Vultr Product > Guide

How to Establish a Private Connection Between Vultr and Amazon Web Services (AWS)

Learn how to set up a secure private connection between Vultr and AWS cloud platforms with our step-by-step guide for seamless cross-cloud networking.

Contents

01	Introduction	3
02	Prerequisites	3
03	Peer Vultr and AWS Using Headscale	4
04	Provision a Private Network Mesh Between Vultr and AWS	6
05	Test and Benchmark the Private Network Mesh	12
06	Conclusion	14

Introduction

Hybrid cloud deployments often require secure, private connectivity across providers. Traditional VPNs rely on complex configurations and centralized gateways, which can introduce bottlenecks and single points of failure.

Headscale is an open-source, self-hosted control server for **Tailscale**, a mesh VPN built on the **WireGuard** protocol. Together, they enable peer-to-peer, encrypted tunnels between nodes without relying on a central gateway, giving you full control over authentication, device management, and network policies while maintaining low-latency, secure connectivity.

By deploying gateway nodes in each cloud and automating configuration with **Terraform** and **Ansible**, you can:

- Enable private, encrypted communication between **Vultr** and **AWS** workloads.
- Maintain control over your network using self-hosted infrastructure.
- Automate repeatable, scalable deployments.

This guide walks you through setting up Headscale to establish a private connection between Vultr and AWS. The implementation leverages Terraform and Ansible to automate provisioning and configuration, minimizing manual effort and ensuring consistent deployments.

Prerequisites

Before you begin, you need to:

- Have access to a Linux-based instance as a non-root user with sudo privileges.
- Install the command-line tools needed for automation: Terraform, Ansible, and Git.
- Install Vultr CLI and AWS CLI to interact with cloud services and list available regions.

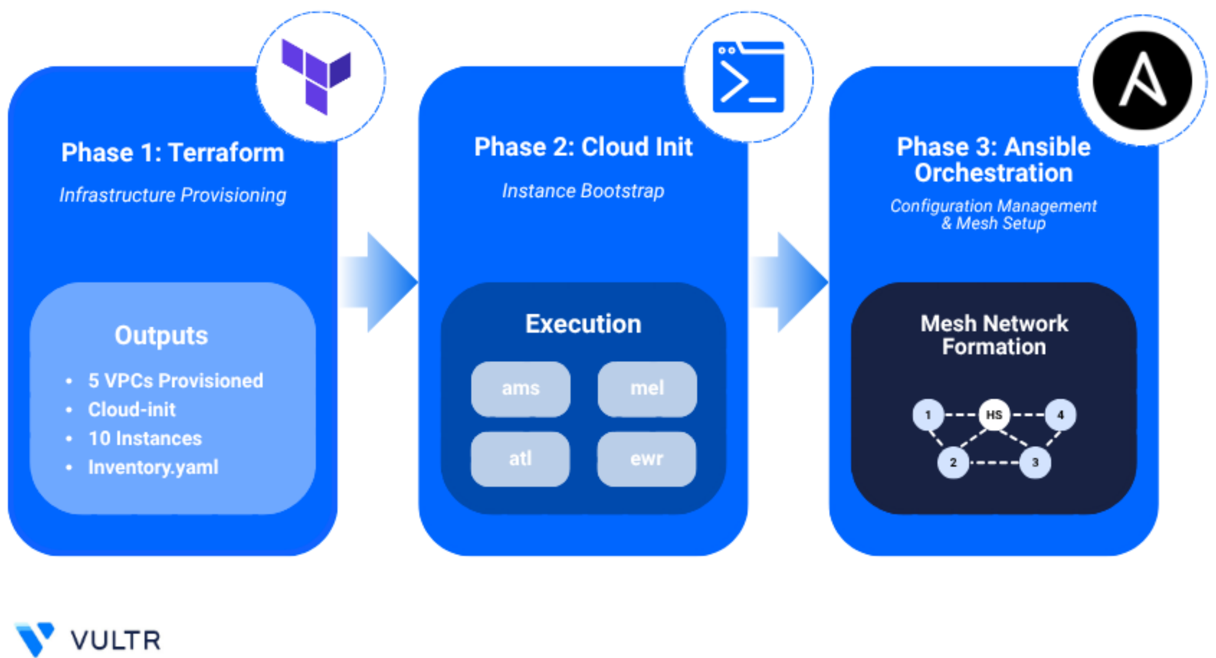
Peer Vultr and AWS Using Headscale

Tailscale with Headscale allows you to self-host a mesh VPN connecting on-prem, site-to-site, or cloud-to-cloud environments. This guide provides instructions to establish a **site-to-site mesh**, where each site is a **Gateway Node** in either a Vultr region or an AWS region. The mesh connects to each cloud’s Virtual Private Cloud (VPC) to expose Client Nodes in each datacenter with the others over a private network.

Components

- **Headscale Control Server**: Self-hosted control server managing Tailscale nodes and coordinating secure connections. In this guide, it runs on a Vultr instance.
- **Virtual Private Cloud**: Isolated network in each cloud (Vultr and AWS) that enables private connectivity between gateway and client nodes.
- **Gateway Node**: A `tailnet` peer or Tailscale client in each Vultr and AWS region to join the mesh network across multiple datacenters.
- **Client Nodes**: Represents example services in each region. For example, a database server in Vultr Bangalore that serves as a backend to an API server in AWS eu-west-1.
- **FRRouting**: Propagates `tailnet` routes from the gateway to Vultr client nodes, enabling cross-datacenter connectivity.
- **AWS Route Table**: Routes for client nodes in AWS are managed by the VPC route table. FRR is not used because AWS's NAT-based architecture overrides OSPF/FRR routes.

Automation With Terraform, Ansible, and Cloud-Init



This implementation is automated using **Terraform**, **Ansible**, and **Cloud-Init**, enabling consistent, repeatable deployments across Vultr and AWS with minimal manual configuration

- **Terraform**: Defines the entire infrastructure including Vultr VPCs, AWS VPCs, gateway/client nodes in both clouds, and the control server.
- **Ansible**: Automates post-provisioning tasks, including configuring the control server, gateway/client nodes and setting up routing rules.
- **Cloud-Init**: Installs Tailscale on first boot, joins the Headscale coordination node, and configures FRR to enable dynamic route propagation across the VPC in each region.
- **Startup Script**: Reusable script for Vultr client nodes (Ubuntu or RHEL) that automates Tailscale installation and mesh onboarding, simplifying scaling and manual enrollment of new Vultr client nodes.

Provision a Private Network Mesh Between Vultr and AWS

This section provides steps to provision the required infrastructure using **Terraform** and configure instances post-deployment with the included **Ansible playbooks**. The following regions are used to demonstrate the peering setup:

Role	Cloud	Region (Code)
Headscale control server	Vultr	Amsterdam (`ams`)
Vultr peer/client nodes	Vultr	Bangalore (`blr`), Frankfurt (`fra`)
AWS peer/client nodes	AWS	EU (Ireland) (`eu-west-1`), US West (N. California) (`us-west-1`)

You can add or remove regions as needed. The setup is flexible and allows multiple Vultr and AWS regions to be included in the mesh network.

Configure terraform.tfvars

1. Clone the GitHub repository.

   CONSOLE
   ```console
   $ git clone https://github.com/vultr-marketing/code-samples.git
   $ cd code-samples/vpc-peering/vultr-aws-regions/terraform
   ```

2. Update the `terraform.tfvars` file with your credentials.

   CONSOLE
   ```console
   $ nano terraform.tfvars
   ```

   Populate your credentials and adjust regions if needed.

HCL

```
# Vultr API Key
vultr_api_key = "YOUR_VULTR_API_KEY"

# AWS Access Keys
aws_access_key = "YOUR_AWS_ACCESS_KEY"
aws_secret_key = "YOUR_AWS_SECRET_KEY"

user_scheme  = "limited"

instance_plan    = "voc-c-2c-4gb-75s-amd"
instance_os_id   = 1743

headscale_region = "ams"

vultr_regions = {
  vultr_1 = {
    region         = "blr"
    v4_subnet      = "10.30.0.0"
    v4_subnet_mask = 24
  }
  ...
}

aws_regions = {
  aws_1 = {
    region        = "eu-west-1"
    instance_type = "t3.micro"
    ami_id        = "ami-00f569d5adf6452bb"
    vpc_cidr      = "10.50.0.0/16"
    public_subnet = "10.50.1.0/24"
  }
  ...
}
```

> Note
>
> Populate the Vultr API Key and AWS Access Keys in the `terraform.tfvars` file to enable API access for provisioning resources.

Modifying or Expanding Regions

Understanding how to adjust regions helps maintain network performance and allows scalable deployments.

- **Update regions:** Update the `vultr_regions` and `aws_regions` maps to change deployment locations for gateway or client nodes.
- **Headscale server location:** Change `headscale_region` to deploy the control server in a new region. The `instance_plan` and `instance_os_id` values are pre-selected to achieve maximum performance and ensure software compatibility.
- **Add regions:** Add entries to `vultr_regions` or `aws_regions` maps. Terraform automatically loops through all defined regions and provisions instances.
- **AWS AMI per region:** Set the `ami_id` for each AWS region to ensure the instances use the correct Ubuntu version. AWS provides different AMI IDs for the same OS in each region. This setup has been tested on **Ubuntu 22.04**, and using this version is recommended for consistency, security, and performance.

> Important
>
> **Tips for Region Selection**:
>
> - List available Vultr regions using the API to choose appropriate datacenter locations for your instances.
>
> CONSOLE
>
> ```console
> $ curl -X GET -s https://api.vultr.com/v2/regions | jq '.regions[] | "\(.id) - \(.city), \(.country)"'
> ```
>
> - List all AWS regions with the AWS CLI to determine which regions to deploy your VPCs and instances.
>
> CONSOLE
>
> ```console
> $ aws ec2 describe-regions --all-regions --query 'Regions[].{Name:RegionName,OptInStatus:OptInStatus}' --output table
> ```

> - Fetch the latest Ubuntu 22.04 AMI ID for a specific AWS region to ensure your instances use the up-to-date OS.
>
> CONSOLE
>
> ```
> $ aws ec2 describe-images --region ca-central-1 --owners 099720109477 \
>   --filters "Name=name,Values=ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*" \
>           "Name=state,Values=available" \
>   --query "Images | sort_by(@, &CreationDate)[-1].ImageId" --output text
> ```

Initialize and Apply Terraform

1. Initialize the Terraform working directory.

   CONSOLE

   ```
   $ terraform init
   ```

   The command above sets up Terraform and downloads any required providers.

2. Review the execution plan.

   CONSOLE

   ```
   $ terraform plan
   ```

3. Apply the Terraform manifests.

   CONSOLE

   ```
   $ terraform apply
   ```

   Your output should look similar to the one below:

   ```
   Do you want to perform these actions?
    Terraform will perform the actions described above.
   ```

```
  Only 'yes' will be accepted to approve.

  Enter a value:
```

When prompted, type **yes** to approve and apply the configuration. Provisioning may take up to 5 minutes depending on the number of regions.

Outputs:

```
client_summary = {
  "blr" = {
    "name" = "vultr-client-blr"
    "private_ip" = "10.30.0.3"
    "provider" = "vultr"
    "public_ip" = "192.18.0.16"
    "subnet" = "10.30.0.0/24"
  }
  ...
  "us-west-1" = {
    "name" = "aws-client-us-west-1"
    "private_ip" = "10.51.1.9"
    "provider" = "aws"
    "public_ip" = "192.8.0.45"
    "subnet" = "10.51.1.0/24"
  }
}
gateway_summary = {
  "blr" = {
    "name" = "vultr-peer-blr"
    "private_ip" = "10.30.0.4"
    "provider" = "vultr"
    "public_ip" = "192.35.0.10"
    "subnet" = "10.30.0.0/24"
  }
  ...
  "us-west-1" = {
    "name" = "aws-peer-us-west-1"
    "private_ip" = "10.51.1.142"
    "provider" = "aws"
    "public_ip" = "192.3.0.7"
    "subnet" = "10.51.1.0/24"
  }
```

```
}
headscale_control_server_ip = "192.2.0.5"
```

These outputs provide important information about your deployed infrastructure:

- **Client and Gateway Summaries**
  - `client_summary`: Contains details of all client nodes (Vultr and AWS), including Name, Private IP, Public IP, Provider, and Subnet.
  - `gateway_summary`: Contains details of all gateway nodes with the same information.
- **Headscale Control Server IP**
  - `headscale_control_server_ip`: Shows the public IP of the Headscale control server, which coordinates the mesh network.

> Note
>
> Terraform generates a private key file, `id_rsa`, inside the `terraform` directory. You can use this key to SSH into the gateway and client nodes for verification or debugging.

Run Ansible Playbooks

1. Navigate to the Ansible directory:

CONSOLE

```console
$ cd ../ansible
```

> Note
>
> Terraform automatically generates an `inventory.yml` file in this directory. Ansible uses this file to configure the mesh network, listing all Vultr and AWS gateway and client nodes.

2. Run the Ansible Playbook to configure the private network mesh.

CONSOLE

```console
$ ansible-playbook playbook.yml
```

This playbook uses the generated `inventory.yml` file to identify instances across Vultr and AWS and configures the Headscale and Tailscale nodes, creating a fully connected private mesh network.

> Note
>
> To manually add more clients to the mesh network:
>
> - **Vultr:** During instance provisioning, select the `vultr-x-aws-mesh-script` startup script and choose the corresponding VPC under **VPC Networks**. The script automatically configures the instance and adds all OSPF-announced routes. This startup script currently supports only Debian and RHEL instances.
> - **AWS:** When provisioning, select the respective VPC. No additional installation is required because the **VPC Route Table** already contains the necessary routes.

Test and Benchmark the Private Network Mesh

After deployment, verify that all regions across Vultr and AWS are connected through the **WireGuard-based** mesh managed by **Headscale**. Use Ansible to benchmark both private and public paths, measuring latency, bandwidth, and overall mesh performance.

1. Run the provided Ansible playbook to execute `iperf3` and `ping` tests between regions using both **private** and **public IPs**.

CONSOLE

```console
$ ansible-playbook network_tests.yml
```

> Note
>
> The network tests may take 5-10 minutes to complete, depending on the number of regions being tested.

After completion, the playbook creates a `network_test_results/` directory which contains the following results files:

- `network_results.txt`: Consolidated output of all tests, including latency and bandwidth for public and private paths.
- `public_results_table.txt`: Tabular summary of region-to-region performance over the public internet.
- `private_results_table.txt`: Tabular summary of private mesh performance via Headscale.

2. Review the test results for the public internet performance between nodes.

CONSOLE

```
$ cat network_test_results/public_results_table.txt
```

Your output should be similar to the one below:

```
 | From             | To                   | Download (Mbps) | Upload (Mbps) |
Latency (ms) |
|----------------|--------------------|----------------|--------------|------
--------|
| vultr-peer-blr  | vultr-peer-fra      | 461.33          | 498.86        |
175.29        |
| vultr-peer-blr  | aws-peer-eu-west-1  | 90.19           | 111.07        |
228.99        |
```

3. Review the Private IP test results to validate the performance over the Tailscale mesh.

CONSOLE

```
$ cat network_test_results/private_results_table.txt
```

Your output should be similar to the one below:

```
 | From             | To                   | Download (Mbps) | Upload (Mbps) |
Latency (ms) |
|----------------|--------------------|----------------|--------------|------
```

```
--------|
| vultr-peer-blr | vultr-peer-fra       | 268.55         | 405.86        |
160.84       |
| vultr-peer-blr | aws-peer-eu-west-1   | 34.66          | 78.64         |
224.87       |
```

> **Note**
> Public tests usually show higher speeds due to direct internet routing and optimized infrastructure. Private mesh tests use encrypted WireGuard tunnels, which add overhead and may reduce throughput.

Conclusion

You have successfully established a secure, multi-region private network between Vultr and AWS using Headscale and Tailscale. Terraform and Ansible automated provisioning and configuration, ensuring consistent deployments across all regions. Gateway and client nodes communicated over encrypted, low-latency tunnels, while FRRouting and VPC route tables provided seamless routes for cross-datacenter connectivity.

VULTR