Here is the transcript of the content displayed in the video.

Configuration Configuration files describe to Terraform the components needed to run a single application or your entire data center. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform can determine what changed and create incremental execution plans that can be applied.

The infrastructure Terraform can manage includes both low-level components such as compute instances, storage, and networking, and high-level components such as DNS entries and SaaS features.

Key features

Infrastructure as code Infrastructure is described using a high-level configuration syntax. This allows a blueprint of your data center to be versioned and treated as you would any other code. Additionally, infrastructure can be shared and re-used.

Execution plans Terraform has a planning step in which it generates an execution plan. The execution plan shows what Terraform will do when you execute the apply command. This lets you avoid any surprises when Terraform manipulates infrastructure.

Resource graph Terraform builds a graph of all your resources and parallelizes the creation and modification of any non-dependent resources. Because of this, Terraform builds infrastructure as efficiently as possible, and operators get insight into dependencies in their infrastructure.

Change automation Complex changesets can be applied to your infrastructure with minimal human interaction. With the previously mentioned execution plan and resource graph, you know exactly what Terraform will change and in what order, which helps you avoid many possible human errors.

Task 1. Verify a Terraform installation Terraform comes pre-installed in Cloud Shell.

- Open a new Cloud Shell tab, and run the following command to verify that Terraform is available:

```
terraform
```

The resulting help output should be similar to this:

```
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init          Prepare your working directory for other commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure
```

```
All other commands:
  console       Try Terraform expressions at an interactive command
  prompt
  fmt           Reformat your configuration in the standard style
  force-unlock  Release a stuck lock on the current workspace
  get           Install or upgrade remote Terraform modules
  graph         Generate a Graphviz graph of the steps in an operation
  import        Associate existing infrastructure with a Terraform
  resource
  login         Obtain and save credentials for a remote host
  logout        Remove locally-stored credentials for a remote host
  metadata      Metadata related commands
  output        Show output values from your root module
  providers     Show the providers required for this configuration
  refresh       Update the state to match remote systems
  show          Show the current state or a saved plan
  state         Advanced state management
  taint         Mark a resource instance as not fully functional
  test          Experimental support for module integration testing
  version       Show the current Terraform version
  workspace     Workspace management

Global options (use these before the subcommand, if any):
  -chdir=DIR    Switch to a different working directory before executing
                the
                given subcommand.
  -help         Show this help output, or the help for a specified
                subcommand.
  -version      An alias for the "version" subcommand.
```

Task 2. Build the infrastructure With Terraform installed, you can immediately start creating some infrastructure. First let's enable Gemini Code Assist, which will help you in creating your Terraform code.

Enable Gemini Code Assist in the Cloud Shell IDE You can use Gemini Code Assist in an integrated development environment (IDE) such as Cloud Shell to receive guidance on code or solve problems with your code. Before you can start using Gemini Code Assist, you need to enable it.

1. In Cloud Shell, enable the **Gemini for Google Cloud** API with the following command:

```
gcloud services enable cloudaicompanion.googleapis.com
```

2. Click **Open Editor** on the Cloud Shell toolbar.

   Note: To open the Cloud Shell editor, click **Open Editor** on the Cloud Shell toolbar. You can switch between Cloud Shell and the code editor by clicking **Open Editor** or **Open Terminal**, as required.

3. In the left pane, click the **Settings** icon, then in the **Settings** view, search for **Gemini Code Assist**.

4. Locate and ensure that the checkbox is selected for **Geminicodeassist: Enable**, and close the **Settings**.

5. Click **Cloud Code - No Project** in the status bar at the bottom of the screen.
6. Authorize the plugin as instructed. If a project is not automatically selected, click **Select a Google Cloud Project**, and choose `qwiklabs-gcp-01-c77a69512747`.
7. Verify that your Google Cloud project (`qwiklabs-gcp-01-c77a69512747`) displays in the Cloud Code status message in the status bar.

Configuration The set of files used to describe infrastructure in Terraform is simply known as a Terraform configuration. In this section, you will write your first configuration to launch a single VM instance. The format of the configuration files can be found in the Terraform Language Documentation. We recommend using JSON for creating configuration files.

1. In Cloud Shell, create an empty configuration file named `instance.tf` with the following command:

```
touch instance.tf
```

2. Click **Open Editor** on the Cloud Shell toolbar.
3. From the file explorer, double-click **instance.tf** to open the `instance.tf` file. This action enables Gemini Code Assist, as indicated by the presence of the spark icon in the upper-right corner of the editor.
4. To generate new code, click the **Gemini Code Assist: Smart Actions** spark icon and select **Generate**. Alternatively, you can press `CTRL+I` (for Windows and Linux) or `CMD+I` (for macOS) to open the **Gemini Code Assist** window, and select **Generate** to generate new code.

Use Gemini Code Assist to generate a Terraform resource In this section, you explore how to use Gemini Code Assist to generate a Terraform resource, such as a VM in this instance.

1. Paste the following provided prompt into the Gemini Code Assist inline text box.

```
Generate the Terraform configuration for a Google Compute Engine virtual machine,
saving it to instance.tf, based on the following specifications:
* Project ID: qwiklabs-gcp-01-c77a69512747
* VM Name: terraform
* Machine Type: e2-medium
* Deployment Zone: us-central1-a
* Boot Disk: Debian 12
* Network: Default network
```

The code in the inline text box should resemble the following code block.

2. Press **Enter** to generate code used to launch a VM instance using Terraform. When prompted in the **Gemini Diff** view, click **Accept**.
3. In the `instance.tf` file, view the generated code.

This is a complete configuration that Terraform is ready to apply. The general structure should be intuitive and straightforward.

Once you accept Gemini Code Assist's recommendation, the resulting `instance.tf` file should resemble the following.

Generated file contents:

```
resource "google_compute_instance" "default" {
  project      = "qwiklabs-gcp-01-c77a69512747"
  zone         = "us-central1-a"
  name         = "terraform"
  machine_type = "e2-medium"
  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-12"
    }
  }
  network_interface {
    network = "default"
  }
}
```

The "resource" block in the `instance.tf` file defines a resource that exists within the infrastructure. A resource might be a physical component such as a VM instance.

The resource block has two strings before opening the block: the **resource type** and the **resource name**. For this lab, the resource type is `google_compute_instance` and the name is `terraform`. The prefix of the type maps to the provider: `google_compute_instance`, which automatically tells Terraform that it is managed by the Google provider.

Within the resource block itself is the configuration needed for the resource.

4. In the Cloud Shell Terminal, run the following code to verify that your new file has been added and that there are no other `*.tf` files in your directory, as Terraform loads all `.tf` files:

```
ls
```

Initialization The first command to run for a new configuration—or after checking out an existing configuration from version control—is `terraform init`. This will initialize various local settings and data that will be used by subsequent commands.

Terraform uses a plugin-based architecture to support the numerous infrastructure and service providers available. Each "provider" is its own encapsulated binary that is distributed separately from Terraform itself. The `terraform init` command will automatically download and install any provider binary for the providers to use within the configuration, which in this case is just the Google provider.

1. Download and install the provider binary:

```
terraform init
```

The Google provider plugin is downloaded and installed in a subdirectory of the current working directory, along with various other book keeping files. You will see an "Initializing provider plugins" message. Terraform knows that you're running from a Google project, and it is getting Google resources.

```
Installing hashicorp/google v6.47.0...
```

Note: Your version number may be higher.

The output specifies which version of the plugin is being installed and suggests that you specify this version in future configuration files to ensure that `terraform init` will install a compatible version.

2. Create an execution plan:

```
terraform plan
```

Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files. This command is a convenient way to check whether the execution plan for a set of changes matches your expectations without making any changes to real resources or to the state. For example, you might run this command before committing a change to version control, to create confidence that it will behave as expected.

Note: The optional `-out` argument can be used to save the generated plan to a file for later execution with `terraform apply`.

Apply changes

1. In the same directory as the `instance.tf` file you created, run this command:

```
terraform apply
```

The resulting output shows the Execution Plan, which describes the actions Terraform will take in order to change real infrastructure to match the configuration. The output format is similar to the diff format generated by tools like Git.

There is a `+` next to `google_compute_instance.terraform`, which means that Terraform will create this resource. Following that are the attributes that will be set. When the value displayed is `<computed>`, it means that the value won't be known until the resource is created.

Example output:

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
```

```
Terraform will perform the following actions:

  + resource "google_compute_instance" "default" {
      + can_ip_forward       = false
      + cpu_platform         = (known after apply)
      + deletion_protection  = false
      + guest_accelerator    = (known after apply)
      + id                   = (known after apply)
      + instance_id          = (known after apply)
      + label_fingerprint    = (known after apply)
      + machine_type         = "e2-medium"
      + metadata_fingerprint = (known after apply)
      + name                 = "terraform"
      + project              = "qwiklabs-gcp-01-c77a69512747"
      + self_link            = (known after apply)
      + tags_fingerprint     = (known after apply)
      + zone                 = "us-central1-a"

      + boot_disk {
          + auto_delete                = true
          + device_name                = (known after apply)
          + disk_encryption_key_sha256 = (known after apply)
          + kms_key_self_link          = (known after apply)
          + source                     = (known after apply)

          + initialize_params {
              + image  = "debian-cloud/debian-12"
              + labels = (known after apply)
              + size   = (known after apply)
              + type   = (known after apply)
            }
        }

      + network_interface {
          + address            = (known after apply)
          + name               = (known after apply)
          + network            = "default"
          + network_ip         = (known after apply)
          + subnetwork         = (known after apply)
          + subnetwork_project = (known after apply)

          + access_config {
              + assigned_nat_ip = (known after apply)
              + nat_ip          = (known after apply)
              + network_tier    = (known after apply)
            }
        }

      + scheduling {
          + automatic_restart   = (known after apply)
          + on_host_maintenance = (known after apply)
          + preemptible         = (known after apply)
```

```
          + node_affinities {
              + key      = (known after apply)
              + operator = (known after apply)
              + values   = (known after apply)
            }
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions? Terraform will perform the actions described
above. Only 'yes' will be accepted to approve.

Enter a value:
```

If the plan was created successfully, Terraform now pauses and waits for approval before proceeding. In a production environment, if anything in the Execution Plan seems incorrect or dangerous, it's safe to cancel at this point and no changes have yet been made to your infrastructure.

2. In this case the plan looks acceptable, so type **yes** at the confirmation prompt to proceed, and press **Enter**. Executing the plan takes a few minutes because Terraform waits for the VM instance to become available.

After this, Terraform is all done!

Test completed task Click **Check my progress** to verify your performed task. If you have completed the task successfully, you will receive an assessment score.

- Create a VM instance in the `us-central1-a` zone with Terraform.

3. In the Google Cloud console, on the **Navigation menu**, click **Compute Engine > VM Instances**. The **VM Instances** page opens and the VM instance you just created displays in the **VM Instances** list.

Terraform has written some data into the `terraform.tfstate` file. This state file is extremely important; it keeps track of the IDs of created resources so that Terraform knows what it is managing.

4. In Cloud Shell, run the following command to inspect the current state:

```
terraform show
```

Example output:

```
# google_compute_instance.default:
resource "google_compute_instance" "default" {
    can_ip_forward       = false
    cpu_platform         = "Intel Haswell"
    deletion_protection  = false
    guest_accelerator    = []
    id                   = "3488292216444307052"
```

```
    instance_id          = "3488292216444307052"
    label_fingerprint    = "42WmSpB8rSM="
    machine_type         = "e2-medium"
    metadata_fingerprint = "s6Is2zTjfmW="
    name                 = "terraform"
    project              = "qwiklabs-gcp-42390cc9da8a4c4b"
    self_link            =
"https://www.googleapis.com/compute/v1/projects/qwiklabs-gcp-
42390cc9da8a4c4b/zones/us-central1-a/instances/terraform"
    tags_fingerprint     = "42WmSpB8rSM="
    zone                 = "us-central1-a"

    boot_disk {
        auto_delete = true
        device_name = "persistent-disk-0"
        source      = "https://www.googleapis.com/compute/v1/projects/qwiklabs-
gcp-42390cc9da8a4c4b/zones/us-central1-a/disks/terraform"
        # ... (rest of output)
    }
    # ...
}
```

As you can tell, by creating this resource you've also gathered a lot of information about it. These values can be referenced to configure additional resources or outputs.

Congratulations! You've built your first infrastructure with Terraform. You've explored the configuration syntax, an example of a basic execution plan, and the state file.

Task 3. Test your understanding The following multiple-choice questions should reinforce your understanding of this lab's concepts. Answer them to the best of your abilities.

- Terraform enables you to safely and predictably create, change, and improve infrastructure.
    - ☐ True
    - ☐ False
- With Terraform, you can create your own custom provider plugins.
    - ☐ True
    - ☐ False

Congratulations! Congratulations on completing this lab! You've explored how to use Terraform to create and manage infrastructure on Google Cloud with Gemini Code Assist.

Next steps/Learn More

- Hashicorp on the Google Cloud Marketplace!
- Terraform Community
- Terraform Google Examples

Google Cloud training and certification ...helps you make the most of Google Cloud technologies. Our classes include technical skills and best practices to help you get up to speed quickly and continue your

learning.