Istanbul DevFest 07.12.2025

From Terraform Chaos to Terragrunt Order

Ceyda Düzgeç

Ceyda Düzgeç

Solution Architect | Platform Team

CCI

About Me

Education:

- MS in Software Engineering @Boğaziçi
- BSc in Computer Science & Engineering @Sabancı

Certifications:

- AWS Solutions Architect Associate
- HashiCorp Terraform Associate

My Experience with Terragrunt:

- Working with multiple production level projects on Cloud with Terragrunt

Open Source:

- Maintainer of github.com/Hipo/university-domains-list

More on Linkedin

Agenda

01 The Chaos: Pure Terraform Problems

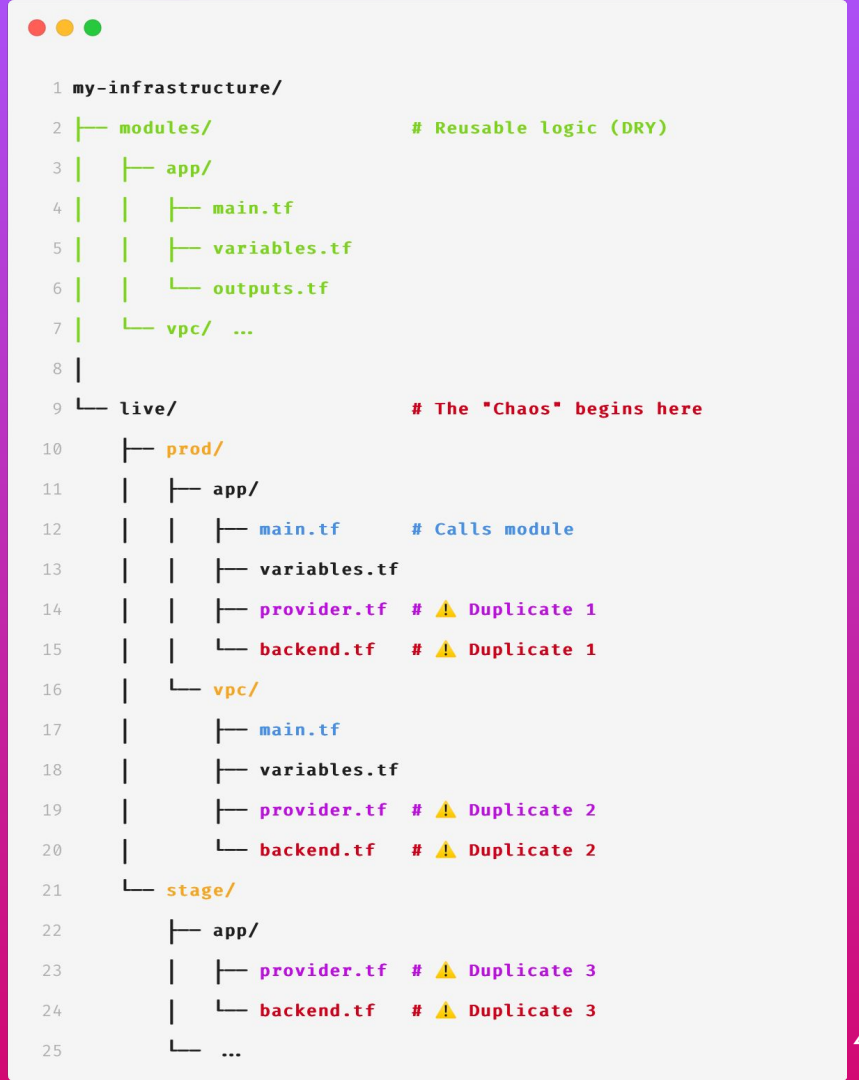
02 The Orchestrator: What is Terragrunt?

03 The Order: 4 Key Solutions

04 The Result: Minimalistic Example

05 Q&A

Terraform Structure



```
# live/prod/app/backend.tf
# WE COPY-PASTE THIS FILE EVERYWHERE ...

terraform {
  backend "gcs" {
    bucket  = "my-project-tf-state"
    # 🔴 PAIN POINT: Hardcoded path manually typed per folder
    prefix  = "prod/app/state"
  }
}
```

Problems with Pure Terraform

01	Code Duplication	Copy-pasting configurations across every environment folder
02	Monolith State	Large state files that are risky to update and slow to plan
03	Dependency Hell	Manually running terraform apply in specific orders (VPC first, then DB, then App)
04	Inconsistent Environments	Drift between environments due to manual variable management

Terragrunt

This is an open source wrapper for Terraform that fills in some gaps in Terraform. You'll see how to use it a bit later in this chapter to deploy versioned Terraform code across multiple environments with minimal copying and pasting.

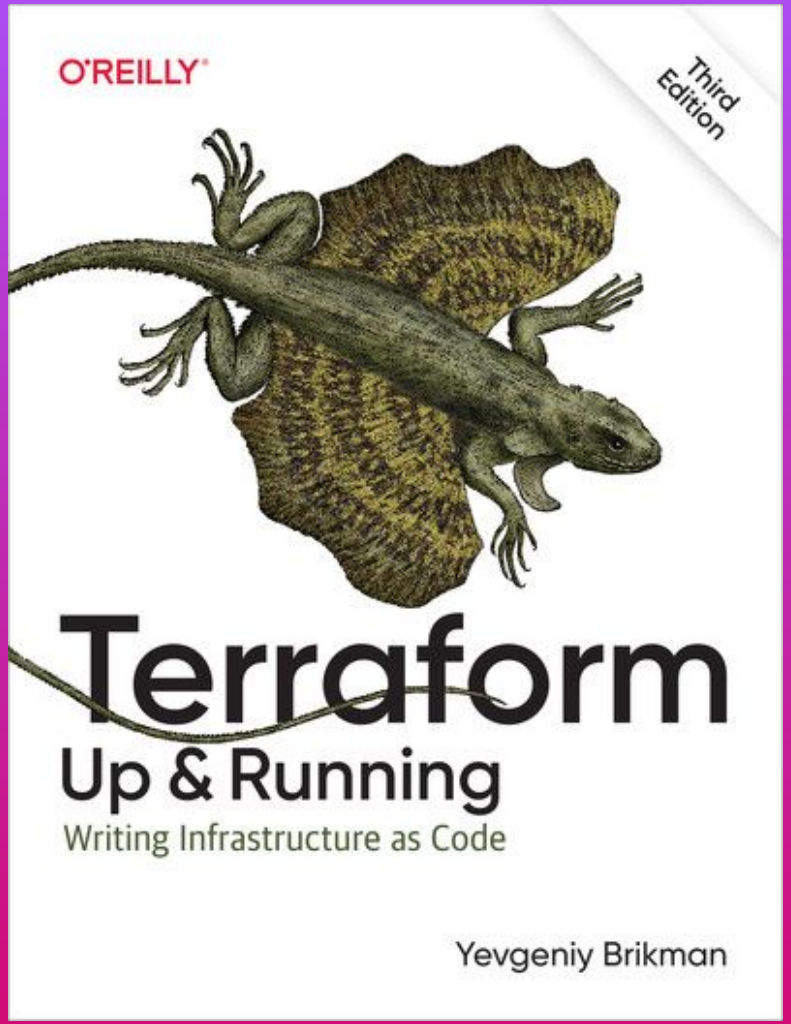
Another option for reducing copy-and-paste is to use Terragrunt, an open source tool that tries to fill in a few gaps in Terraform. Terragrunt can help you keep your entire `backend` configuration DRY (Don't Repeat Yourself) by defining all the basic backend settings (bucket name, region, DynamoDB table name) in one file and automatically setting the `key` argument to the relative folder path of the module.

You'll see an example of how to use Terragrunt in Chapter 10.

Resource dependencies

Breaking the code into multiple folders makes it more difficult to use resource dependencies. If your app code was defined in the same Terraform configuration files as the database code, that app code could directly access attributes of the database using an attribute reference (e.g., access the database address via `aws_db_instance.foo.address`). But if the app code and database code live in different folders, as I've recommended, you can no longer do that.

Solution: One option is to use `dependency` blocks in Terragrunt, as you'll see in Chapter 10. Another option is to use the `terraform_remote_state` data source, as described in the next section.



What is Terragrunt?

A **thin wrapper** that provides extra tools for keeping your configurations DRY, working with multiple modules, and managing remote state.

- ✅ **DRY Backend:** Define state config once in the root
- ✅ **DRY CLI Flags:** Auto-retry, timeouts, and var-files
- ✅ **Auto-Init:** Creates GCS buckets automatically
- ✅ **Dependency Management:** `run-all apply` executes graphs

Terragrunt Structure

Terraform

```
my-infrastructure/
├── modules/                # Reusable logic (DRY)
│   ├── app/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   └── vpc/  ...
│
└── live/                   # The "Chaos" begins here
    ├── prod/
    │   ├── app/
    │   │   ├── main.tf     # Calls module
    │   │   ├── variables.tf
    │   │   ├── provider.tf  # ⚠️ Duplicate 1
    │   │   └── backend.tf   # ⚠️ Duplicate 1
    │   └── vpc/
    │       ├── main.tf
    │       ├── variables.tf
    │       ├── provider.tf  # ⚠️ Duplicate 2
    │       └── backend.tf   # ⚠️ Duplicate 2
    └── stage/
        ├── app/
        │   ├── provider.tf  # ⚠️ Duplicate 3
        │   └── backend.tf   # ⚠️ Duplicate 3
        └── ...
```

Terragrunt

```
my-infrastructure/
│
├── modules/                # Unchanged (Pure Terraform)
│   ├── app/  ...
│   └── vpc/  ...
├── terragrunt.hcl          # 🔥 The Root Parent (Config inherited from here)
└── live/                   # The "Order"
    ├── prod/
    │   ├── app/
    │   │   └── terragrunt.hcl  # Replaces all 4 .tf files!
    │   └── vpc/
    │       └── terragrunt.hcl
    └── stage/
        ├── app/
        │   └── terragrunt.hcl
        └── vpc/
            └── terragrunt.hcl
```

01 Code Duplication

Problem: Copy-pasting Backend/Provider configs
Solving Duplication: Inheritance

```hcl
# root/terragrunt.hcl

# We define the configuration ONCE here.
# (e.g., Common variables, remote state logic)

inputs = {
  org_name = "devfest-gcp"
  region   = "us-east-1"
}
```

```hcl
# live/prod/app/terragrunt.hcl

# 🔗 The Include Block
include "root" {
  path = find_in_parent_folders()
}

terraform {
  source = "../../../modules/app"
}

# ☝️ No backend.tf needed!
# It is inherited automatically.
```

02 Monolith State

Problem:
Managing state files manually,
Monolith risks

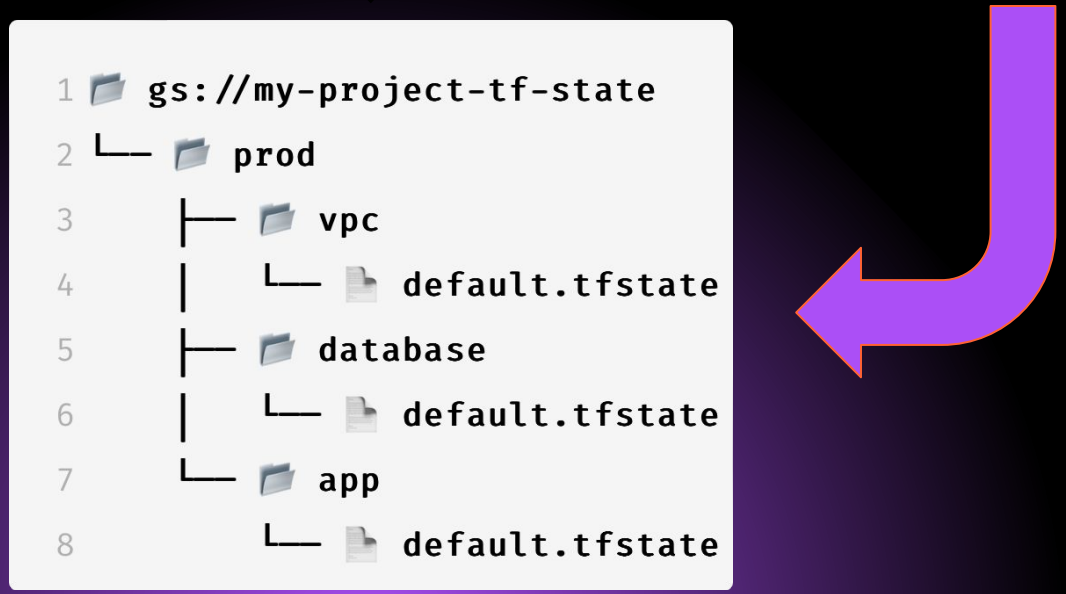
Solving Management:
Dynamic Paths

```
# root/terragrunt.hcl

remote_state {
  backend = "gcs"
  config = {
    bucket  = "my-project-tf-state"
    project = "my-gcp-project-id"
    location = "us-east1"

    # 🪄 SOLUTION: Dynamic State Paths
    # Automatically sets prefix to: "prod/app"
    # based on the folder structure.
    prefix   = "${path_relative_to_include()}"
  }
}
```

```
gs://my-company-tf-state
└── prod
    └── default.tfstate  (⚠ All Resources: VPC + DB + App)
```



03 Dependency Hell

Problem:

Manual deployment orders (VPC -> DB -> App)

Solving Order:

Explicit Dependencies

Command:

terragrunt run-all apply

```
# live/prod/app/terragrunt.hcl

# 🔗 Explicit Dependency
dependency "vpc" {
  config_path = "../vpc"
}

inputs = {
  # Terragrunt grabs the output from VPC
  # and passes it to the App automatically.
  network = dependency.vpc.outputs.network_self_link
}
```

04 Inconsistent Environments

Problem:

Inconsistency between environment versions

Solving Inconsistency:

Code Generation

```
# root/terragrunt.hcl

# ⚙️ Generates provider.tf in EVERY folder
generate "provider" {
  path      = "provider.tf"
  if_exists = "overwrite_terragrunt"
  contents  = <<EOF
provider "google" {
  project = "my-gcp-project-id"
  region  = "us-east1"

  # 🔒 Pinning the version globally!
  version = "~> 5.0"
}
EOF
}
```

The Result: Minimalistic Example

```hcl
# live/prod/app/terragrunt.hcl

# 1️⃣ INHERITANCE
include "root" {
  path = find_in_parent_folders()
}

# 2️⃣ SOURCE
terraform {
  source = "tfr:///terraform-google-modules/vm/google//modules/compute_instance?version=7.0.0"
}

# 3️⃣ DEPENDENCY
dependency "vpc" {
  config_path = "../vpc"
}

# 4️⃣ INPUTS
inputs = {
  # 🔽 INHERITED FROM ROOT (Automatic Merge)
  # region    = "europe-west1"  ⟵ No need to repeat this!

  # 🔽 CHILD SPECIFIC
  machine_type = "e2-medium"

  # 🔽 DYNAMIC DEPENDENCY OUTPUTS
  network    = dependency.vpc.outputs.network_name
  subnetwork = dependency.vpc.outputs.subnetwork_name

  tags = ["http-server"]
}
```

Summary

Feature	The Chaos (Pure Terraform)	The Order (Terragrunt)
Configuration	**Duplicated** in every folder	**Inherited** from Root (DRY)
State Management	**Hardcoded** & Monolithic	**Dynamic** & Isolated
Deployments	**Manual** (Readme-driven)	**Automated** (run-all apply)
Consistency	**Drifting** Provider Versions	**Generated** & Identical

Q&A

Any questions?

Slides

Linkedin