Every organization we work with eventually outgrows their single-account, single-state-file Terraform setup. The inflection point is predictable: the state file takes 5 minutes to plan, someone accidentally destroys a production resource while working on staging, and the security team starts asking why developers have admin access to the production account. This post covers how we structure Terraform for multi-account AWS environments after doing this migration dozens of times.

The Problem with Single-Account Terraform

A single AWS account with one Terraform state file is where everyone starts. It is fine for a team of three running a handful of services. It stops working when:

- **Blast radius is unlimited.** A bad `terraform apply` can take down production, staging, and shared infrastructure in one shot.
- **IAM permissions become impossible.** You cannot give a developer access to deploy their staging app without also giving them access to production databases.
- **State file contention.** Two teams running `terraform plan` at the same time cause state lock conflicts. One waits, the other gets stale output.
- **Cost attribution is guesswork.** When everything is in one account, figuring out which team or project is spending what requires tagging discipline that nobody maintains.

The answer is AWS Organizations with a well-designed account structure.

AWS Organizations Structure That Works

We deploy a standard account topology:

```
Root (Management Account)
├── Security OU
│   ├── security-audit
│   └── security-logs (CloudTrail, Config, GuardDuty)
├── Infrastructure OU
│   ├── shared-services (CI/CD, artifact registries, DNS)
│   └── networking (Transit Gateway, VPN, Direct Connect)
├── Workloads OU
│   ├── dev
│   ├── staging
│   └── production
└── Sandbox OU
    └── developer-sandboxes
```

Key rules:

- The management account runs Organizations, SCPs, and SSO. Nothing else. No workloads, no infrastructure.
- CloudTrail and AWS Config logs go to the security-logs account. Nobody except the security team has write access.
- Shared services (ECR, shared Route 53 zones, CI/CD) live in their own account with cross-account access policies.
- Each workload environment is a separate account. This is your blast radius boundary.

Our Terragrunt Project Structure

We use Terragrunt on top of Terraform for every multi-account setup. Raw Terraform with workspaces does not scale. Workspaces share the same backend config and make it too easy to apply production changes from a staging workspace.

Here is our directory structure:

```
infrastructure/
├── terragrunt.hcl                    # Root config
├── _modules/                         # Terraform modules (versioned
separately)
├── accounts/
│   ├── shared-services/
│   │   ├── account.hcl
│   │   ├── us-east-1/
│   │   │   ├── region.hcl
│   │   │   ├── ecr/
│   │   │   │   └── terragrunt.hcl
│   │   │   ├── vpc/
│   │   │   │   └── terragrunt.hcl
│   │   │   └── dns/
│   │   │       └── terragrunt.hcl
│   │   └── eu-west-1/
│   │       ├── region.hcl
│   │       └── vpc/
│   │           └── terragrunt.hcl
│   ├── production/
│   │   ├── account.hcl
│   │   ├── us-east-1/
│   │   │   ├── region.hcl
│   │   │   ├── vpc/
│   │   │   │   └── terragrunt.hcl
│   │   │   ├── eks/
│   │   │   │   └── terragrunt.hcl
│   │   │   ├── rds/
│   │   │   │   └── terragrunt.hcl
│   │   │   └── elasticache/
│   │   │       └── terragrunt.hcl
```

The root `terragrunt.hcl` handles provider configuration and state backend:

```
# infrastructure/terragrunt.hcl

locals {
  account_vars =
read_terragrunt_config(find_in_parent_folders("account.hcl"))
  region_vars  =
read_terragrunt_config(find_in_parent_folders("region.hcl"))

  account_id   = local.account_vars.locals.account_id
  account_name = local.account_vars.locals.account_name
  aws_region   = local.region_vars.locals.aws_region
}

generate "provider" {
  path      = "provider.tf"
  if_exists = "overwrite_terragrunt"
  contents  = <<EOF
provider "aws" {
  region = "${local.aws_region}"

  assume_role {
    role_arn =
"arn:aws:iam::${local.account_id}:role/TerraformExecutionRole"
  }

  default_tags {
    tags = {
      ManagedBy   = "terraform"
      Account     = "${local.account_name}"
      Region      = "${local.aws_region}"
      Repository  = "infrastructure"
    }
  }
}
EOF
}

remote_state {
  backend = "s3"
  generate = {
```

```
    path      = "backend.tf"
    if_exists = "overwrite_terragrunt"
  }
  config = {
    bucket         = "terraform-
state-${local.account_id}-${local.aws_region}"
    key            = "${path_relative_to_include()}/terraform.tfstate"
    region         = local.aws_region
    encrypt        = true
    dynamodb_table = "terraform-locks"

    s3_bucket_tags = {
      ManagedBy = "terragrunt"
    }
  }
}
```

Each `account.hcl` is simple:

```
# accounts/production/account.hcl
locals {
  account_id   = "123456789012"
  account_name = "production"
  environment  = "prod"
}
```

And each stack's `terragrunt.hcl` references the module:

```
# accounts/production/us-east-1/vpc/terragrunt.hcl
include "root" {
  path = find_in_parent_folders()
}

terraform {
  source = "git::git@github.com:org/terraform-modules.git//vpc?
ref=v2.3.1"
```

```
}

inputs = {
  vpc_cidr             = "10.10.0.0/16"
  availability_zones   = ["us-east-1a", "us-east-1b", "us-east-1c"]
  private_subnets      = ["10.10.0.0/19", "10.10.32.0/19",
"10.10.64.0/19"]
  public_subnets       = ["10.10.96.0/24", "10.10.97.0/24",
"10.10.98.0/24"]
  enable_nat_gateway   = true
  single_nat_gateway   = false
}
```

Why this structure works:

- **One state file per stack.** The VPC state is separate from the EKS state. A bad apply to EKS cannot touch the VPC.
- **Account-level and region-level configs cascade.** You define the account ID once and it flows to every stack.
- **Module versions are pinned.** That `?ref=v2.3.1` means production uses a tested, tagged version. Not `main`. Not `latest`.

Module Design Principles

After writing hundreds of Terraform modules, these are the rules we follow:

1. Validate inputs. Do not trust callers to provide sane values.

```
variable "vpc_cidr" {
  type        = string
  description = "CIDR block for the VPC"

  validation {
    condition     = can(cidrhost(var.vpc_cidr, 0)) && tonumber(split("/",
var.vpc_cidr)[1]) >= 16 && tonumber(split("/", var.vpc_cidr)[1]) <= 24
```

```
    error_message = "VPC CIDR must be a valid CIDR block with a prefix
between /16 and /24."
  }
}
```

2. Output everything downstream modules might need. VPC ID, subnet IDs, security group IDs, ARNs. If someone might need it as an input to another module, output it.

3. Use `for_each`, not `count`. Count-based resources break when you remove an item from the middle of a list. Every resource shifts index and Terraform wants to destroy and recreate half your infrastructure. `for_each` with maps is stable.

4. Semantic versioning on modules. Breaking changes get a major version bump. New features get minor. Bug fixes get patch. Tag releases. Use a changelog.

PR-Based Workflows with Atlantis

We run Atlantis for every client. Terraform should never be applied from a developer's laptop. The workflow:

1. Developer opens a PR with infrastructure changes.
2. Atlantis runs `terraform plan` automatically and posts the output as a PR comment.
3. A second engineer reviews the plan output (not just the code diff).
4. On approval and merge, Atlantis runs `terraform apply`.

Our `atlantis.yaml` repo config:

```
version: 3
automerge: false
parallel_plan: true
parallel_apply: false
```

```yaml
projects:
  - name: production-vpc
    dir: accounts/production/us-east-1/vpc
    workflow: terragrunt
    autoplan:
      when_modified:
        - "*.hcl"
        - "*.tf"
      enabled: true
    apply_requirements:
      - approved
      - mergeable

  - name: production-eks
    dir: accounts/production/us-east-1/eks
    workflow: terragrunt
    autoplan:
      when_modified:
        - "*.hcl"
        - "*.tf"
      enabled: true
    apply_requirements:
      - approved
      - mergeable

workflows:
  terragrunt:
    plan:
      steps:
        - env:
            name: TERRAGRUNT_TFPATH
            command: 'which terraform'
        - run: terragrunt plan -no-color -out=$PLANFILE
    apply:
      steps:
        - run: terragrunt apply -no-color $PLANFILE
```

`parallel_apply: false` **is deliberate.** You want plans to run in parallel for speed, but applies should be sequential to avoid race conditions between dependent

stacks.

Infracost Integration

Every PR gets a cost estimate before anyone clicks approve. Infracost runs alongside Atlantis and comments on the PR with the monthly cost impact:

```
## 💰 Infracost estimate

Monthly cost will increase by $342 (from $12,450 to $12,792)

| Resource                    | Monthly Cost |
|-----------------------------|--------------|
| aws_instance.worker (x3)    | +$280        |
| aws_ebs_volume.data         | +$62         |
```

This catches the "I just need a bigger instance" PRs that would otherwise slip through. We have seen a single PR that would have added $8,000/month to a client's bill. Infracost caught it. The engineer resized.

Policy Enforcement in CI

We run two layers of policy checks:

Checkov for general security and compliance scanning:

```
# .github/workflows/terraform-checks.yml
- name: Run Checkov
  uses: bridgecrewio/checkov-action@v12
  with:
    directory: accounts/
    framework: terraform
    skip_check: CKV_AWS_144,CKV_AWS_145  # Skip cross-region replication
checks for non-prod
```

```
    output_format: sarif
    soft_fail: false
```

OPA (Open Policy Agent) for custom organizational policies. Checkov covers the generic cases, but every organization has specific rules. Examples we have written:

- All RDS instances must use encryption at rest with a customer-managed KMS key.
- No security group may allow ingress from `0.0.0.0/0` on any port except 443.
- All S3 buckets must have versioning enabled.
- EC2 instances in production must use instance types from an approved list.

These policies run as `conftest` checks in the PR pipeline against the Terraform plan JSON output.

Common Anti-Patterns We See in Audits

1. Hardcoded values everywhere. Account IDs, AMI IDs, CIDR blocks embedded in resources instead of variables or data sources. This makes the code impossible to reuse across environments.

2. Monolith state files. One state file with 500+ resources. Plans take 10 minutes. Applies are terrifying. Break it up by logical boundary (networking, compute, data, application).

3. No modules. Copy-pasted resource blocks across environments. When a security patch requires changing an `aws_security_group` configuration, you are updating it in 12 places and missing 3.

4. `count` for everything. As covered above, `count` creates indexed resources. Removing item 2 from a list of 5 causes items 3, 4, and 5 to shift. Use `for_each` with maps.

5. Terraform apply from laptops. No audit trail, no peer review, no consistency. Use Atlantis, Terraform Cloud, Spacelift, or any CI/CD pipeline. Just not laptops.

6. `terraform taint` as a debugging strategy. If you are tainting resources regularly, your module has a design problem. Fix the root cause.

7. Ignoring state drift. If you are not running periodic `terraform plan` to detect drift, you do not actually have infrastructure as code. You have infrastructure as suggestion.

Migrating from ClickOps to IaC

The hardest part of adopting Terraform is not writing new infrastructure. It is importing existing manually-created resources. Our migration workflow:

1. **Inventory.** Use AWS Config or Steampipe to enumerate all resources in the account. Export to a spreadsheet.
2. **Prioritize.** Start with networking (VPC, subnets, route tables), then security (IAM, security groups), then compute and data.
3. **Write the module first.** Do not import into a blank file. Write the Terraform code that represents the desired state, then import.
4. **Import.** Use `terraform import` for each resource. As of Terraform 1.5+, use `import` blocks in HCL for a declarative approach:

```
import {
  to = aws_vpc.main
  id = "vpc-0a1b2c3d4e5f"
}
```

5. **Plan and diff.** After import, run `terraform plan`. The plan should show zero changes if your code matches reality. If it shows changes, decide: does reality need to match your code, or does your code need to match reality?

6. **Iterate.** Do not try to import an entire account in one sprint. Import one service at a time, validate, and move on. We typically budget 2-4 weeks per account depending on complexity.

State surgery (`terraform state mv`, `terraform state rm`) is sometimes necessary when restructuring modules. Always take a state backup before any state operation:

```
terraform state pull > state-backup-$(date +%Y%m%d-%H%M%S).json
```

What We Recommend

If you are starting fresh: use Terragrunt from day one, set up Atlantis, enforce policies in CI, and structure your accounts properly. The upfront cost is a few days of setup. The payoff is years of manageable, auditable infrastructure.

If you are migrating from ClickOps or a messy Terraform setup: do it incrementally. Import one service at a time. Do not try to boil the ocean. And resist the urge to refactor while importing - get it into state first, then refactor in a separate PR.

We have helped dozens of organizations through this migration. The tooling is mature. The patterns are proven. The only variable is the discipline to follow them consistently. If your team needs help structuring or migrating your Terraform estate, contact us.

`terraform` `terragrunt` `aws` `multi-account` `infrastructure-as-code`

Book a free 30-minute architecture review. We'll look at your specific setup and give you honest recommendations.

Book Architecture Review →

← Back to all articles