



Toolsday Presentation by Konstantin Gredeskoul

October 7, 2025

Terraform is Better as a TerraGrunt Sandwich

What We'll Cover Today

- **Infrastructure as Code:** Terraform fundamentals
- **The Problem:** Challenges with pure Terraform projects
- **The Solution:** Enter Terragrunt
- **Practical Example:** Real-world refactoring
- **When to Use:** Benefits vs. overkill scenarios



Infrastructure as Code: A Brief History



Configuration Management vs. Infrastructure Provisioning

2010s

Configuration Management

- Chef, Puppet, Ansible
- Manage software on existing servers
- **Mutable infrastructure**
- Centralized approach
- Potential config drift over time

2018+

Infrastructure Provisioning

- Terraform, CloudFormation
- Create cloud resources
- **Immutable infrastructure**
- Declarative approach
- Version-controlled state



Terraform: The Good Parts

Declarative Language (HCL)

```
resource "aws_instance" "web" {  
  ami          = "ami-0c55b159cbfafa1f0"  
  instance_type = "t2.micro"  
  
  tags = {  
    Name = "WebServer"  
  }  
}
```

- Multi-cloud support via providers
- Resource dependency management
- State tracking
- Modular architecture



The Terraform Challenges

Problem #1: No Standard Repository Structure

Everyone organizes differently:

```
project-a/      project-b/      project-c/
├─ main.tf      ├─ infrastructure/  ├─ modules/
├─ variables.tf │├─ modules/        ├─ environments/
└─ outputs.tf   └─ environments/ └─ terraform/
```

No conventions = confusion and inconsistency



The Terraform Challenges

Problem #2: State File Isolation

Each directory maintains its own state:

```
app-infra/  
├─ main.tf  
└─ terraform.tfstate    # State A  
  
database/  
├─ main.tf  
└─ terraform.tfstate    # State B
```

Requires manual dependency ordering:

1. Deploy database first
2. Get outputs manually
3. Deploy app infrastructure
4. Hope nothing breaks



The Terraform Challenges

Problem #3: Code Duplication (Not DRY)

Need the same infrastructure in dev, staging, and production?

```
environments/  
├─ dev/  
│   ├── main.tf          # Copy-paste  
│   ├── variables.tf     # Copy-paste  
│   └─ backend.tf        # Copy-paste  
├─ staging/  
│   ├── main.tf          # Copy-paste  
│   ├── variables.tf     # Copy-paste  
│   └─ backend.tf        # Copy-paste  
└─ production/  
    ├── main.tf          # Copy-paste  
    ├── variables.tf     # Copy-paste  
    └─ backend.tf        # Copy-paste
```



The Terraform Challenges

Problem #4: Backend Configuration Duplication

Every module needs backend configuration:

```
terraform {  
  backend "s3" {  
    bucket      = "my-terraform-state"  
    key         = "app/terraform.tfstate"  
    region      = "us-east-1"  
    encrypt     = true  
    dynamodb_table = "terraform-locks"  
  }  
}
```

Repeated in **every single directory** with slight variations



The Terraform Challenges

Problem #5: Circular Dependencies

```
Module A needs output from Module B  
Module B needs output from Module C  
Module C needs output from Module A ❌
```

Terraform can't handle this automatically

No built-in dependency resolution between separate state files



Enter Terragrunt



A Thin Wrapper Around Terraform

Terragrunt adds:

- **DRY principles** for backend configuration
- **Dependency management** between modules
- **Automatic state management**
- **Standard repository structure**
- **Remote state handling**
- **Before/after hooks**



Terragrunt Core Concepts

The Unit

A **unit** is the smallest deployable entity:

- One directory with `terragrunt.hcl`
- References a Terraform module
- Has its own inputs and dependencies
- Atomic and hermetic

The Stack

A **stack** is a collection of units:

- Typically represents an environment
- Can be deployed together with `run-all`
- Managed via dependency graph (DAG)



Terragrunt: DRY Backend Configuration

Root `terragrunt.hcl`

```
remote_state {  
  backend = "s3"  
  config = {  
    bucket      = "my-terraform-state"  
    key         = "${path_relative_to_include()}/terraform.tfstate"  
    region      = "us-east-1"  
    encrypt     = true  
    dynamodb_table = "terraform-locks"  
  }  
}
```

Child unit inherits this automatically

```
include "root" {  
  path = find_in_parent_folders()  
}
```



Terragrunt: Dependency Management

```
dependency "database" {  
  config_path = "../database"  
}  
  
dependency "vpc" {  
  config_path = "../vpc"  
}  
  
inputs = {  
  db_endpoint = dependency.database.outputs.endpoint  
  vpc_id      = dependency.vpc.outputs.vpc_id  
}
```

Terragrunt handles the execution order automatically!



Standard Repository Structure

The Terragrunt Way

```
.
├── modules/                # Reusable Terraform modules
│   ├── app/
│   ├── database/
│   └── vpc/
└── live/                   # Environment-specific configurations
    ├── terragrunt.hcl      # Root configuration
    ├── prod/
    │   ├── us-east-1/
    │   │   ├── app/
    │   │   │   └── terragrunt.hcl
    │   │   └── database/
    │   │       └── terragrunt.hcl
    └── dev/
        └── us-west-2/
            └── app/
```



Practical Example: Before Terragrunt

Scenario

Deploy a Python web application with PostgreSQL database across:

- Development (us-west-2)
- Staging (us-east-1)
- Production (us-east-1)



Pure Terraform Structure

```
terraform-project/  
├── modules/  
│   ├── web-app/  
│   │   ├── main.tf  
│   │   ├── variables.tf  
│   │   └── outputs.tf  
│   └── postgresql/  
│       ├── main.tf  
│       ├── variables.tf  
│       └── outputs.tf  
└── environments/  
    ├── dev/  
    │   ├── main.tf  
    │   ├── variables.tf  
    │   ├── backend.tf          # Duplicated  
    │   └── terraform.tfvars
```



Pure Terraform: The Module

modules/web-app/main.tf

```
variable "app_name" {}
variable "docker_image" {}
variable "db_endpoint" {}
variable "environment" {}

resource "aws_ecs_cluster" "app" {
  name = "${var.app_name}-${var.environment}"
}

resource "aws_ecs_service" "app" {
  name           = "${var.app_name}-service"
  cluster        = aws_ecs_cluster.app.id
  task_definition = aws_ecs_task_definition.app.arn
  desired_count  = 1

  load_balancer {
```



Pure Terraform: Environment Config

environments/prod/main.tf

```
terraform {  
  backend "s3" {  
    bucket = "mycompany-terraform-state"  
    key    = "prod/terraform.tfstate"  
    region = "us-east-1"  
  }  
}  
  
module "database" {  
  source = "../../modules/postgresql"  
  
  db_name      = "myapp"  
  environment  = "prod"  
  region       = "us-east-1"  
}
```

Same code repeated for dev and staging with tiny differences!



Pure Terraform: The Pain Points

Deployment Process

```
# For each environment:
cd environments/prod

# Deploy database first
terraform init
terraform plan -target=module.database
terraform apply -target=module.database

# Wait... get outputs manually

# Then deploy app
terraform plan -target=module.web_app
terraform apply -target=module.web_app
```

Manual dependency management 🙄 Lots of copy-paste code 🙄 No DRY backend config 🙄



After Terragrunt: Repository Structure

```
terragrunt-project/
├── modules/                                # Terraform modules (unchanged)
│   ├── web-app/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   └── postgresql/
│       ├── main.tf
│       ├── variables.tf
│       └── outputs.tf
└── live/                                   # Terragrunt configurations
    ├── terragrunt.hcl                     # Root config (DRY!)
    ├── prod/
    │   └── us-east-1/
    │       ├── database/
    │       └── terragrunt.hcl # 20 lines
```



Terragrunt: Root Configuration

live/terragrunt.hcl

```
# DRY backend configuration
remote_state {
  backend = "s3"
  generate = {
    path      = "backend.tf"
    if_exists = "overwrite"
  }
  config = {
    bucket      = "mycompany-terraform-state"
    key         = "${path_relative_to_include()}/terraform.tfstate"
    region      = "us-east-1"
    encrypt     = true
    dynamodb_table = "terraform-locks"
  }
}
```



Terragrunt: Database Unit

live/prod/us-east-1/database/terragrunt.hcl

```
include "root" {  
  path = find_in_parent_folders()  
}  
  
terraform {  
  source = "../../../modules/postgresql"  
}  
  
inputs = {  
  db_name      = "myapp"  
  environment  = "prod"  
  region       = "us-east-1"  
  instance_class = "db.t3.large"  
}
```

That's it! 15 lines vs. 100+ lines of duplicated Terraform



Terragrunt: Web App Unit (With Dependencies)

[live/prod/us-east-1/web-app/terragrunt.hcl](#)

```
include "root" {  
  path = find_in_parent_folders()  
}  
  
terraform {  
  source = "../../../modules/web-app"  
}  
  
dependency "database" {  
  config_path = "../database"  
  
  mock_outputs = {  
    endpoint = "mock-db-endpoint.rds.amazonaws.com"  
  }  
}
```



Terragrunt: Simplified Deployment

Deploy Everything

```
cd live/prod/us-east-1  
terragrunt run-all apply
```

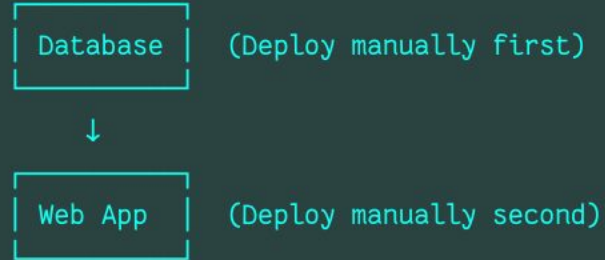
Terragrunt automatically:

1. Analyzes dependency graph
2. Deploys database first
3. Waits for outputs
4. Deploys web-app with database endpoint
5. Handles errors and retries



Terragrunt: Dependency Visualization

Pure Terraform (Manual)



With Terragrunt (Automatic)



Code Comparison: Lines of Code

Pure Terraform Approach

```
Backend configs:    3 × 15 lines = 45 lines
Main configs:      3 × 50 lines = 150 lines
Variables:         3 × 30 lines = 90 lines
Total:                                285 lines (highly duplicated)
```

Terragrunt Approach

```
Root config:                30 lines
Database units:    3 × 15 lines = 45 lines
Web app units:     3 × 25 lines = 75 lines
Total:                                150 lines (DRY, no duplication)
```

~47% reduction in code! 🎉



When Terragrunt Shines

Maximum Benefits When You Have

1. **Multiple environments** (dev, staging, prod)
2. **Complex dependencies** between infrastructure components
3. **Multiple regions** or cloud accounts
4. **Large teams** needing consistent structure
5. **Frequent infrastructure changes** requiring maintainability
6. **Shared modules** across many projects



When Terragrunt Might Be Overkill 🤔

Consider Plain Terraform If

1. **Single environment** deployment
2. **Simple, flat infrastructure** (no dependencies)
3. **Small project** with <5 resources
4. **One-time deployment** with no updates
5. **Solo developer** with no collaboration needs
6. **Proof of concept** or learning Terraform basics

Start simple, add Terragrunt when complexity grows



Key Benefits Summary

1. **DRY Principle:** Write backend configuration once, inherit everywhere
2. **Dependency Management:** Automatic execution order based on dependency graph
3. **Standard Structure:** Clear conventions for organizing infrastructure code
4. **Reduced Code Duplication:** ~40-60% less code to maintain
5. **Version Control:** Pin module versions per environment
6. **Mock Outputs:** Test and plan without deploying dependencies
7. **Bulk Operations:** Deploy/destroy entire stacks with run-all
8. **Hooks:** Execute custom commands before/after Terraform



Common Issues Terragrunt Solves

Without Terragrunt ❌

- Copy-paste backend configs
- Manual dependency ordering
- Inconsistent repo structures
- Hard to scale across teams
- Circular dependency deadlocks
- Error-prone deployments
- State file confusion

With Terragrunt ✅

- Backend config inheritance
- Automatic dependency DAG
- Standard folder structure
- Team-friendly conventions
- Dependency cycle detection
- Orchestrated deployments
- Clear state organization



Terragrunt Advanced Features

Auto-init

Automatically runs `terraform init`

Mock Outputs

Test without deploying dependencies:

```
dependency "vpc" {  
  config_path = "../vpc"  
  mock_outputs = {  
    vpc_id = "vpc-mock123"  
  }  
}
```

Before/After Hooks

```
terraform {  
  before_hook "validate" {  
    commands = ["apply", "plan"]  
    execute   = ["terraform", "fmt", "-check"]  
  }  
}
```



Migration Strategy

Incremental Adoption Path

1. **Start small:** Add root `terragrunt.hcl` with backend config
2. **Refactor modules:** Extract reusable Terraform modules
3. **Create units:** Convert environments to Terragrunt units
4. **Add dependencies:** Define relationships between units
5. **Test thoroughly:** Validate with `plan` before `apply`
6. **Train team:** Share knowledge and best practices

No need to migrate everything at once!



Terragrunt + Terraform = ❤️

Best Practices

- **Keep modules pure:** Terraform modules should be Terragrunt-agnostic
- **Use mock outputs:** Enable testing without dependencies
- **Version everything:** Pin module versions explicitly
- **Automate CI/CD:** Use `terragrunt run-all plan` in pipelines
- **Document dependencies:** Make DAG clear for team members
- **Start with conventions:** Use standard folder structure from day one



Resources & Documentation

Official Documentation

- Terragrunt: terragrunt.gruntwork.io
- Terraform: terraform.io

Recommended Reading

- Terragrunt: How to Keep Your Code DRY
- Terragrunt vs Terraform
- Understanding Terraform



The Bottom Line

Terraform is great for infrastructure provisioning

Terragrunt makes it **maintainable**, **scalable**, and **team-friendly**

Terraform is better as a TerraGrunt sandwich! 🥪





Thanks

Any Questions?
Comments?
Feedback?



By Konstantin Gredeskoul for Fractional.AI