ADMIN:DOCS

NixOS: Der Leitfaden vom Paketmanager Nix

Funktionsweise von Nix und NixOS: Derivations, der Nix Store, deklaratives Konfigurationsmanagement mit Flakes sowie Rollbacks und Isolation im praktischen Sysadmin-Alltag.

Kategorie: Linux-Neulinge **Veröffentlicht:** 24.10.2024 **Zuletzt geprüft:** 08.09.2026

Klassische Linux-Distributionen folgen seit Jahrzehnten einem vertrauten Muster: Ein Paketmanager entpackt Dateien direkt in gemeinsame Systemverzeichnisse wie `/usr/bin`, `/usr/lib` oder `/etc`. Auf Systemen mit APT unter Ubuntu, DNF unter Fedora, Zypper unter openSUSE oder Pacman unter Arch Linux funktioniert das im Alltag verlässlich – stößt aber an prinzipbedingte Grenzen, sobald widersprüchliche Anforderungen aufeinandertreffen.

Benötigt ein älteres Administrationsskript eine bestimmte Version einer OpenSSL- oder Python-Bibliothek, während eine neu installierte Webanwendung zwingend neuere Bibliotheksversionen verlangt, gerät das globale Dateisystem in Konflikt (*Dependency Hell*). Wird ein Paket aktualisiert, überschreibt es geteilte Dateien anderer Programme. Schlägt ein Systemupdate fehl oder bricht mitten im Entpackvorgang ab, verbleibt das System oft in einem inkonsistenten Zwischenzustand, der manuelle Reparaturen erfordert.

Nix wählt ein radikal anderes Architekturmodell:

Statt Software global und zustandsbehaftet in ein veränderliches Dateisystem zu streuen, behandelt Nix jedes Softwarepaket als rein funktionale, unveränderliche (*immutable*) Einheit. Kein Paket kann ein anderes überschreiben, und jedes Programm wird isoliert mit exakt den Bibliotheken verknüpft, mit denen es gebaut wurde.

Quelle: https://admindocs.de/de/linux-neulinge/nixos-der-leitfaden-vom-paketmanager-nix

□□ **Nix vs. NixOS: Nix** ist der paketorientierte Werkzeugkasten und die dazugehörige funktionale Konfigurationssprache. Du kannst Nix gefahrlos als eigenständigen Paketmanager auf bestehenden Distributionen wie Debian, Ubuntu, Fedora, Arch Linux oder macOS installieren, ohne dein Basissystem zu gefährden. **NixOS** hingegen ist eine vollständige Linux-Distribution, die auf dem Linux-Kernel und Systemd aufsetzt, jedoch ihr gesamtes Betriebssystem deklarativ über Nix verwaltet.

Funktionsweise & Kernkonzepte von Nix

Der funktionale Ansatz: Immutability und Derivations

Im mathematischen Sinn liefert eine reine Funktion bei identischen Eingabewerten ausnahmslos dasselbe Ergebnis, vollkommen unabhängig von äußeren Zuständen oder der Tageszeit. Genau dieses Prinzip überträgt Nix auf die Software-Paketierung.

Ein Software-Build in Nix ist das Ergebnis einer eindeutig definierten Funktion:

- **Eingaben (Inputs):** *Quellcode-Archive, Build-Skripte, Compiler, abhängige Bibliotheken und Build-Flags.*
- **Funktion (Build-Prozess):** *Eine isolierte Ausführungsumgebung ohne unkontrollierten Netzzugriff und ohne Zugriff auf fremde Host-Dateien.*
- **Ausgabe (Output):** *Ein unveränderliches Verzeichnis im zentralen Speicherort, dem Nix Store.*

Die formale Bauanleitung für ein solches Paket heißt in Nix **Derivation** (Dateiendung `.drv`). Eine Derivation beschreibt deklarativ:

- Welche Quellarchive heruntergeladen werden müssen (abgesichert über kryptografische SHA-256-Prüfsummen).
- Welche Abhängigkeiten und Compiler-Werkzeuge in welcher Fassung bereitzustellen sind.
- Welche konkreten Phasen (Konfiguration, Kompilierung, Testläufe, Installation) durchlaufen werden.

Sobald der Build abgeschlossen ist, wird das Verzeichnis im Dateisystem mit einem Schreibschutz belegt (`read-only`). Selbst der Root-Benutzer modifiziert installierte Binärdateien im Store nicht nachträglich.

Der Build-Ablauf im Detail

Nix erzwingt Reproduzierbarkeit durch strenge Kapselung. Während des Kompiliervorgangs läuft die Ausführung in einer abgesicherten Sandbox ab:



Der 32 Zeichen lange Hash-Wert im Verzeichnisnamen (z. B. `3b9a7f8e...`) ist kein Zufallsprodukt. Er repräsentiert den kryptografischen Fingerabdruck sämtlicher Eingaben: Quellcode-Hash, Compiler-Version, gesetzte Compiler-Flags und die Hashes aller eingebundenen Bibliotheken.

Wird eine einzige Zeile im Quellcode geändert oder eine abhängige Bibliothek gepatcht, ändert sich der berechnete Hash. Nix legt das neu gebaute Paket in einem völlig eigenständigen Verzeichnis ab. Bestehende Versionen bleiben unangetastet und funktionieren ohne Unterbrechung weiter.

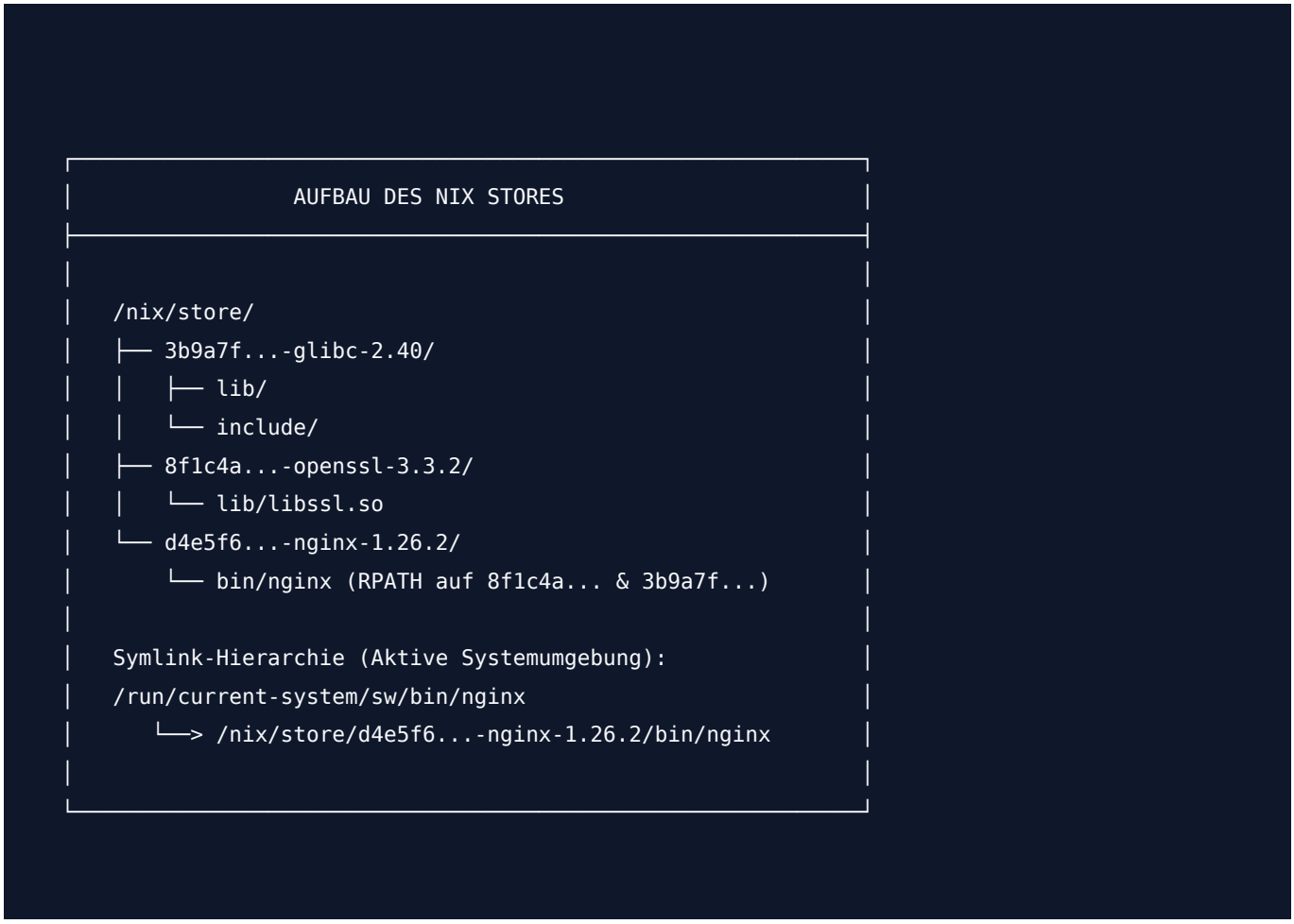
Nix im Vergleich zu klassischen Werkzeugen

Merkmal	Klassische Paketmanager (APT, DNF, Pacman)	Container-Runtimes (Docker, Podman)	Nix / NixOS
Paradigma	Imperativ (Dateisystem wird schrittweise verändert)	Image-basiert (Gekapselte Dateisystem-Schichten)	Funktional und deklarativ
Dateisystem-Struktur	Shared FHS (`/usr/bin` , `/usr/lib`)	Eigene Namespaces pro Container	Pfad-Isolation im `/nix/store`
Parallele Bibliotheken	Konfliktträchtig bei Namensgleichheit	Problemlos über getrennte Container	Nativ über eindeutige Store-Hashes
Rollback-Verhalten	Manuell, unvollständig oder dateisystemabhängig	Image-Austausch mit Neustart	Atomare Umschaltung via Symlinks
Reproduzierbarkeit	Zeitabhängig vom Repository-Spiegel	Abhängig von Basis-Images und Build-Schritten	Vollständig deterministisch
Overhead	Minimal (direkte Host-Binaries)	Speicher- und I/O-Overhead durch Schichten	Minimal (native Binaries, Deduplizierung)

Die Architektur: Nix Store, Profile & Generationen

Der Nix Store (`/nix/store`): Isolation durch Hashes

Das Fundament jedes Nix-Systems ist das Verzeichnis `/nix/store` . Hier residieren sämtliche Binärdateien, dynamische Bibliotheken, Konfigurationsschnipsel und Handbuchseiten nebeneinander:



Klassische Linux-Programme suchen dynamische Bibliotheken zur Laufzeit über den dynamischen Linker (`ld.so`) in globalen Pfaden wie `/lib64` oder `/usr/lib64`. Ändert sich dort eine Datei, betrifft das alle Programme des Systems.

Nix bricht diese Kopplung auf: Beim Kompilieren schreibt Nix den exakten, unveränderlichen Pfad der benötigten Bibliotheken direkt in den Header der Binärdatei ein (über das ELF-Attribut `RPATH` bzw. `RUNPATH`). Ein Programm unter `/nix/store/...-nginx` lädt seine Abhängigkeiten ausschließlich aus den kryptografisch verknüpften Store-Verzeichnissen. Ein systemweites Upgrade einer Bibliothek berührt laufende oder ältere Programme in keiner Weise.

> **Abhängigkeiten im Store inspizieren:** Mit dem Befehl `nix-store -q --references /path/to/binary` lässt sich für jede Binärdatei exakt auflisten, welche anderen Pfade im Store als Abhängigkeit eingebunden sind. Umgekehrt zeigt `nix-store -q --referrers /path/to/library`, welche installierten Programme eine spezifische Bibliothek aktiv

> referenzieren.

Profile und Generationen: Atomare Updates & Rollbacks

Da im Verzeichnis `/nix/store` Dutzende Versionen desselben Programms friedlich koexistieren können, muss geregelt werden, welche Programme für einen Benutzer im Suchpfad (`$PATH`) sichtbar sind. Diese Aufgabe übernehmen **Profile**.

Ein Profil ist ein Verzeichnisbaum aus symbolischen Links (Symlinks), der auf die gewünschten Programme im Nix Store verweist. Jede Modifikation eines Profils erzeugt eine neue, unveränderliche **Generation**:

```
# Paket im aktuellen Benutzerprofil installieren
nix profile install nixpkgs#htop

# Alle im aktiven Profil installierten Pakete anzeigen
nix profile list

# Historie der bisherigen Profil-Generationen einsehen
nix profile history
```

Wird ein Paket installiert, aktualisiert oder entfernt, kopiert Nix keine Dateien um. Stattdessen wird im Hintergrund ein neuer Symlink-Baum erstellt. Erst wenn dieser vollständig bereitsteht, biegt Nix den Zeiger des Profils auf die neue Generation um.

Tritt nach einer Aktualisierung ein unerwartetes Verhalten auf, lässt sich das Profil in Sekundenbruchteilen auf den vorherigen Stand zurücksetzen:

```
# Sofortiger Rollback zur unmittelbar vorherigen Generation
```

```
nix profile rollback

# Gezielt zu einer bestimmten Generationsnummer wechseln
nix profile rollback --to 3
```

Da das Umschalten über die atomare POSIX-Operation `rename()` auf Symlinks erfolgt, existieren zu keinem Zeitpunkt halbe oder unvollständige Installationszustände.

Temporäre Umgebungen mit `nix shell`

Ein herausragender Vorteil im Sysadmin-Alltag ist die Möglichkeit, Werkzeuge ad-hoc auszuführen, ohne sie dauerhaft auf dem Host zu installieren oder Abhängigkeiten in das Basissystem einzuschleppen.

Früher nutzte man hierfür `nix-shell -p`. Im modernen Nix-Standard übernimmt der Flake-fähige Befehl `nix shell` diese Aufgabe:

```
# Temporäre interaktive Shell mit Python 3.12 und PostgreSQL-Client öffnen
nix shell nixpkgs#python312 nixpkgs#postgresql_16
```

Innerhalb dieser Sub-Shell stehen `python3` und `psql` sofort zur Verfügung. Sobald du die Shell mit `exit` verlässt, ist deine reguläre Umgebung unverändert: Kein Binary verbleibt in deinem globalen `$PATH`, keine Konfiguration wurde modifiziert.

□□ Praktisches Beispiel:

Du musst auf einem Server ein komplexes JSON-Logfile analysieren, auf dem System ist jedoch weder `jq` noch `ripgrep` installiert. Anstatt das Basissystem über administrative Paketmanager zu verändern, startest du eine isolierte Sitzung:

```
# Starte eine temporäre Umgebung mit den benötigten Analysewerkzeugen
nix shell nixpkgs#jq nixpkgs#ripgrep

# Führe die Analyse direkt aus
rg "ERROR" /var/log/app/service.log | jq '.payload.error_code'

# Beende die Umgebung
exit
```

Nach dem Verlassen der Subshell zeigt ein Aufruf von `which jq`, dass das System sauber geblieben ist. Der Nix Store hält die Daten zwar im Cache, das Systemverzeichnis bleibt jedoch unberührt.

NixOS: Das vollkommen deklarative Betriebssystem

Während Nix als Paketmanager auf beliebigen Betriebssystemen läuft, treibt **NixOS** dieses Konzept auf die Spitze: Das gesamte Betriebssystem – von Kernel-Parametern und Dateisystem-Mounts über Netzwerk-Routing und Systemd-Units bis hin zu Benutzerkonten – wird als deklarativer Soll-Zustand in Konfigurationsdateien beschrieben.

Deklarative Systemsteuerung mit configuration.nix

Die zentrale Steuerdatei eines Standard-NixOS-Systems liegt unter `/etc/nixos/configuration.nix`. Anstatt Befehle wie `systemctl enable`, `useradd` oder `ufw allow` manuell einzugeben, definierst du die Maschine als Code:

```
{ config, pkgs, ... }:

{
```

```
  imports = [
    ./hardware-configuration.nix
  ];

  # Bootloader und EFI-Integration
  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;

  # Netzwerkidentität und Firewall
  networking.hostName = "production-srv01";
  networking.firewall.allowedTCPPorts = [ 22 80 443 ];

  # SSH-Dienst absichern
  services.openssh = {
    enable = true;
    settings.PermitRootLogin = "no";
    settings.PasswordAuthentication = false;
  };

  # Systemweit verfügbare Basispakete
  environment.systemPackages = with pkgs; [
    vim
    git
    curl
    htop
    tmux
    ripgrep
    jq
  ];

  # Unprivilegierter Administrator-Account
  users.users.adminuser = {
    isNormalUser = true;
    extraGroups = [ "wheel" "networkmanager" ];
    openssh.authorizedKeys.keys = [
      "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIExampleKeyValidAdmin admin@workstation"
    ];
  };

  # Zustandskompatibilität der NixOS-Installation
  system.stateVersion = "26.05";
}
```

Die deklarative Definition von SSH-Schlüsseln und Firewall-Regeln ersetzt manuelle Konfigurationsdateien unter `/etc/ssh/sshd_config`. Wie du moderne Schutzmaßnahmen wie hardwarebasierte Authentifizierung und adaptive Blocklisten etablierst, vertieft der Leitfaden zur Linux-Server-Härtung mit FIDO2 und CrowdSec.

> ⚠ **Häufiger Irrtum zu `system.stateVersion`:** Der Parameter `system.stateVersion = "26.05"` steuert **nicht** das Betriebssystem-Upgrade! Er teilt NixOS lediglich mit, mit welchen Standardannahmen für zustandsbehaftete Daten (z. B. Datenpfade von Datenbanken oder Systemd-Dateiformate) das System ursprünglich aufgesetzt wurde. Das Ändern dieser Variable ohne gezielte Datenmigration kann Dienste beschädigen. Upgrades werden stattdessen ausschließlich über die Paketquellen (Channels oder Flake-Inputs) und den anschließenden Build gesteuert.

Um die definierte Konfiguration zu aktivieren, genügt ein Aufruf des Wiederherstellungswerkzeugs:

```
# System evaluieren, neue Generation bauen und sofort darauf umschalten
sudo nixos-rebuild switch
```

NixOS analysiert die Differenz zwischen dem aktuellen Zustand und der neuen Definition, generiert die erforderlichen Systemd-Dienste, passt Konfigurationsdateien unter `/etc` an und startet betroffene Daemons atomar neu.

□□ Praktisches Beispiel:

Du möchtest eine fehlerhafte Systemänderung gefahrlos testen. Du hast in der `/etc/nixos/configuration.nix` versehentlich den SSH-Port verstellt oder einen Dienst fehlerhaft konfiguriert und angewendet.

Um unverzüglich zum bewährten Zustand zurückzukehren, führst du aus:

Quelle: https://admindocs.de/de/linux-neulinge/nixos-der-leitfaden-vom-paketmanager-nix

```
# Sofortige Rückkehr zur vorherigen fehlerfreien Systemgeneration
sudo nixos-rebuild switch --rollback
```

Sollte eine Fehlkonfiguration so schwerwiegend sein, dass das System nicht mehr ordnungsgemäß startet oder kein Netzwerkzugriff besteht, wählst du beim Booten im Menü von `systemd-boot` oder GRUB einfach den vorherigen Menüeintrag aus. Jede Generation bildet einen eigenständigen, bootfähigen Systemkern.

Moderne Konfiguration mit Flakes (Stand 2026)

Frühere Nix-Versionen verließen sich auf sogenannte *Channels*. Channels brachten den Nachteil mit sich, dass der Build-Zustand vom exakten Abrufzeitpunkt abhing: Führte man `nix-channel --update` auf zwei verschiedenen Servern an unterschiedlichen Tagen aus, konnte das System trotz identischer Konfigurationsdatei abweichende Softwarestände installieren.

Flakes lösen diese Schwachstelle vollständig. Ein Flake ist ein in sich geschlossenes Projektverzeichnis mit zwei essenziellen Dateien:

1. `flake.nix`: Beschreibt Eingabequellen (*Inputs*) und bereitgestellte Artefakte (*Outputs*).
2. `flake.lock`: Eine automatisch generierte Sperrdatei, die jeden Input auf einen unveränderlichen Git-Commit-Hash und SHA-256-Wert festnagelt.

```
┌──────────────────────────────────────────────────────┐
│                AUFBAU EINES NIX FLAKES               │
├──────────────────────────────────────────────────────┤
│                                                      │
│   flake.nix (Deklaration)                            │
│   ├── Inputs: nixpkgs (Branch: nixos-26.05)          │
│   └── Outputs: nixosConfigurations.srv01             │
│                                                      │
```

```
│   flake.lock (Automatisch gepinnt)                              │
│   └── Exakte Git-Commits & SHA-256 aller Inputs                 │
│                                                                  │
│   Ergebnis:                                                      │
│   Bitgenau identischer System-Build auf jedem Server            │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Eine praxistaugliche `flake.nix` für ein modernes NixOS 26.05-System sieht wie folgt aus:

```
{
  description = "Produktions-Server Flake nach Standard 2026";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  };

  outputs = { self, nixpkgs, ... }: {
    nixosConfigurations.production-srv01 = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        ./configuration.nix
      ];
    };
  };
}
```

Um Flakes auf einem System permanent zu aktivieren, wird in `/etc/nixos/configuration.nix` (oder in `~/.config/nix/nix.conf` bei Standalone-Nix) die folgende Direktive hinterlegt:

```
nix.settings.experimental-features = [ "nix-command" "flakes" ];
```

Das System wird anschließend direkt aus dem Flake heraus aktualisiert und aktiviert:

```
# System anhand der Flake-Definition neu bauen und anwenden
sudo nixos-rebuild switch --flake .#production-srv01

# Sämtliche Abhängigkeiten in flake.lock kontrolliert auf den neuesten Upstream-Stand heben
nix flake update
```

Projektisolierte Entwicklungsumgebungen mit `nix develop`

Flakes ermöglichen nicht nur die Konfiguration kompletter Betriebssysteme, sondern revolutionieren auch Entwicklungs- und Administrations-Workflows. Über das Attribut `devShells` lässt sich eine vollständige Arbeitsumgebung im Quellcode-Repository versionieren:

```
{
  description = "Entwicklungsumgebung für Infrastruktur-Skripte";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  };

  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in {
      devShells.x86_64-linux.default = pkgs.mkShell {
        buildInputs = with pkgs; [
          python312
          python312Packages.requests
          python312Packages.pyyaml
          ansible-lint
        ];
```

```
        shellHook = ''
          echo "Infrastruktur-DevShell geladen (Python 3.12 & Ansible-Tools aktiv)"
        '';
      };
    };
}
```

Jedes Teammitglied, das in diesem Verzeichnis den Befehl `nix develop` eingibt, erhält auf die Sekunde genau dieselbe Toolchain mit denselben Interpreter-Versionen – unabhängig davon, ob als Host-System Ubuntu, Fedora, NixOS oder macOS im Einsatz ist. Wie du eigene Automatisierungsskripte für solche Umgebungen strukturiert entwickelst, vermittelt der praxisorientierte Bash-Grundkurs für Linux-Administratoren.

Installation: Standalone-Nix vs. vollwertiges NixOS

Je nach Ausgangslage und Zielsetzung existieren zwei etablierte Bereitstellungsmodelle:

- **Standalone-Installation auf bestehenden Linux-Systemen:**

Auf bestehenden Servern oder Workstations (Debian, Ubuntu, Arch Linux, macOS) wird der offizielle Multi-User-Daemon eingerichtet. Dabei werden unprivilegierte Build-Benutzer (`nixbld1` bis `nixbld32`) angelegt, die Kompiliervorgänge strikt ohne Root-Rechte ausführen:

```
# Offizielles Multi-User-Installationsskript ausführen
sh <(curl -L https://nixos.org/nix/install) --daemon
```

- **Vollwertige NixOS-Installation auf Bare-Metal oder VMs:**

Nach dem Starten des minimalen NixOS-Installations-Images partitionierst du die Ziellaufwerke (vorzugsweise mit GPT, EFI-Systempartition und ext4/Btrfs) und bindest diese unter `/mnt` ein:

```
# Hardware des Zielsystems automatisch erkennen und Vorlagen generieren
nixos-generate-config --root /mnt

# Installation gemäß generierter /mnt/etc/nixos/configuration.nix starten
nixos-install
```

Fortgeschrittenes Paketmanagement & DevOps-Workflows

Overlays und Overrides: Eigene Patches und Build-Flags

Im Betriebsalltag entsteht gelegentlich die Notwendigkeit, eine Software mit speziellen Compiler-Flags zu übersetzen oder einen eigenen Sicherheitspatch einzuspielen, bevor dieser offiziell im Upstream-Repository freigegeben wurde.

Anstatt das gesamte Paket-Repository zu forken, bietet Nix das Konzept von **Overrides** (für einzelne Pakete) und **Overlays** (für das globale Paketökosystem):



```
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Gezielter Override in `configuration.nix`:

```
environment.systemPackages = with pkgs; [
  # Vim mit expliziter Python3-Unterstützung, aber ohne grafische X11-Bibliotheken bauen
  (vim.override {
    python3Support = true;
    guiSupport = false;
  })
];
```

Globales Overlay für Patches:

```
nixpkgs.overlays = [
  (final: prev: {
    nginx = prev.nginx.overrideAttrs (oldAttrs: {
      patches = (oldAttrs.patches or []) ++ [
        ./patches/custom-security-fix.patch
      ];
    });
  })
];
```

Nix wendet das Overlay deterministisch an: Alle Systemkomponenten, die auf `nginx` verweisen, nutzen fortan automatisch die gepatchte Version im Store.

Parallele Software-Versionen ohne Namenskonflikte

Da Nix keine globalen Pfade wie `/usr/lib` nutzt, können unterschiedliche Haupt- und Nebenversionen derselben Programmiersprache oder Datenbankbibliothek parallel auf demselben System installiert und betrieben werden:

```
# Python 3.11 und Python 3.12 zeitgleich im Benutzerprofil registrieren
nix profile install nixpkgs#python311 nixpkgs#python312
```

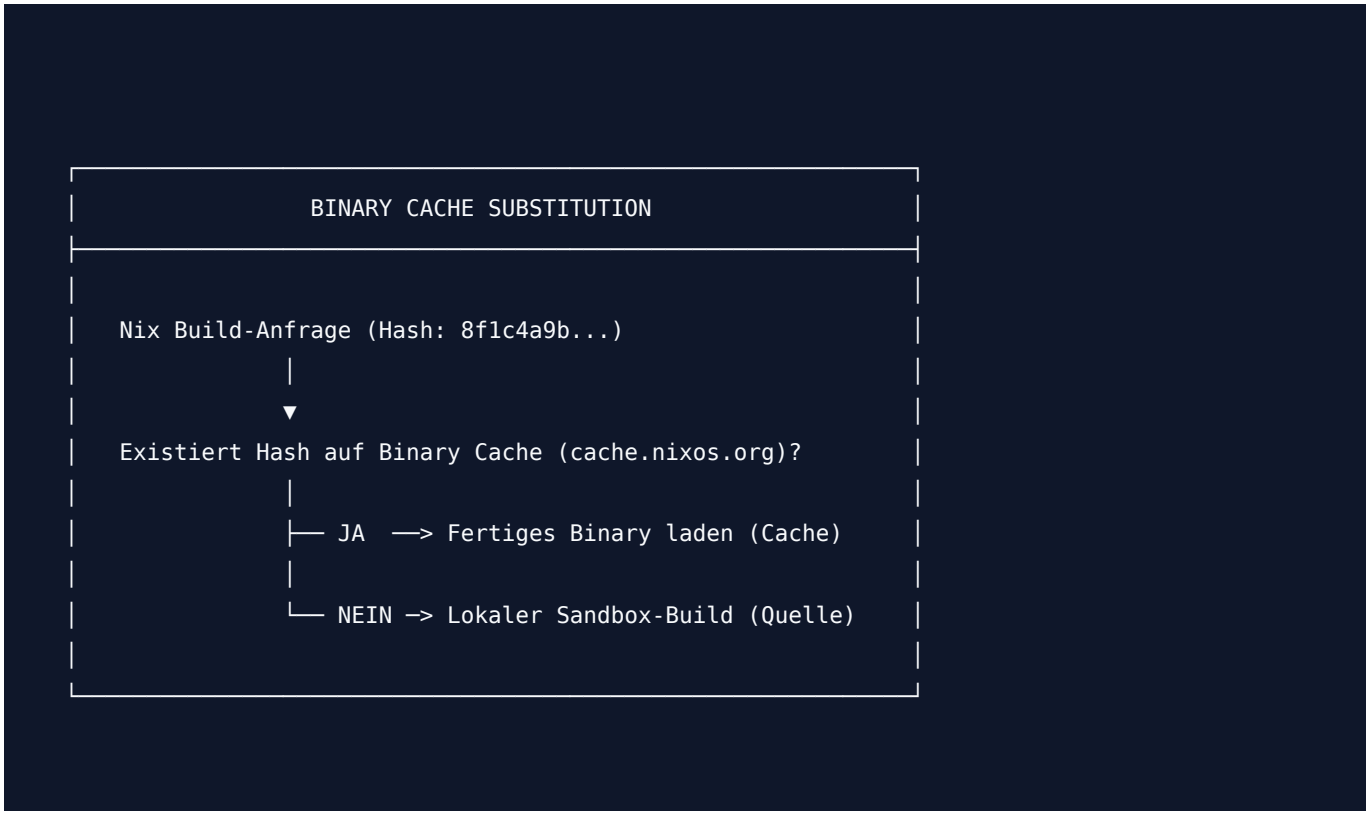
Klassische Paketmanager scheitern an solchen Anforderungen, da Header-Dateien und Symlinks kollidieren. In Nix erhält jedes Paket seinen individuellen Store-Pfad. Konflikte sind architektonisch ausgeschlossen.

> ⚠ **Warnung vor dem veralteten `nix-env`:** In älteren Anleitungen stößt man häufig auf Befehle wie `nix-env -iA nixpkgs.htop`. Im modernen Nix gilt `nix-env` als **Anti-Pattern**, da es unsauber gepinnte, schwer nachvollziehbare Zustände erzeugt. Verwende für ad-hoc Werkzeuge `nix shell`, für Benutzerinstallationen `nix profile` oder Home-Manager und für Systeme die deklarative `configuration.nix`.

Binary Caches & Substituters: Schnelle Bereitstellung

Trotz des funktionalen Quellcode-Ansatzes muss auf einem Nix-System nicht jedes Programm mühsam lokal kompiliert werden.

Sobald Nix ein Paket bauen soll, berechnet es vorab den resultierenden Hash-Wert der Derivation. Anschließend prüft Nix, ob dieser Hash bereits auf einem vorkompilierten Binär-Cache (*Substituter*) vorliegt:



Standardmäßig fragt Nix den offiziellen Cache `cache.nixos.org` ab. Liegt dort das fertige Archiv vor, wird es heruntergeladen und im lokalen `/nix/store` entpackt. Nur wenn benutzerdefinierte Flags oder eigene Patches den Hash verändern, schaltet Nix automatisch auf die lokale Kompilierung in der Sandbox um.

In Unternehmensnetzwerken lassen sich eigene Binary Caches (z. B. via Attic, Harmonia oder Cachix) einbinden, um selbst erstellte Softwarepakete einmal zentral in einer CI/CD-Pipeline zu bauen und sekundenschnell an Hunderte Zielsysteme auszuliefern.

OCI-Container mit `pkgs.dockerTools` bauen

Ein mächtiges Einsatzgebiet von Nix im DevOps-Umfeld ist das Erstellen von Docker- und OCI-Containern. Anstelle eines klassischen `Dockerfile` mit vielen unkontrollierten `RUN apt-get update`-Schritten erzeugt Nix Container-Images rein deklarativ und reproduzierbar:

```
{ pkgs ? import <nixpkgs> {} }:
```

```
pkgs.dockerTools.buildLayeredImage {
  name = "production-app";
  tag = "latest";
  contents = [
    pkgs.nodejs_22
    pkgs.curl
  ];
  config = {
    Cmd = [ "${pkgs.nodejs_22}/bin/node" "/app/server.js" ];
    WorkingDir = "/app";
    ExposedPorts = {
      "3000/tcp" = {};
    };
  };
}
```

```
# Container-Image ohne laufenden Docker-Daemon deterministisch bauen
nix-build docker-image.nix

# Das erzeugte Tarball-Image direkt in die lokale Docker-Engine laden
docker load < result
```

Da Nix exakt weiß, welche Dateien aus dem Store benötigt werden, enthalten die erzeugten Images weder unbenutzte Paketmanager-Reste noch temporäre Cache-Dateien. Das Resultat sind extrem schlanke, minimal angreifbare Container-Images.

Wartung, Speicherbereinigung & Systemdiagnose

Garbage Collection und Store-Deduplizierung

Weil Nix bei Aktualisierungen keine alten Dateien überschreibt und frühere Systemgenerationen für Rollbacks vorhält, wächst der Speicherplatzbedarf im Verzeichnis `/nix/store` kontinuierlich

an.

Die Bereinigung erfolgt über den zweistufigen Prozess der **Garbage Collection**:

1. **Alte Generationen verwerfen:** Solange ein Symlink (z. B. aus einer früheren System- oder Profil-Generation) auf einen Store-Pfad zeigt, betrachtet Nix dieses Paket als aktiv (*GC Root*). Erst wenn alte Generationen gelöscht werden, wird der Pfad zur Bereinigung freigegeben.
2. **Speicher freigeben:** Nicht mehr referenzierte Store-Pfade werden gelöscht.

```
# 1. Nur aktuell verwaiste Pakete ohne Bindeglied aus dem Store löschen
nix-collect-garbage

# 2. Alte Profil- und System-Generationen verwerfen und gründlich bereinigen
sudo nix-collect-garbage -d

# 3. Alternative über die moderne Flake-CLI:
nix profile wipe-history
nix store gc
```

Store-Optimierung durch Hardlinks

Enthalten unterschiedliche Pakete identische Dateien (z. B. identische Lizenztexte, Hilfedateien oder unveränderte Binärdateien), kann Nix diese über Hardlinks deduplizieren:

```
# Identische Dateien im gesamten Nix Store durch Hardlinks ersetzen
nix-store --optimise
```

Dieser Vorgang spart oft 20 bis 40 Prozent Festplattenplatz ein, ohne die Isolation zu gefährden, da alle Dateien im Store schreibgeschützt sind.

Automatische Bereinigung in `configuration.nix` verankern:

```
# Automatische Deduplizierung bei jedem Build aktivieren
nix.settings.auto-optimise-store = true;

# Wöchentliche Garbage Collection für Generationen älter als 14 Tage
nix.gc = {
  automatic = true;
  dates = "weekly";
  options = "--delete-older-than 14d";
};
```

Debugging mit `nix repl` und Systemdiagnose

Wenn unklar ist, warum ein bestimmtes Paket nicht gelöscht werden kann oder welche Konfigurationsoptionen existieren, helfen die integrierten Diagnosebefehle:

```
# Interaktive Evaluierungsumgebung der Nix-Sprache öffnen
nix repl --expr 'import <nixpkgs> {}'

# Prüfen, warum ein Paket im System verbleibt (Abhängigkeitskette aufdecken)
nix why-depends /run/current-system nixpkgs#openssl

# Konsistenz und SHA-256-Signaturen des gesamten Stores verifizieren und reparieren
sudo nix-store --verify --check-contents --repair
```

Befehlsreferenz (Cheatsheet)

Befehl / Aufruf	Kontext	Zweck / Funktion im Sysadmin-Alltag
`nix shell nixpkgs#paket`	Temporär	Startet eine Subshell mit bereitgestelltem Werkzeug ohne Host-Installation
`nix run nixpkgs#paket -- [args]`	Temporär	Führt eine Binärdatei direkt aus dem Cache aus, ohne sie persistent zu halten
`nix develop`	Projekt	Lädt die in `flake.nix` definierte Entwicklungs- und Toolchain-Umgebung
`nix profile list`	Benutzer	Zeigt alle im aktiven Benutzerprofil installierten Softwarepakete an
`nix profile install nixpkgs#paket`	Benutzer	Installiert ein Paket isoliert im Profil des angemeldeten Benutzers
`nix profile history`	Benutzer	Listet alle bisherigen Profilgenerationen chronologisch auf
`nix profile rollback`	Benutzer	Setzt das Benutzerprofil atomar auf die vorherige Generation zurück
`sudo nixos-rebuild switch`	NixOS	Evaluiert `/etc/nixos/configuration.nix`, baut das System und aktiviert es sofort
`sudo nixos-rebuild switch --rollback`	NixOS	Schaltet das gesamte Betriebssystem unmittelbar auf die letzte Generation zurück
`sudo nixos-rebuild boot`	NixOS	Baut das System für den nächsten Neustart, ohne laufende Dienste zu stören
`nix flake update`	Flakes	Aktualisiert alle Abhängigkeiten in `flake.lock` auf den neuesten Upstream-Commit
`nix-collect-garbage -d`	Wartung	Löscht alte Generationen und entfernt ungenutzte Store-Pfade vollständig
`nix store optimise`	Wartung	Dedupliziert identische Dateien im `/nix/store` über Hardlinks

Befehl / Aufruf	Kontext	Zweck / Funktion im Sysadmin-Alltag
`nix why-depends /run/current-system <paket>`	Diagnose	Zeigt den exakten Abhängigkeitspfad, warum ein Paket im Store verbleibt
`sudo nix-store --verify --check-contents`	Diagnose	Prüft sämtliche Store-Dateien auf Bit-Fehler gegen ihre Soll-Prüfsummen

Weiterführende Ressourcen

Ressource / Dokumentation	Typ	Beschreibung / Einsatzzweck
Offizielle NixOS Dokumentation	Handbuch	Vollständiges Referenzhandbuch für Systemadministration und Konfiguration
Nixpkgs Manual	Entwickler-Doku	Detaillierte Richtlinien für Paketierung, Overlays und Build-Helfer
NixOS Paket- & Optionssuche	Suchportal	Zentrale Websuche nach Paketen und allen verfügbaren `configuration.nix`-Optionen
Nix Pills Leitfaden	Tutorial	Tiefgehender Kurs in die funktionale Programmiersprache und Derivations
Nix Flakes Referenz	Konzept-Doku	Offizielle Dokumentation zur modernen Flake-Architektur und Lockfiles
NixOS Community Discourse	Forum	Offizielle Diskussions- und Support-Plattform der weltweiten Nix-Community

Fazit

Nix und NixOS erfordern von Administratoren ein grundlegendes Umdenken: Weg von manuellen, zustandsbehafteten Eingriffen auf dem Server, hin zu einem rein deklarativen, funktionalen Infrastrukturmodell.

Quelle: https://admindocs.de/de/linux-neulinge/nixos-der-leitfaden-vom-paketmanager-nix

Wer die anfängliche Lernkurve meistert, wird mit einer Plattform belohnt, die eine ganze Klasse klassischer Administrationsprobleme vollständig eliminiert. Updates verlieren ihr Risiko durch verlässliche, atomare Rollbacks; Entwicklungsumgebungen lassen sich bitgenau reproduzieren; und Server-Konfigurationen lassen sich lückenlos über Versionskontrollsysteme wie Git verwalten.

> **Praxis-Tipp für deinen Einstieg:** Beginne mit Nix als Standalone-Paketmanager auf deiner gewohnten Linux-Distribution, um temporäre Umgebungen mit `nix shell` und ersten Flakes zu testen. Sobald die Konzepte von Derivations und Store-Hashes verinnerlicht sind, ist der Schritt zu einem voll deklarativen NixOS-Server nahtlos und sicher.

Wenn du nach dem deklarativen Paketmanagement den nächsten logischen Schritt zur vollständigen Infrastruktur-Automatisierung gehen möchtest, zeigt dir der Leitfaden zu Ansible: Grundlagen der Automatisierung für Linux-Administratoren, wie du Konfigurationen über heterogene Serverlandschaften hinweg idempotent ausrollst.

Quelle: https://admindocs.de/de/linux-neulinge/nixos-der-leitfaden-vom-paketmanager-nix