Prodapt powering global telecom

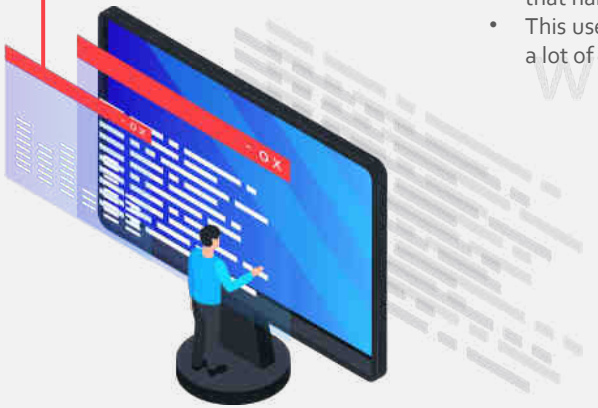
Don’t let the infrastructure management cloud your mind

Implement Infrastructure as Code (IaC) to reduce provisioning time by 65%

Credits | Deepak Jayagopal | Kanapathi Raja | Sathya Narayanan

Higher elasticity and scale of infrastructure has made it cumbersome to manually provision even after the adoption of cloud computing

Evolution of infrastructure provisioning



Legacy manual provisioning of infrastructure

- Traditionally, Infrastructure provisioning has always been a manual process.
- Teams would rack and stack the servers and will manually configure them.
- Finally, they will install the application over that hardware.
- This used to be a slow process and there were a lot of accuracy and consistency issues.

Cloud manual provisioning

- Using cloud computing (Infrastructure as a Service), an instant computing infrastructure could be provisioned and managed over the internet.
- Cloud computing reduced the need for high upfront capital expenditure as well as maintenance costs.

Infrastructure as Code (IaC)

Infrastructure as Code (IaC) is the process of provisioning and managing the entire infrastructure through a series of software.

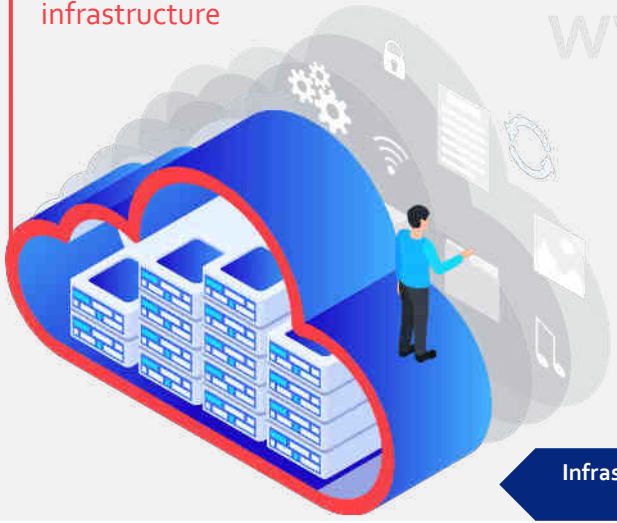
IaC allows DevOps engineers to view complex infrastructure in a codified way.

IaC takes cloud computing to its maximum potential by automating the manual, error-prone provisioning tasks.

Cloud infrastructure solved many issues pertaining to the legacy provisioning methods. However, there are still issues persisting with manual provisioning of cloud infrastructure.

Major challenges with manual provisioning of cloud infrastructure

DSPs across the globe have a great necessity to adopt a strong cloud strategy to deliver digital services across their customer ecosystem. However, they face several challenges with manual provisioning of cloud infrastructure



Time consuming in provisioning infrastructure

In the case of huge infrastructures, the time taken to manual cloud provisioning gets increased tremendously. The complexity and time consumed further increases when the provisioning involves multi-country or multi geographic locations.

Higher cost

Cost is calculated per hour by cloud vendors. It becomes tough to manually decommission infrastructure every time there's less demand.

Configuration consistency

While commissioning huge infrastructures, multiple cloud architects work in it. It is very tough to achieve configuration consistency with cloud manual provisioning when multiple architects are working on a provisioning same infrastructure.

Limited efficiency

The efficiency of provisioning depends on the efficiency and expertise (Experience & expertise of the architect with respect to that cloud vendor) of the architect working on it. When multiple cloud providers are used by a DSP, the efficiency of the architect is limited to his efficiency with that cloud vendor.

Infrastructure as Code (IaC) the solution to the above-mentioned issues. It allows devops engineers to encapsulate the entire infrastructure similar to a software code.

Potential scenarios where Digital Service Providers (DSPs) can embrace Infrastructure as code

Spin up dynamic Disaster Recovery (DR) on cloud

In case of large infrastructure, setting up an always-active (hot) Disaster Recovery site is very expensive. Spinning up a dynamic DR on cloud can bring down costs by up to 90%.

Migration from On-Premises to cloud infrastructure

Migration of Infrastructure from on-premise to Cloud can be done effectively using IaC. This reduces the total operating expenses and speeds up the migration process.

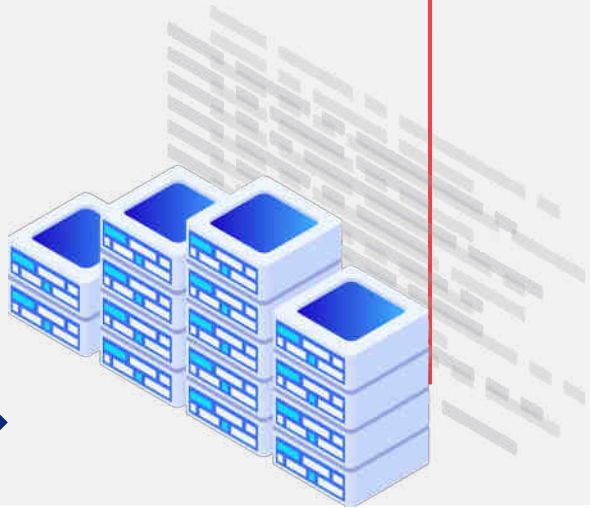
Expanding the existing cloud infrastructure

Expanding the existing cloud environment using IaC reduces complexity, time and improves configuration consistency.

Managing the existing cloud infrastructure

Managing the existing cloud infrastructure can be done in a more efficient way using configuration codes. This reduces the downtime in case of an infrastructure or application failure.

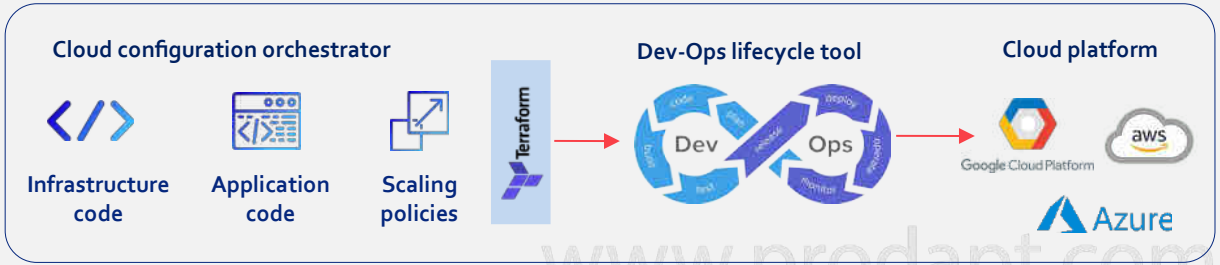
As the size of the infrastructure increases (Multi-geographic, multiple cloud providers), the effectiveness and benefits realized through IaC gets compounded



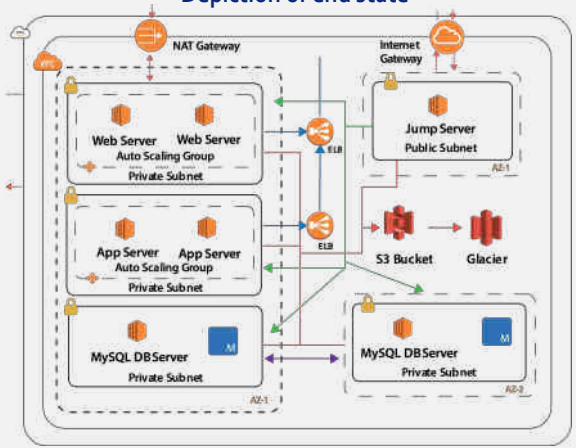
Infrastructure as Code (IaC) allows easy reproducibility of systems, dynamic design and provides a high level of flexibility for provisioning and managing the infrastructure.

Leveraging cloud configuration orchestrator for effective implementation of DevOps Infrastructure as Code

Infrastructure as Code



Depiction of end state



Components of cloud configuration orchestrator

Components of cloud configuration orchestrator

Infrastructure code
Defining the Infrastructure in a codified way

Application code
Defining the application that gets provisioned over infrastructure in a codified way

Scaling policies as code
Defining the dynamic autoscaling policies as code

DevOps Integration

- An efficient DevOps tool (Jenkins, GitLab, Docker) is required to maintain a clean repository and CI/CD integration.
- A Cloud provider (AWS, GP, Azure, etc.) is used to provide the infrastructure

Components of End state infrastructure on chosen cloud provider

- A virtual private cloud (VPC) provides an isolated and highly-secure environment to run your virtual machines and applications.
- Storehouse to provide low-cost data storage with high availability and durability.
- Monitoring tools monitor the metrics and send notifications when the metrics fall out of the threshold levels.
- Backup server to have an image of infrastructure and will be used as a recovery tool in case of any disaster.

Infrastructure code: Defining the cloud infrastructure in a codified way using customizable configuration files

The below features of Cloud configuration orchestrator aids in achieving efficient codification of infrastructure.

Pre-Built configuration files

The orchestrator contains pre-built configuration files for different cloud providers to provide the infrastructure. The configuration files in JSON format can be easily adapted in accordance with the chosen cloud provider and business needs.

Execution plan- Generation and preview

The orchestrator shows a preview of actions before it modifies/manipulates the infrastructure to execute the plan. This allows DevOps engineers to correct any errors that might have happened during the codification process.

Single command decommission

The orchestrator contains commands to decommission/destroy the infrastructure. For example, a sample QA infrastructure deployed to test the features can be destroyed using a single command.

Multi-cloud deployment

A single script can be used to perform multi-cloud deployment thereby mixing together resources from multiple cloud vendors in a single deployment plan to build an application that is more resilient to cloud service outages.

Infrastructure code (sample)

```
resource "aws_instance" "jump" {
  ami           = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type = "t2.micro"

  # the VPC subnet
  subnet_id = "${aws_subnet.moonshot-public.id}"

  # the security group
  vpc_security_group_ids = ["${aws_security_group.allow-ssh.id}"]

  # the public SSH key
  key_name = "${aws_key_pair.mykey.key_name}"
  tags = {
    Name = "Jump"
  }
}
```

Execution plan (sample)

```
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:

  + vsphere_virtual_machine.vm
      id:                                                   <computed>
      boot_retry_delay:                                     "10000"
      change_version:                                       <computed>
      clone.#:                                              "1"
      clone.0.customize.#:                                  "1"
      clone.0.customize.0.ipv4_gateway:                     "10.1.149.1"
      clone.0.customize.0.network_interface.#:              "1"
      clone.0.customize.0.network_interface.0.ipv4_address: "10.1.149.29"
      clone.0.customize.0.network_interface.0.ipv4_netmask: "24"
      clone.0.customize.0.timeout:                          "10"
      clone.0.customize.0.windows_options.#:                "1"
      clone.0.customize.0.windows_options.0.auto_logon_count: "1"
      clone.0.customize.0.windows_options.0.computer_name:  "winiltest1"
      clone.0.customize.0.windows_options.0.domain_admin_password: <sensitive>
      clone.0.customize.0.windows_options.0.domain_admin_user:  "administrat
r@cloud.local"
```

Sample use case 1: A leading DSP in Europe leveraged this codification process to setup a disaster recovery infrastructure (700 servers spread across 7+ countries)

Cloud configuration orchestrator

Infrastructure codes

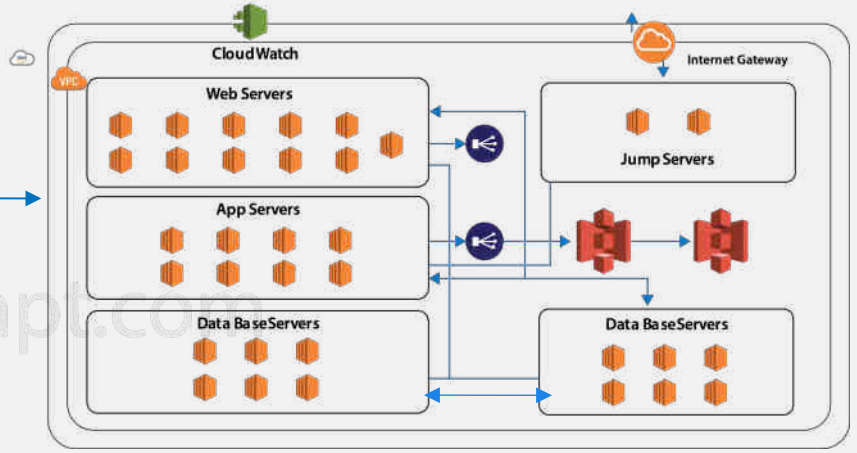
Execution plan

*Sample representation

Input Code contains codes for:

- 1 Virtual private cloud
- 700 EC2 instances comprising 550 T2 instances (General purpose), 50 C4 instances (Computer-optimized), 30 R4 instances (Memory-optimized instances), 40 G3 instances (accelerated computing) and 30 I3 instances (Storage optimized)
- 6 load balances comprising of 3 application loan balancers, 2 network load balancers, and 1 classic load balancer.
- 1 cloud watch monitoring tool
- Storage components with Amazon S3 and Amazon glacier

*The entire infrastructure is encapsulated as codes within cloud configuration orchestrator using configuration templates and a single script covers the entire infrastructure envisioned at the start.



*Sample representation

Defining infrastructure in a codified form reduces the provisioning time by 55-60% and reduces the dependency of a cloud architect's expertise.

Application code: Defining the applications deployed on infrastructure provisioned by building customizable schemas

The features of the Cloud configuration orchestrator aids in defining the applications and their configurations are mentioned below.

Reusable and customizable schemas

Reusable and customizable schemas are coded to provision applications on top of the infrastructure commissioned. This reduces significant time compared to installing separate applications on top of the infrastructure. The schemas are created in JSON format and can be easily reused for managing or upgrading the applications in the infrastructure.

Grouping applications

Within this orchestrator, DSPs can describe the application groups and install applications for the entire group. For example, 8 app servers can be grouped together and video recording management applications can be installed on all the databases in a single go.

Parallel provisioning of applications

Once the infrastructure is provisioned, the deployment of applications can be done in parallel with one script. This reduces a significant amount of time. Video recording management and recommendation engine can be parallelly installed across 2 different app servers or two groups of app servers parallelly at the same time.

Resource schedulers

The resource grid can be used to schedule provisioning of applications during specific time frames when the expected incoming load on infrastructure is least. For example, patch upgrades for RMS application can be scheduled post the peak hours of incoming traffic to avoid any configuration drifts.

Third-party integrations

This orchestrator can be integrated with other provisioning tools such as Chef, Ansible, Puppet, etc for specific application provisioning.

Application code (sample)

```
hosts: customerbs_add
  become: True
  tasks:
    - name: stop BSLGI services
      service: name="{{ item }}"
               state=stopped
      with_items:
           - 'vrm-lgienhapi-bs'

    - name: install VRM Patches
      command: yum update -y --disablerepo=* --enablerepo=vrm
      args:
        warn: no

    - name: Reload systemd
      command: systemctl daemon-reload
```

Integration with other provisioning tools

```
provider "chef" {
  server_url = "https://api.chef.io/organizations/example/"

  # You can set up a "client" within the Chef Server management console.
  client_name  = "terraform"
  key_material = "${file("chef-terraform.pem")}"
}

# Create a Chef Environment
resource "chef_environment" "production" {
  name = "production"
}

# Create a Chef Role
resource "chef_role" "app_server" {
  name = "app_server"

  run_list = [
    "recipe[terraform]",
  ]
}
```

Sample use case 2: A leading DSP in Europe leveraged codification process to provision applications on the infrastructure spawned

Cloud configuration orchestrator

Infrastructure codes

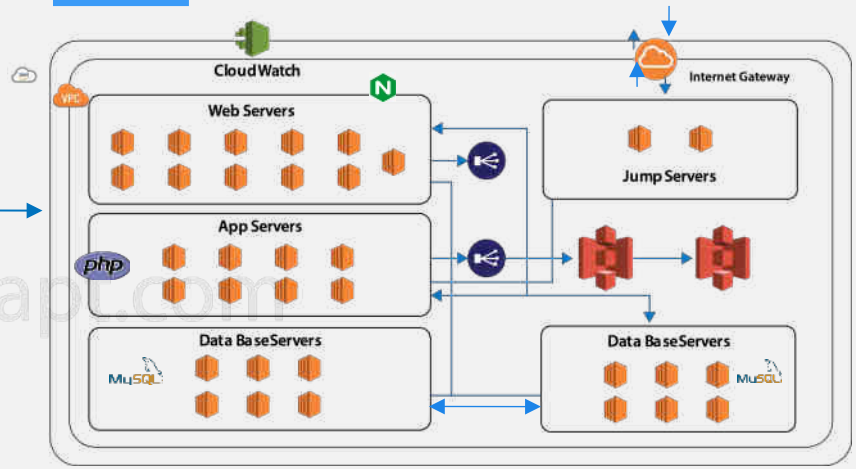
```
hosts: customerbs add
  become: True
  tasks:
    - name: stop BSLGI services
      service: name="{{ item }}"
               state=stopped
      with_items:
          - 'vrm-lgienhapi-bs'

    - name: install VRM Patches
      command: yum update -y --disablerepo=* --enablerepo=vrm
      args:
        warn: no

    - name: Reload systemd
      command: systemctl daemon-reload
```

Execution plan

```
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:

+ aws_autoscaling_attachment.asg_attachment_bar_app
      id:
      autoscaling_group_name:
      elb:

+ aws_autoscaling_attachment.asg_attachment_bar_api
      id:
      autoscaling_group_name:
      elb:
```

Input code contains the configuration of DSP video Backoffice applications such as Airflow, Seachange, Think analytics Recommendation engine (RENG), Nokia Video Recording Management (VRM), Axiros auto Configuration server

Output



*Sample representation

Defining applications in a codified form reduces the total time to market by 35-40%. The benefit is further compounded as the complexity in infrastructure increases.

Scaling policies as code: setting up the monitoring mechanism and defining the most appropriate scaling policies is critical for efficient auto-scale implementation

Autoscaling policies can be coded as configuration files in a cloud configuration orchestrator. This coded configuration files can be pushed into the respective cloud vendor's scaling engine.

Auto-scaling techniques

Predictive scaling

Predictive scaling analyzes each resource's historical workload and regularly forecasts the future load

Limitations

Predictive scaling cannot identify irregular spikes and during such situations, the infrastructure might face heavy loads and might breakdown.

Dynamic scaling

Dynamic scaling creates target tracking scaling policies for the resources in the scaling plan. These scaling policies adjust resource capacity in response to live changes in resource utilization.

Limitations

Dynamic scaling policy has a cooling period in between successive instance initiation and sudden loads during the cooling period might bring the application down.

Step scaling

Using step scaling policy the user can specify one or more adjustments that automatically scale the number of instances dynamically based on scale of the load increase.

Advantage

Step scaling policy can initiate successive instances even during the cooling period and can accommodate sudden spikes.

The agility of the infrastructure provisioned mainly depends on the strength of scaling policy defined. Step scaling policy is the most recommended technique to handle peak irregular loads.

Cloud configuration orchestrator leverages customizable configuration files to set complex step-scaling policies in an efficient way

The below configuration files in cloud configuration orchestrator aids in setting up a robust step scaling policy

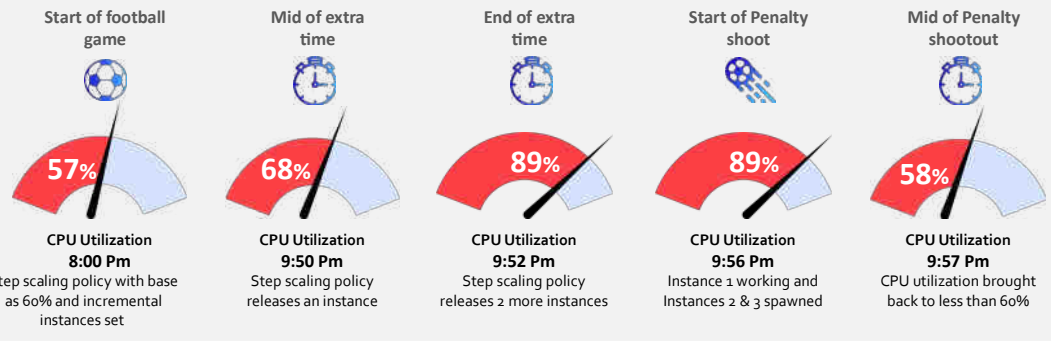
Instance launch configuration
This is a customizable configuration file that an auto-scaling group uses to launch instances.

Autoscaling group
Customizable configuration file to group instances. Grouping can be done based on the instance characteristics. This enables scaling policies to be applied at the group level.

Autoscaling policy
This contains the type of autoscaling policy to be carried out (Dynamic, Step or predictive) and the variables under each of the auto-scaling policy

w

An example of sudden load increase during finals of English premier league



Cloud configuration orchestrator

```
resource "aws_autoscaling_policy" "worker_node_scale_policy" {
  name = "worker_node_scale_out"
  adjustment_type = "PercentChangeInCapacity"
  policy_type = "StepScaling"
  autoscaling_group_name = aws_autoscaling_group.worker_nodes.name

  step_adjustment {
    scaling_adjustment = 20
    metric_interval_lower_bound = 0
  }

  step_adjustment {
    # AWS sees this as another +20 adjustment
    scaling_adjustment = -20
    metric_interval_upper_bound = 0
  }
}

resource "aws_cloudwatch_metric_alarm" "bpi_alarm_high" {
  alarm_name          = "BPIAlarmHighStep"
  comparison_operator = "GreaterThanOrEqualToThreshold"
  evaluation_periods  = "2"
  metric_name         = "BacklogPerInstance"
  namespace           = var.cloudwatch_namespace
  period              = "60"
  statistic           = "Average"
  threshold           = "1"

  dimensions = {
    Queue = aws_autoscaling_group.worker_nodes.name
  }

  alarm_description = "Monitors job backlog presence"
  alarm_actions     = [aws_autoscaling_policy.worker_node_scale_policy.arn]
}

resource "aws_cloudwatch_metric_alarm" "bpi_alarm_low" {
  alarm_name          = "BPIAlarmLowStep"
  comparison_operator = "LessThanThreshold"
  evaluation_periods  = "7"
  metric_name         = "BacklogPerInstance"
  namespace           = var.cloudwatch_namespace
  period              = "60"
  statistic           = "Average"
  threshold           = "1"

  dimensions = {
    Queue = aws_autoscaling_group.worker_nodes.name
  }

  alarm_description = "Monitors job backlog presence"
  alarm_actions     = [aws_autoscaling_policy.worker_node_scale_policy.arn]
}
```

Sample use case 3: A leading DSP in Europe setting up autoscaling policies leveraging cloud configuration orchestrator

Cloud configuration orchestrator

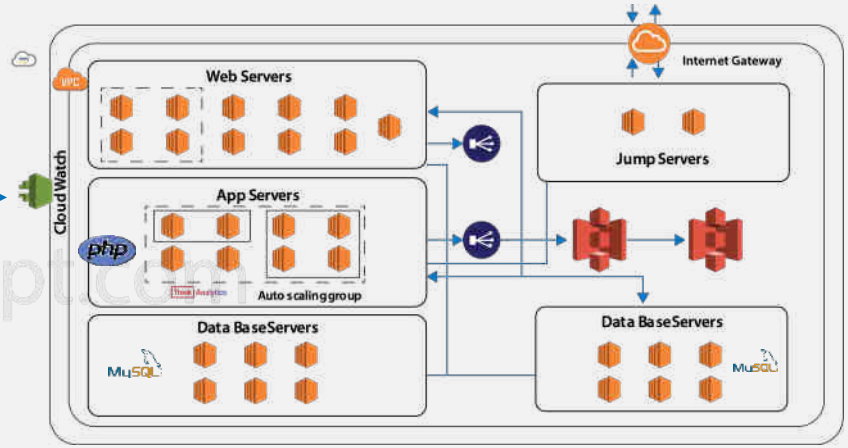
Scaling policies as code

```
resource "aws_launch_configuration" "moonshot-launchconfig-web" {
  name_prefix          = "moonshot-launchconfig-web"
  image_id             = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type        = "t2.micro"
  count                = "${var.count}"
  key_name             = "${aws_key_pair.mykey.key_name}"
  security_groups      = ["${aws_security_group.allow-ssh.id}"]
  #user_data = "${file("webmount.sh")}"
}

resource "aws_launch_configuration" "moonshot-launchconfig-app" {
  name_prefix          = "moonshot-launchconfig-app"
  image_id             = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type        = "t2.micro"
  count                = "${var.count}"
  key_name             = "${aws_key_pair.mykey.key_name}"
  security_groups      = ["${aws_security_group.allow-ssh.id}"]
  #user_data = "${data.template_file.appvm.rendered}"
  #user_data = "${file("appmount.sh")}"
}
```

Input Code contains codes for Scaling policy with parameters such as upper bound, lower bound limits, auto-scaling group name, scaling adjustment, AMI id, Instance type, key pair, security groups, health check grace period and scaling options

Execution plan

```
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:

+ aws_autoscaling_attachment.asg_attachment_bar_app
      id:
      autoscaling_group_name:
      elb:

+ aws_autoscaling_attachment.asg_attachment_bar_web
      id:
      autoscaling_group_name:
      elb:
```



*Sample representation

*The scaling policies from the orchestrator gets integrated with monitoring tool (Amazon cloud watch). It dynamically monitors the entire infrastructure and triggers the scaling mechanism on reaching the threshold.

Codification of scaling policy allows the management of infrastructure without involving the console. This reduces the complexity and dependency on cloud architects.

Estimated benefits for DSPs leveraging Infrastructure as Code (IaC)

Key Benefits

Implementing Infrastructure as Code with cloud configuration orchestrator provides the following benefits

Infrastructure as Code (IaC)

	Cloud manual provisioning	Leveraging cloud configuration orchestrator
Average time taken for Infrastructure provisioning	9 hrs	3 hrs
Configuration consistency	70%	97%
Average time taken for application provisioning	3-5 hrs	2 hrs

Time taken for provisioning the infrastructure reduces by **65%**

Total lead time for entire process reduces by **58%**

Configuration consistency improves by **27%**

Get in touch

USA

Prodapt North America, Inc.
Tualatin: 7565 SW Mohawk St.,
Phone: +1 503 636 3737

Dallas: 1333, Corporate Dr., Suite 101, Irving
Phone: +1 972 201 9009

New York: 1 Bridge Street, Irvington
Phone: +1 646 403 8161

CANADA

Prodapt Canada, Inc.
Vancouver: 777, Hornby Street,
Suite 600, BC V6Z 1S4
Phone: +1 503 210 0107

PANAMA

Prodapt Panama, Inc.
Panama Pacifico: Suite No 206, Building 3815
Phone: +1 503 636 3737

UK

Prodapt (UK) Limited
Reading: Davidson House,
The Forbury, RG1 3EU
Phone: +44 (0) 11 8900 1068

IRELAND

Prodapt Ireland Limited
Dublin: 31-36 Ormond Quay Upper
Phone: +44 (0) 11 8900 1068

EUROPE

Prodapt Solutions Europe &
Prodapt Consulting B.V.
Rijswijk: De Bruyn Kopsstraat 14
Phone: +31 (0) 70 4140722

Prodapt Germany GmbH
Münich: Brienner Straße, 80333
Phone: +31 (0) 70 4140722

Prodapt Switzerland GmbH
Zürich: Mühlebachstrasse 54,
8008 Zürich

Prodapt Austria GmbH
Vienna: Cityport 11, Simmeringer
Hauptstraße 24, 1110
Phone: +31 (0) 70 4140722

SOUTH AFRICA

Prodapt SA (Pty) Ltd.
Johannesburg: No. 3,
3rd Avenue, Rivonia
Phone: +27 (0) 11 259 4000

INDIA

Prodapt Solutions Pvt. Ltd.
Chennai: Prince Infocity II, OMR
Phone: +91 44 4903 3000

"Chennai One" SEZ, Thoraipakkam
Phone: +91 44 4230 2300

IIT Madras Research Park II,
3rd floor, Kanagam Road, Taramani
Phone: +91 44 4903 3020

Bangalore: "CareerNet Campus"
2nd floor, No. 53, Devarabisana Halli,
Phone: +91 80 4655 7008

THANK YOU!

Prodapt Delivering good Telecom