Terraform Lifecycle Configuration

Controlling How Resources Are Created, Updated, and Destroyed

Terraform-intermediate-Day1-Module-4

MODULE 4

Terraform Lifecycle Configuration

Controlling How Resources Are Created, Updated, and Destroyed

Learning Objectives

By the end of this module, you will be able to:

- *Use create_before_destroy for zero-downtime updates*
- *Use prevent_destroy to protect critical resources*
- *Use ignore_changes to handle external modifications*
- *Decide when and how to combine lifecycle settings*

📌 This module teaches how to make Terraform safer, smarter, and production-ready

⚠️ *The Challenge*

Why Do We Need Lifecycle Configuration?

Scenario: Managing Real Infrastructure

Your team manages:

- 🔐 Security Groups attached to running EC2 instances
- 🗄️ S3 buckets storing critical business data
- 🏷️ EC2 instances where AWS adds automatic tags

🚨 *The Problems*

1️⃣ *Replacement Order Issues*

Terraform tries to **destroy first**, but AWS blocks deletion of attached security groups → Deployment fails

2️⃣ *Accidental Deletion*

Someone runs terraform destroy → Critical database or bucket deleted

3️⃣ *Fighting External Tools*

Security scanners add tags → Terraform tries to remove them every run

💡 We need fine-grained control over how Terraform handles resource lifecycle

🧩 *The Solution*

The lifecycle Block

Terraform provides a special **meta-argument** called:

lifecycle

It lets you control **how Terraform treats a resource** during changes.

🔧 *What You Can Control*

Setting	Purpose
create_before_destroy	Avoid downtime during replacements
prevent_destroy	Protect critical resources
ignore_changes	Ignore externally modified attributes

- 📍 The lifecycle block is placed **inside a resource block**
- 📍 Works with **any Terraform resource**

💻 *Syntax*

Lifecycle Block Syntax

```
resource "aws_instance" "web" {
  ami           = "ami-12345678"
  instance_type = "t3.micro"

  lifecycle {
    create_before_destroy = true
  }
}
```

✓ Lifecycle rules are defined **inside the resource**

✓ You can combine multiple lifecycle settings

✓ Helps Terraform behave **safely in production**

🚀 *Zero-Downtime Updates*

create_before_destroy: Zero-Downtime Updates

Default Terraform Behavior (BAD for production)

01

Destroy old resource

02

Create new resource

⛔ Causes downtime gap

With create_before_destroy = true

01

Create new resource

02

Switch dependencies

03

Destroy old resource

✅ No service interruption

```
lifecycle {
  create_before_destroy = true
}
```

🎯 Perfect for security groups, load balancers, IAM roles

🔐 *Real-World Example*

Real Example: Security Group Replacement

Without lifecycle control:

- ❌ Security group deleted
- ❌ EC2 instances temporarily lose access
- ❌ App downtime

With create_before_destroy:

- ✓ New security group created first
- ✓ Instances updated to use it
- ✓ Old group removed safely
- ✓ No traffic interruption

```
lifecycle {
  create_before_destroy = true
}
```

📌 Best practice for attached security groups

🔒 *The Setting*

lifecycle { prevent_destroy = true }

What It Does

🚫 *Terraform refuses to destroy the resource*

🛑 *Stops the entire plan if destruction is attempted*

Examples:

- Production databases
- S3 buckets with business data
- Long-lived stateful systems

💥 *What Happens If Someone Runs Destroy?*

If a resource has `prevent_destroy = true` and someone runs:

```
terraform destroy
```

Terraform responds with:

❌ ***Error: Instance cannot be destroyed***

❌ ***Terraform halts execution***

❌ ***Nothing gets deleted***

This acts as a safety brake for production data

🔧 *How to Remove a Protected Resource (Safely)*

Sometimes deletion is necessary (e.g., system decommissioning).

Option 1 — Temporarily Comment It

```
lifecycle {
  # prevent_destroy = true
}
```

1. Run terraform apply
2. Then run terraform destroy

Option 2 — Set to False

```
lifecycle {
  prevent_destroy = false
}
```

Apply first → Then destroy

⚠ Always document and get approval before doing this

🧠 *When Should You Use prevent_destroy?*

Best candidates:

- *S3 buckets with important data*
- *RDS databases*
- *DynamoDB tables*
- *Shared production infrastructure*

🚨 Note: This only protects from **Terraform deletion**, not manual AWS Console deletion

🔚 *ignore_changes: Handling External Modifications*

Sometimes attributes change outside Terraform:

- AWS adds system tags
- Security tools add compliance tags
- Auto Scaling modifies metadata

Terraform normally tries to undo those changes – causing unnecessary churn.

Solution

```
lifecycle {
  ignore_changes = [attribute_name]
}
```

Terraform will ignore drift for those attributes

🏷️ *Example: Ignoring External Tag Changes*

```
resource "aws_instance" "web" {
  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t3.micro"

  tags = {
    Name        = "web-server"
    Environment = "dev"
  }

  lifecycle {
    ignore_changes = [
      tags["LastScannedBy"],
      tags["ComplianceStatus"]
    ]
  }
}
```

✓ Terraform will not try to remove those tags

✓ Prevents conflict with security/compliance tools

🎯 *What Can Be Ignored?*

You can ignore:

01

Specific Attributes

```
ignore_changes = [instance_type, ami]
```

02

Nested Attributes

```
ignore_changes = [
  tags["ExternalTag"],
  root_block_device[0].volume_size
]
```

03

Everything (Rare)

```
ignore_changes = all
```

⚠ Using `all` means Terraform **won't update the resource after creation**

⚠️ *When Not to Use ignore_changes*

Do not use it to hide real problems.

Bad reasons:

- ❌ Avoiding proper configuration fixes
- ❌ Ignoring unexpected drift
- ❌ Skipping investigation of root cause

Good reasons:

- ✓ Known external automation
- ✓ AWS-managed attributes
- ✓ One-time initialization settings

🔗 *Combining Lifecycle Settings*

You can use multiple lifecycle rules together:

```
lifecycle {
  create_before_destroy = true
  ignore_changes = [
    tags["LastScannedBy"],
    tags["ComplianceStatus"]
  ]
}
```

But ⚠ Important Rule

> ❌ prevent_destroy + create_before_destroy together doesn't make sense
>
> If Terraform can't destroy, it also can't replace

🧭 *Best Practices*

🔐 *Security Groups*

```
lifecycle {
  create_before_destroy = true
}
```

🛡️ *Critical Data*

```
lifecycle {
  prevent_destroy = true
}
```

🏷️ *External Tool Modifications*

```
lifecycle {
  ignore_changes =
[tags["LastScannedBy"]]
}
```

📌 Use lifecycle deliberately – not by default

📝 *Always Document Lifecycle Usage*

Future engineers (and you in 6 months) need context.

```
lifecycle {
  # Create new SG before destroying old to avoid downtime
  create_before_destroy = true

  # Security scanner adds this tag automatically
  ignore_changes = [tags["LastScannedBy"]]
}
```

Good comments = fewer production surprises 🚀

🎓 *Key Takeaways*

✓ ***create_before_destroy***

Zero-downtime replacements

✓ ***prevent_destroy***

Protection from accidental deletion

✓ ***ignore_changes***

Avoid fighting external systems

✓ ***Multiple settings***

Can be combined wisely

✓ ***Lifecycle rules***

Make Terraform production-safe

Knowledge Check

Q1: Default Terraform replacement order?

- A) Create then destroy
- B) Destroy then create
- C) Update in place
- D) Prompt user

Solution: Q1: B