INFRASTRUCTURE AS CODE (IAC): THE COMPLETE BEGINNER'S GUIDE

Infrastructure is one of the core tenets of a software development process—it is directly responsible for the stable operation of a software application. This infrastructure can range from servers, load balancers, firewalls, and databases all the way to complex container clusters.

Infrastructure considerations are valid beyond production environments, as they spread across the complete development process. They include tools and platforms such as CI/CD platforms, staging environments, and testing tools. These infrastructure considerations increase as the level of complexity of the software product increases. Very quickly, the traditional approach for manually managing infrastructure becomes an unscalable solution to meet the demands of DevOps-based modern rapid software development cycles.

And that's how Infrastructure as Code (IaC) has become the de facto solution in development today. IaC allows you to meet the growing needs of infrastructure changes in a scalable and trackable manner.

What is infrastructure as code?

Infrastructure as Code or IaC is the process of provisioning and managing infrastructure defined through code, instead of doing so with a manual process.

As infrastructure is defined as code, it allows users to easily edit and distribute configurations while ensuring the desired state of the infrastructure. This means you can create reproducible infrastructure configurations.

Moreover, defining infrastructure as code also:

- **Allows infrastructure to be easily integrated** into version control mechanisms to create trackable and auditable infrastructure changes.
- **Provides the ability to introduce extensive automation for infrastructure management**. All these things lead to IaC being integrated into CI/CD pipelines as an integral part of the SDLC.
- **Eliminates the need for manual infrastructure provisioning and management**. Thus, it allows users to easily manage the inevitable config drift of underlying infrastructure and configurations and keep all the environments within the defined configuration.

Declarative vs imperative Infrastructure as Code

When dealing with IaC tools, there are two major differentiating approaches for writing code. These two approaches are declarative and imperative. Simply put:

- **An imperative approach** allows users to specify the exact steps to be taken for a change, and the system does not deviate from the specified steps.
- **A declarative approach** essentially means users only need to define the end requirement, and the specific tool or platform handles the steps to take in order to achieve the defined requirement.

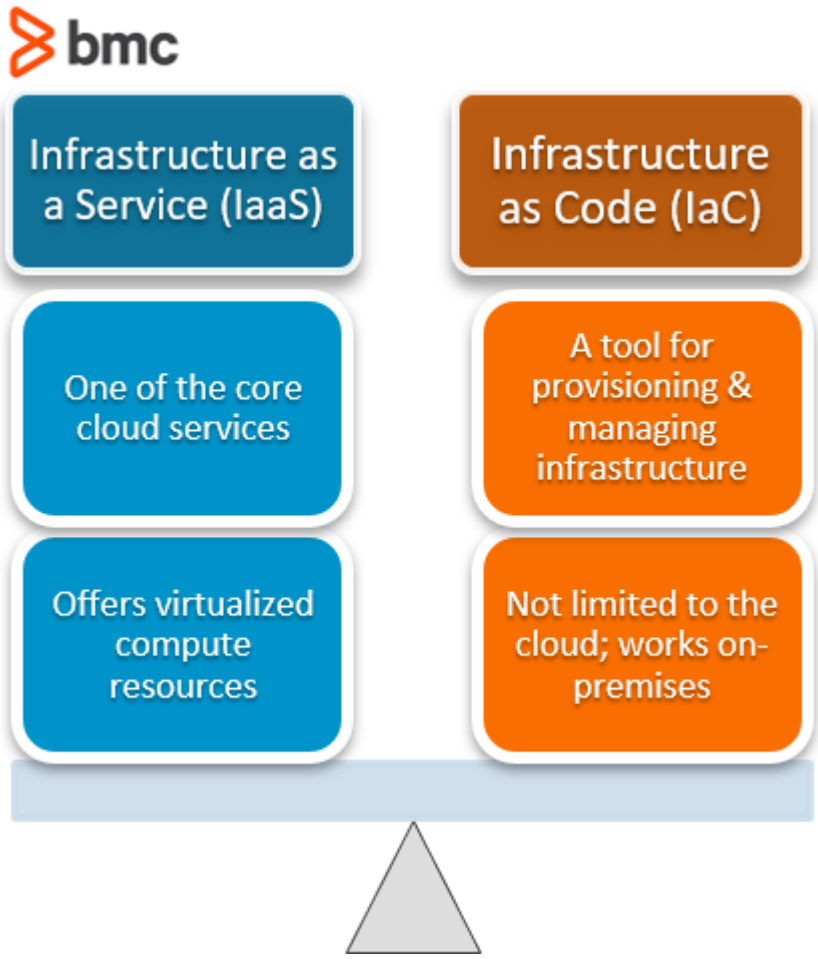
The declarative approach is preferred in most infrastructure management use cases as it offers a greater degree of flexibility when managing infrastructure.

Chef is considered an imperative tool, where Terraform, Pulumi, CloudFormation, ART, Puppet are all declarative. Uniquely, Ansible is mostly declarative with support for imperative commands.

IaC vs IaaS

Importantly, IaC is not a derivative of infrastructure as a service (IaaS). They are two different concepts.

- **Infrastructure as a Service** is one of the core cloud services: virtualized computing resources—servers, networking infrastructure, storage, etc.—are provided via the cloud service.
- **Infrastructure as Code** is a tool that can be used to provision and manage infrastructure. It is not limited to only cloud-based resources. In fact, you apply IaC to a wide variety of environments, including on-premises.



When & how to use Infrastructure as Code

IaC may seem unnecessary for simpler, less complex infrastructure requirements, but that isn't accurate. Any—every—modern software development pipeline should use infrastructure as Code to handle the infrastructure.

Besides, the advantages of IaC far outweigh any implementation and management overheads.

Advantages of IaC

Here are the top benefits of IaC:

- Reducing shadow IT within organizations and allowing timely and efficient infrastructure changes that are done in parallel to application development.
- Integrating directly with CI/CD platforms.
- Enabling version-controlled infrastructure and configuration changes leading to trackable and auditable configurations.
- Easily standardizing infrastructure with reproducible configurations.
- Effectively managing configurating drift and keeping infrastructure and configurations in their desired state.
- Having the ability to easily scale up infrastructure management without increasing CapEx or OpEx. With IaC, you'll reduce CapEx and OpEx spending overall, as automation eliminates the need for time-consuming manual interactions and reduces incorrect configurations.

When to use IaC

Not sure when to use IaC? The simplest answer is whenever you have to manage any type of infrastructure.

However, it becomes more complex with the exact requirements and tools. Some may require strict infrastructure management, while others may require both infrastructure and configuration management. Then comes platform-specific questions like if the tool has the necessary feature set, security implication, integrations, etc. On top of that, the learning curve comes into play as users prefer a simpler and more straightforward tool than a complex one.

The below table shows a categorization of the tools mentioned above according to their ideal use cases.

Use Case	Tools to use
Infrastructure management	Terraform, Pulumi, AWS CloudFormation, Azure Resource Templates
Configuration management with somewhat limited infrastructure management capabilities	Ansible, Chef, Puppet
Configuration management	CFEngine

One tool may not be sufficient in most scenarios. For instance, Terraform may be excellent for managing infrastructure across multiple cloud environments yet may be limited when in-depth configurations are required. In those kinds of situations, users can utilize a tool such as Ansible to carry out the necessary configurations.

Likewise, users can mix and match any IaC tool and use them in their CI/CD pipelines depending on the exact requirements.

(Learn how to set up your own CI/CD pipeline.)

Infrastructure as Code tools & platforms

Under the big IaC umbrella, there are all sorts of tools, from dedicated infrastructure management tools to configuration management, from open-source tools to platform-specific IaC options.

Let's look at some of the most popular IaC tools and platforms.

Terraform

Terraform by HashiCorp is the leading IaC tool specialized in managing infrastructure across various platforms from AWS, Azure, GCP to Oracle Cloud, Alibaba Cloud, and even platforms like Kubernetes and Heroku.

As a platform-agnostic tool, Terraform can be used to facilitate any infrastructure provisioning and management use cases across different platforms and providers while ensuring the desired state across the configurations.

Ansible

Ansible is not a dedicated Infrastructure management tool but more of an open-source configuration management tool with IaC capabilities. Ansible supports both cloud and on-prem environments and can act through SSH or WinRM as an agentless tool. Ansible excels at configuration management and infrastructure provisioning yet is limited when it comes to managing said infrastructure.

(Find out why people often compare Ansible & Control-M.)

Pulumi

Pulumi is a relatively new tool that aims to provide a developer-first IaC experience. Unlike other tools that force users to use a specific language or format, Pulumi offers freedom to use any supported programming language any way they like.

This tool supports Python, TypeScript, JavaScript, Go, C#, F#, and the state is managed through Pulumi service by default.

Chef/Puppet

Chef and Puppet are two powerful configuration management tools. Both aim to provide configuration management and automation capabilities with some infrastructure management capabilities across the development pipeline.

- Chef is developed to be easily integrated into DevOps practices with greater collaboration tools.
- Puppet evolved by targeting sheer processes automation. Today, Puppet has automated built-in watchers to identify configuration drift.

(Check out Puppet's State of DevOps report.)

CFEngine

CFEngine is one of the most mature tools solely focused on configuration management. Even though there is no capability to manage the underlying infrastructure, CDEngine can accommodate even the most complex configuration requirements, covering everything from security hardening to compliance.

AWS CloudFormation

CloudFormation is the AWS proprietary platform specific IaC tool to manage AWS infrastructure. CloudFormation has deep integration with all AWS services and can facilitate any AWS configuration as a first-party solution.

Azure Resource Templates

Microsoft Azure uses JSON-based Azure Resource Templates to facilitate IaC practices within the Azure platform. These resource templates ensure consistency of the infrastructure and can be used for any type of resource configuration.

In addition to the above, there are specialized tools aimed at specific infrastructure and configuration management tasks such as:

- Packer, EC2 Image Builder, and Azure Image Builder create deployable custom os images.
- Cloud-Init is the industry-standard cross-platform cloud instance initialization tool. It enables users to execute the script when provisioning resources (servers).
- (R)?ex is a fully featured infrastructure automation framework

(Get acquainted with Azure DevOps.)

Examples of Infrastructure as Code

Let's consider a simple scenario of provisioning an AWS EC2 Instance. In the following example, we can see how Terraform, Ansible, and AWS CloudFormation codes are used for this requirement.

Terraform

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.27"
    }
  }
}

provider "aws" {
  access_key = "aws_access_key"
  secret_key = "aws_secret_key"
  // shared_credentials_file = "/Users/.aws/creds"
  region = "us-west-1"
}

resource "aws_instance" "web_server" {
  ami                    = "ami-0123456"
  instance_type          = "t3.small"
  subnet_id              = "subnet-a000111x"
  vpc_security_group_ids = "sg-dfdd00011"
  key_name               = "web_server_test_key"

  tags = {
    Name = "Web_Server"
  }
}
```

Ansible

```
- hosts: localhost
gather_facts: False
vars_files:
- credentials.yml
tasks:
- name: Provision EC2 Instance
ec2:
aws_access_key: "{{aws_access_key}}"
aws_secret_key: "{{aws_secret_key}}"
key_name: web_server_test_key
```

```
group: test
instance_type: t3.small
image: "ami-0123456"
wait: true
count: 1
region: us-west-1
instance_tags:
Name: Web_Server
register: ec2
```

AWS CloudFormation

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
WebInstance:
Type: AWS::EC2::Instance
Properties:
InstanceType: t3.small
ImageId: ami-0123456
KeyName: web_server_test_key
SecurityGroupIds:
- sg-dfdd00011
SubnetId: subnet-a000111x
Tags:
-
Key: Name
Value: Web_Server
```

A real world example: IaC for DevOps

Within the context of software development, a fundamental constraint is the need for the environment where recently developed software code is tested to exactly mirror the live environment where such code will be deployed to. This is the only way of assuring that the new code will not collide with existing code definitions: by generating errors or conflicts that may compromise the entire system.

In the past, software delivery would follow this sort of pattern:

1. A System Administrator would setup up a physical server and install the operating system with all necessary service packs and tuning to mirror the status of the main operating live machine that supports the production environment.
2. Then a Database Administrator would undergo the same process for the support database, and the machine would be handed off to a test team.
3. The developer would deliver the code/program by copying it to the test machine, and the test team would run several operational and compliance tests.
4. Once the new code has gone through the entire process, you can deploy it to the live, operational environment. In many cases, the new code won't work correctly, so additional troubleshooting and rework are necessary.

(Understand the differences between deploying & releasing software.)

Manual recreation of a live environment leaves doors open to a multitude of most likely minor but potentially quite important human errors, regarding:

- OS version
- Patch level
- Time zone
- Etc.

A live environment clone, created using the exact same IaC as the live environment, has the absolute guarantee that that if it works in the cloned environment it will work in live.

Imagine a software delivery process involving separate environments for DEV, UAT, and Production. There's seemingly little value in having a DEV and UAT environment that isn't an exact mirror of the prod environment given that those early environments are critical to measuring the quality and production readiness of a software build version.

The introduction of virtualization enabled this process to be expedited, especially regarding the phase of creating and updating a test server that would mirror the live environment. Yet the process was manual, meaning a human would have to create and update the machine accordingly and in a timely fashion. With the introduction of DevOps, these process became even more "agile". Adding automation to the server virtualization and testing phases replaces human intervention, improving productivity and efficiency.

To summarize: In the past, several man-hours and human resources were required to complete the software deployment cycle (Developers, Systems Administrators, Database Administrators, Operation testers). Now, it is possible to have the developer alone complete all tasks:

1. The developer writes the application code and the configuration management-related instructions that will trigger actions from the virtualization environment, and other environments such as the database, appliances, testing tools, delivery tools, and more.
2. Upon new code delivery, the configuration management instructions will automatically create a new virtual test environment with an application server plus database instance that exactly mirrors the live operational environment structure, both in terms of service packs and versioning as well as live data that is transferred to such virtual test environment. (This is the Infrastructure as Code part of the process.)
3. Then a set of tools will perform necessary compliance tests and error identification and resolution. The new code is then ready for deployment to the live IT environment.

Quick, trackable infrastructure changes

Infrastructure as Code has become a vital part of modern application development and deployment pipelines. It is achieved by facilitating quick and trackable infrastructure changes that directly integrate into CI/CD platforms. Infrastructure as Code is crucial for both:

- Facilitating scalable infrastructure management
- Efficiently managing the config drift in all environments

Getting started with Infrastructure as Code may seem daunting with many different tools and platforms targeted at different use cases. However, cross this hurdle, and you will have a powerful

infrastructure management mechanism at your fingertips.

Related reading

- BMC DevOps Blog
- Software Project Management Phases & Best Practices
- GitHub, GitLab, Bitbucket & Azure DevOps: What's The Difference?
- Container Sprawl: What It Is & How To Avoid It
- SRE vs DevOps: What's The Difference?
- Low Code vs No Code Explained