Automation is the foremost requirement of any cloud-oriented activity. Being the chief tool for Infrastructure as Code (IaC), Terraform helps automate infrastructure management in a slick way. Yet provisioning infrastructure is not always enough; often, you need a way to customize or configure the servers post-deployment. That is the duty of a Terraform provisioner.

Provisioners in Terraform help execute scripts or commands on local or remote machines during resource creation or destruction. They are normally utilized to do things such as setting up of servers, executing shell scripts, or starting up any application on the deployment of infrastructure. While provisioners in Terraform can automate post-deployment actions, HashiCorp recommends using them sparingly, favoring configuration management tools like Ansible or cloud-init for ongoing setups. Provisioners are a final intervention when native Terraform providers or resources are unable to reach particular configurations.

Terraform provisioners allow you to execute actions such as copying or running a script with a local or remote machine once a resource has been created or when a resource is about to be destroyed. To invoke one, you simply state a `provisioner` block within a resource.

The main types are:

- file: Copies files or directories from the local machine to the newly created resource.
- local-exec: Executes a command on the host running the Terraform process.
- remote-exec: Executes a script on the remote resource after the resource is created.

Note: Provisioners need to be a last resort. When virtual machines are to be configured, there is practically no better option than setting up cloud-init scripts (through 'user_data'), as well as configuration management programs designed specifically to carry this out, i.e., dedicated configuration management tools, such as Ansible.

Examples of Terraform Provisioners:

File Provisioner:

```
provisioner "file" {
source      = "configs/sample.conf"
destination = "/home/azureuser/sample.conf"
connection {
```

...

}

}

- Copies a local file (configs/sample.conf) to the VM.
- Useful for pushing configuration files or scripts.

Local Provisioner (Pre-deployment):

```
resource "null_resource" "deployment_prep" {
triggers = { always_run = timestamp() }
provisioner "local-exec" {
command = "echo 'Deployment started at ${timestamp()}' > deployment-${timestamp()}.log"
}
}
```

- Runs locally on your machine before VM creation.
- Creates a log file with the current timestamp.
- `depends_on` ensures the VM waits until this step finishes.

Remote Provisioner:

```
provisioner "remote-exec" {
inline = [
"sudo apt-get update",
"sudo apt-get install -y nginx",
"echo '<html><body><h1>#28daysofAZTerraform is Awesome!</h1></body></html>' | sudo tee /var/www/html/index.html",
"sudo systemctl start nginx",
"sudo systemctl enable nginx"
]
connection {
...
}
}
```

- Runs on the VM itself via SSH.
- Installs Nginx, creates a custom welcome page, and ensures the service is running.
- Demonstrates how Terraform can configure software after provisioning.

Provisioners are powerful but come with limitations. They should only be used when native Terraform resources or configuration management tools cannot achieve the desired setup.

- Provisioners provide low-level access to OS and application configuration.
- Actions performed by provisioners are not tracked in Terraform state.
- Failures in provisioners can be hard to debug and recover from.
- Provisioners can introduce hidden dependencies and increase maintenance overhead.