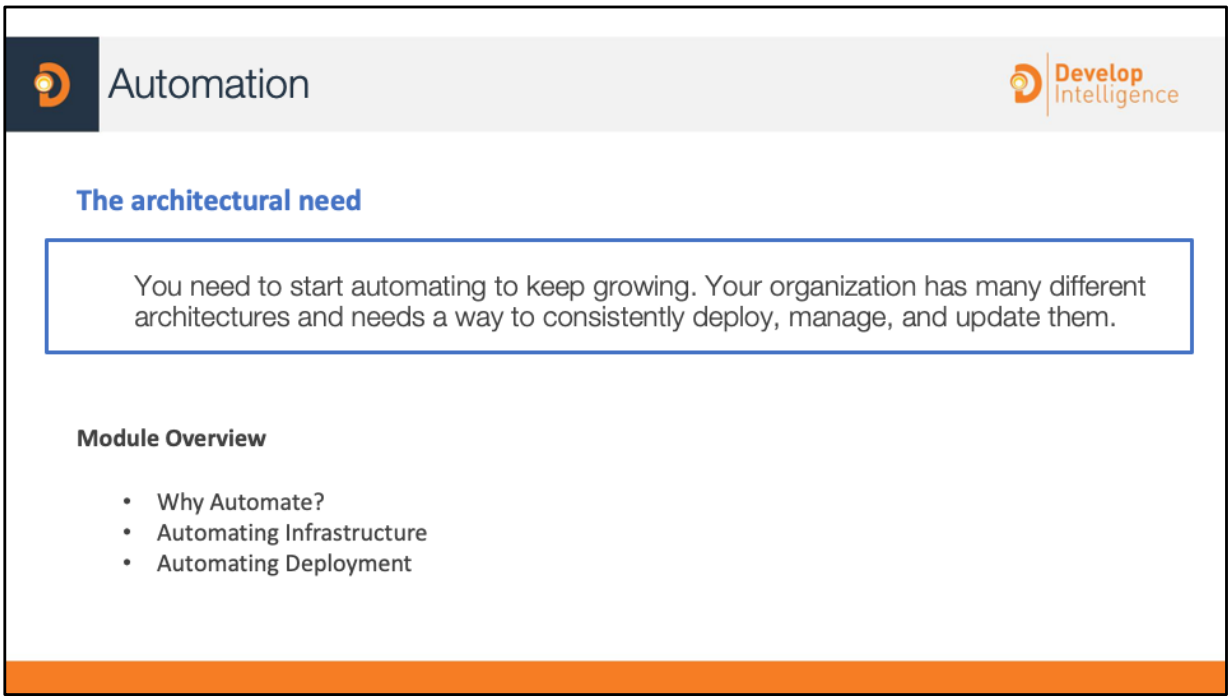
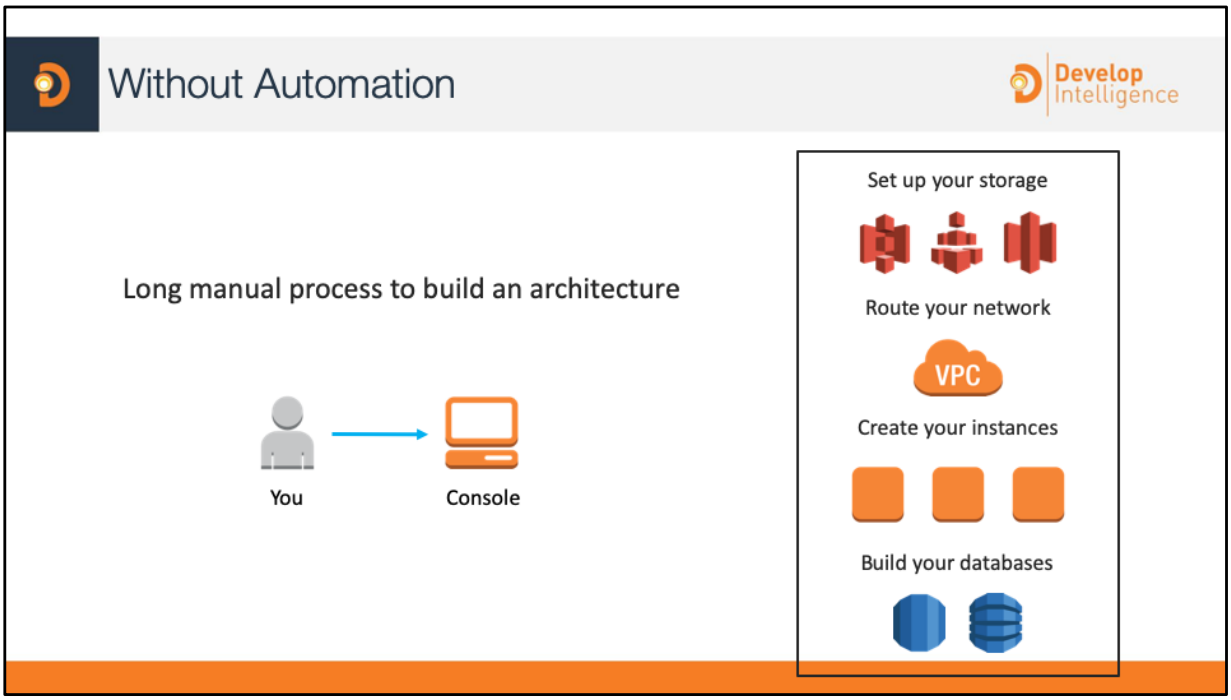
Automation

Develop Intelligence

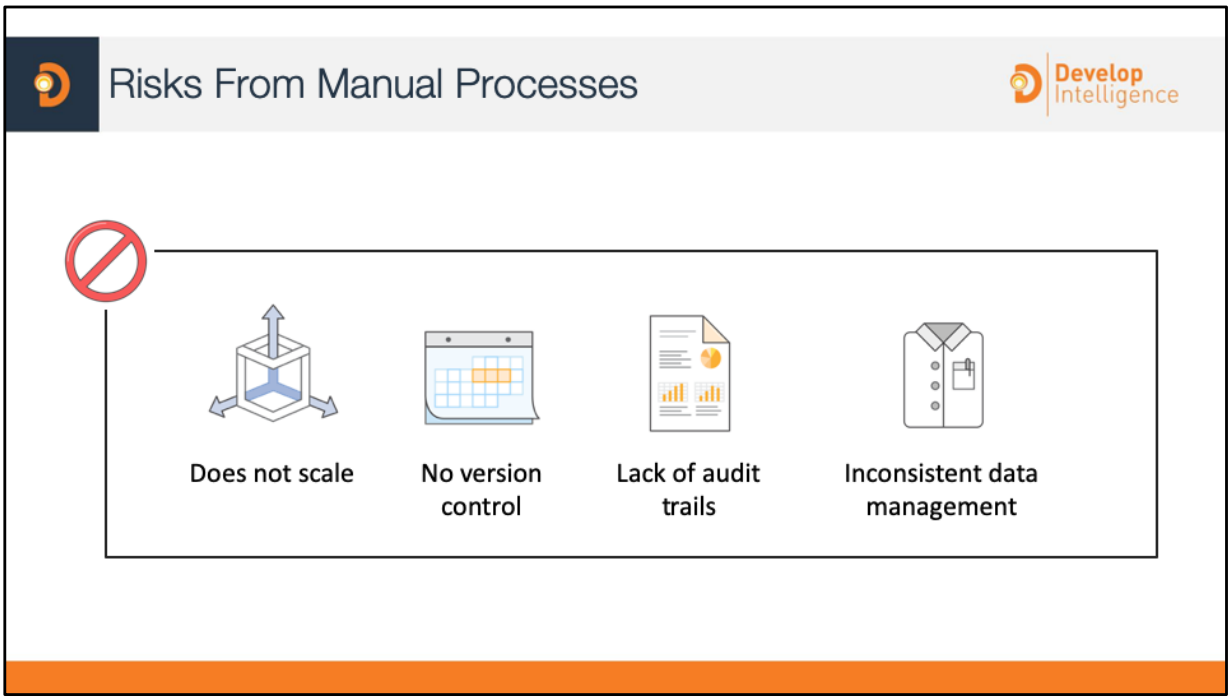
- Module 9


Automation
Develop Intelligence
The architectural need
You need to start automating to keep growing. Your organization has many different architectures and needs a way to consistently deploy, manage, and update them.
Module Overview
• Why Automate?
• Automating Infrastructure
• Automating Deployment



It takes significant time and energy to build a large-scale computing environment. Here are some questions to consider:

- Where do you want to put your efforts: the design or the implementation? And what are the risks of manual implementations?
- How do you update production servers? How are you going to roll out deployments across multiple geographic regions? When things break, and they will, how do you manage the rollback?
- What about debugging deployments? How do you find what's wrong and then fix it so that it says fixed?
- How will you manage dependencies on the various systems and subsystems in your organization?
- Finally, can you do all of this by hand?

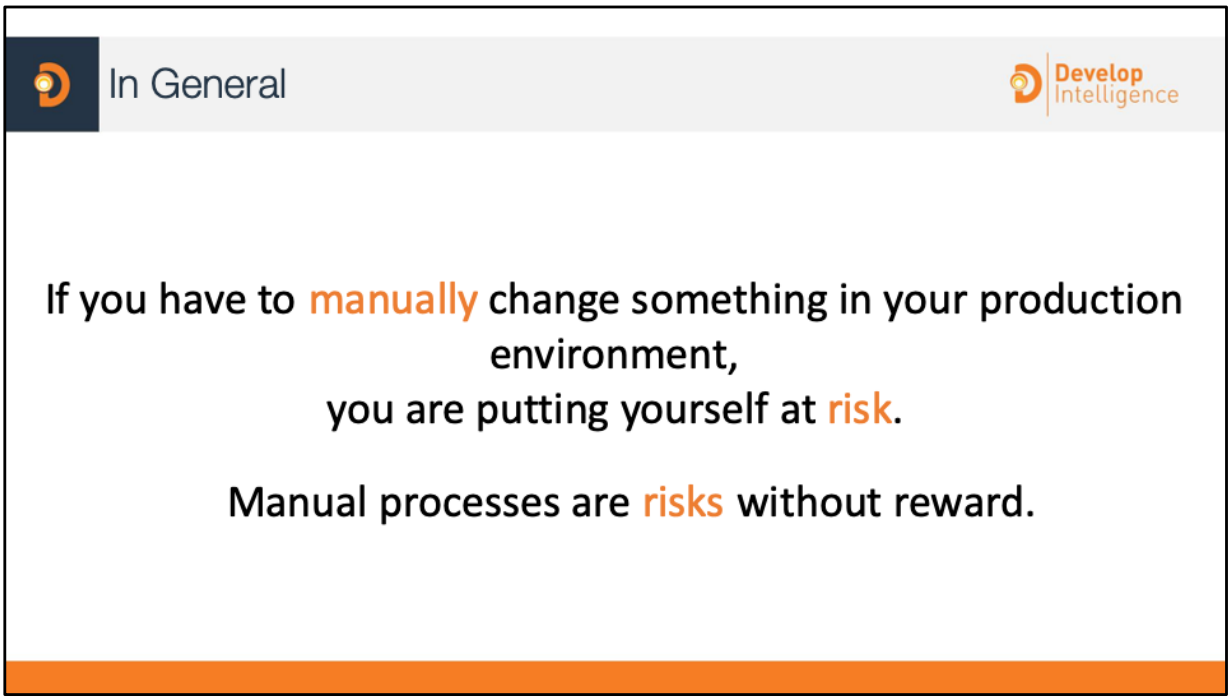


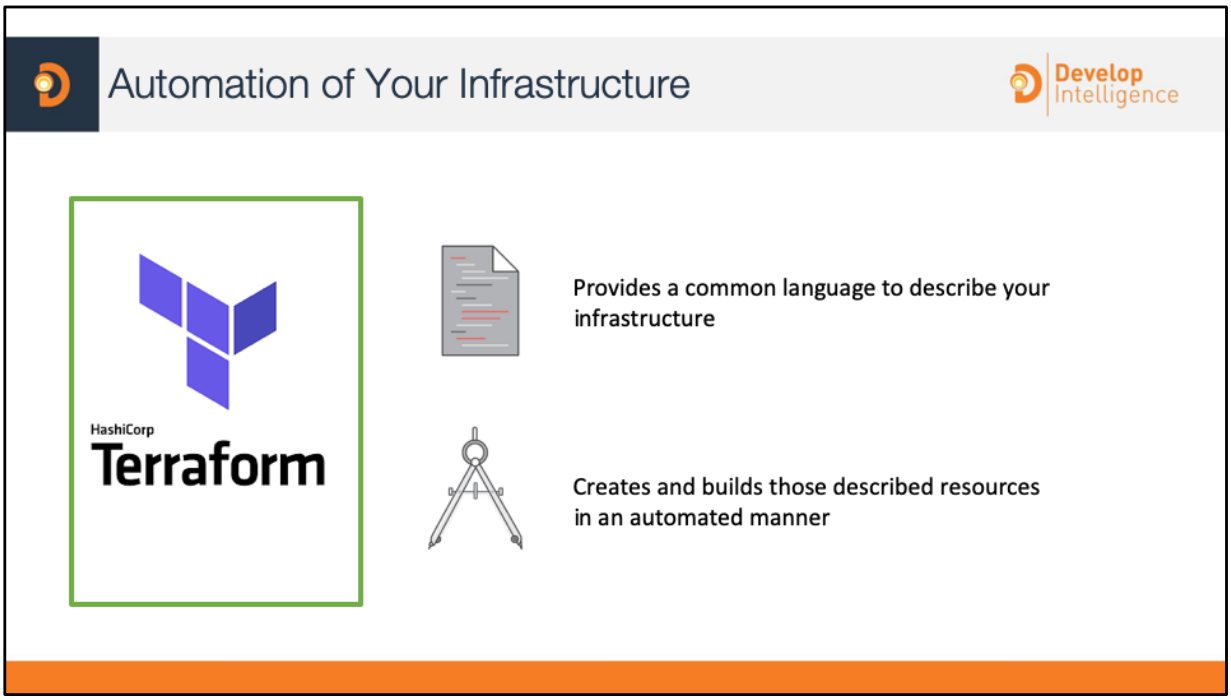
Having to manually create and add pieces to your environment does not scale. If you are responsible for a large corporate application, there are not enough people to manually sail the ship.

Creating architecture and applications by scratch doesn't have inherent version control. In case of emergency, it's helpful to roll back the production stack to a previous version—but that's not possible when you create your environment manually.

Having an audit trail is very important for many compliance and security situations. It's dangerous to allow people to manually control and edit your environment.

Finally, consistency is critical when minimizing risks. Automation allows you to maintain consistency.


In General
Develop Intelligence
If you have to manually change something in your production environment,
you are putting yourself at risk.
Manual processes are risks without reward.



AWS CloudFormation provisions resources in a safe, repeatable manner, allowing you to build and rebuild your infrastructure and applications without having to perform manual actions or write custom scripts.

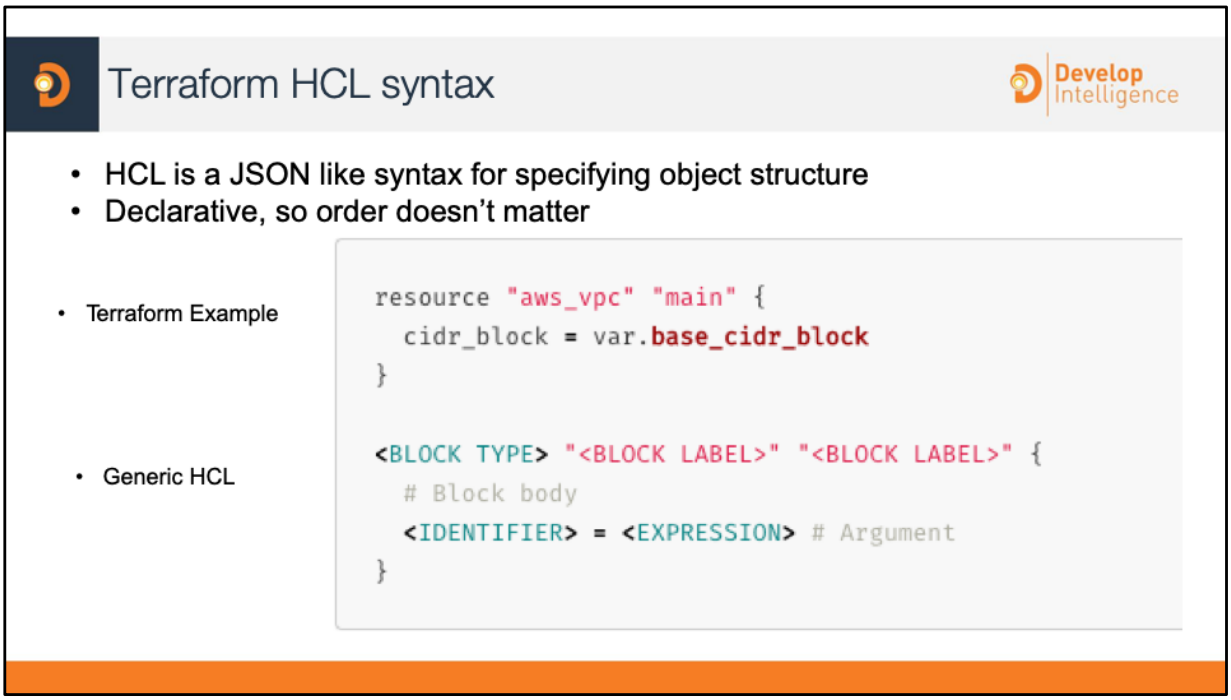
Using AWS CloudFormation allows you to treat your infrastructure as code. Author it with any code editor, check it into a version control system such as GitHub or AWS CodeCommit, and review files with team members before deploying into the appropriate environments.

What is Terraform?

Develop Intelligence

- Open Source Tool CLI tool for Infrastructure Automation (creation of resources)
- Can create any resources in any environment
- Utilizes plugin architecture for extensibility (Providers and Provisioners)
- Powered by API calls
- Uses declarative language for idempotent resource creation and modification
  - HCL (HashiCorp Configuration Language) or JSON
- Detects implicit dependencies between resources and dependency graph
- Builds in parallel where possible and sequentially for dependent resources

HashiCorp Terraform

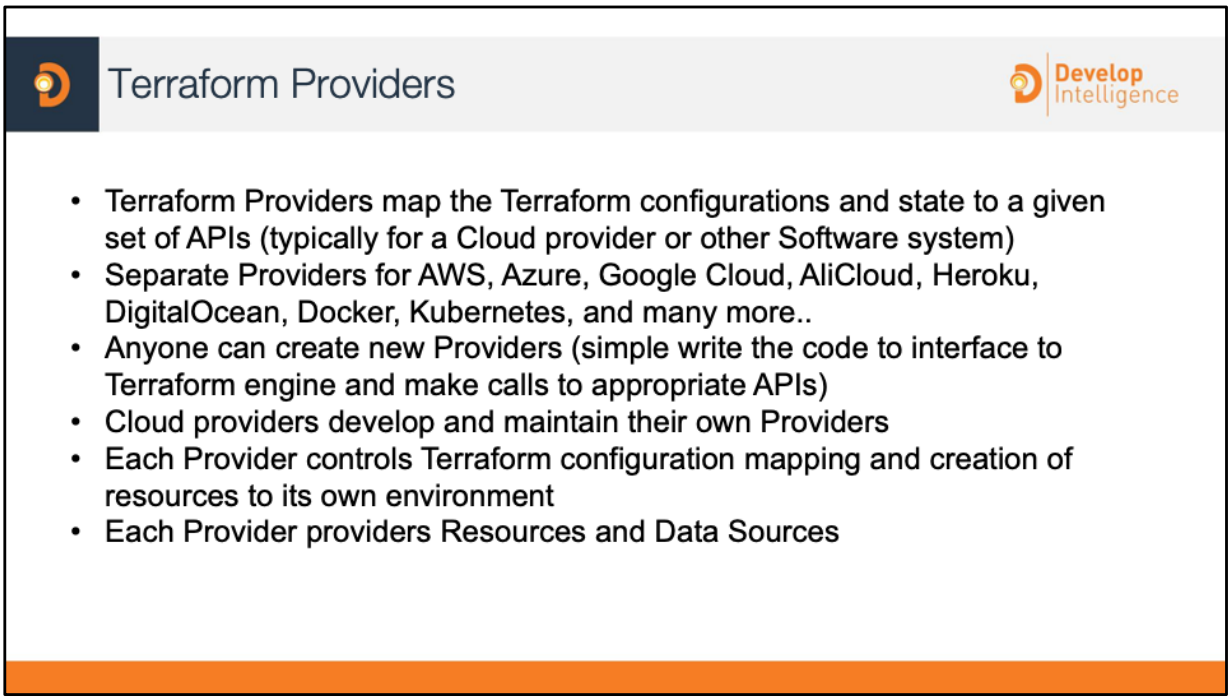


HCL Reference: https://github.com/hashicorp/hcl
Terraform Configuration Syntax:
https://www.terraform.io/docs/configuration/index.html
https://www.terraform.io/docs/configuration/syntax.html
Terraform JSON Configuration Syntax:
https://www.terraform.io/docs/configuration/syntax-json.html

Terraform Project

Develop Intelligence

- A folder that has been initialized with `terraform init`
- Collection of `*.tf` (HCL) or `*.tf.json` (JSON) configuration files in that folder
- Hidden `.terraform` sub-folder
- Optional local state file `*.tfstate` (JSON) generated by terraform when it creates resources from configuration files, also including attributes read from generated resources
  - State could be stored elsewhere instead (Backends)
- Optional variables files `terraform.tfvars` or `*.auto.tfvars`
  - Variables can be provided in command-line, as input during execution, or via files in project folder
- Modular architecture (Project is root module and can have child modules)



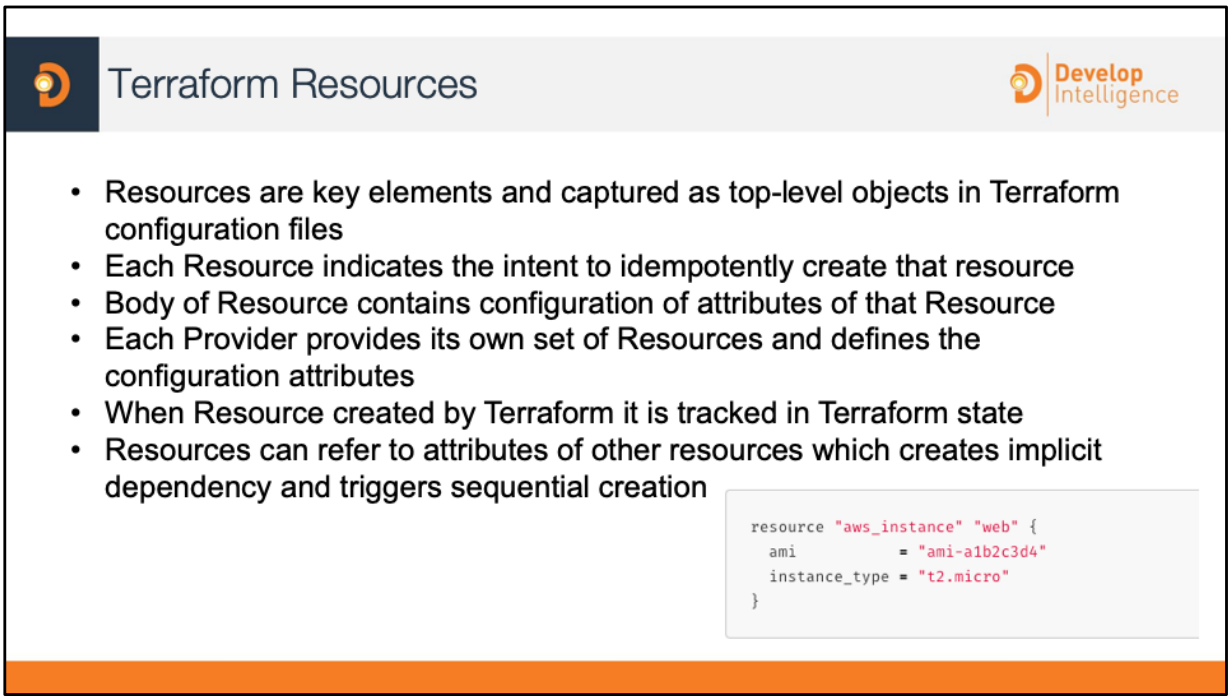
Terraform Providers: https://www.terraform.io/docs/providers/index.html
AWS Provider: https://www.terraform.io/docs/providers/aws/index.html

Terraform Providers

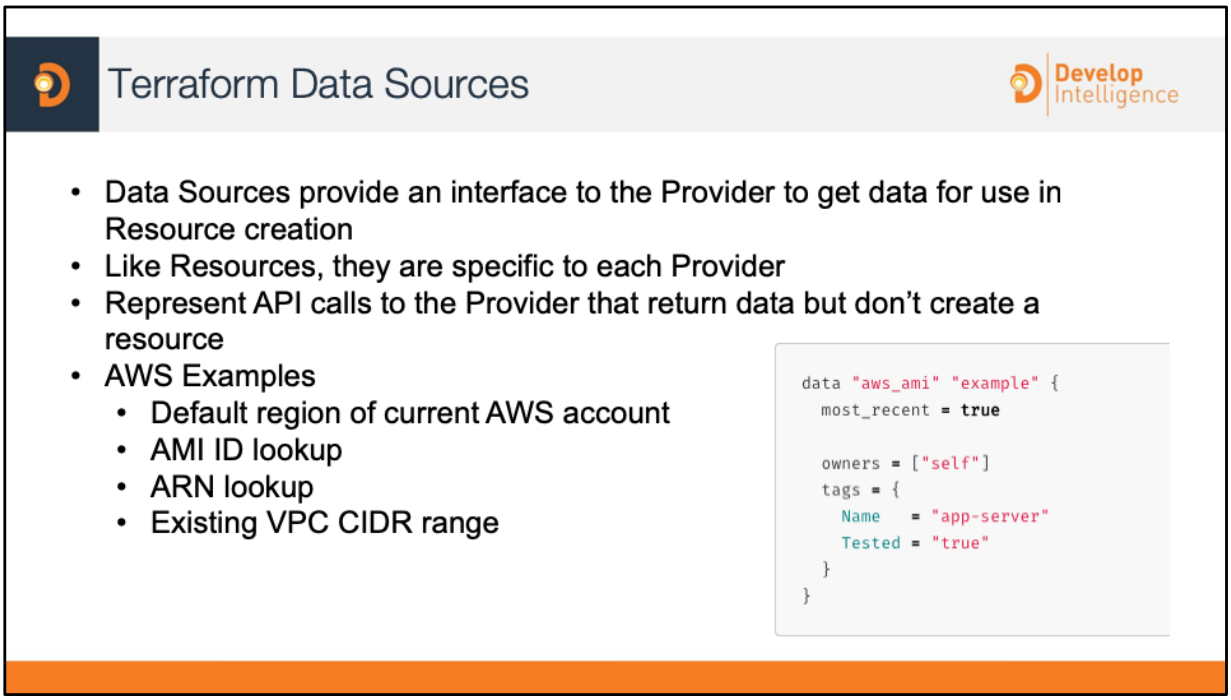
Develop Intelligence

- Providers can be configured
- If Provider configuration is missing but has Resources present it is implied with default configuration
- Multiple Providers can be configured and used in same Terraform configuration (if of same type should use the `alias` to distinguish)

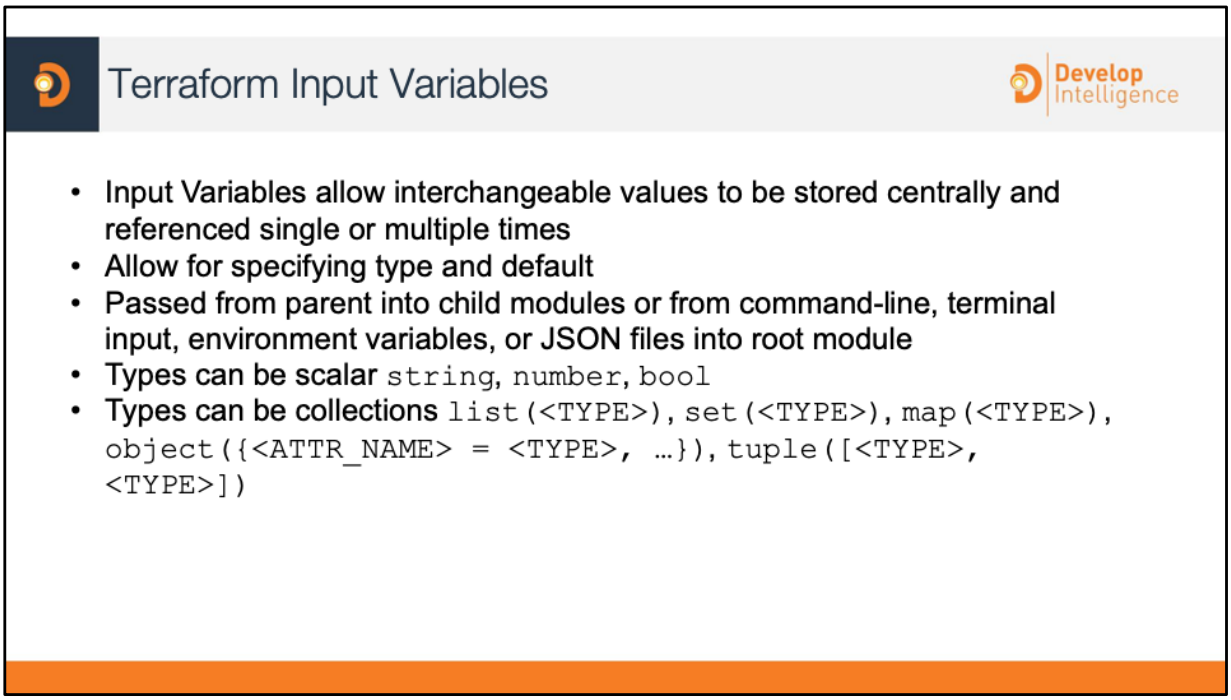
```
# The default provider configuration
provider "aws" {
  region = "us-east-1"
}

# Additional provider configuration for west coast region
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```



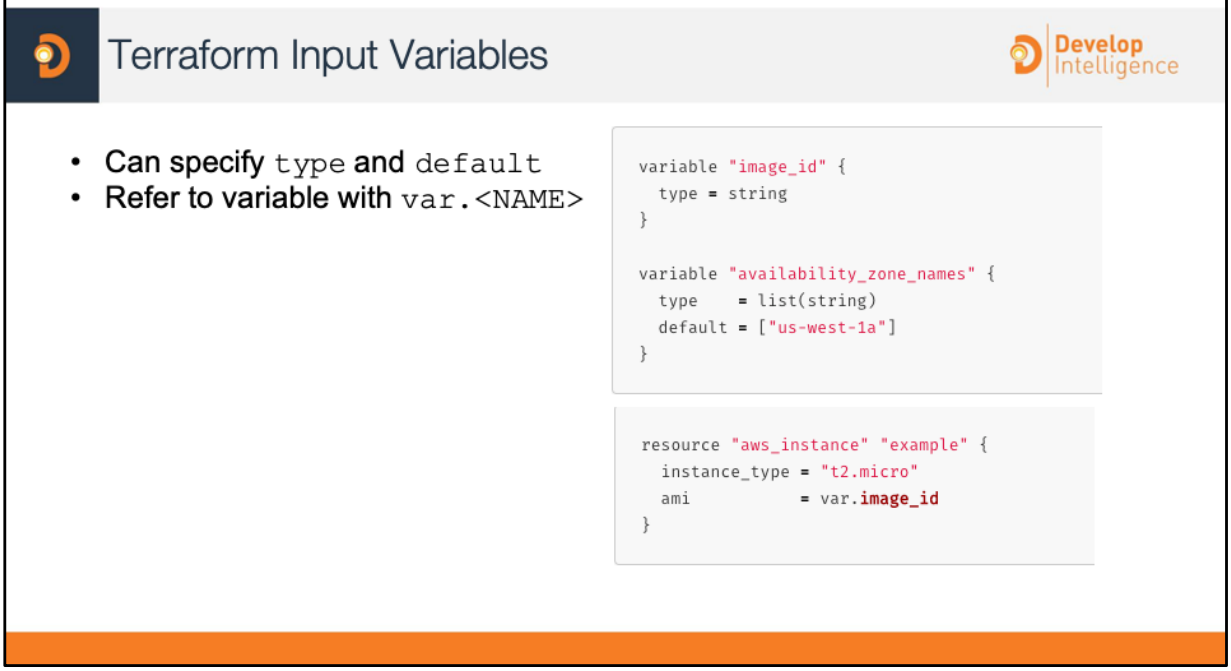
Resource Syntax reference:
https://www.terraform.io/docs/configuration/resources.html

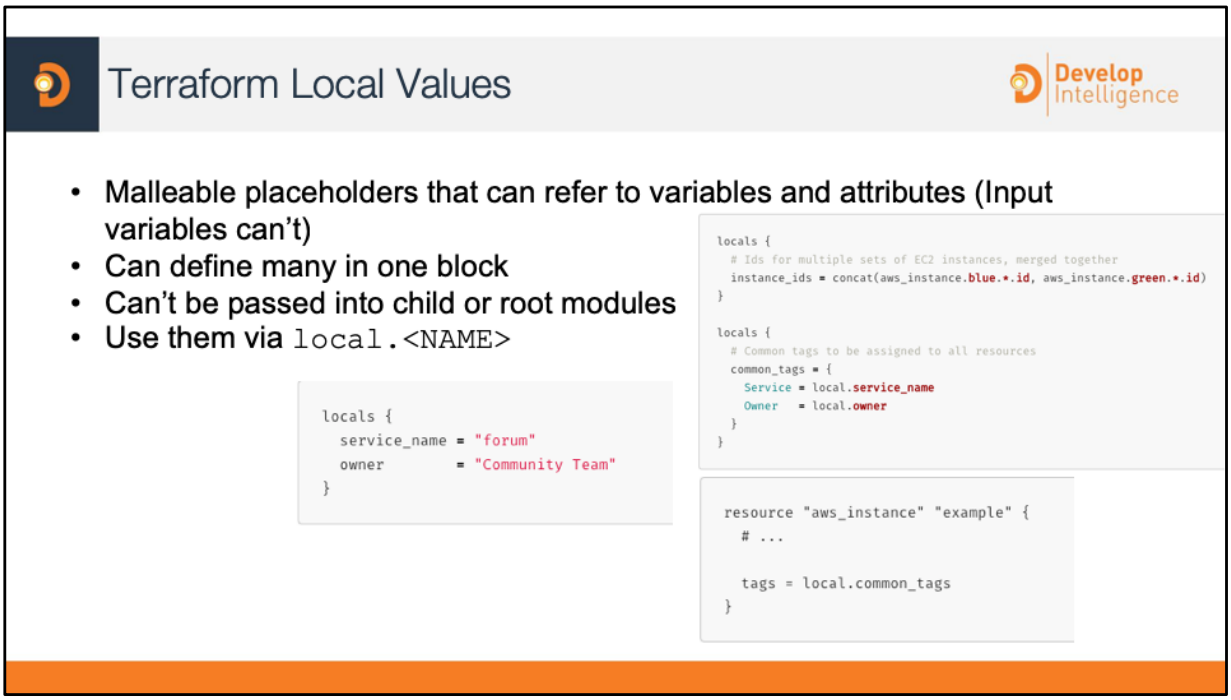


Data Source Syntax reference: https://www.terraform.io/docs/configuration/data-sources.html

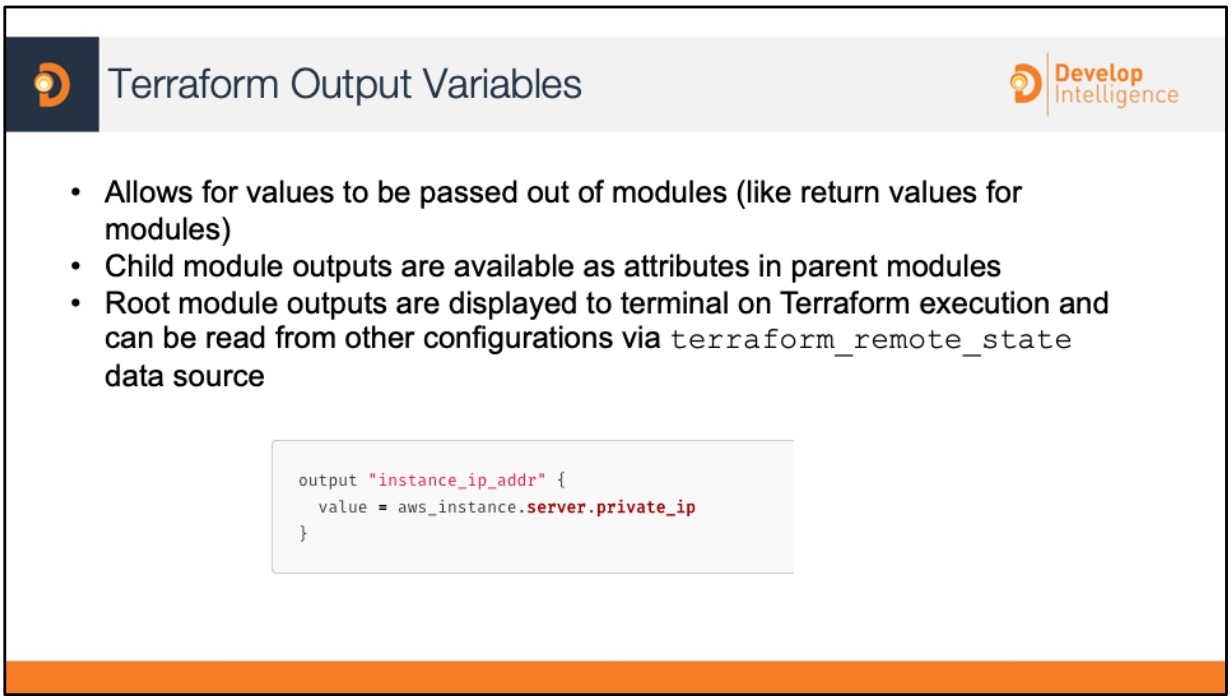


Input Variables Syntax reference:
https://www.terraform.io/docs/configuration/variables.html
Input Variable Types: https://www.terraform.io/docs/configuration/types.html

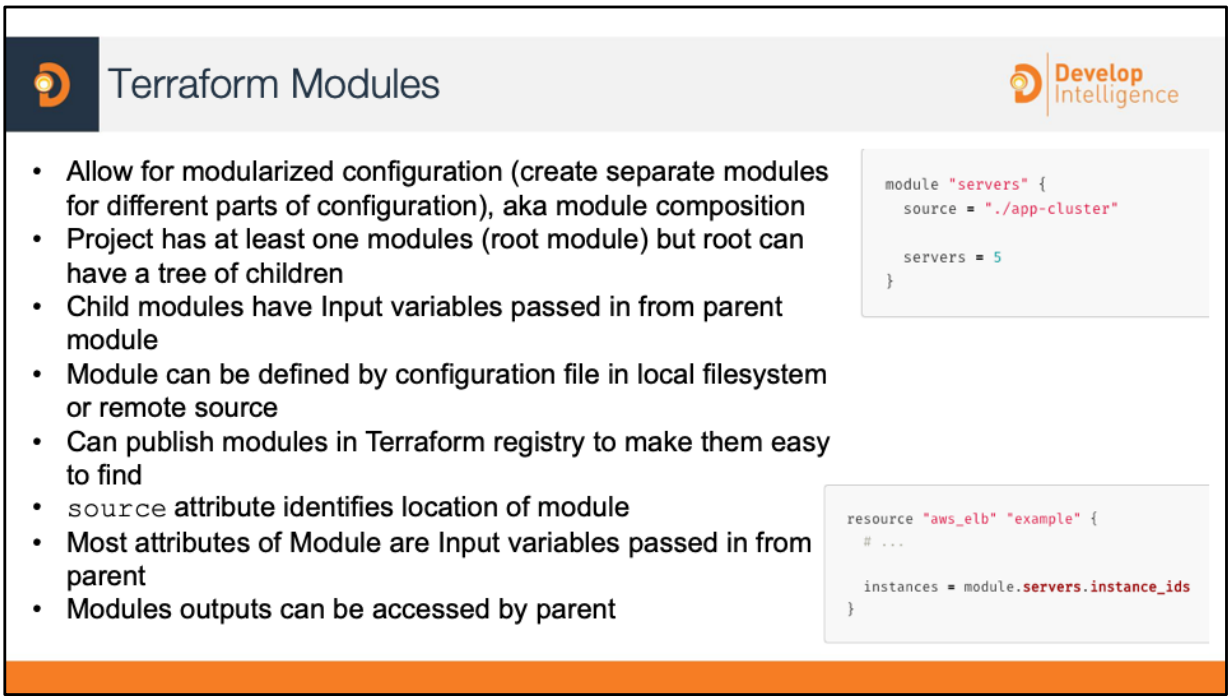

Terraform Input Variables
Develop Intelligence
• Can specify type and default
• Refer to variable with var.<NAME>
variable "image_id" {
type = string
}
variable "availability_zone_names" {
type = list(string)
default = ["us-west-1a"]
}
resource "aws_instance" "example" {
instance_type = "t2.micro"
ami = var.image_id
}



Local Values Syntax Reference:
https://www.terraform.io/docs/configuration/locals.html



Output Variables Syntax Reference:
https://www.terraform.io/docs/configuration/outputs.html
terraform_remote_state Reference:
https://www.terraform.io/docs/providers/terraform/d/remote_state.html



Modules Syntax Reference:
https://www.terraform.io/docs/configuration/modules.html
Modules Reference: https://www.terraform.io/docs/modules/index.html

Terraform Modules

Develop Intelligence

- Modules inherit parent providers implicitly but can also be passed explicitly
- Can have multiple instances of same module (should use `name`)
- Can have and specify versions

```
# The default "aws" configuration is used for AWS resources in the root
# module where no explicit provider instance is selected.
provider "aws" {
  region = "us-west-1"
}

# A non-default, or "aliased" configuration is also defined for a different
# region.
provider "aws" {
  alias  = "usw2"
  region = "us-west-2"
}

# An example child module is instantiated with the _aliased_ configuration,
# so any AWS resources it defines will use the us-west-2 region.
module "example" {
  source    = "./example"
  providers = {
    aws = "aws.usw2"
  }
}
```

```
provider "aws" {
  alias  = "usw1"
  region = "us-west-1"
}

provider "aws" {
  alias  = "usw2"
  region = "us-west-2"
}

module "tunnel" {
  source    = "./tunnel"
  providers = {
    aws.src = "aws.usw1"
    aws.dst = "aws.usw2"
  }
}
```

```
module "assets_bucket" {
  source = "./publish_bucket"
  name   = "assets"
}

module "media_bucket" {
  source = "./publish_bucket"
  name   = "media"
}
```

```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"

  servers = 3
}
```

Terraform Modules

Develop Intelligence

- Modules can be included from different `source` types
  - Local paths, Terraform registry, Github, Bitbucket, HTTP URLs, S3 buckets, etc..

```
module "consul" {
  source = "./consul"
}
```

```
module "consul" {
  source = "github.com/hashicorp/example"
}
```

```
module "vpc" {
  source = "https://example.com/vpc-module.zip"
}
```

```
module "consul" {
  source = "hashicorp/consul/aws"
  version = "0.1.0"
}
```

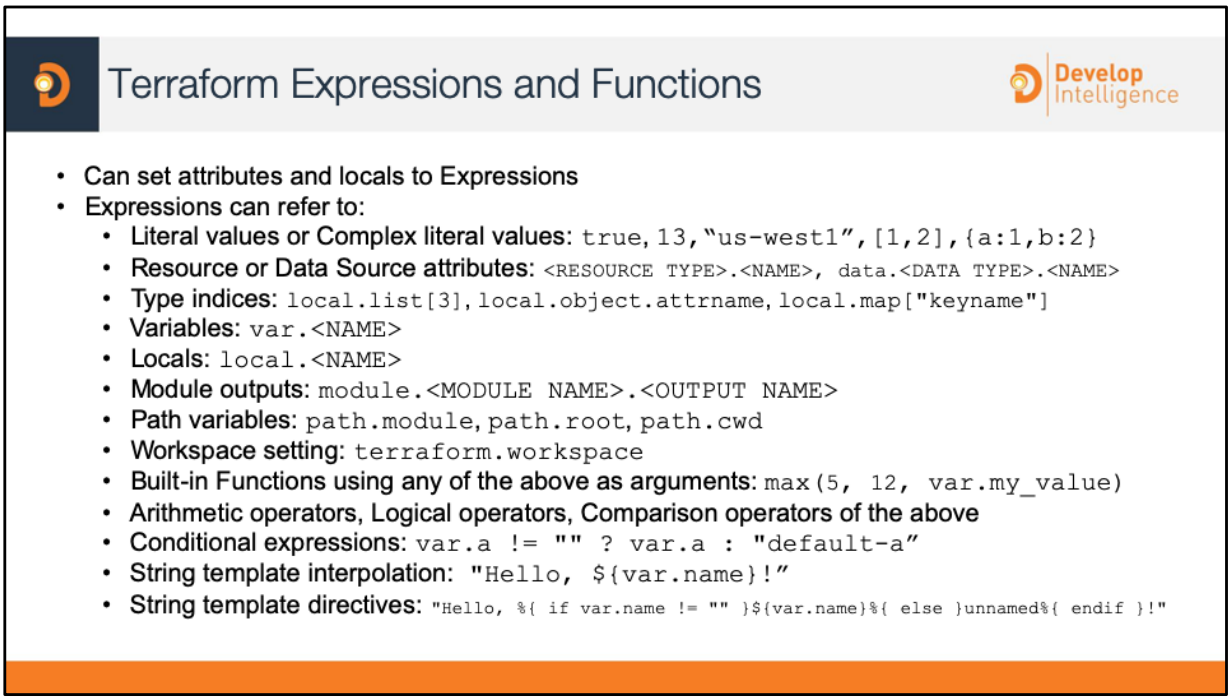
```
module "consul" {
  source = "git@github.com:hashicorp/example.git"
}
```

```
module "vpc" {
  source = "git::https://example.com/vpc.git"
}

module "storage" {
  source = "git::ssh://username@example.com/storage.git"
}
```

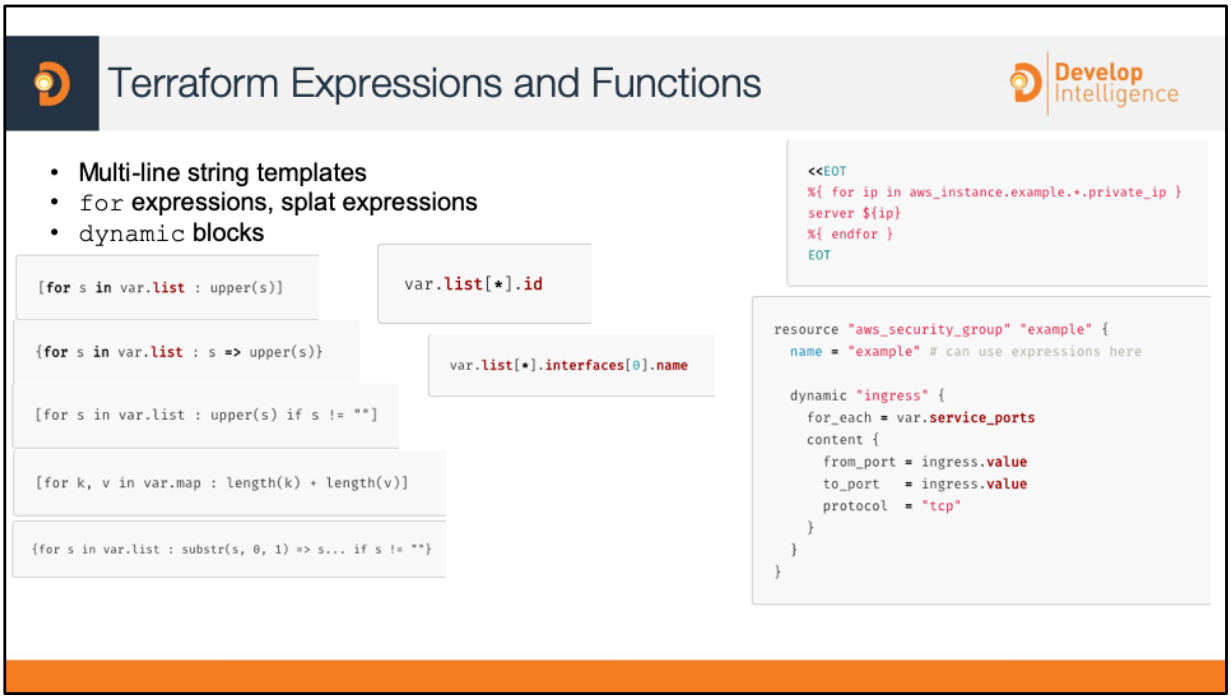
```
module "consul" {
  source = "app.terraform.io/example-corp/k8s-cluster/azurerm"
  version = "1.1.0"
}
```

```
module "consul" {
  source = "s3::https://s3-eu-west-1.amazonaws.com/examplecorp-terraform-modules/vpc.zip"
}
```


Terraform Module Composition
Develop Intelligence
module "network" {
source = "./modules/aws-network"
base_cidr_block = "10.0.0.0/8"
}
module "consul_cluster" {
source = "./modules/aws-consul-cluster"
vpc_id = module.network.vpc_id
subnet_ids = module.network.subnet_ids
}
data "aws_vpc" "main" {
tags {
Environment = "production"
}
}
data "aws_subnet_ids" "main" {
vpc_id = data.aws_vpc.main.id
}
module "consul_cluster" {
source = "./modules/aws-consul-cluster"
vpc_id = data.aws_vpc.main.id
subnet_ids = data.aws_subnet_ids.main.ids
}



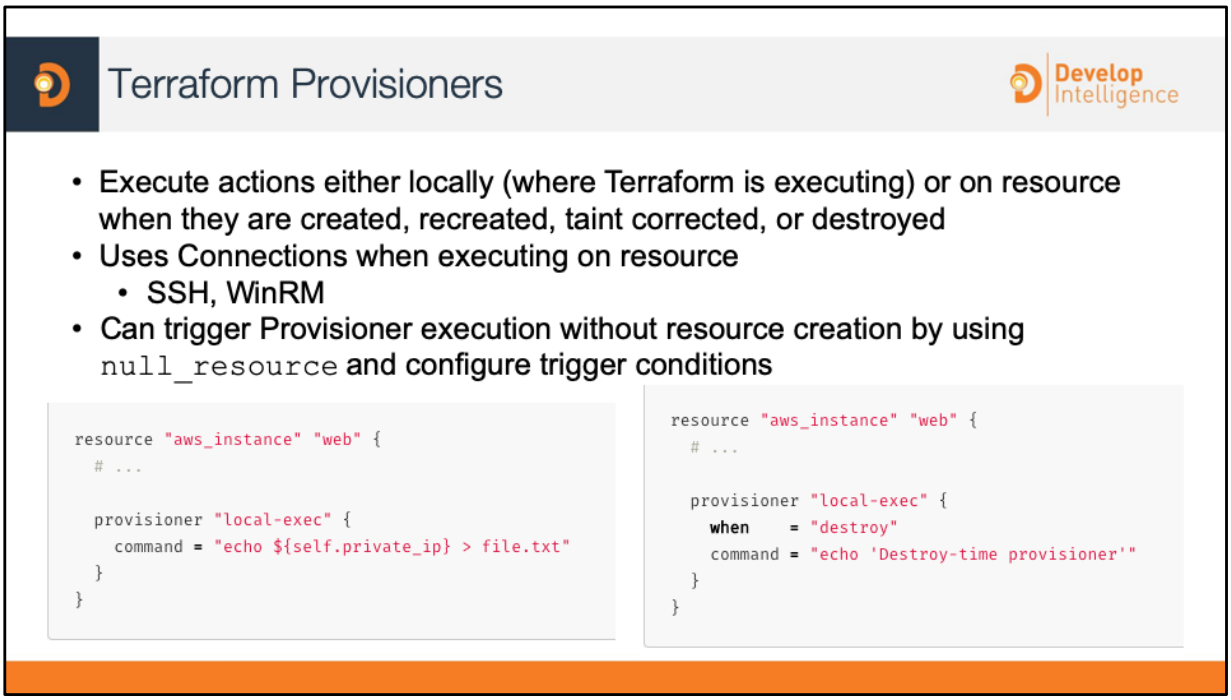
Terraform Expressions Syntax Reference:
https://www.terraform.io/docs/configuration/expressions.html
Terraform Functions Syntax Reference:
https://www.terraform.io/docs/configuration/functions.html


Terraform Expressions and Functions
Develop Intelligence
• Multi-line string templates
• for expressions, splat expressions
• dynamic blocks
[for s in var.list : upper(s)]
{for s in var.list : s => upper(s)}
[for s in var.list : upper(s) if s != ""]
[for k, v in var.map : length(k) + length(v)]
{for s in var.list : substr(s, 0, 1) => s... if s != ""}
var.list[*].id
var.list[*].interfaces[0].name
<<EOT
%{ for ip in aws_instance.example.*.private_ip }
server ${ip}
%{ endfor }
EOT
resource "aws_security_group" "example" {
name = "example" # can use expressions here
dynamic "ingress" {
for_each = var.service_ports
content {
from_port = ingress.value
to_port = ingress.value
protocol = "tcp"
}
}
}

Terraform Meta-attributes

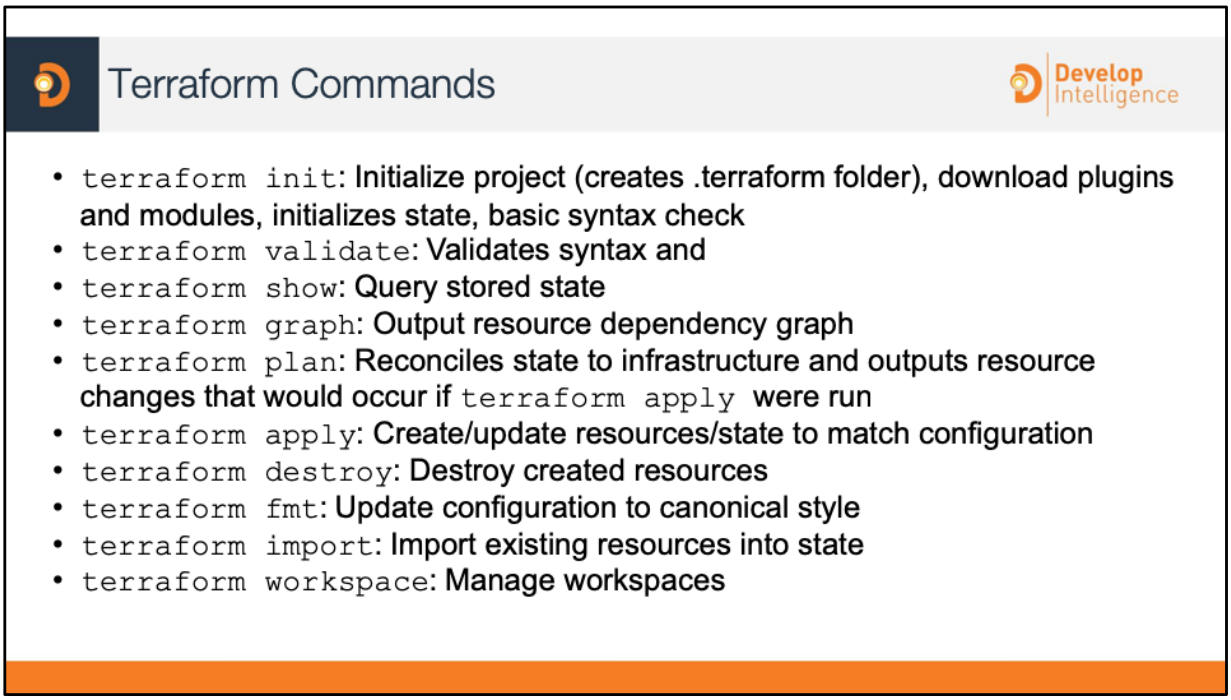
Develop Intelligence

- Many Terraform config objects also have meta-attributes which are available regardless of provider
- Resource and Data Source have same set
  - `depends_on`: explicit dependencies
  - `count`: create multiple copies of resource or conditionally create (0 or 1)
  - `provider`: select non-default provider config
  - `lifecycle`: lifecycle customizations
  - `provisioner` and `connection`: extra actions after resource creation



Terraform Provisioners: https://www.terraform.io/docs/provisioners/index.html
Null resource Provisioners:
https://www.terraform.io/docs/provisioners/null_resource.html


Terraform Provisioners
Develop Intelligence
# Copies the file as the root user using SSH
provisioner "file" {
source = "conf/myapp.conf"
destination = "/etc/myapp.conf"
connection {
type = "ssh"
user = "root"
password = "${var.root_password}"
}
}
resource "aws_instance" "cluster" {
count = 3
# ...
}
resource "null_resource" "cluster" {
# Changes to any instance of the cluster requires re-provisioning
triggers = {
cluster_instance_ids = "${join(",", aws_instance.cluster.*.id)}"
}
# Bootstrap script can run on any instance of the cluster
# So we just choose the first in this case
connection {
host = "${element(aws_instance.cluster.*.public_ip, 0)}"
}
provisioner "remote-exec" {
# Bootstrap script called with private_ip of each node in the cluster
inline = [
"bootstrap-cluster.sh ${join(" ", aws_instance.cluster.*.private_ip)}",
]
}
}

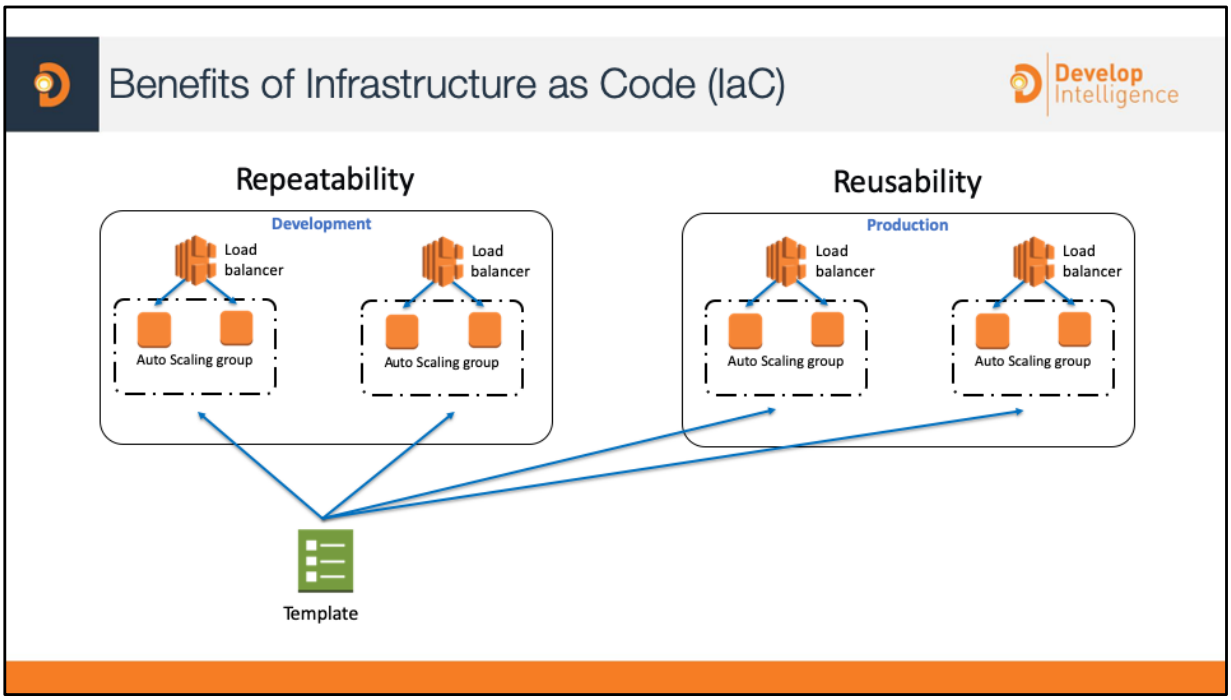


Terraform command reference:
https://www.terraform.io/docs/commands/index.html

Terraform State and Backends

Develop Intelligence

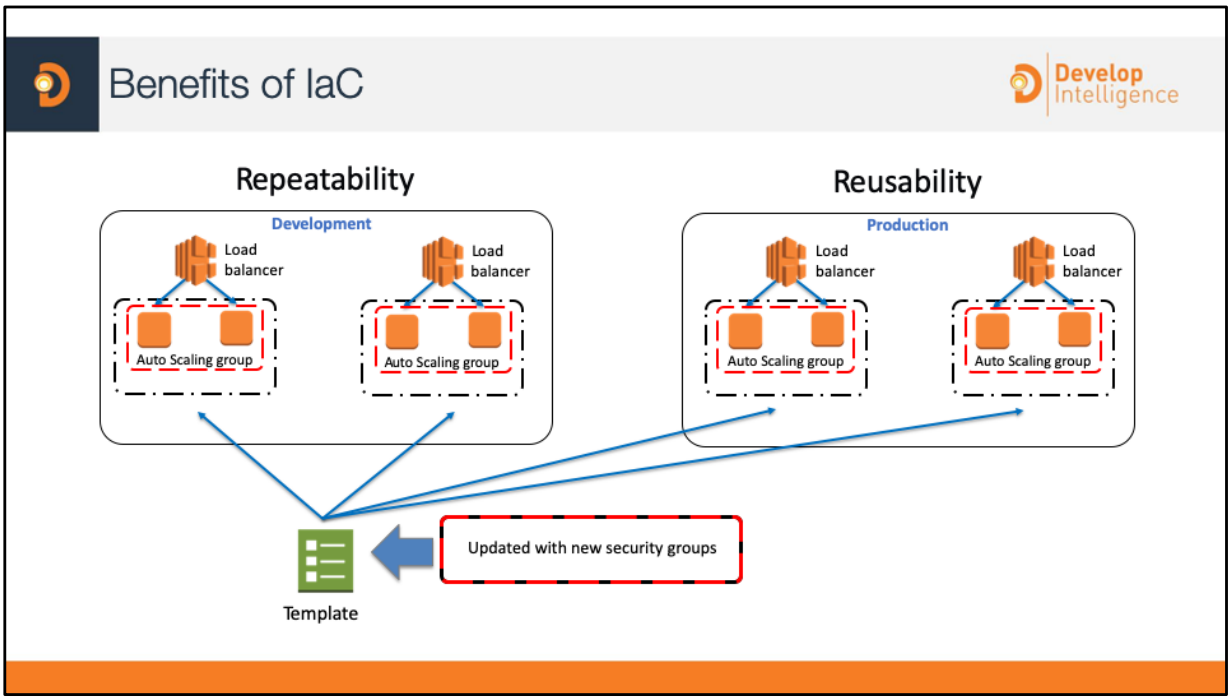
- Terraform stores state information about created resources, by default it is stored locatlly in `.tfstate` file but can be configured to be stored remotely in Backend
- Backend can be S3 bucket, Terraform Enterprise, Consul, others..
- Can query state via `terraform show`, or `terraform_remote_state` Data Source
- Can optionally have multiple Workspaces (each with own state), though all Workspaces for a given configuration are stored in same Backend
- When working in teams should use remote backend with access control (also provides locking)



If you build infrastructure with code, you gain the benefits of repeatability and reusability while building your environments.

With one template (or a combination of templates), you can build the same complex environments over and over again.

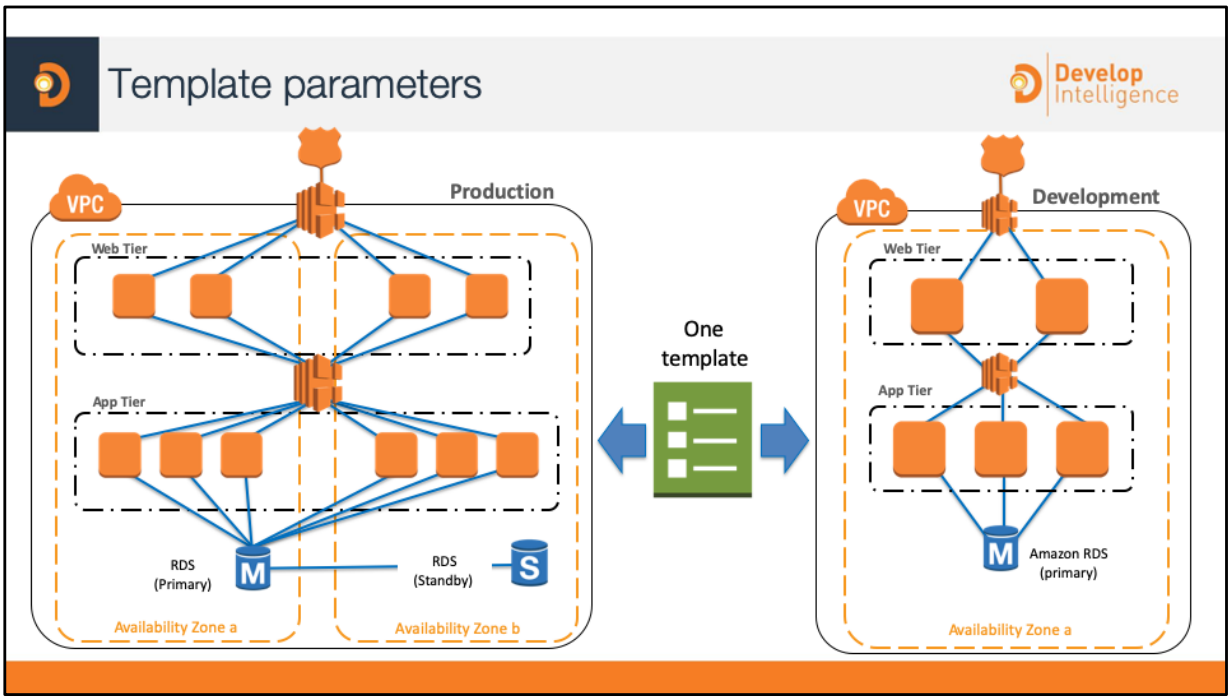
When doing this with AWS, you can even create environments dependent upon conditions, so that what ends up being built is specific to the context in which you've created it. For instance, a template can be designed so that different AMIs are used based on whether or not this template was launched into the development or the production environments.



In this scenario, the template has been updated to add new security groups to the instance stacks.

With one change to the template used to launch these environments, all four environments can have the new security group resource added.

This feature provides the benefit of easier maintainability of resources, as well as greater consistency and a reduction in effort through parallelization.

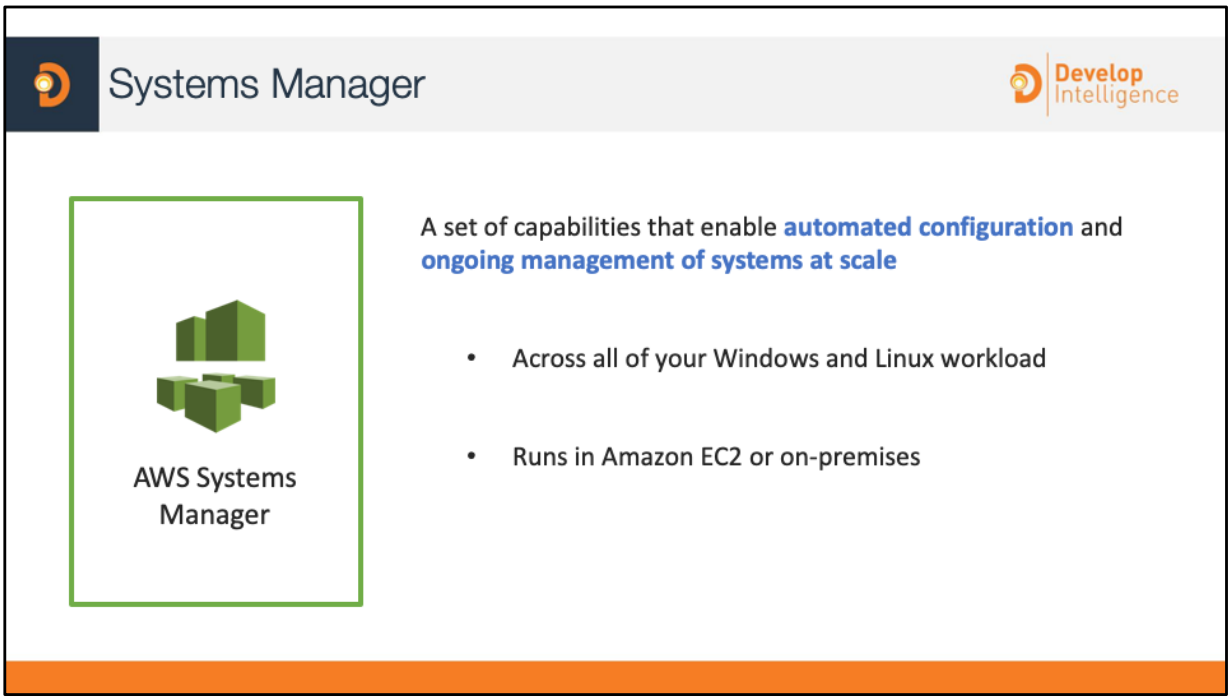


Your production environment and development environment should be built from the same template. This ensures that your application works in production the way it was designed and developed.

Additionally, your development environment and testing environment must use the same template. All environments will have identical applications and configurations.

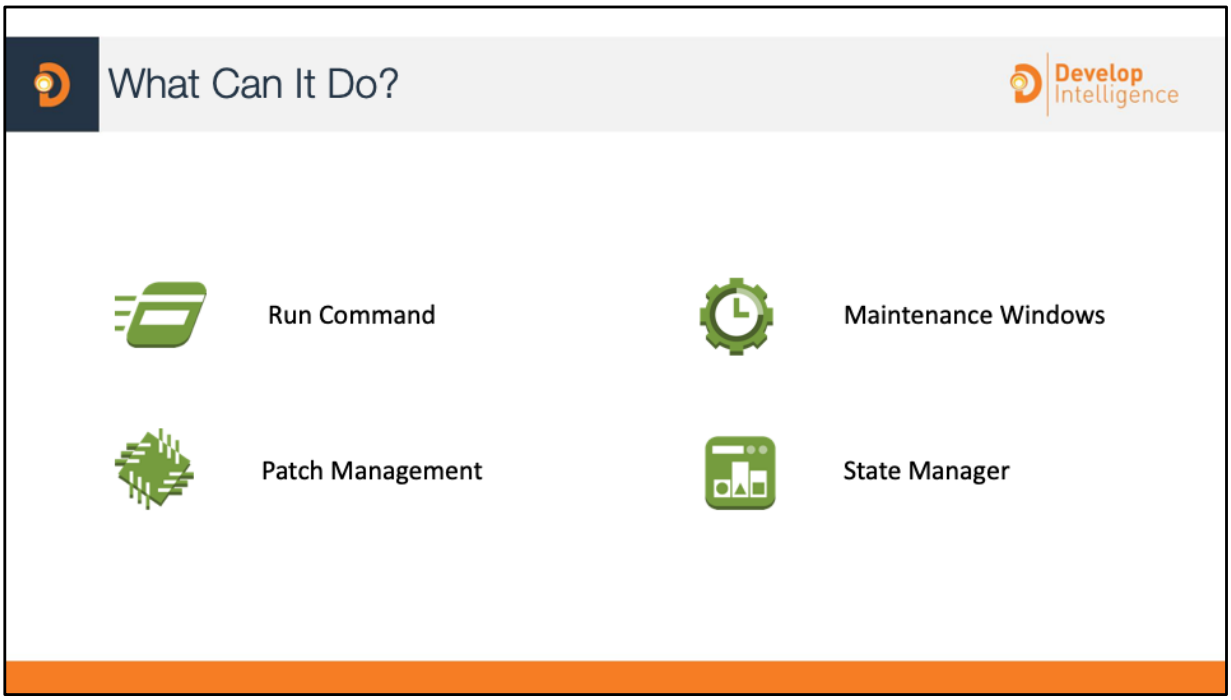
You might need several testing environments for functional testing, user acceptance testing, and load testing. Creating those environments manually comes with great risk.

You can use Variables and Inputs to allow the same template to create the minor differences in size, scope, and configuration for your production vs test vs development systems



AWS Systems Manager is a management service that helps you automatically collect software inventory, apply OS patches, create system images, and configure Windows and Linux operating systems. These capabilities help you define and track system configurations, prevent drift, and maintain software compliance of your Amazon EC2 and on-premises configurations. By providing a management approach that is designed for the scale and agility of the cloud but extends into your on-premises data center, AWS Systems Manager makes it easier for you to seamlessly bridge your existing infrastructure with AWS.

You can open AWS Systems Manager from the Amazon EC2 console. Select the instances you want to manage, and define the management tasks you want to perform. AWS Systems Manager is available at no cost to manage both your Amazon EC2 and on-premises resources.



Use Systems Manager Run Command to remotely and securely manage the configuration of your managed instances at scale. Use Run Command to perform on-demand changes like updating applications or running Linux shell scripts and Windows PowerShell commands on a target set of dozens or hundreds of instances.

Use Patch Manager to automate the process of patching your managed instances. This capability enables you to scan instances for missing patches and apply missing patches individually or to large groups of instances by using Amazon EC2 instance tags. For security patches, Patch Manager uses patch baselines that include rules for auto-approving patches within days of their release, as well as a list of approved and rejected patches. Security patches are installed from the default repository for patches configured for the instance. You can install security patches on a regular basis by scheduling patching to run as a Systems Manager Maintenance Window task. For Linux operating systems, you can define the repositories that should be used for patching operations as part of your patch baseline. This allows you to ensure that updates are installed only from trusted repositories regardless of what repositories are configured on the instance. For Linux, you also have the ability to update any package on the instance, not just those that are classified as operating system security updates.

Use Maintenance Windows to set up recurring schedules for managed instances to execute administrative tasks like installing patches and updates without interrupting business-critical operations.

Use Systems Manager State Manager to automate the process of keeping your managed instances in a defined state. You can use State Manager to ensure that your instances are bootstrapped with specific software at startup, joined to a Windows domain (Windows instances only), or patched with specific software updates.

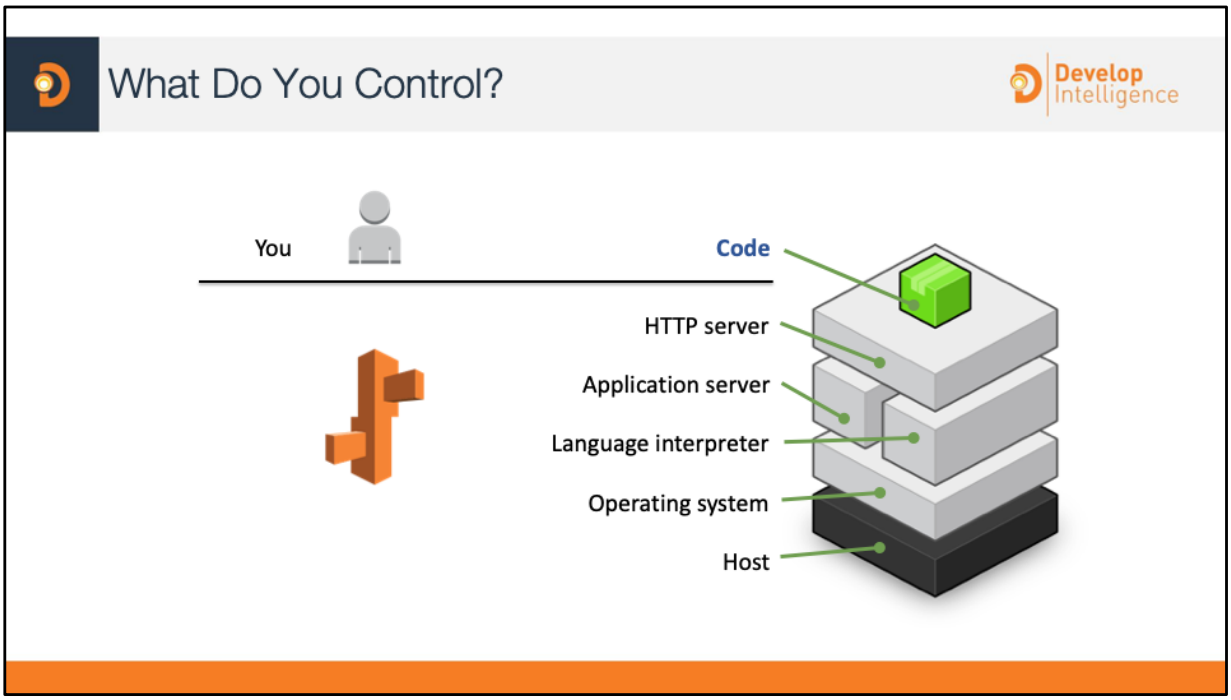
AWS Elastic Beanstalk

Develop Intelligence

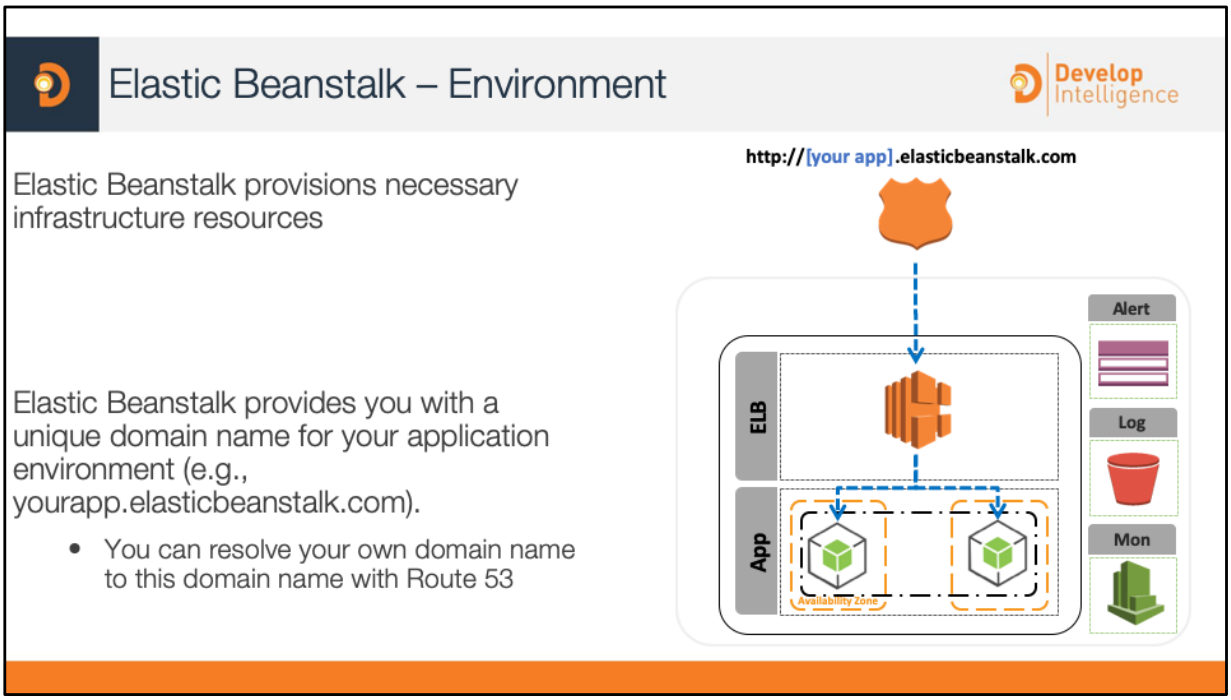
Provisions and operates the infrastructure and **manages the application stack for you**

Completely transparent—you can see everything that is created

Impossible to outgrow; **automatically scales your application** up and down



The goal of AWS Elastic Beanstalk is to help developers to deploy and maintain scalable web applications and services in the cloud without having to worry about the underlying infrastructure. Elastic Beanstalk configures each EC2 instance in your environment with the components necessary to run applications for the selected platform. You don't have to worry about logging into instances to install and configure your application stack.



There are two types of environments you can choose from when working with AWS Elastic Beanstalk. The single-instance environment will allow you to launch a single EC2 instance and will not include load balancing or Auto Scaling. The other type of environment can launch multiple EC2 instances and includes load balancing and Auto Scaling configuration.

Elastic Beanstalk provisions necessary infrastructure resources, such as ELB, Auto Scaling groups, security groups, and databases (optional).

Lab 4:
Automation with Terraform

Develop Intelligence