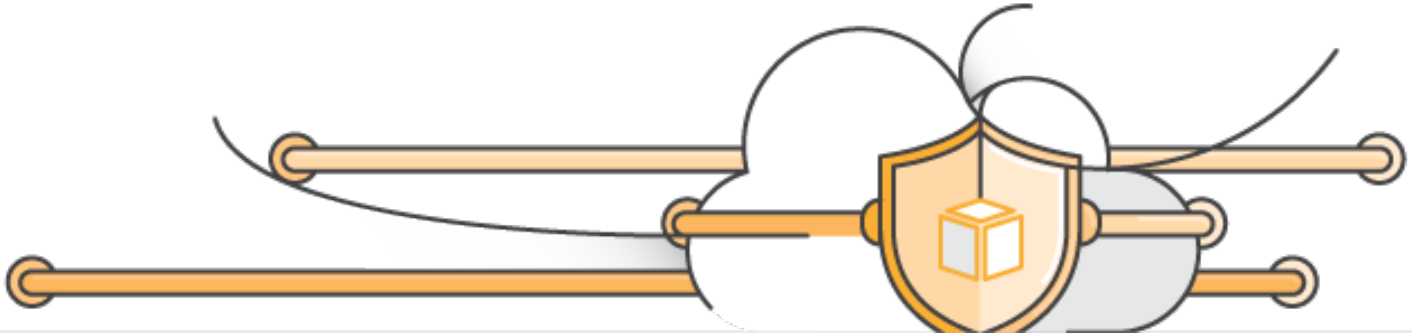


Securing Systems at Cloud Scale

Ian Massingham, Technical Evangelist

@IanMmmm

amazon web services

DevSecOps

Integrating Security with DevOps

What is DevOps?

Cultural Philosophy  Practices  Tools

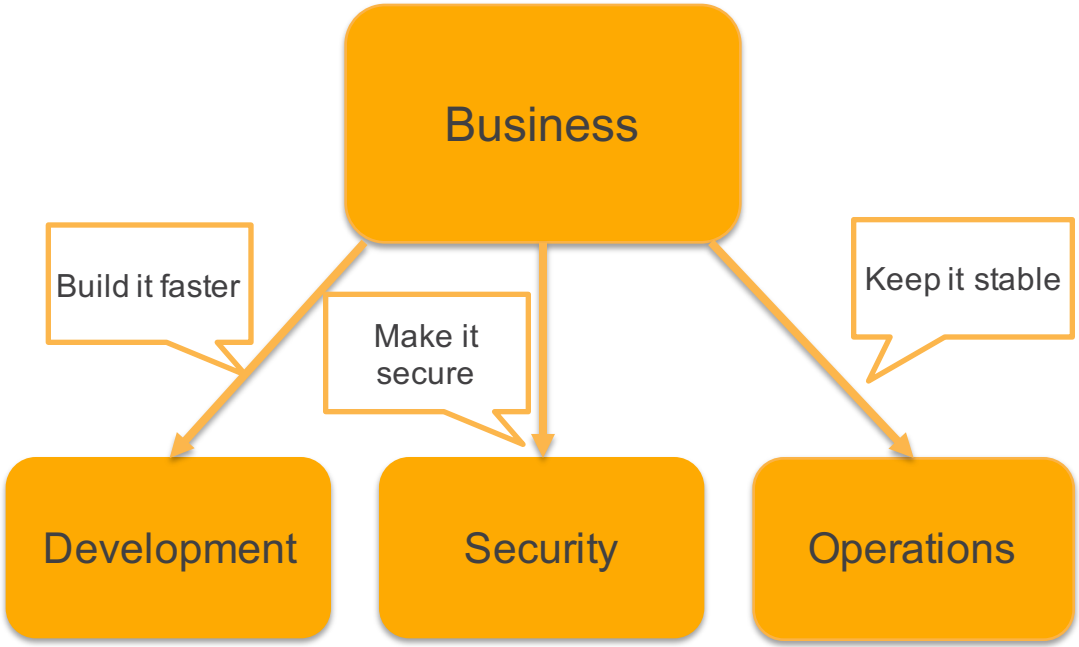
What is DevOps

- Breakdown the barriers
- Work as one team end to end
- Support business and IT agility
- Collaboration & Communication
- Treat Infrastructure as code
- Automate everything
- Test, measure & monitor everything

Culture

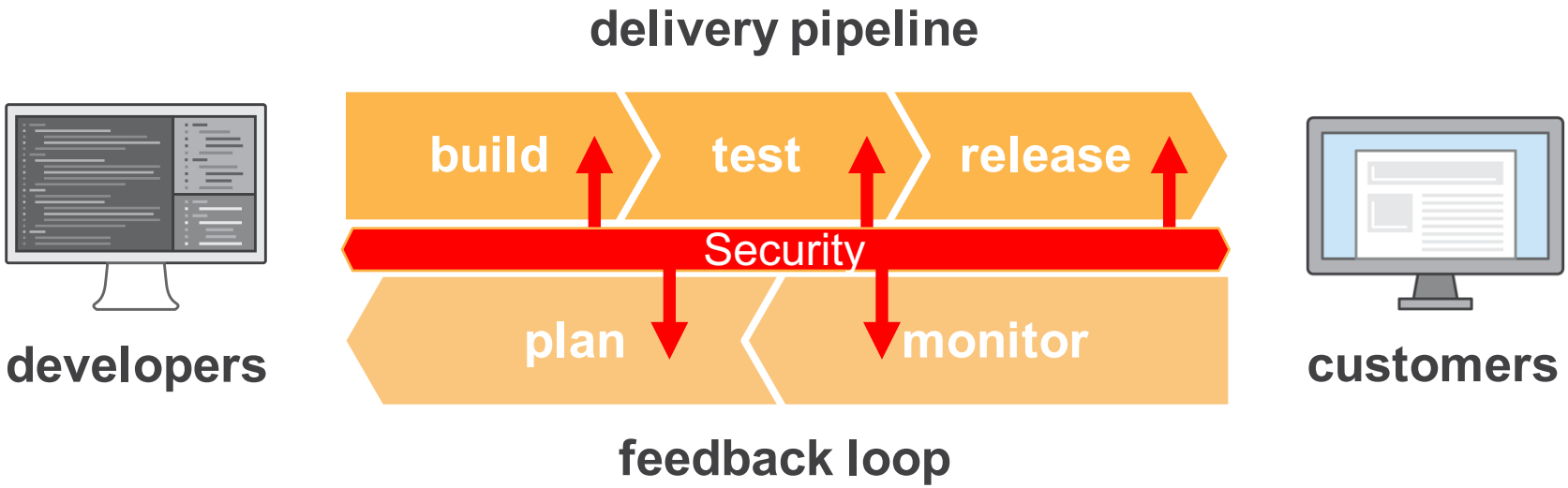
Technology

Competing Forces



What is DevSecOps

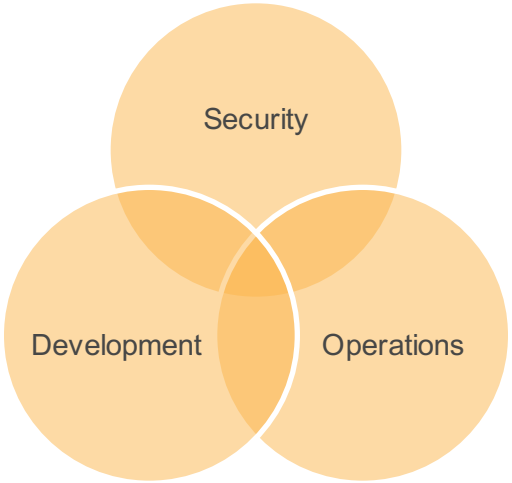
Software development lifecycle



DevOps = Efficiencies that speed up this lifecycle
DevSecOps = Validate building blocks without slowing lifecycle

Who is DevSecOps

- DevSecOps is
  - Team/Community, not a person
  - Automated and autonomous security
  - **Security at scale**



- DevSecOps role
  - **Not** there to audit code
  - Implement the control segments to validate and audit code and artifacts as part of the CI/CD process

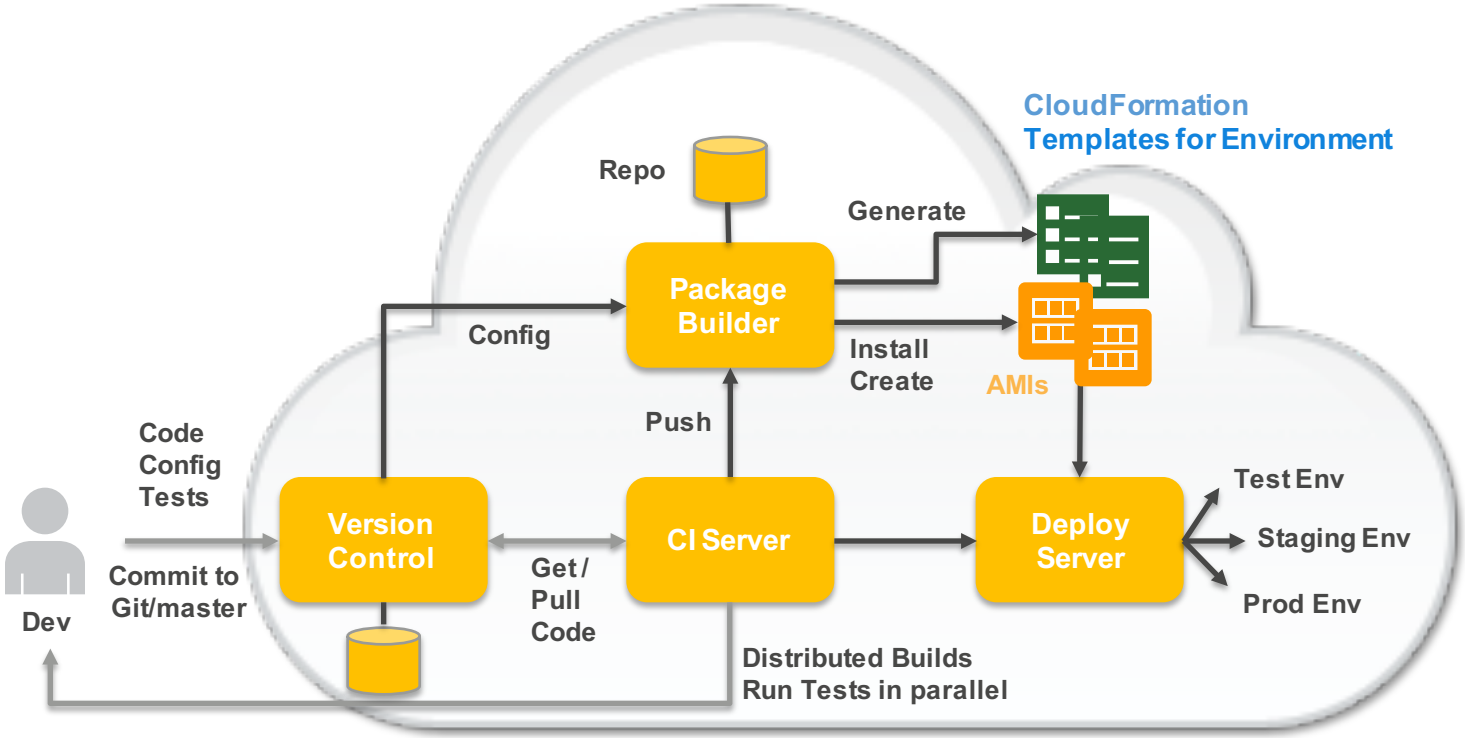
Continuous Integration

Techniques and tools to implement the **continuous process of applying quality control**; in general, small pieces of effort, applied frequently, to improve the quality of software, and to reduce the time taken to deliver it.

Continuous Delivery

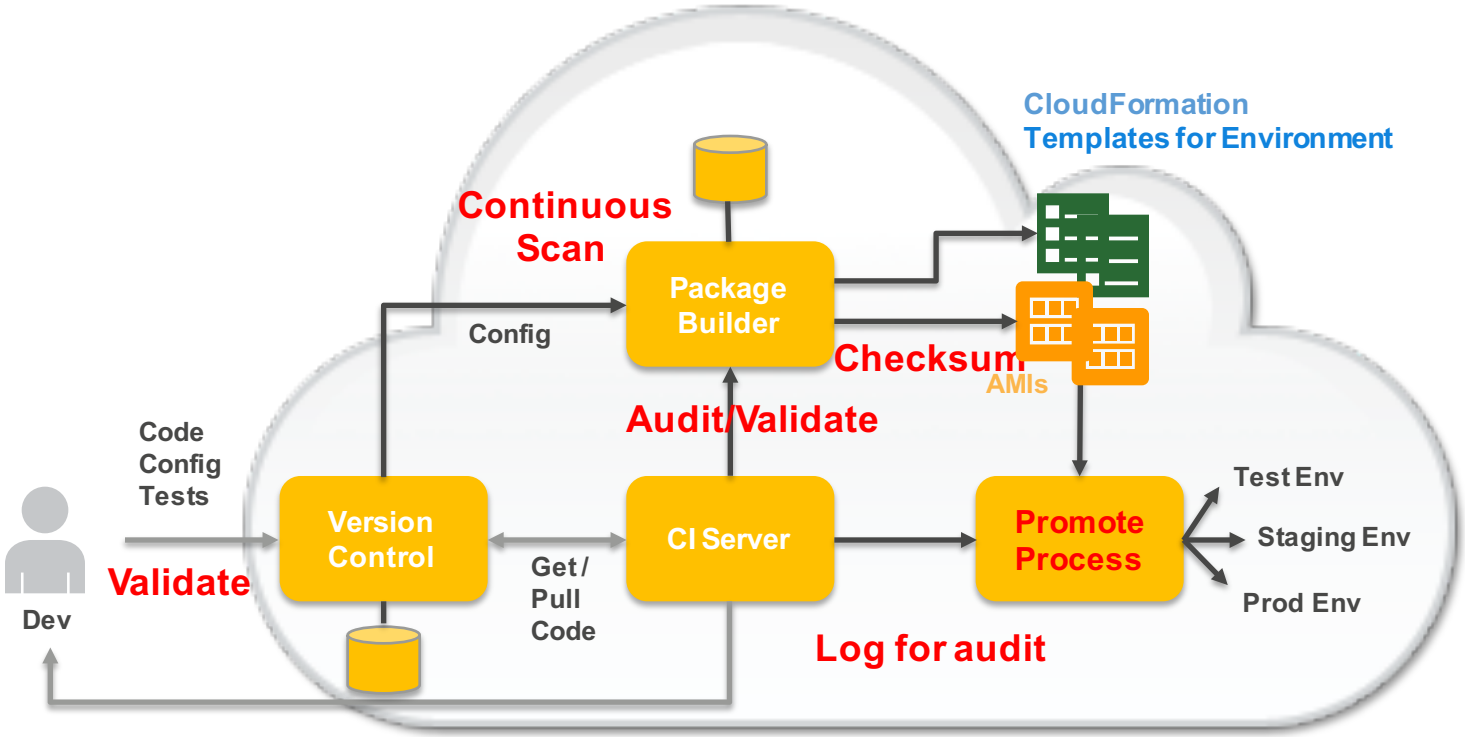
Techniques and tools to **improve the process of software delivery**, resulting in the ability to rapidly, reliably, and repeatedly push out enhancements and bug fixes to customers at low risk and with minimal manual overhead.

CI/CD for DevOps



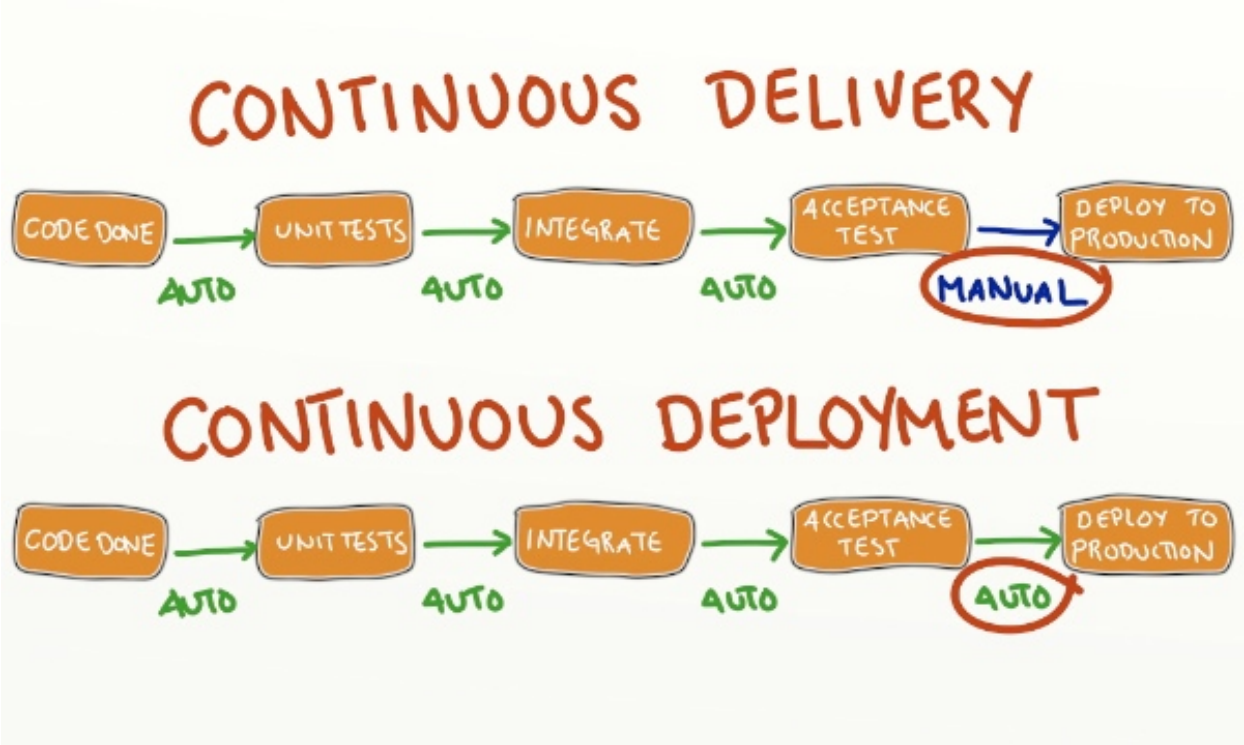
Send Build Report to Dev
Stop everything if build failed

CI/CD for DevSecOps



Send Build Report to Security
Stop everything if audit/validation failed

Promotion Process in Continuous Deployment



What Does DevSecOps CI/CD Give Us?

- Confidence that our code is validated against corporate security policies.
- Avoid infrastructure/application failure in a later deployment due to different security configuration
- Match DevOps pace of innovation
- Audit and alert
- Security at scale!

Governance First

Infrastructure is code

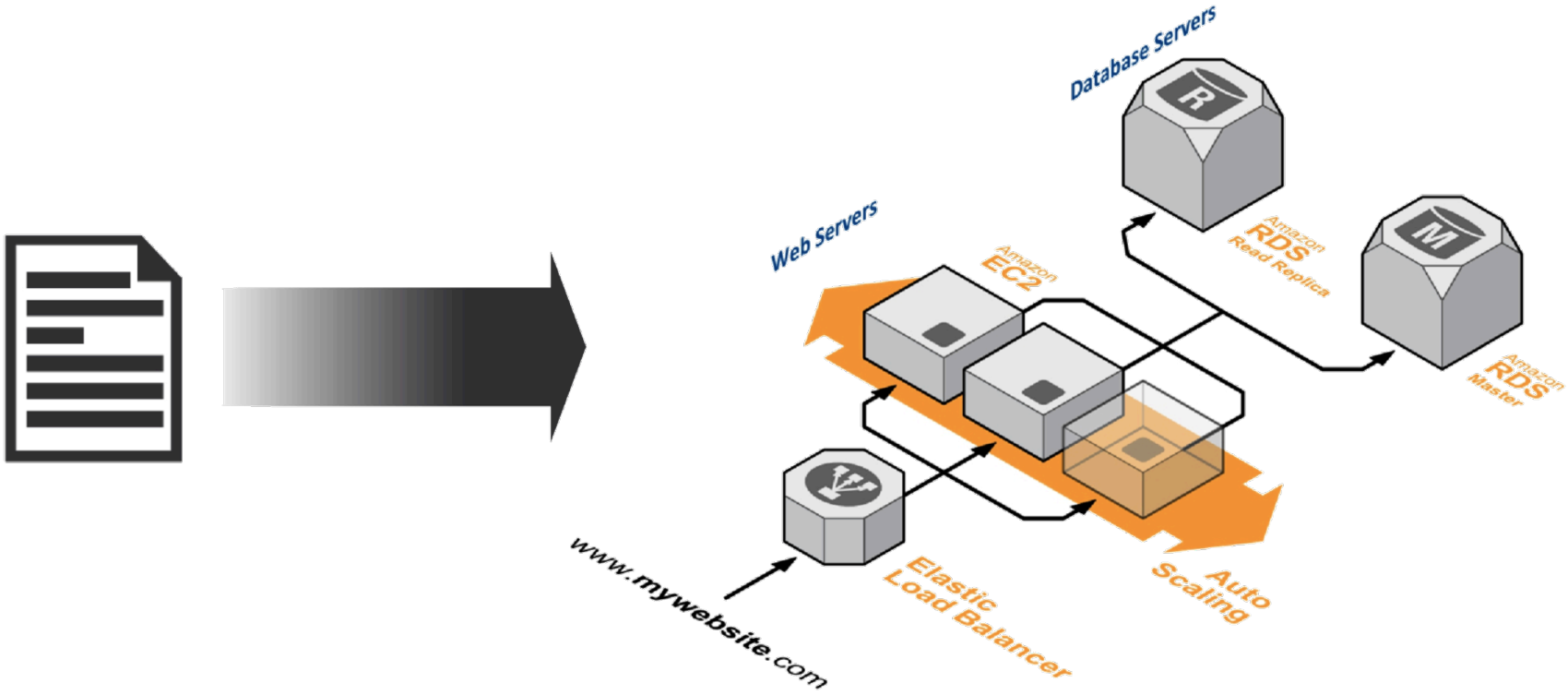
AWS CloudFormation Primer

Allows you to define a “template” which is composed of different “resources” and then provision that template into repeatable, live, “stacks”.

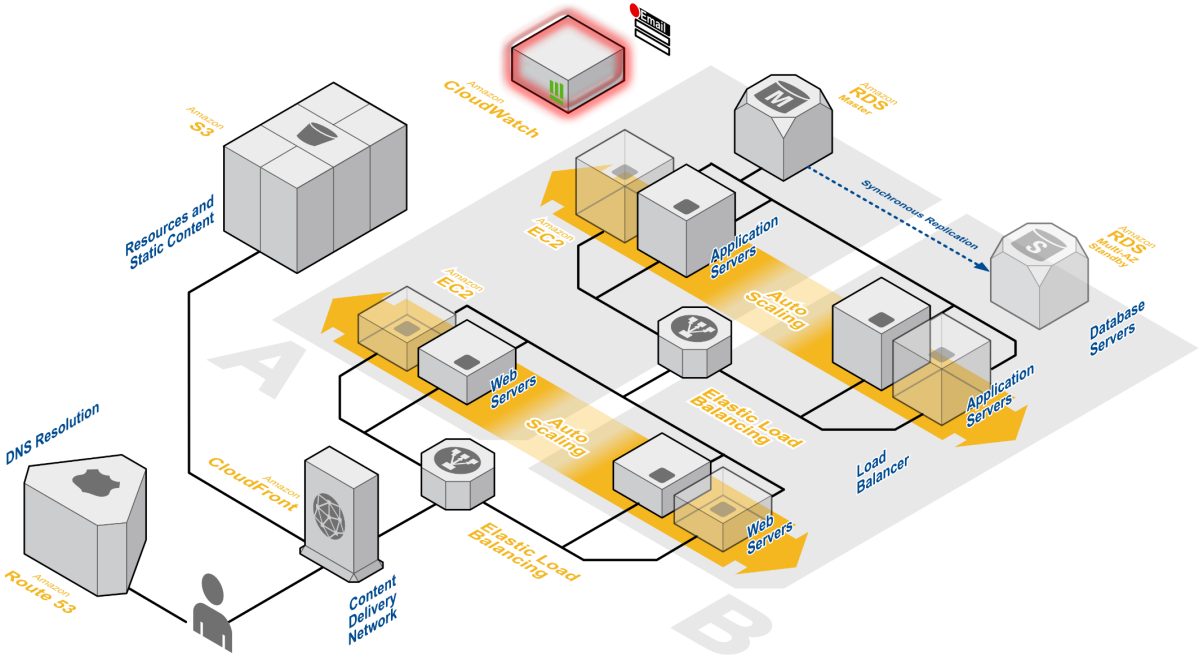
CloudFormation (CFn) providing a single service interface and abstracts the underlying API semantics and implementation mechanics.

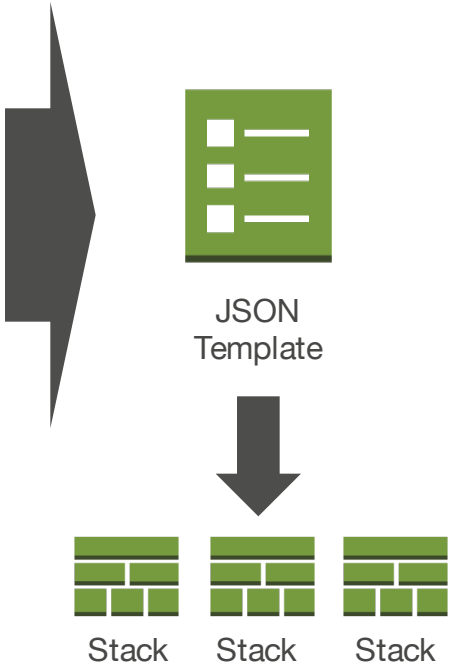
CFn templates can hook into external configuration management frameworks such as Jenkins/Chef/Puppet/etc.

AWS CloudFormation Primer



AWS CloudFormation Stacks





CFn for Security Practitioners

"Infrastructure **is** code"

- We already know how to manage "code" lifecycle
- Let CFn perform state changes and govern who calls CFn

Split ownership templates

- Merge code from Sec, Dev and Ops
- Baseline and build/state management

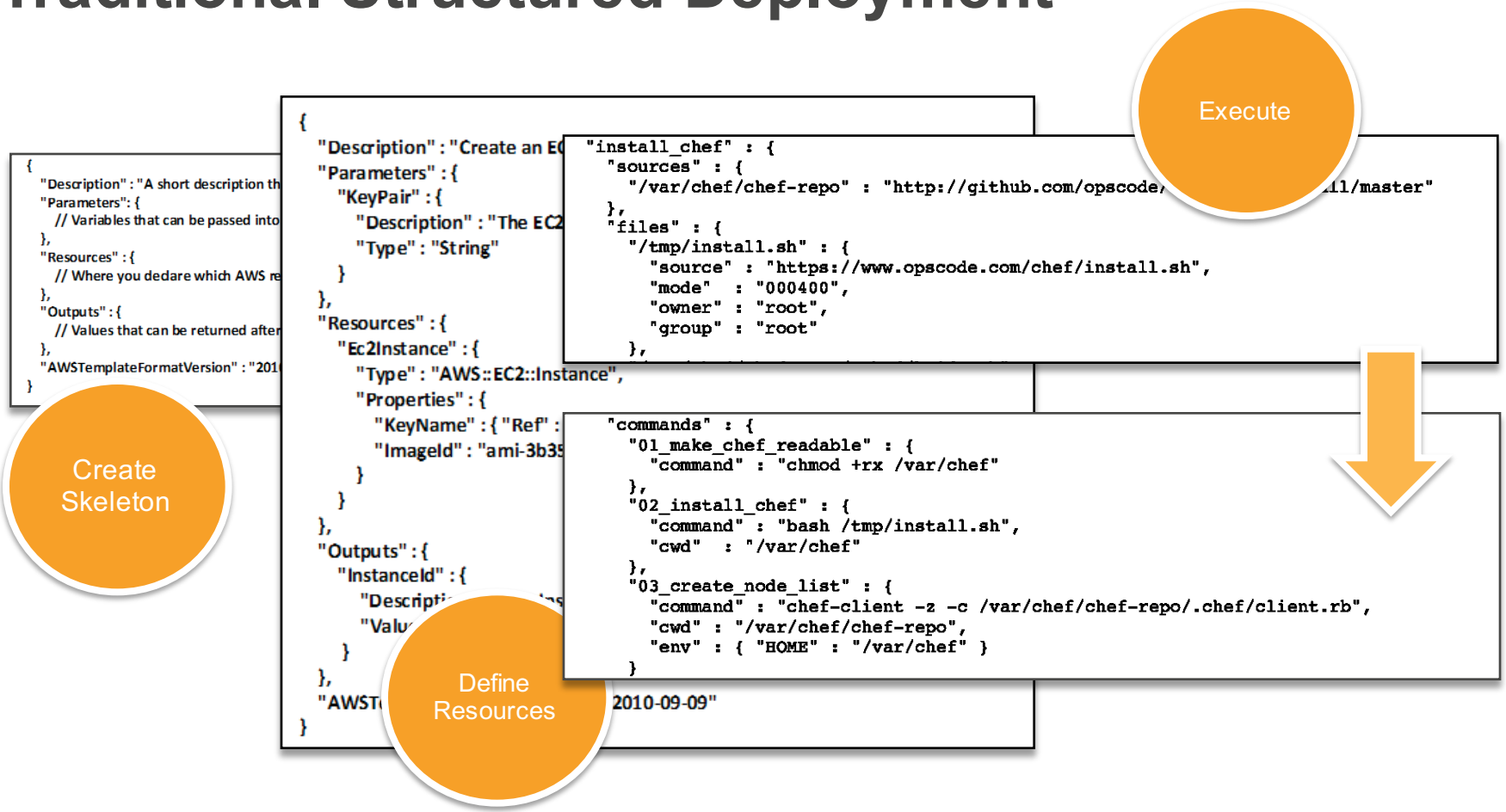
Provides inventory and documentation

- Resources are the same, just scaled

Integrate with Service Catalog

- Provides UI
- Use constraints (example: tags)

Traditional Structured Deployment



Split Ownership Configurations

Who knows your solution best?

- Dev, Infra, Sec…?
- Delegate ownership

Use Yaml and split file into chunks or functions

- Separate file sources with access control – Use IAM/VPC-E/etc.
- Push files -> **Validate** -> Merge files -> **Validate** -> Deploy -> **Validate**

Jenkins for deployment

- Promotion flows
  - Move from manual to Automation based on validation quality
- Excellent for merging jobs of split configurations

Merging

From single file or multiple files

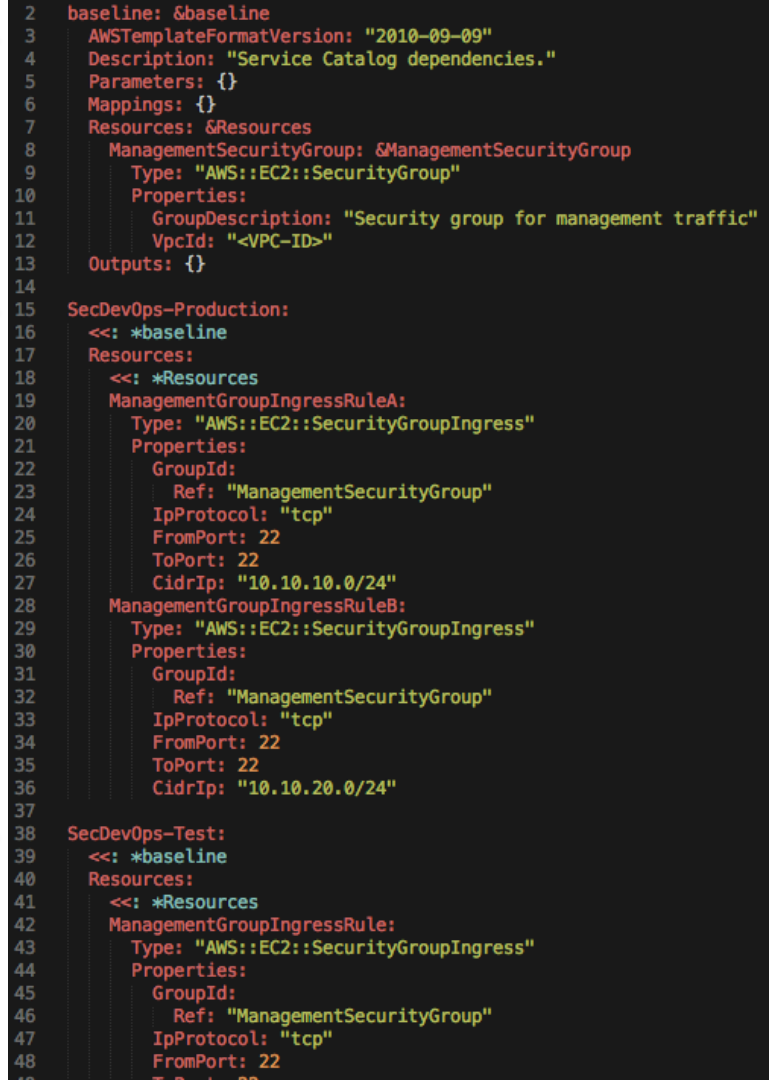
- Maintain access control using policies
- Use different source stores if needed

Based on function/state

Reusable patterns

Maintain order, especially of validation

- Security validation last to execute
- Security should always win



```
baseline: &baseline
  AWSTemplateFormatVersion: "2010-09-09"
  Description: "Service Catalog dependencies."
  Parameters: {}
  Mappings: {}
  Resources: &Resources
    ManagementSecurityGroup: &ManagementSecurityGroup
      Type: "AWS::EC2::SecurityGroup"
      Properties:
        GroupDescription: "Security group for management traffic"
        VpcId: "<VPC-ID>"
  Outputs: {}

SecDevOps-Production:
  <<: *baseline
  Resources:
    <<: *Resources
    ManagementGroupIngressRuleA:
      Type: "AWS::EC2::SecurityGroupIngress"
      Properties:
        GroupId:
          Ref: "ManagementSecurityGroup"
        IpProtocol: "tcp"
        FromPort: 22
        ToPort: 22
        CidrIp: "10.10.10.0/24"
    ManagementGroupIngressRuleB:
      Type: "AWS::EC2::SecurityGroupIngress"
      Properties:
        GroupId:
          Ref: "ManagementSecurityGroup"
        IpProtocol: "tcp"
        FromPort: 22
        ToPort: 22
        CidrIp: "10.10.20.0/24"

SecDevOps-Test:
  <<: *baseline
  Resources:
    <<: *Resources
    ManagementGroupIngressRule:
      Type: "AWS::EC2::SecurityGroupIngress"
      Properties:
        GroupId:
          Ref: "ManagementSecurityGroup"
        IpProtocol: "tcp"
        FromPort: 22
```

Validation

Keep track of what section you are validating

- Stage vs Prod
- Merged vs separated

Validate often and log/alert

- Validate part and end result
- Run-time validation

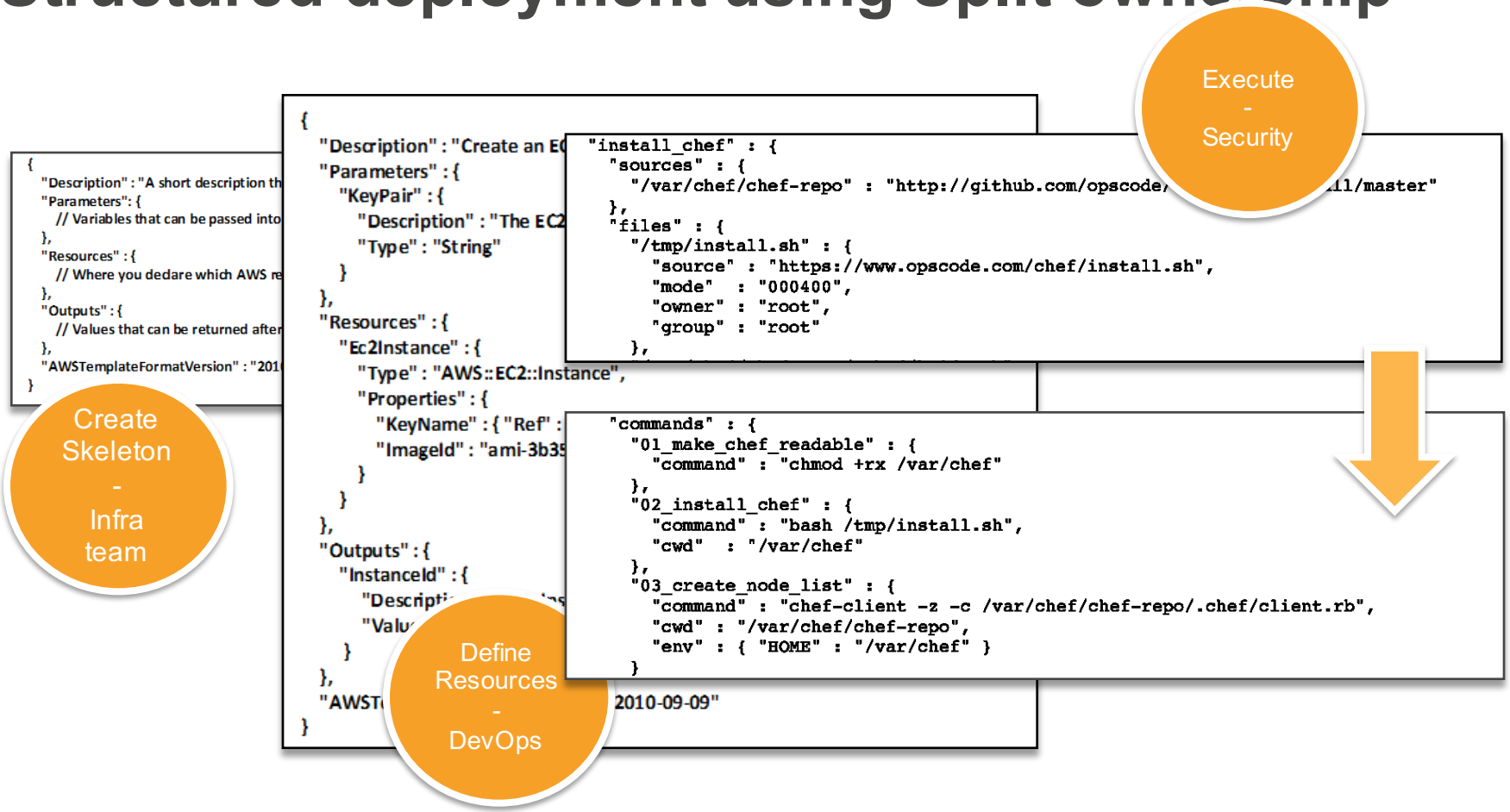
Use external agents

- AWS Simple Work Flow
- AWS Lambda
- Etc.

```
f = File.open("template.json", "rb")
g = f.read
g.gsub(",",",",\n").each_line {|line|
    if (line.downcase.include? "cidrip") && !line.include?("0.0.0.0/0")
        cfnCorpCIDRArray.push line[/(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|
    elsif (line.downcase.include? "cidrip") && line.include?("0.0.0.0/0")
        cfnCIDRArray.push line[/(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-
        p errorCIDRSource
        fail = true
    elsif (line.downcase.include?("fromport") && line.include?("0"))
        cfnPortArray.push line[/(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-
        p errorPortSource
        fail = true
    end
    }

# Check Corp CIDR blocks
strFail = ""
cfnCorpCIDRArray.each {|y|
    found = false
    corpCIDRArray.each {|x|
        cidr = NetAddr::CIDR.create(x)
        if ((cidr.contains?(y)) || (y == x))
            found = true
        end
        }
    if !found
        p errorCorpCIDRSource + " (#{y})"
        fail = true
    end
}
```

Structured deployment using Split ownership



Applied Governance

Building processes to segment and encapsulate application packages

Applied Governance deconstructed

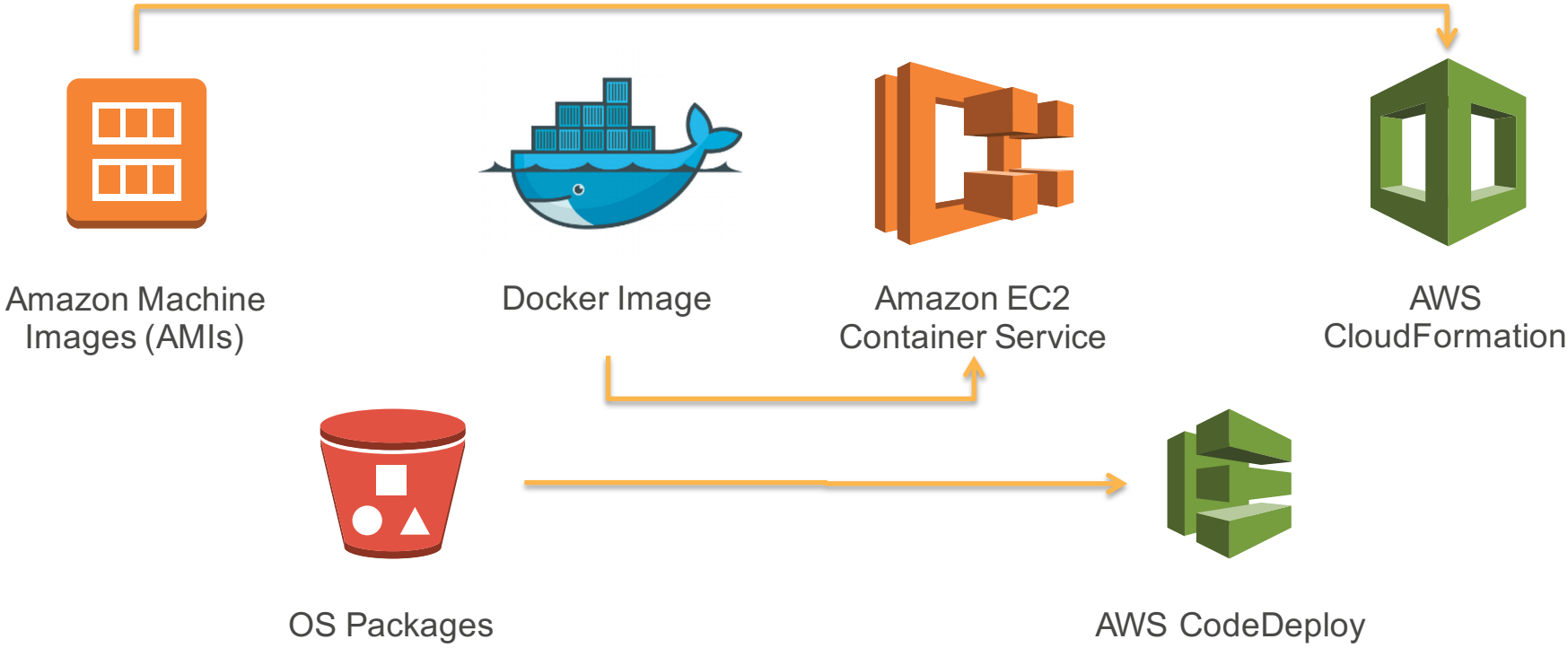
Deployment mechanisms for software artifacts

Configuration building blocks

Managing segmentation and security dependencies

…and a real scenario to wrap this all together.

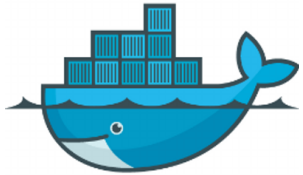
Deployment Mechanisms for Software Artifacts



Deployment Mechanisms for Software Artifacts

Software Artifacts

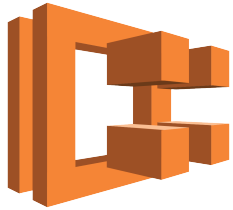
Amazon Machine Images (AMIs)



Docker Images

OS Packages

Deployment Services

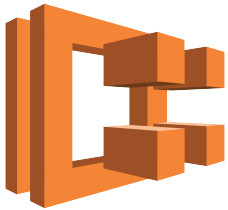


Amazon EC2 Container Service

AWS CloudFormation

AWS CodeDeploy

Configuration building blocks



```
"name": "wordpress",
"links": [
  "mysql"
],
"image": "wordpress",
"essential": true,
"portMappings": [
  {
    "containerPort": 80,
    "hostPort": 80
```

```
version: 0.0
os: operating-system-
files:
  source-destination-
permissions:
  permissions-specific
hooks:
  deployment-lifecycle
```

CloudFormation Template

Task Definition

Application Specification File (AppSpec file)

…and more.

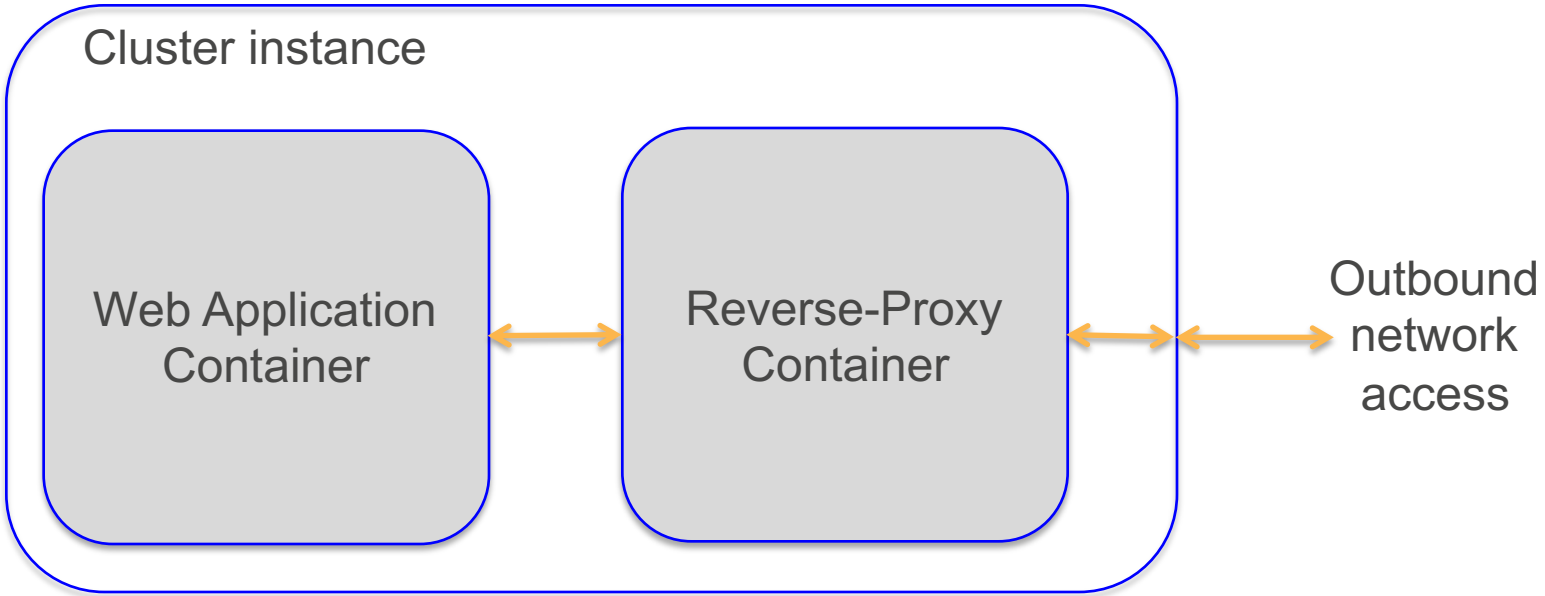
Managing segmentation and security dependencies

Why use these configuration building blocks to enforce security?

- Remove accidental conflicts
- Make security processes continuous and automatic
- Encapsulate software artifacts and implement controls one level up
- Control changes to these mechanisms via IAM

Example - Amazon EC2 Container Service

Rate limiting proxy



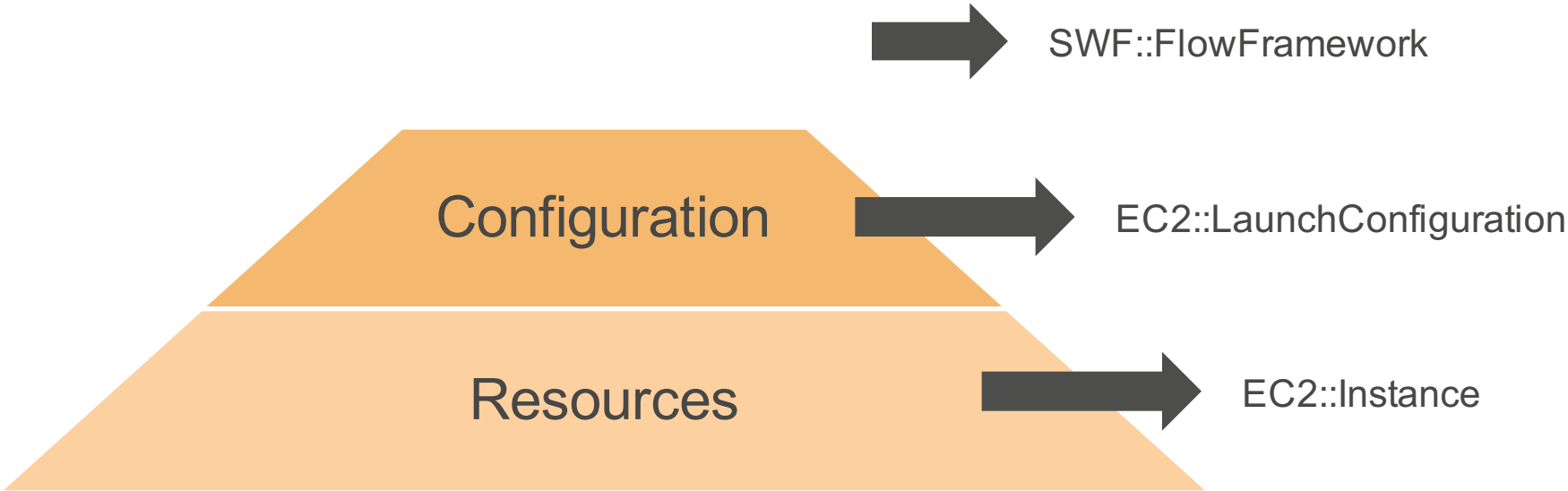
Scaling Trust Decisions

Autonomous security

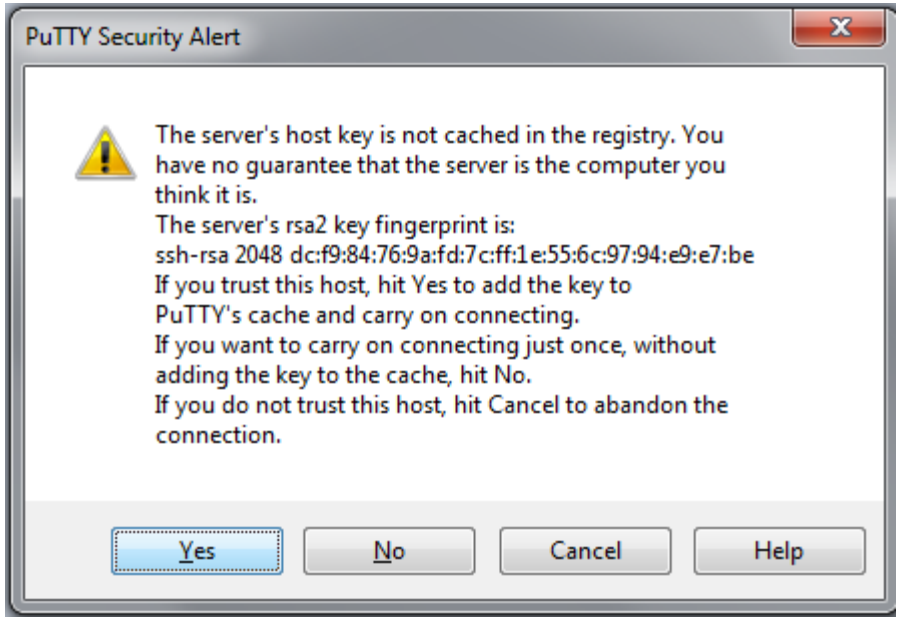
Does this flow feel familiar

1. Deploy Instance (*Chef got it so no worries…*)
2. Follow checklist #1 for patches (*Oops…New patches arrived…time to update Chef*)
3. Follow latest security list (Sec team don't have access to Chef yet…)
4. Put in production
5. SecOps called…they have a new list (*oh dear…*)
6. Pull from production
7. Rinse and repeat

Design time vs run time decisions



Why do we need these trust decisions?



Thought experiment: what is the root of trust for your Amazon EC2 instances?

How can we translate that into our W's?

- **Who** [is that instance]?
- **What** [is it's configuration]?
- **Where** [is the instance provisioned]?
- **When** [was it launched]?
- **Why** [is this instance alive]?

Scaling trust decisions - Allows

Allows Split Ownership

Allows intelligent decisions

- Inspect – Decide – Apply – Validate – Action - Log

Forced

- Comply or be destroyed

Security inventory for audits at scale

- Gather information from multiple sources
  - Host based
  - EC2 service
  - Created artifacts

Unique per resource action and information

- Keys/Certificates

Scaling trust decisions - Do

Use combination of design time and run time decisions

Tie to authentication on-demand for secure access of automated processes

Combine with Continuous scanning

- Code artifacts
- Resources (AWS Config)

Don’t mistake run time decisions with post deployment

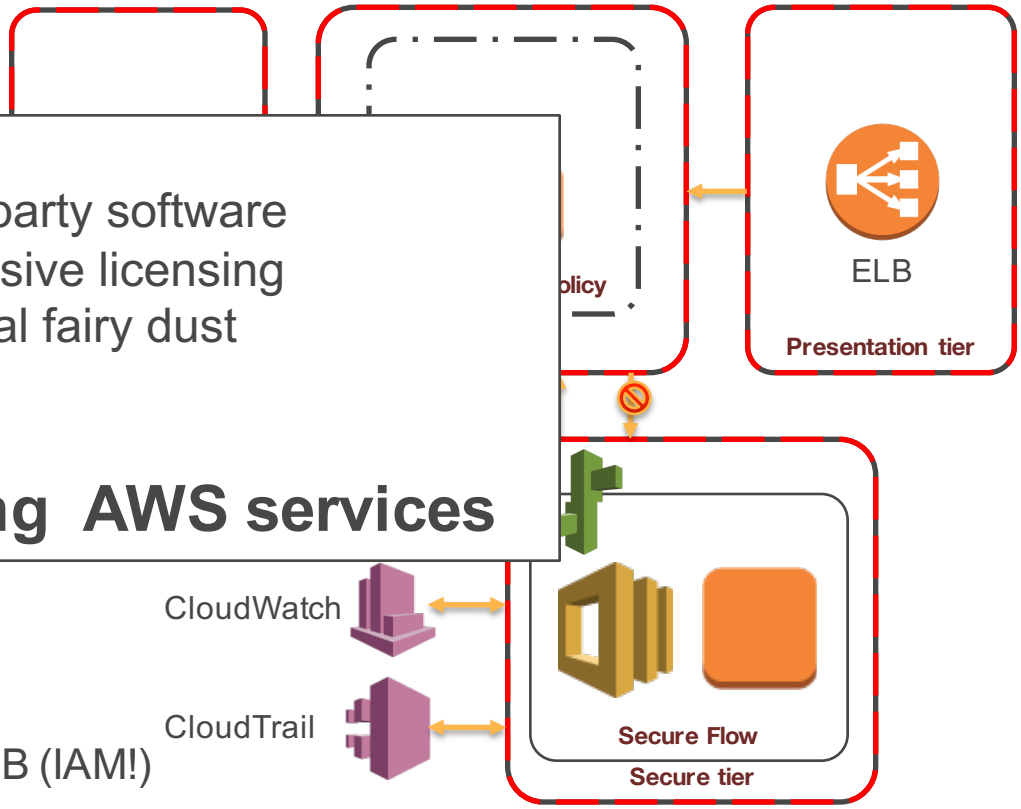
- Run-time happen at creation but before being made available

Push pattern – web server certificate

1. AutoScaling policy triggers
   - Hook Pending:Wait
2. Flow
   1. Create certificate *U
   2. Start web service
      - Certificate in me
   3. Delete certificate fr
   4. Install AWS CloudW
   5. Create SSH_RSA k
   6. Replace SSH_RSA
3. Validation
   1. Check CW Logs API
   2. Run WGET
4. Logging
   1. Log information/state in DynamoDB (IAM!)
5. Continue/terminate instance/hook
   1. Instance InService/terminated

No need for:
- Third party software
- Expensive licensing
- Magical fairy dust

All using AWS services



Instances have no knowledge on process or about the secure flow

Run-time security flow – best practices

Avoid pull patterns

- Requires an available source/path from target system
- Own/compromise the target and you own the access process

Push ensures only outgoing traffic

- Instances have no path back to source
- Instances have no knowledge of the process/flow

Ephemeral/on-demand scripting on target

- Store in memory
- Create scripts on target on-demand at execution time
- Temporary credentials for first run

Keep source store on-instance (worker node) or S3 using dedicated VPC with VPC-Endpoint for S3 policy

Tooling - Triggering

AutoScaling Group

- Use Lifecycle hooks
- Automatic repair of non approved instances
- SNS for integration with Workers/Lambda
- Use least privileges on AutoScaling Groupd instances
  - Use IAM Instance Roles
- Unique credentials per instance

Tooling - Worker

Simple WorkFlow

- Flow engine for Ruby/Java
- Allow synchronous or asynchronous flows
- Deciders/Workers
  - EC2 or Lambda
- Can be integrated into Elastic Beanstalk
- Use internal or external source for validation tasks/tests
  - Checksum files before use
  - Use Lambda/Elastic Beanstalk workers

Real-life trust applications

- Agent deployment/validation
- Integration with Configuration Management (Chef/Puppet)
- Unique per-instance configuration
- Cross-instance network authentication
- Joining an Active Directory

Ask yourself what makes this instance trustworthy for any of the above and how can I scale that trust?

Thank You

Please let us have your feedback on this event!

Don’t forget to leave your badge and collect your $50 AWS Credit Token