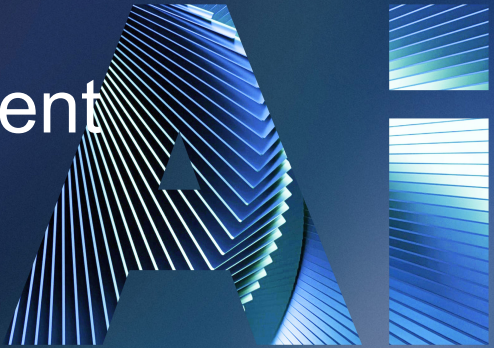
Oct. 29
Industrial Open Source

Oct. 30
Open Source Robotics

Open Source 西门子

Beijing · 2026

Nix Builds for Parallel Development



Build Once, Reuse Everywhere
29 October 2026

Michael Adler | Senior Key Expert | FT RPD CPC OES-DE | Siemens AG

SIEMENS

Motivation: Why Not 10x Faster?

AI makes coding 10x faster. Why doesn't the team ship 10x faster?

Theorem (Amdahl's Law). *Overall speedup is limited by the fraction of work that remains unchanged:*

$$Speedup = \frac{1}{(1-p) + \frac{p}{s}}$$

p = fraction accelerated, s = speedup on that fraction



$s = 10$ (illustrative)

- $p = 30\%$ (coding): **1.37x**
- $p = 50\%$ (coding): **1.8x**

Everything after the code is unchanged

- Review queues grow
- CI and builds take the same wall time

Build and CI time is the constraint

Speeding up the code generation is one part. Next: speed up the build.

SIEMENS

Traditional Solutions

CI-provider caches

Provider-specific APIs and cache keys; switching CI providers can mean reworking the setup. Might not work locally across all systems.

Compiler caches

`ccache`, `sccache`: cache compiler outputs; `sccache` supports remote storage. Neither caches toolchains or system libraries.

Build-tool caches

Reuse task outputs, locally or remotely; each tool is limited to its own ecosystem.

Hermetic build systems

Bazel, Buck2: cache everything, but adoption means rewriting the build; third-party dependencies are painful.

Siloed or Costly

Wanted: *one* cache and build-offload model for *all* dependencies; wrap existing builds instead of rewriting them.

SIEMENS

Introducing Nix: The universal, composable build system

Hermetic

Builds run in a sandbox without network access; only declared inputs are visible.

Input-Addressed Builds

Output paths derive from the derivation and its inputs; the store path *is* the cache key (trivial invalidation).

Rich Ecosystem

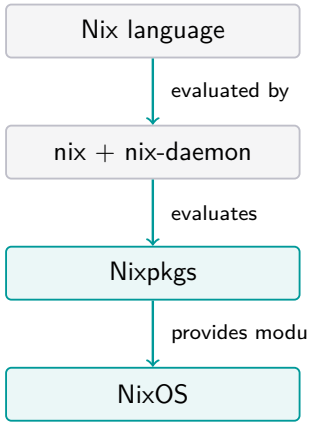
~150,000 packages; existing builds are wrapped, not rewritten.

Reuse instead of rebuild

Nix either fetches matching store paths from a binary cache

SIEMENS

Nix Language, Tool, and Ecosystem



Lazy, purely functional language.

Evaluates expressions into derivations (build recipes: inputs, builder, env $\rightarrow$ output paths); manages the store and runs builds.

Collection of package expressions and build functions.

Linux distribution configured with Nix modules.

SIEMENS

Nix Language: A Quick Primer

The Nix language consists of expressions that evaluate to values:

```
let
  add = a: b: a + b;
  libs = [ "openssl" "zlib" ];
  src = ./.; # caveat: copies . to store
in {
  inherit src; # src = src;
  sum = add 1 2;
  names = map (l: "lib-" + l) libs;
}
# => { src = /path/here; sum = 3;
#      names = [ "lib-openssl"
#                "lib-zlib" ]; }
```

- `let` binds names; expression after `in` is the result
- `a: b: expr` defines a function of two arguments; `add 1 2` applies it
- `[ ... ]` is a space-separated list; `map` applies a function to each item
- `./.` is a path literal, not a string; copied to the store when coerced to a string (e.g., in a derivation)
- A `{ ... }` is an attribute set, the main data structure of Nix

SIEMENS

Nix Language: Attribute Sets and Merging

```
let
  common = { pname = "app"; version = "1"; };
  mkPkg = { pname, version, ... }: {
    inherit pname version;
    description = "${pname} ${version}";
  };
in mkPkg (common // { version = "2"; })
# => { pname = "app"; version = "2";
#      description = "app 2"; }
```

```
{ hello = "ok"; huge = throw "no"; }.hello
# => "ok"
```

- `x = { a = 1; }` is a set; `x.a` selects
- `//` merges sets (shallow); the right side wins
- `{ a, b, ... }: ...` takes a set and destructures it; `...` ignores extra attributes
- `inherit a;` means `a = a;`
- `"${a}"` is string interpolation
- Evaluation is lazy: `huge` is never evaluated, so no error

SIEMENS

Nix Packages: It's All About Functions

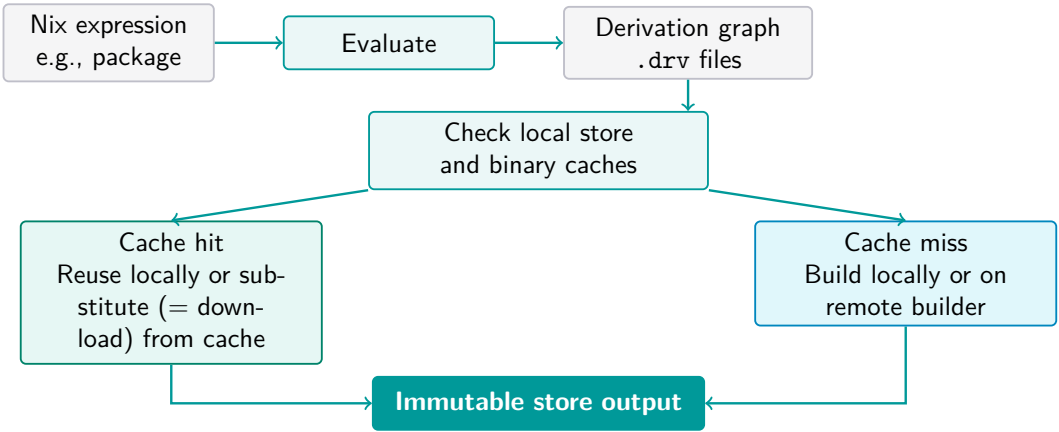
A Nix package is a function, with the arguments being its dependencies.

```
{ craneLib, pkg-config, openssl }: # deps are Nix expressions

let
  commonArgs = {
    src = craneLib.cleanCargoSource ./.;  # Filter source to Cargo-relevant files
    nativeBuildInputs = [ pkg-config ];   # Build-platform tools
    buildInputs = [ openssl ];            # Host-platform deps (e.g., libraries)
  };
  cargoArtifacts = craneLib.buildDepsOnly commonArgs; # deps only (release)
  cargoDebugArtifacts = craneLib.buildDepsOnly (commonArgs // {
    env.CARGO_PROFILE = "dev"; # debug deps to pre-populate devShell target/
  });
in
craneLib.buildPackage (commonArgs // {
  inherit cargoArtifacts;                      # only workspace code built here
  passthru = { inherit cargoDebugArtifacts; }; # expose for CI cache + devShell
})
```


SIEMENS

Evaluation: From Nix Expression to Artifact



SIEMENS

Flakes: It's All About Composition

Nix *flakes* pin inputs and define packages and dev shells:

```
{ inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-unstable";
  inputs.crane.url = "github:ipetkov/crane";
  outputs = { nixpkgs, crane, ... }: let
      system = "x86_64-linux";
      pkgs = import nixpkgs { inherit system; };
      craneLib = crane.mkLib pkgs;
      my-app = pkgs.callPackage ./package.nix { inherit craneLib; };
    in {
      packages.${system} = {
        inherit my-app;
        deps-debug = my-app.cargoDebugArtifacts; # for CI to populate cache
      };

      devShells.${system}.default = craneLib.devShell { inputsFrom = [ my-app ]; };
    };
}
```


SIEMENS

DX: Two Feedback Loops

Inner loop

- Enter dev shell: `nix develop .`
- Run tools directly: `cargo check`, `cargo test`
- Pre-populate with CI-built dependencies

Outer loop

- Verify build: `nix build .#my-app`
- Isolated derivation catches undeclared inputs

SIEMENS

Inner Loop: Pre-populate Rust `target/` from a Binary Cache

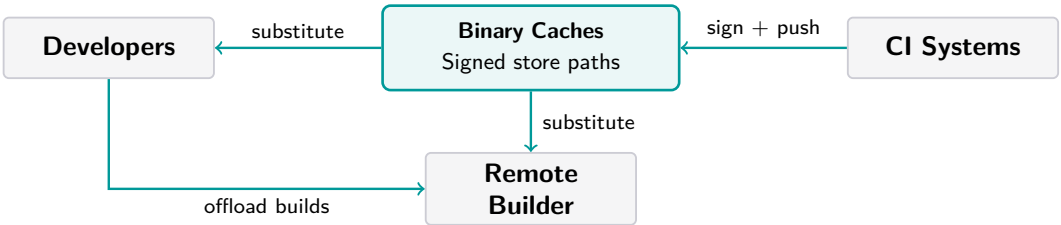
Build all dependencies *once* (e.g., in CI) and reuse them for local builds:

```
let
  deps = my-app.cargoDebugArtifacts;
  prepopulate = ''
  STAMP="target/.deps-stamp"
  if [ "$(cat "$STAMP" 2>/dev/null)" \
       != "${deps}" ]; then
    mkdir -p target
    tar -I zstd -C target -xf \
      ${deps}/target.tar.zst
    chmod -R u+w target
    echo -n "${deps}" > "$STAMP"
  fi
  # + point CARGO_HOME at vendored sources
'';
in
devShells.${system}.default = craneLib.devShell {
  inputsFrom = [ my-app ];
  packages = [ pkgs.zstd ];
  shellHook = prepopulate;
};
```

- `nix develop` substitutes `cargoDebugArtifacts` (published via `.#deps-debug`) and unpacks it into `target/`
- `cargo build` compiles workspace crates
- Delete `target/` freely, deps stay cached in `/nix/store`
- Needs identical `rustc`/flags and vendored store paths, else full rebuild
- For per-crate cache reuse, use `crate2nix`

SIEMENS

Caching Strategy



Binary Caches

- **Upstream:** `cache.nixos.org` (read-only, S3-backed)
- **Mirrors:** e.g., USTC, TUNA, SJTU (read-only)
- **Self-hosted:** Attic, Harmonia
- **SaaS:** Cachix, FlakeHub Cache

Key Mechanics

- **Trust:** artifacts are cryptographically signed
- **Builders:** access via SSH
- **CI savings:** cache hits cut build time and CI minutes
- **Caching:** optimization, not hard dependency

SIEMENS

Measured Results

Scenario	Plain Cargo	Warm sccache	Nix, pre-populated
Fresh target, wall time	7.59 s	5.06 s	**2.36 s**
Fresh target, CPU time	42.13 s	20.28 s	**4.69 s**
4 parallel builds, last finish	18.57 s	9.67 s	**4.71 s**
4 parallel builds, total CPU	249.85 s	116.76 s	**28.87 s**
4 parallel builds, peak RSS	4,045 MiB	2,787 MiB	**1,780 MiB**
Incremental leaf, wall time	0.51 s	0.52 s	0.51 s
Incremental leaf, CPU time	0.82 s	0.67 s	0.72 s

Takeaway

- Fresh-build CPU: **9**× faster
- 4 parallel builds, CPU: **8.7**× faster
- Incremental rebuilds: comparable

Amdahl: end-to-end gain

Speedup on 4 parallel builds: $s = 18.57/4.71 \approx 3.9$

- Builds+CI wait $p = 20\%$ of cycle: **1.18x**
- $p = 40\%$: **1.42x**

Measure your own p.

Five-run medians, 16 CPUs, 32 GiB RAM. Small one-crate local check, no remote cache or builder; not a scale forecast.

SIEMENS

Conclusion

One pinned flake defines packages and dev shells; binary cache and remote builders reuse its outputs.

Benefits

- Same declared dependencies in development, agents, and CI
- Matching outputs shared across machines instead of rebuilt
- Builds can be trivially offloaded to remote builders

Costs

- Steep learning curve (though LLMs greatly lower it)
- Coarse invalidation: any `Cargo.lock` change rebuilds all of `cargoDebugArtifacts`
- `nix build` is not incremental: a late failure reruns the derivation
- Pre-populating build directories is per-language work; Nix eval adds overhead

SIEMENS

Thanks

Questions?

Michael Adler

michael.adler@siemens.com

SIEMENS

Backup: Sandboxed Agent Execution via Nix

The same flake can ship a jailed agent (example only, a small third-party repo, not a recommendation):

```
inputs.jailed-agents.url = "github:andersonjoseph/jailed-agents";
# ...
devShells.${system}.default = craneLib.devShell {
  inputsFrom = [ my-app ];
  shellHook = prepopulate;
  packages = [ pkgs.zstd (jailed-agents.lib.${system}.makeJailedOpencode {
    # Crane's toolchain, so prepopulated target/ stays valid
    extraPkgs = [ pkgs.nodejs pkgs.python3 craneLib.cargo craneLib.rustc ];
  }) ];
};
```

- **Jailed execution:** Wrapper restricts agent access to the declared tools and the worktree
- Isolation depends on the jail configuration; it is not a guarantee against host contamination

SIEMENS

Backup: Beyond Package Builds

Systems and Provisioning

- **NixOS:** Declarative Linux system configuration
- **home-manager:** Declarative user and home configuration
- **disko + nixos-anywhere:** Define disk layouts; install NixOS remotely

Development and Delivery

- **Container images:** Build OCI/Docker images from Nix
- **agenix:** Manage age-encrypted secrets for NixOS systems
- **devenv:** Reproducible project development environments

SIEMENS

Oct. 29
Industrial Open Source

Oct. 30
Open Source Robotics

Open Source 西门子

Beijing · 2026