**Terraform**

- Infrastructure management tool made by HashiCorp.

- lets us provision, manage and maintain cloud resources (servers, storage etc) and containers (dockers, kubernetes etc).

- It is a tool (and language) or command line program used to define and make changes to the deployed infrastructure. (Terraform cloud is the hosted version of terraform that connects to version control system, Git).

For ex :- create a server(EC2) instance and place it behind load balancer. It can't change what's running on that server when it is deployed. (For this, we need config management tools like ansible/chef/puppet to our rescue).

So, terraform lets you create server and then you can transform it into web server with ansible configuring specific application code to run on it.

**Provisioners** are used to execute scripts on a local or remote machine as part of resource creation or destruction. **Provisioners** can be used to bootstrap a resource, cleanup before destroy, run configuration management, etc.

Packer and Terraform belong to **"Infrastructure Build Tools"** category of the tech stack.

Developers describe **Packer** as "Create identical machine/base images for multiple platforms from a single source configuration". Packer images allow you to launch completely provisioned and configured machines i.e. it installs and configures all the software for a machine at the time the image is built.

 On the other hand, **Terraform** is detailed as "Describe your complete infrastructure as code and build resources across providers". With Terraform, you describe your complete infrastructure as code, even as it spans multiple service providers. Your servers may come from AWS, your DNS may come from CloudFlare, and your database may come from Heroku. Terraform will build all these resources across all these providers in parallel.

**"Cross platform builds"** is the primary reason why developers consider Packer over whereas **"Infrastructure as code"** was stated as the key factor in picking Terraform.

Terraform can handle infrastructure updates just fine. The key thing to understand, however, is that Terraform largely follows an **immutable infrastructure** paradigm, which means that to "update" a resource, you delete the old resource and create a new one to replace it. This is much like functional programming, where variables are immutable, and to "update" something, you actually create a new variable.

The typical pattern with Terraform is to use it to deploy a *server image*, such as an Virtual Machine (VM) Image (e.g. an [Amazon Machine Image (AMI)](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html)) or a Container Image (e.g. a [Docker Image](http://docker.com/)). When you want to "update" something, you create a new version of your image, deploy that onto a new server, and undeploy the old server.

Here's an example of how that works:

Imagine that you're building a Ruby on Rails app. You get the app working in dev and it's time to deploy to prod. The first step is to package the app as an AMI. You could do this using a tool like [Packer](https://www.packer.io/). Now you have an AMI with id ami-1234.

Here is a Terraform template you could use to deploy this AMI on a server (an [EC2 Instance](https://aws.amazon.com/ec2/)) in AWS with an [Elastic IP Address](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html) attached to it:

resource "aws\_instance" "example" {

ami = "ami-1234"

instance\_type = "t2.micro"

}

resource "aws\_eip" "example" {

instance = "${aws\_instance.example.id}"

}

When you run terraform apply, Terraform deploys the server, attaches an IP address to it, and now when users visit that IP, they will see v1 of your Rails app.

Some time later, you update your Rails app and want to deploy the new version, v2. To do that, you build a new AMI (i.e. you run Packer again) to get an ami with ID "ami-5678". You update your Terraform templates accordingly:

resource "aws\_instance" "example" {

ami = "ami-5678"

instance\_type = "t2.micro"

}

When you run terraform apply, Terraform undeploys the old server (which it can find because [Terraform records the state of your infrastructure](https://blog.gruntwork.io/how-to-manage-terraform-state-28f5697e68fa)), deploys a new server with the new AMI, and now users will see v2 of your code at that same IP.

Of course, there is one problem here: in between the time when Terraform undeploys v1 and when it deploys v2, your users would see downtime. To work around that, you could use Terraform's create\_before\_destroy lifecycle setting:

resource "aws\_instance" "example" {

ami = "ami-5678"

instance\_type = "t2.micro"

lifecycle {

create\_before\_destroy = true

}

}

With create\_before\_destroy set to true, Terraform will create the replacement server first, switch the IP to it, and then remove the old server. This allows you to do zero-downtime deployment with immutable infrastructure (note: zero-downtime deployment works better with a load balancer that can do health checks than a simple IP address, especially if your server takes a long time to boot).

For more information on this, check out the book [*Terraform: Up & Running*](http://www.terraformupandrunning.com/). The code samples for the book include an example of a zero-downtime deployment with a cluster of servers and a load balancer: <https://github.com/brikis98/terraform-up-and-running-code>

Terraform can also be used to provision a kubernetes cluster on AWS (or any cloud provider) and then use the terraform kubernetes provider to provision containers on that cluster.