Mastering Terraform Variables & Outputs

Variables and outputs form the backbone of flexible, reusable Terraform configurations. Understanding how to leverage these features transforms static infrastructure code into dynamic, environment-aware deployments that scale across teams and projects.

Terraform Variables: The Foundation

What Are Variables?

Variables in Terraform store dynamic values that can be reused throughout your configuration. They enable you to avoid hardcoding values, making your infrastructure code adaptable across different environments, regions, and use cases.

You can override variable values using multiple methods: command-line flags, environment variables, or dedicated .tfvars files. This flexibility is essential for CI/CD pipelines and multi-environment deployments.

Common Use Cases

- Environment-specific configurations (dev, staging, prod)
- Regional deployments with different parameters
- Reusable modules with customizable inputs
- Team collaboration with standardized values
- Automated deployments in CI/CD workflows

Variable Types in Terraform

string

Text values like instance types, region names, or identifiers

```
type = string
```

number

Numeric values for counts, ports, or capacity settings

```
type = number
```

bool

True/false flags for feature toggles or conditionals

```
type = bool
```

list

Ordered collections of values, like availability zones or CIDR blocks

```
type = list(string)
```

map

Key-value pairs for tags, labels, or configuration dictionaries

```
type = map(string)
```

object

Complex structures with multiple typed attributes for advanced configs

```
type = object({...})
```

Basic Variable Implementation

Declaring and Using Variables

Here's a practical example showing how to declare a variable with a default value and reference it in your resource configuration:

```
variable "instance_type" {
  type        = string
  default     = "t2.micro"
  description = "EC2 instance type for web servers"
}
```

Reference the variable in your resources using the `var` prefix:

```
resource "aws_instance" "web" {
  instance_type = var.instance_type
  ami           = "ami-0c55b159cbfafe1f0"
}
```

This pattern enables you to change the instance type across your entire infrastructure by modifying a single value, whether through CLI overrides, environment variables, or .tfvars files.

Advanced Variable Features

Validation Rules

Terraform's validation blocks ensure only acceptable values enter your infrastructure, catching configuration errors before resources are provisioned. This is crucial for maintaining standards and preventing costly mistakes.

```
variable "env" {
  type = string
  validation {
    condition = contains(
      ["dev", "staging", "prod"],
      var.env
    )
    error_message = "Environment must be dev, staging, or prod."
  }
}
```

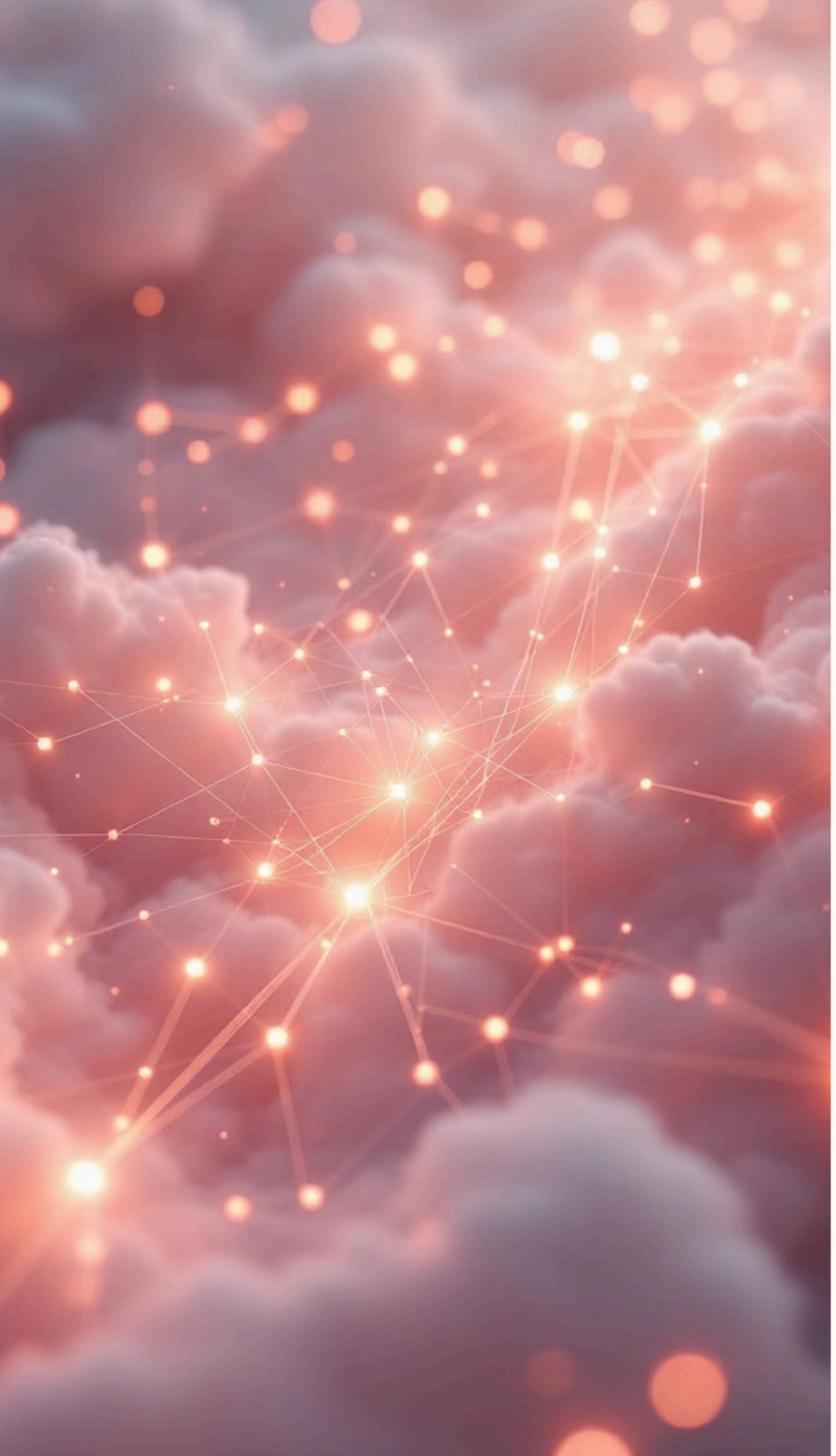
Sensitive Variables

Marking variables as sensitive prevents their values from appearing in CLI output or logs, protecting passwords, API keys, and other confidential data throughout your deployment process.

```
variable "db_password" {
  type      = string
  sensitive = true
}
```

Terraform will mask these values in all outputs, but they're still available for resource configuration internally.

Pro Tip: Always validate critical variables like environment names, instance sizes, and region values to catch configuration drift early in your deployment pipeline.



Outputs: Exposing Infrastructure Information

01

Define What to Expose

Choose which resource attributes should be accessible after deployment — IP addresses, DNS names, resource IDs, or connection strings.

02

Create Output Blocks

Declare output blocks with descriptive names and value references to specific resource attributes you want to expose.

03

Apply Configuration

Run terraform apply to provision resources and automatically display output values upon completion.

04

Query Anytime

Use terraform output to retrieve values later for automation, documentation, or sharing with team members.

Output Implementation & Use Cases

Basic Output Configuration

```
output "public_ip" {
  value       = aws_instance.web.public_ip
  description = "Public IP address of web server"
}
```

Sensitive Outputs

Just like variables, outputs can be marked sensitive to hide confidential information from logs and terminal output:

```
output "db_password" {
  value     = var.db_password
  sensitive = true
}
```

CI/CD Integration

Outputs feed critical values into deployment pipelines, enabling automated testing and downstream configurations.

Team Communication

Share connection strings, endpoints, and resource identifiers with developers and operators efficiently.

Module Composition

Pass data between modules, creating composable infrastructure patterns that scale across projects.

Module Communication Patterns

Modules communicate through a clear contract: inputs go in, outputs come out. This pattern enables reusable infrastructure components that can be composed into larger systems while maintaining clean boundaries and dependencies.

Root Module Configuration

```
module "network" {
  source = "./modules/network"
  cidr   = var.vpc_cidr
  env    = var.environment
}

output "vpc_id" {
  value = module.network.vpc_id
}
```

Child Module Outputs

```
output "vpc_id" {
  value       = aws_vpc.main.id
  description = "VPC identifier for reference by other modules"
}

output "subnet_ids" {
  value = aws_subnet.private[*].id
}
```

This approach creates a data flow where the network module provisions infrastructure and exposes key identifiers that other modules or root configurations can consume. The pattern scales beautifully across complex multi-tier architectures.

Using .tfvars Files: Best Practices

Why .tfvars Files?

Separating variable values into dedicated files provides clean environment management, version control safety, and clear deployment targeting. This approach is industry standard for production Terraform workflows.

Each environment gets its own file with appropriate values, while your main configuration remains environment-agnostic and reusable.

Example Structure

dev.tfvars

```
instance_type = "t2.micro"
env           = "dev"
db_size       = "db.t3.small"
enable_backup = false
```

prod.tfvars

```
instance_type = "t3.large"
env           = "prod"
db_size       = "db.r5.xlarge"
enable_backup = true
```

1

Development

```
terraform plan \
  -var-file=dev.tfvars
```

2

Staging

```
terraform plan \
  -var-file=staging.tfvars
```

3

Production

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

Key Takeaways

1 Master Variable Types

Use the right type for each use case — strings for identifiers, objects for complex configs, and lists for collections. Type safety prevents configuration errors.

2 Implement Validation

Add validation rules to critical variables to catch mistakes early. Protect your infrastructure from invalid configurations before they're applied.

3 Leverage Outputs

Expose the right information through outputs for automation, team collaboration, and module composition. Outputs are your infrastructure API.

4 Organize with .tfvars

Use dedicated variable files for each environment to maintain clean separation, enable version control, and streamline deployments across your infrastructure lifecycle.

These patterns form the foundation of scalable, maintainable infrastructure as code. Apply them consistently to build robust Terraform configurations that serve your organization's needs.