Infrastructure as Code

Time to start automating things

Why IaC?

In order to talk about Infrastructure as Code (IaC) we first need to discuss what it is and what problems it solves.

There are tons of definitions for Infrastructure as Code:

- The process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools
- The management of infrastructure in a descriptive model, using the same versioning as DevOps team uses for source code
- Applying software engineering tools and practices to infrastructure

Why IaC?

Thusfar in this course, we have taken a look at the classic approach to infrastructure, starting with hardware and moving to VMs that have all been manually configured.

Now it’s time to change that.

Many of the companies that you’ll be working for are at this stage, are transitioning some or all of their infrastructure to IaC, or have recently transitioned some or all of their infrastructure to IaC?

Why?

Speed

- Get something out to market quickly
- Iterate on it once released
- Continue to improve it and scale as necessary (Agile!)
- Daily or weekly deployments into production

Cloud IaaS

- Cloud Infrastructure (Be it Public Cloud or Private Cloud) enables faster deployment and turnaround time
- Lowers barriers for automation and making changes
- Drastically reduces lead time for new projects to get spun up

What problems does this introduce?

- Performance
- Stability
- Maintainability
- Compliance
- Security
- Budget

What is/isn’t IaC?

- While IaC and automation now relies heavily on the Cloud, it doesn’t mean that everything in the Cloud uses IaC
- IaC shouldn’t be just a buzzword
- IaC isn’t limited to one technology (i.e. Terraform, Ansible, Chef, Puppet, CloudFormation, MaaS, etc)

- IaC is simply an automated way to deploy infrastructure which treats it like software

What does it look like without IaC

Before IaC the infrastructure world was pretty much as you've experienced it thusfar in this class

- Snowflake servers/deployments
  - Someone's pet server/deployment - one of a kind (even if you are setting up 100s of the same deployment, no two are identical)
- Manual deployment
  - Someone has to set up the deployment manually, be it on bare metal, on a VM, etc
- Undocumented Infrastructure (thankfully this isn't the case for you)
  - Most deployments had a 'bus factor' - if that person left (or encountered a bus) the knowledge regarding the server or deployment was lost
- IT SLAs, Change Windows, and CABs
  - Not all of these go away with IaC (sorry to inform you) - but they should hopefully be more streamlined and less tedious

Developers

Traditionally developers and operators are broken into two teams, with developers:

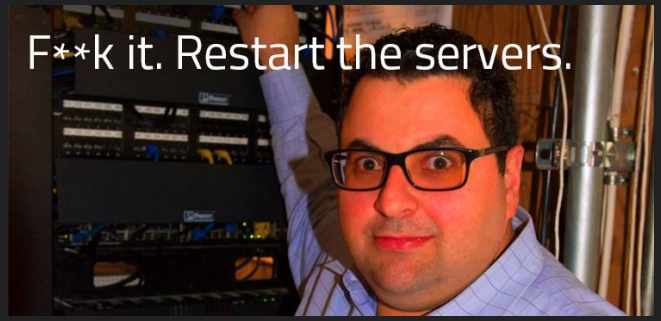
- Taking user requirements
- Writing Code
- Writing Tests (If you're lucky)
- Manual Testing/Demos
- Refactoring
- Releasing - Works on my machine
-

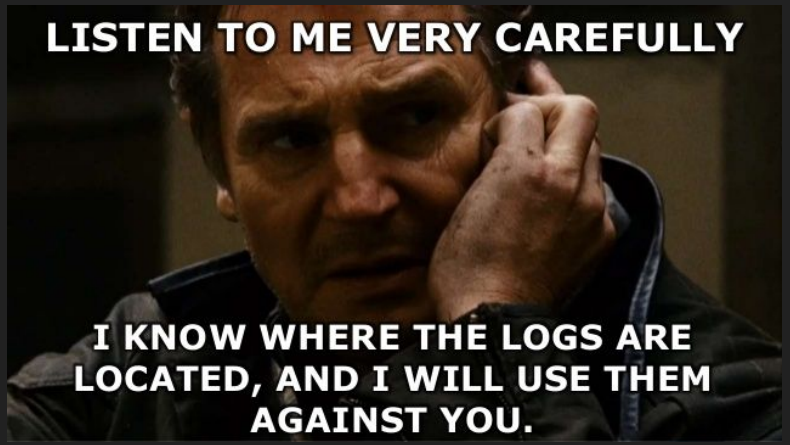
And then they chuck the 'finished' software over the wall to the ops team to deploy it

Operations

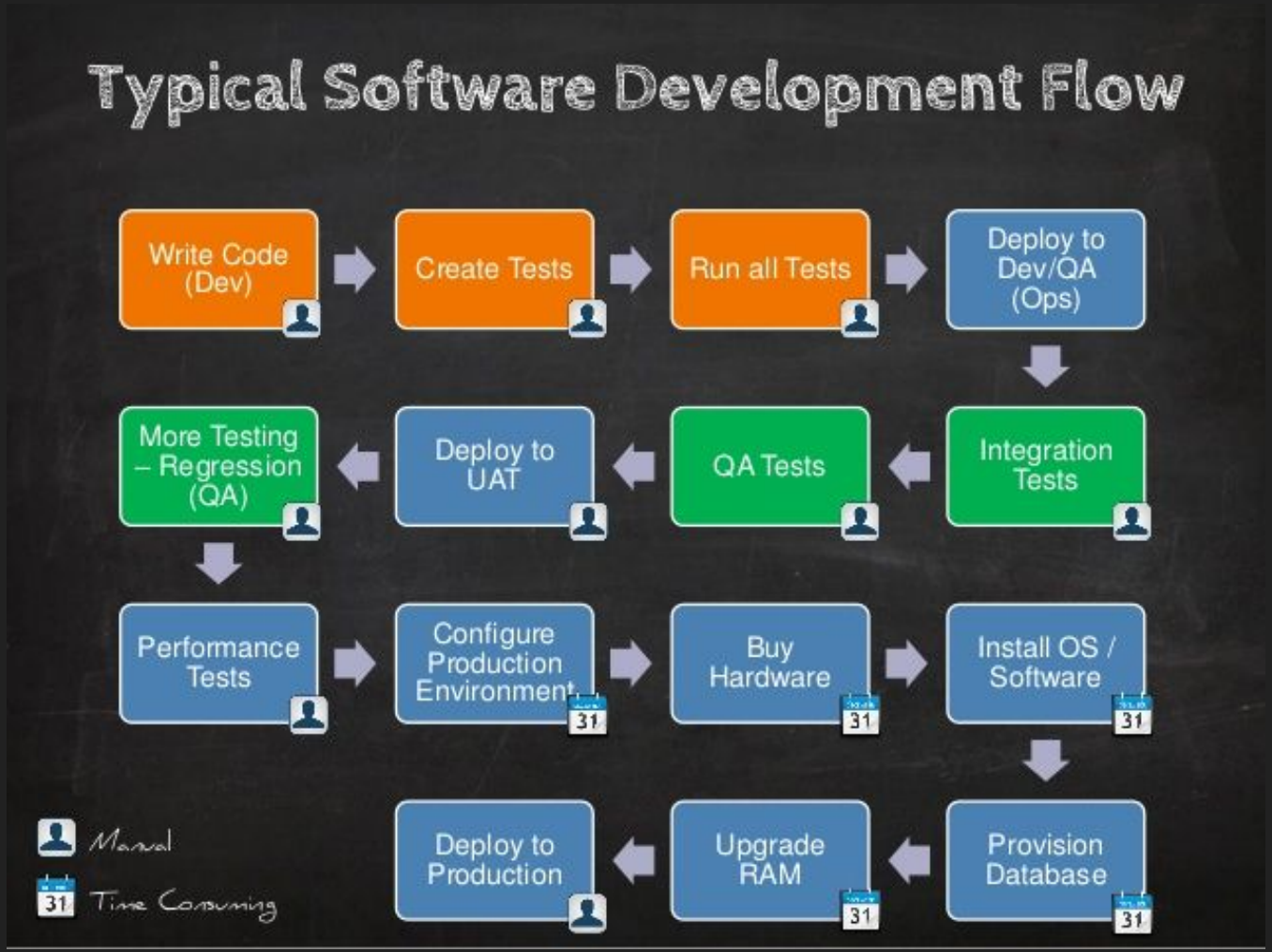
Operations traditionally picks up where developers drop off and take the released (and hopefully tested) software for deployment to infrastructure. So Ops teams typically handle:

- Server Administration
- Networking
- Database Administration
- Security
- Monitoring
- Maintenance and Upgrades





Typical Software Development Flow



IaC Goals

The goal of Infrastructure as Code are to be able to make changes and roll out deployments:

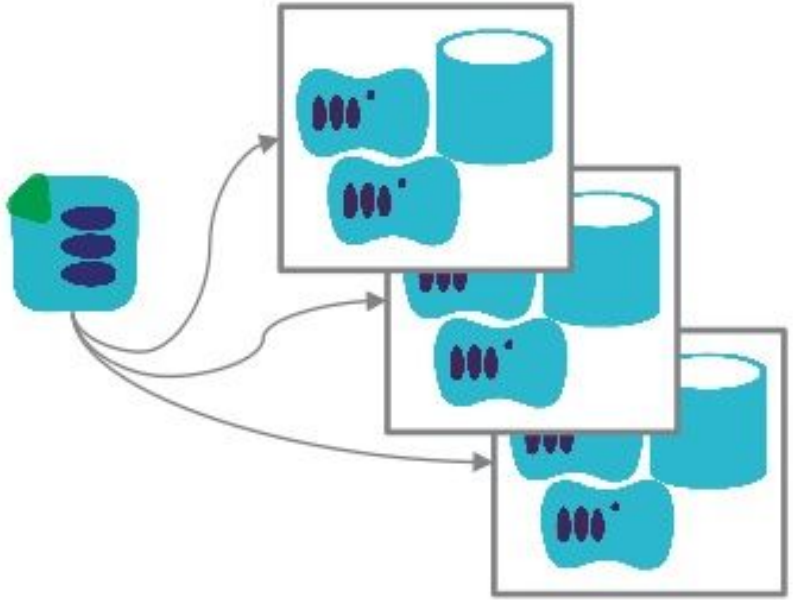
- Quickly
- Consistently
- Idempotently
- Responsibly
- Process Oriented

So how does IaC work in practice?

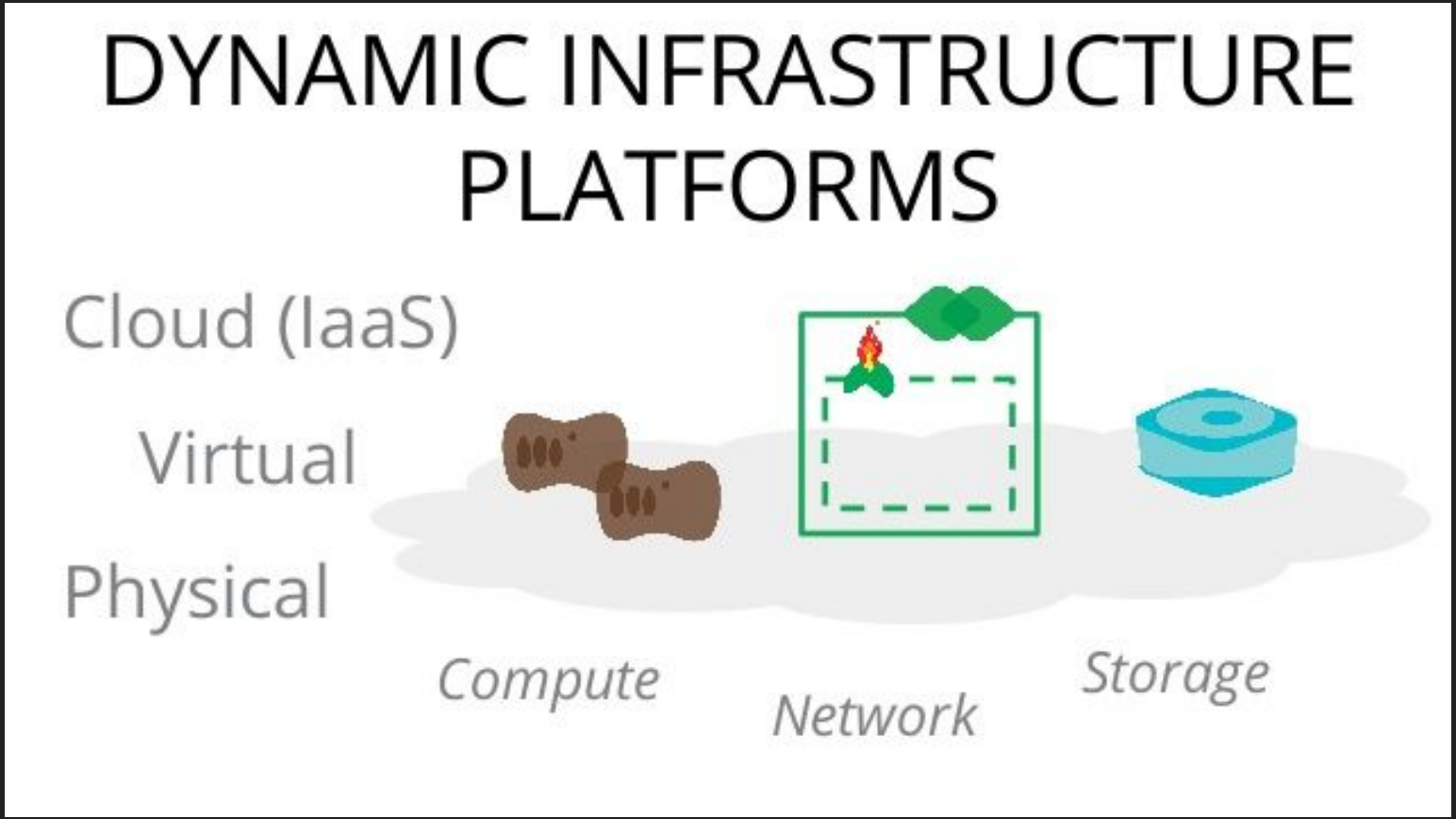
DEFINE SYSTEMS AS CODE

System design is:

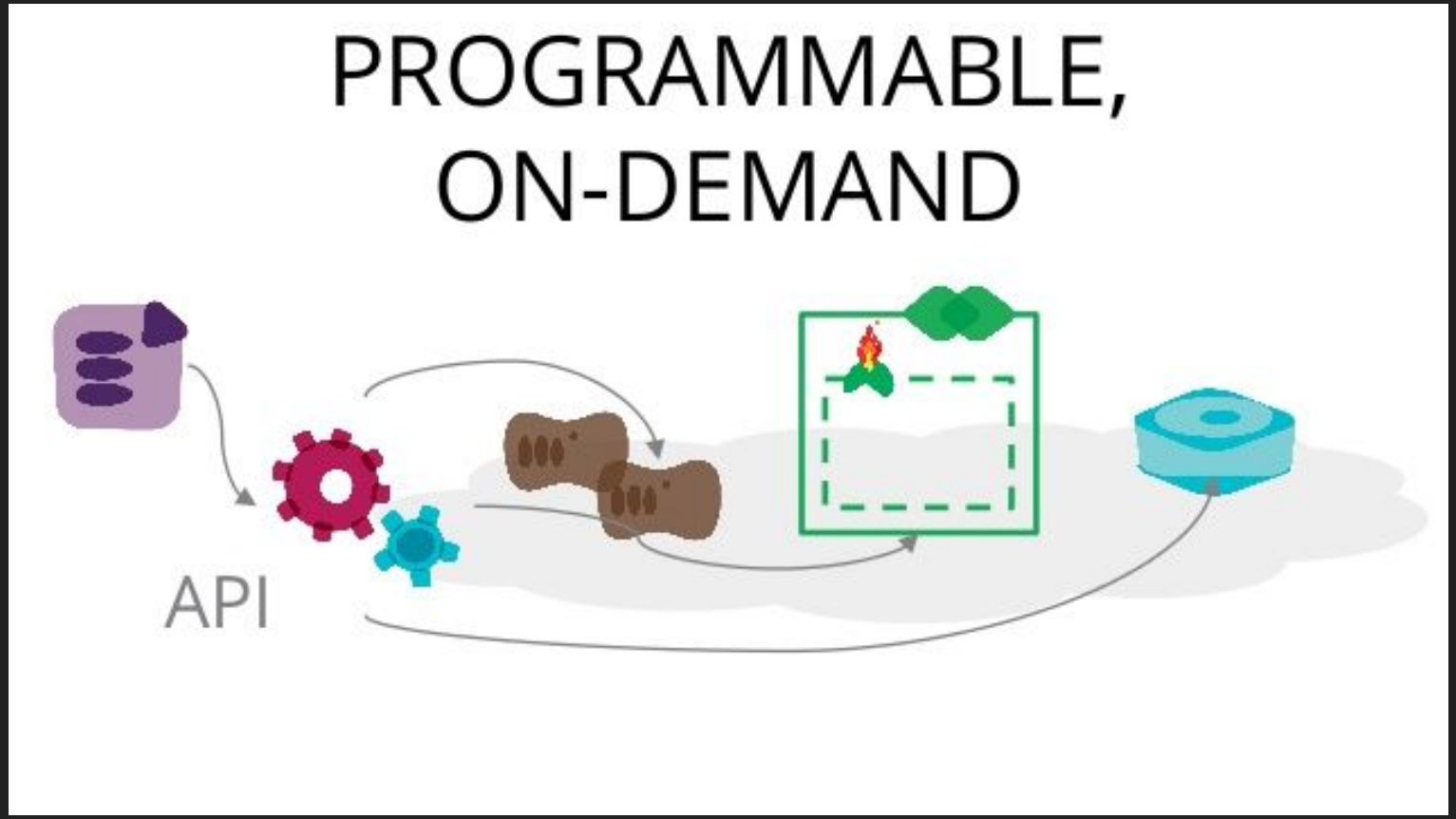
- Reusable
- Consistent
- Visible
- Versioned



So how does IaC work in practice?



So how does IaC work in practice?



So how does IaC work in practice?

So what tools do we have at our disposal to implement infrastructure as code?

- Definition Tools
  - Responsible for the creation of servers
  - Can perform limited configuration deployments
  - Configuration is often deployed via mostly preconfigured templates/vm images
- Configuration/Provisioning Tools
  - Used on existing servers
  - Often take over after definition tools create vms and install configuration agents
  - Used to manage and update configuration on long running servers
- Alerting/Logging/Monitoring
  - To be discussed in a future lecture

Definition Tools

Terraform, AWS CloudFormation, Azure Resource Manager, GCP Deployment Manager are all examples of Definition Toos

Terraform Example:

```
resource "aws_instance" "web" {
  ami = "ami-12345678"
  instance_type = "t1.micro"
  tags {
    Name = "HelloWorld"
  }
  security_groups = [ "${aws_security_group.my_security_group.id}" ]
}
```

Configuration Tools

Ansible, Chef, Puppet, and SaltStack are all examples of Configuration Tools

Ansible Example:

```
- hosts: server
  sudo: yes

  tasks:
  - name: install mysql-server
    apt: name=mysql-server state=present update_cache=yes

  - name: install ansible dependencies
    apt: name=python-mysqldb state=present

  - name: Ensure mysql is running
    service: name=mysql state=started
```

Configuration Tools

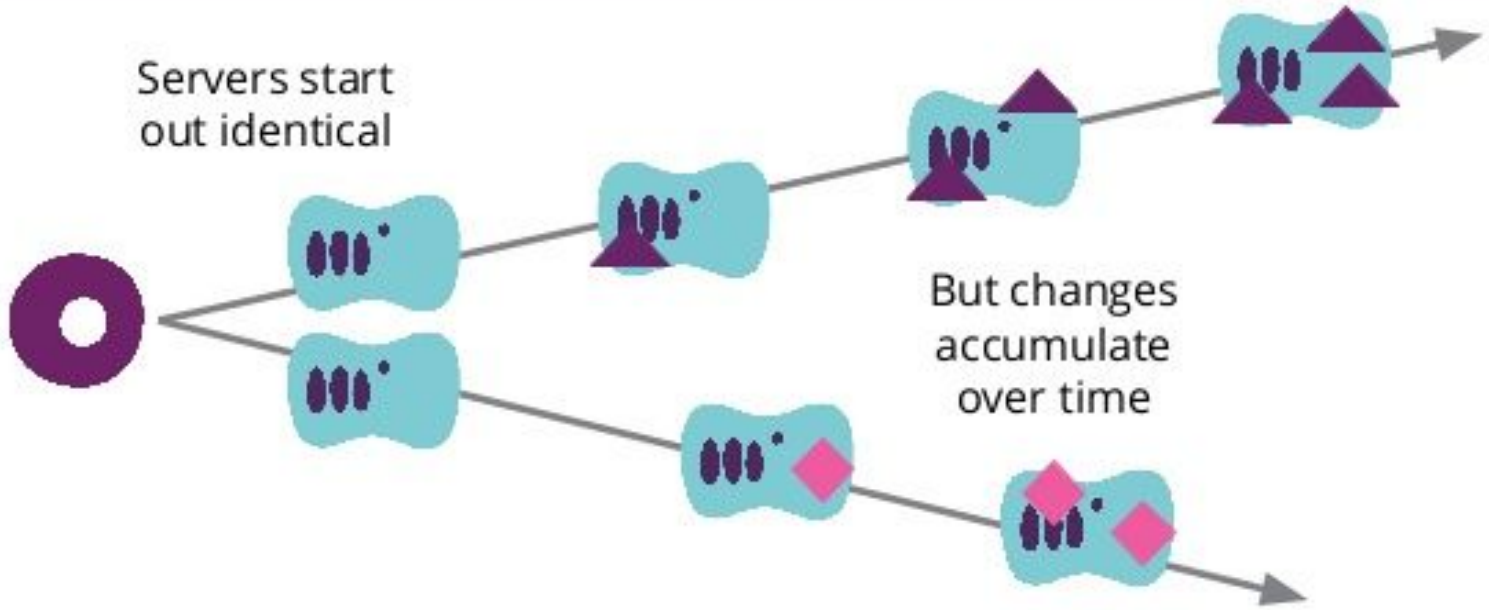
Chef Example:

```
mysql_service 'default' do
  port '3306'
  version '5.5'
  name 'localhost'
  initial_root_password 'p4ssw0rd'
  action [:create, :start]
end

mysql_config 'default' do
  source 'my.cnf.erb'
  notifies :restart, 'mysql_service[default]'
  action :create
end
```

Terraform Demo

CONFIGURATION DRIFT



AUTOMATION FEAR CYCLE

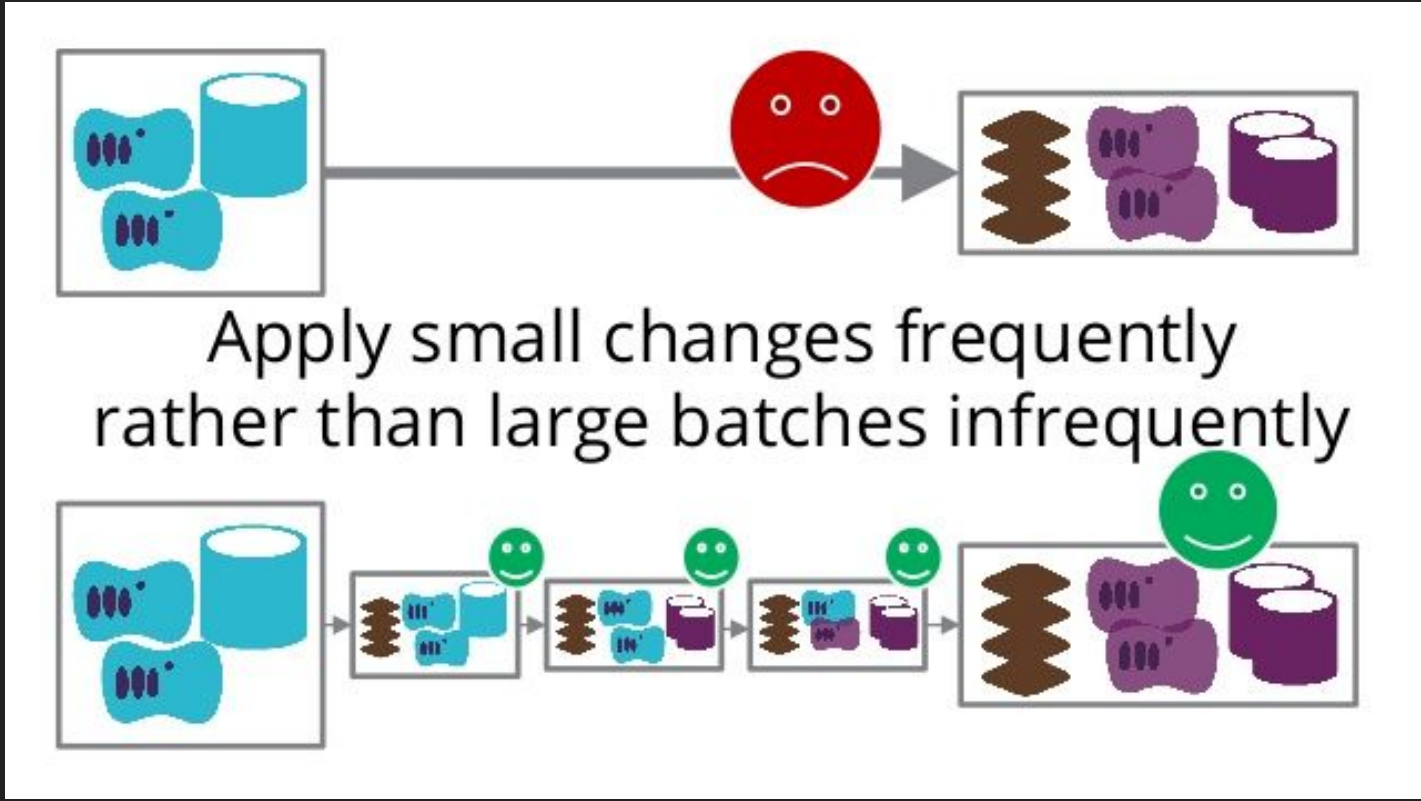
So how did we get into this predicament?

Automation Lag - the more time that has passed since the last time the automation deployed the more work is required to make it work smoothly again

So do we fix this?

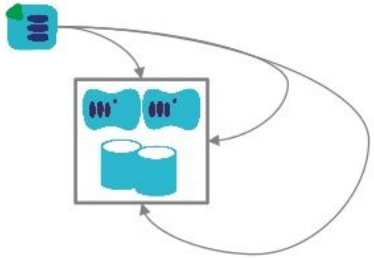
Automate the deployments on a regular basis and roll out small changes often instead of large sweeping changes infrequently

So how did we get into this predicament?

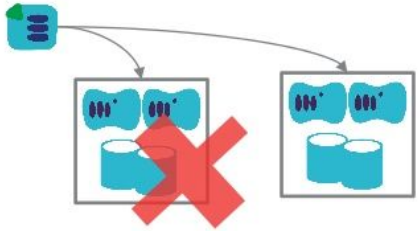


Automated Deployments

CONTINUOUSLY SYNCHRONIZE



OR CONTINUOUSLY REBUILD



Automated Deployments

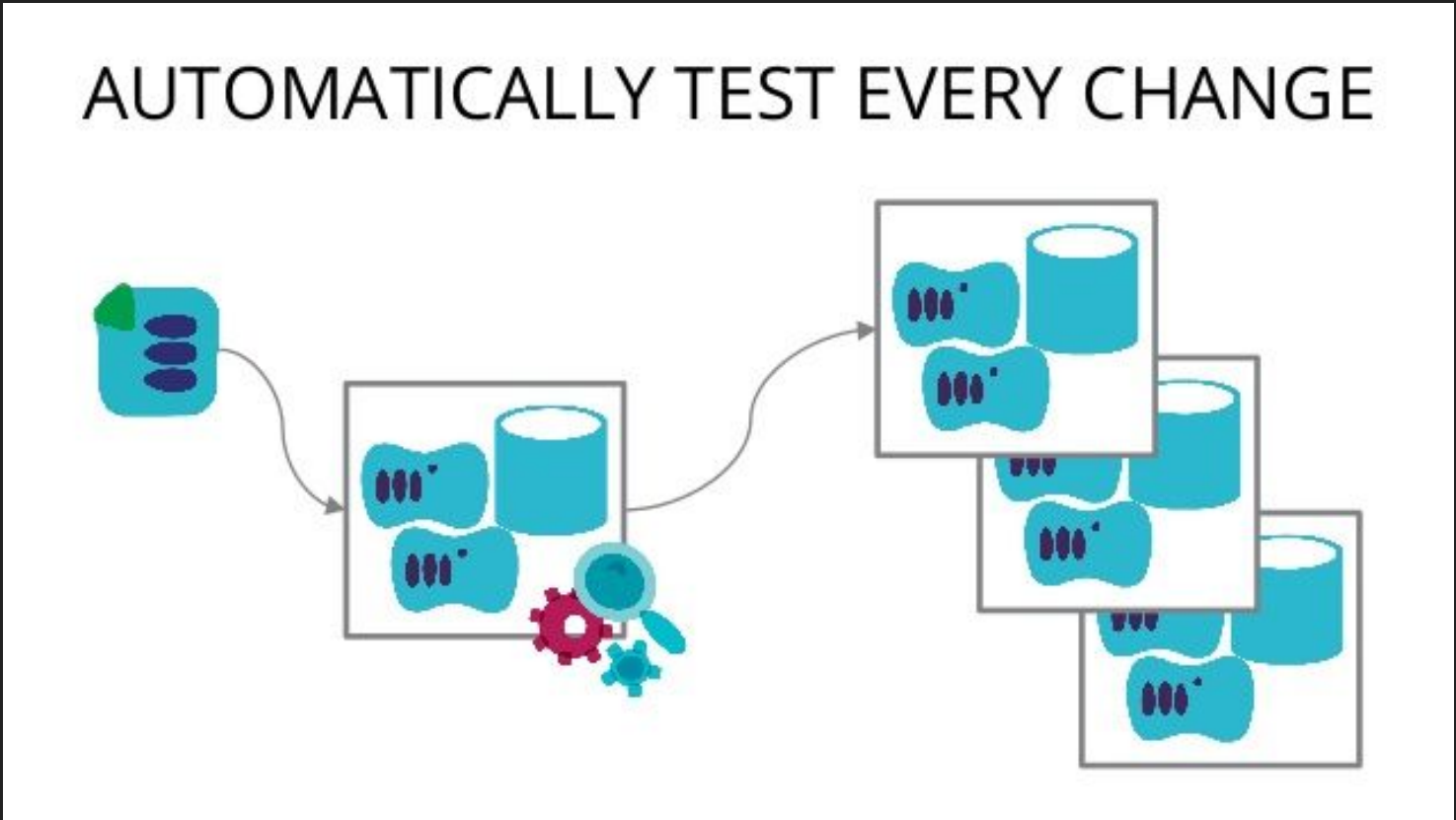
So now that we have automated deployments running daily, weekly, etc - What's the potential problem here?

How do we know if changes are 'ready' to be deployed?

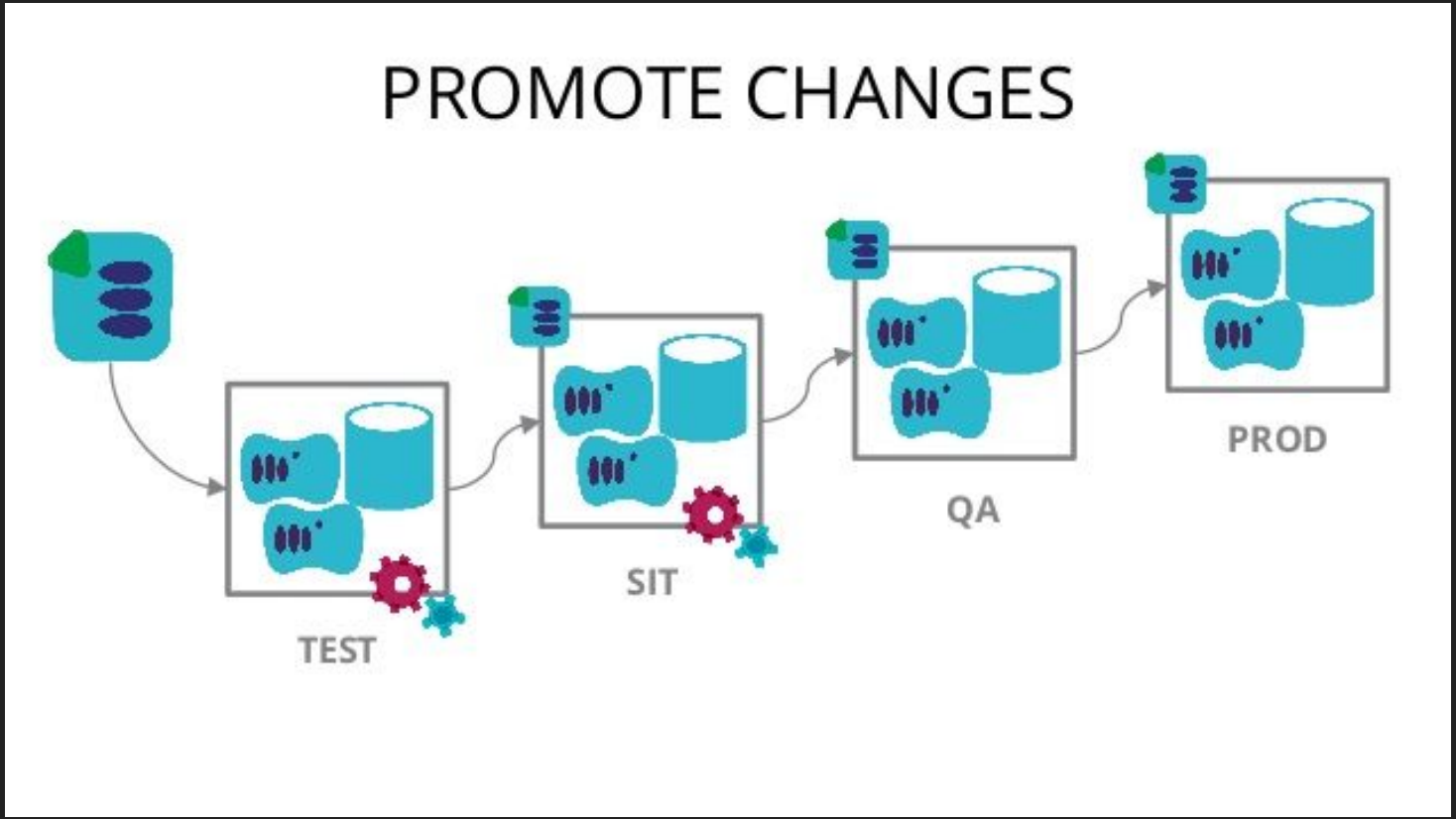
What if the automation starts breaking things?

How can we reduce the damage or 'blast radius' of our automated deployments?

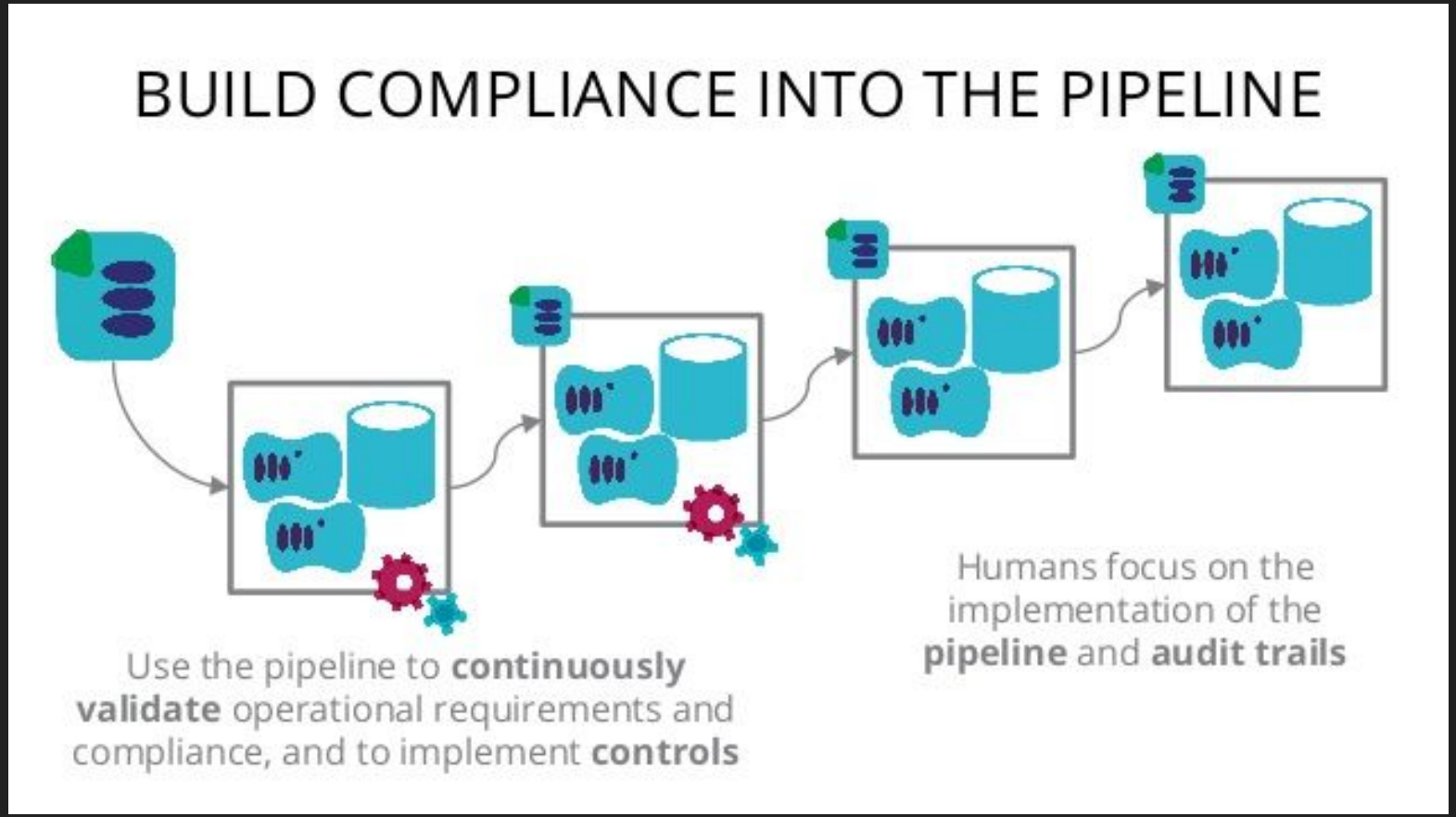
Automated Deployments



Automated Deployments



Automated Deployments



IaC Repo Structure

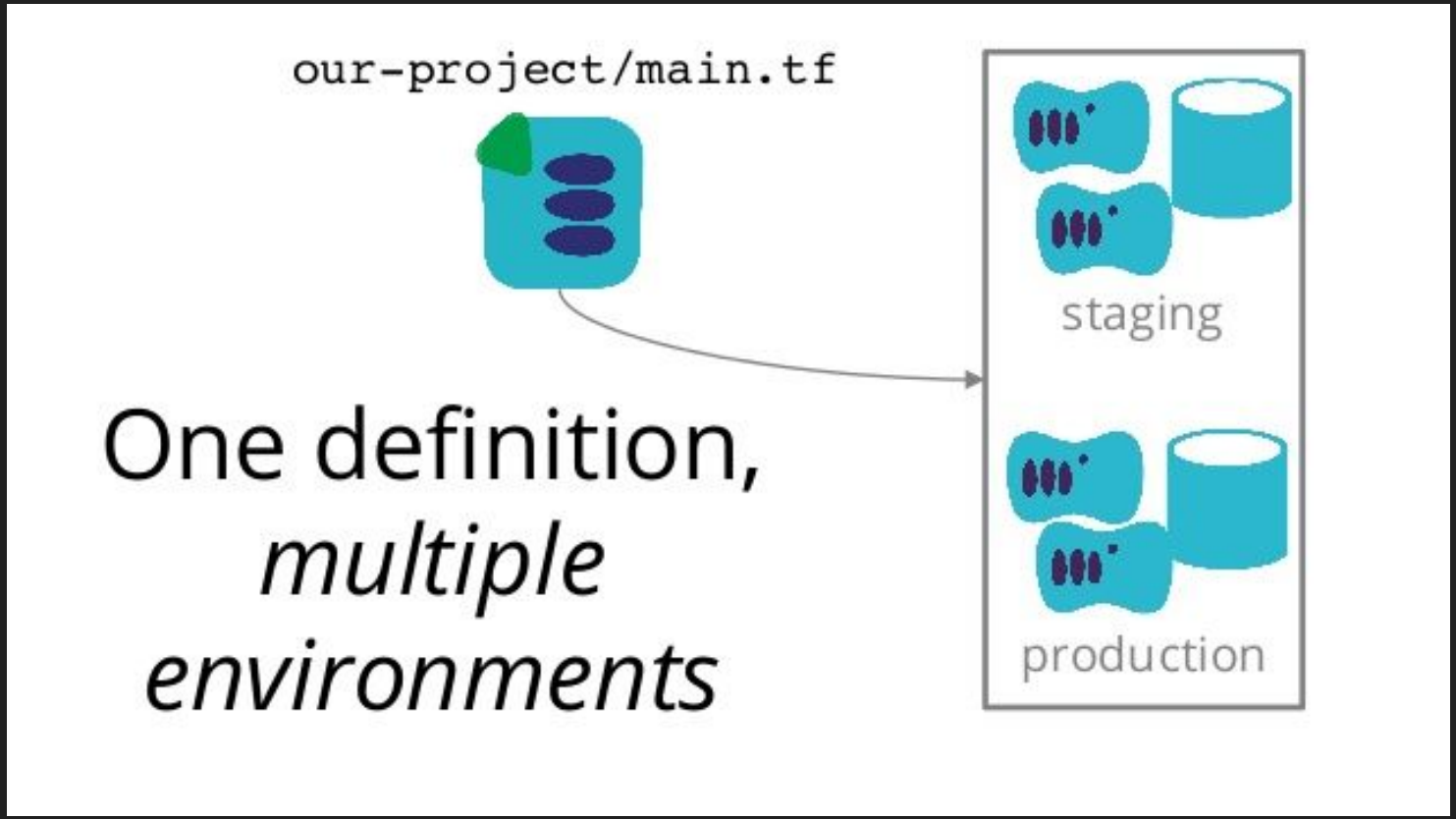
When considering how to structure your IaC repo, there are a few approaches all with differing tradeoffs

The goal should be to allow minimizing of risks for making small frequent changes and allow for the “pipelining” of change sets that we just discussed.

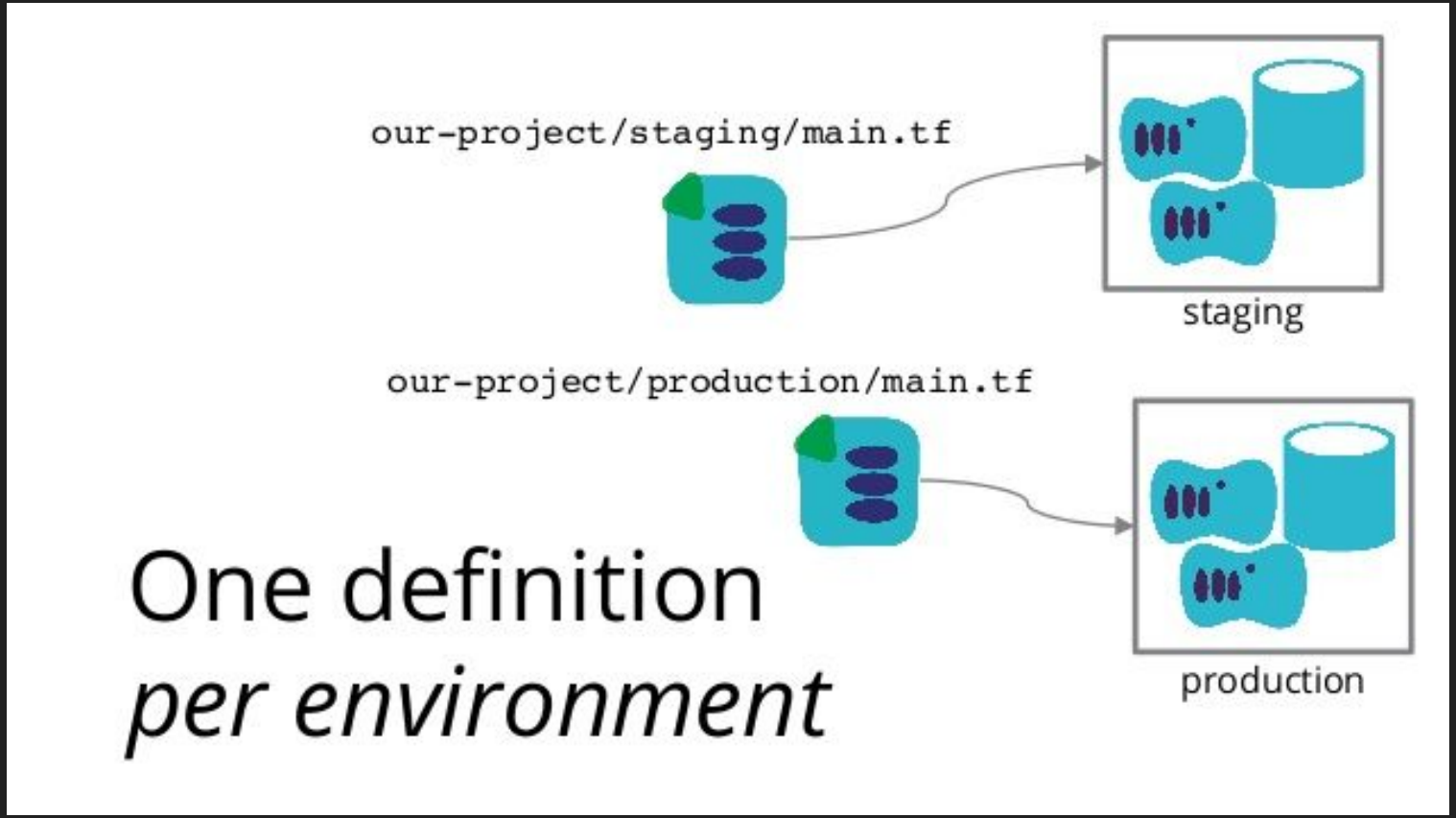
The usual approaches are:

- One definition - multiple environments
- One definition per environment
- One definition template - promoted across environments

IaC Design Patterns

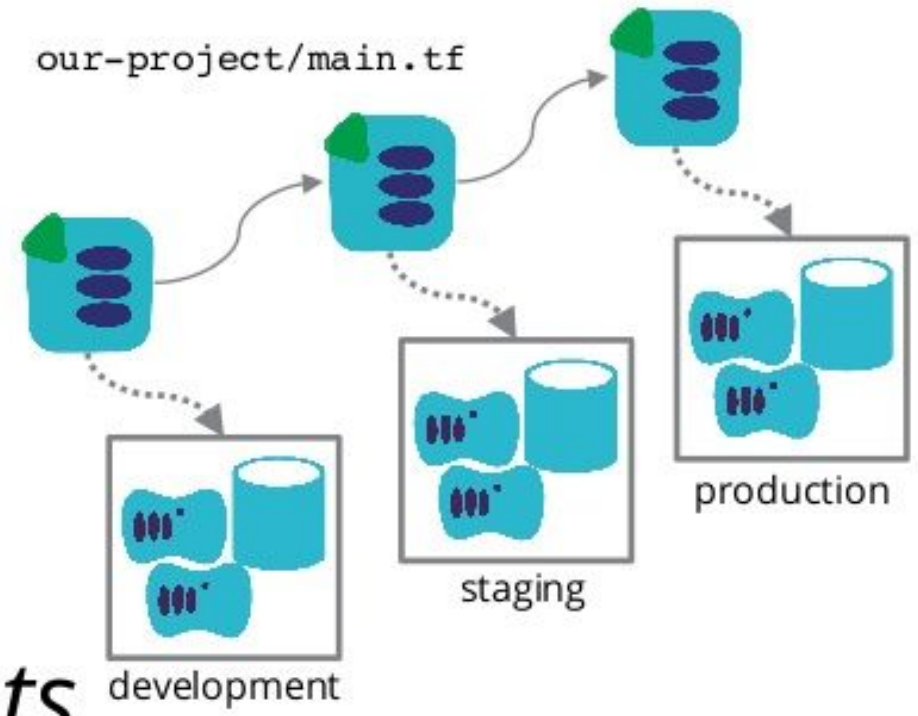


IaC Design Patterns



IaC Design Patterns

Single definition **template**, *promoted across environments*



IaC Testing

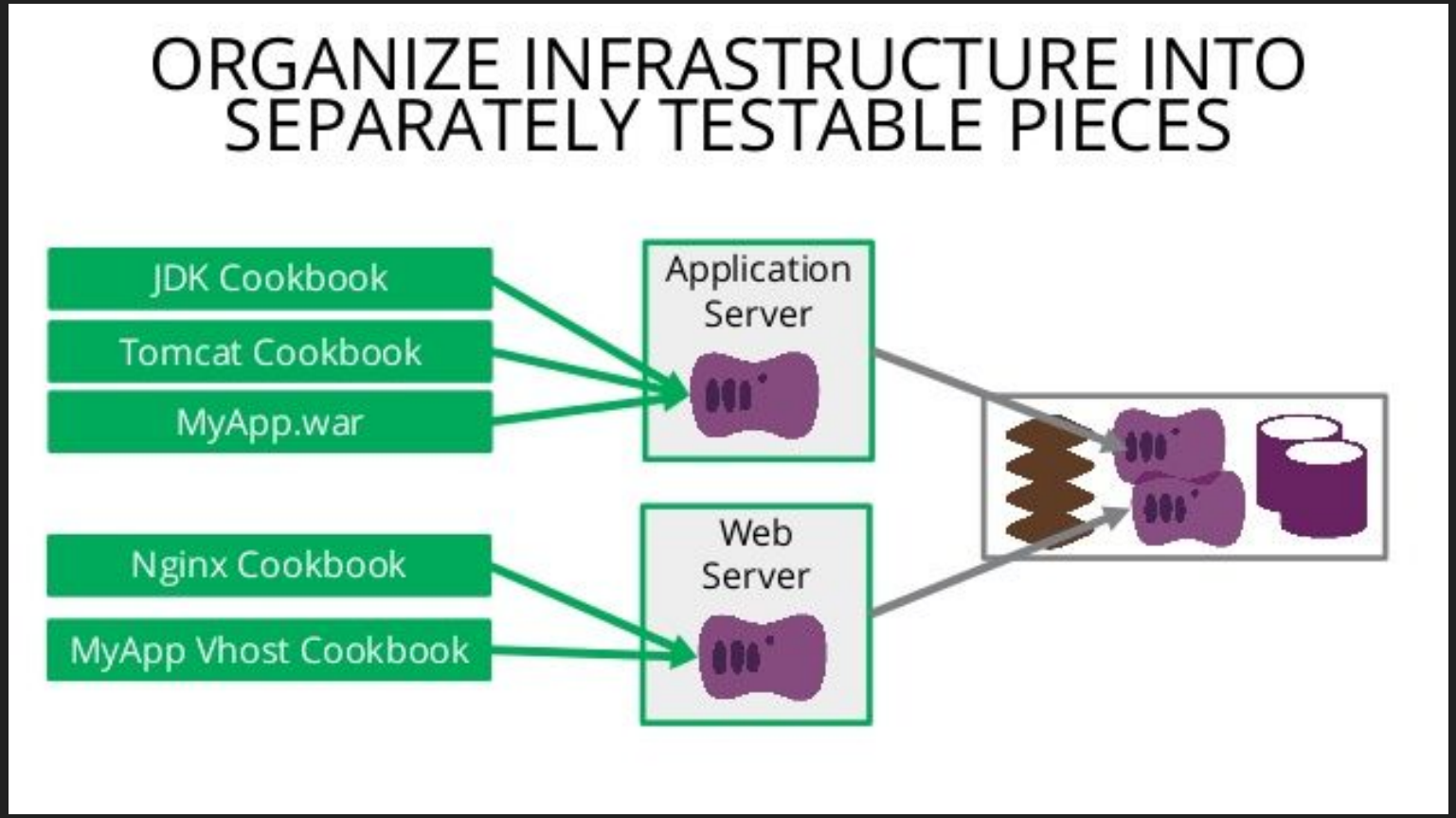
Now that we have a layout for our IaC repo, how do we perform testing and promote change sets?

- IaC testing is often super slow - need to wait for VMs to spin up, get configured, and often one component depends on other components

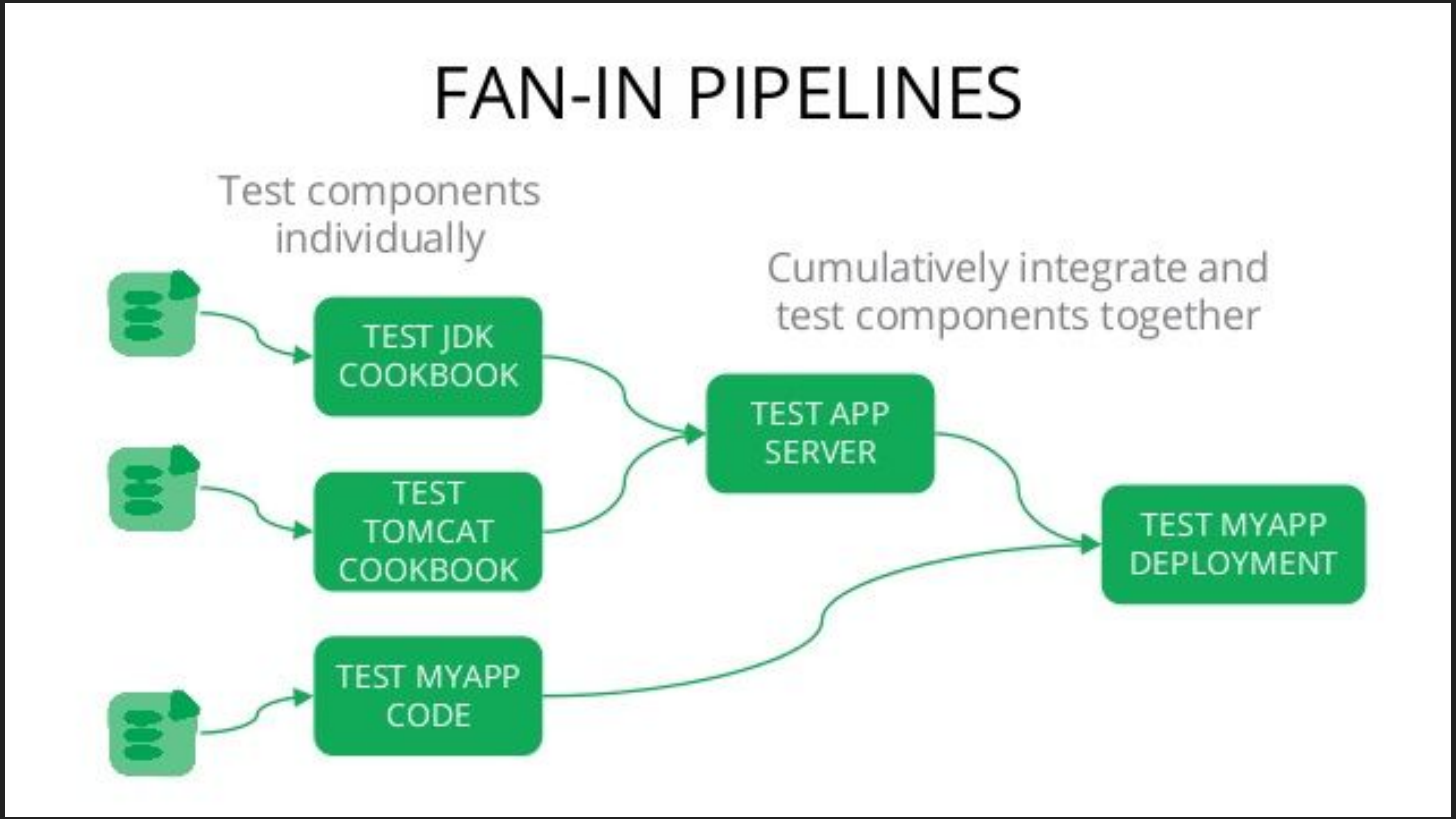
- Often services are interdependent and rely on other components that are already deployed

The trick is to split up the overall IaC into individually testable pieces and only after individual testing do you test with dependent components

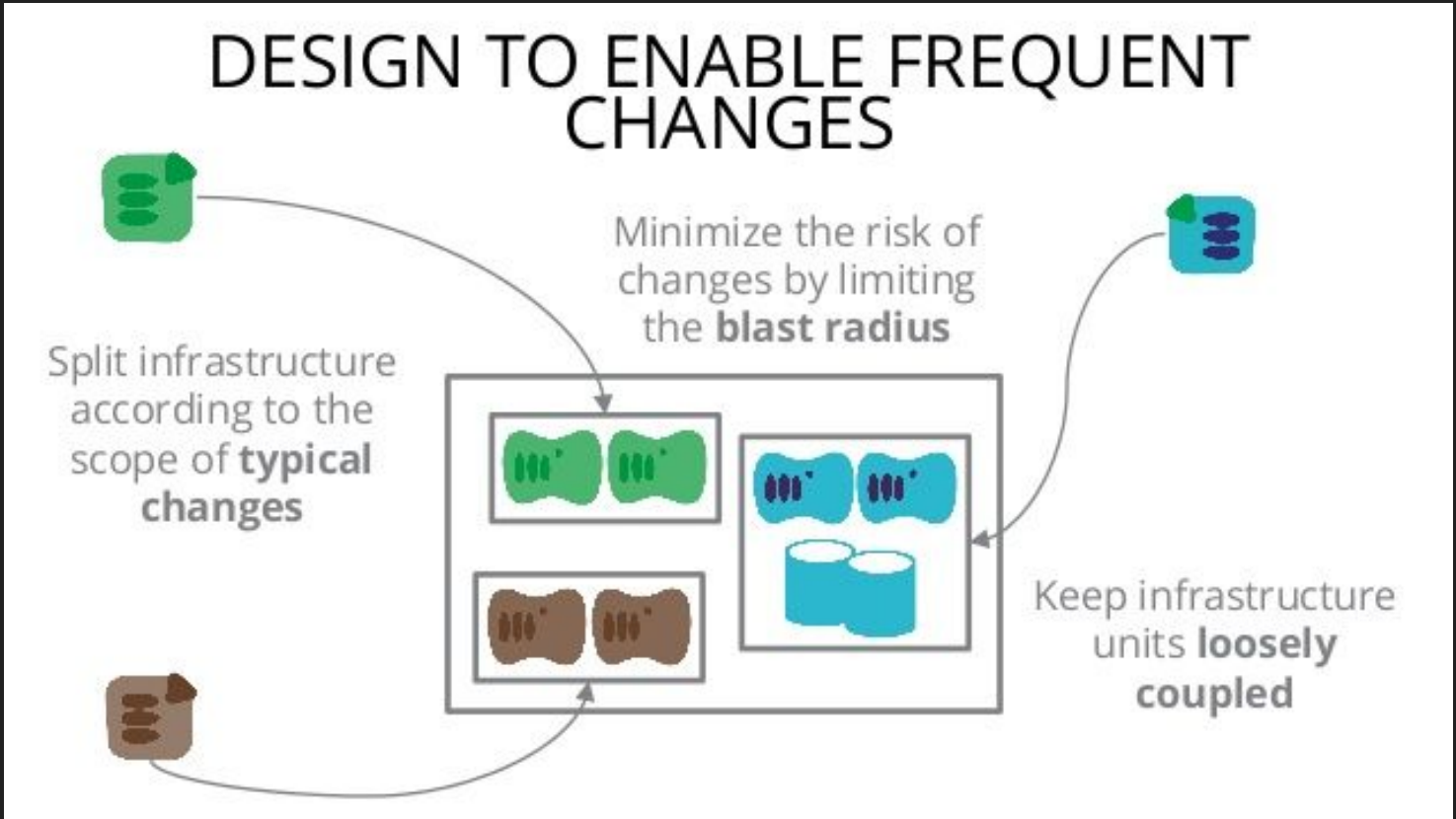
IaC Design Patterns



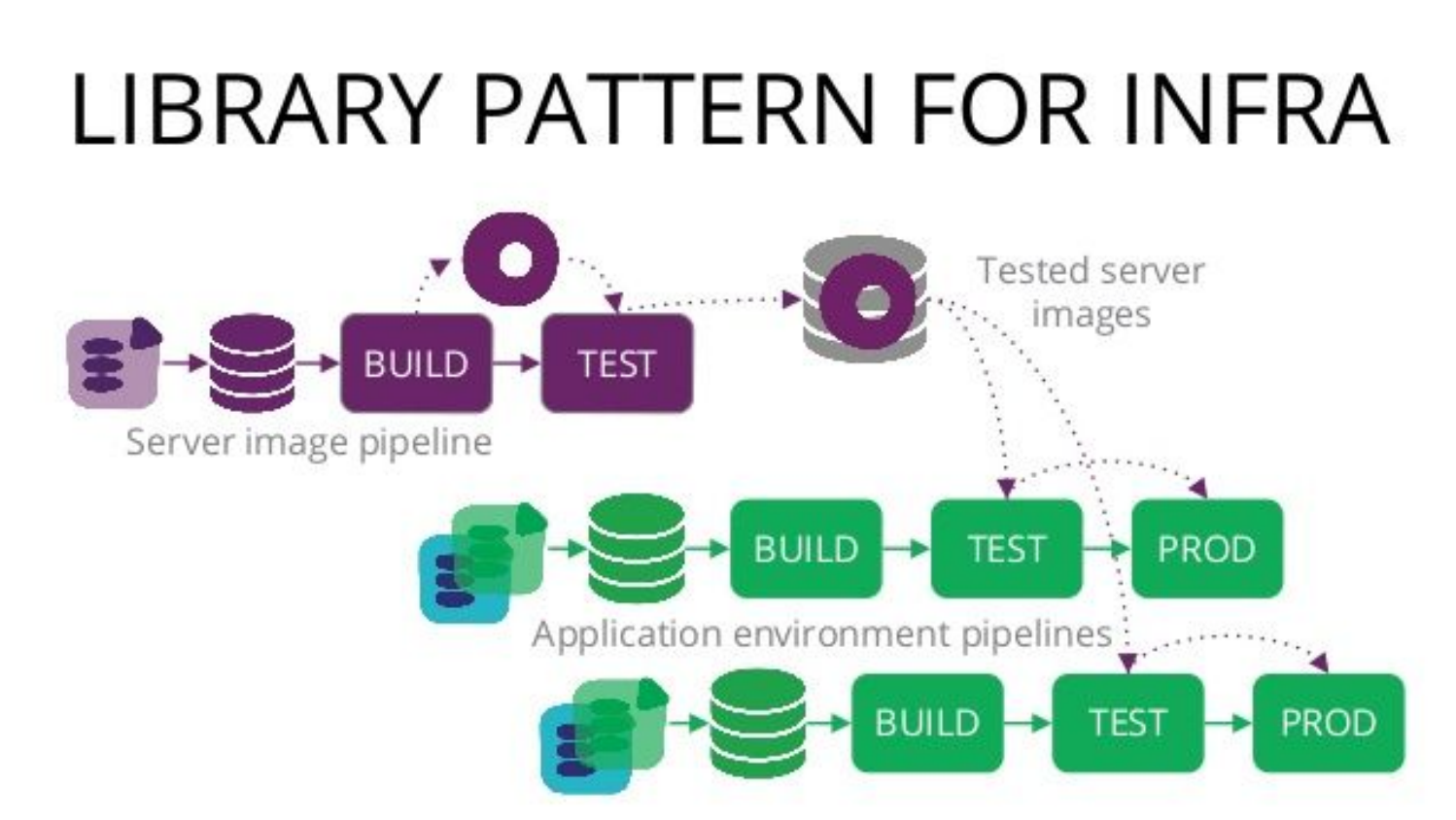
IaC Design Patterns



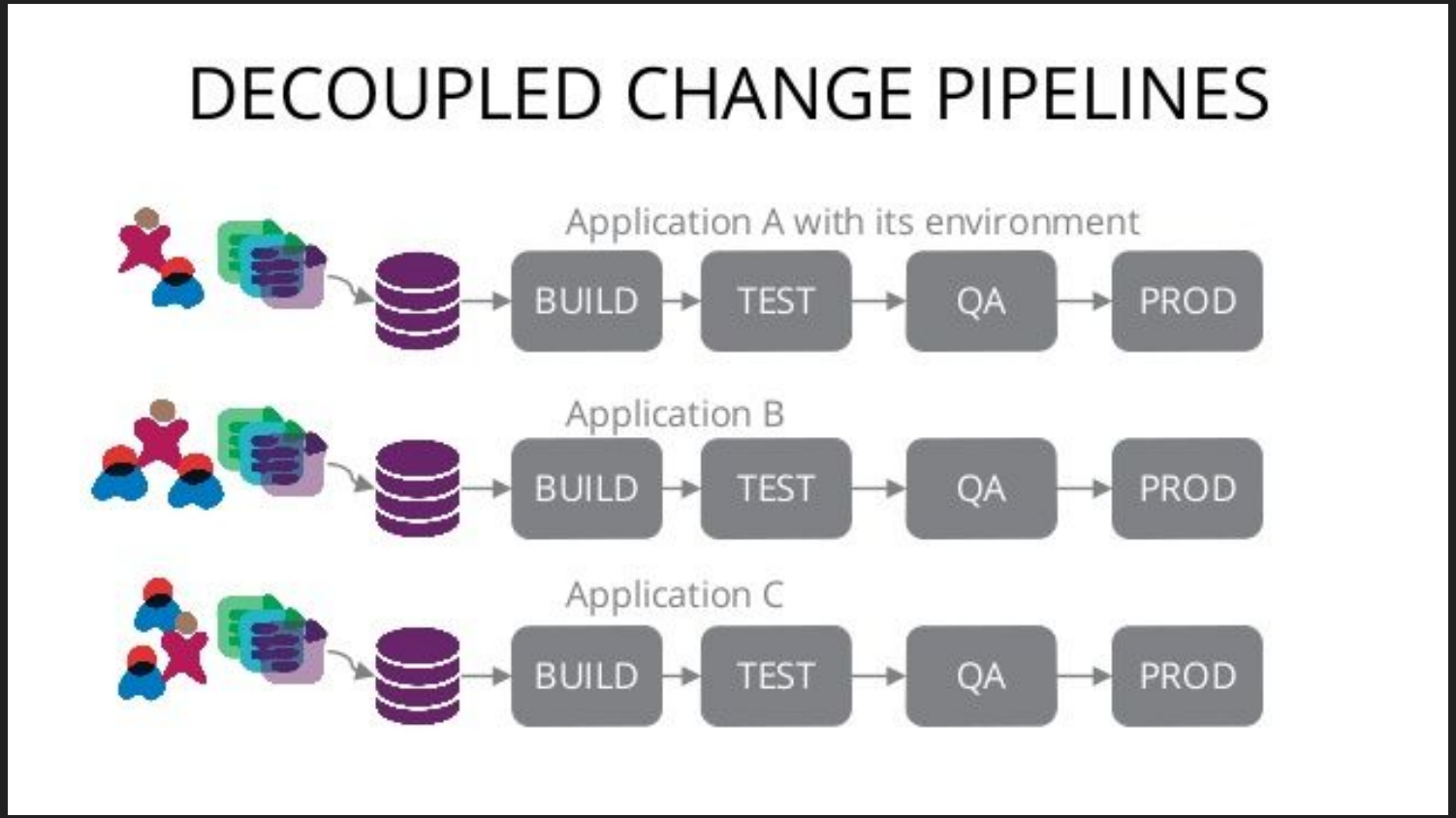
IaC Design Patterns



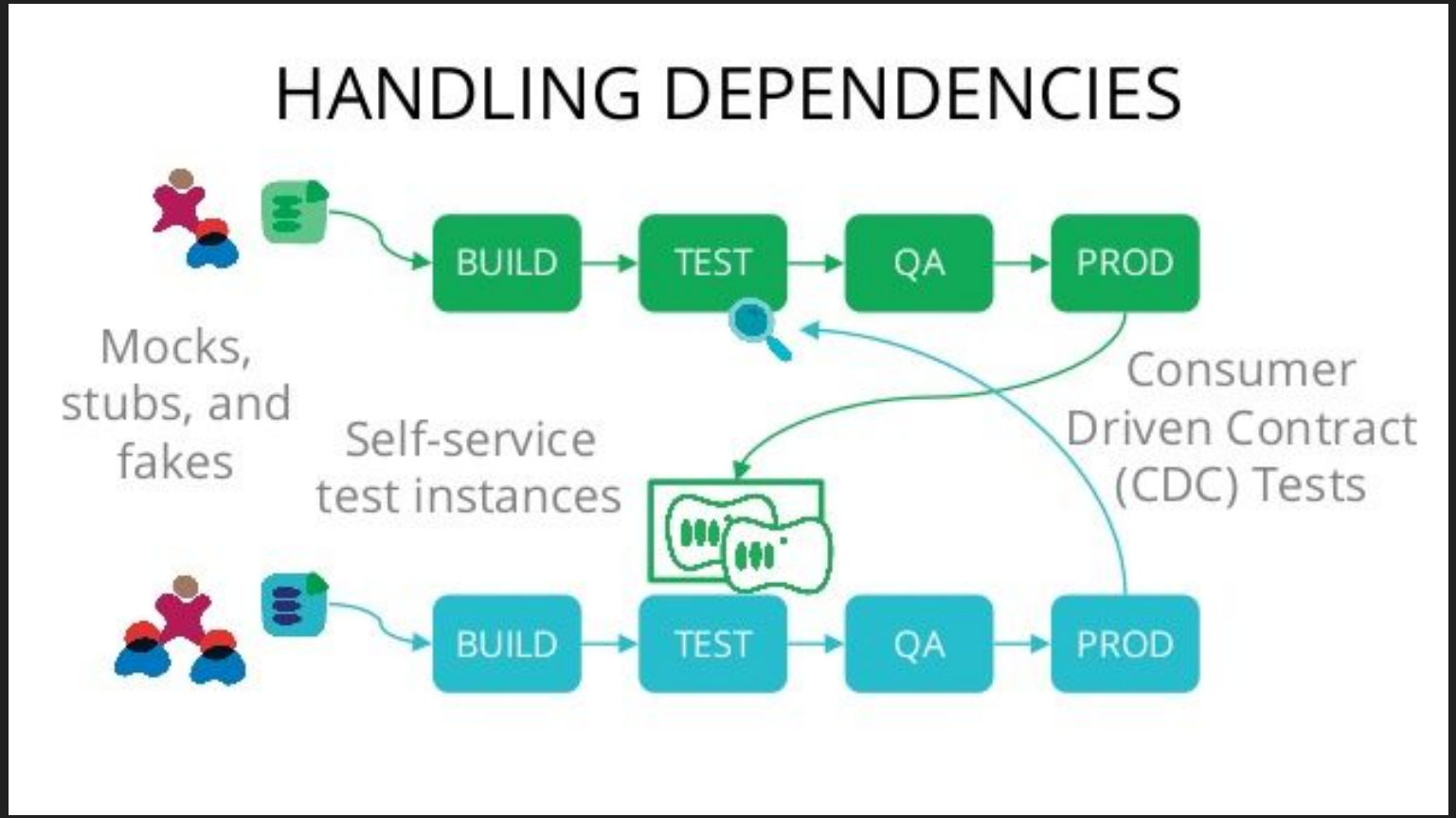
IaC Design Patterns



IaC Design Patterns



IaC Design Patterns



Questions?