




Terraform vs. Cloudify



Tasio Méndez Ayerbe
tasio.mendez@mail.polimi.it
Politecnico di Milano

Table of Contents

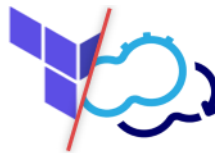
1. Introducing Terraform



2. Introducing Cloudify



3. Comparison



Introduction to Terraform



Terraform from HashiCorp



Terraform is the **infrastructure-as-code (IaC)** tool from HashiCorp. It is a tool for building, changing, and managing infrastructure in a safe, repeatable way. Operators and Infrastructure teams can use Terraform to manage environments with a configuration language called the HashiCorp Configuration Language (HCL) for human-readable, automated deployments.

Infrastructure-as-code is the process of managing infrastructure in a file or files rather than manually configuring resources in a user interface. A resource in this instance is any piece of infrastructure in a given environment, such as a virtual machine, security group, network interface, etc.

At a high level, Terraform allows operators to use HCL to author files containing definitions of their desired resources on almost any provider (AWS, GCP, GitHub, Docker, etc) and automates the creation of those resources at the time of apply.

Using HCL configuration files, Terraform can manage entire buildouts in a state file (*tfstate*). This state file allows Terraform to create, destroy, and modify components idempotently in a dependency understood methodology.

Installation & Login



Terraform can be used in the main Cloud Providers such as Amazon Web Services, Microsoft Azure or Google Cloud Platform. However, it also works with others, such as Digital Ocean, VMware, and VSphere.

There are different ways to configure the account of the main Cloud providers. The easiest way comes from authenticating using the CLI from the cloud provider.

1. Install the azure-cli package on local machine
2. Get the subscription ID from account details

```
$ az login
```

3. Set the subscription ID we want to work with in case we have different ones

```
$ az account-set --subscription="SUBSCRIPTION_ID"  
$ az account show
```

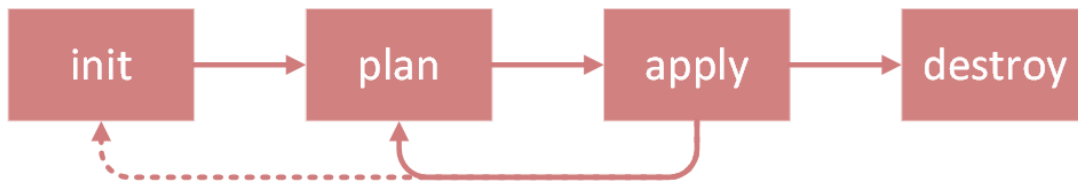
Alternative Login



Another option is to indicate the subscription ID on the HCL file when asking for the cloud provider. The fields for the login will depend on the Cloud Provider

```
provider "azurerm" {  
  version = "=1.44.0"  
  subscription_id = "<SUBSCRIPTION_ID>"  
}  
  
provider "google" {  
  credentials = file("<NAME>.json")  
  project = "<PROJECT_ID>"  
  region  = "us-central1"  
  zone    = "us-central1-c"  
}
```

Terraform Workflow



The workflow of terraform can be simplified in four different stages.

- The **terraform init** command looks through all of the *.tf files in the current working directory and automatically downloads any of the providers required for them.
- The **terraform plan** generates an execution plan indicating which are the resource actions that will be performed in case of a **terraform apply**.
- The **terraform apply** deploy the resources. The output will be the same as the previous stage but it will ask for a confirmation prompt.
- The **terraform destroy** command cleans up the resources.

Change Infrastructure



Terraform builds an execution plan by comparing your desired state as described in the configuration to the current state, which is either saved in the *terraform.tfstate* file or in a remote state backend.

When a configuration is changed, the execution plan shows what actions Terraform will take to effect the change. A plan can be saved as file by providing the correct flag.

```
$ terraform plan -out=newplan
```

Saving an execution plan with the out flag ensures your terraform apply operation runs the exact plan rather than applying new changes you may not have approved.

```
$ terraform apply "newplan"
```

Your plan output indicates that the resource will be updated in place with the ~ symbol beside the resource group. Your new resource attributes, indicated with the + symbol, will be added to the resource group.

Terraform State



Terraform must store state about your managed infrastructure and configuration. This state is used by Terraform to **map real world resources** to your configuration, **keep track of metadata**, and to **improve performance** for large infrastructures.

This state is stored by default in a local file named *terraform.tfstate* in **JSON**, but it can also be stored remotely, which works better in a team environment.

Prior to any operation, Terraform will update its state in case some changes have occurred. Once updated, it will use this local state to create plans and make changes to the infrastructure.

The primary purpose of Terraform state is to store bindings between objects in a remote system and resource instances declared in your configuration. When Terraform creates a remote object in response to a change of configuration, it will record the identity of that remote object against a particular resource instance, and then potentially update or delete that object in response to future configuration changes.

Terraform Remote State



Terraform supports team-based workflows with a feature known as **remote backends**. Remote backends allow Terraform to use a **shared storage space for state data**, so any member of a team can use Terraform to manage the same infrastructure.

The state is stored on the cloud, and it could be stored in Terraform Cloud or in other cloud provider. It will be configured on the configuration as shown below.

```
terraform {
  backend "remote" {
    organization = "<ORG_NAME>"

    workspaces {
      name = "Example-Workspace"
    }
  }
}
```

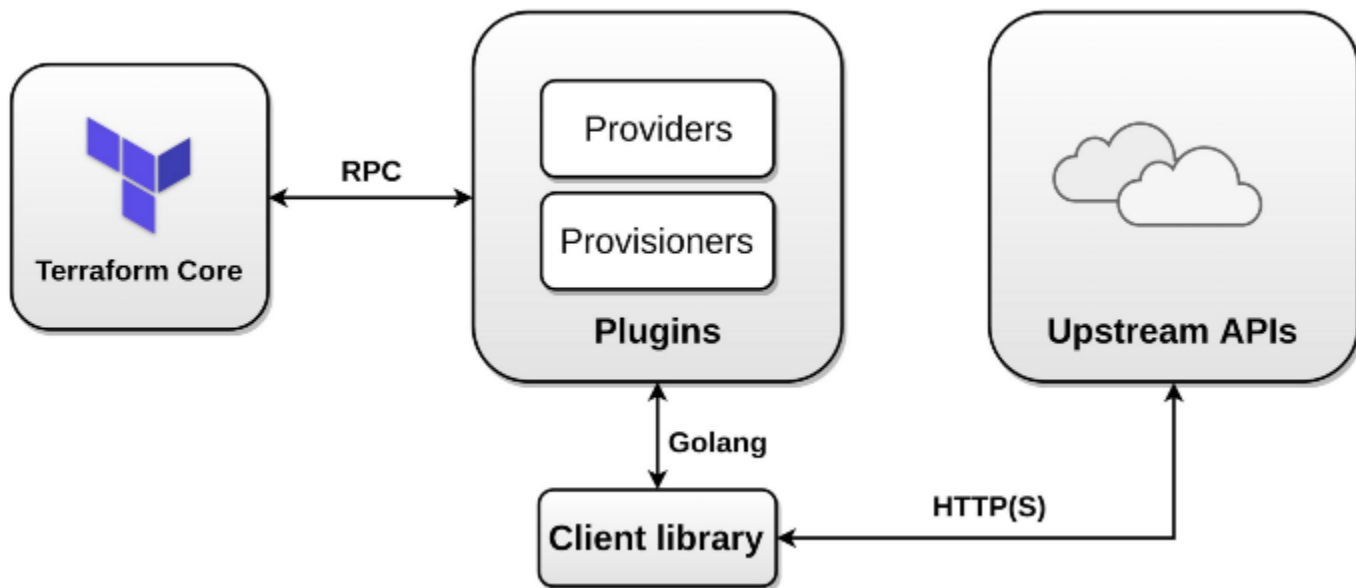
Terraform Key Concepts



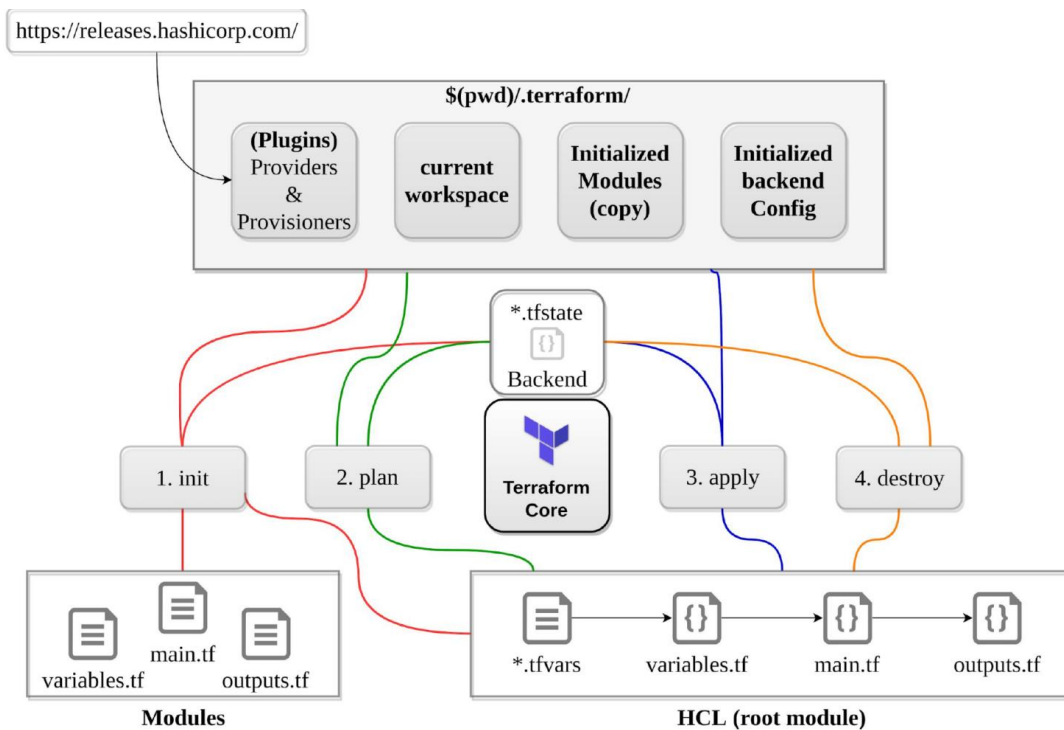
Before introducing the core and how the different components of Terraform are interconnected among them, we should introduce some concepts which are involved in the architecture.

- **Providers** are responsible for understanding API interactions and exposing resources. Some examples are Azure, AWS or GCP.
- **Resources** are the basic building block of terraform scripts. Terraform is used to create, manage and update infrastructure resources such as physical machines, VMs, network switches, containers...
- **Data sources** allow data to be fetched or computed for use elsewhere in Terraform configuration. For example, artifacts identifiers are supplied by providers.
- **Provisioners** can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service. They are used to execute scripts on a local or remote machine as well as cleaning up the resources or run configuration management.

Terraform Core



Terraform Core



File Structure



Before getting into the syntax of HashiCorp Configuration Language (HCL), the structure of a Terraform project could get complex. In the following table, the main files we can find when building a Terraform project from scratch.

All code could be written on the same file, but it will be better to having several files split logically, having different ones for different purposes.

main.tf	Configuration file. It calls modules, locals and data-sources to create all resources
variables.tf	It contains declarations of variables used in main.tf
outputs.tf	It contains outputs from the resources created in main.tf
terraform.tfvars	It persists variable values that will be loaded to populate variables

Configuration File



A **Terraform configuration** is a complete document in the Terraform language that tells Terraform how to manage a given collection of infrastructure. A configuration can consist of multiple files and directories.

Code in the Terraform language is stored in plain text files with the `.tf` file extension. There is also a JSON-based variant of the language that is named with the `.tf.json` file extension.

Terraform evaluates **all the configuration files in a module**, effectively treating the entire module as a single document. Separating various blocks into different files is purely for the convenience of readers and maintainers and has no effect on the module's behavior.

A Terraform module can use module calls to explicitly include other modules into the configuration. These child modules can come from local directories (nested in the parent module's directory, or anywhere else on disk), or from external sources like the Terraform Registry.

Sintax Introduction



The Terraform language is declarative, describing an intended goal rather than the steps to reach that goal. The ordering of blocks and the files they are organized into are generally not significant.

- **Blocks** are containers for other content and usually represent the configuration of some kind of object, like a resource. Blocks have a block type, can have zero or more labels, and have a body that contains any number of arguments and nested blocks. Most of Terraform's features are controlled by top-level blocks in a configuration file.
- **Arguments** assign a value to a name. They appear within blocks.
- **Expressions** represent a value, either literally or by referencing and combining other values. They appear as values for arguments, or within other expressions.

```
<BLOCK TYPE> "<BLOCK_LABEL>" "<BLOCK_LABEL>" {  
  # Block body  
  <IDENTIFIER> = <EXPRESSION> # Argument  
}
```


Sintax Example



```
# Configure the Azure provider
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = ">= 2.26"
    }
  }
}

provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "rg" {
  name     = "<NAME_RESOURCE_GROUP>"
  location = "westus2"
}
```

The format for resource identifier in Terraform configuration is

```
<_type_>.<_name_>
```

In the sample configuration, the resource ID is *azurerm_resource_group.rg*

This configuration provisions an *azurerm_resource_group* resource named *rg*. The resource name is used to reference the Terraform resource created in the resource block throughout the configuration. It is not the same as the name of the resource group in Azure which is described with the attribute *name*.

Introducing Variables



Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

```
variable "image_id" {
  type      = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    condition     = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"
    error_message = "The image_id value must be a valid AMI id, starting with \"ami-\"."
  }
}
```

Variable definition also accept a *default* value which will make the variable optional.

Besides, setting a variable as *sensitive* prevents Terraform from showing its value in the plan or apply output, when that variable is used within a configuration.

Input Variables



Terraform will load variables automatically from all files which match *terraform.tfvars* or **.auto.tfvars* present in the current directory.

Terraform can populate also variables using values from a custom filename or from the Command Line.

```
$ terraform apply -var 'admin_username=admin' -var 'admin_password=admin'  
$ terraform apply -var-file="testing.tfvars"
```

As a fallback for the other ways of defining variables, Terraform searches the environment of its own process for environment variables named `TF_VAR_` followed by the name of a declared variable.

```
$ export TF_VAR_admin_username=admin
```

Using Input Variables



Within the module that declared a variable, its value can be accessed from within expressions as `var.<NAME>`, where `<NAME>` matches the label given in the declaration block.

There are also built-in functions such as the *lookup* function which does a dynamic lookup in a map for a key where the first parameter is the mapping variable and the second is the key to look for.

```
variable "sku" {
  default = {
    westus2 = "16.04-LTS"
    eastus  = "18.04-LTS"
  }
}

storage_image_reference {
  sku      = lookup(var.sku, "westus2")
  version  = "latest"
}
```

Output Variables



Output values are like the return values of a Terraform module, and have several uses:

- A child module can use outputs to expose a subset of its resource attributes to a parent module.
- A root module can use outputs to print certain values in the CLI output after running terraform apply.
- When using remote state, root module outputs can be accessed by other configurations via a *terraform_remote_state* data source.

```
output "db_password" {  
  value      = aws_db_instance.db.password  
  description = "The password for logging in to the database."  
  sensitive  = true  
}
```

Data Sources



Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration. Use of data sources allows a Terraform configuration to make use of information defined outside of Terraform or defined by another separate Terraform configuration.

A data block requests that Terraform read from a given data source and export the result under the given local name. The name is used to refer to this resource from elsewhere in the same Terraform module but has no significance outside of the scope of a module.

```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name    = "app-server"
    Tested = "true"
  }
}
```

Modules



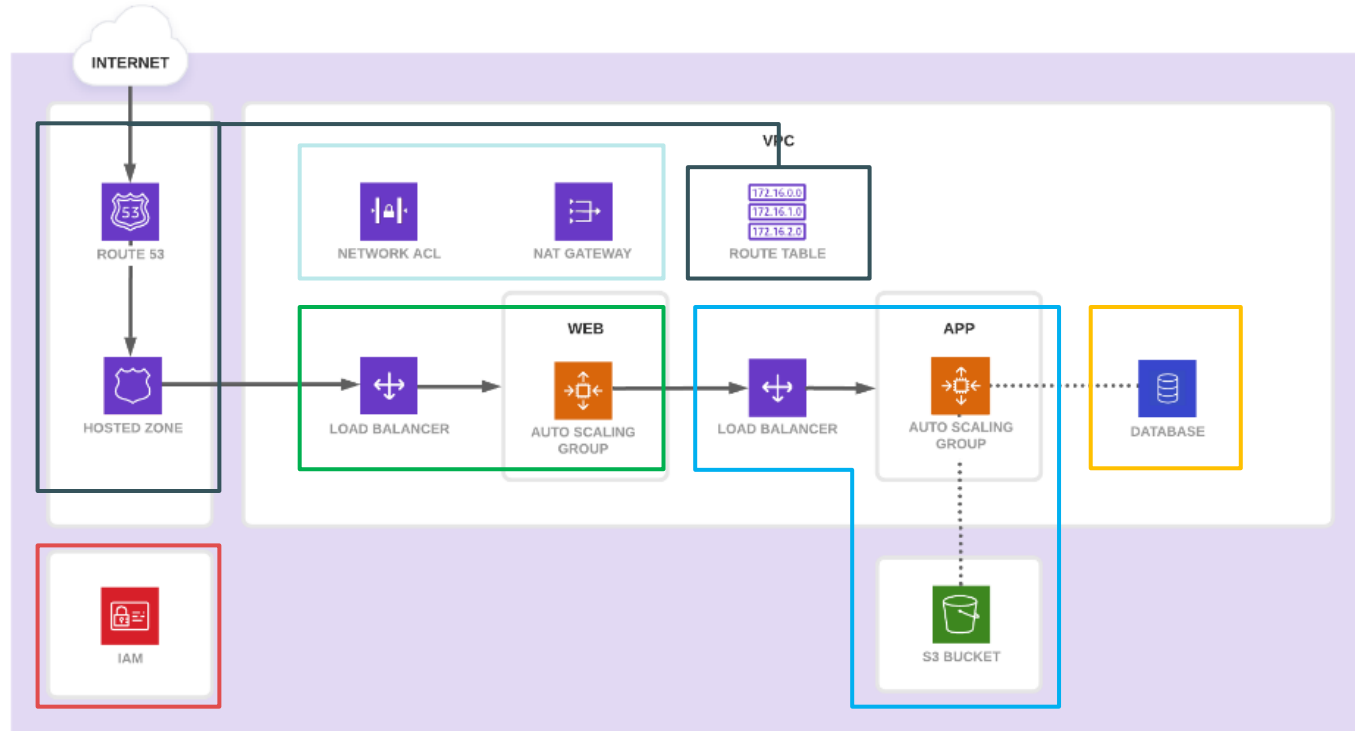
Modules make it easier to navigate, understand, and update the configuration by keeping related parts of the configuration together. Even moderately complex infrastructure can require hundreds or thousands of lines of configuration to implement. By using modules, you can organize the configuration into logical components.

Another benefit of using modules is to encapsulate configuration into distinct logical components. Encapsulation can help prevent unintended consequences, such as a change to one part of the configuration accidentally causing changes to other infrastructure and reduce the chances of simple errors like using the same name for two different resources.

Modules also provides a way for re-using configurations as well as providing consistency in the configurations. Not only does consistency make complex configurations easier to understand, but it also helps to ensure that best practices are applied across all the configuration.

Terraform modules are self-contained pieces of infrastructure-as-code that abstract the underlying complexity of infrastructure deployments.

Modules Example



Modules Overview



A Terraform module is a set of Terraform configuration files in a single directory. Even a simple configuration consisting of a single directory with one or more .tf files is a module. When you run Terraform commands directly from such a directory, it is considered the **root module**.

A Terraform module can call other modules to include their resources into the configuration. A module that has been called by another module is often referred to as a **child module**. Child modules can be called multiple times within the same configuration, and multiple configurations can use the same child module.

In addition to modules from the local filesystem, Terraform can load modules from a public or private registry. This makes it possible to publish modules for others to use, and to use modules that others have published.

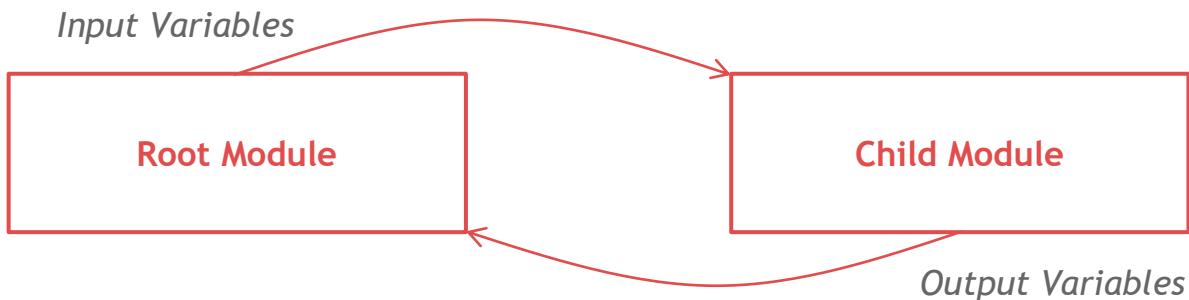
The public **Terraform Registry** can be used to find useful modules. This way it can be more quickly and confidently implement a configuration by relying on the work of others to implement common infrastructure scenarios

Module Structure



Re-usable modules are defined using all of the same configuration language concepts we use in root modules. Most commonly, modules use:

- **Input variables** to accept values from the calling module.
- **Output values** to return results to the calling module, which it can then use to populate arguments elsewhere.
- **Resources** to define one or more infrastructure objects that the module will manage.



Standard Structure



The **root module** is the only required element for the standard module structure. Terraform files must exist in the root directory of the repository.

main.tf, *variables.tf*, *outputs.tf* are the recommended filenames for a minimal module, even if they're empty. The main file should be the primary entrypoint.

For a complex module, resource creation may be split into multiple files, but any nested module calls should be in the main file.

```
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── ...
└── modules/
    ├── nestedA/
    │   ├── README.md
    │   ├── variables.tf
    │   ├── main.tf
    │   └── outputs.tf
    ├── nestedB/
    └── .../
```

Modules Input Variables



When using a module, variables are set by passing arguments to the module in your configuration. These variables are set when calling this module from the root module's main.tf.

Variables defined in modules that aren't given a default value are required, and so must be set whenever the module is used.

```
module "website_s3_bucket" {  
  source = "../modules/aws-s3-static-website-bucket"  
  
  bucket_name = "<UNIQUE_BUCKET_NAME>"  
  
  tags = {  
    Terraform    = "true"  
    Environment = "dev"  
  }  
}
```

Modules Output Variables



Like variables, outputs in modules perform the same function as they do in the root module but are accessed in a different way. A module's outputs can be accessed as read-only attributes on the module object, which is available within the configuration that calls the module. They can be referenced in expressions as `module.<MODULE NAME>.<OUTPUT NAME>`

```
module "network" {  
  source = "../modules/aws-network"  
  
  base_cidr_block = "10.0.0.0/8"  
}  
  
{  
  ...  
  vpc_id = module.network.vpc_id  
  ...  
}
```

Introduction to Cloudify



CLOUDIFY

Introduction to Cloudify



Cloudify is an open-source multi-cloud and edge orchestration platform. Cloudify allows organizations an effortless transition to public cloud and Cloud-Native architecture by enabling them to automate their existing infrastructure alongside cloud native and distributed edge resources.

Cloudify also allows users to manage different orchestration and automation domains as part of one common CI/CD pipeline. Key Features Everything as a Code Service Composition Domain-Specific Language (DSL) - enabling modeling of a composite service, containing components from multiple Cloudify services and other orchestration domains.

Cloudify enables you to deploy applications using two main methods: using the **CLI** only or using a **Cloudify Manager**. The Cloudify Manager comprises Cloudify's code and several underlying open-source tools, which have been integrated to create a dynamic environment, and will support the different operational flows that someone might be interested in when deploying and managing an application.

Based on TOSCA



TOSCA (Topology Orchestration Specification for Cloud Applications) is governed by the OASIS organization and has two basic building blocks on which the concept relies: nodes, and relationships.

- Nodes can be infrastructure components (such as a subnet, network, server, or cluster of servers), or software components (such as services or runtime environments).
- A relationship defines how nodes are connected to one another.
- TOSCA also contains the concept of types. For example, a “*compute*” node representing a resource with a CPU. This type can be used in “*service templates*” or, as referred to in Cloudify’s dashboard, a “*blueprints*”.

The fact that TOSCA is backed by a standards body (OASIS) makes it a great platform for defining a standard container orchestration specification that is portable across various cloud environments and container providers.

TOSCA Introduction



The TOSCA metamodel uses the concept of service templates to describe cloud workloads as a topology template, which is a graph of node templates modeling the components a workload is made up of and as relationship templates modeling the relations between those components.

TOSCA further provides a type system of node types to describe the possible building blocks for constructing a service template, as well as relationship type to describe possible kinds of relations. Both node and relationship types may define lifecycle operations to implement the behavior an orchestration engine can invoke when instantiating a service template.

An orchestration engine processing a TOSCA service template uses the lifecycle operations to instantiate single components at runtime, and it uses the relationship between components to derive the order of component instantiation.

Template authors in many cases will not have to define types themselves but can simply start writing service templates that use existing types. In addition, the simple profile will provide means for easily customizing existing types, for example by providing a customized 'create' script for some software.

Blueprints



Blueprints are **YAML documents** written in Cloudify's DSL (**Domain Specific Language**), which is based on TOSCA. Blueprints will describe the logical representation, or topology, of an application or infrastructure.

A blueprint can contain multiple files. These files can reside under a single directory with subdirectories or in an archive. Although the Cloudify CLI can manage the archiving process for you during upload, someone might want to create archives prior to uploading the blueprint, so that you can keep them in a fileserver, upload them via the Cloudify Management Console, or send them to others.

Blueprints use the Cloudify DSL to model an application. The model, or `node_templates` section, describes a topology which includes node templates and relationships between node templates.

Workflows actualize the topology by defining the order of operations that will be performed. For example, the built-in Install Workflow calls, among other things, the create, configure, and start operations defined for a particular node type.

Blueprint Structure



There are four main keys in a blueprint file.

<code>tosca_definitions_version</code>	This is the version of Cloudify's DSL. You should not need to change this value.
<code>imports</code>	This is a list of URLs or paths to more Cloudify DSL files. These define <code>node_types</code> and relationships, among other things, that will be used in your blueprint.
<code>inputs</code>	These are parameters that you should know before running your blueprint, such as API endpoints.
<code>node_templates</code>	These are the nodes that you will handle throughout your deployment, such as VMs, applications, or network components.

Blueprint Example



This is an example of a blueprint for a computing node.

```
tosca_definitions_version: cloudify_dsl_1_3
imports:
  - http://cloudify.co/spec/cloudify/5.0.0/types.yaml
inputs:
  ...
node_templates:
  my_server:
    type: cloudify.nodes.Compute
    properties:
      ip: { get_input: ip_address }
      ...
outputs:
  server_ip:
    description: The private IP address of the provisioned server.
    value: { get_attribute: [ my_server, private_address ] }
```

Plugins



Plugins contain the Python code that each workflow operation calls. In the blueprint, they are imported under the import tag, for example:

```
imports:  
  - plugin: cloudify-azure-plugin
```

You can then map node template and relationship operations to plugin code, or if your plugin plugin.yaml has custom node types, these operations may already be mapped for you.

- If you want to create a VM in Azure, you will need the Azure plugin.
- If you want to configure a server with an Ansible playbook, you will use the Ansible plugin.
- If you have existing scripts, you may simply use the built-in Script plugin.

Login w/ blueprints



Blueprints provide the necessary plugins to login into the main Cloud Providers. The credentials will be passed as arguments when using the plugin. These could be stored as secrets and then accessed inside a blueprint as follows for login into Microsoft Azure.

```
resource_group:
  type: cloudify.azure.nodes.ResourceGroup
  properties:
    name: my_resource_group
    location: { get_secret: location }
    azure_config:
      subscription_id: { get_secret: subscription_id }
      tenant_id: { get_secret: tenant_id }
      client_id: { get_secret: client_id }
      client_secret: { get_secret: client_secret }
```

Interfaces



Interfaces are reusable entities that define a set of operations that can be included as part of a Node type or Relationship Type definition. Each named operations may have code or scripts associated with them that orchestrators can execute for when transitioning an application to a given state.

```
node_templates:
  vm:
    type: cloudify.openstack.nodes.Server
  nodejs:
    type: nodejs_app
    interfaces:
      my_deployment_interface:
        ...
        start: scripts/start_app.sh
```

In this way, you can define your interfaces either in `node_types` or in `node_templates`, depending on whether you want to reuse the declared interfaces in different nodes or declare them in specific nodes.

Cloudify Execution



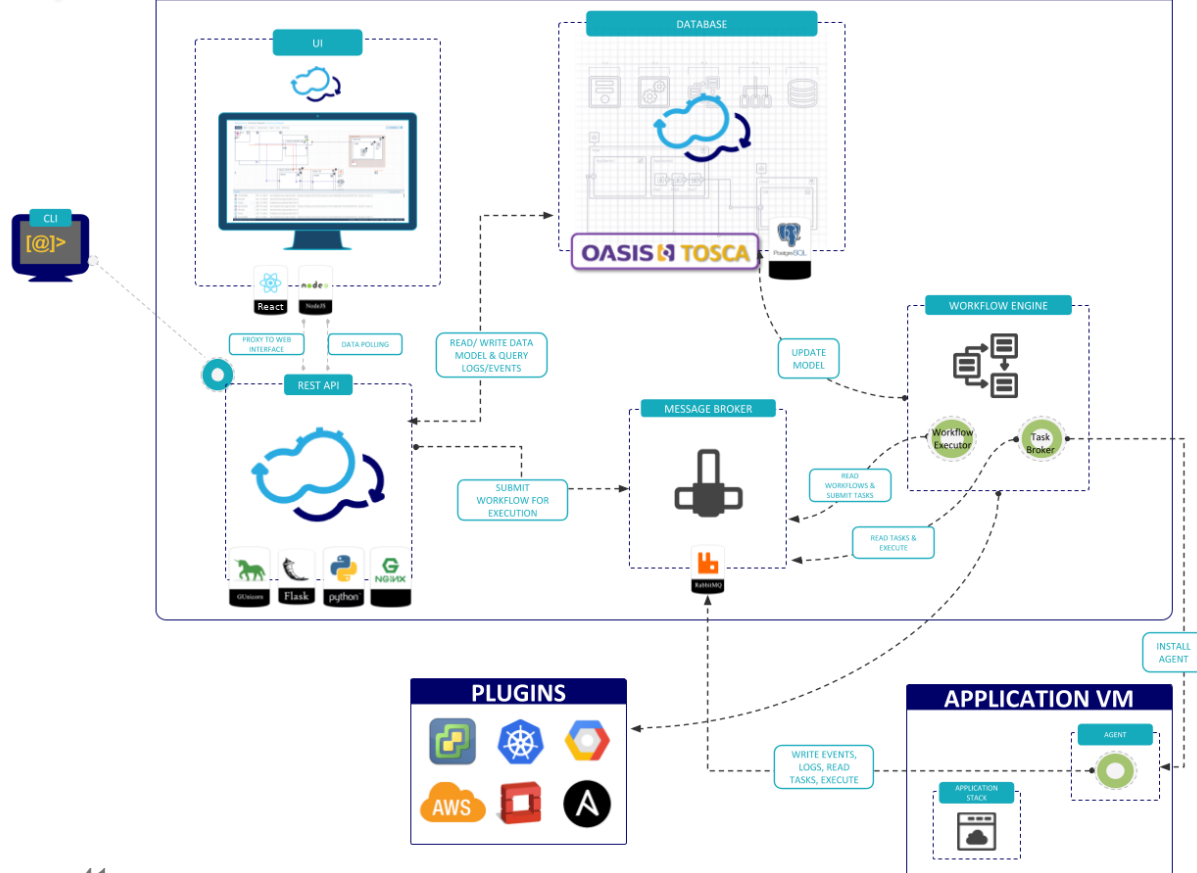
The Cloudify command-line interface (CLI) is the default method for interacting with the Cloudify management environment, to manage your applications. It enables you to execute workflows on your local machine, and to interact with a running Cloudify Manager via SSH to upload and delete blueprints, create deployments, execute workflows, retrieve events, and more.

Working locally refers to running workflows directly from the machine on which the CLI is installed. Working with an instance of Cloudify Manager refers to executing workflows directly from that Cloudify Manager instance.

When you use the CLI to control a Cloudify Manager instance, additional commands appear in the CLI that are not available for use in local mode, for example communicating with a running Cloudify Manager using SSH, downloading its logs, creating snapshots, uploading plugins and so on.

```
$ cfy blueprints upload [options] BLUEPRINT_PATH
```


CLOUDIFY MANAGEMENT ENVIRONMENT



Terraform vs. Cloudify



Terraform



CLOUDIFY

Terraform Strengths



- **Diversity.** Terraform provides support for a diverse list of providers. This means you can get up and running with basic configurations on major cloud platforms such as AWS, Azure, Google...

Using HCL configuration files, Terraform can manage entire buildouts in a state file (tfstate). This state file allows Terraform to create, destroy, and modify components idempotently in a dependency understood methodology.

- **Easy to get started.** Terraform's HCL is similar in nature to most YAML-like configurations with plenty of examples in its documentation as well as online articles. You can quickly create a few instances, a VPC, or an entire stack.
- **Fast Standup.** There is also a fast evaluation of your managed environment and then deciding additions, deletions, resource cycling, and updates. Using resource imports, Terraform has the flexibility to bring manually created resources into management.
- **Wide Adoption (Open Source).** There is broad market adoption of Terraform. Terraform requires a much lighter lift to get changes into production compared to some of its competitors.

Cloudify Strengths



- **Full-Scale Service Orchestration.** Cloudify can be used to deploy a web server, an application ecosystem, or an entire environment. By creating blueprints, the application allows you to deploy and reuse entire configurations without having to DRY your code.
- **Controlled Operations as Code or GUI.** Cloudify offers a rich experience for using CLI or GUI operations. Whether you want to configure blueprints that are fully TOSCA-compliant or manage blueprints and deployments through the Cloudify portal.

It enables you to one-click your way through architectural relationship diagrams. For simple server-to-database models, this isn't so important. But when you consider entire ecosystems, the connecting tissue is not so obvious from code observation alone.

- **Manage All Sizes of Environments.** Cloudify handles calls, error handling and provider connections (even Terraform) through plugins. Blueprints are reusable and can be easily expanded to deploy additional and/or larger environments without having to redesign your deployment architecture.

Terraform Use Cases



- **Stacks, Environments.** Terraform shines when it manages stacks or environments. Since Terraform is already very good at understanding resource dependencies and allows for customizable dependencies, stacks can be created with confidence rather than having to manually handle the orchestration.
- **Small to Large.** The more an environment grows, the more difficult it gets to DRY out Terraform code. Also, evaluating very large environments can be a slow process, not to mention dangerous if not planned correctly. As the environment grows larger, it's best to design sections of your environment as small chunks of managed state files.

This will help control potential changes, keep compilation of changes shorter, and make it easier to manage multiple changes.

- **Fully Cloud Native.** Terraform really shines when it is managing stacks utilizing statelessness, such as auto-scaling groups, lambda functions, and network resources. It's better to avoid having Terraform manage individual stateful instances or volumes because it can easily destroy resources.

Cloudify Use Cases



- **Ecosystems, end-to-end.** Given Cloudify's ability to manage large deployments in a graphical and codified way, it provides the greatest value when showing dependencies and managing those dependencies in a seamless method.
- **Small to Very Large.** Blueprints can be a single component or mixed ecosystems containing thousands of servers. Since the API handles the calls and regulates the deployment of resources, it removes some of the design burdens for very large constructs. Cloudify's portal allows the user to gain quick insights across all deployments or individual deployments without the additional implementation of logging and metrics.

Cloudify is TOSCA compliant and is designed to provide end-to-end orchestration. This cradle-to-grave construct means that users can deploy an environment and manage that environment through Cloudify until decommission. This provides enterprise customers change management, auditing, and provisioning capabilities to control deployments of interdependent resources through their entire lifecycle.

Features Comparison



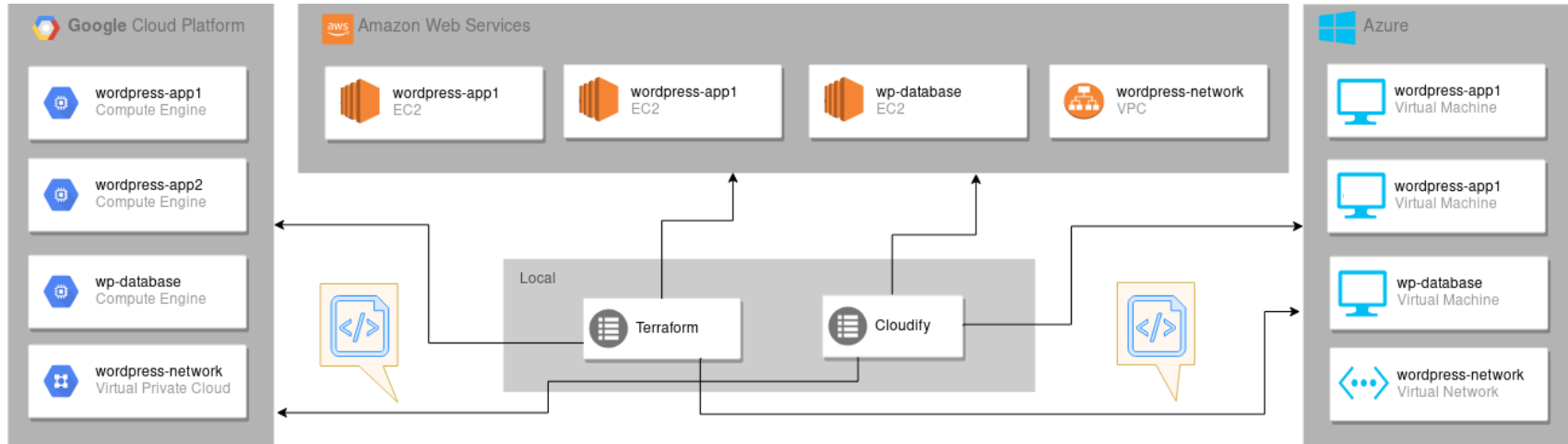
This comparative table presents the relevant features of both tools in order to be able to see the comparison at a sight.

Orchestrator	Integrable with	OpenSource	Artifact	CLI	API	Graphic UI
Terraform	AWS, GCP, Azure, OpenStack, Oracle, vSphere, and more than 30	Yes	HCL JSON	Yes	Online functions	No
Cloudify	AWS, GCP, Azure, OpenStack, vCloud and vSphere	Yes	TOSCA YAML	Yes	Yes	Yes

Performance Research



For this research, the following architecture was used for running Wordpress and a MySQL Server by using similar machines to mitigate the interference of differences in the provision of services.



Performance Research

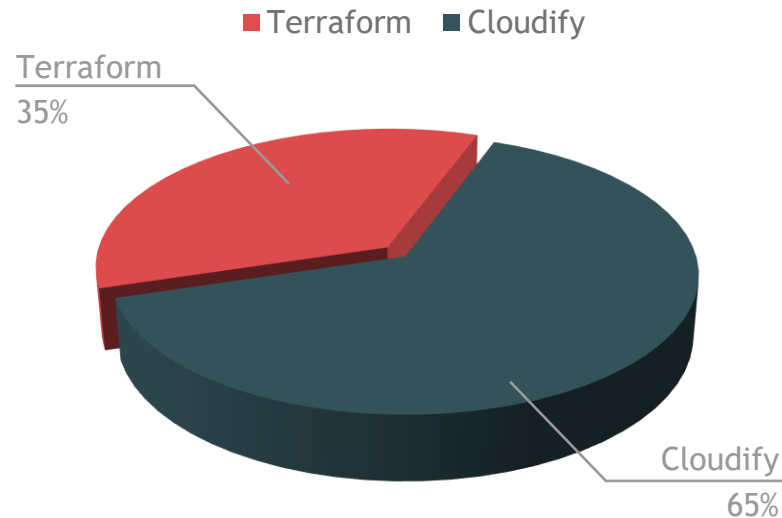


This graph demonstrates the distribution of all computational resources involved in the experiment. During the process, Cloudify kept about 2/3 of the resources while Terraform kept the remainder 1/3.

Thus, Cloudify seems to be less efficient than Terraform in the proposed scenario in terms of CPU usage.

In the following slides, other parameters such as RAM memory usage, network average traffic, average execution time and I/O average activity are compared

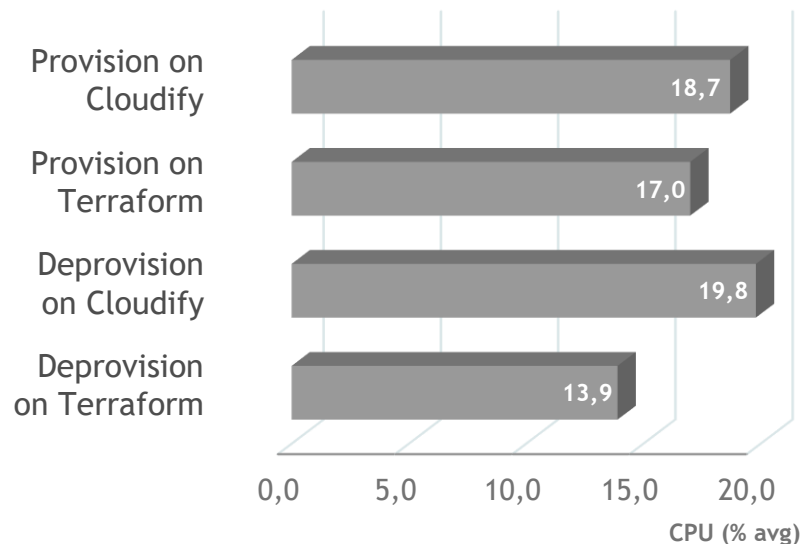
Global Resource Allocation



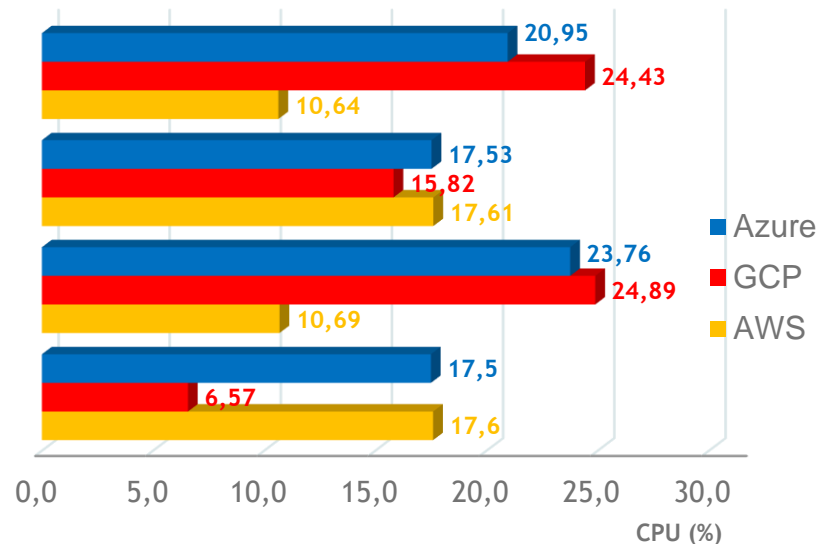
CPU Usage



CPU Average Usage



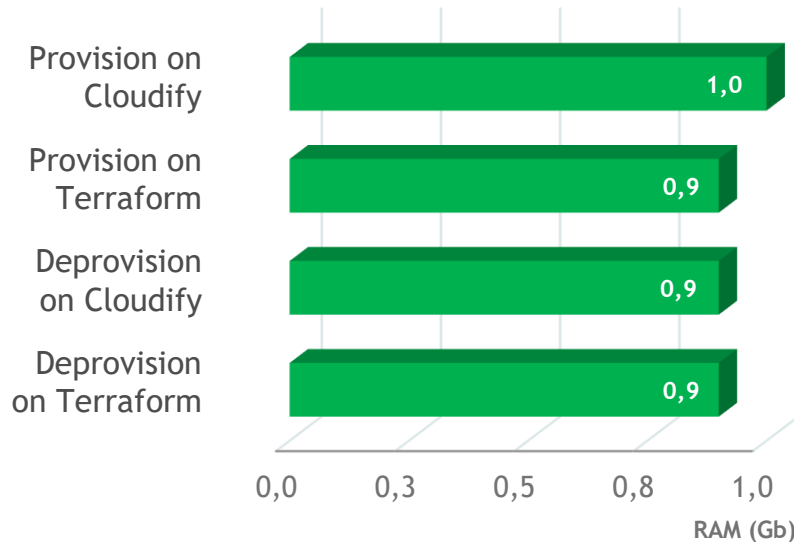
CPU Usage



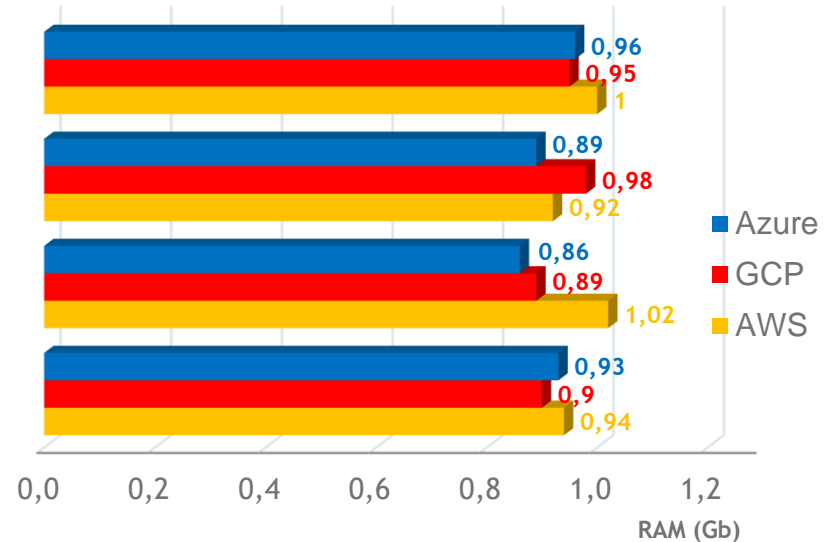
RAM Memory Usage



RAM Memory Average Usage



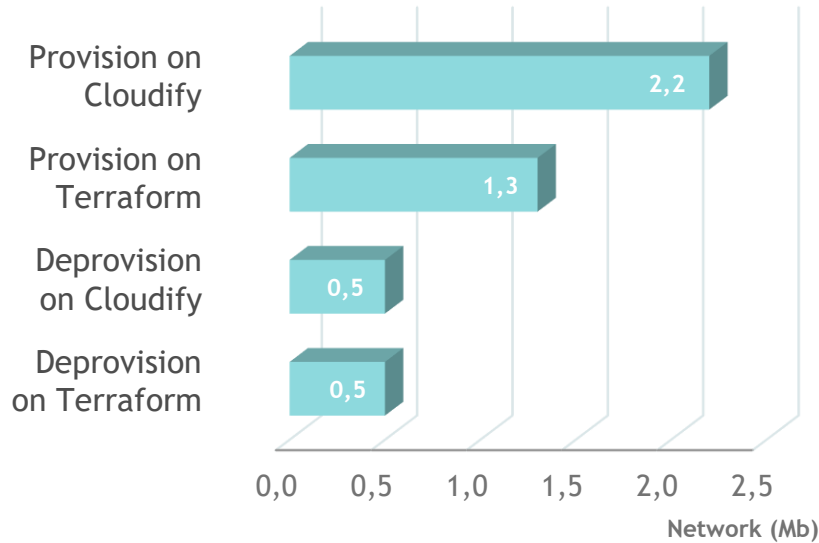
RAM Memory Usage



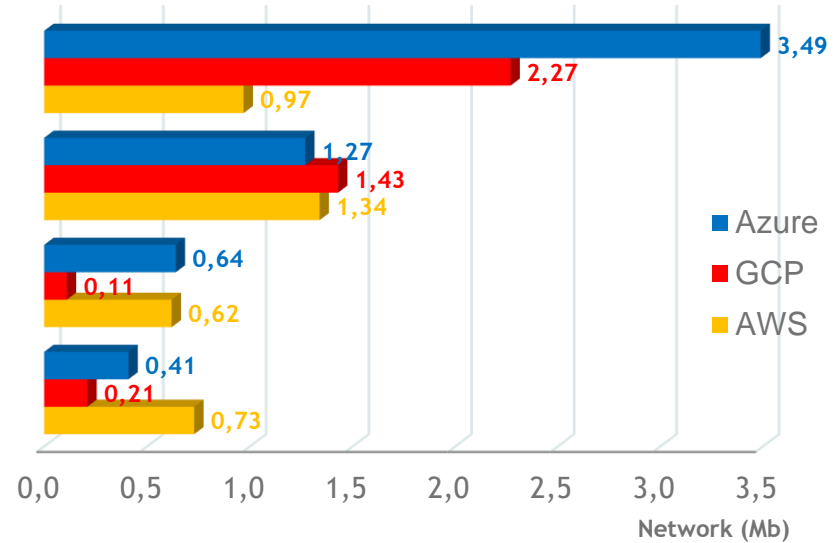
Network Traffic



Network Average Traffic



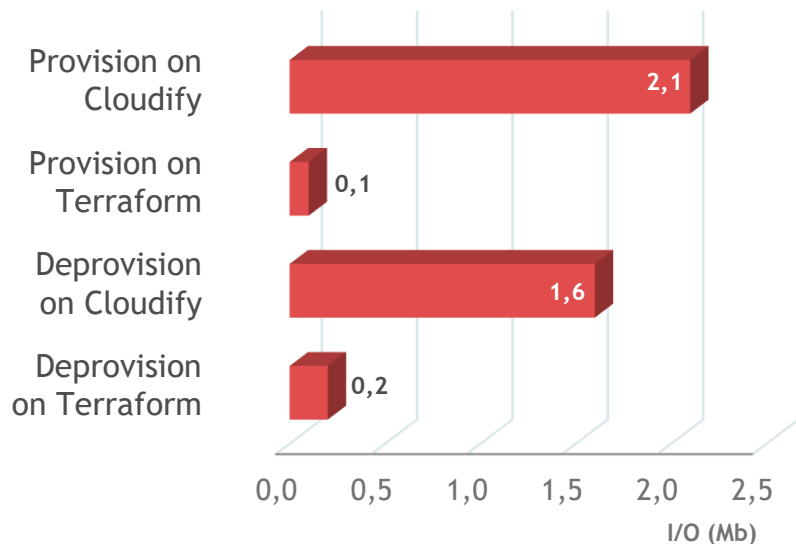
Network Traffic



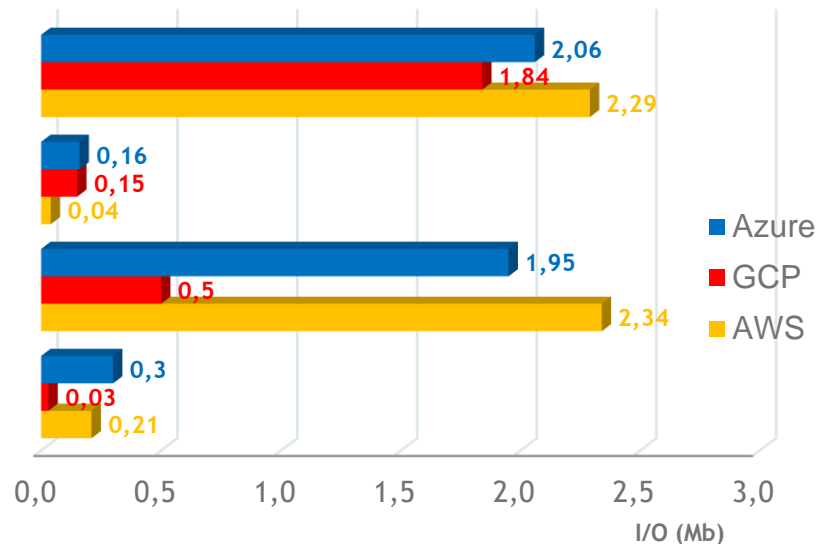
I/O Activity



I/O Average Activity



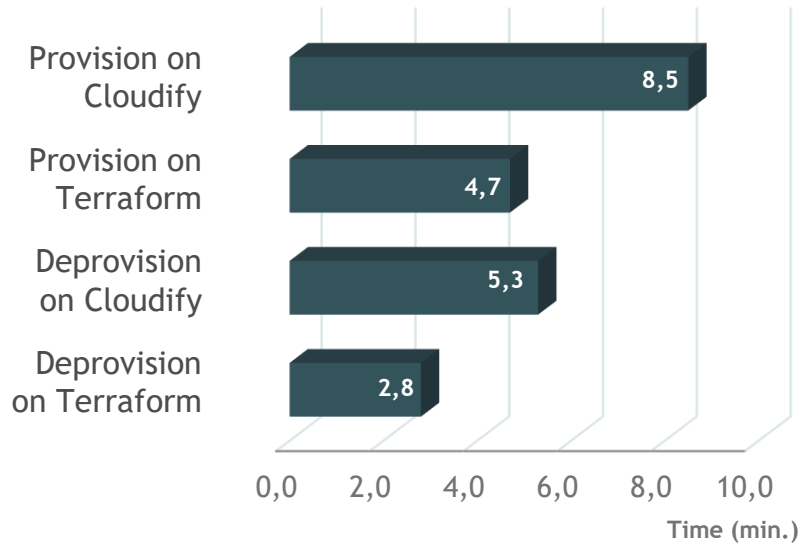
I/O Activity



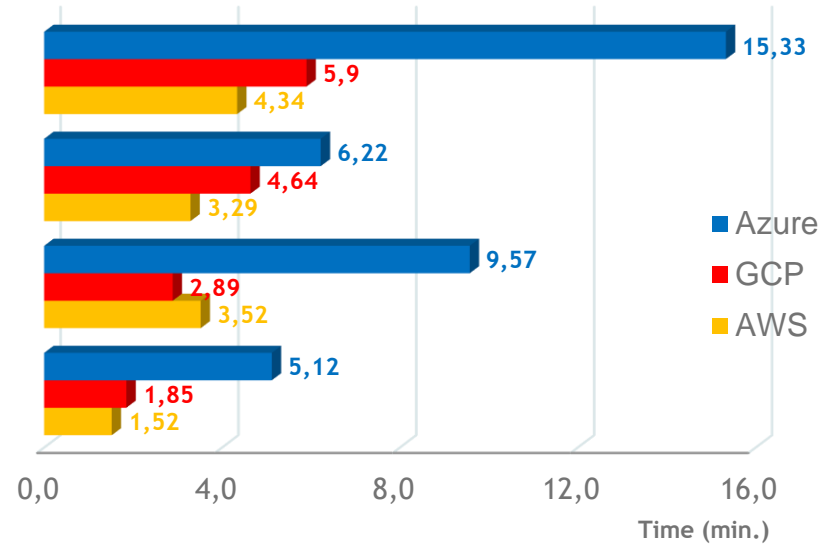
Execution Time



Average Execution Time



Execution Time



Thanks

Tasio Méndez Ayerbe
tasio.mendez@mail.polimi.it

Politecnico di Milano



POLITECNICO
MILANO 1863

References

de Carvalho, L. R., & de Araujo, A. P. F. (2020, May). *Performance Comparison of Terraform and Cloudify as Multicloud Orchestrators*. In 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID) (pp. 380-389). IEEE.

Wurster, M., Breitenbücher, U., Falkenthal, M., Krieger, C., Leymann, F., Saatkamp, K., & Soldani, J. (2020). *The essential deployment metamodel: a systematic review of deployment automation technologies*. SICS Software-Intensive Cyber-Physical Systems, 35(1), 63-75.

Terraform by HashiCorp. *Terraform Documentation*. Retrieved December, 2020, from <https://www.terraform.io/docs/>

Cloudify. *Cloudify Documentation*. Retrieved December, 2020, from <https://docs.cloudify.co/latest/>

Leonardo Rebouças de Carvalho. GitHub. Retrieved December, 2020, from <https://github.com/leonardoreboucas/cloud-orchestrators-exam>