

How to Establish a Private Connection Between Vultr and Google Cloud Platform (GCP)

Learn how to set up a secure private connection between Vultr and Google Cloud Platform (GCP) with our step-by-step guide for seamless cloud integration.

Contents

01	Introduction	3
02	Prerequisites	3
03	Peer Vultr and GCP Using Headscale	4
04	Provision a Private Network Mesh Between Vultr and GCP	6
05	Test and Benchmark the Private Network Mesh	12
06	Conclusion	13

Introduction

Modern hybrid cloud setups often require seamless, secure communication between services across cloud providers. While traditional VPNs can serve this purpose, they typically involve centralized architectures and complicated routing setups that may limit scalability and introduce latency.

Headscale, an open-source alternative to Tailscale's coordination server, allows you to self-host and manage your own **Tailscale** mesh VPN. Built on **WireGuard**, Tailscale creates direct, encrypted tunnels between endpoints without routing traffic through a central hub, delivering secure, low-latency networking with full control over device access and network rules.

By deploying lightweight gateway nodes in **Vultr** and **Google Cloud Platform (GCP)**, and using **Terraform** and **Ansible** to handle provisioning and configuration, you can:

- Establish a private, encrypted mesh network between workloads across Vultr and GCP.
- Use self-hosted Headscale to manage identity, routing, and ACLs.
- Streamline deployment with automation for consistency and repeatability.

In this guide, you'll build a fully automated setup to connect Vultr and GCP environments using Headscale, enabling reliable peer-to-peer communication without relying on a traditional VPN gateway.

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 [Google Cloud CLI](#) to interact with cloud services and list available regions.

Peer Vultr and GCP Using Headscale

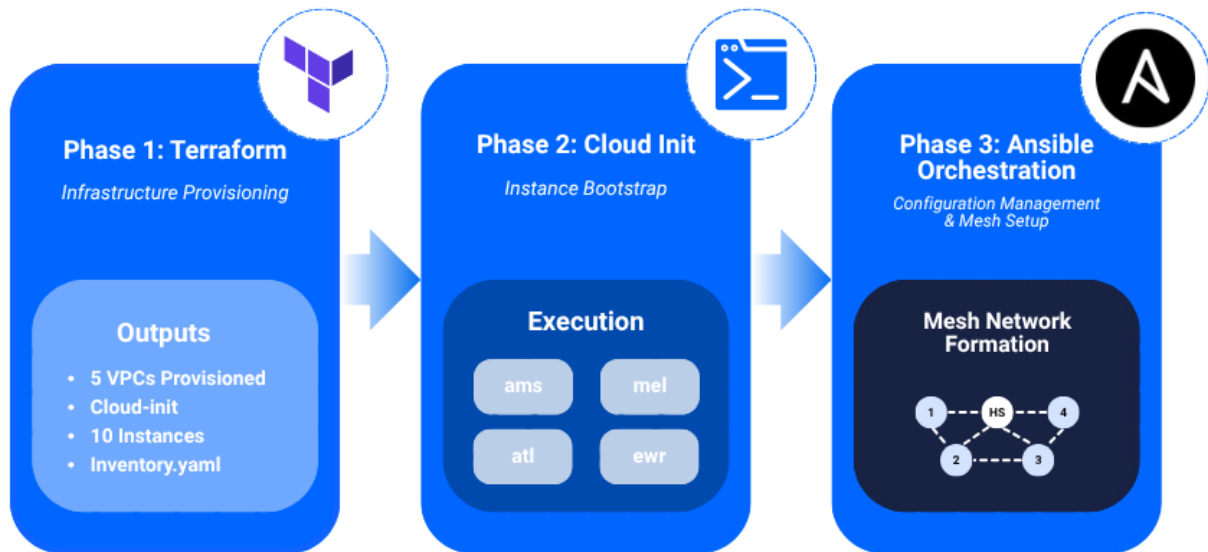
Using **Tailscale** and a self-hosted **Headscale** controller, you can create a fully decentralized mesh VPN across cloud environments. This setup enables direct, encrypted communication between geographically or logically separated networks, including cloud providers like Vultr and GCP.

In this implementation, each cloud provider hosts a **Gateway Node** that participates in the mesh network. These gateways bridge their respective **Virtual Private Clouds (VPCs)**, allowing traffic from internal application servers (Client Nodes) in one cloud to securely reach resources in the other, without exposing public services or relying on complex VPN hubs.

Key Components

- **Headscale Controller:** A self-hosted coordination server for Tailscale, responsible for authenticating peers and managing encrypted tunnels. In this guide, it's hosted on a Vultr instance.
- **VPC Networks:** Private, isolated subnets in both Vultr and GCP, each hosting their own workloads and a gateway node.
- **Gateway Nodes:** A `tailnet` peer or Tailscale client in each Vultr and GCP region to join the mesh network across multiple datacenters..
- **Client Nodes:** Regular application VMs residing in the VPC. They don't run Tailscale; instead, the gateway handles traffic routing to and from the mesh.
- **FRRouting (on Vultr):** A routing daemon that announces and manages routes from the Tailscale gateway to other VMs in the Vultr VPC.

Automation With Terraform, Ansible, and Cloud-Init



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

- **Terraform:** Defines Vultr VPCs, GCP VPCs/subnets, gateway/client nodes on both the cloud platforms, the control server, and network plumbing. It also:
 - Creates GCP custom routes for all Vultr VPC CIDRs with next hop set to the GCP gateway instance in the same VPC.
 - Enables IP forwarding on GCP gateways (`can_ip_forward = true`).
- **Ansible:** Enrolls only gateways (`tailscale_servers`) into Headscale, advertises VPC subnets, approves routes on Headscale, and injects Tailscale routes into the main table while preserving each node's local VPC kernel route.
- **Cloud-Init:** Used on gateways to install Tailscale and configure FRR (on Vultr) to expose the VPC network to the gateway.
- **Startup Script:** A reusable script is available for Vultr client nodes to automate Tailscale installation and onboarding. GCP client nodes do not require Tailscale or a startup script in this setup.

Provision a Private Network Mesh Between Vultr and GCP

In this section, you set up the infrastructure with **Terraform** and configure instances after deployment using the provided **Ansible playbooks**. The following example uses these regions to illustrate the peering mesh:

Role	Cloud Region (Code)
Headscale control server Vultr	Amsterdam (<code>ams</code>)
Vultr peer/client nodes	Vultr Bangalore (<code>blr</code>), Frankfurt (<code>fra</code>)
GCP peer/client nodes	GCP US West (<code>us-west1</code>), Europe West (<code>eu-west1</code>)

You can expand or reduce the regions to match your requirements. The design supports multiple Vultr and GCP locations in the same mesh.

Configure terraform.tfvars

1. Clone the GitHub repository and navigate to the `terraform` directory under `vpc-peering/vultr-gcp-regions`.

CONSOLE

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

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

CONSOLE

```
$ nano terraform.tfvars
```

Populate your credentials and adjust regions if needed.

HCL

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

# GCP Project ID and Service Account Credentials
gcp_project_id      = "YOUR_GCP_PROJECT_ID"
gcp_credentials_file = "path/to/service_account.json"

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
  }
  ...
}

gcp_regions = {
  gcp_1 = {
    region          = "us-west1"
    zone            = "us-west1-a"
    subnet_cidr     = "10.40.0.0/24"
    machine_type    = "e2-micro"
  }
  ...
}
```

Note

Populate the [Vultr API Key](#) and [GCP Service Account Key](#) in the `terraform.tfvars` file to enable API access for provisioning resources.

Modifying or Expanding Regions

Adjusting regions allows you to scale deployments and optimize network performance.

- **Change regions:** Edit the `vultr_regions` and `gcp_regions` maps to redefine where gateway and client nodes deploy.

- **Control server location:** Update `headscale_region` to move the Headscale server to a different region. The `instance_plan` and `instance_os_id` values are pre-selected to achieve maximum performance and ensure software compatibility.
- **Add new regions:** Insert additional entries in the `vultr_regions` or `gcp_regions` maps. Terraform automatically iterates through all listed regions to provision nodes.

This setup has been validated on **Ubuntu 22.04**, which is recommended for stability, security, and consistent results.

i Note

Tips for Region Selection:

- List available Vultr regions using the API to choose appropriate datacenter locations for your instances.

CONSOLE

```
$ curl -X GET -s https://api.vultr.com/v2/regions | jq  
'regions[] | "\(.id) - \(.city), \(.country)''
```

- List all GCP regions and zones with the gcloud CLI to determine where to deploy your VPCs and instances.

CONSOLE

```
$ gcloud compute regions list --format='value(name)'  
$ gcloud compute zones list --format='value(name)'
```

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-west1" = {
    "name" = "gcp-client-us-west"
```

```
"private_ip" = "10.40.0.2"
"provider" = "gcp"
"public_ip" = "192.8.0.45"
"subnet" = "10.40.0.0/24"
}
}
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-west1" = {
    "name" = "gcp-peer-us-west"
    "private_ip" = "10.40.0.3"
    "provider" = "gcp"
    "public_ip" = "192.3.0.7"
    "subnet" = "10.40.0.0/24"
  }
}
}
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 GCP), 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.

Run Ansible Playbooks

1. Navigate to the Ansible directory:

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 GCP gateway and client nodes.

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

CONSOLE

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

This playbook waits for SSH and cloud-init, creates Headscale users and preauth keys for each gateway, enrolls the gateways into Headscale while advertising their VPC subnets, approves routes, enables IP forwarding, and updates the main routing table without affecting existing VPC routes.

Note

To manually add more clients to the mesh network:

- **Vultr:** During instance provisioning, select the `vultr-x-gcp-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.
- **GCP:** When provisioning, select the respective VPC. No additional installation is required because the **GCP VPC routes** already contains the necessary routes.

Test and Benchmark the Private Network Mesh

Once deployment is complete, confirm that all Vultr and GCP regions are connected through the **WireGuard mesh** controlled by **Headscale**. Run the included Ansible playbooks to test both private and public routes, and capture metrics such as latency, throughput, and overall mesh stability.

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

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	438.12	512.47	168.92
vultr-peer-blr	gcp-peer-us-west	85.73	106.41	221.34

- 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	254.38	392.17	155.61
vultr-peer-blr	gcp-peer-us-west	31.22	74.05	217.98

📌 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

This guide demonstrated how to establish a secure, multi-region private mesh network between Vultr and GCP using Headscale and Tailscale. By leveraging Terraform and Ansible, you automated provisioning and configuration to ensure

consistent deployments across regions. Gateway and client nodes exchanged traffic over encrypted, low-latency tunnels, while FRRouting and GCP custom routes enabled seamless cross-datacenter connectivity.



VULTR

