

OpenTofu Day
NORTH AMERICA

# Private Data Providers for Clean Data Parsing

## Clean HCL Through Go

Craig Sebenik
Aurora Innovation
OpenTofu Day North America 2026

https://github.com/craig5/private-data-provider-example

# Who am I? What is Aurora?

## Who am I?

- Programming for almost 50 years
- Drug design and computational physics
- Lead for Cloud Observability at Aurora
- Passion for Food:
  - Le Grand Diplôme from Le Cordon Bleu
  - Master's in Italian Cuisine from Apicius (Florence, Italy)
  - Master's in Gastronomy, University of Reims Champagne-Ardenne

## What is Aurora?

- Self-driving trucks
- Large engineering problem
- Engineers with a very wide variety of skills
- Some are PhDs in their fields, but know very little about cloud services

# Terraform = OpenTofu

# I hate Terraform.

# What is Terraform?

| | |
|---|---|
| **The ecosystem** | Providers, modules and the registry |
| **The dependency graph** | What depends on what, so plan and apply happen in the right order |
| **State** | What already exists, so a plan is a diff |
| **The language** | HCL syntax, plus Terraform's expressions and functions |

# The problem

# The problems at Aurora

- **Vendor migrations**
  Change vendors without rewriting every module that uses them
- **Vendor providers do not catch every error**
  Bad input can get through plan, and fail at apply – or not at all
- **Simple abstractions**
  Many of our engineers do not know HCL, so what they touch has to be simple

# Vendor migrations

- Goal: change vendors while changing Terraform as little as possible
- The modules that use a vendor should not care which vendor it is
- But every vendor describes the same things in its own data format
- Example: moving observability vendors – the two had slightly different ideas of what defines an alert

# Vendor providers do not catch every error

- A third-party provider may validate some data. But, rarely does it check everything.
- Many real checks happen in the vendor's API, at apply.
- Our CI/CD can prevent a bad **plan** from getting committed.
- Example: names with dashes that should use underscores.

# Parsing YAML in HCL

```
users = {
  for u in flatten([
    for t in local.teams : [
      for u in t.users : {
        username  = u.username
        full_name = u.full_name
        team      = t.team
        email     = try(u.email, "${u.username}@${var.email_domain}")
        roles     = tolist(try(u.roles, t.default_roles, []))
      } if !try(u.disabled, false)
    ]
  ]) : u.username => u
}

bindings = {
  for b in flatten([
    for u in local.users : [
      for r in u.roles : { user = u.username, role = r }
    ]
  ]) : "${b.user}/${b.role}" => b
}
```

# Real life data has quirks

One person writes a list, the next writes a string:

```
- username: alan
  roles: [developer, oncall]
- username: grace
  roles: admin
```

```
# clean data (the previous slide)
roles = tolist(try(u.roles, t.default_roles, []))

# real data
roles = tolist(flatten([
  try(u.roles, t.default_roles, [])
]))
```

# The trouble with parsing in HCL

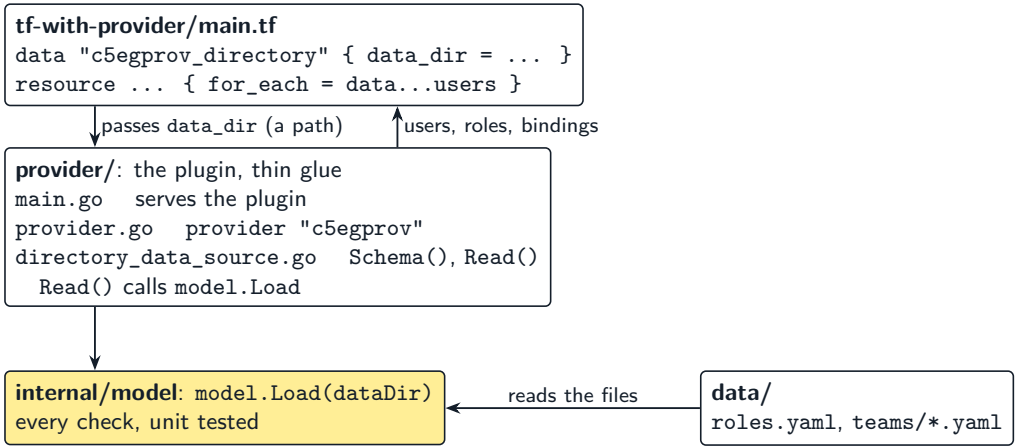
- **Hidden data model:** the shape of the data is implied by nested expressions
- **Hard to test:** the main check is running `plan` and reading the output
- **Fragile:** `try()` can turn a mistake in the data into a silent fallback
- **Difficult reviews:** complex code can be hard to understand and reason about

# Parse with Go and the Plugin Framework

# What is Terraform?

| | |
|---|---|
| **The ecosystem** | Providers, modules and the registry |
| **The dependency graph** | What depends on what, so plan and apply happen in the right order |
| **State** | What already exists, so a plan is a diff |
| **The language** | HCL syntax, plus Terraform's expressions and functions |

# How the code fits together



# Unit testing the parsing

One case, from `model_test.go`:

```
{
  name: "role instead of roles",
  teams: map[string][]byte{"a.yaml": []byte(`
team: a
default_roles: [developer]
users:
  - username: ada
    full_name: Ada Lovelace
    role: [admin]
`)},
  want: `line 7: unknown key "role" in user entry`,
},
```

`go test -v`, trimmed:

```
TestParseErrors
  PASS  unknown_role
  PASS  no_roles_and_no_default
  PASS  role_instead_of_roles
  PASS  roles_is_a_string
  PASS  no_full_name
  PASS  explicit_empty_roles_list
  PASS  roles_key_with_no_value
  PASS  explicitly_empty_email
  PASS  role_listed_twice
  PASS  no_team_name
  PASS  typo_in_a_team_file_key
  PASS  same_user_in_two_teams
ok  .../internal/model  0.194s
```

# The same check, twice

**Before:** the check in HCL…

```
precondition {
  condition = contains(
    keys(local.roles), each.value.role)
}
```

…and again in Python, to test the data

```
def test_only_known_roles(entry):
    _, team, user = entry
    roles = effective_roles(team, user)
    assert set(roles) <= set(load_roles())
```

**After:** one copy, tested where it runs

```
for _, r := range roles {
    if _, ok := d.Roles[r]; !ok {
        return nil, fmt.Errorf(
            "%s: unknown role %q",
            at, r)
    }
}
```

# Exposing it as a data source

Schema(): what the HCL sees

```
Attributes: map[string]schema.Attribute{
  // In: set in the HCL
  "data_dir": schema.StringAttribute{
    Required: true},
  "email_domain": schema.StringAttribute{
    Required: true},

  // Out: filled in by Read()
  "users": schema.MapNestedAttribute{
    Computed: true, ...},
  "roles":    ...,
  "bindings": ...,
},
```

Read(): runs during plan

```
// Get data_dir and email_domain
// from the config ...
dir, err := model.Load(
  m.DataDir.ValueString(),
  m.EmailDomain.ValueString())
if err != nil {
  resp.Diagnostics.AddAttributeError(
    path.Root("data_dir"),
    "Invalid data directory", err.Error())
  return
}

// Copy dir.Users, dir.Roles and
// dir.Bindings into m ...

resp.Diagnostics.Append(
  resp.State.Set(ctx, &m)...)
```

# The HCL, with the provider

```
data "c5egprov_directory" "this" {
  data_dir     = "${path.module}/../data"
  email_domain = var.email_domain
}
resource "terraform_data" "user" {
  for_each = data.c5egprov_directory.this.users
  input = {
    username  = each.value.username
    full_name = each.value.full_name
    email     = each.value.email
    team      = each.value.team
  }
}
# resource "terraform_data" "role": the same shape, over .roles

resource "terraform_data" "binding" {
  for_each = data.c5egprov_directory.this.bindings
  input = {
    user = terraform_data.user[each.value.user].output.username
    role = terraform_data.role[each.value.role].output.name
  }
}
```

# Local development

# See the data first

- Write a small CLI that wraps the same core code as the provider
- Look at the resulting data structure completely outside of Terraform: it is *just* the parsing
- Ours is `c5egprov-show`: a table, or the exact JSON the HCL gets – and it runs under a debugger

```
$ c5egprov-show -data-dir data
USERNAME  FULL NAME        TEAM      EMAIL                      ROLES
ada       Ada Lovelace     data      ada@example.com            analyst
alan      Alan Turing      platform  alan.turing@example.org    developer, oncall
dennis    Dennis Ritchie   platform  dennis@example.com         developer
edsger    Edsger Dijkstra  data      edsger@example.com         analyst, developer
grace     Grace Hopper     platform  grace@example.com          admin
...

$ c5egprov-show -data-dir data     # with role: for roles:
data/teams/platform.yaml: line 7: unknown key "role" in user entry
```

# Local dev gotchas

- Install into `~/.terraform.d/plugins`: a plain `init` finds it, no CLI config needed

  `<host>/<namespace>/<name>/<version>/<os>_<arch>/terraform-provider-<name>_v<version>`
- Use a hostname that is not a real registry, so it will not resolve to a public provider with the same name
- Rebuilt the same version? The lock file's checksum rejects it – even with `init -upgrade`. Delete `.terraform.lock.hcl`, or bump the version
- For a faster loop: `dev_overrides` in the CLI config
- Abstract out common commands and arguments with other Unix tools, e.g. a bash script or a Makefile

# Running it for real

# Getting it to CI and laptops

- At Aurora: a third-party CI/CD platform that includes a private provider registry
- Rolling your own is simple: a *network mirror* is just static files over HTTPS – much like a yum or apt repo
  - `tofu providers mirror` generates the files
  - Clients point at it with `network_mirror` in the CLI config
- A full registry adds service discovery and GPG-signed releases
- Give every change a new version, rather than rebuilding an existing one

# Naming it

```
data "c5egprov_directory" "this" { ... }
```

- No _: it separates the provider from the resource name
- – is legal, but whether you can use it depends on how you serve or host the provider binaries; our private registry rejects it
- So: squash it, and prefix the org (c5) so names are unlikely to collide with anyone else's provider

# What it costs you

# The costs

- A build and release pipeline to own
- Go skills on an infrastructure team
- Debugging can cross the HCL/provider boundary
- Schema changes ripple: release the provider before the modules that use it
- Onboarding: one more thing to install and understand
- More pieces, more edge cases: our build never marked the binary executable. That worked for over a year – until a new edge case made it matter

# When not to do this

- The data is simple and stable → `yamldecode` is fine
- One consumer, one format → probably not worth it
- A simple pre-processing script does the job → use the script

# Wrap-up

# What about AI?

- Tools like Claude make writing the Go provider easier
- They also make writing the complex HCL easier – but AI-written HCL is still HCL: just as hard to review and test
- When writing code is cheaper, checking it matters more
- Fine-grained tests are a big plus: each check in the parsing has a test that says exactly what broke
- AI can write the tests too: each case is a YAML string and an expected error, which is easy to review
- A provider gives AI a clear target: a typed schema and a test suite to pass

# Takeaways

- When your HCL starts parsing data, consider a private provider
- Keep the parsing in a plain Go package, and the provider thin
- Wrap the same package in a CLI, to see and debug the data outside Terraform
- Unit test the parsing, and catch bad data at plan time – with the file and line
- Keep the HCL simple, for the people who are not HCL experts
- Go in knowing the costs: you now own a provider

# You might be wondering...

**Why not convert the YAML with a script (Python, jq) and feed OpenTofu JSON?**

At Aurora, our tools – Python included – live in a large Bazel monorepo with its own library management, and none of that is available in our Terraform CI/CD. A script would mean rebuilding all of it just to run there. A provider is one binary that runs wherever OpenTofu does.

And in general: a script is an extra step outside the plan, and its output can drift from the source.

**Does it work with both Terraform and OpenTofu?**

Yes. The demo repo is tested with both: its Makefile uses `tofu` if it is installed, `terraform` otherwise.

# You might be wondering... (continued)

**Why not hide the parsing in a shared module?**

I was a big fan of "data modules": HCL that does all of the parsing and returns a clean data structure, just like the provider does. But the HCL got very messy, and I could not get the fine-grained testing I wanted. Extreme edge cases were hard to test, so it was hard to be sure they were handled.

**Isn't this overkill?**

Maybe – see "When not to do this". For us, the clean parsing was a net positive. And once you have *a* custom provider, the boilerplate is done: when an upstream provider turned out to be missing key features, extending our own provider to fill the gap became a real option.

# You might be wondering... (continued)

**OpenTofu has added a lot: provider-defined functions, `tofu test`... Isn't that enough?**

I have used Terraform since about 2015, when HCL was much more basic: `yamldecode` arrived in 2019, `try()` in 2020. It has grown a lot, but it still has no subroutines, no fine-grained testing, and no debugger I can step through line by line. I wanted a "data provider" for years; SDKv2 (2020) was cumbersome, but the Plugin Framework (1.0 in December 2022) made it easier.

**How much code is it?**

In the demo: about 210 lines of Go for the parsing and checks, 180 for the provider glue, 280 for the tests and 60 for the CLI. That is more than the 76 lines of HCL it replaces – but the HCL version catches less, and every line of the Go can be tested.

# Thank you

Questions?

`https://github.com/craig5/private-data-provider-example`