Learn more

Engineering

Baking Images with Nix for Robust Private Networks at Bowtie

Learn how Bowtie uses Nix to build network appliance images, enabling us to support a variety of formats with a high degree of code reuse and an elegant upgrade process. Ensuring that network appliances are easy to run, test, update, and operate reliably is important and underpins Bowtie's commitment to robust simplicity.

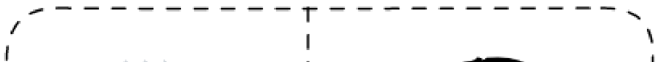
TYLER LANGLOIS
APRIL 11, 2023

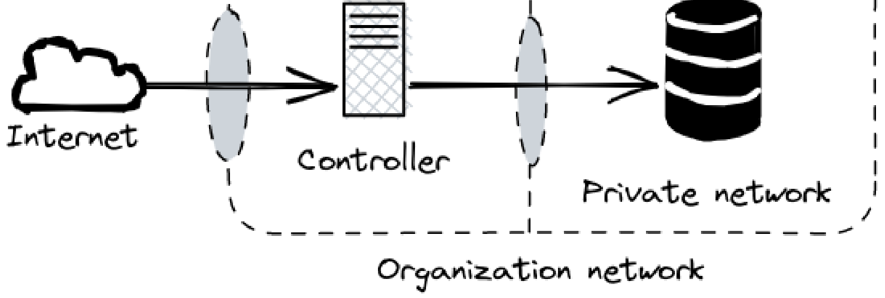
In this engineering deep dive, I'll explain how we build Bowtie network appliance images. Our approach enables us to support a variety of formats with a high degree of code reuse and an elegant upgrade process.

If hacking on Nix to package Rust daemons for cloud images sounds like fun to you, read on!

A Fistful of Hypervisors

To reach private resources, roaming client devices typically need a gateway or endpoint to connect to. At Bowtie, we call these network appliances Controllers, and we have a few unique constraints that inform how we need to build them.





First and foremost is that Bowtie Controllers are installed at your sites and in your cloud environments, which means that we need to provide images in some sort of user-deployable form, as opposed to signing up for a hosted service. If you're on a public cloud like AWS, this means an AMI ID. Other environments may need something downloadable, like a qemu qcow2 disk image or VMDK file. Running on as many platforms as possible ensures we support users where they are rather than impose narrow runtime requirements.

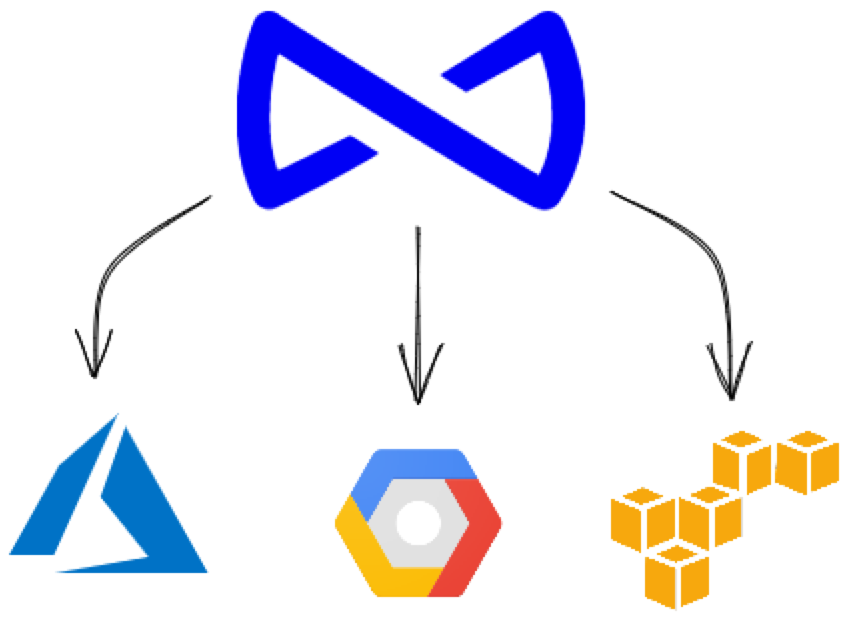
When I first started thinking about potential solutions, my initial instinct was to try and simplify where possible. Could we just ship a container, perhaps? Most operators are pretty comfortable deploying OCI-compliant images into their infrastructure, so offering a Controller image that you could simply docker pull would be immediately approachable and prioritize a smooth user experience. We could even provide a Helm chart or Kubernetes operator if we really wanted to vibe with the zeitgeist.

However, one consideration that we need to take into account is our use of some specific Linux kernel modules. While you might be able to run a privileged container with elevated privileges, hooking into the host’s kernel — without any guarantees of version compatibility and risk of interfering with other workloads — introduces some poor operational hygiene, so we'll need to provide our own kernel. What’s the best way to do that, both from an end-user experience perspective and architectural design?

But Doctor, I Am ~~Pagliacci~~ Packer

We’re all thinking the same thing right now: Packer! This is the type of thing Packer is designed for. Automation-driven machine image creation with pluggable backends fits this use case nicely. So why not use it?

My first hesitation to bet the farm on Packer was fanning out into additional builder types. For example, if you start with the Amazon builder (EBS backend) and add support later for VMWare, constructing image variants is not a matter of changing one parameter and churning out a new image file. In many cases, you will need an API or locally-adjacent hypervisor available as the backend for Packer to use when it spins up a machine to create a new image. This requirement certainly is not a show-stopper, but the limitation felt burdensome. After all, we just need to ship image files to customers, and I would hope that my build tool could create those bits on-disk without the need for a running hypervisor for each platform that we support. Presenting a working hypervisor API like Proxmox or Hyper-V endpoint also becomes more complicated in contexts like CI.



I had other reservations as well. If we know the Controller profile that constitutes a complete appliance, do I really need to loop in a provisioner like Salt or Puppet? Can I consolidate both the image creation process in addition to packaging our server software? Could I tackle both of these tasks in one system?

To be clear, you certainly can build a solution that spans multiple hypervisor formats using Packer and a provisioner of your choice. The fundamentals are there and most of us can envision stitching together the requisite pieces of Packer, Ansible, and a CI pipeline somewhere. Nobody ever got fired for choosing Packer to build cloud images, right? But we can do better.

A Wild Nix Appears

Most people treat Nix like a uranium fuel rod: they know that it's capable of amazing things, but hesitate to experiment with it for fear of irreparable harm to their brain. Allow me to share that I have returned from the core with positive reports and free of damage.

For the uninitiated Nix user, you can write nix (the programming language) code that you feed into nix (the command line tool) that builds reproducible artifacts thanks to its inputs being pure, or total and well-defined. For example, Alice can write a Nix package for curl, send the recipe over to Bob, and his

Nix build of curl will be identical due to every dependency and every required package captured as an input. (curl is already in nixpkgs – the Nix package repository – so don't go and do that.)

NixOS takes this idea and extends it to an entire operating system. In addition to saying, “here’s how to build curl from the revision of libc on up”, NixOS applies the principle to the entire system and then also helpfully bundles up common configurations into modules. For example, here is what you would place into a configuration.nix file to ask NixOS to start and enable OpenSSH: it sets appropriate firewall rules and installs the sshd daemon along with all its dependencies:

```
{
  services.openssh.enable = true;
}
```

That is all well and good, but you could just as easily pull in an Ansible module as a Packer provisioner to build up a fully-featured appliance with several services. Why bother with Nix, then?

That aforementioned OpenSSH option – along with everything else in a NixOS configuration – ends up as a nix (the programming language) value at the end of the day. Think of this “system derivation” value as a data structure, like a map or a hash, that describes how to construct and configure a system from primitive files like /etc/fstab on upwards. Some intrepid engineers noticed this and, with the entire system definition available as a simple value, asked, “can we do anything interesting with it?”

It turns out that you can. When you check out the nixos-generators project and scroll down to the supported image formats, you get the idea: nixos-generators accepts a NixOS configuration value and transmogrifies it into any image format you want.

In addition:

- No external API calls are necessary at build-time – nixos-generators knows how to natively bake the system into the correct bits for the chosen format.
- Swapping between image formats is literally one string change away. Little (or no!) changes are necessary to build an Amazon AMI disk versus a Proxmox one.
- We gain the ability to configure sidecar services on the appliance for free, like Prometheus or Grafana, out of the box with native NixOS options like services.prometheus and services.grafana.

How Controller Sausages are Made

A NixOS system is an esoteric beast: you cannot drop an x86-64 Linux executable onto a NixOS machine and expect it to work because /lib does not exist. Each executable is strictly pinned to its own, isolated set of shared libraries, so we cannot ship our plain Bowtie server software that we build with cargo build on a network controller. To do that, we must write a Nix package for it – technically we could skip this step if we used containers, but defining a proper Nix package offers us other benefits that I will get into later.

That means that job number one is to express our build steps in Nix. We build our Rust-based server-side daemon using the crane library. Doing so requires no changes on the Rust project side because a version-locked Cargo.lock is sufficient. We feed the path to our source code to a Nix function, our lockfile, and a Nix derivation comes out the other side. Crane is great! Thank you for the hard work, crane maintainers. We also hook into our repository's package.lock via dream2nix to bundle the frontend. dream2nix is also suspiciously good software.

Our daemon does not run in isolation, though, and we need some associated plumbing like directories for state, systemd service units, and so on. We modeled our own bowtie_server NixOS module after many of the upstream NixOS modules so that pulling in our daemon along with all its necessary system changes is just an option call away:

```
{
  # Install and enable the server-side daemon:
  services.bowtie_server.enable = true;
}
```

With our own bowtie_server NixOS module prepared, we roll up all the nix code that enables the module and configures the rest of the system – like installing kernel modules and other packages – into a top-level controller NixOS configuration. This is the value that you might point nixos-rebuild or nix build at, ready to configure and build a new system.

Golden-brown Bowtie Flakes

I avoided using the term “nix flake” until now, but this is the right time to introduce it. All of the packages, NixOS modules, and NixOS configurations we write end up in the outputs for our repository's flake.nix. Here's a snippet from that file's nix flake show command that displays each image type (this output has been truncated and cleaned up a little bit):

```
$ nix flake show
git+file:///home/tylerjl/src/bowtie
└───nixosConfigurations
    ├───amazon: NixOS configuration
    ├───ami: NixOS configuration
    ├───gce: NixOS configuration
    ├───hyperv: NixOS configuration
    ├───proxmox: NixOS configuration
    ├───qcow: NixOS configuration
    ├───qcow-efi: NixOS configuration
    ├───qemu: NixOS configuration
    └───vmware: NixOS configuration
```

Per the nixos-generators documentation, we can feed that nixosConfigurations.amazon Nix flake output, for example, into a function called nixosGenerate and a disk image appears inside ./result. We throw around the word “magic” a lot in software, but the ease with which I can reliably churn out arbitrary cloud images for our product really **does** feel like magic.

Capturing all of the nix artifacts we define within our flake.nix has all sorts of benefits, and most of these were not intentional goals but rather fortuitous outcomes.

For example, we can quickly reconfigure a host by pointing nixos-rebuild at its corresponding nixosConfiguration (an EC2 instance would reference nixosConfigurations.amazon). This changes a Controller in-place, letting us build the system either locally or remotely. Passing the test argument instead of switch also permits us to experiment freely without fear of “bricking” a Controller (test does not update the bootloader, so “have you tried rebooting?” actually fixes broken systems, in this case).

We can easily obtain packages in any development environment with nix build because flake.nix bundles cargo and other dependencies. Portable devShell environments are almost a free by-product of this fact, which alleviates the need to manage version-specific installations of tool chains for Rust, Python, or Nodejs. My workstation has **no** python, cargo, or npm executables in my $PATH – all of our development requirements are sandboxed in our repository's main devShell.

Local qemu-based virtual machines are easy to build by targeting the config.system.build.vm system attribute. As the engineer responsible for our Controller appliances, I use this nearly every day. Creating a new Controller from scratch (but with already-compiled cargo artifacts) takes me about 14 seconds, so I experiment with upgrades and variations often. We took this one step further and integrated deeply into the nixosTest framework to actually spin up multiple qemu virtual machines to validate network clustering behavior, but that's an interesting topic for another time.

My personal favorite, though, is how we enable system updates.

Hard-to-break Updates

Fresh Ubuntu installations are configured to use apt repositories to retrieve compiled packages (not raw source code). In a similar fashion, NixOS systems draw their built binary packages from a common, pre-configured cache (even though the source of all those packages ultimately lives at https://github.com/NixOS/nixpkgs). These remote binary caches are not mechanically complex, and operators can very easily stand up their own using object stores like S3 for easy-to-use custom caches.

As a preliminary CI/CD step before building and publishing Bowtie Controller images, our pipelines build the "base" of the Controller – complete with all the Nix derivations that eventually get folded into each image type (AMI, qcow2, etc.) – and then recursively signs all these paths before pushing them into a private Nix cache. Here is the GitLab matrix for a recent Controller release:

The first effect this has is that any of us can trust the cache and benefit from our own private build cache. It is particularly beneficial in CI where subsequent builds can avoid building derivations more than once.

The second benefit is nearly-instant and atomic updates. Controllers may fetch some simple key/value information about our latest appliance builds from a metadata service endpoint and then retrieve the system store paths from our private, signed cache and activate them to perform system updates. An aptly-named update script is wonderfully simple and relies solely on nix copy to download the complete Nix store path and then calls switch-to-configuration to activate it. This capability is simple and powerful enough that a user can leverage it however they would prefer: either as a one-off command or scheduled and automated via an event like a systemd timer. When run non-interactively, we simply select the latest system derivation to retrieve and activate rather than presenting a list of options.

This update strategy sidesteps entire problem categories. Users cannot suffer configuration drift because the system derivation is always consistent. Our cache ensures that **every** system revision is available for rollback or recovery. We never need to worry about deploying bleeding-edge updates to older appliances, either: without globally-shared libraries, our daemon ships alongside its entire dependency chain.

Leaning into the native system update mechanism with our own cache reaps other benefits, too: updated services know when they should restart without explicitly configuring this, and rollbacks are available for fail-back operations. If an update fails, just boot into the previous system configuration.

Future Work and Improvements

There is always more work to do, and no solution is perfect. While we are happy with this approach, Nix expertise is hard to find and develop. With that said, our team is productive with this setup despite very few of us having prior experience with Nix. Some developers new to Nix have even successfully packaged their Rust crates. The interface for flakes in particular has enabled others to clearly reason about where and how to bring in updates to our Controller image. The learning curve is still steep, but not insurmountable.

```
# Example of one of our developer-contributed package derivations.
pkgs.rustPlatform.buildRustPackage {
  pname = "slash-setup-rust";
  version = "0.1.0";
  src = ./setup-rs;
  buildInputs = [ pkgs.openssl pkgs.pkg-config ];
  cargoLock.lockFile = ./setup-rs/Cargo.lock;
};
```

Additionally, cross-platform support is improving, but not yet at parity with most Linux systems. For example, assembling and running an x86_64-linux NixOS qemu virtual machine is seamless on my x86-64 Linux workstation, but the process is less straightforward for engineers on macOS (darwin in Nix-speak). This path is full of paper cuts for now, but contributors like Gabriella Gonzalez have made recent strides with improvements like nixpkgs support for Linux builders running on macOS. I'm hopeful this trend will continue.

Some of these topics could fill their own blog posts, but I hope this has given you a taste of how to leverage Nix for tremendously positive outcomes. Ensuring that network appliances are easy to run, test, update, and operate reliably is important, and underpins Bowtie's commitment to robust simplicity.

Thanks to Drew Raines, Chris Kuchin, and Domen Kožar as beta readers for this piece.

ABOUT THE AUTHOR

Tyler Langlois

Tyler is a Principal Engineer at Bowtie and is focused on all things operational, as well as documentation and packaging. In previous roles, he's accumulated diverse experience as a software engineer, devops engineer, site reliability engineer, and as a frontend engineer.

Related Articles

Engineering

Baking Images with Nix for Robust Private Networks at Bowtie

TYLER LANGLOIS • APRIL 11, 2023

Networking

The Importance of Network Resiliency

WILLIAM ESHAGH • FEBRUARY 21, 2023

Networking

Bowtie Always Connects

JUSTIN FRANCESCONI • FEBRUARY 21, 2023

Company

Move the Enterprise Network to the Edge with Secure Access from Bowtie

WILLIAM ESHAGH • FEBRUARY 21, 2023