Terraform Language Elements

Plan

- Loops
- If-Statements
- Deployment
- Gotchas
- Other language elements

希福

Declarative Languages

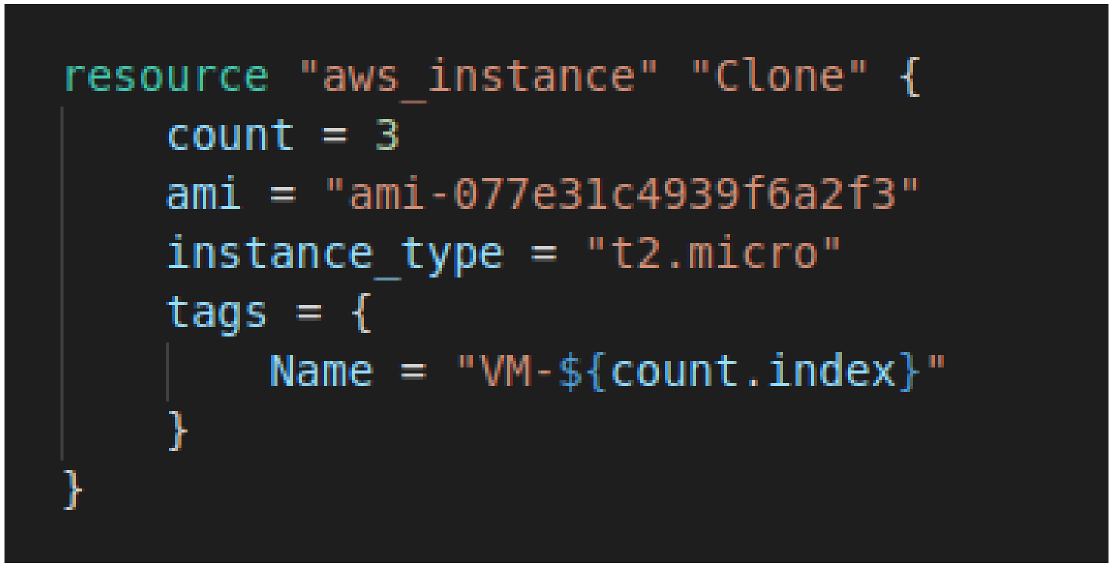
- Declarative languages, like Terraform, normally do not have typical programming constructs like loops
- The challenge is expressing scenarios that require the conditional configuration of resources
  - For example, creating a module that creates resources only for certain users and not others
- Terraform primitives allow certain kinds of operations to allow dynamic and conditional configuration to be done
  - These do not look like standard constructs in programming languages that have the same functionality

Loops

- Terraform has several loop constructs to provide looping functionality in different scenarios
  - *count* parameter: to loop over resources
  - *for_each* expressions: to loop over resources and inline blocks within a functionality
  - *for* expressions: to loop over lists and maps
  - *for* string directive: to loop over lists and maps withing a string

Loops with "count"

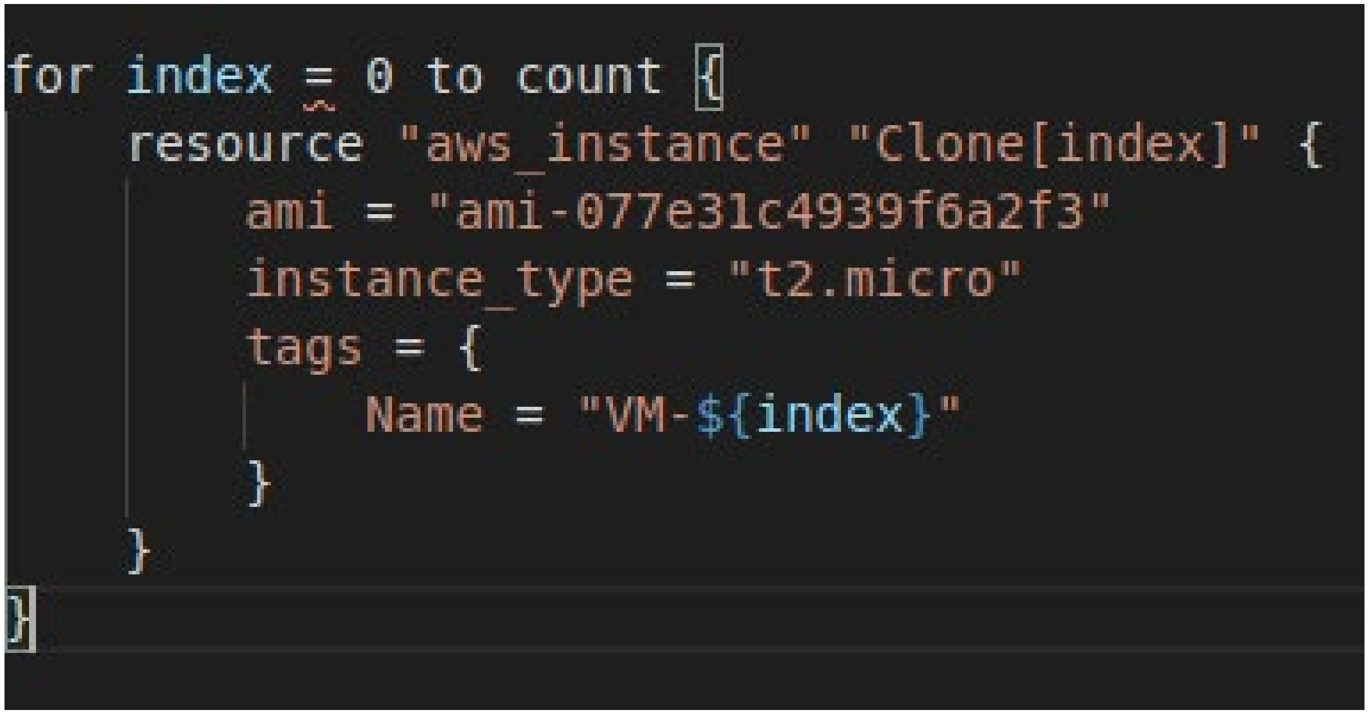
- The looping procedural code is implied and generated under the hood by terraform
- We specify the number of iterations with the count, which often represents the number of copies of a resource
  - The following code creates three instances with the names *VM-0* , *VM-1* and *VM-2*



```
resource "aws_instance" "Clone" {
    count = 3
    ami = "ami-077e31c4939f6a2f3"
    instance_type = "t2.micro"
    tags = {
        Name = "VM-${count.index}"
    }
}
```

What Count is doing

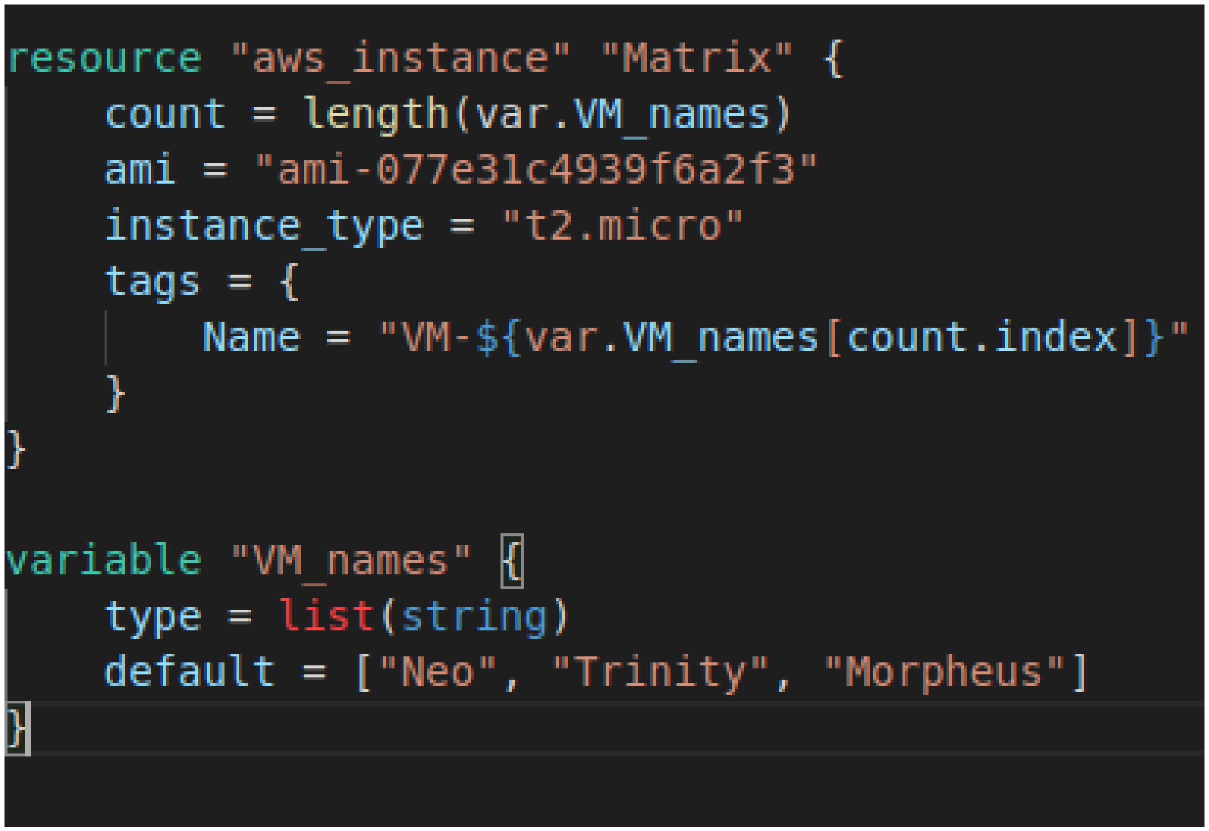
- Under the hood something like this is conceptually happening:



```
for index = 0 to count {
    resource "aws_instance" "Clone[index]" {
        ami = "ami-077e31c4939f6a2f3"
        instance_type = "t2.micro"
        tags = {
            Name = "VM-${index}"
        }
    }
}
```

Array Lookups

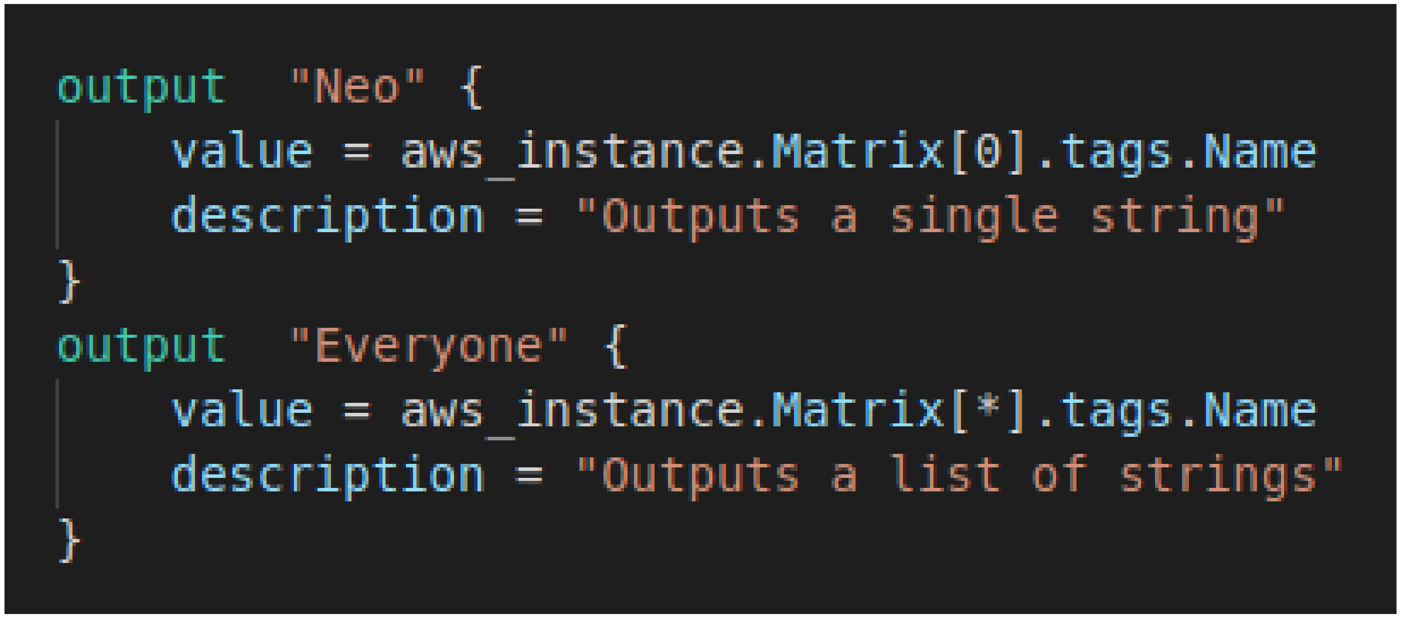
- We can supply list of value in arrays
  - Array elements can be reference with array notation
  - The length of the array (also strings and maps) returned from the built-in function *length()*



```
resource "aws_instance" "Matrix" {
    count = length(var.VM_names)
    ami = "ami-077e31c4939f6a2f3"
    instance_type = "t2.micro"
    tags = {
        Name = "VM-${var.VM_names[count.index]}"
    }
}

variable "VM_names" {
    type = list(string)
    default = ["Neo", "Trinity", "Morpheus"]
}
```

Arrays of Resources

- Using *count* on a resource creates an array of resources rather than just one resource
- To get all the users, a "splat" expression is used



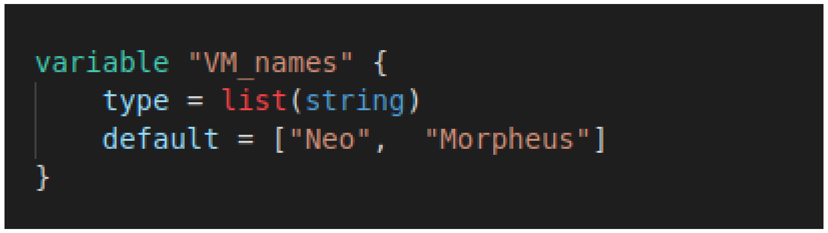
```
output  "Neo" {
    value = aws_instance.Matrix[0].tags.Name
    description = "Outputs a single string"
}
output  "Everyone" {
    value = aws_instance.Matrix[*].tags.Name
    description = "Outputs a list of strings"
}
```

Limitations of Count

- *count* can loop over resources but not inline blocks
- For example, we cannot iterate over the inline block for `tag` to generate multiple tag blocks dynamically
- Changing the values in a list modifies the created infrastructure
- If we create the infrastructure with this list and iff trinity is removed then the correspondence between the array of resources and the list of names no longer is valid
  - Terraform restores the mapping by recreating the resources

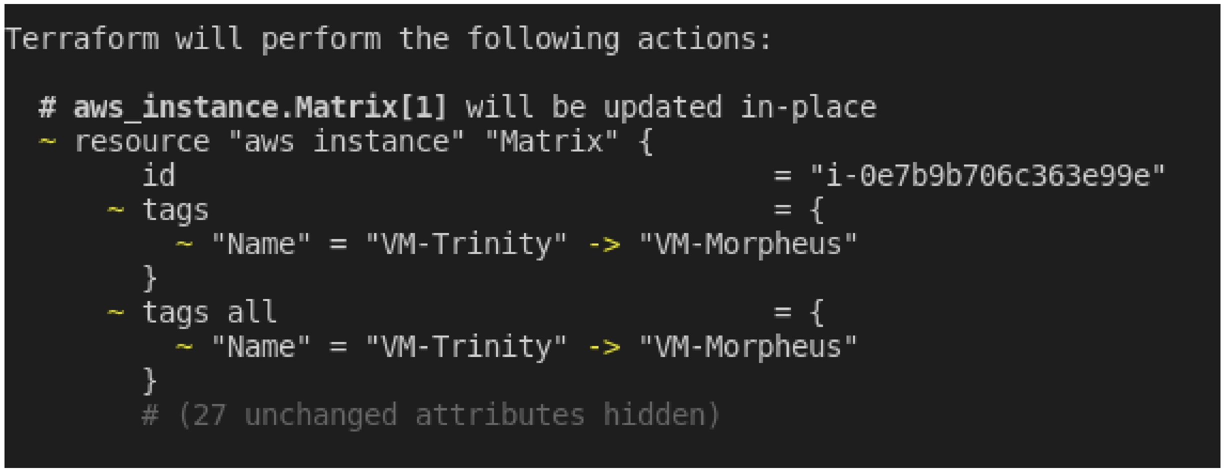
Limitations of Count

- Using



```
variable "VM_names" {
    type = list(string)
    default = ["Neo",  "Morpheus"]
}
```

- Then *terraform plan* produces the following output
  - Trinity is not destroyed, Morpheus is, then Trinity is renamed to Morpheus



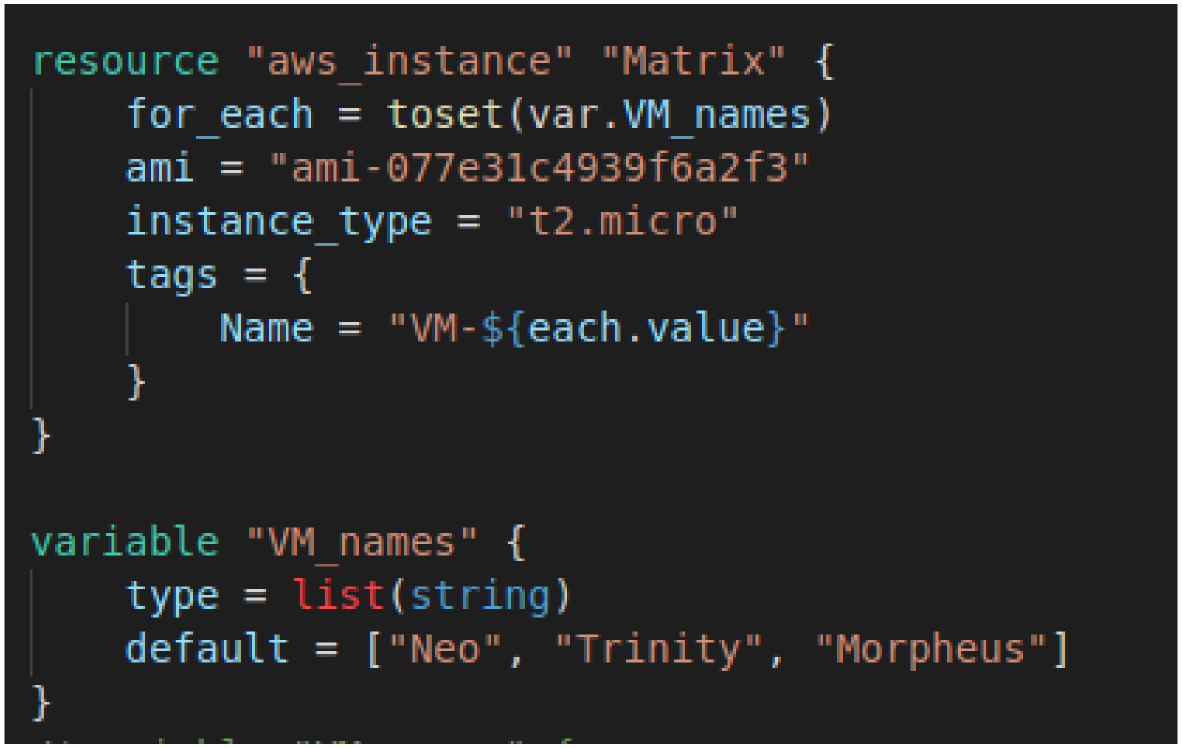
```
Terraform will perform the following actions:

  # aws_instance.Matrix[1] will be updated in-place
  ~ resource "aws_instance" "Matrix" {
        id                                   = "i-0e7b9b706c363e99e"
      ~ tags                                 = {
          ~ "Name" = "VM-Trinity" -> "VM-Morpheus"
        }
      ~ tags_all                             = {
          ~ "Name" = "VM-Trinity" -> "VM-Morpheus"
        }
        # (27 unchanged attributes hidden)
```

Lab 4-1

- Please do lab 4-1

Loops with "for_each" Expressions

- The for_each expression allows looping over lists, sets, and maps to create either:
  - multiple copies of an entire resource, or
  - multiple copies of an inline block within a resource
- The previous example is now:



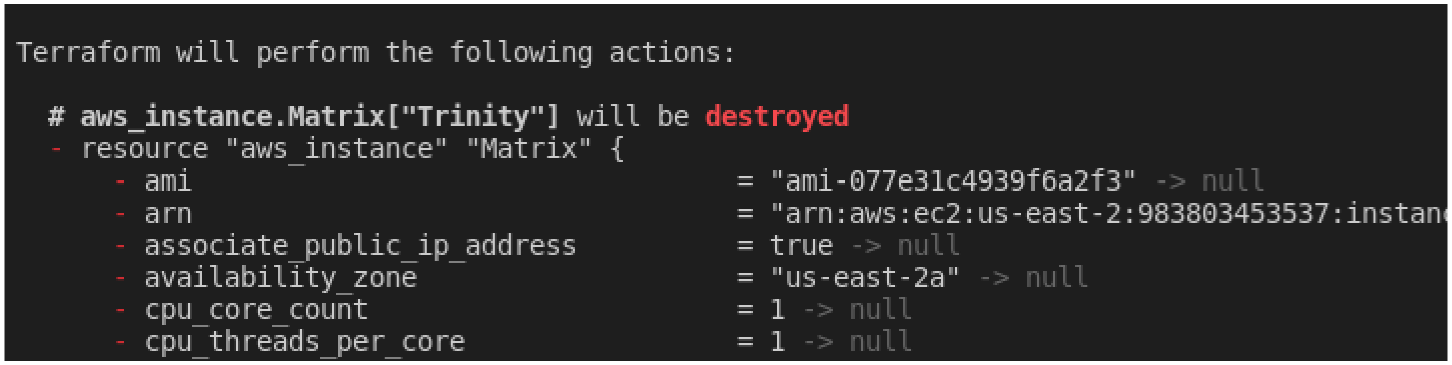
```
resource "aws_instance" "Matrix" {
    for_each = toset(var.VM_names)
    ami = "ami-077e31c4939f6a2f3"
    instance_type = "t2.micro"
    tags = {
        Name = "VM-${each.value}"
    }
}

variable "VM_names" {
    type = list(string)
    default = ["Neo", "Trinity", "Morpheus"]
}
```

Use of "for_each"

- The function *toset()* converts the var.user_names list into a set
  - *for_each* supports sets and maps only when used on a resource
- When for_each loops each name in the list is made available in the *each* value
  - The user name will also be available in *each.key* , but this is usually used only with maps of key/value pair.
- Once *for_each* is used on a resource, it creates a map of resources rather than array of resources
  - This is why we can't use a list with possible duplicates
  - This would lead to duplicate keys

Map Advantages

- Maps do not rely on position like lists do
  - Allows us to remove items from the middle of a collection safely
- Going back to the problem of deleting "trinity" with a map of resources we get:



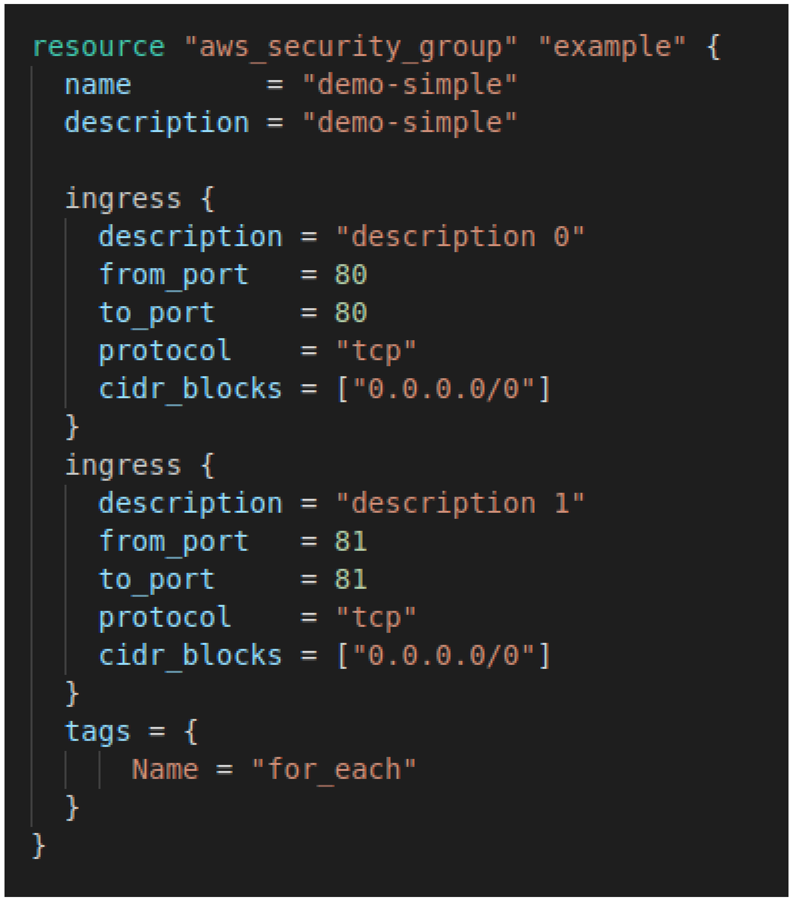
```
Terraform will perform the following actions:

  # aws_instance.Matrix["Trinity"] will be destroyed
  - resource "aws_instance" "Matrix" {
      - ami                                  = "ami-077e31c4939f6a2f3" -> null
      - arn                                  = "arn:aws:ec2:us-east-2:983803453537:instan
      - associate_public_ip_address          = true -> null
      - availability_zone                    = "us-east-2a" -> null
      - cpu_core_count                       = 1 -> null
      - cpu_threads_per_core                 = 1 -> null
```

Lab 4-1

- Please do Lab 4-2

Inline Blocks with "for_each"

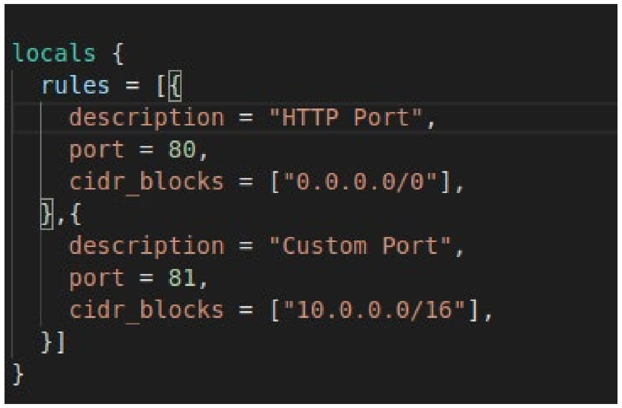
- We may want to configure, for example, multiple sets of ingress rules from a standard set of configurations like:



```
resource "aws_security_group" "example" {
  name        = "demo-simple"
  description = "demo-simple"

  ingress {
    description = "description 0"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    description = "description 1"
    from_port   = 81
    to_port     = 81
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  tags = {
      Name = "for_each"
  }
}
```

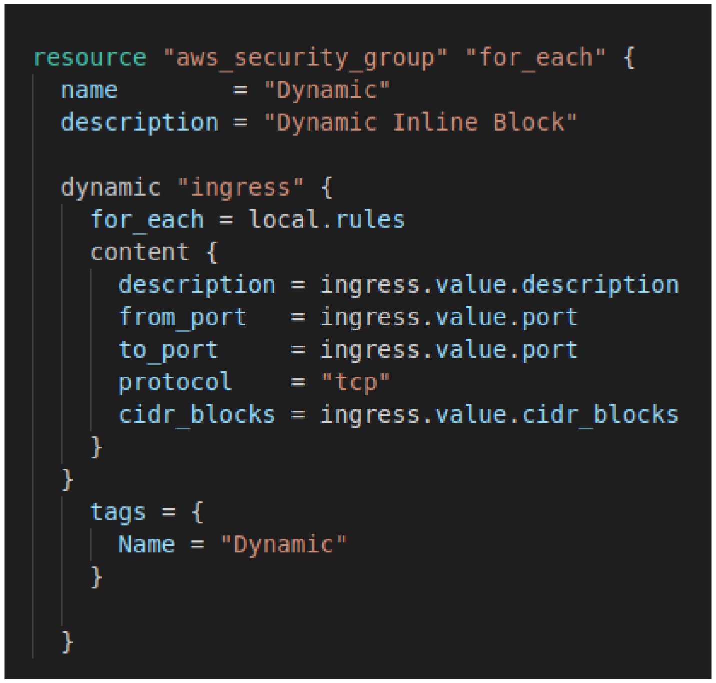
Reusable Ingress Rules

- We can define reusable ingress rules, here with local variable which is a list of ingress rule terraform objects



```
locals {
  rules = [{
    description = "HTTP Port",
    port = 80,
    cidr_blocks = ["0.0.0.0/0"],
  },{
    description = "Custom Port",
    port = 81,
    cidr_blocks = ["10.0.0.0/16"],
  }]
}
```

- Implemented as a dynamic block



```
resource "aws_security_group" "for_each" {
  name        = "Dynamic"
  description = "Dynamic Inline Block"

  dynamic "ingress" {
    for_each = local.rules
    content {
      description = ingress.value.description
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = "tcp"
      cidr_blocks = ingress.value.cidr_blocks
    }
  }
    tags = {
      Name = "Dynamic"
    }

  }
```

Lab 4-3

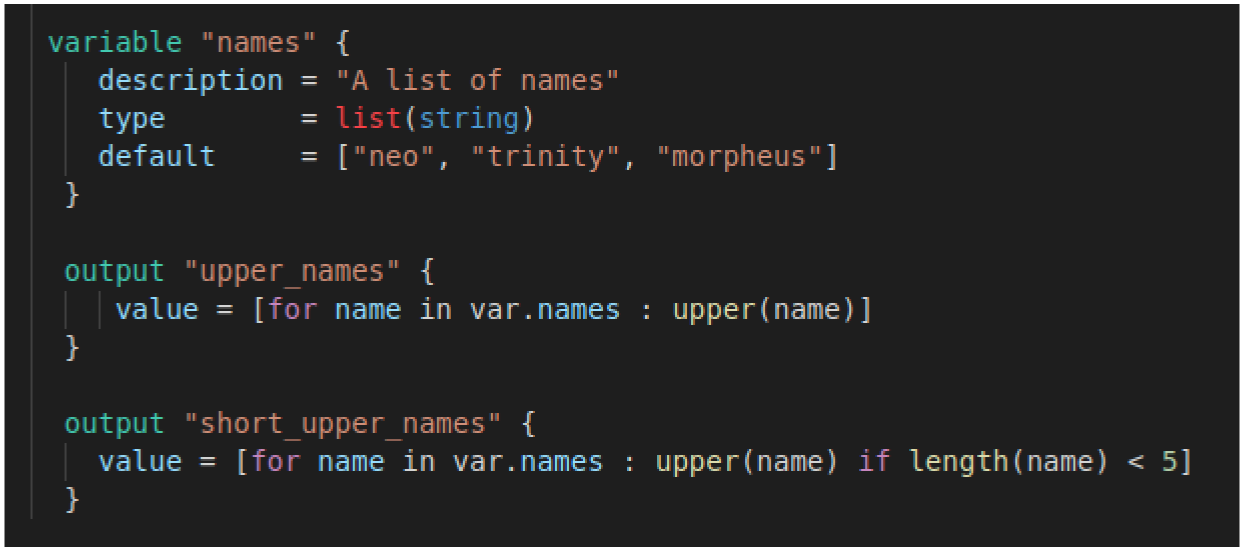
- Please do Lab 4-3

Looping with Expressions

- Terraform allows operations on the data similar to operations in a programming language
- Syntax is:

```
[for < ITEM > in < LIST > : < OUTPUT >]
```

- Demonstrated in this code:



```
variable "names" {
    description = "A list of names"
    type        = list(string)
    default     = ["neo", "trinity", "morpheus"]
 }

 output "upper_names" {
     value = [for name in var.names : upper(name)]
 }

 output "short_upper_names" {
    value = [for name in var.names : upper(name) if length(name) < 5]
 }
```

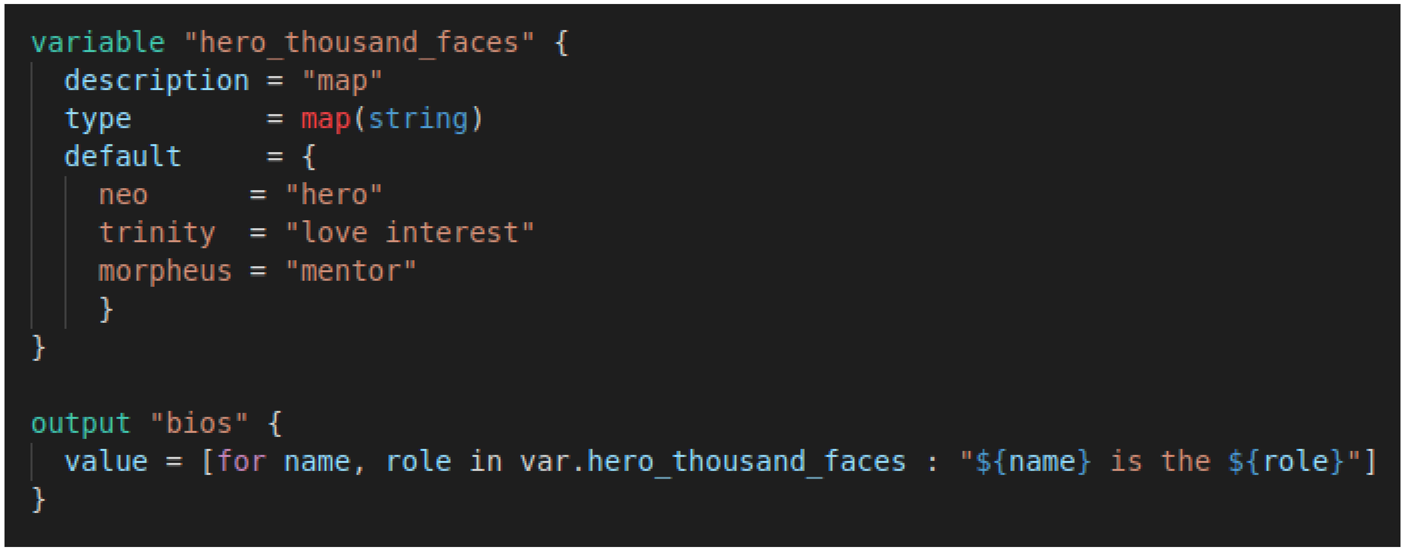
- Note that this resembles a "map" operation in a functional programming language

Working with Map Inputs

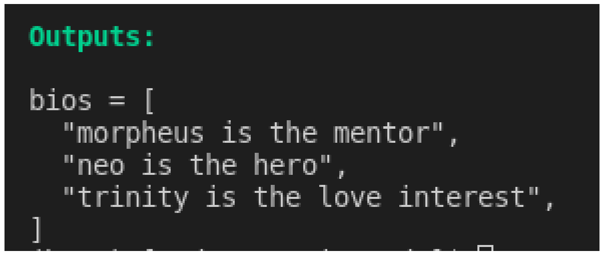
- The for expression can loop over a map as well

```
[for < KEY >, < VALUE > in < MAP > : < OUTPUT >]
```

- Example of use:



```
variable "hero_thousand_faces" {
  description = "map"
  type        = map(string)
  default     = {
    neo      = "hero"
    trinity  = "love interest"
    morpheus = "mentor"
  }
}

output "bios" {
  value = [for name, role in var.hero_thousand_faces : "${name} is the ${role}"]
}
```

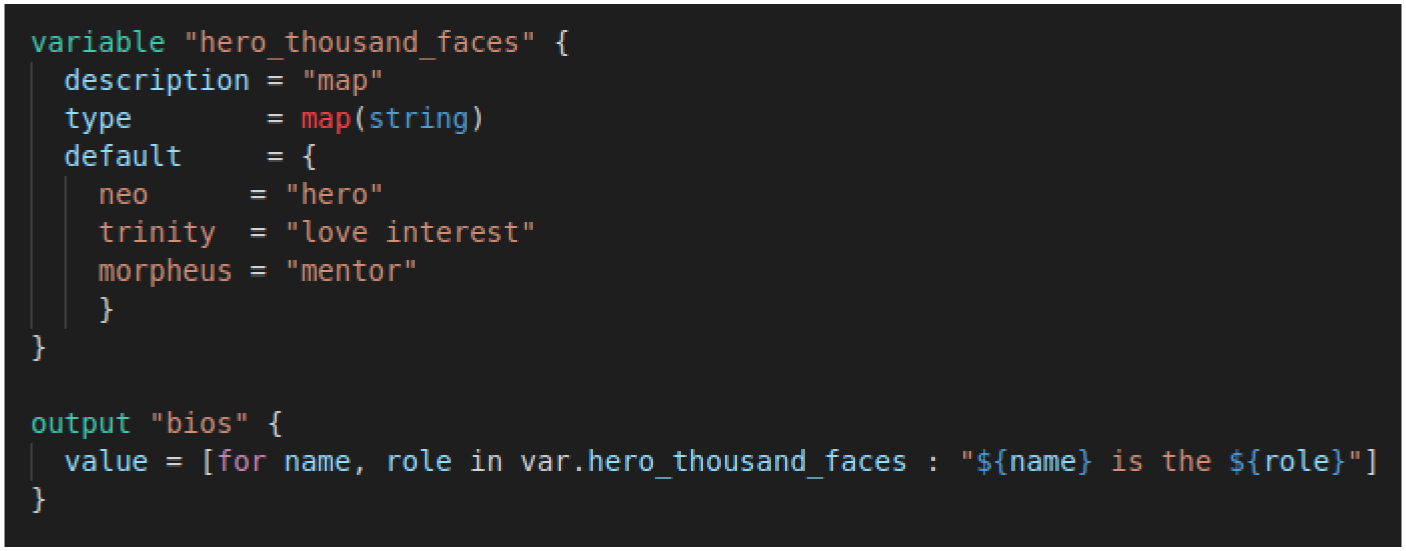


```
Outputs:

bios = [
  "morpheus is the mentor",
  "neo is the hero",
  "trinity is the love interest",
]
```

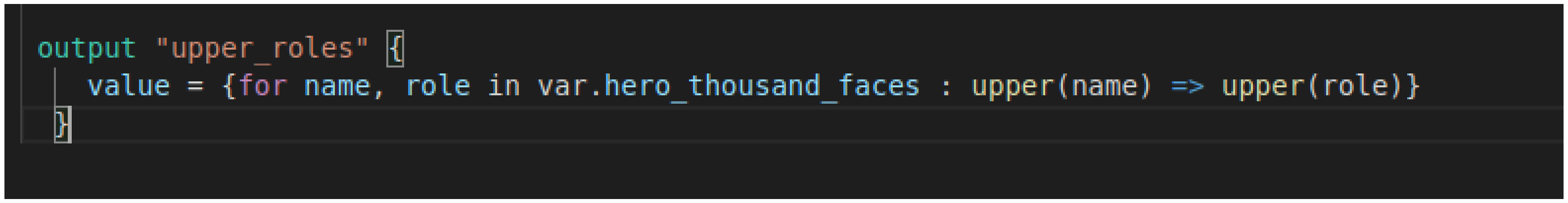
Outputing a Map

- Looping over a list or map can output a map using the syntax:

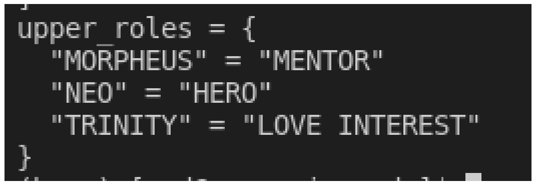
```
// Loop over a map and output a map
{for < KEY >, < VALUE > in < MAP > : < OUTPUT_KEY > => < OUTPUT_VALUE >}
```



```
variable "hero_thousand_faces" {
  description = "map"
  type        = map(string)
  default     = {
    neo      = "hero"
    trinity  = "love interest"
    morpheus = "mentor"
    }
}

output "bios" {
  value = [for name, role in var.hero_thousand_faces : "${name} is the ${role}"]
}
```



```
output "upper_roles" {
   value = {for name, role in var.hero_thousand_faces : upper(name) => upper(role)}
}
```



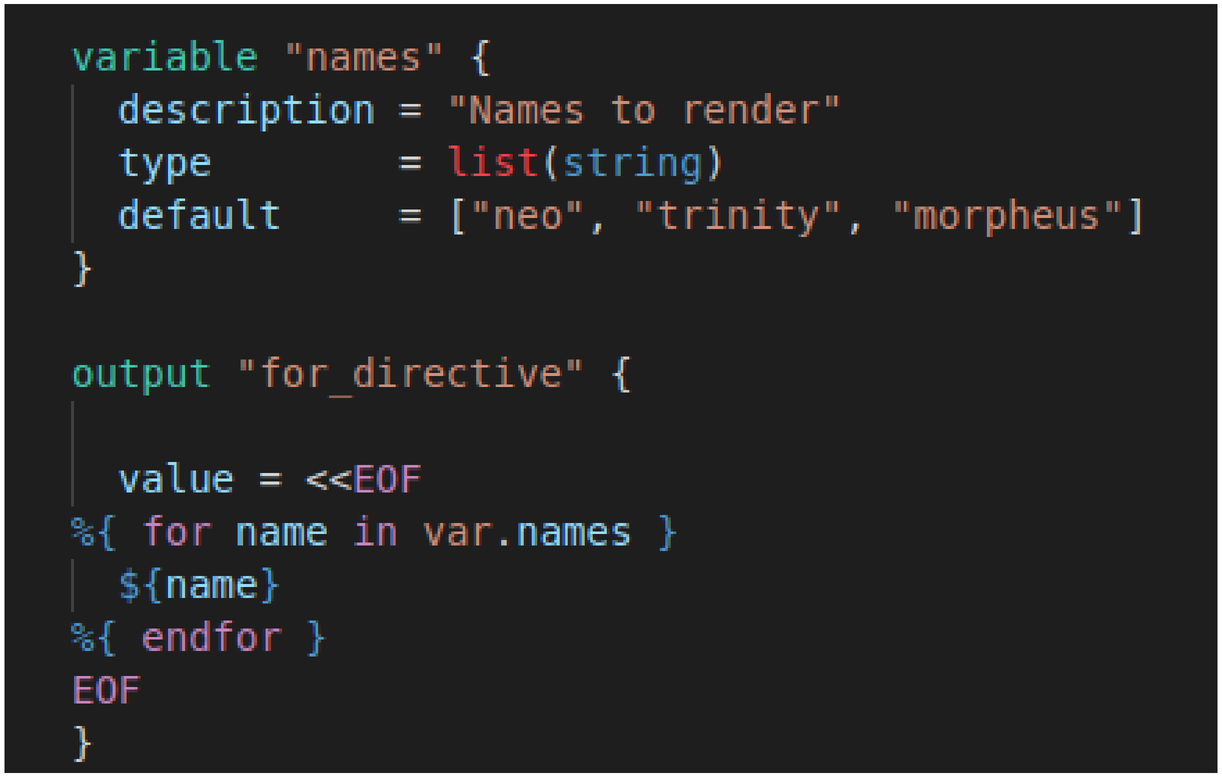
```
upper_roles = {
  "MORPHEUS" = "MENTOR"
  "NEO" = "HERO"
  "TRINITY" = "LOVE INTEREST"
}
```

Loops with the "for" String Directive

- String directives allow for-loops and if-statements in strings using a syntax similar to string interpolations but instead of a dollar sign and curly braces (${…}), it uses a percent sign and curly braces (%{…})
  - Terraform supports two types of string directives: for-loops and conditionals
- For loop syntax (collection is a list or map)

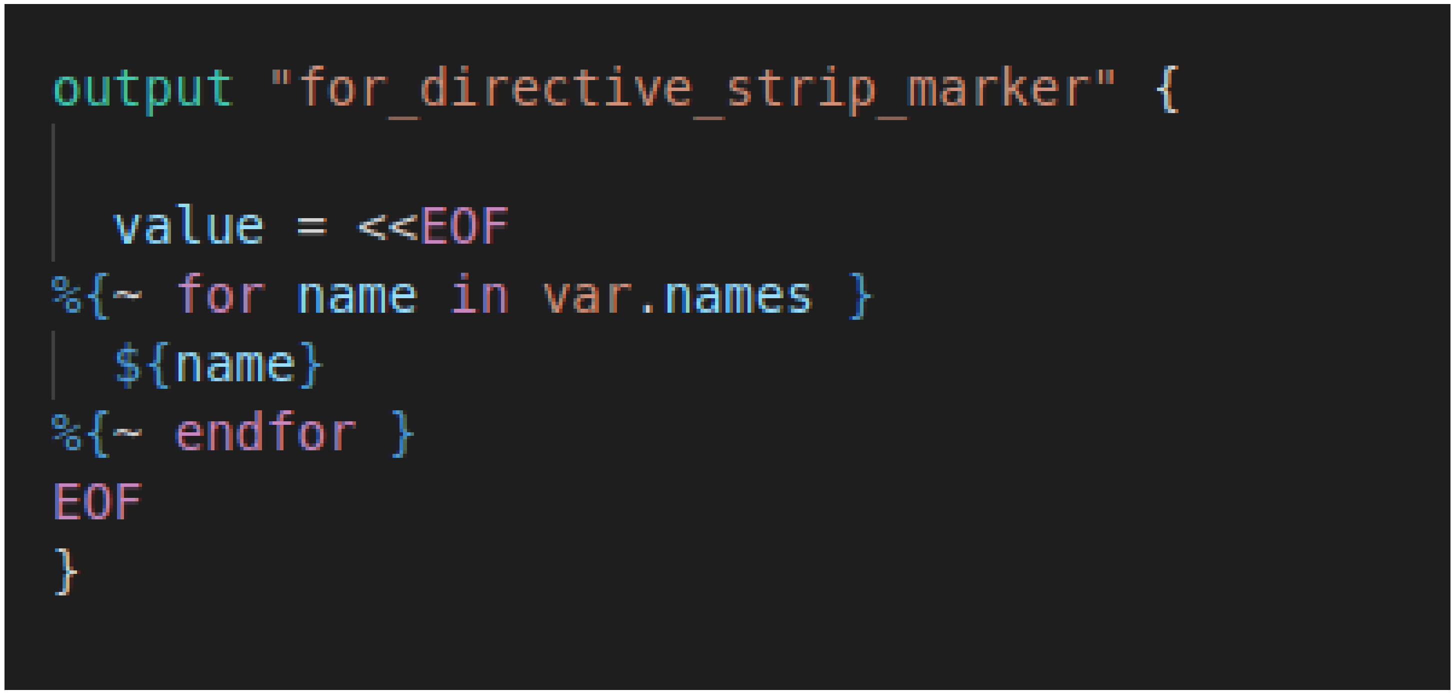
```
%{ for < ITEM > in < COLLECTION > }< BODY >%{ endfor }
```

String for loop Example



```
variable "names" {
  description = "Names to render"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}

output "for_directive" {

  value = <<EOF
%{ for name in var.names }
  ${name}
%{ endfor }
EOF
}
```

Trimming Whitespace



```
output "for_directive_strip_marker" {

  value = <<EOF
%{~ for name in var.names }
  ${name}
%{~ endfor }
EOF
}
```

Conditionals

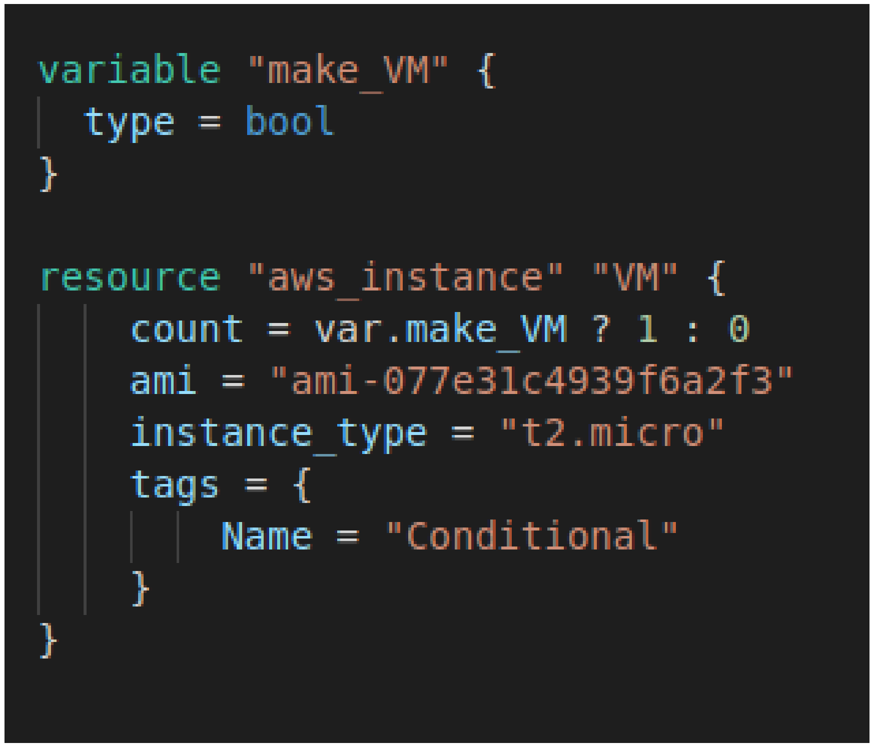
- There are also several different ways to do conditionals, each intended to be used in a slightly different scenario:
  - *count parameter* : Used for conditional resources
  - *for_each and for expressions* : Used for conditional resources and inline blocks within a resource
  - *if string directive* : Used for conditionals within a string

Conditionals with "count"

- We can define a Boolean variable as our test condition "enable_autoscaling"
- We can set the count on a resource to "0" which means that resource is not created
- Terraform allows ternary conditionals of the form:

```
< CONDITION > ? < TRUE_VAL > : < FALSE_VAL >
```

- This allows for conditional creation of resources:



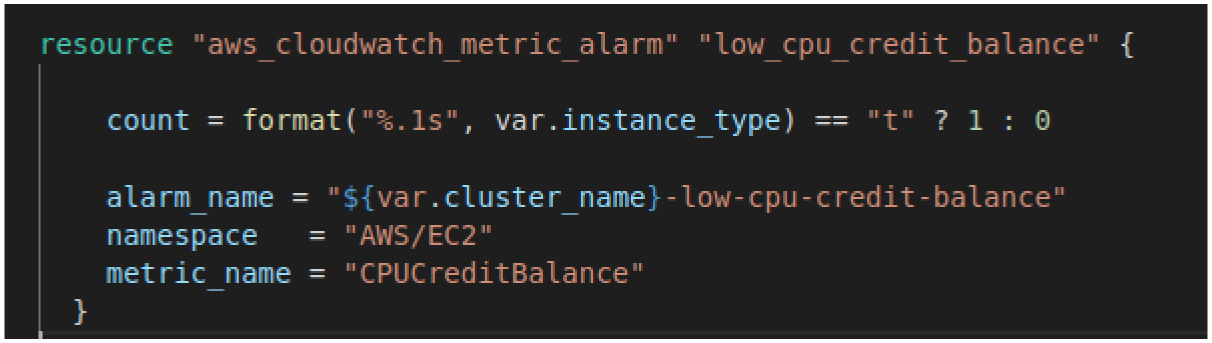
```
variable "make_VM" {
  type = bool
}

resource "aws_instance" "VM" {
    count = var.make_VM ? 1 : 0
    ami = "ami-077e31c4939f6a2f3"
    instance_type = "t2.micro"
    tags = {
        Name = "Conditional"
    }
}
```

Working with Non-boolean

- The previous example worked because we could define a boolean variable
  - However, we may have to decode information in a string to make a decision
- Example: We want to set a cloud-watch alarm that triggers when CPU credits are low
  - However, CPUcredits only "txxx" instances
  - Larger instance like m4.large do not return a CPU credit metric and will always appear to be in an INSUFFICIENT_DATA state
  - We want the metric to apply to only txxx instance but we don't want to create a special Boolean

Working with Non-boolean

- The solution is to utilize the fact that first letter of the instance type should be a "t"



```
resource "aws_cloudwatch_metric_alarm" "low_cpu_credit_balance" {

    count = format("%.1s", var.instance_type) == "t" ? 1 : 0

    alarm_name = "${var.cluster_name}-low-cpu-credit-balance"
    namespace   = "AWS/EC2"
    metric_name = "CPUCreditBalance"
  }
```

- The format function to extract just the first character from var.instance_type.
  - If that character is a “t” (e.g., t2.micro), it sets the count to 1;
  - otherwise, it sets the count to 0
  - This way, the alarm is created only for instance types that actually have a CPUCreditBalance metric.

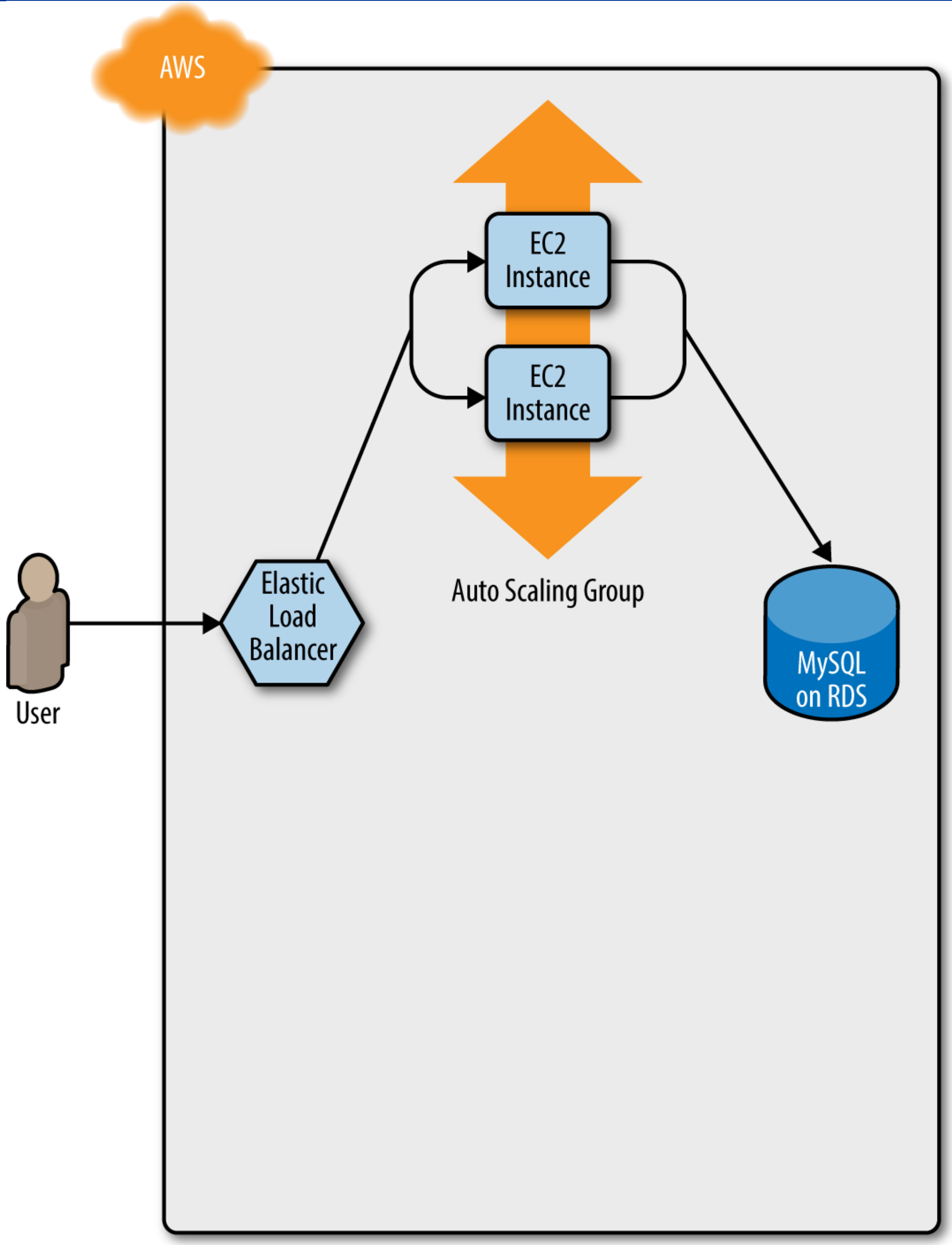
Zero-Downtime Deployment

- The challenge is to update a cluster without causing downtime for users
  - How do you deploy a new Amazon Machine Image (AMI) across the cluster?
- If we are deploying a new version of our app, we don't want there to be downtime as we switch over
- We full test and deploy our app in a test are to ensure it is working before we make the transition
- We then deploy the application into a new launch configuration which will be the target of the auto-scaling group
- However the challenge is switching launch configurations, if destroy the old one, then we have downtime while the new one is being created

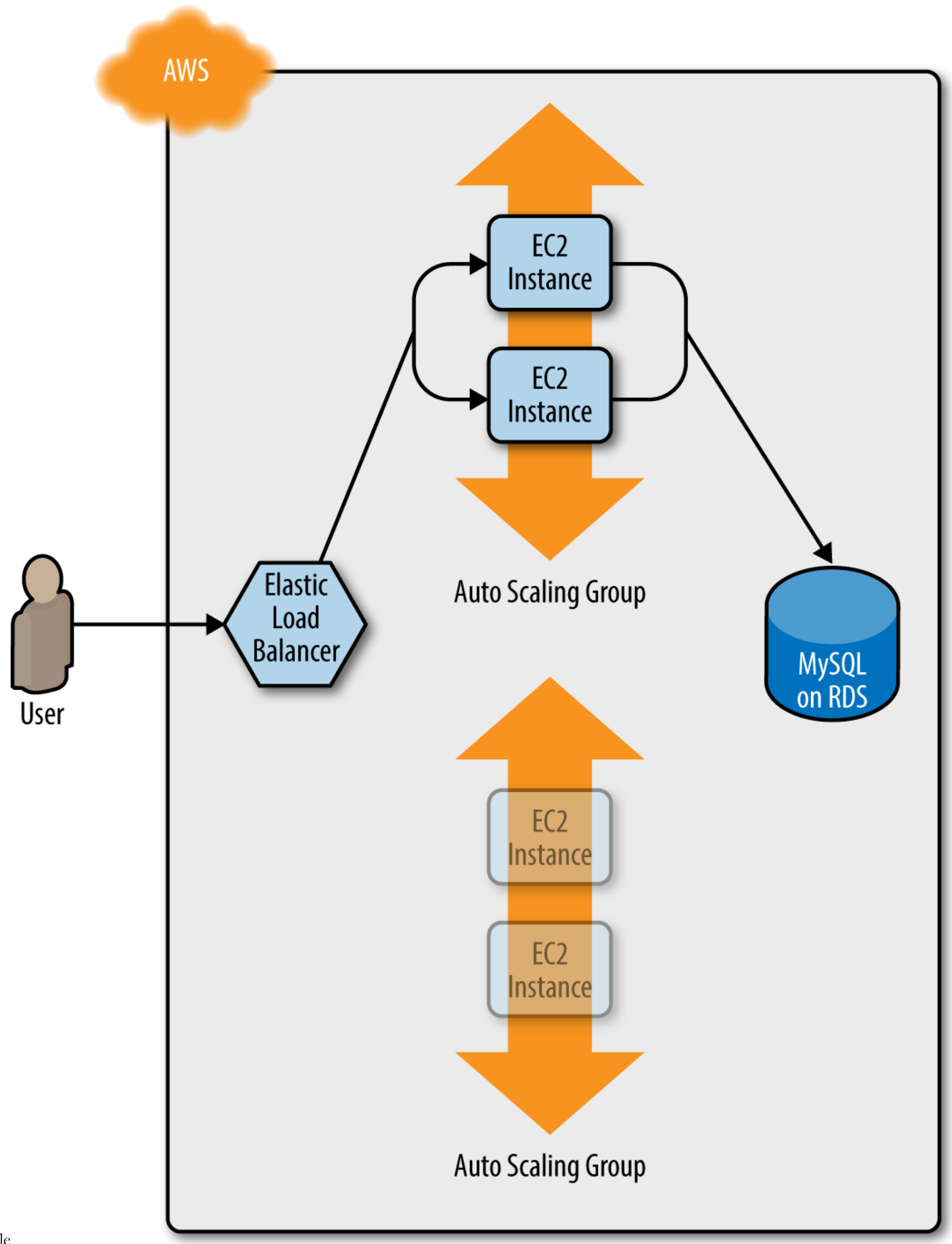
Zero-Downtime Deployment

- The way to accomplish that is to create the replacement ASG first and then destroy the original one
- Configure the name parameter of the ASG to depend directly on the name of the launch configuration
- Each time the launch configuration changes (which it will when you update the AMI or User Data), its name changes, and therefore the ASG's name will change, which forces Terraform to replace the ASG
- Set the create_before_destroy parameter of the ASG to true, so that each time Terraform tries to replace it, it will create the replacement ASG before destroying the original
- Set the min_elb_capacity parameter of the ASG to the min_size of the cluster so that Terraform will wait for at least that many servers from the new ASG to pass health checks in the ALB before it will begin destroying the original ASG

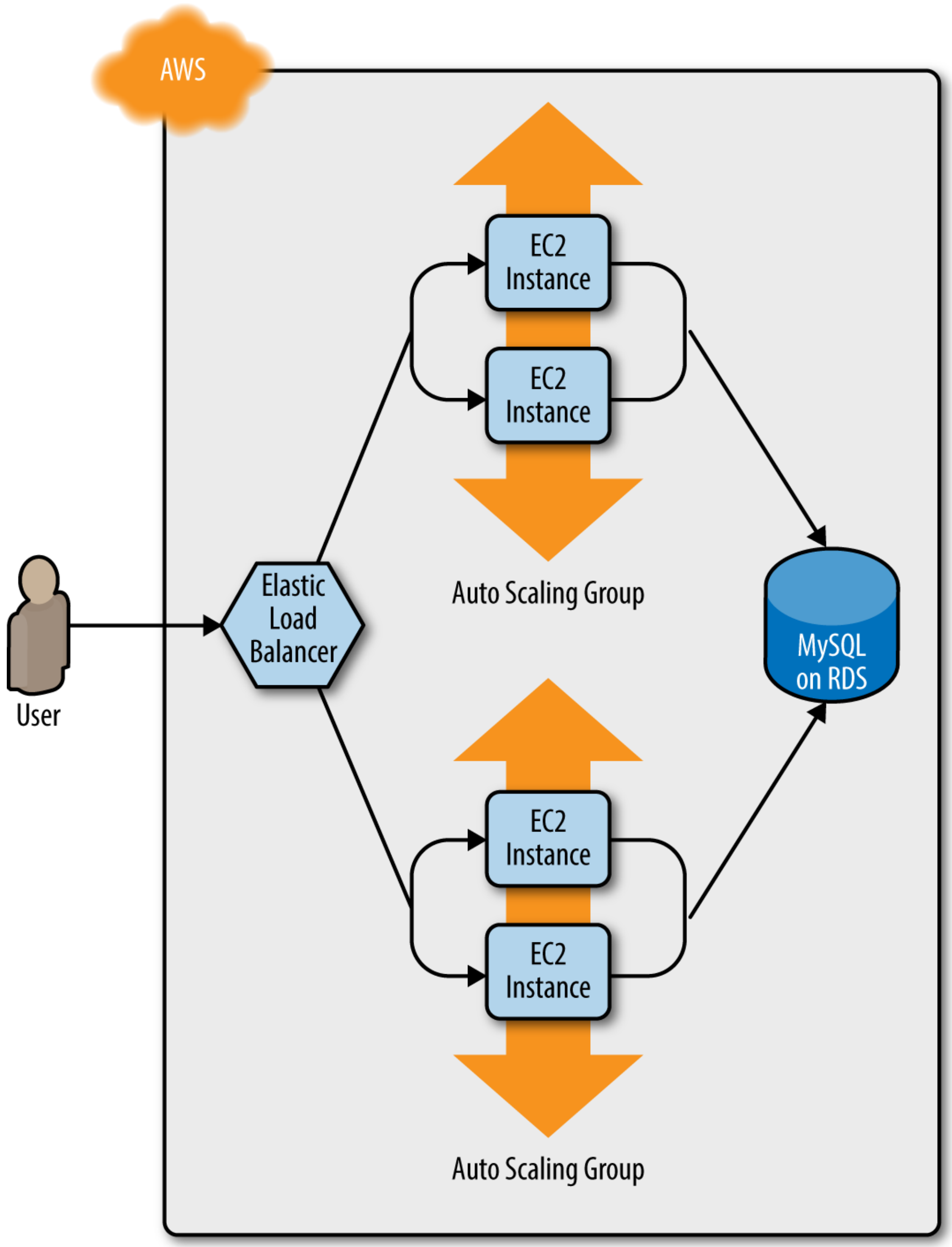
Zero-Downtime Deployment 1



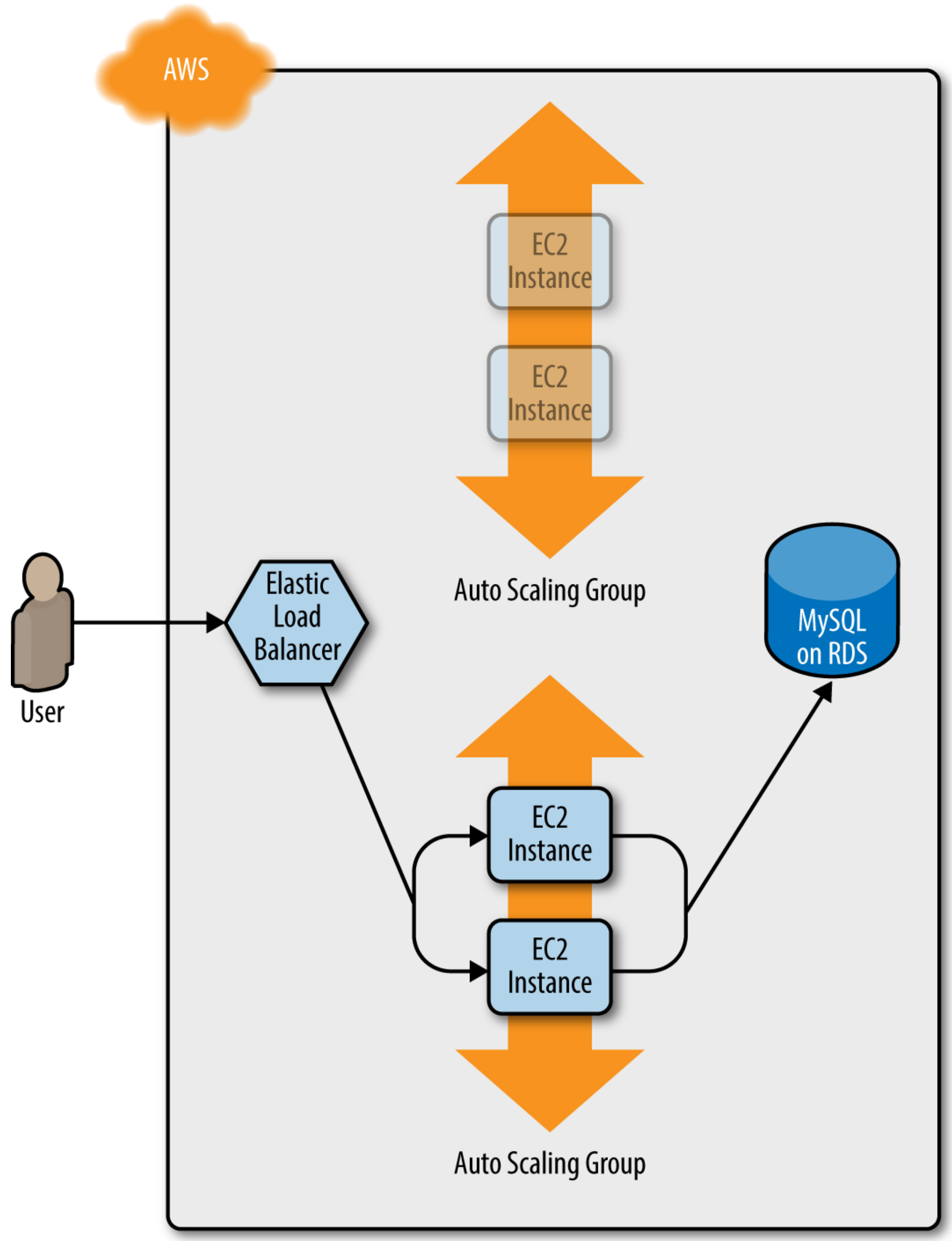
Zero-Downtime Deployment 2



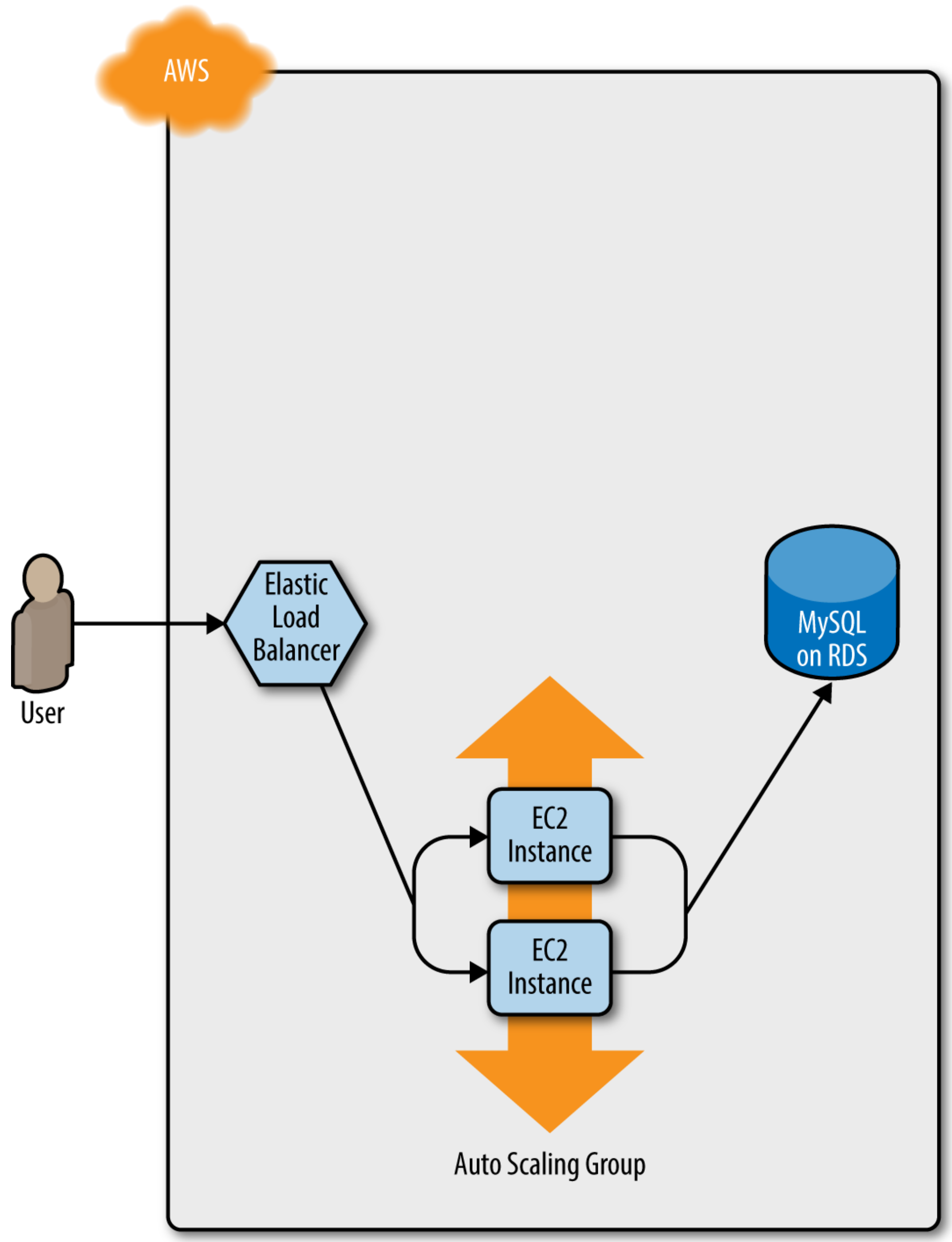
Zero-Downtime Deployment 3



Zero-Downtime Deployment 4



Zero-Downtime Deployment 5



Terraform Gotchas

- We now take a step back and point out a few gotchas
- count and for_each have limitations
  - You cannot reference any resource outputs in count or for_each
  - You cannot use count or for_each within a module configuration
- Zero-downtime deployment has limitations
  - it doesn’t work with auto scaling policies
  - it resets your ASG size back to its min_size after each deployment
- Valid plans can fail
  - Terraform only looks at resources in the state file and doesn't take into account other resources
  - Plans that look good may fail because of resource conflicts
  - Ideally infrastructure should only rely on Terraform
  - Import existing infrastructure

Terraform Gotchas

- Refactoring can be tricky
  - Changes can have major effects
  - Changing the name parameter of certain resources will delete the old version of the resource and create a new version to replace it (immutable infrastructure)
- Refactoring points:
  - Always use the plan command
  - Create before destroy
  - Keep in mind that changing identifiers requires changing state
  - Some parameters are immutable so changing them requires replacing the resource

Terraform Gotchas

- Eventual consistency is consistent… eventually
- APIs for some cloud providers, such as AWS, are asynchronous and eventually consistent
  - Asynchronous means that the API might send a response immediately, without waiting for the requested action to complete
  - Eventually consistent means that it takes time for a change to propagate throughout the entire system
  - For some period of time, you might get inconsistent responses depending on which data store replica happens to respond to your API calls
- Generally, re-running *terraform apply* solves the problem