# Infrastructure as Code (IaC) Introduction

CSYE 6225: Network Structure & Cloud Computing
Northeastern University

Instructor: Raja Alomari,PhD



## M5: Overview

Topics include:

- Overview of IaC Tools (Terraform, AWS CloudFormation, ARM, GDM).
- YAML vs JSON.
- AWS CloudFormation Stack and Changeset.



# Infrastructure as Code (IaC)





Objectives:

- Understand the Fundamentals of Infrastructure as Code (IaC).
- Familiarize with Popular IaC Tools.
- Compare YAML and JSON Formats.
- Deploy and Manage AWS Resources Using CloudFormation.
- Develop Hands-on Skills in Cloud Infrastructure Automation.



## What is IaC?

Infrastructure as Code (IaC) automates infrastructure **provisioning** and **management** by defining cloud resources in machine-readable configuration files. This eliminates manual configurations and provides consistency across environments.

IaC enables automation, repeatability, and version control for infrastructure. Instead of manually clicking in cloud provider dashboards, engineers describe their infrastructure in code, which can be deployed automatically.

## Why IaC?

**Faster deployments:** Infrastructure can be set up in minutes rather than days.

**Consistency and repeatability:** Every time you deploy, the same configuration is applied.

**Reduced human error:** By eliminating manual steps, IaC helps avoid configuration drift.

**Disaster recovery:** Rapid restoration of infrastructure from code in case of failure.

**Scaling:** Easily scale infrastructure up or down based on need, using predefined rules and configurations.



## IaC Tools Overview

Terraform (Open Source)

Overview: Terraform by HashiCorp is an open-source IaC tool that allows you to manage infrastructure across multiple cloud providers (AWS, Azure, Google Cloud, etc.) using HCL (HashiCorp Configuration Language).

**https://www.terraform.io/**

Key Concepts

**Declarative Approach:** Define what infrastructure you want (not how to build it), and Terraform handles the process.

**State Management:** Terraform keeps a local or remote "state" file to track what resources have been created and ensure that it knows the current state of your infrastructure. This allows incremental updates.

**Execution Plan (Plan/Apply):** Terraform generates a plan to show what will happen before applying any changes, and then it applies the changes to create or modify resources.



## IaC Tools Overview

Advantages

**Multi-cloud support:** You can manage resources across different cloud providers in the same configuration file.

**Modular:** Terraform configurations can be split into reusable modules, allowing for better organization and reuse.

**Providers:** Terraform supports hundreds of providers, from major cloud platforms like AWS, GCP, and Azure to SaaS solutions like GitHub and Kubernetes.

Terraform Workflow

**terraform init:** Initializes the configuration and downloads the necessary providers.

**terraform plan:** Compares the current state of infrastructure with the desired state described in the configuration files. Shows what changes will be made.

**terraform apply:** Applies the changes described in the configuration to bring the infrastructure to the desired state.

**terraform destroy:** Removes all resources defined in the configuration.



## Example: AWS EC2 Instance

```
provider "aws" {
  region = "us-west-2"
}
resource "aws_instance" "example" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"
  tags = {
    Name = "ExampleInstance"}}
```

## Terraform Features:

**Providers:** Terraform has a large library of providers, enabling management across platforms (AWS, Azure, GCP, Kubernetes, etc.).

**State Management:** The state file (terraform.tfstate) keeps track of what Terraform manages and allows it to perform incremental changes efficiently.

**Modules:** Modules are reusable configurations that make it easy to organize and share infrastructure definitions.



## AWS CloudFormation

Overview: CloudFormation is AWS's native IaC tool that enables you to define and provision AWS infrastructure using JSON or YAML templates.

It supports complex setups and automates resource creation and management in an AWS environment.




## CloudFormation Components:

**Stacks:** A collection of AWS resources that are managed together. Updates to the stack propagate to all resources.

**Templates:** Written in JSON or YAML, these templates define the desired infrastructure. Each template has sections such as Resources, Parameters, Mappings, and Outputs.

**Changesets:** Changesets allow you to preview changes before applying them, helping avoid accidental disruptions.

**Rollback:** If a deployment fails, CloudFormation can roll back to a previous stable state automatically.



## CloudFormation Template Example (EC2 Instance)

```
Resources:
  MyEC2Instance:
    Type: "AWS::EC2::Instance"
    Properties:
      InstanceType: "t2.micro"
      ImageId: "ami-0abcdef1234567890"
      KeyName: my-key
```

## CloudFormation Advantages

**Fully integrated with AWS:** It natively supports all AWS services.

**Automatic dependency management:** CloudFormation automatically determines the order of resource creation based on dependencies between resources.

**Drift Detection:** Detects changes that were made outside CloudFormation, ensuring the infrastructure stays consistent with the code.



## Azure Resource Manager (ARM)

Overview: ARM templates are used to deploy and manage Azure resources declaratively.

ARM supports JSON templates and integrates deeply with Azure services.




## Azure Resource Manager (ARM)

Key Features

**Resource Groups:** Organize Azure resources into resource groups, which allow you to manage, deploy, and monitor related resources together.

**Declarative:** ARM templates are written in JSON, where you declare the state you want the infrastructure to be in.

**Template Validation:** Azure Resource Manager validates your template before deploying it, ensuring that the syntax is correct and that dependencies between resources are met.




## ARM Template Example

```
{ "$schema":
"https://schema.management.azure.com/schemas/2019-
04-01/deploymentTemplate.json#",
 "contentVersion": "1.0.0.0",
 "resources": [{
    "type": "Microsoft.Compute/virtualMachines",
    "apiVersion": "2020-06-01",
    "name": "myVM",
    "location": "eastus",
```

```
"properties": {
    "hardwareProfile": {
      "vmSize": "Standard_DS1_v2"
    },
    "osProfile": {
      "computerName": "myVM",
      "adminUsername": "azureuser"
    }}}]}
```

## Azure Resource Manager (ARM)

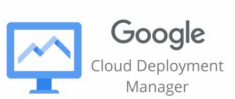
Features

- **Integration with CI/CD:** ARM templates are commonly used with Azure Pipelines for Continuous Integration and Continuous Deployment (CI/CD).
- **Role-based Access Control (RBAC):** Control who can deploy resources and what permissions they have within resource groups.
- **Tags and Policies:** Use tags for resource categorization and apply policies to enforce organizational standards.



## Google Cloud Deployment Manager

**Overview:** Google Cloud Deployment Manager (GDM) is a declarative tool for defining and managing Google Cloud resources using YAML, Python, or Jinja templates. Like AWS CloudFormation and Azure ARM, GDM simplifies cloud resource management on Google Cloud.






## Google Cloud Deployment Manager

**Features**

**Templates:** Define reusable and modular infrastructure templates in YAML or Python.

**Repeatability:** Create the same environment repeatedly across different projects.

**Complex Deployments:** Handle the deployment of complex resources like Kubernetes clusters, GCE instances, or BigQuery datasets.



## Example: GDM YAML

```
resources:
  - name: my-vm
    type: compute.v1.instance
    properties:
      zone: us-central1-a
      machineType:
zones/us-central1-a/machineTypes/
n1-standard-1
```

```
 disks:
        - deviceName: boot
          type: PERSISTENT
          boot: true
          autoDelete: true
          initializeParams:
            sourceImage:
projects/debian-cloud/global/images
/family/debian-9
```

## Example: GDM Python

```
def GenerateConfig(context):

  resources = [

    {

      'name': 'my-vm',

      'type': 'compute.v1.instance',

      'properties': {

        'zone': context.properties['zone'],

        'machineType': 'zones/' +
context.properties['zone'] +
'/machineTypes/n1-standard-1',
```

```
'disks': [{

          'deviceName': 'boot',

          'boot': True,

          'autoDelete': True,

          'initializeParams': {

            'sourceImage':
'projects/debian-cloud/global/images/family/debian-9'

          }}]}]

  return {'resources': resources}
```



## More Features: GDM

**Dynamic Templates:** Allow for conditional logic, loops, and other advanced configurations using Python or Jinja.

**Infrastructure Deployment:** Automate the deployment of Compute Engine VMs, Cloud Storage, BigQuery datasets, and more.



## YAML vs JSON

**YAML**

**YAML (Yet Another Markup Language)** is a data serialization format that is human-readable and widely used in IaC.

Key Features:

- Readable: Indentation-based syntax that is easy to read and write.
- Hierarchical: Represents nested data structures clearly using indentation.
- Supports comments: # to include comments, making configurations easier to document.

When to Use YAML:

- YAML is more readable for humans and is often used in complex configurations, such as Kubernetes manifests or AWS CloudFormation templates.



## YAML vs JSON

**JSON**

**JSON (JavaScript Object Notation)** is a lightweight data interchange format that is easily parsed by machines but less human-readable than YAML.

**Key Features:**

- Widely used: JSON is the standard for many APIs and is native to JavaScript.
- Strict format: Requires curly braces, commas, and quotation marks around keys.
- Does not support comments: JSON does not support comments, which can make larger configurations harder to document.

**When to Use JSON:**

- JSON is preferred when working with systems that require strict format compliance and can easily parse JSON (e.g., ARM templates in Azure or API responses).

## YAML vs JSON

**YAML**

```yaml
Resources:
  MyS3Bucket:
    Type: "AWS::S3::Bucket"
    Properties:
      BucketName: "my-bucket"
      AccessControl: "Private"
```

**JSON**

```json
{
  "Resources": {
    "MyS3Bucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": "my-bucket",
        "AccessControl": "Private"
  }}}}
```



## AWS Cloud Formation: Deploying a Stack

What is a Stack?

- A stack is a collection of AWS resources that are created, updated, or deleted together as a single unit. You can use AWS CloudFormation to create, update, or delete these resources by uploading a CloudFormation template.

Deploying a Stack:

- **Template Upload:** Write your infrastructure in YAML/JSON, upload the template to the CloudFormation console, and specify any parameters.
- **Stack Creation:** The stack creates all resources defined in the template in the order specified by their dependencies.
- **Stack Updates:** Modify your template to update resources (e.g., change EC2 instance size, add new services). The stack will apply changes without redeploying everything.



## AWS Cloud Formation: Deploying a Stack

ChangeSet Workflow

**Create a ChangeSet:** Modify your template, upload it, and create a ChangeSet. CloudFormation will show you a list of changes (additions, modifications, deletions).

**Review Changes:** Before applying, you can review how your resources will change (e.g., replacing an EC2 instance, updating security groups).

**Apply Changes:** Once satisfied with the ChangeSet, apply it, and CloudFormation will update the stack accordingly.



## Terraform: Deployment

**Terraform State and Plan/Apply**

**State:** Terraform stores the current state of your infrastructure in a local or remote state file. This state file allows Terraform to understand which resources exist and what needs to be created, modified, or destroyed.

**Plan and Apply:**

- **terraform plan:** Shows the execution plan, comparing the desired state with the current state, and previewing the changes.
- **terraform apply:** Executes the plan and applies the necessary changes to your infrastructure.
- **Handling Drift:** Terraform compares actual resources with the desired state in the state file. If something changes outside of Terraform (e.g., manual changes in the AWS console), Terraform will detect this drift and attempt to correct it.

# Case Study – AWS CloudFormation Deployment of EC2



## Terraform: Deployment

**Scenario:** Deploying a highly available web application using AWS CloudFormation

**Objective:** Deploy a simple web application running on EC2 instances in an Auto Scaling Group with a load balancer in front of it and an RDS database for persistence.

Steps:

- **VPC Setup:** Create a VPC with public and private subnets across multiple Availability Zones.
- **Security Groups:** Define security groups to control inbound and outbound traffic for the EC2 instances, RDS, and Load Balancer.
- **Elastic Load Balancer:** Set up an Application Load Balancer (ALB) to distribute traffic across EC2 instances.
- **Auto Scaling Group:** Deploy EC2 instances across multiple Availability Zones and attach them to the ALB.
- **RDS:** Set up a managed RDS database instance in the private subnets, with security groups to allow EC2 access.
- **Outputs:** Provide key information like the Load Balancer DNS and EC2 instance IDs as outputs for the stack.



## Example: CloudFormation Template

```
Resources:
  WebServerGroup:
    Type: AWS::AutoScaling::AutoScalingGroup
    Properties:
      AvailabilityZones: !GetAZs ''
      LaunchConfigurationName: !Ref LaunchConfig
      MinSize: '1'
      MaxSize: '3'
      DesiredCapacity: '2'
      LoadBalancerNames: [!Ref LoadBalancer]
```



## Example: CloudFormation Template

```
  LaunchConfig:
    Type: AWS::AutoScaling::LaunchConfiguration
    Properties:
      ImageId: "ami-0abcdef1234567890"
      InstanceType: t2.micro
      SecurityGroups: [!Ref WebServerSecurityGroup]
```

## Example: CloudFormation Template

```
LoadBalancer:
  Type: AWS::ElasticLoadBalancing::LoadBalancer
  Properties:
    Subnets: !Ref PublicSubnets
    SecurityGroups: [!Ref LoadBalancerSecurityGroup]
    Listeners:
      - LoadBalancerPort: '80'
        InstancePort: '80'
        Protocol: HTTP
```



## AWS CloudFormation: Advanced Features

**Stack Outputs:** Define outputs to provide useful information (e.g., load balancer DNS, EC2 IP addresses) that can be referenced by other stacks or used in CI/CD pipelines.

**Stack Policies:** Apply policies to restrict which resources can be updated during stack changes, preventing accidental deletions of critical resources.

**Rollback Triggers:** Automate rollbacks if certain resources fail to create or update within specified time limits.



# Best Practices in IaC



## Best Practices in IaC

**Version Control and GitOps**

Store IaC templates in version control (e.g., Git). Every infrastructure change is tracked, and history can be reviewed or reverted if needed.

GitOps: Use version control as the single source of truth for infrastructure and application deployments. Changes are pushed to Git, and automation tools apply the changes to the infrastructure.

**Testing and Validation**

Use linting tools to validate IaC templates for syntax errors and best practices (e.g., terraform validate, cfn-lint for CloudFormation).

Implement unit tests and integration tests to ensure that infrastructure behaves as expected in different environments (e.g., dev, staging, production).

## Best Practices in IaC

**CI/CD Integration**

Integrate IaC with Continuous Integration/Continuous Deployment pipelines (e.g., Jenkins, CircleCI, GitHub Actions) to automate infrastructure provisioning and updates.

Ensure that infrastructure changes are tested and approved before being deployed to production.

**Modularity and Reusability**

Break large infrastructure definitions into smaller, reusable components (e.g., Terraform modules, nested CloudFormation stacks).

Use parameters to make templates flexible and reusable across different environments.



## Best Practices in IaC

**Security and Compliance**

Implement IAM policies and role-based access control (RBAC) to ensure that only authorized users can deploy or modify infrastructure.

Use encryption for sensitive data (e.g., encrypt RDS databases and S3 buckets in CloudFormation).

Regularly perform drift detection to identify and address configuration drift between code and actual infrastructure.



## Hands-on

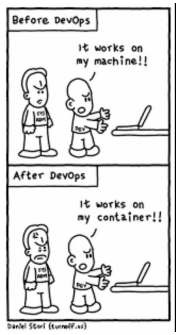
- Objective:
  - Install Terraform
  - Deploy and destroy multiple Stacks on AWS.
  - Usage of AWS CloudFormation and Terraform.



## Recommended Readings

1. Infrastructure as Code: Dynamic Systems for the Cloud Age 2nd Edition by Kief Morris.
2. Terraform: Up and Running: Writing Infrastructure as Code by Yevgeniy Brikman


## IaC






## Module 5 Conclusion

- Infrastructure as Code (IaC) Enhances Automation and Consistency.
- Choosing the Right IaC Tool Depends on Your Cloud Environment.
- YAML is Preferred for Human Readability, JSON for Machine Interoperability.
- ChangeSets in AWS CloudFormation Provide a Safe Way to Update Infrastructure.