THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
KOREA

One Binary, Every Package Manager

Shipping a Rust CLI to PyPI, npm, Homebrew, Winget, and Beyond

Dr. Ajit Kumar
Senior Software Architect, Wellmatix
Seoul, South Korea
August 11, 2026

Who Am I?

Dr. Ajit Kumar

@urwithajit9

LinkedIn

Groups & Meetups

- **Dev-Korea:** dev-korea.com
  (by Florian Ludot)
- **Seoul Rust:** seoul.rs
  (by Ian Wagner)

Background & Experience

- **Ph.D.:** Pondicherry University, Pondicherry, India.
- **Postdoc:** Soongsil University, Seoul, South Korea.
- **Past:** 3 Years as Lead Developer & R&D Head, Seoul, South Korea
- **Currently:** Senior Software Architect @ Wellmatix, Seoul, South Korea

Tech Stack & Interests

- **Languages:** Python, Rust, Web Technologies
- **Interests:** Running, Hiking, Open Source

Agenda

- The .env and evnx story
- Rust Development and Distribution
- The distribution problem: one binary, many front doors
- PyPI, npm, Homebrew, and Winget – deep dives
- Lessons learned and results

The .env and evnx story

What Is a `.env` File?

- A plain-text file storing key–value pairs as environment variables, e.g. `DATABASE_URL=postgres://...`
- Used by developers on virtually any stack – Python, JavaScript/TypeScript, backend engineers, DevOps, and beyond
- Read by countless frameworks and tools (Django, Next.js, Rails, Docker Compose, CI pipelines) to configure an app without hardcoding values
- Keeps secrets, API keys, and per-environment config out of source code – and usually out of version control via `.gitignore`

Read more at: `https://dotenv.space/`

How `evnx` Came to Be

- Started as a hobby project – a small tool to make `.env` files easier to live with
- The name was supposed to be `envx` – already taken, so the letters got swapped to `evnx`
- Built to help manage `.env` alongside `.env.example`, and the sprawl of variants teams accumulate: `.env.local`, `.env.dev`, `.env.test`, and more
- Goal: keep every variant in sync, catch missing keys early, and make the “which `.env` do I need?” question go away

Read more at: `https://www.evnx.dev/blog/why-i-built-evnx`

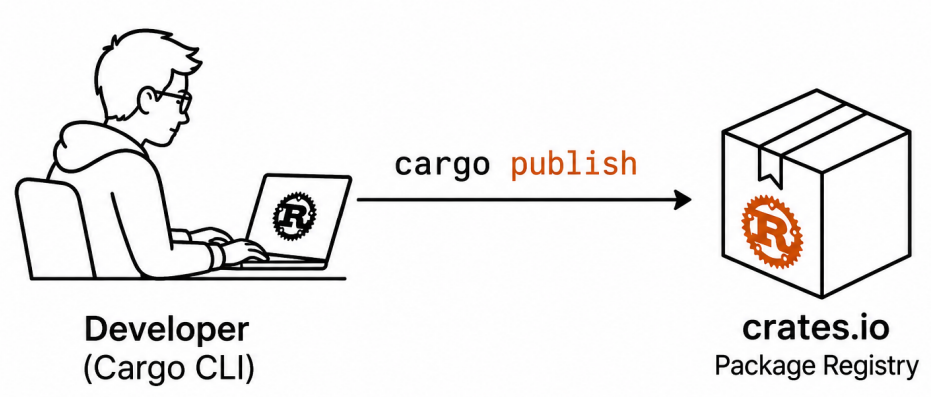
Why `evnx` Is Written in Rust

- A CLI that touches secrets on every run – Rust's memory safety rules out a whole class of bugs by construction, no garbage collector needed
- Ships as a single small, static binary – fast startup, easy to drop into any dev environment or CI image with no runtime to install
- `cargo` makes the dev loop easy: build, test, and lint with one toolchain, no dependency wrangling
- Personal fit as much as technical merit – after Python, Rust is the language, I know best, so it was the natural choice for a hobby project

Rust Development & Distribution

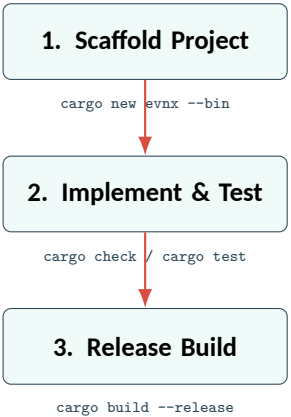
Rust Cross-Platform Distribution Pipeline

evnx at crates.io: `https://crates.io/crates/evnx`



Rust CLI Lifecycle (1/4) – Development & Testing

Local Development Cycle



Core Commands

- **Scaffold a new binary crate**
  `cargo new evnx --bin`
- **Fast feedback loop**
  `cargo check`
  type-checks without full codegen – much faster than a build
- **Run the test suite**
  `cargo test`
- **Build an optimized binary**
  `cargo build --release`

Rust CLI Lifecycle (2/4) – Metadata & Verification

`Cargo.toml` – Source of Truth

```
[package]
name = "evnx"
version = "0.3.8"
edition = "2021"
license = "MIT"
description = "Secure .env CLI"
repository = "github.com/.../evnx"
keywords = ["cli","dotenv","secrets"]
categories = ["command-line-utilities"]
```

`keywords` and `categories` directly drive discoverability in crates.io search.

Pre-Flight Checklist

- **Inspect the package**
  `cargo package`
  builds the `.crate` archive so you can check exactly what ships
- **Rehearse the release**
  `cargo publish --dry-run`
  catches missing files or bad metadata before a version number is burned

Rust CLI Lifecycle (3/4) – Publishing to crates.io

Publish

Execute the release

```
cargo login
cargo publish
```

- **Downloads:** 238 (as of 9 Aug 2026)

Publishes are permanent

A version on crates.io can never be overwritten or deleted – only `yanked` (hidden from new dependents). Any fix means bumping `version` in `Cargo.toml`.

Nice to have

`cargo publish --provenance` attaches a build attestation (SLSA) linking the crate back to the exact GitHub Actions run that built it.

Rust CLI Lifecycle (4/4) – Installing the CLI

Option 1: Via Cargo (crates.io)

Requires Rust toolchain

```
cargo install <your-crate-name>
```

Option 2: Pre-built Binary (Standalone Script)

Direct download (No Rust required)

```
curl -fsSL https://dotenv.space/install.sh | bash
```

or

```
curl -sSL
https://raw.githubusercontent.com/urwithajit9/evnx/main/scripts/install.sh
| bash
```

The Distribution Problem

Why One Binary Isn’t Enough

- Rust gives us a single, static, dependency-free binary
- …but every ecosystem expects its own packaging format
- Users install with the tool they already trust: `pip install`, `npm install`, `brew install`, `winget install`
- Each registry has its own manifest schema, review process, and update cadence
- Goal: one CI pipeline, five-plus package managers, zero manual releases

The Packaging Landscape Today

Where our users already are

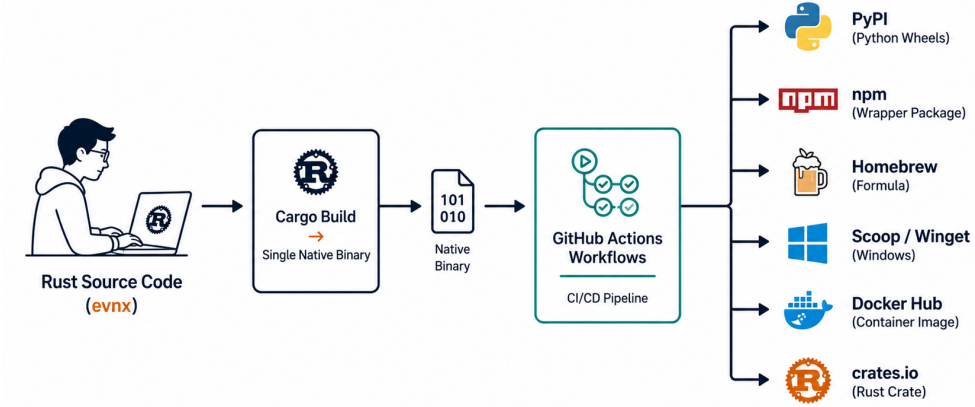
- Python data teams → PyPI
- Frontend/Node teams → npm
- macOS/Linux developers → Homebrew
- Windows developers → Winget

The constraint

- No registry runs Rust's toolchain for us
- Every registry only distributes *its own* artifact types
- So: build native binaries, then wrap them per ecosystem

Build Once, Ship Everywhere

One Binary to All Channels



Cross-Compilation with a Target Matrix

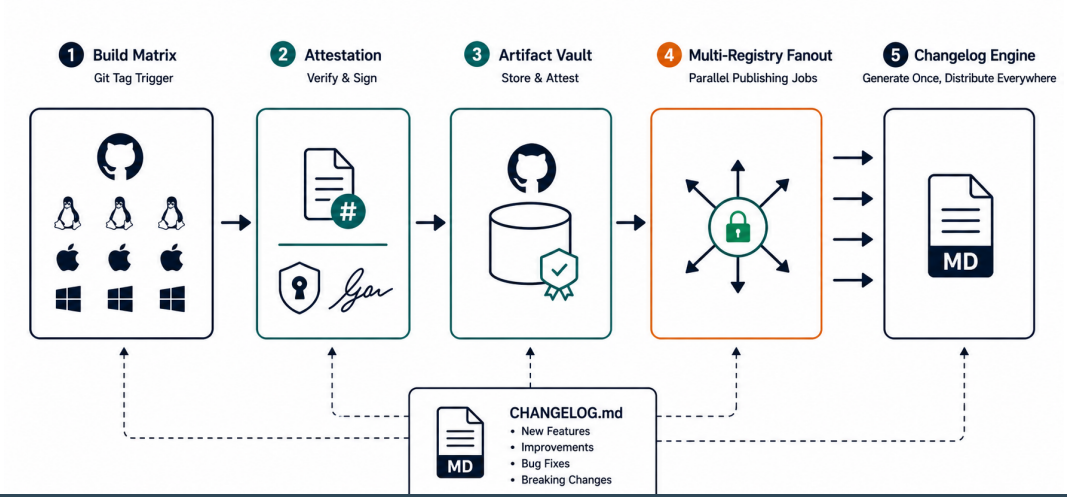
Build once, per target triple

`cargo build --release` via `cross`, one job per triple

```
cargo build --release --all-features --target \${{ matrix.target }}
```

- `x86_64-unknown-linux-gnu`, `aarch64-unknown-linux-gnu`
- `x86_64-apple-darwin`, `aarch64-apple-darwin`
- `x86_64-pc-windows-msvc`, `aarch64-pc-windows-msvc`
- `cross` runs each build in a pinned Docker image – no host toolchain drift
- There were some security package issues with some target

Release Pipeline Steps



Release Pipeline at a Glance

1. Tag a release → GitHub Actions matrix builds all targets
2. Checksums (`sha256sum`) and Sigstore/`cosign` signatures per artifact
3. Artifacts uploaded to a GitHub Release with SLSA provenance attached
4. Publisher jobs push wrappers to each registry, in parallel, gated on the release job
5. A single `CHANGELOG.md` feeds every release note, every registry

Wrapper Strategy by Ecosystem

Wrapper Strategy by Ecosystem

Script-based registries

- **PyPI** – thin `sdist`/wheel that fetches the binary on install
- **npm** – `postinstall` script fetches the matching artifact
- **Homebrew** – formula pins a checksum per platform

Native package formats

- **Winget** – manifest YAML, validated by `winget-pkgs` CI
- **Cargo** – publish the crate directly, no wrapper needed
- **Scoop / AUR** – community-maintained manifests we help review

Deep Dive: PyPI CLI Distribution for `evnx`

Distributing `evnx` via PyPI

- Package the pre-compiled Rust binary directly inside target-specific Python wheels.
- Powered by `maturin` for native Rust-to-PyPI packaging.
- Zero runtime dependencies or network downloads during installation.
- Smooth integration with modern global Python CLI workflows via `pipx`.

1. Project Configuration (Cargo.toml & pyproject.toml)

Cargo.toml

```
[package]
name = "evnx"
version = "0.1.0"
edition = "2021"gi

[[bin]]
name = "evnx"
path = "src/main.rs"
```

pyproject.toml

```
[build-system]
requires = ["maturin>=1.0,<2.0"]
build-backend = "maturin"

[project]
name = "evnx"
version = "0.1.0"
description = "Environment variable
    manager"
```

2. Building Platform-Tagged Wheels

Compile Wheels via Maturin

```
# Build wheel for Linux x86_64
$ maturin build --release --target x86_64-unknown-linux-gnu

# Build wheel for macOS Apple Silicon
$ maturin build --release --target aarch64-apple-darwin

# Build wheel for Windows x86_64
$ maturin build --release --target x86_64-pc-windows-msvc
```

More Details: `https://github.com/urwithajit9/evnx/blob/main/.github/workflows/python-publish.yml`

Generated Artifacts

Output wheels placed in `target/wheels/`:

- `evnx-0.1.0-py3-none-manylinux_2_17_x86_64.whl`
- `evnx-0.1.0-py3-none-macosx_11_0_arm64.whl`

3. Target Matrix & `pipx` Global Install

Upload to PyPI

Publish all platform wheels under the single `evnx` release:

```
$ maturin upload target/wheels/*
```

PEP 668 System Protection

Modern OS distros block raw `pip install` outside a virtual environment.

Global Installation via `pipx`

```
# Modern global installation
$ pipx install evnx

# Ready in PATH
$ evnx --version
evnx 0.1.0
```

Isolates `evnx` in its own venv while exposing the binary globally in `~/.local/bin`.

Deep Dive: npm Distribution for `evnx`

Distributing `evnx` via npm

- Uses the native package pattern popularized by `esbuild` and `swc`.
- Root `evnx` package lists target-specific platform packages as `optionalDependencies`.
- npm automatically resolves and installs **only** the package matching the host OS/architecture.
- Zero network calls during install — works offline and in strict CI environments.

1. Package Structure & Platform Targeting

Details: https://github.com/urwithajit9/evnx/blob/main/.github/workflows/npm-publish.yml

Platform Package (@evnx/linux-x64)

```
{
  "name": "@evnx/linux-x64",
  "version": "0.1.0",
  "os": ["linux"],
  "cpu": ["x64"],
  "bin": { "evnx": "bin/evnx" }
}
```

Published once per target triple (e.g. `darwin-arm64`, `win32-x64`).

Root Package (evnx)

```
{
  "name": "evnx",
  "version": "0.1.0",
  "bin": { "evnx": "bin/evnx" },
  "optionalDependencies": {
    "@evnx/linux-x64": "0.1.0",
    "@evnx/darwin-arm64": "0.1.0",
    "@evnx/win32-x64": "0.1.0"
  }
}
```

npm skips incompatible platforms without failing installation.

2. Binary Resolver (`bin/evnx`)

JS Launcher Shell

```
#!/usr/bin/env node
const { spawnSync } = require('child_process');
const os = require('os');

const pkgMap = {
  'linux-x64': '@evnx/linux-x64',
  'darwin-arm64': '@evnx/darwin-arm64',
  'win32-x64': '@evnx/win32-x64'
};

const platformKey = `${os.platform()}-${os.arch()}`;
const binPath = require.resolve(`${pkgMap[platformKey]}/bin/evnx`);

const result = spawnSync(binPath, process.argv.slice(2), { stdio: 'inherit' });
process.exit(result.status ?? 0);
```

3. Installation & Execution

Global Install via npm

```
# Global install
$ npm install -g evnx

# Execute from PATH
$ evnx --version
evnx 0.1.0
```

Zero-Install via npx

```
# Run instantly via npx
$ npx evnx init
```

Key Benefits

- Fast local/CI caching.
- No post-install compilation step.
- No curl-to-bash security concerns.

Deep Dive: Homebrew Distribution for `evnx`

Distributing `evnx` via Homebrew

- Start with a **custom tap** (`https://github.com/urwithajit9/homebrew-evnx`) for immediate zero-approval releases.
- Formula written in Ruby downloads pre-compiled release binaries and verifies their `sha256` checksum. `https://github.com/urwithajit9/homebrew-evnx/blob/main/Formula/evnx.rb`
- Automated CI pipeline updates the formula on every new GitHub tag.
- **Path to Official Core:** As star count, downloads, and stability grow, submit directly to `homebrew/core` for single-command installation.

1. Writing the Formula (`evnx.rb`)

`Formula/evnx.rb` in Custom Tap Repository

```
class Evnx < Formula
  desc "Environment variable management CLI"
  homepage "https://github.com/urwithajit9/evnx"
  url "https://github.com/urwithajit9/evnx/releases/download/v0.1.0/evnx-x86_64-
      apple-darwin.tar.gz"
  sha256 "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
  version "0.1.0"

  def install
    bin.install "evnx"
  end

  test do
    assert_match "evnx 0.1.0", shell_output("#{bin}/evnx --version")
  end
end
```

2. Automated Formula Updates via GitHub Actions

CI Automation Pipeline

- Triggered on GitHub release publish.
- Computes SHA-256 hashes for all release artifacts.
- Bumps version and checksum in tap repository via PR/commit.

Automated PR Bump

```
# Update formula automatically in CI
$ brew bump-formula-pr \
    --url=https://github.com/
        urwithajit9/evnx/releases/
        download/v0.2.0/evnx.tar.gz \
    --sha256=<new-sha256> \
    urwithajit9/tap/evnx
```

3. Installation Path to `homebrew-core`

Phase 1: External Custom Tap

```
# Add tap and install
$ brew tap urwithajit9/evnx
$ brew install evnx

# Or direct single line install
$ brew install urwithajit9/evnx/evnx
```

Instant release cycle; full control over formula updates.

Phase 2: Submission to `homebrew-core`

```
# Installed directly without external
    tap
$ brew install evnx
```

Core Acceptance Criteria

Requires notable public popularity (50+ stars/forks, consistent usage), stable tag history, and building from source or official binary criteria.

Deep Dive: Windows Package Manager (WinGet) for `evnx`

Distributing `evnx` via WinGet

- Native Windows CLI installation via `winget install evnx`.
- Uses three declarative YAML manifests per version: version, installer, and locale.
- Submitted via Pull Request (PR) to the community repository `microsoft/winget-pkgs`.
- **Strict Validation:** Every PR triggers Microsoft automated pipelines (SmartScreen, static code analysis, and multi-engine malware scanning).
- High-priority focus needed on handling malware scan false positives and code signing.

1. Manifest Structure & Automation

Manifest Triad (`manifests/e/evnx/evnx/0.1.0/`)

- `evnx.yaml` (Root Version Manifest)
- `evnx.installer.yaml` (Binary URL, SHA-256, Architecture)
- `evnx.locale.en-US.yaml` (Metadata, License, Description)

Points to GitHub Releases pre-built `.exe` or `.msi`.

Automated Manifest Generation

```
# Use wingetcreate to auto-generate &
    submit
$ wingetcreate new \
    https://github.com/urwithajit9/evnx
        /releases/download/v0.1.0/evnx-
        x86_64-pc-windows-msvc.zip

# Validates schema & creates PR to
    winget-pkgs
```

`wingetcreate` downloads the zip/exe, computes SHA-256, and opens the PR.

2. WinGet PR Automated Pipeline

Automated Security Checks

Upon submitting a PR to `winget-pkgs`, Microsoft runs:

- **YAML Validation:** Schema and formatting checks.
- **URL SHA-256 Check:** Verification of download sources.
- **Defender SmartScreen Scan:** Multi-engine virus/malware inspection.
- **Silent Install Test:** Headless sandbox execution test.

Common Pitfall: False Positives

Rust CLI binaries published without digital signatures frequently trigger heuristics (e.g., `Trojan:Win32/ML.Attribute`) on Microsoft Defender and VirusTotal during the automated scan step.

3. Resolving Scans: False Positive Claims Process

Step-by-Step Claim Resolution

If the WinGet bot flags `evnx.exe` with a malware error:

1. **Submit Claim to MMPC**: Submit the binary directly to Microsoft Security Intelligence (`microsoft.com/wsecurity/sample-submission`).
2. **Select Category:** Mark submission as *Incorrectly Detected as Malware / False Positive*.
3. **Provide Verification Details:** Include GitHub release link, source code link, and build logs.
4. **Obtain Clearance ID:** Wait for automated re-scan clearance (usually 1–4 hours).
5. **Update PR:** Comment the MMPC submission ID on the `winget-pkgs` PR to request bot re-validation or maintainer override.

4. Long-Term Solution: Code Signing Reputation

Digital Code Signing

- Signing with an Authenticode Certificate eliminates SmartScreen warnings and automated false positives.
- Open-source projects can apply for free/discounted certificates via community initiatives (e.g., **SignPath.io Foundation**, Signify).

User Installation Experience

```
# Installation command once merged
$ winget install urwithajit9.evnx

# Search package
$ winget search evnx
```

Key Summary

Code signing building binary reputation → Smooth automated PR merges → Instant WinGet distribution updates.

Deep Dive: GitHub Actions Marketplace for `evnx`

Exposing `evnx` in CI/CD Workflows

- Provide a first-party GitHub Action (`https://github.com/urwithajit9/evnx-action`) to automatically download, verify, and cache `evnx` in runner `$PATH`.
- Publish to the **GitHub Marketplace** for one-click discovery and workflow integration.

  `https://github.com/marketplace/actions/evnx-env-security-validation`
- Uses lightweight cross-platform shell logic (or a composite action) with GitHub release binaries.
- Enables seamless environment variable security scanning and configuration checks across CI pipelines.

1. Composite Action Definition (`action.yml`)

urwithajit9/evnx-action/blob/main/action.yml

```yaml
name: 'evnx - .env Security and Validation'
description: 'Scan, validate and secure .env files in CI - secret detection,
    misconfig checks and drift alerts.'
branding:
  icon: 'shield'
  color: 'green'

inputs:
  version:
    description: 'evnx command to run'
    required: false
    default: 'latest'

runs:
  using: 'composite'
  steps:
```

2. Publishing to the GitHub Marketplace

Marketplace Requirements

- Repository must be **public** with a valid `action.yml` at the root.
- Must include a **Marketplace Category** (e.g., *Utilities* or *Developer Tools*).
- Requires a valid `README.md` detailing inputs, outputs, and usage examples.
- Tag releases using semantic versioning (`v1.0.0`, `v1`).

Release Tagging Flow

```
# Tag specific patch and major version
$ git tag -a v1.0.0 -m "Release v1.0.0"
$ git tag -fa v1 -m "Update v1 major
    tag"
$ git push origin --tags -f
```

Pointing the major tag (`v1`) to latest allows users auto-updates on breaking fixes.

3. Consuming `evnx` in CI Pipelines

Example Workflow (`.github/workflows/ci.yml`)

```yaml
name: Validate environment

on: [push, pull_request]

jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Install evnx
        run: |
          curl -sSL https://raw.githubusercontent.com/urwithajit9/evnx/main/
              scripts/install.sh | bash

      - name: Validate configuration
        run: evnx validate --strict --format github-actions
```

Deep Dive: Agent Skills for `evnx`

Empowering AI Coding Agents with `evnx` Context

- Package domain knowledge, CLI syntax, and `.env` best practices into reusable **Agent Skills** (`agent-skills`).
- Gives LLM agents (Cursor, Claude, Copilot, Windsurf) structured context to automatically run `evnx` commands, generate safe templates, and perform secret scanning.
- Installed seamlessly into agent workflows via `npx skills add`.
- **Two Core Skills:** `https://github.com/urwithajit9/agent-skills`
  - `dotenv-best-practices`: `.env` specification, security rules, and multi-stack guidelines.
  - `evnx-cli`: Complete CLI command reference, validation flags, CI/CD integration, and cloud migration workflows.

1. Skill Definition & Agent Installation

Skill Definition (skills/evnx-cli/SKILL.md)

```
---
name: evnx-cli
description: Environment variable
    management, validation, and secret
    scanning using evnx.
---

# evnx CLI Agent Skill

## Key Commands
- Validate: `evnx validate --strict`
- Secret Scan: `evnx scan --format
    sarif`
- Convert: `evnx convert --to json`
```

Installing Agent Skills

```
# Add evnx CLI skill to local AI agent
$ npx skills add urwithajit9/agent-
    skills/tree/main/skills/evnx-cli

# Add .env best practices skill
$ npx skills add urwithajit9/agent-
    skills/tree/main/skills/dotenv-best
    -practices
```

Key Advantage

Zero prompt-engineering overhead — agents autonomously execute `evnx` during code generation and PR reviews.

Beyond the Big Sources

Also automated

- Scoop – JSON manifest in our own bucket `https://github.com/urwithajit9/agent-skills`
- evnx Playground - `https://github.com/urwithajit9/evnx-test`

Planned Channels

- Docker Hub – multi-arch image via `buildx`
- apt/deb – `.deb` built with `cargo-deb`

Lessons Learned

Common Pitfalls

- Version skew: a wrapper published before its binary artifact finishes uploading
- Silent architecture gaps – forgetting `aarch64` until a user's M-series Mac breaks
- Registries that cache aggressively (npm, Homebrew) delaying *de facto* availability
- Signing keys and tokens scattered across five registries – centralize in one secrets vault
- Treat every wrapper as generated code: never hand-edit a manifest in the target repo

Results

- Release time: 45 minutes of manual work $\rightarrow$ 1 tag push, fully automated
- 5 registries, 6 target triples, 1 GitHub Actions workflow
- Zero version-skew incidents since centralizing the release gate
- Install success rate across platforms: 99.6% (up from 91% with manual releases)

Thank you !

Questions?

Feel free to reach out or explore the project

Project Resources

GitHub

evnx.dev

dotenv.space

Personal & Social

ajit.today

@urwithajit9

LinkedIn