Learning Nix with LLMs

Unfamiliar Languages & DSLs — using LLMs as a Teacher

Maxime · Meetup Talk · 30 min

Why This Talk?

- The more proficient we get with LLMs, the better our outcomes
- Tooling evolves fast — yesterday’s copy-paste into ChatGPT, today’s agentic Claude Code and harnesses like OpenClaw
- Most guides are **output-oriented** — but what about our own **competency development**?

So What Does This Have to Do with Nix?

Nix is the **perfect example** for LLM-assisted learning because it's *foreign*:

- A **DSL** based on a dynamically typed, functional language
- A **package manager** that replaces imperative software distribution with atomic, reversible declarations
- It does almost everything **differently** from mainstream tools
- Unlike Docker — runs **natively** on Linux, FreeBSD, and macOS (aka Darwin)

> Learn something a different way — and enjoy the result too.

Level 0 — What is Nix?

Nix: A Package Manager with a Twist

Like `apt` or `Homebrew` — but every package gets its **own isolated folder** with a unique hash.

- **No version conflicts** — Python 3.9 and 3.12 side by side
- **Reproducible setups** — same config file → same environment, every time
- **Easy rollbacks** — update broke something? Flip back instantly

"Nix" also refers to:

- **The Nix language** — a small functional language for config files
- **NixOS** — an entire Linux distribution built on the same idea

Code → Derivation → Package

A **derivation** is a compiled build recipe with no conditional logic.

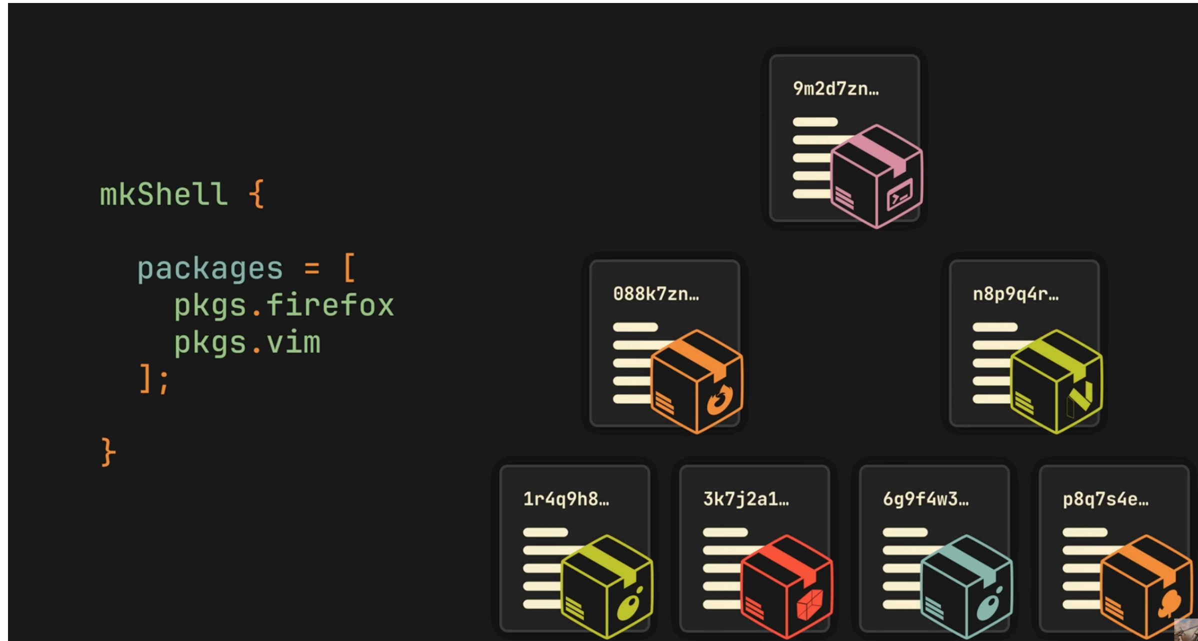
It only knows: **(1)** what it builds, **(2)** how to build it, **(3)** which other derivations it depends on.

Like a pure function — same inputs always produce the same output.

The Pipeline

Nix Code → Derivation → Package

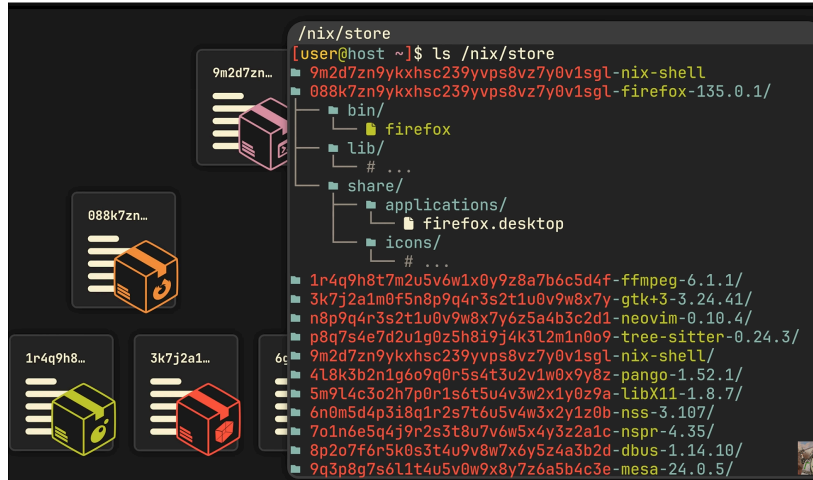
Derivations Build on Derivations



Code gets translated into derivations. Those are converted to packages — or become **inputs for other derivations**.

The Nix Store

Every package lives in its own directory named after a **hash of its derivation** + **name** + **version**.



- Same hash? **Reuse** — no rebuild, no duplicate disk usage
- Nix checks a **remote cache** before compiling locally
- Source-based package manager that behaves like a **binary** one

Writing Derivations — The Primitive Way

The built-in `derivation` function — rarely used in practice:

```
derivation {
  name = "name";
  builder = "/bin/sh";
  args = ["-c" "echo 'hello world' > $out"];
  system = "x86_64-linux";
}
```

Writing Derivations — mkDerivation & Friends

`stdenv.mkDerivation` from nixpkgs adds: `src`, `buildInputs`, `nativeBuildInputs`, and build phases.

```
stdenv.mkDerivation {
  name = "myPackage";
  src = ./path/to/source;
  buildInputs = [
    pkgs.ffmpeg
  ];
  nativeBuildInputs = [
    pkgs.pkg-config
  ];
  unpackPhase = '' # shell logic '';
  buildPhase = '' # shell logic '';
  installPhase = '' # shell logic '';
  # ...
}
```

mkDerivation
buildPythonPackage
buildMavenPackage
derivation
buildNpmPackage
buildRustPackage

Specialized builders like `buildRustPackage`, `buildNpmPackage` extend it further.

LLM-Assisted Learning Techniques

Technique 1 — Translate

Use the LLM to **translate mental models** you already have.

Go developer? Translate to pseudo-Go. Python? Use Python. The goal is **structural understanding**, not runnable code.

```
# Prompt (excerpt)
Translate the Nix expression into a structural sketch in pseudo-code.
- No runnable code; only a structural sketch
- Mark Nix-specific constructs with [Nix-specific: ...]
- Mark special features
```

We create prompts that **don’t solve a problem** — we use them to **explore structure**.

Technique 2 — Evaluation Tracing

My challenge: I understood data structures and functions, but I still **couldn't reason about Nix programs**. Why? Everything is an expression — no assignments, no imperative flow.

The first symptom: reading code and having **no idea what happens in which order**.

Ask the LLM to **trace** nix programs step by step.

```
# Prompt (excerpt)
Trace the Nix expression in evaluation order.
- Separate [EVAL] and [BUILD] phases
- For each step: phase marker, trigger, what's computed, why
- Maintain a running thunk list — strike through when forced
```

This makes the **invisible flow visible** — especially the eval/build boundary.

Technique 3 — Socratic Debugging

For stuff I really want **"brain native"** — where I want to convince my brain to *memorize* it, not just understand it.

The idea: force the LLM **not to provide answers** — only questions. Improve recollection, not just comprehension.

```
# Prompt (excerpt)
You are a Nix tutor. Your job is NOT to explain or solve.

Ask exactly three questions:
1. Surface — What is concretely happening?
2. Mechanism — How does Nix produce this behavior?
3. Edge — What would happen if one part changed?

If you solve it, you have failed.
```

Variations: "quiz me hard" — feels like being back at university. But it **works**.

Technique 4 — Three Variations

Learn by **comparing solutions** across competency levels.

```
# Prompt (excerpt)
Show three ways to solve a problem in Nix:

1. Naive – after learning Nix for one week
2. Idiomatic – two years of experience
3. Advanced – module system knowledge

For each: code block (max 5 lines), why, trade-off.
```

Creates a **hierarchy of approaches** — see where you are, where you're going.

Why Nix is Formidable for Agentic Workers

Nix × Agents

- **Declarative** — the agent describes the target state, not a choreographed sequence. Robust against hallucinations.
- **Pure & reproducible** — what works for the agent works for you. No "works on my machine."
- **Statically evaluable** — `nix flake check`, not just `go build`. Dry-run without side effects.
- **Atomic rollbacks** — mistakes cost nothing. Roll back one generation, done.
- **System as text** — your entire machine state fits into a single context window.

Thank You

Learn something a different way — and enjoy the result too.

Maxime

https://mxmlabs.de/