ElastiNix

Running NixOS smoothly on AWS (with a twist)

Pim Snel

September 2026

TechNative

About me

Hello, I’m Pim

- worked 20 years at **Lingewoud**, a software agency I founded in 2000.
- Full stack and infrastructure…
- current preferred languages: Nix (duh), Crystal, Go, and Elixir
- Always chasing the same thing: **one source of truth**
- Today: **Architect at TechNative**

My first encounter with Nix

I loved the core idea of Nix but I hated the syntax…

I care about the aesthetics of source code. I loved Ruby.

```ruby
5.times do
  puts "I love ruby"
end
```

Compared to Nix …

```
let
  loop = n: if n == 0 then null
            else builtins.trace "It harder to love nix" (loop (n - 1));
in loop 5
```

I was seriously planning a Nix dialect that looked like Ruby.

I dropped that plan.

Nix grows on you.

Now I want *everything* described in Nix.

TechNative

TechNative

- Almost **7 years** old, based in Amersfoort, NL
- We do fun cloud and AI stuff and we do smart FinOps as well
- **AWS Advanced Partner** DevOps competency, ISO 27001, 100+ launches
- **Open source first**: 80+ public repositories, Apache-2.0
- **Terraform** and **Nix** are our core cloud technologies

ElastiNix

Running NixOS smoothly on AWS

TechNative

Why we built it

- One **source of truth** per machine, the OS *and* the infrastructure it runs on
- Fits **CatStack**, our Terraform framework: domains, environments, DRY
- Made for **teams**
- Made for **GitOps**
- **One command** deploys both: the NixOS machine and the Terraform infra

A blog post which started everything

> “Terraform and Nix are both pieces of software whose success comes from their ability to provide a declarative interface to a procedural world.”

Jonas Carpay, 2022 — End-to-end declarative deployment

Where it came from

- **Sept 2022**. Carpay publishes the core idea
- **March 2025**. our first commit
- Grew **organically**, one real client project at a time
- Since `nixos-25.05`: versioned against official NixOS releases
- Two repos: the **flake** and the **Terraform module**

The idea we borrowed

Split the machine in two:

- **bootstrap image** minimal NixOS, provisioned once, left alone
- **live config** the actual machine, deployed onto it

Change your application → you do **not** reprovision the hardware.

Features

- A **remote library flake** — shared modules across every client project
- **Auto Scaling Groups**
- The matching **Terraform module**, so the infra ships with the machine
- **Smol** — work in progress: making the images as small as we can

Made for AWS

- EC2-optimized **kernel drivers**
- **CloudWatch** integration
- **SSM Agent** — deployment over SSM, no public SSH
- **agenix** for secrets

The module library

What grew on top, from real projects:

- Vaultwarden
- Prometheus & Grafana
- central vulnerability scanning (vulnix + trivy)
- Postfix/SES relay
- Atuin
- Documenso
- OptScale
- Twenty CRM

Architecture

- Nix builds **both** artifacts
- Terraform receives them as **store paths**

One command

```
packages = {
  prodApply = elastinix.lib.tf_command (machineArgs // {
    inherit runSystem;
    varsfile = ../../infra_environments/prod/prod.tfvars.json;
  });
};
```

```
nix run .#prodApply
```

What that command really is

```
export TF_VAR_ec2_bootstrap_img_path="${bootstrapImage}/nixos_image.vhd"
export TF_VAR_ec2_host_live_path="${liveConfig...system.build.toplevel}"

terraform apply -var-file=prod.tfvars.json
```

Change one line of NixOS config → new store path → Terraform sees a diff.

That is the whole trick.

Baked vs. Brewed

Two ways to get a machine onto AWS.

Baked

1. Build the **full** NixOS image
2. Terraform uploads it, imports the snapshot, registers the AMI, boots it

Brewed

1. Build a **minimal** NixOS image
2. Terraform uploads, imports, boots it
3. Copy the rest of the closure onto the running machine

Baked vs. Brewed when

	Baked	Brewed
Full deploy	~30 minutes	fast
Shape	Auto Scaling Group	single EC2
Image changes	rarely	often
Typical use	production fleets	development

Brewed is also how you *develop* the image that will later be baked into an ASG.

Demo

A Vaultwarden stack on AWS

Demo — the files

```
flake.nix                  demoPlan · demoApply · demoDestroy
nix/hostconf.nix           the machine
infra_environments/demo/   account, region, domain, subnet
terraform/                 AMI · instance · volume · address · IAM · backups
```

`demos/elastinix-vaultwarden/` — in this repo.

The machine

```
elastinix.services.vaultwarden.enable       = true;
elastinix.services.postgresql.enable        = true;
elastinix.services.nginx.enable             = true;
elastinix.services.cloudwatch-agent.enable  = true;
```

- The vhost, the ACME certificate and TLS come with the module
- The word “AWS” does not appear in this file

Run it

```
1  nix run .#demoPlan
2  nix run .#demoApply
3  nix run .#demoDestroy
```

One command builds the image, uploads it, registers the AMI, boots the machine, and deploys the configuration onto it.

Properties

Vault and database	their own EBS volume, `prevent_destroy`
Address	Elastic IP — DNS does not churn when the machine is replaced
Ingress	80 for ACME, 443 for the vault. Nothing else
Deployment path	SSM. No public SSH
Secrets	an env file, delivered out of band — never in the Nix store
Backups	nightly `pg_dump` to S3; the instance may write, nothing more
Config change	new store path → deploy onto the running machine
Image change	new AMI → new instance, volume and address survive

One name, two languages

```nix
# nix/hostconf.nix
bucket = "vaultwarden-${tfvars.infra_environment}-${tfvars.aws_account_id}-backups";
```

```hcl
# terraform/main.tf
bucket = "vaultwarden-${var.infra_environment}-${var.aws_account_id}-backups"
```

Same file, read twice. Neither side hard-codes it — and that is still two languages agreeing by hand.

Hold that thought.

Load balancer, or not

Code only — we will not run this one.

Roadmap

Roadmap

- Adopt **deploy-rs** instead of our own deploy mechanism
- Smaller images
- Better ASG story

- **Make the Terraform files unnecessary. Describe everything in Nix.**

Wait. Describe everything in Nix. we fixed this

We fixed this **big time**.

We created Nivis.

TechNative

One source of truth

for your entire tech stack.

Where that roadmap item went

- I wanted to delete our Terraform files

- **15 June 2026** — first commit
- **Today** — twelve weeks later

Why

The wall

- Inside a machine, Nix is **total**: every package, every setting, every version
- Outside it — servers, networks, DNS, databases — Nix **stops**
- There you switch to another language, with other rules

That wall is **artificial**. Both server and infra are describable things.

We were standing on that wall

```
1  export TF_VAR_ec2_host_live_path="${liveConfig...system.build.toplevel}"
```

- That string **is** the wall
- Nix computed it. Terraform consumed it.
- Nothing ever came back.

The problem

> Complexity does not arise because systems are large, but because we describe them in different ways.

- The same reality, described twice
- One person writes Terraform, another writes NixOS
- The boundary is not in the system — it is in our tools
- Energy goes into **translation** instead of improvement

What Nivis is

- OpenTofu/Terraform **provider resources as first-class Nix values**
- A thin **Go executor** speaks the plugin protocol to **unmodified provider binaries**
- No HCL. No second state language.

Nix is the configuration. The provider does the work.

The neighbours

Tool	In one line
ElastiNix	What you just saw. Nix builds the machine, Terraform ships it. Two languages, one direction, AWS only
Nivis	Provider resources as first-class Nix values; spawns unmodified provider binaries
OpenTofu / Terraform	The engine and the provider ecosystem. HCL, mature, everywhere
Terranix	Generates HCL from Nix, then hands it to Terraform
NixOps 4	Nix-native deployment orchestrator, NixOS-deployment heritage
Pulumi	Real languages; reuses TF providers by **compiling** them through a bridge
CDK / CloudFormation	Real languages synthesizing CloudFormation. AWS-first

Where Nivis actually differs

	ElastiNix	Nivis	OpenTofu	Terranix	NixOps 4	Pulumi
Config language	Nix + HCL	Nix	HCL	Nix → HCL	Nix	TS/Py/Go
Written in	Nix	**Nix** + Go	Go	Nix	**Nix** + Rust	Go
How it uses TF providers	through Terraform	spawns them	native	generates HCL	in progress	compiles them
Outputs feed back into config	no	yes	HCL refs only	no	yes	yes
Targets	AWS only	any provider	any	any	any	any
License	Apache-2.0	Apache-2.0	MPL-2.0	MIT	LGPL-2.1	Apache-2.0

NixOps 4 — the real difference

NixOps4 is closest by **intent** — and by **mechanism** too.

	NixOps 4	Nivis
What it is	a **platform**: its own resource-provider interface	only the Terraform-provider coupling
Terraform providers	*in progress*; one provider kind among many	the whole mechanism — every OpenTofu or Terraform provider, day one
Written in	Rust	Go
Providers come from	packaged in Nix	the OpenTofu registry, checksum-verified
License	LGPL-2.1	Apache-2.0

So it is **scope**, and it is licensing. NixOps is building a platform. This is one coupling — and you can build a business on Apache-2.0.

How

The hard part: two ideas of time

- **Nix** wants to compute everything up front, then freeze it
- **Providers** find out along the way — an IP does not exist until the machine boots
- And your next step needs that IP

This is why the combination stayed a dream for so long.

The round trip

1. Nix evaluates as far as it can
2. The executor drives the providers; they return **real** values
3. Those values are injected back into Nix, which **re-evaluates**
4. Repeat until nothing new resolves — a **fixpoint**

The round trip, in the IR

```
__ref      →  another resource's output
__derived  →  a value Nix computed from an output
```

- Not-yet-known values become **typed placeholders**
- The **ledger** carries the real outputs back in
- A deeper chain simply takes more phases

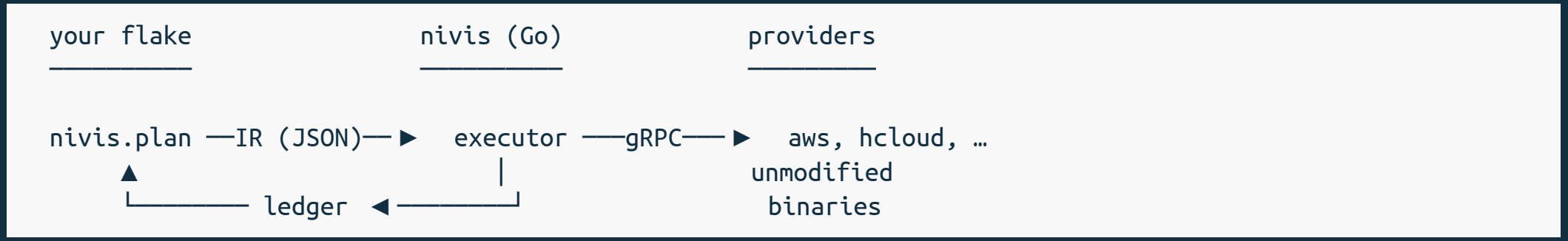
Why not `Output<T>`?

- Pulumi gets promises for free — a Pulumi program is a **running process**
- Nix is a **batch evaluator**. There is no live runtime to await in.

So Nivis does the honest, Nix-shaped thing: **re-evaluate to a fixpoint.**

Architecture

The pieces



The flake interface

```
nivis.plan = ledger: lib.toIR {
  providers.aws = lib.mkProvider {
    source = "registry.opentofu.org/hashicorp/aws";
    config.region = "eu-central-1";
  };
  resources = [ ... ];
  inherit ledger;
};
```

`ledger → IR`. That is the entire interface.

Spawn, don’t link

- Nivis speaks `tfprotov5`/`v6` to the **same binaries** OpenTofu runs
- Pulumi *compiles* every provider through a bridge, one at a time

Spawn-not-link means **every** OpenTofu provider works on day one.

Two frozen contracts

- **The IR** — `IR-CONTRACT.md` + `ir-schema.json`; the Nix producer *and* the Go consumer validate against it
- **The flake interface** — `nivis.plan = ledger → IR`

Change the spec before you change the shape.

The CLI

```
nivis plan      # + create   ~ update   -/+ replace   = no change
nivis apply     # resolve across phases, to a fixpoint
nivis state show <resource-id>
nivis destroy
```

Features

The Nix library

- `mkResource` · `mkData` · `mkProvider`
- `refAttr` · `str` · `derived` — wiring one resource into the next
- `mkVars` — typed variables from `--var`, `--var-file`, environment
- `mkResources` (`count`, `for_each`) · `evalModules`

Pure: **builtins only**, no nixpkgs. It evaluates without the binary cache.

Typed constructors, from any provider

```
nivis gen --provider <binary> --out <dir>
```

- Reads the provider's **own schema**
- Emits typed Nix constructors — nested blocks, correct list-vs-single shapes
- The generated constructor **is** the reference documentation

The machine is a derivation

- `drv` / `drvFile` mark a Nix build output
- `nivis apply` realises it, then hands it to the provider
- Build a NixOS image in Nix → snapshot → boot it

There is no install step.

State

- Local state file by default
- **S3 backend**, with locking
- `nivis state migrate --to-remote` — locks both sides, copies, verifies by reading back, *then* removes the source

Licensing

- Nivis's own code: **Apache-2.0**
- Vendored Terraform protocol files: MPL-2.0

No BUSL anywhere.

Where we are weak

- **Alpha.** Small surface, never run at scale
- No policy as code, no RBAC, no audit trail
- Provider registry integration still needs hardening
- No production track record, no commercial support

Maturity is a feature too — and there Terraform and Pulumi lead.

Demos

Demo 1 — Vaultwarden on EC2

The same workload you just saw, driven by Nivis — no Terraform module.

```
nix build ─► NixOS image ─► AMI ─► EC2 ─► Caddy + TLS
```

- The vault lives on its **own EBS volume**, the address on an **Elastic IP**
- So replacing the machine keeps both
- The DNS record is bound from the allocated address **inside the same apply**

That binding is the round trip, doing real work.

Two domains, on purpose

```
./stackctl demo 010_dns              apply   # the hosted zone
./stackctl demo 020_vaultwarden_ec2  apply   # the workload
```

- A zone **outlives** every machine it points at
- Destroying a demo must not take your name servers with it

The admin token is delivered out of band, through SSM. **No secret enters an IR, a plan, or a state file.**

Demo 2 — the same workload, on Hetzner

- `nixos/vaultwarden` is a **cloud-agnostic module**
- The Hetzner demo imports it **unchanged**
- Booted from a Nix-built snapshot, through our own `terraform-provider-hcloudimage`

One workload description. Two clouds. Nothing rewritten.

Demo 3 — the registry

- Every OpenTofu provider is Nivis-compatible **by design**
- So the registry does not port anything — it makes them **discoverable**
- Documentation is **derived from the provider's own schema**, via `nivis gen`
- Which means the docs always match the binary you actually run

https://github.com/nivis-project/registry

Modules, not just resources

- `nivis-aws-amplify-site` — a GitHub-connected Amplify site
- `nivis-aws-form-action` — altcha proof-of-work captcha → Lambda → SES

Both public, both plain Nivis modules. That form endpoint is what answers the contact form on technative.eu.

Roadmap

Already landed

- The round trip, across two providers, to a fixpoint
- Real AWS: create · update · replace · destroy
- Variables, datasources, schema codegen
- NixOS image → AMI, end to end
- Remote state on S3, with locking

Next: a daily driver

- Legible plan output — colour by change type, show the phases
- Per-provider reference documentation
- Shell completion, state ergonomics

Done when: a Nix developer runs a real project day to day without falling back to Terraform.

Then: team-ready, and beyond

- Drift detection · multiple environments
- **Policy as code — in Nix**, not in a second engine
- RBAC, teams, audit
- Secrets that never transit the Nix store
- Scale: measure first, optimise only what is measured

Try it

```
1  nix run github:nivis-project/nivis#nivis -- --version
```

- Docs: https://nivis-project.github.io/nivis/
- Code: https://github.com/nivis-project/nivis
- Demos: https://github.com/nivis-project/nivis-demos
- ElastiNix: https://github.com/wearetechnative/elastinix

These slides: https://github.com/wearetechnative/talk-nixos-meetup-2026

It is early. That is exactly why now is a good time to break it.

Everything in one place

These slides	`wearetechnative/talk-nixos-meetup-2026`
ElastiNix	`wearetechnative/elastinix`
ElastiNix Terraform module	`wearetechnative/terraform-aws-module-elastinix`
Nivis	`nivis-project/nivis`
Nivis docs	https://nivis-project.github.io/nivis/
Nivis demos	`nivis-project/nivis-demos`
Registry	`nivis-project/registry`
Amplify module	`wearetechnative/nivis-aws-amplify-site`
Form module	`wearetechnative/nivis-aws-form-action`
Introducing Nivis	https://technative.eu/en/blog/introducing-nivis/

Colophon

Talk	Pim Snel — Architect, TechNative
Slides	Quarto + reveal.js, Apache-2.0
ElastiNix owes its core design to	Jonas Carpay, *End-to-end declarative deployment* (2022)
Nivis	Apache-2.0. No BUSL anywhere