Jake Runzer | Mar 4, 2025

Why We’re Moving on From Nix

Today we’re excited to release Railpack — the next iteration of the Railway builder, developed from the ground up and based on everything we’ve learned from building over 14 million apps with Nixpacks.

We first announced Nixpacks nearly 3 years ago and it quickly became the default way to build images from user code on Railway. While Nixpacks works great for 80% of users, that still left us with 200k Railway users who might encounter limitations daily.

It became clear we needed a major builder upgrade to scale our user base from 1M to 100M.

Cumulative builds with Nixpacks over time

Here are the highlights of Railpack:

- **Granular Versioning**: Support for `major.minor.patch` versions of packages (instead of Nix’s approximate versions)
- **Smaller Builds**: We’ve been able to reduce image sizes between 38% (Node) and 77% (Python), enabling faster deploys on Railway
- **Better caching**: Railpack interfaces directly with BuildKit to control the layers and filesystem, resulting in more cache hits (with sharable caches across environments)

You can opt-in to using Railpack for your builds today. It is already powering builds for railway.com and central station.

Our problems with Nix

The biggest problem with Nix is its commit-based package versioning. Only the latest major version of each package is available, with versions tied to specific commits in the nixpkgs repo. We tried to support every patch version, but it looked like this:

```
const AVAILABLE_SWIFT_VERSIONS: &[(&str, &str)] = &[
    // ...
    ("5.4", "c82b46413401efa740a0b994f52e9903a4f6dcd5"),
    ("5.4.2", "c82b46413401efa740a0b994f52e9903a4f6dcd5"),
    ("5.5.2", "7592790b9e02f7f99ddcb1bd33fd44ff8df6a9a7"),
    ("5.5.3", "7cf5ccf1cdb2ba5f08f0ac29fc3d04b0b59a07e4"),
    ("5.6.2", "3c3b3ab88a34ff8026fc69cb78febb9ec9aedb16"),
    ("5.7.3", "8cad3dbe48029cb9def5cdb2409a6c80d3acfe2e"),
    ("5.8", "9957cd48326fe8dbd52fdc50dd2502307f188b0d"),
];
```

This approach isn't clear or maintainable, especially for contributors unfamiliar with Nix's version management.

For languages like Node and Python, we ended up only supporting their latest major version.

But even this was problematic because versions are tied to a single commit SHA. Updating the commit hash to support the latest version of a package meant all other package versions would also update. If a default version changed, there was a high likelihood that a user's build would suddenly fail with unexpected errors.

We feel bad when users can't access the latest packages, but feel worse when previously functional builds suddenly fail.

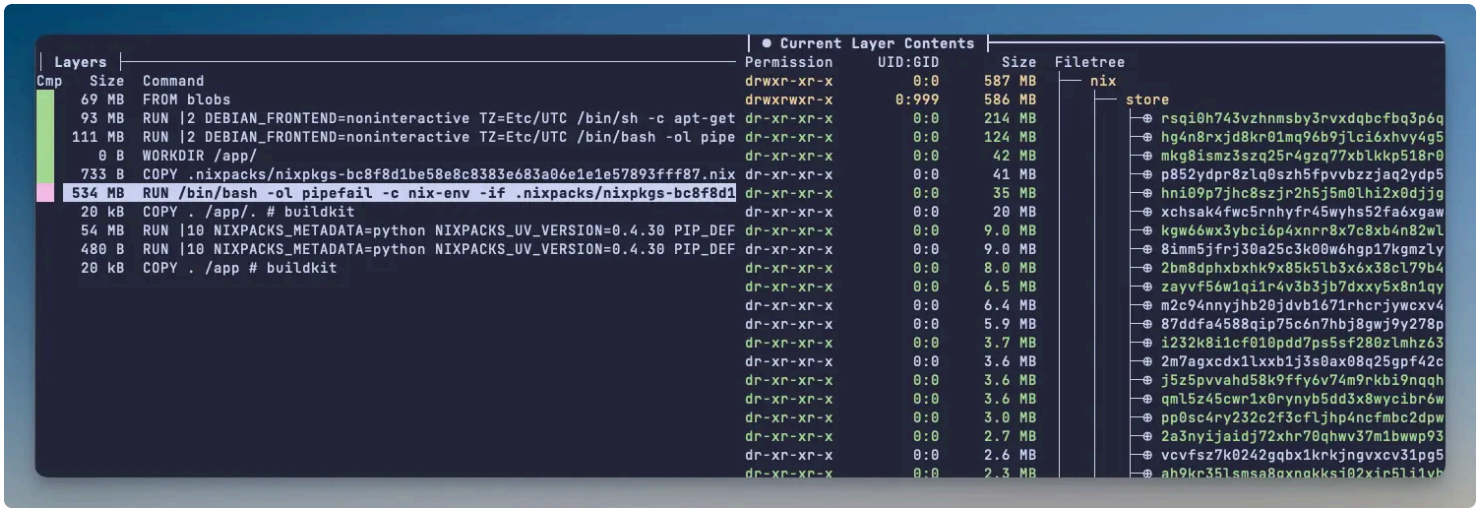
Image sizes and caching

The way Nixpacks uses Nix to pull in dependencies often results in massive image sizes with a single `/nix/store` layer ... all Nix and related packages and libraries needed for *both* the build and runtime are here.

With no way of splitting up the Nix dependencies into separate layers, there was not much we could do to reduce the final image sizes. Not a problem with Nix per se but certainly a problem with how we were using it.

Caching was also problematic as we had little control over when layer caches were invalidated.

Railway injects a deployment ID environment variable into all builds. This means that any layers that run after these variables are added to the Dockerfile are always invalidated and can never be cached.



Result of running

I want to be clear, we don’t have any problem with Nix itself. But there is a problem with how we were using it. Trying to abstract all the parts of Nix that make Nix… Nix, just fundamentally doesn't work.

We don’t want our users to have to understand what a derivation is or why Node 22.14.0 is available on archive version `757d2836919966eef06ed5c4af0647a6f2c297f4` of the unstable channel.

Introducing Railpack

To fix the issues we’ve had with Nixpacks, we built Railpack.

Since we transitioned away from Nix, we also transitioned away from the name Nixpacks in favor of Railpack. We also changed the codebase from Rust to Go because of the Buildkit libraries.

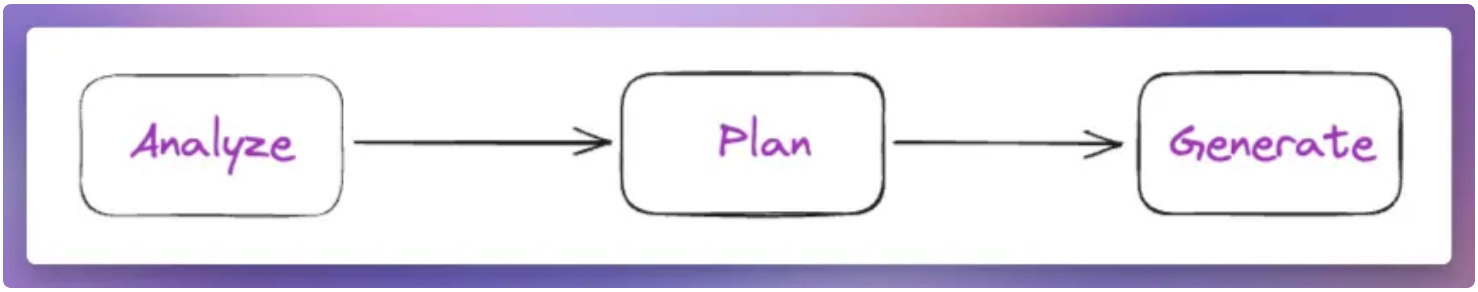
Here are some architectural highlights:

- We generate a custom BuildKit LLB + Frontend to give us much more control over how the final image is constructed — resulting in 38% smaller base Node and 77% smaller base Python images compared to building with Nixpacks
- We use Mise for version resolution and most package installation, though it leaves room to support other executable sources in the future
- We're now able to *lock* the dependencies used when a successful build happens. This means that builds won't break when we update the default Node version from 22 to 24
- We improved secret environment variable management. Railpack leverages BuildKit secrets to prevent variables from appearing in build logs or the final image

How it works

The Railpack process is split into three parts:

- **Analyze**: Look at the code and determine what packages should be installed, what commands should be run, and what the start command should be
- **Plan**: Create a build plan in a JSON-serializable format that contains several steps, each with inputs derived from other steps or entire images.
- **Generates**: Construct a BuildKit build graph based on the inputs and outputs from the plan.



While Dockerfiles are very linear in nature, BuildKit graphs are extremely parallel. Each command runs in its own stage of a multi-stage build and provide precise control over the input layers and how the final file system is assembled.

Railpack analyzes the code and generates a build plan of all the necessary steps needed to build.

Each step specifically defines what previous step or image is required — a format that is much lower-level than what was used in Nixpacks. This plan is then turned into a graph in LLB format and solved.

BuildKit starts at the end and works backwards, pulling from the cache if possible or running the commands to resolve each requested layer.

To invalidate layers when specific environment variables change, Railpack will hash the used variable values and mount a file with the hash to an input filesystem. If the code and used variables don’t change, the layer cache will be hit.

Railpack can therefore fully define how an image is made.

Deploy inputs for a static Vite build

What else does Railpack unlock? We're glad you asked:

- Support for building and deploying Vite, Astro, CRA, and Angular static sites with zero config
- Tight integration between your builds and the Railway UI
- Support for the latest versions of languages with no Railpack release necessary
- Optimized layer caching for a project across environments

How you can use it today

Railpack is available in Beta today. Just enable it in your service settings.

It currently supports Node, Python, Go, Php, and Static HTML deployments, including out-of-the-box support for Vite, Astro, CRA, and Angular static sites, making Railway the easiest place to deploy both your frontend and backend.

We are adding more framework and language support actively, so let us know in Help Station what you want to see first. We are prioritizing depth on the more commonly used languages rather than breadth, at least until the core API and abstraction are nailed.

Railpack is also open source with documentation available at railpack.com.

Continue Reading...

View All Engineering →

ENGINEERING

Incident Report: June 6th, 2025

We experienced an issue with the Railway GitHub login and Dashboard backend.…

 Jake Cooper / Jun 6, 2025

ENGINEERING

Incident Report: April 30th, 2025

We recently experienced an outage that affected our backend API. During this…

 Ray Chen / Apr 30, 2025

PRODUCT

Changelog

Pricing

Starters

Feedback

OSS Kickback

COMPANY

About

Careers

Blog

Shop

CONTACT

Discord

Twitter

GitHub

Email

LEGAL

Fair Use

Privacy Policy

Terms of Service