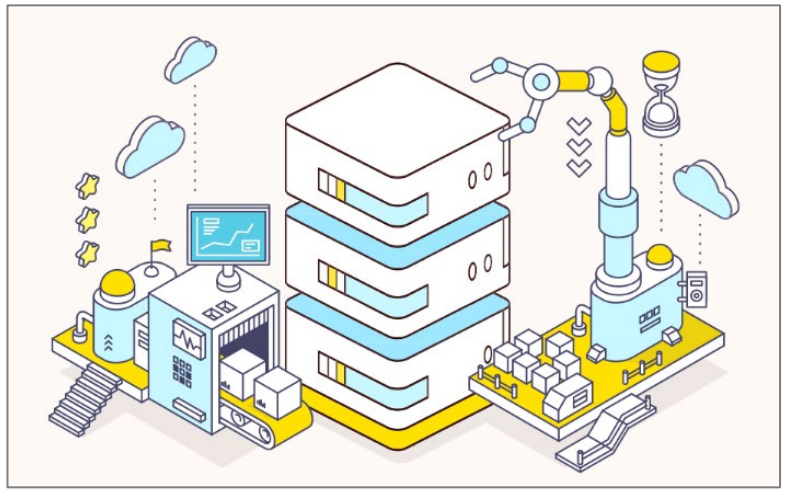

# Introduction to Infrastructure as Code

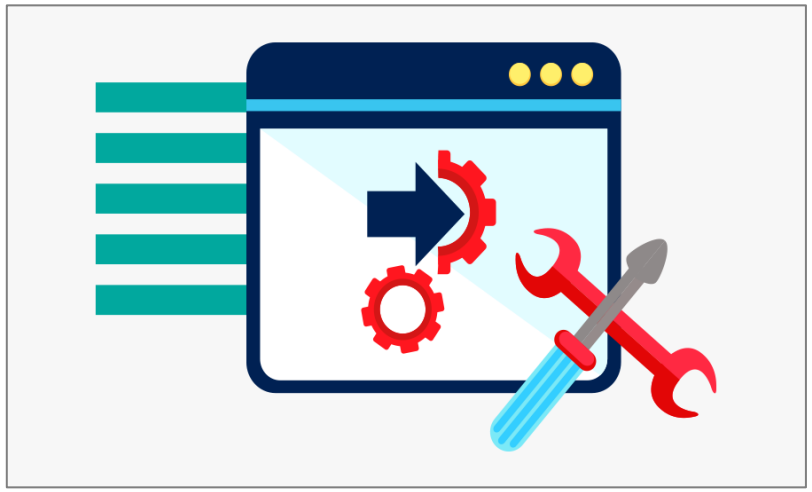
# Deploying an Application

## Provisioning



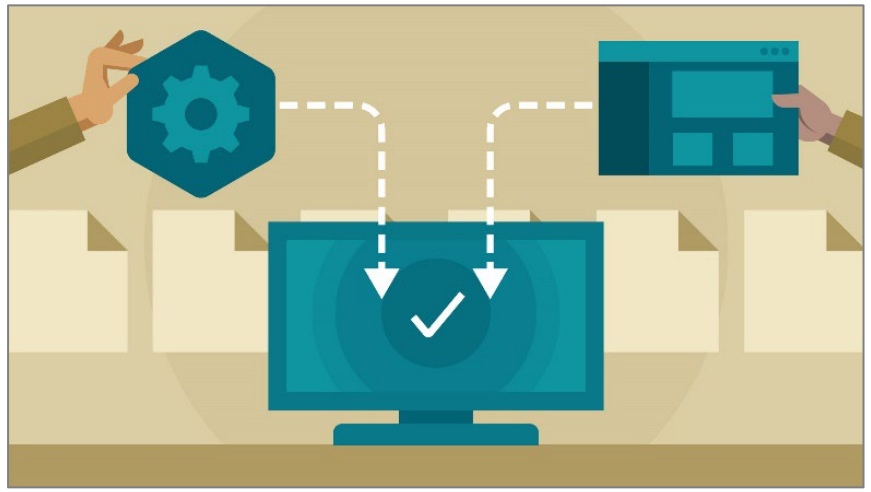
Topology of the data center

## Configuration



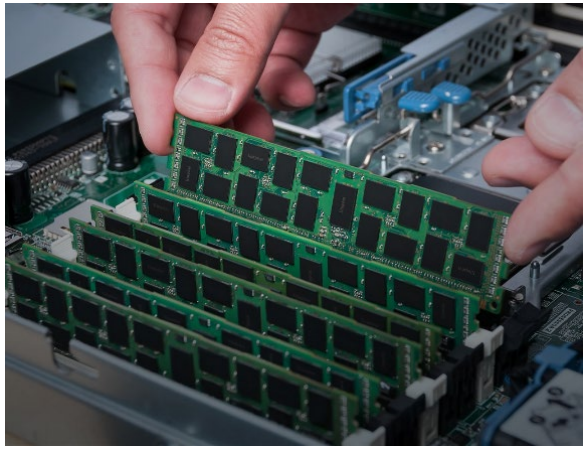
Setup and configure the environment

## Deployment



Deploy the application

# Provisioning Servers - The Old Way

# Migrating to the Cloud

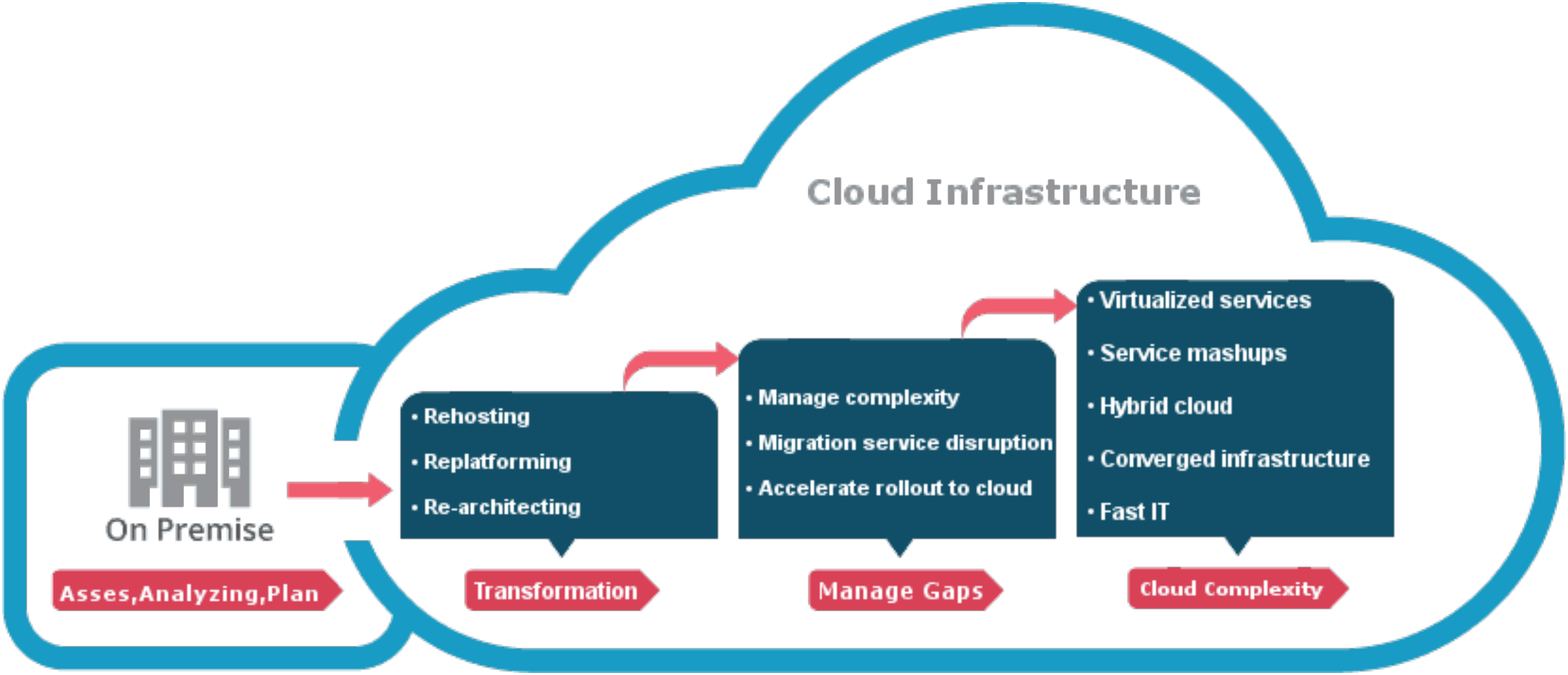


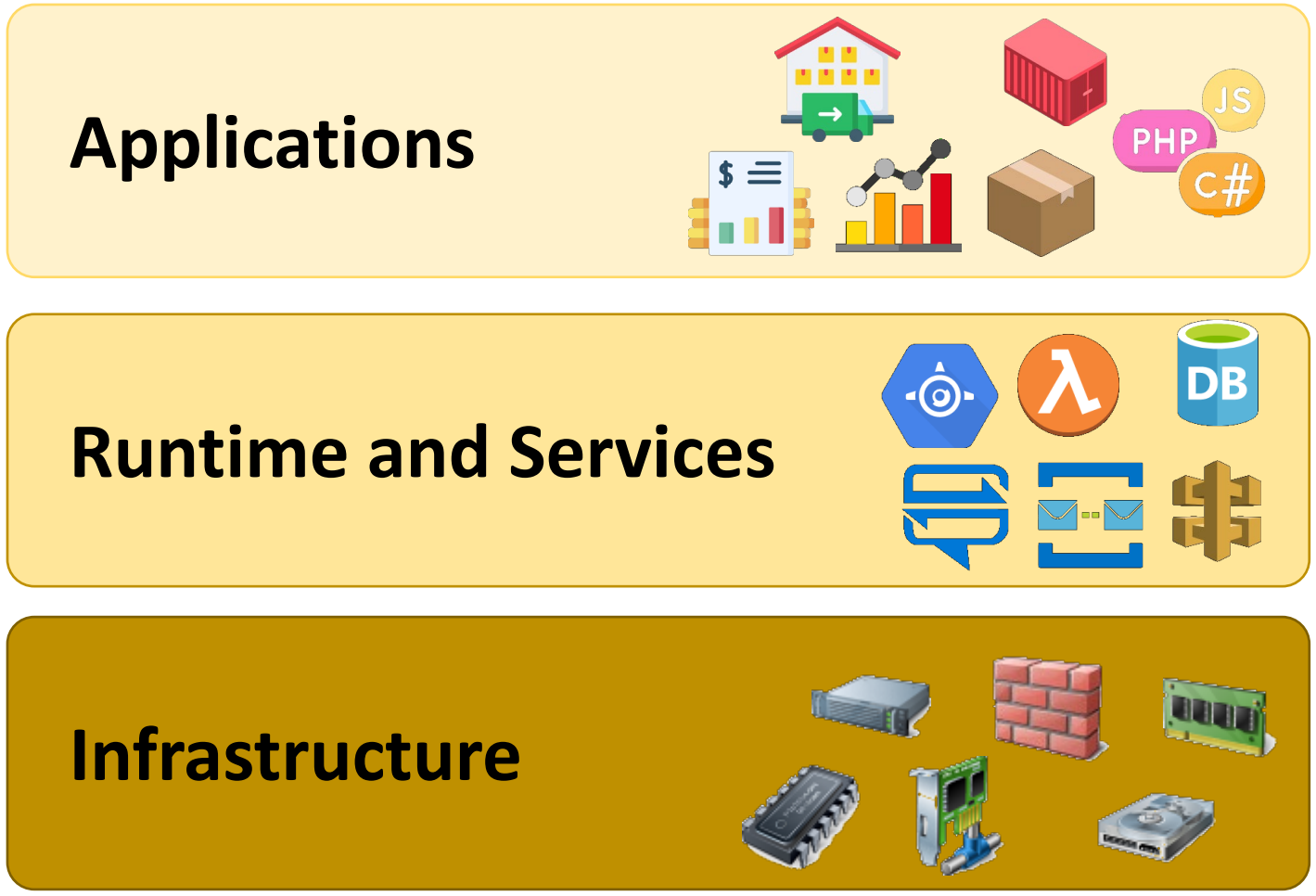

Image from https://interpole.net/cloud-blog/how-to-pick-the-best-applications-for-cloud-migration/

# Typical Cloud Platform

**Software as a Service**
Managed applications and custom applications

**Platform as a Service**
Managed services eg. database, runtime, serverless, message queues, API gateway

**Infrastructure as a Service**
Traditional IT resources eg. servers, disk, subnets, cluster, backups



# Principles of Cloud Computing

- Assume system is unreliable
  - The hardware is unreliable because you have no visibility or control over them
  - Application must be reliable to compensate for the hardware's unreliability
- Create disposable things
  - Systems should be able to withstand dynamic changes to its constituent components
  - System components should be able to gracefully add, remove, stop, start, modify and migrate
- Make everything reproducible
  - Allow you to quickly recover from failures
  - Minimize statefulness especially in applications

From O'Reilly Infrastructure as Code

# Principles of Cloud Computing

- Minimize variation
  - Easier to automate the management if systems is largely the same eg. same OS, package versions, configuration, etc
  - Basis of building reproducible systems
- Ensure that you can repeat any action
  - Run and re perform any action will result in the same system/configuration
  - Capture list of actions in script, allow you to create systems with minimal variation

From O'Reilly Infrastructure as Code


# Principles of Cloud Computing

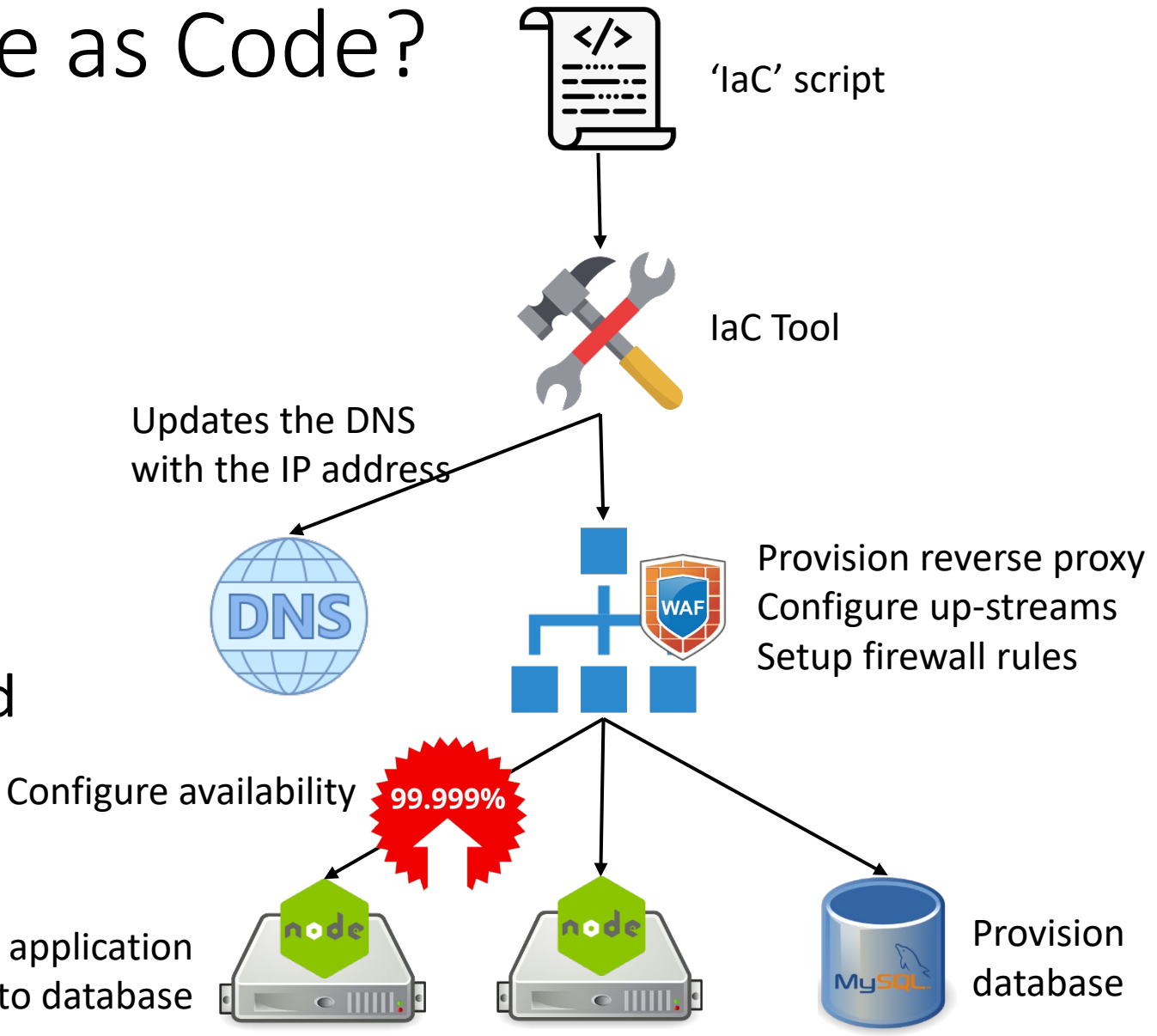
**Immutable**

**Ephemeral**

**Idempotent**

# What is Infrastructure as Code?

- Process of provisioning and managing IT resources through machine readable files
- Script/program describes what are the required setup and configurations
- Tool to provision, configure and install



# Example of an IaC Script

```
resource “digitalocean_ssh_key” “web_ssh_key” {
  name = “web_ssh_key”
  public_key = file(“./mykey.pub”)
}

resource “digitalocean_droplet” “webapp_droplet” {
  image = “ubuntu-20-40-x64”
  name = “webapp_droplet”
  region = “sgp1”
  size = “s-1vcpu-1gb”
  ssh_keys = [ digitalocean_ssh_key.web_ssh_key.fingerprint ]
  provisioner “remote-exec” {
    inline = [
      “apt update”,
      “apt upgrade -y”,
      “apt install nginx -y”
    ]
  }
}
```

Provision a server with a SSH key
Install Nginx


# Why is this Possible?

- IT resources are mostly virtualized
  - Through cloud providers, Kubernetes/Docker
- Result of virtualization, resources can be provisioned in minutes via browser or command line

```
doctl compute droplet create \
  --image ubuntu-20-04-x85 --size s-1vcpu-1gb \
  --region sgp1 myserver
```

  - Expose endpoint to support tools and applications
- Developers can leverage these API
  - Through official SDKs
  - Calling the endpoints directly, typically HTTP

# Benefit of IaC - Immutable Infrastructure

- Infrastructure is codified in scripts
  - Will deploy the same exact setup every time the script is executed
  - Captures know-how
- Mitigate configuration drift
  - “Works on my machine”
  - Consistent updates, server configurations the same across all servers
  - No snowflakes or fragile infrastructure
- Parity between development, testing and deploy
  - Fewer issues when moving from development to testing to deployment
  - More confident when application goes into production because the testing environment is the same as the production
- Promotes infrastructure reuse
  - Modularize and parameterize the infrastructure eg. what goes into a basic server
  - Consistently use the same basic building blocks, increase security and availability

# Benefits of IaC - Programmers Paradigm

- Tools to help with script development
  - Syntax checking, code completion, linting
- Abstraction and reuse
  - Package commonly used infrastructure as ‘components’ to be shared
- Testing
  - Unit, integration, smoke, end to end
- Tracked and versioned controlled
  - Reviewed via pull request


# Result of Adopting IaC

- Decrease the time for implementing, testing and deploying application/infrastructure changes into production
  - Easier to setup consistent environment for testing
  - Bug fixes, new feature releases can easily be tested
- Increase deployment frequency
  - Environment can be created easily, can lead to more frequent releases for new features
  - Align with agile practice - small frequent updates

# Result of Adopting IaC

- Reduce the number of failures when an change is applied to a production application
  - Parity of environment in development, testing, staging and production
  - Reduce the probability of misconfiguration resulting in failure
- Reduce the time it takes to restore an application
  - Entire application's configuration and setup is describe and capture in scripts
  - Application can be recreated by running the scripts
  - Note: Need to factor data restoration time

# Infrastructure as Code - Mindset Changes

| Manual Way | Automation Way |
|---|---|
| Cost of changes are high | Cost of changes are low |
| Changes represent failure, therefore must be managed and controlled | Changes represent learning and improvement |
| Reduce opportunity to fail | Maximize speed of improvement |
| Application features are delivered in large batches of changes, test end-to-end | Deliver small changes, test continuously |
| Applications have long release cycle | Applications have short release cycle |
| Monolithic architecture - larger, less moving parts | Microservices architecture - smaller, more moving parts |
| Physical or GUI driven changes | Code as configuration |

Table from Infrastructure As Code 2nd Ed: Table 1-2

# IaC Tools

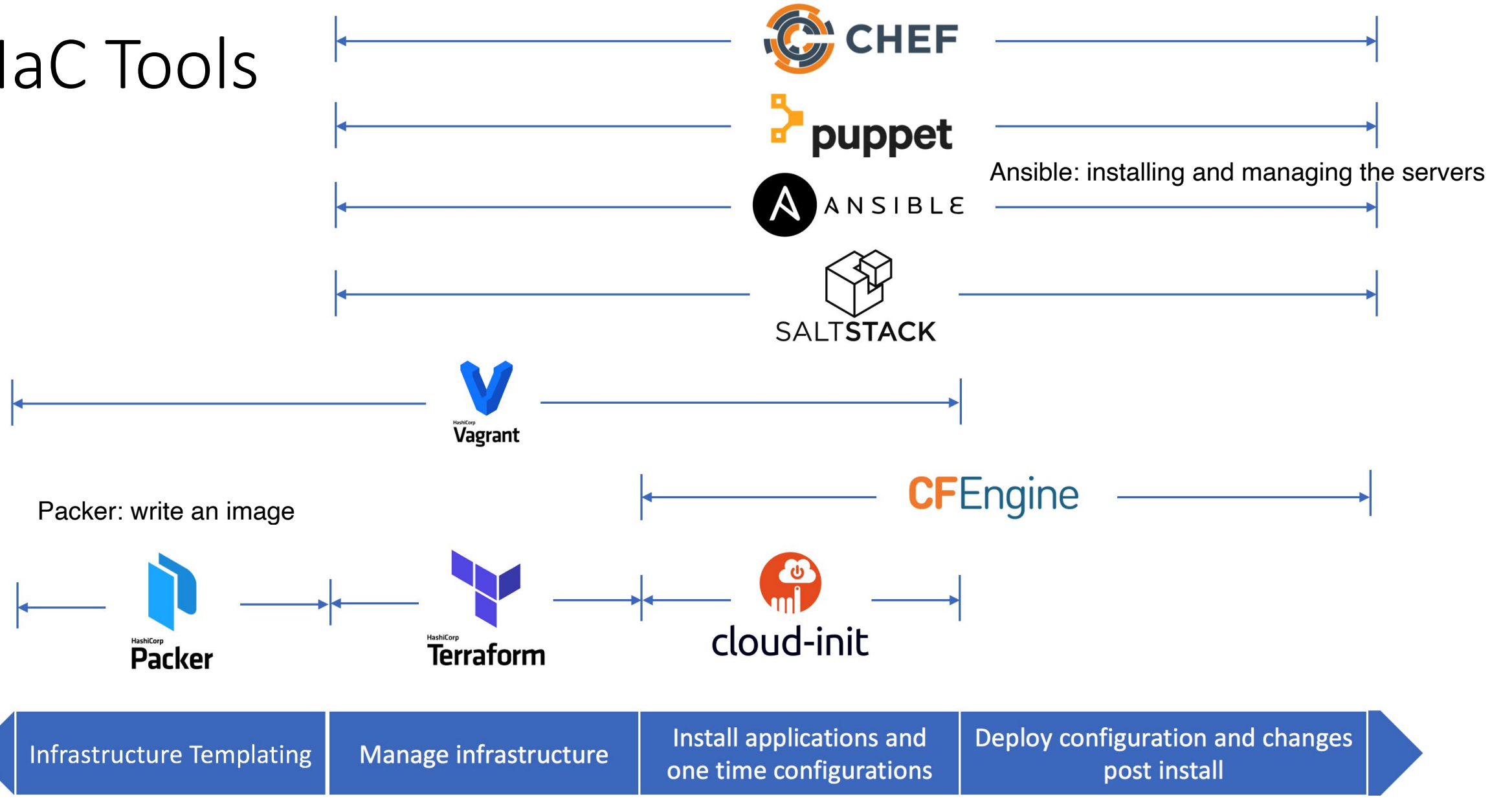
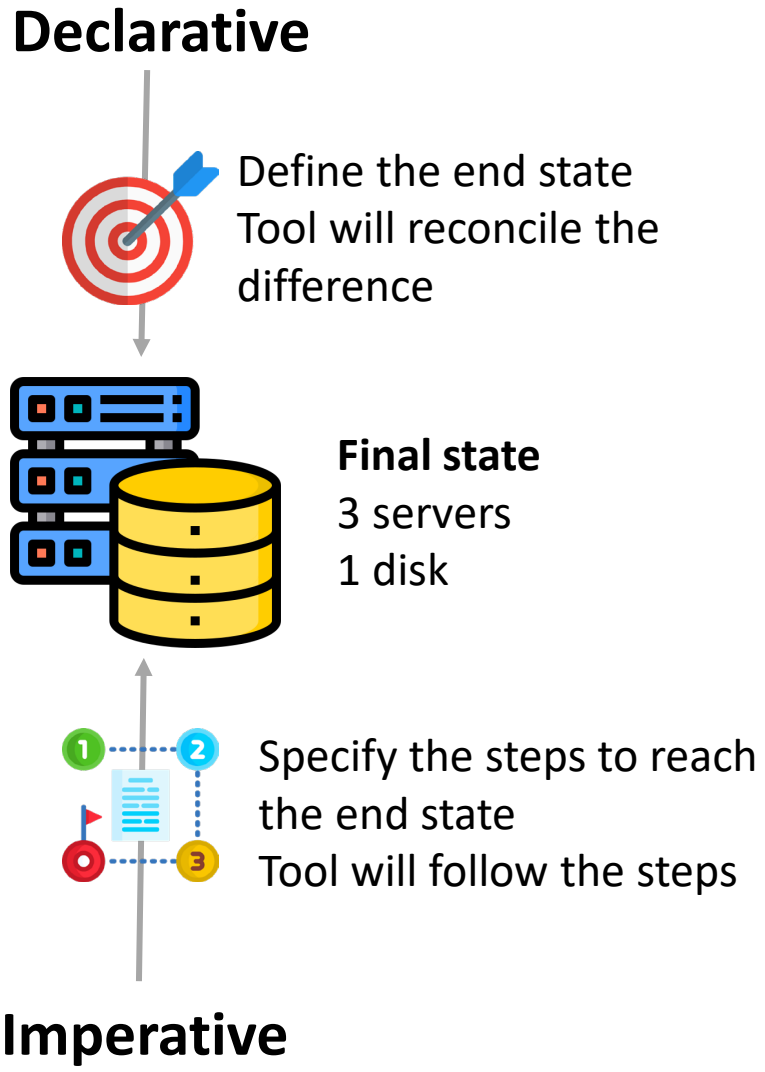



Image from https://medium.com/cloudnativeinfra/when-to-use-which-infrastructure-as-code-tool-665af289fbde

# IaC – Declarative vs Imperative

- Two ways of writing IaC scripts
  - Declarative – define the final state of the infrastructure, tool will determine how best to arrive at the final state from the present state
  - Imperative – specify the steps to arrive at the final state, the tool will follow the steps to provision the required resources
- Scripts may be written in
  - Domain specific language (DSL)
    - Eg. Terraform, Ansible, Kubernetes
  - General purpose programming language
    - Eg. Pulumi, AWS CDK in Golang, Java, JavaScript, Python, etc



# Example: IaC with Programming Language

```
import * as awsx from "@pulumi/awsx";
import * as pulumi from "@pulumi/pulumi";

// Create an elastic network listener to listen for requests
// and route them to the container.
const listener = new awsx.elasticloadbalancingv2.NetworkListener("nginx", { port: 80 });

// Define the service to run. Pass the listener to hook up the network load balancer
// to the containers the service will launch.
const service = new awsx.ecs.FargateService("nginx", {
  desiredCount: 3,
  taskDefinitionArgs: {
    containers: {
      nginx: {
        image: awsx.ecs.Image.fromPath("nginx", "./app"), memory: 512,
        portMappings: [listener] }
    }
  }
});
export let frontendURL = pulumi.interpolate `http://${listener.endpoint.hostname}/`;
```

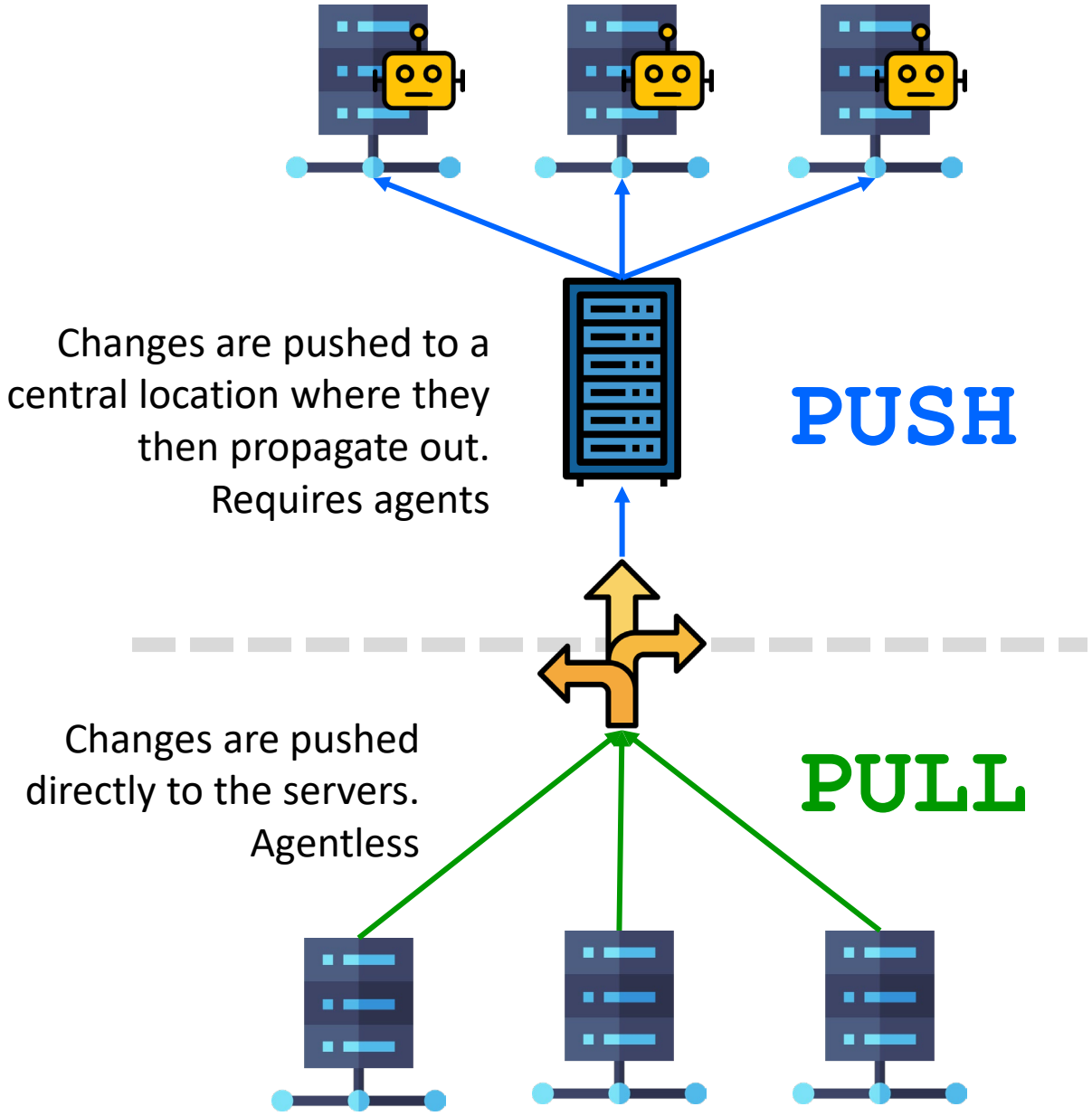
Example from https://github.com/pulumi/examples/blob/master/aws-ts-containers/index.ts

# Idempotent

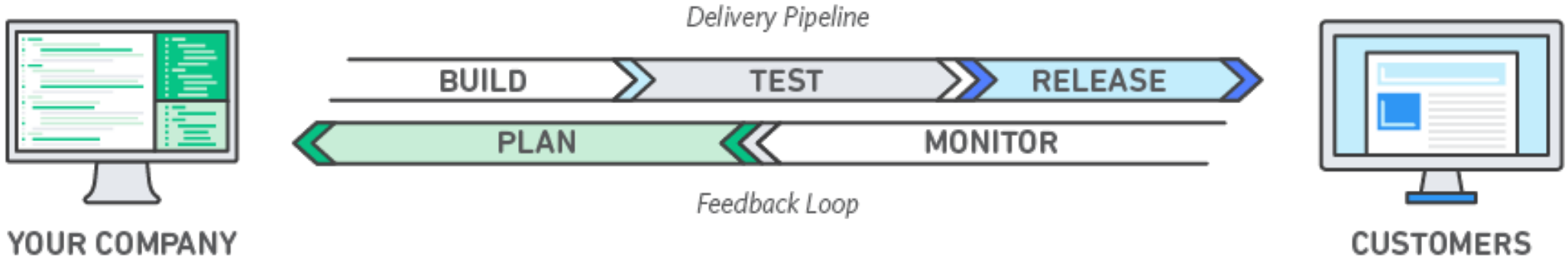
- Property of an operation where they can be applied many times without changing the beyond the initial application
- IaC should be idempotent
  - Will not deploy more resources when the final state has been reached
  - Some imperative operations may not be idempotent
- Tools required to maintain the state of the resources under its management
  - Compare the actual versus the desired
  - Can be a potential source of contention especially when working in a team

# Maintaining State

- Need some way to understand the state of the deployment for impotency
- State reconciliation is either done via push or pull
- States can be shared across a team
  - Some tools supports state locking
  - Prevent multiple updates that can corrupt the state



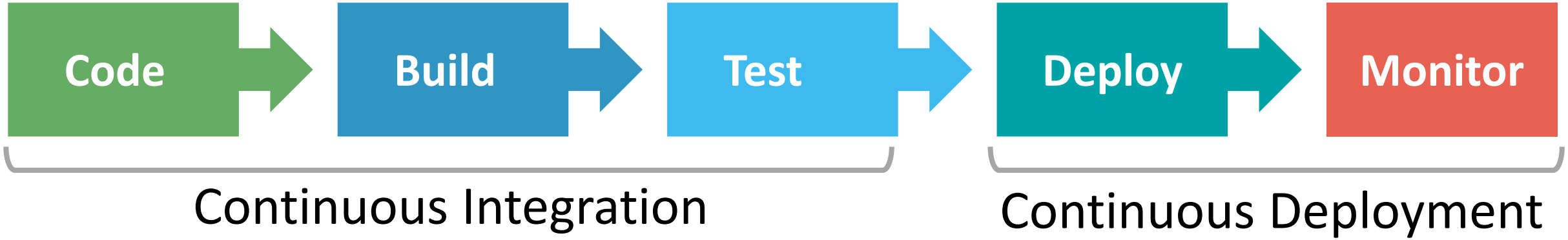
# What is DevOps?




- DevOps is a combination of culture, practices and tools
- Aim is to shorten the systems development lifecycle
  - By streamlining software building, testing and release
- Benefit is the improve and evolve applications at a faster pace
- Automation is they key to the fast feedback/delivery loop
  - Key tool is CICD

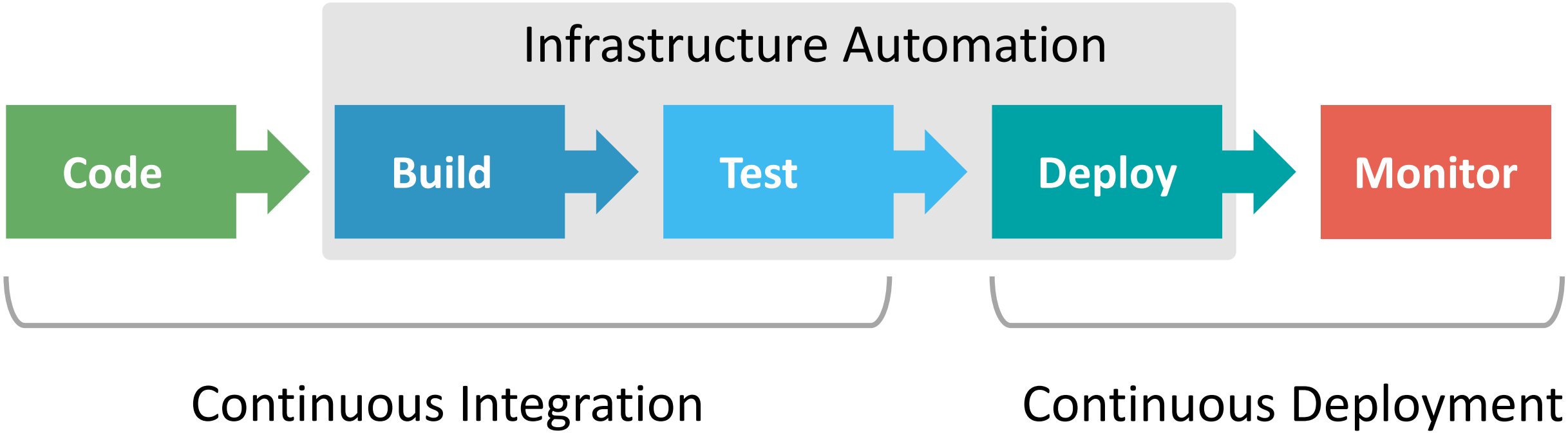
Image from https://aws.amazon.com/devops/what-is-devops/

# What is CI/CD?




- Refers to the practice of continually building and delivering an application
  - 2 phase process
- Continuous Integration where modules that make up an application are continuously tested and integrated
- Continuous Delivery is the practice of ensuring that the code is always at a deployable state
- Leverage tools to automate the integration and deployment

# Infrastructure as Code in CI/CD Pipeline




# Appendix

# Testing

- Software testing focus on the application
  - Unit, stress, smoke, end-to-end
  - Testing known conditions
  - Test if the application conforms the requirements and data flow
- What about unknown or unforeseen errors in infrastructure, application, processes
  - Eg misconfigured database cluster
  - Eg. the application itself is down due to hardware failure
  - Eg. data corruption
- ‘Regular’ testing do not test these scenarios

# What is Chaos Engineering?

- Chaos engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production.
  - From https://principlesofchaos.org
- Idea is to break your system on purpose at random time during production
  - Inject fault into the system like vaccine, inject into body to build immunity
  - Observe how the infrastructure, application, process and people react and respond
- Reveal readiness and build confidence in handling crisis

# Build Resilient System

**Software**
- Memory leaks
- License
- Certificate expiration
- Incompatible version

**Infrastructure**
- Redundancy
- Self healing
- Isolation
- Infrastructure as Code

**Application**
- Request timeouts
- Data corruption
- Rate limiting
- Exception handling

**Operations**
- Logging
- Monitoring and observability
- Incident response
- Metrics

Adapted from https://speakerdeck.com/adhorn/applying-chaos-engineering-principles-for-building-fault-tolerant-applications

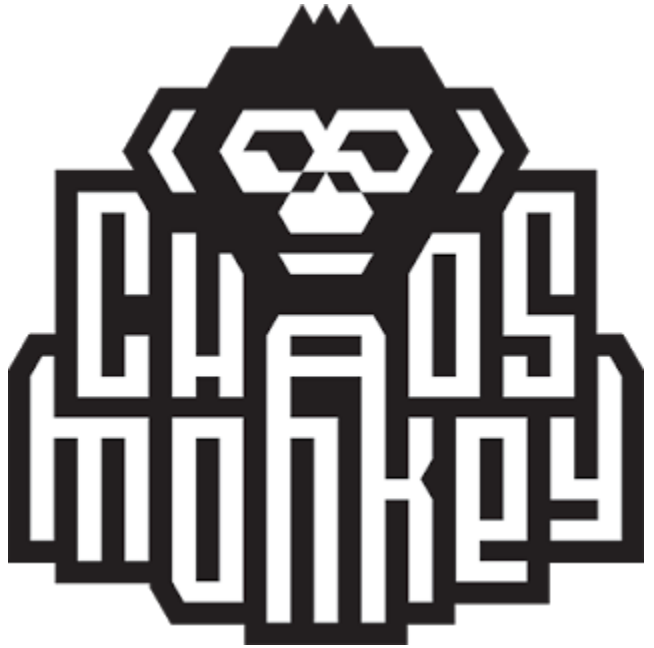
# DYI Simple Fault Injection

- Stop the database
  - `sudo systemctl stop mysql`
- DDoS your application
  - `siege -c 100 https://myapp.com`
- Max out your system resources
  - `stress-ng --cpu 8 --vm 4`
- Randomly drop packets on network interface
  - `tc qdisc add dev eth0 root netem loss 20%`
- Block access to DNS
  - `iptable -A INPUT -p tcp -m tcp --dport 53 -j DROP`

Adapted from https://speakerdeck.com/adhorn/applying-chaos-engineering-principles-for-building-fault-tolerant-applications

# Chaos Engineering Toolkit

- Chaos Monkey is a set of tools to implement chaos engineering
  - Developed by Netflix
  - https://github.com/netflix/chaosmonkey
- Set of agents that performs the following
  - Shuts down services randomly
  - Slows down performance
  - Can be scheduled
- Deployable on Kubernetes
  - Other options: Kubernetes ‘native’ - Chaos Mesh



# Infrastructure Elements as Code

- Server image definition
  - Operating system
- Software packages to be applied to server
  - What additional packages to be installed on a generic server
- Application artefacts eg. binaries, sources, configurations
- Infrastructure stack
  - Collection of servers, networks, API gateways
- Operation services configurations
  - Firewall rules, API gateway healthchecks,
- CI/CD pipeline definitions
  - Include testing and validation

# Idempotent

- Property of an operation where they can be applied many times without changing the beyond the initial application
- Most IaC are idempotent
  - Will not deploy more resources when the final state has been reached
  - Some imperative operations may not be idempotent
- Tools required to maintain the state of the resources under its management
  - Compare the actual versus the desired
  - Can be a potential source of contention especially when working in a team