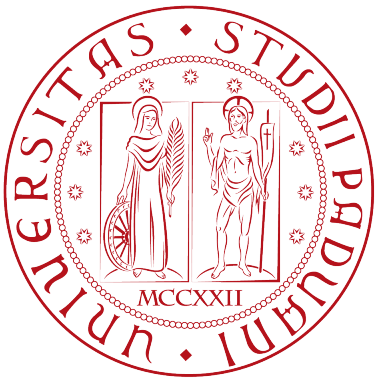
Università degli Studi di Padova



Dipartimento di Matematica "Tullio Levi-Civita"

Scuola di Scienze

Corso di Laurea Magistrale in Informatica

Il Parallelismo in Terraform

Francesco Antonio Migliorin
1207422
Anno 2020/2021

Indice

Elenco delle figure

1 Approfondimento Terraform

Terraform[1] è uno strumento di Infrastructure as Code (IaC), creato da HashiCorp, che fornisce una sintassi comune, chiamata HCL, per interagire con più attori, che nella maggior parte dei casi sono cloud provider. É uno strumento molto giovane, infatti il 08/06/2021 é stata rilasciata la versione 1.0 di Terraform.

> Questo approfondimento non vuole focalizzarsi sul linguaggio HCL ma sul parallelismo di Terraform. Per questo motivo verranno trattate solamente le basi del linguaggio e del funzionamento di Terraform, necessarie per comprendere meglio l'argomento trattato. Non si prefigge quindi di essere una guida esaustiva.

1.1 Le basi di Terraform

Terraform é uno strumento dichiarativo, scritto in Go, che utilizza il linguaggio di alto livello HCL per descrivere l'infrastruttura cloud o on-premise. Terraform legge tutti i file, presenti in una directory, `.tf` che contengono la configurazione dell'infrastruttura e genera poi un piano, ovvero le azione necessarie per raggiungere lo "stato finale" desiderato, e successivamente lo attua per eseguire l'effettivo provisioning dell'infrastruttura.

1.1.1 I comandi principali

Terraform Init Il comando `terraform init` viene utilizzato per inizializzare una directory contenente dei file di configurazione per terraform `.tf`. Durante il processo di inizializzazione, i moduli e i *providers* esterni vengono automaticamente rilevati cercando nel codice sorgente. Vengono quindi referenziati i moduli ed eventualmente scaricato il codice necessario per eseguirli, così come vengono inizializzati i provider, scaricando e installando il relativo plug-in, prima del loro primo utilizzo.
Durante l'inizializzazione vengono inoltre controllare le versioni dei moduli e dei provider per fare in modo che corrispondano.

Terraform Plan Il comando `terraform plan` viene utilizzato per creare un piano di esecuzione. Terraform esegue il comando `refresh`, a meno che non sia disabilitato, e successivamente determina quali azioni sono necessarie per raggiungere lo "stato finale" specificato nei file di configurazione `.tf`. Questo comando è un modo utile per poter verificare se il piano prodotto corrisponde alle modifiche che ci aspettavamo, senza apportare realmente, modifiche all'infrastruttura reale. Questo stato puó essere salvato in un file e passato al comando di `terraform apply`, in questo modo terraform eseguirà le operazioni nella stessa sequenza descritta nel piano.

Terraform Apply Il comando di `terraform apply` viene utilizzato per applicare le modifiche rilevate dal comando `plan` al fine di raggiungere lo stato descritto nella configurazione. Durante l'esecuzione, a ogni task, Terraform scriverà lo stato attuale nel file `terraform.tfstate`. La funzione di questo file é quella di tenere memorizzata lo stato corrente della nostra infrastruttura, in questo modo Terraform sa cos'ha creato, cosa deve modificare e cosa deve distruggere.

Terraform Refresh Il comando `terraform refresh` viene utilizzato per riconciliare l'infrastruttura reale con lo stato di cui Terraform è a conoscenza (lo stato per Terraform si trova nel

file `terraform.tfstate`). In questo modo terraform mappa le modifiche che sono state fatte all'infrastruttura e aggiorna il file `terraform.tfstate`.

Terraform Destroy Il comando `terraform destroy` viene utilizzato per distruggere l'infrastruttura descritta nel file `terraform.tfstate`. Il suo comportamento é analogo al comando `terraform apply`, tuttavia al posto di creare risorse le distrugge.

Terraform Import Terraform è in grado di importare componenti di infrastruttura giá esistenti. Questo permette di portare elementi esistenti sotto la gestione di Terraform, che verranno mappati con la configurazione desiderata e inseriti nel file `terraform.tfstate`.

1.1.2 I principi del linguaggio

Providers La risorsa principale messa a disposizione da terraform sono i *providers*. Questi si occupano dell'interazione tra Terraform e il suo target, che normalmente avviene tramite API.

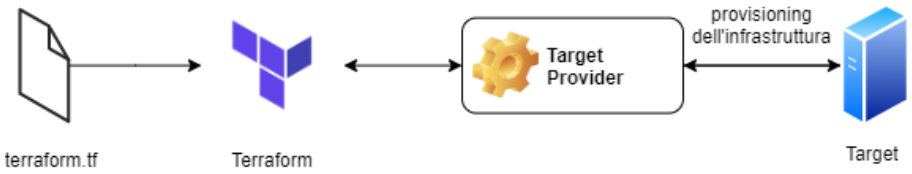


Figura 1: Architettura dei provider

I *providers* sono distribuiti separatamente da Terraform stesso e sono anch'essi scritti in Go. Tutti quelli disponibili, possono essere trovati nel loro registro ufficiale[2]. Ciascun provider aggiunge risorse e/o dati che Terraform può gestire. Senza *providers*, Terraform non può gestire nessun tipo di infrastruttura.

```
provider "aws" {
  region     = "us-west-2"
  access_key = "my-access-key"
  secret_key = "my-secret-key"
}
```

Source Code 1: Esempio di dichiarazione di un provider

Resources Ogni blocco di risorsa dichiara che si desidera che un particolare oggetto dell'infrastruttura esista con le impostazioni fornite in input.

```
resource "aws_instance" "vm" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

Source Code 2: Esempio di dichiarazione di una risorsa

Quando Terraform crea un nuovo oggetto dell'infrastruttura, rappresentato da un blocco risorsa, l'identificatore per quell'oggetto reale viene salvato nel file `terraform.tfstate`. Questo consentirà l'aggiornamento e la distruzione della risorsa o eventuali modifiche future. Se il

blocco risorsa é già associato ad un elemento dello stato, Terraform confronta la configurazione effettiva dell'oggetto, con i parametri forniti dalla configurazione e se necessario, aggiorna l'oggetto in modo che corrisponda alla configurazione.
É possibili inoltre, utilizzare le informazioni di una risorsa per configurare altre risorse. Utilizzando la sintassi `<RESOURCE TYPE>.<NAME>.<ATTRIBUTE>` andiamo a fare riferimento a un attributo di una determinata risorsa. In questo modo possiamo andare a recuperare le informazioni che non possono essere conosciute fino a quando la risorsa non viene effettivamente creata, andando quindi a creare una dipendenza implicita tra due risorse. Terraform analizza qualsiasi espressione all'interno di un blocco risorsa, per trovare riferimenti ad altre risorse e trasforma questi riferimenti in requisiti di ordinamento impliciti durante la creazione, l'aggiornamento o la distruzione delle risorse.

```
resource "tls_private_key" "global_key" {
  algorithm = "RSA"
  rsa_bits  = 2048
}

resource "aws_key_pair" "test" {
  key_name   = "test-key"
  public_key      = tls_private_key.global_key.public_key_openssh
}
```

Source Code 3: In questo esempio vediamo come la risorsa `aws_key_pair.test` dipenda dal risultato di `tls_private_key.global_key`

La maggior parte delle risorse di una configurazione non ha alcuna dipendenza particolare oppure é sempre implicita e Terraform puó lavorare in modo parallelo. Tuttavia, alcune dipendenze non possono essere riconosciute, da Terraform, implicitamente nella configurazione. In questi casi, possiamo creare una dipendenza esplicata utilizzando il meta-argomento `depend_on`.

```
resource "aws_s3_bucket" "bucket" {
  acl    = "private"
}

resource "aws_instance" "vm" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  depends_on = [aws_s3_bucket.bucket]
}
```

Source Code 4: In questo esempio vediamo come la risorsa `aws_instance.vm` dipenda, in modo esplicito, dal risultato di `aws_s3_bucket.bucket`

Data Sources Un blocco di dati richiede che Terraform legga da una determinata sorgente dati. In questo modo possiamo leggere informazioni dall'esterno e importarle per utilizzarle in Terraform.

```
data "aws_ami" "example" {
  most_recent = true
```

```
3
4   owners = ["self"]
5 }
```

Source Code 5: Esempio di dichiarazione di un data sources

> I *Data Sources* hanno lo stesso comportamento e vengono trattati da Terraform come se fossero delle *Resources*. L'unica differenza é che i blocchi di dati vengono effettivamente eseguiti durante il comando `terraform plan` per rilevare eventuali difformità. Per questo motivo d'ora in avanti verranno considerati come se fossero delle *Resources*, se non diversamente menzionato.

Provisioners I *provisioners* possono essere utilizzati per effettuare azioni specifiche sulla macchina locale o su una remota e posso essere creati solamente all'interno di un blocco risorse.

```
1 resource "aws_instance" "vm" {
2   ami           = "ami-a1b2c3d4"
3   instance_type = "t2.micro"
4
5   provisioner "local-exec" {
6     command = "echo The server's IP address is ${self.private_ip}."
7   }
8 }
```

Source Code 6: Esempio di dichiarazione di un provisioner

Per impostazione predefinita, i *provisioners* vengono eseguiti quando viene creata la risorsa in cui sono definiti e solamente durante l'esecuzione del comando `terraform apply`.

Se un provisioner fallisce, la risorsa a cui é associato viene contrassegnata come "contaminata". Una risorsa contaminata sarà pianificata per la distruzione e successivamente verrá riscreata al prossimo comando `terraform apply`. Terraform esegue questa procedura perché ritiene di aver lasciato la risorsa contaminata in uno stato "parzialmente configurata" e quindi l'unico modo per garantire la corretta creazione di una risorsa contaminata è la sua ricreazione. É possibile però disabilitare questo comportamento tramite il meta-argomento `on_failure`.

É possibile dire a un *provisioner* di essere eseguito durante la fase di distruzione, invece che di creazione, mediante il meta-argomento `when = destroy`. Durante la fase di distruzione i *provisioner* vengono eseguiti prima che la risorsa venga distrutta. Se falliscono, Terraform si bloccherà con un errore.

```
1 resource "aws_instance" "vm" {
2   ami           = "ami-a1b2c3d4"
3   instance_type = "t2.micro"
4
5   provisioner "local-exec" {
6     when        = destroy
7     command = "echo The server's ${self.private_ip} will be destroyed."
8   }
9 }
```

Source Code 7: Esempio di dichiarazione di un provisioner che verrà eseguito in fase di distruzione

È possibile specificare più *provisioners* all'interno di una risorsa, quest'ultimi verranno eseguiti nell'ordine in cui sono definiti nel file di configurazione. É possibile combinare i *provisioners* di creazione e distruzione nella stessa risorsa, si occuperà Terraform di eseguire solo quelli validi il comando in esecuzione.

```
resource "aws_instance" "vm" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo first"
  }

  provisioner "local-exec" {
    command = "echo second"
  }
}
```

Source Code 8: Esempio di dichiarazione di un multi-provisioner

> Terraform sconsiglia l'utilizzo dei *provisioners* e consiglia di utilizzarli come ultima risorsa, quando non si hanno altre alternative.

Modules I moduli sono dei contenitori di risorse, ovvero consistono in una raccolta di file `.tf` presenti in una directory.
I moduli sono l'unico modo per creare pacchetti di codice riutilizzabile, mettendo insieme le configurazioni delle risorse con Terraform. Esiste sempre almeno un modulo, noto come *root module*, che consiste nelle risorse definite nei file `.tf` nella directory dove viene eseguito Terraform.
Un modulo Terraform può chiamare altri moduli per includere le loro risorse nella configurazione, questi sono chiamati *child modules* e possono essere chiamati più volte all'interno della stessa configurazione o con configurazioni diverse.

```
module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "17.1.0"
  # insert the 9 required variables here
}
```

Source Code 9: Esempio di dichiarazione del *child modules* `terraform-aws-eks` v17.1.0 [3]

Durante l'esecuzione di un modulo, Terraform lo tratta come una scatola nera, quindi conoscendo gli input sappiamo quali output aspettarci ma il come quell'input viene trasformato

in quell'output, non é dato sapersi. Lo stesso vale per le risorse all'interno del modulo che non hanno alcuna conoscenza dell'ambiente esterno. Per questo motivo é molto importante progettare bene gli input e gli output durante la creazione di un modulo.

```
variable "name" {
  description = "The name of the virtual machine created"
  type          = string
  default      = "pippo"
}
```

Source Code 10: Esempio di dichiarazione di un input per un modulo

```
output "private_ip" {
  description = "The created virtual machines private ip"
  value         = aws_instance.vm.private_ip
}
```

Source Code 11: Esempio di dichiarazione di un output per un modulo

I moduli possono essere caricati dal filesystem locale oppure Terraform può scaricare i moduli dal registro ufficiale[2] o da uno privato. Ciò rende possibile pubblicare i propri moduli che altri potranno utilizzare o utilizzare i moduli che altri hanno pubblicato.

1.2 Il parallelismo in Terraform

I grafici sono strutture matematiche utilizzate per modellare le relazioni presenti tra gli oggetti. Possiamo quindi pensare alle moderne infrastrutture come grafici: le risorse computazionali normalmente dipendono dalle risorse di rete, che a loro volta dipendono dalle risorse di sicurezza, etc... Sfruttando questa teoria Terraform crea un grafico per modellare le relazioni tra le risorse, in modo che i provider possano gestire e modificare in sicurezza le risorse dell'infrastruttura, prevedendo, in questo modo, i cambiamenti con conseguenze impreviste. Un altro vantaggio implicito nell'utilizzare Terraform é la possibilità di avere una mappa di tutte le relazioni dell'infrastruttura, cosa impossibile su larga scala. Utilizzando questa filosofia Terraform é in grado di eseguire in modo parallelo piú operazioni senza incorrete in rischi.

1.2.1 I tipi di nodo

Ci sono solo tre tipi di nodi che possono esistere all'interno del grafico creato da Terraform:

- *Nodo risorsa:* rappresenta una singola *resource*;
- *Nodo di configurazione del provider:* rappresenta il tempo necessario per configurare completamente un *provider*. Questo viene creato quando é presente un blocco di configurazione del provider;
- *Meta-nodo risorsa:* rappresenta un gruppo di risorse, ma non rappresenta alcuna azione da solo. Questo viene fatto per comodità sulle dipendenze e per creare un grafico più semplice. Questo nodo è presente solo per le risorse che hanno un parametro `count/for_each` maggiore di 1.

1.2.2 Costruzione del grafo

La costruzione del grafico avviene in una serie di passaggi sequenziali:

1. I *nodi risorsa* vengono aggiunti in base alla configurazione rilevata automaticamente. Se è presente una difformità rispetto al piano o allo stato fornito, vengono aggiunti dei metadati a ciascun *nodo risorsa*, in modo da tener traccia del cambiamento;

2. Le risorse vengono mappate ai *provisioners*, se questi sono stati definiti. Questa operazione viene eseguita dopo la creazione di tutti i *nodi risorsa*, in modo che le risorse con lo stesso tipo di *provisioner* possano condividere la stessa implementazione;

3. Vengono create le dipendenze esplicite, quindi quelle con esplicitato il parametro `depend_on`;

4. Se è presente uno stato, tutte le risorse "orfane" vengono aggiunte al grafico. Per "risorse orfane" si intende tutte le risorse che non sono più presenti nella configurazione ma presenti nel file di stato `terraform.tfstate`. Alle "risorse orfane" non è associata alcun *provider*, poiché la configurazione di quest'ultimo non viene memorizzata nel file di stato `terraform.tfstate`;

5. Le risorse vengono mappate ai *provider* definiti nella configurazione. I *nodi di configurazione del provider* vengono creati per i *provider* e vengono creati degli archi in modo tale che le risorse dipendano dalla configurazione del rispettivo *provider*;

6. Terraform, al fine di creare le dipendenze implicite, analizza l'attuale configurazione di risorse e provider per individuarne le dipendenze. Per fare ció trasforma gli attributi della risorsa in dipendenze, creando un arco con la risorsa a cui fa riferimento;

7. Viene creato il nodo radice, detto *root*. Il nodo radice punta a tutte le risorse e viene creato in modo che ci sia unico. Durante l'attraversamento del grafico il nodo radice viene ignorato. Insieme al nodo radice vengono creati anche una serie di nodi per le operazioni interne di Terraform, tutti questi nodi iniziano con il prefisso `[root]`;

8. Terraform attraversa tutti i *nodi risorsa* e gli analizza per trovare quelli che devono essere distrutti. Questi *nodi risorsa* vengono divisi in due: un nodo che distruggerà la risorsa e un altro che creerà la risorsa (se questa deve essere ricreata). Questo viene fatto perché l'ordine di distruzione è spesso diverso dall'ordine di creazione e quindi non possono essere rappresentati da un singolo nodo del grafico;

9. Il grafico viene convalidato per verificare che non ci siano cicli e che contenga un unico nodo radice.

1.2.3 Controllo dei cicli

Per controllare la presenza di cicli Terraform utilizza l'Algoritmo di Tarjan, come possiamo vedere dal suo codice sorgente [4], liberamente accessibile su Github.
L'algoritmo di Tarjan , prende il nome del suo inventore Robert Tarjan e l'idea si basa sul fatto che la ricerca Deep-First Search (DFS) produce in output un albero/foresta. Si potrà quindi andare a trovare le componenti Strongly connected component (SSC), come sotto-alberi dell'albero/foresta di partenza. Qualsiasi vertice che non fa parte di un ciclo forma, da solo, una SSC: per esempio, un qualsiasi vertice di un grafo aciclico. Per fare ció, a ogni nodo è assegnato un indice, in base alla profondità della ricerca. Durante la ricerca DFS viene calcolato anche il valore *lowlink*, ovvero l'indice che deve avere per essere raggiungibile dalla radice. Quindi un

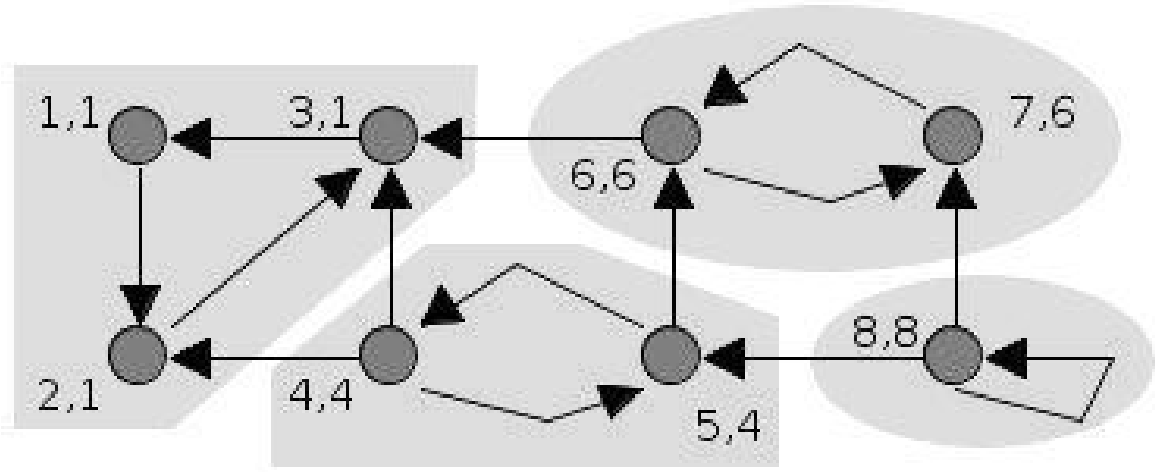


Figura 2: Animazione dell'applicazione dell'algoritmo di Tarjan su un grafo
Fonte: `https://wikipedia.org/wiki/File:Tarjan%27s_Algorithm_Animation.gif`

vertice é la radice di una SSC se e solo se il suo indice corrisponde al *lowlink* degli altri nodi. Nel caso peggiore la complessità é pari a $O(|E| + |V|)$.

1.2.4 Attraversamento del grafo

Per percorrere il grafico, viene eseguito un attraversamento con un algoritmo DFS eseguito in parallelo. In questo modo un nodo puó essere percorso non appena vengono percorse tutte le sue dipendenze.
La quantità di parallelismo è limitata a un valore massimo, impostato utilizzando il flag `-parallelism` da linea di comando, per evitare che troppe operazioni simultanee sovraccaricano le risorse della macchina che esegue Terraform.

> É bene precisare che l'attraversamento del grafo avviene in modo parallelo e non concorrente, come precisa anche il suo creatore Mitchell Hashimoto [5].

1.2.5 Il comando `graph`

Il comando `terraform graph` viene utilizzato per generare una rappresentazione visiva, in formato DOT, della configurazione attuale.
Il flag `-type` può essere usato per controllare il tipo di grafico mostrato. Terraform crea grafici in base al tipo di operazione: `plan`, `destroy`, `apply`, `validate`, `import`, `refresh`. Un argomento molto utile é `-draw-cycles` che va ad evidenziare tutti i cicli presenti nel grafico.

1.2.6 Esempio di grafo

Vediamo insieme un esempio di output del comando `terraform graph`. Per questo esempio é stata utilizzata la versione 1.0.3 della terraform cli, disponibile qui [6], e la versione 3.52.0 del provider terraform Amazon Web Services (AWS), disponibile qui [7].

Prendiamo il seguente codice come esempio, andando a creare un file `demo.tf`:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "3.52.0"
    }
  }
}

provider "aws" {
  region = "us-west-2a"
}

data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  owners = ["099720109477"]
}

resource "aws_instance" "vm" {
  ami               = data.aws_ami.ubuntu.id
  availability_zone = "us-west-2a"
  instance_type     = "t2.micro"
}

resource "aws_ebs_volume" "disk" {
  availability_zone = "us-west-2a"
  size              = 1
}

resource "aws_volume_attachment" "ebs_att" {
  device_name = "/dev/sdh"
  volume_id   = aws_ebs_volume.disk.id
  instance_id = aws_instance.vm.id
}
```

Source Code 12: Esempio di codice terraform

Di seguito la spiegazione passo passo del codice sopra mostrato:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "3.52.0"
    }
  }
}
```

Source Code 13: Blocco di codice `required_providers`

In questo esempio con il blocco di codice 13 andiamo a fornire a Terraform una lista di *providers*, andando a specificare da quale registro reperire il codice sorgente e l'eventuale versione che vogliamo utilizzare. Questo blocco di codice é opzionale e se non valorizzato terraform troverà in automatico i *providers*, andando a prendere sempre la versione *lastest* dal registro ufficiale[2].

```
provider "aws" {
  region = "us-west-2a"
}
```

Source Code 14: Blocco di codice `provider`

Nel blocco di codice 14 andiamo a inizializzare il provider AWS, in questo esempio andiamo a definire solo il parametro obbligatorio `region`. Se si desidera provare ad effettuare un `terraform apply` é necessario autenticarsi tramite variabili d'ambiente come descritto qui [8].

```
data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  owners = ["099720109477"]
}
```

Source Code 15: Blocco di codice `aws_ami.ubuntu`

Tramite il blocco di codice 15 indichiamo a Terraform di reperire delle informazioni da AWS, riguardanti una precisa versione di ubuntu. Questi dati ci serviranno poi per avere il corretto AMI da passare alla virtual machine che creeremo.

```
resource "aws_instance" "vm" {
  ami               = data.aws_ami.ubuntu.id
  availability_zone = "us-west-2a"
  instance_type     = "t2.micro"
}
```

Source Code 16: Blocco di codice `aws_instance.vm`

Tramite il blocco di codice 16 diciamo a Terraform che vogliamo una virtual machine di tipo *t2.micro*[9] nella zona *us-west-2a*. Per il sistema operativo utilizziamo l'AMI ottenuto dal blocco di codice precedente, andando in questo modo a creare una dipendenza implicita tra le due risorse.

```
resource "aws_ebs_volume" "disk" {
  availability_zone = "us-west-2a"
  size              = 1
}
```

Source Code 17: Blocco di codice `aws_ebs_volume.disk`

Tramite il blocco di codice 17 diciamo a Terraform che vogliamo un disco di 1GB nella zona *us-west-2a*.

```
resource "aws_volume_attachment" "ebs_att" {
  device_name = "/dev/sdh"
  volume_id   = aws_ebs_volume.disk.id
  instance_id = aws_instance.vm.id
}
```

Source Code 18: Blocco di codice `aws_volume_attachment.ebs_att`

Tramite il blocco di codice 18 diciamo a Terraform che vogliamo attaccare il disco creato alla macchina virtuale che abbiamo creato.

Nella stessa directory dove abbiamo creato il file `demo.tf` andiamo a lanciare il comando `terraform init` per inizializzare la cartella e scaricare il codice necessario per eseguire il provider AWS. Una volta che il comando `terraform init` é terminato, possiamo andare a vedere il grafico che Terraform genera, utilizzando il comando `terraform graph`.

```
digraph {
        compound = "true"
        newrank = "true"
        subgraph "root" {
                "[root] aws_ebs_volume.disk (expand)" [label = "aws_ebs_volume.disk", shape = "box"]
                "[root] aws_instance.vm (expand)" [label = "aws_instance.vm", shape = "box"]
                "[root] aws_volume_attachment.ebs_att (expand)" [label = "aws_volume_attachment.ebs_att", shape = "box"]
                "[root] data.aws_ami.ubuntu (expand)" [label = "data.aws_ami.ubuntu", shape = "box"]
                "[root] provider[\"registry.terraform.io/hashicorp/aws\"]" [label = "provider[\"registry.terraform.io/hashicorp/aws\"]", shape = "diamond"]
                "[root] aws_ebs_volume.disk (expand)" -> "[root] provider[\"registry.terraform.io/hashicorp/aws\"]"
                "[root] aws_instance.vm (expand)" -> "[root] data.aws_ami.ubuntu (expand)"
                "[root] aws_volume_attachment.ebs_att (expand)" -> "[root] aws_ebs_volume.disk (expand)"
                "[root] aws_volume_attachment.ebs_att (expand)" -> "[root] aws_instance.vm (expand)"
                "[root] data.aws_ami.ubuntu (expand)" -> "[root] provider[\"registry.terraform.io/hashicorp/aws\"]"
                "[root] meta.count-boundary (EachMode fixup)" -> "[root] aws_volume_attachment.ebs_att (expand)"
                "[root] provider[\"registry.terraform.io/hashicorp/aws\"] (close)" -> "[root] aws_volume_attachment.ebs_att (expand)"
                "[root] root" -> "[root] meta.count-boundary (EachMode fixup)"
                "[root] root" -> "[root] provider[\"registry.terraform.io/hashicorp/aws\"] (close)"
        }
}
```

Source Code 19: Risultato del comando `terraform graph`

Di per se questo output non é molto significativo, infatti vediamo come con solamente sei blocchi di codice abbiamo un output di venti righe davvero poco leggibili. Fortunatamente é possibile convertire questo output, che ricordiamo essere nel formato standard DOT, in un output grafico. Per farlo possiamo usare uno dei tanti tool da linea di comando oppure online. Per questo approfondimento ho utilizzato il tool da linea di comando di Graphviz [10].

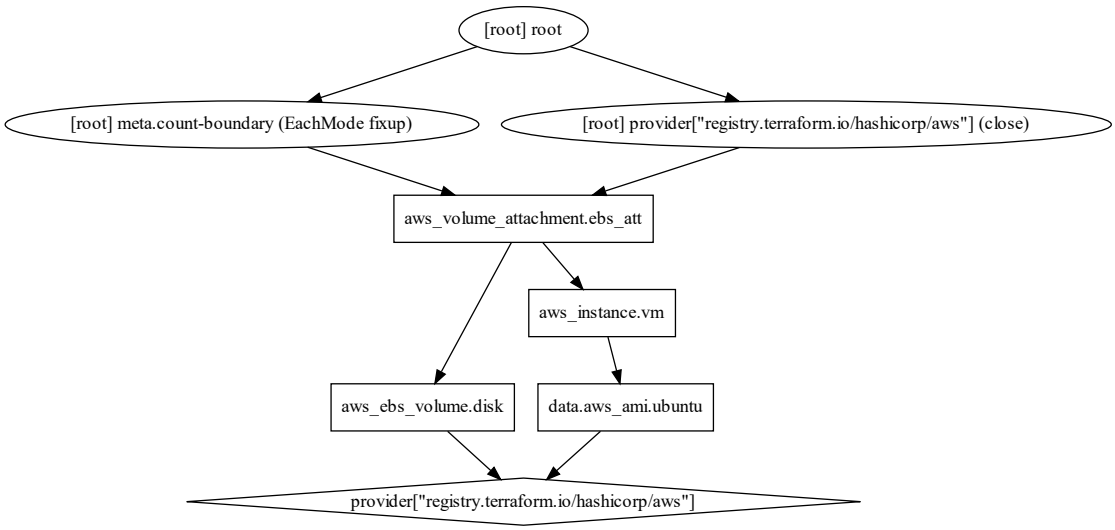


Figura 3: Visualizzazione grafica del risultato del comando `terraform graph`

Dal grafico notiamo subito due nodi con il prefisso `[root]` collegati al *nodo di root*. Questi sono dei nodi che ha aggiunto Terraform, infatti sono operazioni interne di terraform che questo andrà a fare e che in futuro verranno nascoste o avranno un output apposito per essere mostrare [11].
Simulando il comportamento di Terraform partiamo dal nodo
`[root] provider["registry.terraform.io/hashicorp/aws"]` dove avviene l'inizializzazione del provider. Successivamente in modo parallelo vengono eseguiti i nodi `aws_ebs_volume.disk` e `data.aws_ami.ubuntu`, al termine di quest'ultimo viene risolta la dipendenza per il nodo `aws_instance.vm` e viene quindi eseguito. Risolte tutte le dipendenze precedenti viene eseguito l'ultimo nodo `aws_volume_attachment.ebs_att`.

1.3 Conclusione

Aspetti Positivi

- Crea o modifica l'infrastruttura in modo rapido e affidabile;
- Favorisce il riutilizzo del codice tramite l'utilizzo di moduli;
- Possiede un'ottima community che da supporto e pubblica molte soluzione sul registro ufficiale [2];
- Da la possibilità di vedere in anteprima i cambiamenti, grazie al comando `terraform plan`.

Aspetti Negativi

- Nessuna funzionalità di rollback, per realizzarlo é necessario tornare alla versione precedente del codice;
- ll linguaggio HCL é semplice ma molto limitato, spesso per semplici azioni é necessario scrivere diverse linee di codice;
- L'output degli errori é spesso criptico o impreciso.

In sostanza Terraform é ancora un linguaggio molto giovane e poco flessibile, tuttavia non ci sono altre alternative in grado di garantire lo stesso livello di servizio ed affidabilità.

Glossario

Amazon Web Services è un'azienda statunitense di proprietà del gruppo Amazon, che fornisce servizi di cloud computing [13].

Deep-First Search è un algoritmo di ricerca su alberi e grafi. Il nome deriva dal fatto che in un albero, ancora prima di avere visitato i nodi vicino alla radice, l'algoritmo può ritrovarsi a visitare vertici lontani dalla radice, andando così "in profondità" [13].

DOT è un linguaggio di descrizione del grafico. I grafici DOT sono in genere file con l'estensione del nome file `.gv` o `.dot`. Normalmente viene utilizzata l'estensione `.gv`, per evitare confusione con l'estensione `.dot` utilizzata dalle versioni di Microsoft Word prima del 2007. [13].

Go è un linguaggio di programmazione open source sviluppato da Google. Il lavoro su Go nacque nel settembre 2007 da Robert Griesemer, Rob Pike e Ken Thompson [13].

HashiCorp è una società di software con sede a San Francisco, California. HashiCorp fornisce strumenti open source e prodotti commerciali che consentono a sviluppatori, operatori e professionisti della sicurezza di fornire, proteggere, eseguire e connettere l'infrastruttura di cloud computing. È stata fondata nel 2012 da Mitchell Hashimoto e Armon Dadgar [13].

Infrastructure as Code è il processo di gestione e provisioning dei data center e computer tramite file di definizione leggibili, piuttosto che un processo di configurazione hardware fisico o tramite strumenti di configurazione interattivi [13].

Strongly connected component un grafo orientato è fortemente connesso se esiste un percorso tra tutte le coppie di vertici [13].

Acronimi

AMI Amazon Machine Images.

AWS Amazon Web Services.

DFS Deep-First Search.

HCL HashiCorp Configuration Language.

IaC Infrastructure as Code.

SSC Strongly connected component.

Riferimenti bibliografici

[1] "Terraform homepage." `https://www.terraform.io`. Accessed: 7 settembre 2021.

[2] "Terraform registry homepage." `https://registry.terraform.io`. Accessed: 7 settembre 2021.

[3] "Modulo ufficale Terraform per EKS su AWS v17.1.0." `https://registry.terraform.io/modules/terraform-aws-modules/eks/aws/17.1.0`. Accessed: 7 settembre 2021.

[4] "Source code dell'algoritmo di Tarjan utilizzato da Terraform." `https://github.com/hashicorp/terraform/blob/2f152f1139971c6c32b0d4f65aec0a392ba48d68/internal/dag/tarjan.go`. Accessed: 7 settembre 2021.

[5] "Issue Github 11766." `https://github.com/hashicorp/terraform/issues/11766`. Accessed: 7 settembre 2021.

[6] "Terraform cli v1.0.3." `https://releases.hashicorp.com/terraform/1.0.3/`. Accessed: 7 settembre 2021.

[7] "Provider ufficale Terraform per AWS v3.52.0." `https://registry.terraform.io/providers/hashicorp/aws/3.52.0`. Accessed: 7 settembre 2021.

[8] "Autenticazione per il provider ufficale Terraform per AWS v3.52.0." `https://registry.terraform.io/providers/hashicorp/aws/3.52.0/docs#environment-variables`. Accessed: 7 settembre 2021.

[9] "Amazon t2." `https://aws.amazon.com/ec2/instance-types/t2/`. Accessed: 7 settembre 2021.

[10] "Graphviz homepage." `https://www.graphviz.org/`. Accessed: 7 settembre 2021.

[11] "Issue Github 14511." `https://github.com/hashicorp/terraform/issues/14511`. Accessed: 7 settembre 2021.

[12] "Documentazione Terraform." `https://www.terraform.io/docs`. Accessed: 7 settembre 2021.

[13] "Wikipedia homepage." `https://wikipedia.org/`. Accessed: 7 settembre 2021.

[14] L. Aceto, A. Ingolfsdottir, K. Larsen, and J. Srba, *Reactive systems*. Cambridge University Press, 2007.