Introduction

This repository contains the final class project for YCIT 018 - Cloud Networking & Security at McGill University. The goal of the project is to set up a simple cloud infrastructure with two VPC, four VMs, a VPN, along with a number of firewall rules. To get the 10% extra points, I took on the challenge of learning Terraform at the same time as I learned GCP. It has been a fun and challenging ride!

If you are reading this from a McGill pdf upload, the repository and all Terraform code is found on github.com/sohnya/cloud-project.

Goal Architecture

The following figure (taken from the project description here) outlines the expected results of the cloud infrastructure for this lab project.

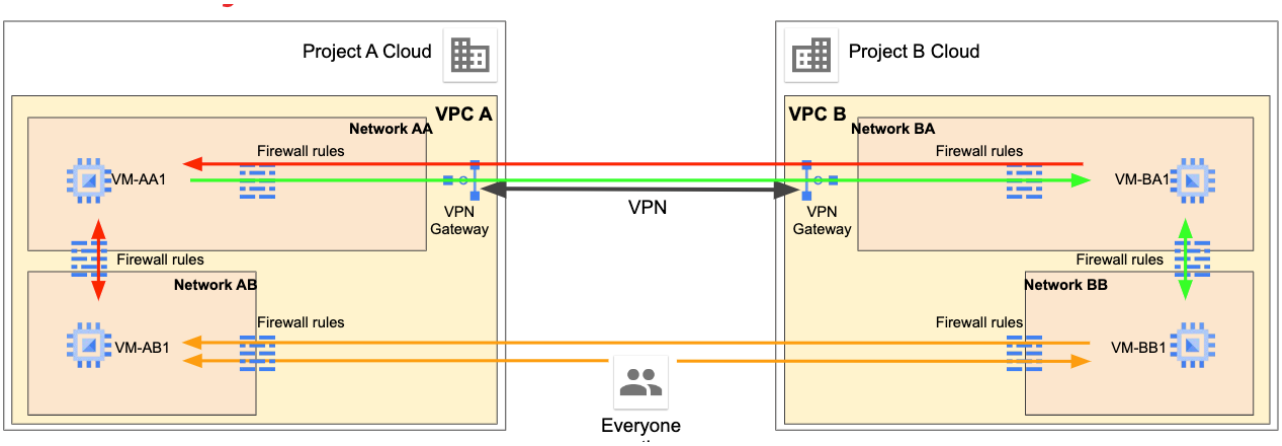


Figure: Firewalls and VPN

Google Cloud and Terraform

To begin with, I set up all infrastructure manually using the Google Cloud Console. After setting up project A, its VPN tunnel and an example firewall rule, I realized that

- Since project B was almost exactly the same (with minor differences), it would be super error prone and boring to continue
- Especially firewall rules were error prone to set up, and fiddly to change in the UI. This is why I decided to combine this project with another tool that I was interested in learning - Terraform.

The advantages of setting up the infrastructure with Terraform

- All configuration is in one place (easy to find especially for beginner)
- Mistakes are easy to find and fix
- Repetitive tasks become less error prone and boring

I set up Terraform to connect to a Google Cloud account using a GCP service account key (file saved locally) that is then called in `main.tf`. For more details, see here.

Project structure

The desired project configuration had (almost) the same configuration on both sides, which is why a module `project` was created. The `project` module itself contains custom Terraform modules specific to our use case - `vm`, `webserver-vm` and `vpn`. In order to clean up the `main.tf` file, I also chose to move the firewall rules to their own modules. Since they were different between A and B, they were implemented with `firewall-rules-a` and `firewall-rules-b`. These two modules contained the firewall rules and their corresponding network connectivity tests.

The `main.tf` file looks like follows (the full file is here):

```
module "project-a" {
  source = "./modules/project"
  project_id="org-a-309016"
  project_name="a"
  google_credentials_file = "/Users/sonjahiltunen/Secrets/gcloud/org-a-
961b663ea9dd.json"
  region = var.region
  zone = var.zone

  # Network
  subnet_a_ip_range = var.subnet_aa_ip_range
  subnet_b_ip_range = var.subnet_ab_ip_range

  ## VPN
  local_static_ip_address = var.a_static_ip
  remote_static_ip_address = var.b_static_ip
  vpn_destination_range = var.subnet_ba_ip_range
  vpn_shared_secret = var.vpn_shared_secret
}

module "firewall-rules-a" {
  source = "./modules/firewall-rules-a"
  project_id="org-a-309016"
  google_credentials_file = "/Users/sonjahiltunen/Secrets/gcloud/org-a-
961b663ea9dd.json"
  region = var.region
  zone = var.zone
  network = "vpc-a"

  subnet_aa_ip_range = var.subnet_aa_ip_range
  subnet_ab_ip_range = var.subnet_ab_ip_range
  subnet_ba_ip_range = var.subnet_ba_ip_range
  subnet_bb_ip_range = var.subnet_bb_ip_range
  vm_ab_ip_address = var.vm_ab_ip_address
  vm_bb_ip_address = var.vm_bb_ip_address
}
```

In order to reduce copy pasting, a main `variables.tf` file contains some of the static values of the project.

The `project` module contains the VPC, VM and VPN modules, the details of which will be discussed later. Note the `depends_on` arguments that are required for Terraform to build the infrastructure in the correct order.

```
provider "google" {
  project     = var.project_id
```

```
  region      = var.region
  zone        = var.zone
  credentials = file(var.google_credentials_file)
}

module "vpc" {
  ...
}

module "vm" {
  source      = "../vm"
  name        = "vm-${var.project_name}a"
  subnet_name = "network-${var.project_name}a"
  depends_on  = [module.vpc]
}

module "webserver_vm" {
  source      = "../webserver_vm"
  name        = "vm-${var.project_name}b"
  subnet_name = "network-${var.project_name}b"
  depends_on  = [module.vpc]
}

module "vpn" {
  source = "../vpn"
  project_name = var.project_name
  local_static_ip_address = var.local_static_ip_address
  remote_static_ip_address = var.remote_static_ip_address
  vpn_shared_secret = var.vpn_shared_secret
  vpn_destination_range = var.vpn_destination_range
  depends_on = [module.vm]
}
```

Projects

Requirement 1.1: The first lab requirement was to create two projects to represent sides A and B of the architecture diagram. The two projects were created in the Google Cloud UI.

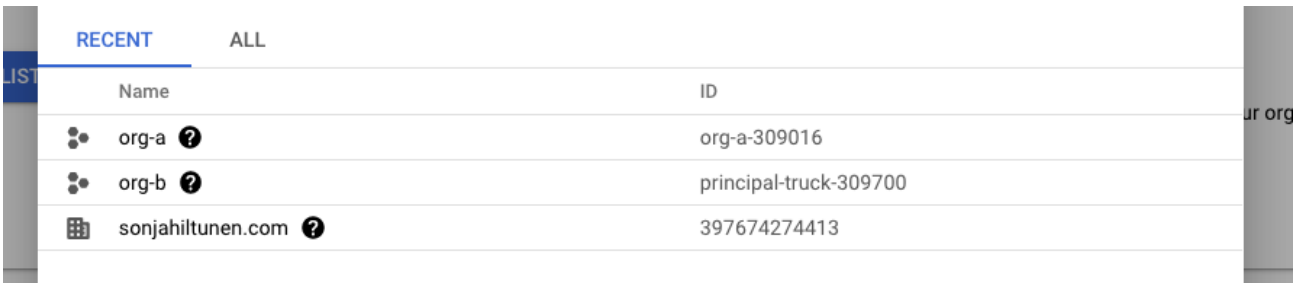


Figure: Two projects in my organization

IAM

In order to add users, I created an organization related to the sonjahiltunen.com domain, and added four new users

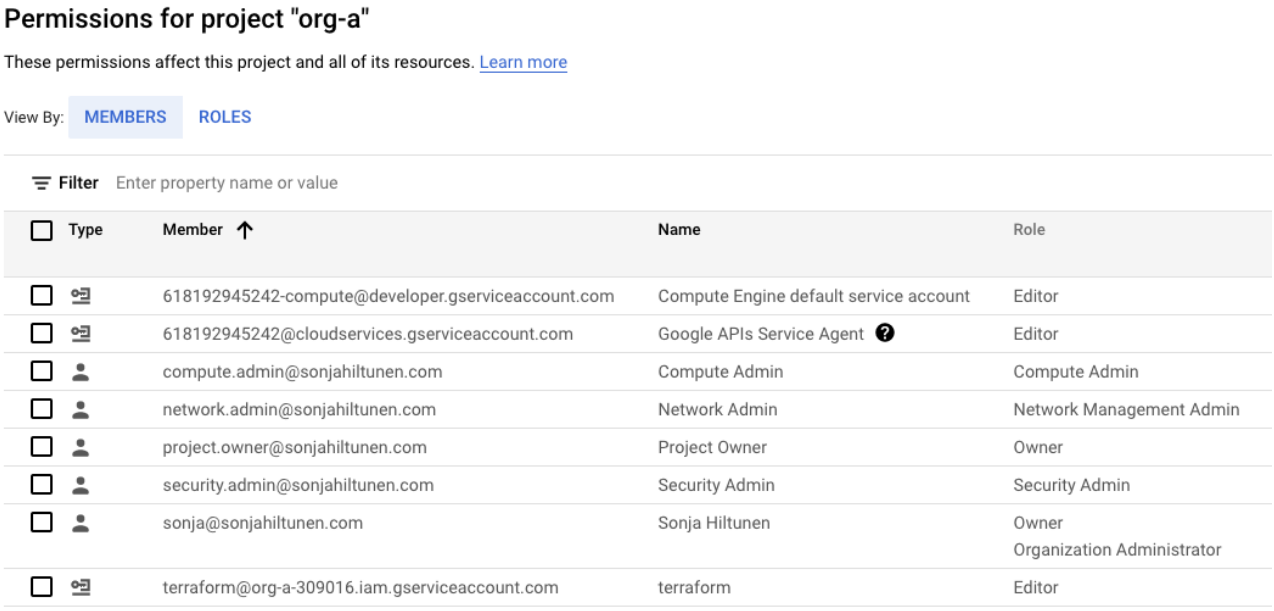
- project.owner@sonjahiltunen.com
- compute.admin@sonjahiltunen.com
- security.admin@sonjahiltunen.com
- network.admin@sonjahiltunen.com

The users were then added to the projects and given roles according to what they should be able to do with the resources in the projects. These are the roles, users and required permissions (taken from role definitions in predefined roles):

- Project Owner `roles/owner` project.owner@sonjahiltunen.com
  - All editor permissions and permissions to:
  - Manage roles and permissions for a project and all resources within the project.
  - Set up billing for a project.
- Compute Admin `roles/compute.admin` compute.admin@sonjahiltunen.com
  - Full control of all Compute Engine resources.
- Security Admin `roles/iam.securityAdmin` security.admin@sonjahiltunen.com
  - Security admin role, with permissions to get and set any IAM policy.
- Network Management Admin `roles/networkmanagement.admin` network.admin@sonjahiltunen.com
  - Full access to Network Management resources.

The main account (sonja@sonjahiltunen.com) is kept intact (with maximum permissions), as per the lab requirements.

Requirement 1.2 The four required roles can be seen in the screenshot below.



Permissions for project "org-a"

These permissions affect this project and all of its resources. Learn more

View By: MEMBERS ROLES

Filter Enter property name or value

Type	Member	Name	Role
	618192945242-compute@developer.gserviceaccount.com	Compute Engine default service account	Editor
	618192945242@cloudservices.gserviceaccount.com	Google APIs Service Agent	Editor
	compute.admin@sonjahiltunen.com	Compute Admin	Compute Admin
	network.admin@sonjahiltunen.com	Network Admin	Network Management Admin
	project.owner@sonjahiltunen.com	Project Owner	Owner
	security.admin@sonjahiltunen.com	Security Admin	Security Admin
	sonja@sonjahiltunen.com	Sonja Hiltunen	Owner Organization Administrator
	terraform@org-a-309016.iam.gserviceaccount.com	terraform	Editor

A similar setup has been done for project B.

Notes

- Note the service account for Terraform. This was added in the API credentials section in the cloud console (following the guidelines here).
- If I had more time, I would have looked into reducing the permissions so that they are specific to our use case. The required roles were very wide and do not follow the least privilege principle. We want to use IAM securely.
- I ran `terraform apply` as an owner. The Terraform project structure could have been optimized so that different teams (with different roles / permissions) can easily use Terraform separate. This is for an advanced use case with Terraform that I will save for later.

Networks and VMs

Network setup

The VPC and its subnets were created using the Terraform module vpc. The configuration is as follows:

```
module "vpc" {
  source       = "terraform-google-modules/network/google"
  version      = "~> 3.0"
```

```
  project_id   = var.project_id
  network_name = "vpc-${var.project_name}"
  routing_mode = "GLOBAL"

  subnets = [
    {
      subnet_name           = "network-${var.project_name}a",
      subnet_ip             = var.subnet_a_ip_range,
      subnet_region         = "us-east1",
      subnet_private_access = "true"
    },
    {
      subnet_name           = "network-${var.project_name}b",
      subnet_ip             = var.subnet_b_ip_range,
      subnet_region         = "us-east1",
      subnet_private_access = "true"
    }
  ]
```

and

VM setup

The VMs that didn't need external IPs were created using a module: `vm`, configured as follows:

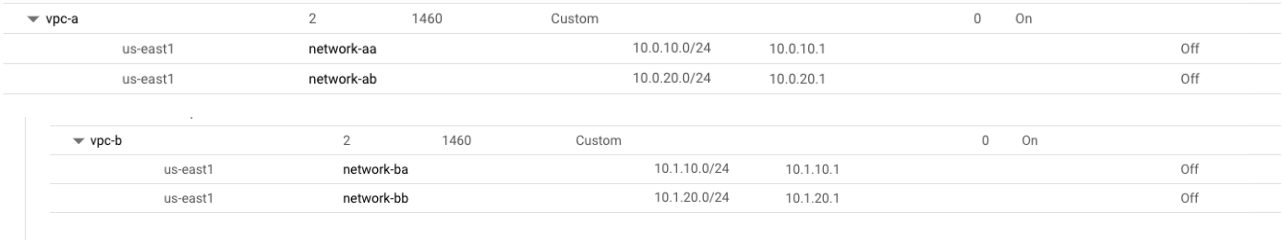
```
resource "google_compute_instance" "vm" {
  name         = var.name
  machine_type = "f1-micro"
  tags         = [var.name]

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
    }
  }

  network_interface {
    subnetwork = var.subnet_name
  }
}
```

For the webserver, I created a module `webserver_vm`. It is similar to a VM, but the network interface contains an empty `access_config`, which produces an ephemeral external IP for that VM.

```
  network_interface {
    subnetwork = "network-ab"
    access_config {
      // Gives the VM an external ephemeral IP address
    }
  }
```

It also contains a reference to the webserver startup script - more about this later.

Requirement 3.1 - Create 4 networks



VPN

Requirement 4.1

The VMs `vm-aa` and `vm-ba` only have private IP addresses. They are not directly accessible from the internet. The communicate together using a router with static routes and a VPN gateway.

VPN Gateway

The VPN gateway is created using the module compute_vpn_gateway, as follows:

```
resource "google_compute_vpn_gateway" "gateway_a" {
  name    = "vpn-a"
  network = "vpc-a"
  depends_on = [
    module.vpc
  ]
}
```

Note: The Google Cloud UI automatically creates the necessary forwarding rules when we select a classic VPN in the UI. This is not the case for the Terraform module - the forwarding rules have to be explicitly created as follows:

```
resource "google_compute_forwarding_rule" "fr_esp" {
  name        = "forwarding-rule-esp"
  ip_protocol = "ESP"
  ip_address  = var.local_static_ip_address
  target      = google_compute_vpn_gateway.gateway.id
}

resource "google_compute_forwarding_rule" "fr_udp500" {
  name        = "forwarding-rule-udp500"
  ip_protocol = "UDP"
  port_range  = "500"
  ip_address  = var.local_static_ip_address
  target      = google_compute_vpn_gateway.gateway.id
}

resource "google_compute_forwarding_rule" "fr_udp4500" {
  name        = "forwarding-rule-udp4500"
  ip_protocol = "UDP"
  port_range  = "4500"
  ip_address  = var.local_static_ip_address
  target      = google_compute_vpn_gateway.gateway.id
}
```

VPN Tunnel

A potentially confusing difference between the Google Cloud Console and the Terraform module `compute_vpn_tunnel` is that the module does not contain an explicit choice of routing configuration. The routing configuration is implicitly defined with the tunnel parameters.

As described here: "When you use the Cloud Console to create a route-based tunnel, Classic VPN [...]:

- Sets the tunnel's local and remote traffic selectors to any IP address (0.0.0.0/0).
- For each range in Remote network IP ranges, creates a custom static route whose destination (prefix) is the range's CIDR and whose next hop is the tunnel."

The compute_vpn_tunnel was therefore set up as follows:

```
resource "google_compute_vpn_tunnel" "tunnel" {
  name               = "tunnel-${var.project_name}"
  peer_ip            = var.remote_static_ip_address
  shared_secret      = var.vpn_shared_secret
  target_vpn_gateway = google_compute_vpn_gateway.gateway.id

  local_traffic_selector  = ["0.0.0.0/0"]
```

```
  remote_traffic_selector = ["0.0.0.0/0"]

  depends_on = [
    google_compute_forwarding_rule.fr_esp,
    google_compute_forwarding_rule.fr_udp500,
    google_compute_forwarding_rule.fr_udp4500,
  ]
}
```

and the static route added as

```
resource "google_compute_route" "route_a" {
  name                = "route-${var.project_name}"
  network             = "vpc-${var.project_name}"
  dest_range          = var.vpn_destination_range
  priority            = 1000
  next_hop_vpn_tunnel = google_compute_vpn_tunnel.tunnel.id
}
```

The static route definition show up in the Google Cloud Console as

Name	Description	Destination IP range	Priority	Instance tags	Next hop	Network
route-a		10.1.10.0/24	1000	None	VPN tunnel tunnel-a	vpc-a
route-b		10.0.10.0/24	1000	None	VPN tunnel tunnel-b	vpc-b

The following screenshots show the working configuration of the VPN tunnel:

Tunnel name	Cloud VPN gateway (IP)	Peer VPN gateway (IP)	Cloud Router BGP IP	BGP Peer IP	Routing type	VPN tunnel status	Bgp session status	Google network	Region	Description	Labels
tunnel-a (Classic)	vpn-a 35.237.20.53	35.237.166.103	none	none	Route-based	Established	–	vpc-a	us-east1		

VPN tunnel in project A

Tunnel name	Cloud VPN gateway (IP)	Peer VPN gateway (IP)	Cloud Router BGP IP	BGP Peer IP	Routing type	VPN tunnel status	Bgp session status	Google network	Region	Description	Labels
tunnel-b (Classic)	vpn-b 35.237.166.103	35.237.20.53	none	none	Route-based	Established	–	vpc-b	us-east1		

VPN tunnel in project B

Firewall rules and connectivity tests

The firewall rules were implemented in Terraform using the resource compute_firewall. Example configuration:

```
# VM-AA1 CANNOT ping VM-AB1 using Firewall rules
resource "google_compute_firewall" "requirement_4_2_1a" {
  name    = "r4-2-1a-aa-cannot-ping-ab"
  network = var.network
```

```
    deny {
      protocol = "icmp"
    }

    source_tags = ["vm-aa"]
    target_tags = ["vm-ab"]
  }
```

The full list of firewall rules can be found in the modules firewall-rules-a and firewall-rules-b.

The resulting rules in the UI are as follows (in projects A and B):

☐	r4-1-4-ba-cannot-ping-aa	Ingress	vm-aa	IP ranges: 10.1.10.0/24	icmp	Deny	1000	vpc-a
☐	r4-2-1a-aa-cannot-ping-ab	Ingress	vm-ab	Tags: vm-aa	icmp	Deny	1000	vpc-a
☐	r4-2-1b-ab-cannot-ping-aa	Ingress	vm-aa	Tags: vm-ab	icmp	Deny	1000	vpc-a
☐	r4-4-4-internet-cannot-ssh-ab-public	Ingress	vm-ab	IP ranges: 0.0.0.0/0	tcp:22	Deny	1000	vpc-a
☐	r4-4-1-internet-can-http-80-ab-public	Ingress	vm-ab	IP ranges: 0.0.0.0/0	tcp:80	Allow	1000	vpc-a
☐	r4-4-3-internet-can-ping-ab-public	Ingress	vm-ab	IP ranges: 0.0.0.0/0	icmp	Allow	1000	vpc-a

Figure: Firewall rules in org-a

☐	r4-5-1-bb-cannot-http-80-ab-public	Egress	vm-bb	IP ranges: 35.231.62.201	tcp:80	Deny	1000	vpc-b
☐	r4-5-4-bb-cannot-ping-8-8-8-8	Egress	vm-bb	IP ranges: 8.8.8.8	icmp	Deny	1000	vpc-b
☐	r4-4-2-internet-cannot-http-80-bb-public	Ingress	vm-bb	IP ranges: 0.0.0.0/0	tcp:80	Deny	1000	vpc-b
☐	r4-4-4-internet-cannot-ssh-bb-public	Ingress	vm-bb	IP ranges: 0.0.0.0/0	tcp:22	Deny	1000	vpc-b
☐	r4-1-3-aa-can-ping-ba	Ingress	vm-ba	IP ranges: 10.0.10.0/24	icmp	Allow	1000	vpc-b
☐	r4-3-1a-ba-can-ping-bb	Ingress	vm-bb	Tags: vm-ba	icmp	Allow	1000	vpc-b
☐	r4-3-1b-bb-can-ping-ba	Ingress	vm-ba	Tags: vm-bb	icmp	Allow	1000	vpc-b
☐	r4-4-3-internet-can-ping-bb-public	Ingress	vm-bb	IP ranges: 0.0.0.0/0	icmp	Allow	1000	vpc-b

Figure: Firewall rules in org-b

To test the the configuration works as expected, I set up connectivity tests from the Network Connectivity tool in GCP, and it's corresponding Terraform module (network_management_connectivity_test).

Below are screenshots of the connectivity test results.

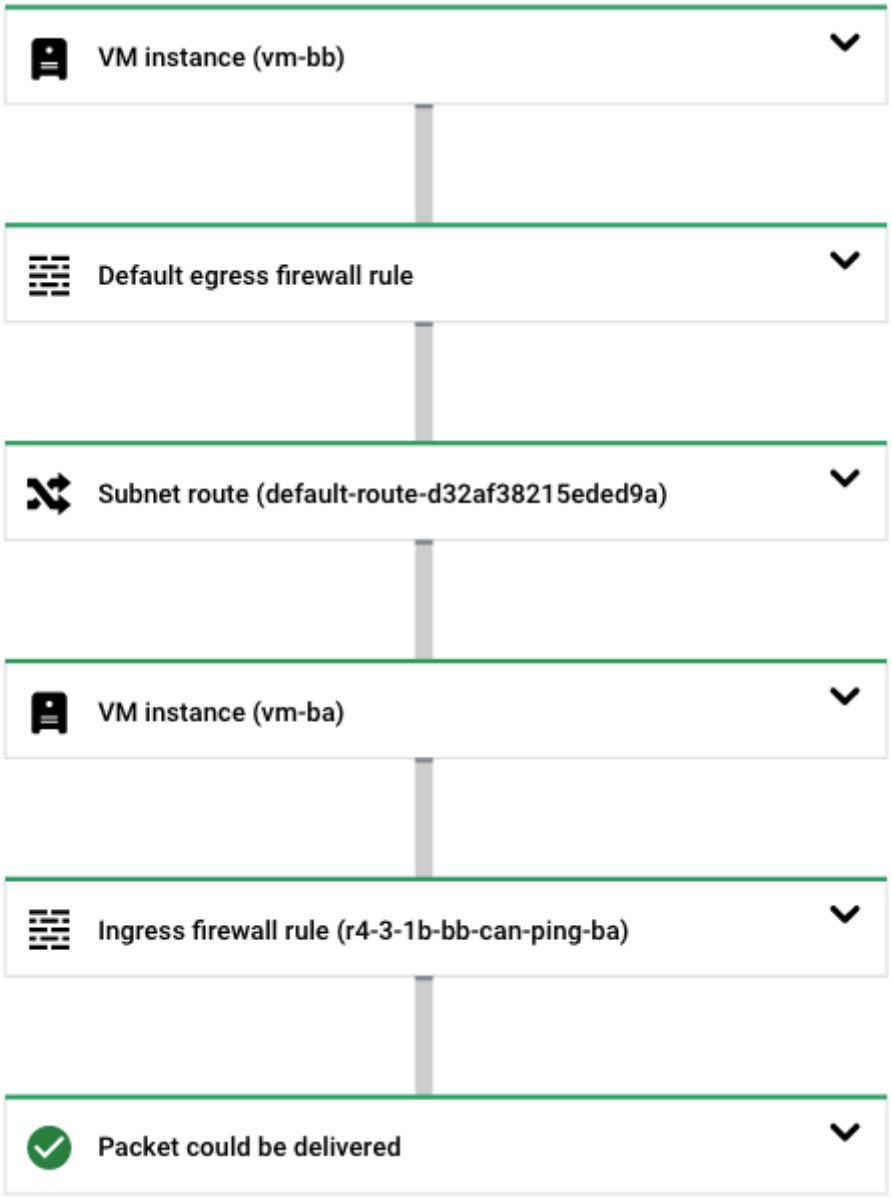
	Name ↑	Protocol	Source	Destination	Destination port	Last test time	Last configuration analysis result
☐	r4-1-3-aa-can-ping-ba	icmp	vm-aa	vm-ba	-	2021-04-05 (14:07:58)	Reachable
☐	r4-2-1a-aa-cannot-ping-ab	icmp	vm-aa	vm-ab	-	2021-04-05 (14:07:58)	Unreachable
☐	r4-2-1b-ab-cannot-ping-aa	icmp	vm-ab	vm-aa	-	2021-04-05 (14:07:57)	Unreachable
☐	r4-4-1-internet-can-http-80-ab-public	tcp	70.83.57.252	35.231.62.201	80	2021-04-05 (14:07:58)	Reachable
☐	r4-4-3-internet-can-ping-ab-public	icmp	70.83.57.252	35.231.62.201	-	2021-04-05 (14:07:57)	Reachable
☐	r4-4-4-internet-cannot-ssh-ab-public	tcp	70.83.57.252	35.231.62.201	22	2021-04-05 (14:07:57)	Unreachable

Figure: Connectivity tests in org-a

Name ↑	Protocol	Source	Destination	Destination port	Last test time	Last configuration analysis result
r4-1-4-ba-CANNOT-ping-aa	icmp	vm-ba	vm-aa	-	2021-04-05 (14:08:47)	Unreachable
r4-3-1a-ba-can-ping-bb	icmp	vm-ba	vm-bb	-	2021-04-05 (14:08:47)	Reachable
r4-3-1b-bb-can-ping-ba	icmp	vm-bb	vm-ba	-	2021-04-05 (14:08:47)	Reachable
r4-4-2-internet-cannot-http-80-bb-public	tcp	70.83.57.252	35.231.103.20	80	2021-04-05 (14:08:47)	Unreachable
r4-4-3-internet-can-ping-bb-public	icmp	70.83.57.252	35.231.103.20	-	2021-04-05 (14:08:47)	Reachable
r4-4-4-internet-cannot-ssh-bb-public	tcp	70.83.57.252	35.231.103.20	22	2021-04-05 (14:08:46)	Unreachable
r4-5-1-bb-cannot-http-80-ab-public	tcp	vm-bb	35.231.62.201	80	2021-04-05 (14:08:47)	Unreachable
r4-5-2-bb-can-ping-ab-public	icmp	vm-bb	35.231.62.201	-	2021-04-05 (14:08:47)	Reachable
r4-5-3-bb-can-ssh-ab-public	tcp	vm-bb	35.231.62.201	22	2021-04-05 (14:08:47)	Reachable
r4-5-4-bb-cannot-ping-8-8-8-8	icmp	vm-bb	8.8.8.8	-	2021-04-05 (14:08:47)	Unreachable

Figure: Connectivity tests in org-b

Each connectivity test contains more information about the route taken, which firewall rule was used etc. Below is an example for `bb-can-ping-ba`



Notes

- The firewall rules and connectivity tests are hard coded in the modules `firewall-rules-a` and `firewall-rules-b`, respectively. I couldn't figure out a nicer way to setup the firewall rules in Terraform, since they are very specific to the project configuration for each side.

Web Server

```
apt update
apt install -y apache2
cat <<EOF > /var/www/html/index.html
```

```
  <html>
    ...
  </html>
  EOF
```

To add a startup script to the VMs, we add the argument `metadata_startup_script` in our Terraform configuration.

```
resource "google_compute_instance" "vm-ab1" {
  ...

  metadata_startup_script = file("./modules/webserver_vm/startup.sh")

  ...
}
```

The result of the startup script is a fun

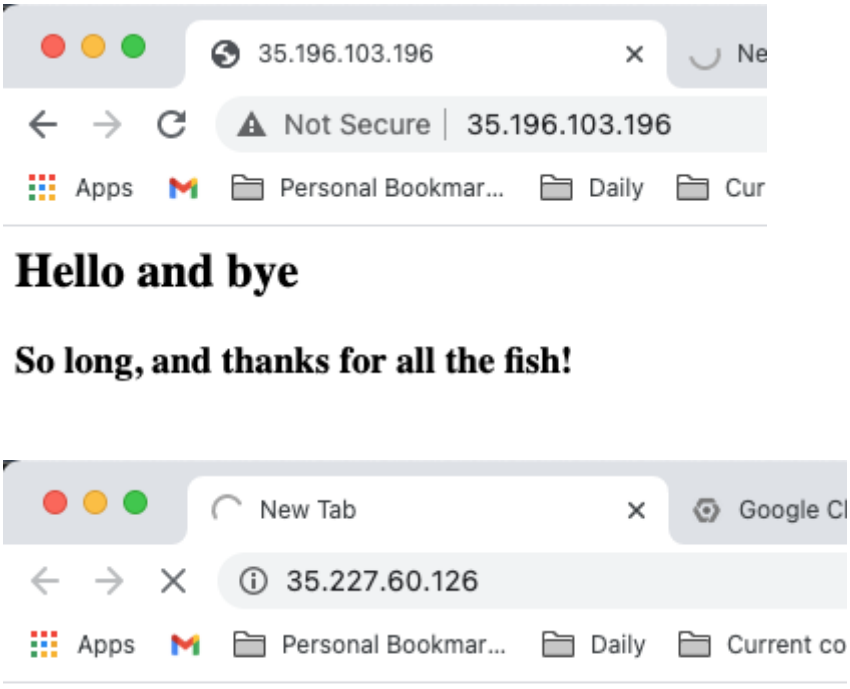


Figure: Web server AB: Open to the internet | Web server BB: Blocked with firewall rule

Note that since `vm-bb` blocks incoming TCP traffic on port 80, the fun stuff cannot be seen from the internet.