so the shell will exit after running the command. To prevent this, add `return` at the end; e.g. `--command "echo Hello; return"` will print `Hello` and then drop you into the interactive shell. This can be useful for doing any additional initialisation.

- `--run cmd`

Like `--command`, but executes the command in a non-interactive shell. This means (among other things) that if you hit Ctrl-C while the command is running, the shell exits.

And if u want to run the program directly without installing in the user environment. U can do this 🔥 🔥:

```
nix run nixpkgs#cowsay -- "hello, world"
```

Which is equivalent to the normal command:

```
cowsay "hello, world"
```

Everything after the “--” are just usual flags and arguments usage.

Try it yourself:

Now make this command deterministic
with Nix:

```
echo "hello, world" | cowsay | lolcat
```

Answer:
(next slide)

(All one liner commands)

Ans:

```
nix-shell -p cowsay lolcat --run `echo  
"hello, world" | cowsay | lolcat`
```

Bonus Tip (U can also pipe commands like this):

```
echo "hello, world" | nix run nixpkgs#cowsay  
| nix run nixpkgs#lolcat
```

Towards more reproducibility and determinism 🔥💪:

```
nix-shell -p cowsay lolcat \  
--pure \  
-I nixpkgs=<NIXPKGS_REFERENCE> \  
--run 'echo "hello, world" | cowsay | lolcat'
```

where <NIXPKGS_REFERENCE> can be:

channel:nixos-<NIXPKGS_CHANNEL>

(or)

<https://github.com/NixOS/nixpkgs/archive/<COMMIT>.tar.gz>

where <COMMIT> can be:

refs/heads/nixos-<NIXPKGS_CHANNEL>

(or)

dad564433178067be1fbdfcce23b546254b6d641

#use <https://www.nixhub.io/> for finding the commit SHA for your package version

where <NIXPKGS_CHANNEL> can be:

25.05 (or) 25.11 (or) nixos-unstable etc...

Some flags:

`--keep` *name*

When a `--pure` shell is started, keep the listed environment variables.

`--pure`

If this flag is specified, the environment is almost entirely cleared before the interactive shell is started, so you get an environment that more closely corresponds to the “real” Nix build. A few variables, in particular `HOME`, `USER` and `DISPLAY`, are retained.

And explore lot more options and flags with:

`nix-shell --help`

Any Doubts so far  ?



NOTIFICATION

Leveled up!

Let's Continue 🔥

Shell Scripting

Reproducible scripts using Nix:

Imagine a simple script like this:

```
#!/usr/bin/env bash
```

```
echo "Hello, myself!" | figlet | lolcat
```

```
echo "You're a great person!" | cowsay | lolcat
```

```
python --version
```

Qn) What is wrong with this script?

👁👁 What if others run this?

One of the Ans) It assumes the system have installed the dependencies like python, lolcat, cowsay and figlet.

But what about something fundamentally wrong with this script?

Any Guess?

Ans: Next Slide

First of all, the script assumes the system already have bash installed 😓

How can we improve this?

a) Pre & Post Script installations? Nah

It'll increase the risk of polluting the user space.
Think python version override in pre script and again installing old version in postscript? What if the user installed the old one with some flags?
It's messier.

b) We can utilize nix shells we saw earlier?
Okayish.... But Nah

Using nix-shell command in each line affects the readability and not easy to maintain. Eg: If we want to bump a version we might need to go all over the scripts and change the value. It's not easy to maintain.

So how to solve all these? 🙄

**Reproducible
interpreted
scripts with Nix**

Reproducible interpreted scripts with Nix

```
#!/usr/bin/env nix-shell
#! nix-shell -i bash --pure --quiet
#! nix-shell -p bash cowsay figlet lolcat python312 nix
#! nix-shell -I nixpkgs=channel:nixos-25.05
echo "Hello, myself!" | figlet | lolcat
echo "You're a great person!" | cowsay | lolcat
```

It's the same script with nothing changed in the body 😊

But why there are multiple shebangs? 🤔 What's going on? ***Any Guess?***

```
#!/usr/bin/env nix-shell
```

The first line is the usual shebang mentioning the interpreter to use which is nix-shell in this case.

```
#! nix-shell -i bash --pure --quiet
```

```
#! nix-shell -p bash cowsay figlet lolcat python312 nix
```

```
#! nix-shell -I nixpkgs=channel:nixos-25.05
```

And the following shebang like lines are interpreted by nix-shell.

Instead of giving out the nix-shell command with all flags in a lengthy format, for readability we are splitting it into multiple lines.

Are we nesting the shell with different flags? That will start unnecessary child processes and not efficient right? Haha

Actually the nix-shell interpreter combines all those flags and options and start a single environment instead of nesting. As u might guess the shebang like syntax is the signal for it to combine the flags and options.

Now with just adding these 3 more lines at the top without even disturbing the body we have just made this script 100% deterministic to work on any system.

See we didn't even expect bash to be present, Nix is the only dependency we need. Super cool right? 😁

Let's break down the new flags:

```
#! nix-shell -i bash --pure --quiet
```

`--quiet` : Just avoid the verbose output while building the dependencies

`-i bash` : Chooses the interpreter to use for the of the script

Let's extend the script body just a little:

```
#python from script
```

```
echo Your python version for this script is:
```

```
python --version
```

```
echo -----
```

Imagine: In the dependencies we mentioned python 312 and in between the script we want to use another particular python version for a legacy program. Is that even possible?

At this point HOW should be the question. Let's reuse that ad hoc shell idea here. It's simple as that.

```
#another version of python as a nested nix shell
```

```
echo Your another python version is:
```

```
nix-shell -p python314 \  
  --pure --quiet \  
  -I nixpkgs=channel:nixos-25.05 \  
  --command 'python --version'
```

Now let's look at the output of the script, just to verify.

~/Documents/ksvniworkshop

>>> ./script.sh

WELCOME, NAME, EXID

< You're a great person! >

^ ^
^ ^
(oo)\
()\)\ \

Your python version for this script is:
Python 3.12.11

Your another python version is:
Python 3.14.0b4

~/Documents/ksvniworkshop

took 4s

>>> □

So here just for a quick one time run for a program we literally used another python version. 😊 U can literally use

- any number of different versions of
- any program together in
- any script without conflict 😎 and super reliable.

All with just only Nix as dependency 🔥

Any Doubts so far  ?



NOTIFICATION

Leveled up!

**Want to
Level Up
more?**



Let's do it



Scripting inside flakes(portable):-

The same scripting, but instead of in a separate .sh file we gonna do it inside a nix flake(just a config format).

Some advantages are:

- a) Easily shareable and when saved in a git repo, just a single command is needed to run your app on anyone's machine.
- b) More precise control over the dependencies,
- c) Lock file,
- d) tight integration with nix ecosystem etc

Don't get overwhelmed. It's easy and nix does the heavy lifting for you 😊.

Arc 5: Unlocking the Functional Texts: Nix Language Fundamentals



Let's learn some Nix language basics in just few mins.

It's a super easy language.

(kind of like JSON with Functions 😏)

Data Types (Primitive)

```
integer    = 4
```

```
float      = 3.14
```

```
string     = "nix"
```

```
multilineString = ''
```

```
    hello
```

```
    Greetings my friend
```

```
''
```

```
paths     = ./Documents
```

String Interpolation(substitution):

```
string = "nix";  
    stringWithInterpolation = '  
        hello ${string}  
    ';  
;
```

Data Types (List)

```
# list values are separated by  
#whitespaces(space or new line)  
list = [hello 1 2.24]  
#same list more readable  
list1 = [  
    hello  
    1  
    2.24  
]
```

Data Types (Set/ Attribute)

```
# set/attributes values  
#are separated by ;  
set = {  
    name = "hello";  
    values = [1 2 3 ];  
    set2 = {a = 1; b = 2;};  
}
```

Nested sets can be
defined
#in a piecewise
fashion.

```
{  
    a.b    = 1;  
    a.c.d  = 2;  
    a.c.e  = 3;  
} .a.c  
#=> { d = 2; e = 3; }
```

Same thing when expanded:

#Nix

```
{  
  a = {  
    b = 1;  
    c = {  
      d = 2;  
      e = 3;  
    };  
  };  
}
```

#JSON

```
{  
  "a": {  
    "b": 1,  
    "c": {  
      "d": 2,  
      "e": 3  
    }  
  }  
}
```

let in bindings

```
let
```

```
  x = "a";
```

```
in
```

```
{
```

```
  name = x;
```

```
}
```

Nix have only lambda functions

```
# nix
```

```
x: x + 2
```

```
#python
```

```
lambda x: x + 2
```

```
#javascript
```

```
(x) => x + 2;
```

```
#java
```

```
(x) -> x + 2;
```


parameter

body

$x : x + 2$

colon

$(x : x + 2) 10$

> 12

Try it out yourself!

```
[user@host ~]$ nix repl
```

```
Nix 2.24.12
```

```
Type :? for help.
```

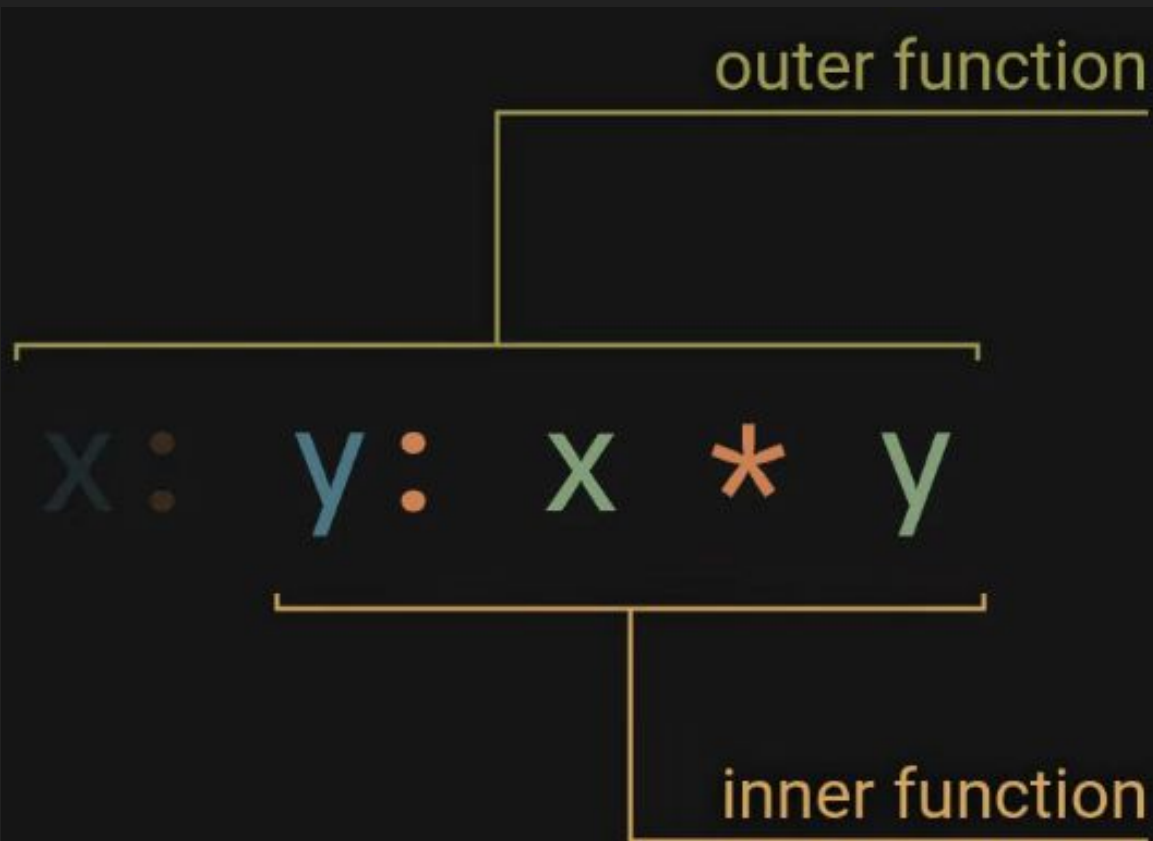
```
nix-repl> (x: x + 2) 10
```

```
12
```

```
nix-repl>
```

$x:$ $y:$ $x * y$

**Nix
functions
have only
one
parameter**



So we give a set as the parameter and access the values from it

```
param: param.x * param.y
```

```
let
```

```
    mul = param: param.x * param.y
```

```
in
```

```
    mul { x = 2; y = 10; }
```

```
> 20
```

```
let  
  mul = param: param.x * param.y  
in  
  mul { x = 2; y = 10; }  
  
> 20
```

**<- Bad
readability**

**Better
Readability ->**

```
let  
  mul = { x, y }: x * y  
in  
  mul { x = 2; y = 10; }  
  
> 20
```

Default values for the function

```
let  
  func = { x ? 6, y ? 5 }: x + y  
in  
  func { }  
  
> 11
```

Both idea into one: (flexible)

Set and destructured set values

```
{ x, y }: x + y
```

```
param: param.x + param.y
```

```
{ x, y } @ param: x + param.y
```

Any Doubts so far  ?



NOTIFICATION

Leveled up!

Arc 6: The Flake Awakening: Configuration of Destiny



Nix flakes:-

Portable

**configuration(mostly
boilerplate) to declare things
precisely.**

Boilerplate (Don't worry about the details for now)

```
{  
  description = "flake";  
  
  inputs = {  
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-25.05";  
    flake-utils.url = "github:numtide/flake-utils";  
  };  
  
  outputs = { self, nixpkgs, flake-utils, ... }:  
    flake-utils.lib.eachDefaultSystem (system:  
      let  
        pkgs = import nixpkgs { inherit system; };  
  
      in {  
  
        packages.default = {  
  
          };  
        }  
      );  
}
```

Flakes is just a set with 3 parts;

1) description: Type - String

Name of the flake configuration. (optional but good practice)

```
description = "flake";
```

2) inputs: Type - Set

We set all the inputs here

```
inputs = {  
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-25.05";  
    flake-utils.url = "github:numtide/flake-utils";  
};
```

3) outputs: Type - Function

```
outputs = {}:
  let
    in {
      in {
      }
    }
  );
```

```
outputs = { flake-utils, ... } @ inputs:
  flake-utils.lib.eachDefaultSystem (system:
    let
      in {
      }
    ) ;
```

Full outputs Boilerplate:

```
outputs = { self, nixpkgs, flake-utils, ... } @ inputs:
  flake-utils.lib.eachDefaultSystem (system:
    let
      pkgs = import nixpkgs { inherit system; };

      shell-app = pkgs.writers.writeBashBin "mybashshellapp" {} '''' ;

    in {

      packages.default = shell-app;
    }
  );
```

Nix Writer function for Bash

3 arguments:

name: String

args: Set

script : String

```
shell-app = pkgs.writers.writeBashBin
  #app name
  "mybashshellapp"
  #extra arguments
  {
    makeWrapperArgs = [
      "--prefix" "PATH" ":" "${pkgs.lib.makeBinPath
[
  #add dependencies here
] }"
  ];
}
#script without shebang
''
```


Adding dependencies

```
#extra arguments
{
    makeWrapperArgs = [
        "--prefix" "PATH" ":" "${pkgs.lib.makeBinPath [
            #add dependencies here
            pkgs.cowsay
            pkgs.lolcat
            pkgs.figlet
            pkgs.python312
            pkgs.nix
        ]}"
    ];
}
```

Checkout the script inside flake at:

[https://github.com/vivekanandan-ks/
ksvnixscriptdemo/blob/main/flake.nix](https://github.com/vivekanandan-ks/ksvnixscriptdemo/blob/main/flake.nix)

Note this portion:

The first interpolated version will work perfectly, but the second time hello is called it'll throw an not found error. Guess what's happening here?

```
echo hello version is:
```

```
${pkgs.hello}/bin/hello --version
```

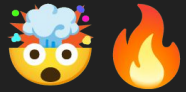
```
echo
```

```
echo again hello version is:
```

```
hello --version
```

Ans:

Interpolated method runs the binary only that time and doesn't even expose the package to the script



So it's possible to run a different version of application without polluting not just the user space, but also not polluting even the script environment 🔥 🔥 🔥

There are many such writers for other languages too with their own specific options, so feel free to explore scripting in your favourite language without worrying about dependencies management, version conflicts etc etc

Check out :

<https://noogle.dev/q?term=pkgs.writers>

At this point u can aura farm even
with ~~the Ant~~ Veteran scripters.



Congratulations U have scratched
the surface of Nix's powers



What we learnt today is how we can use Nix to supercharge scripting.
And it'll help u understand further into the nix ecosystem where nix revolutionize things like:



NOTIFICATION

Leveled up!

Arc 7: Beyond the Horizon: The Vastness of the Nix Ecosystem



```
pythonEnv = pkgs.python313.withPackages (p: with p; [  
    #Add the dependencies here  
    requests  
]);
```

```
myShell = pkgs.mkShell {  
    packages = [  
        pythonEnv  
        #add your packages  
    ];  
  
    shellHook = ''  
        echo "Welcome to custom nix shell"  
    '';  
};
```

Development Shell Environments

```
# The Python application
myApp = pkgs.stdenv.mkDerivation {
  name = "my-python-app";
  src = gitignoreSource ./.; # Use the current directory as source

  # Install phase: copy the Python files to the output
  installPhase = ''
    mkdir -p $out/app
    cp -r app $out/
    mkdir -p $out/bin
    cat > $out/bin/start-app <<EOF
    #!${pkgs.bash}/bin/bash
    exec ${pythonEnv.interpreter} $out/app/main.py
    EOF
    chmod +x $out/bin/start-app
  '';
};
```

App build

```
dockerLayeredImage = pkgs.dockerTools.buildLayeredImage {  
  name = "my-python-app";  
  tag = "latest";  
  contents = [ myApp ];  
  config = {  
    Cmd = [ "/bin/start-app" ];  
    WorkingDir = "/app";  
  };  
};
```

docker
(container)
image

NixOS

The declarative and functional OS which is atomic, reliable, deterministic, easy rollbacks etc.

Imperative

```
$ sudo apt install package1  
$ sudo apt install some-service  
$ sudo systemctl enable some-service.service  
$ sudo systemctl disable old-service.service  
$ sudo apt install package2
```

Declarative

```
{  
  services.some-service.enable = true;  
  
  environment.systemPackages = with pkgs; [  
    package1  
    # package2  
  ];  
}
```

Firefox enable option:

1. Adds firefox to `environment.systemPackages`
2. Configures firefox with related options



```
programs.firefox = {  
  enable = true;  
  package = pkgs.firefox-beta;  
  policies.Homepage.StartPage = "https://nixos.org";  
  policies.DisableTelemetry = true;  
}
```


Git enable option:

1. Adds git to `environment.systemPackages`
2. Configures git with related options

```
programs.git = {  
  enable = true;  
  config = {  
    user.name = "vimjoyer";  
    user.email = "hello@vimjoyer.com";  
    init.defaultBranch = "main";  
    pull.rebase = true;  
  };  
};
```





Openssh enable option:

1. Installs sshd
2. Configures sshd with related options
3. Set up a systemd service for sshd
4. Open required firewall ports

```
services.openssh = {  
    enable = true;  
    settings.PermitRootLogin = false;  
    settings.PasswordAuthentication = false;  
    ports = [ 22 ];  
}
```

Steam enable option:



1. Installs steam
2. Configures steam with related options
3. Enables hardware accelerated graphics drivers
4. Enables 32bit pulseaudio/pipewire support
5. Ensures steam hardware works

```
programs.steam = {  
    enable = true;  
  
    extraCompatPackages = [ pkgs.proton-ge-bin ];  
}
```

```
# Podman
virtualisation.containers.enable = true;
virtualisation = {

    podman = {
        enable = true;
        dockerCompat = true;
        dockerSocket.enable = true;

        # Default network settings
        defaultNetwork.settings = {
            dns_enabled = true;
        };
    };
};
```

```
virtualisation.docker.rootless = {
    enable = true;
    setSocketVariable = true;
};
```

```
#limine boot
boot.loader = {
  limine = {
    enable = true;
    style.wallpapers = lib.filesystem.listFilesRecursive ./limine-images; #list of wallpaper paths
    #style.wallpaperStyle = "centered";
    extraEntries = ''
      /Windows
      protocol: efi
      path: uuid(1c135138-506a-45ed-8352-6455f45e9fea):/EFI/Microsoft/Boot/bootmgfw.efi
    '';
    extraConfig = ''
      remember_last_entry: yes
    '';
  };
};
```



```
#enable flatpak
services.flatpak.enable = true;
```

```
# enable appimage support
programs.appimage.enable = true;
programs.appimage.binfmt = true;
```

```
#enable fish shell
programs.fish = {
  enable = true;
  shellAliases = {
    rm = "echo Use 'rip' instead of rm." ;
    rip = "rip --graveyard ~/.local/share/Trash" ;
  };
};
```

```
# Enable the X11 windowing system.
```

```
# You can disable this if you're only using the Wayland session.
```

```
services.xserver.enable = true;
```

```
# Enable the KDE Plasma Desktop Environment.
```

```
services.displayManager.sddm.enable = true;
```

```
services.desktopManager.plasma6.enable = true;
```

```
# Enable sound with pipewire.  
services.pulseaudio.enable = false;  
security.rtkit.enable = true;  
services.pipewire = {  
  enable = true;  
  alsa.enable = true;  
  alsa.support32Bit = true;  
  pulse.enable = true;  
  #jack.enable = true;  
};
```



```
#root password
users.users.root hashedPassword = "$6$/Yo/IR.A6rGbFVr6$a6c7yhjPYGu
# Define a user account
users.users.ksvnixospc = {
  isNormalUser = true;
  description = "ksvnixospc";
  extraGroups = [ "networkmanager" "wheel" "podman" ];
  hashedPassword = "$6$DmrUUL7YWFMar6aA$sAoRlSbFH/GYETfXGTGa6GSTEs
  shell = pkgs.fish;
  packages = (with pkgs; [
    #stable
    kdePackages.kate
    python3
  ])
}
```

```
environment.systemPackages =  
  (with pkgs; [  
    #stable  
    vim  
    kdePackages.partitionmanager  
  ])
```

```
++
```

```
(with pkgs-unstable; [  
  #unstable  
  wget  
  nano  
  micro
```

```
# Open ports in the firewall.  
networking.firewall.allowedTCPPorts = [ ... ];  
networking.firewall.allowedUDPPorts = [ ... ];  
# Or disable the firewall altogether.  
networking.firewall.enable = false;
```

NixOS generators

```
# build with flakes | nixos-generators
```

```
[user@nixos]$ nix run nixpkgs#nixos-generators -- \
--format iso --flake /path/to/flake#exampleIso -o result
```

NixOS VMs

```
nixos-rebuild -flake .#vm build && \  
./result/bin/run-vm
```

Ask your doubts

Thought of the day:

Hard work < Smart work < Right Work

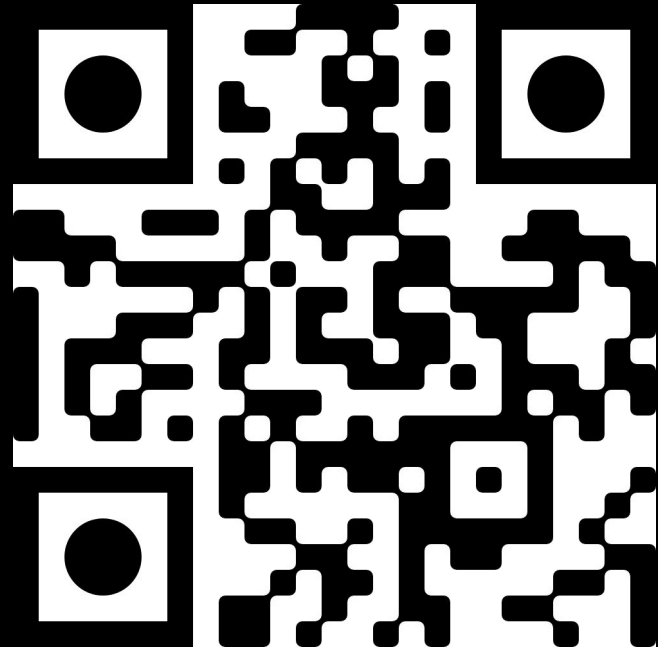


Thank You All 😊

Vivekanandan KS

<https://linksta.cc/@vivekanandanks>

Feedback Form: (Next Slide)



Feedback Form:

<https://forms.gle/7F4V1CdrfTvBKDcZ8>

