Electric Cloud

Process-as-Code: Real-World Examples that Scale

Marco Morales

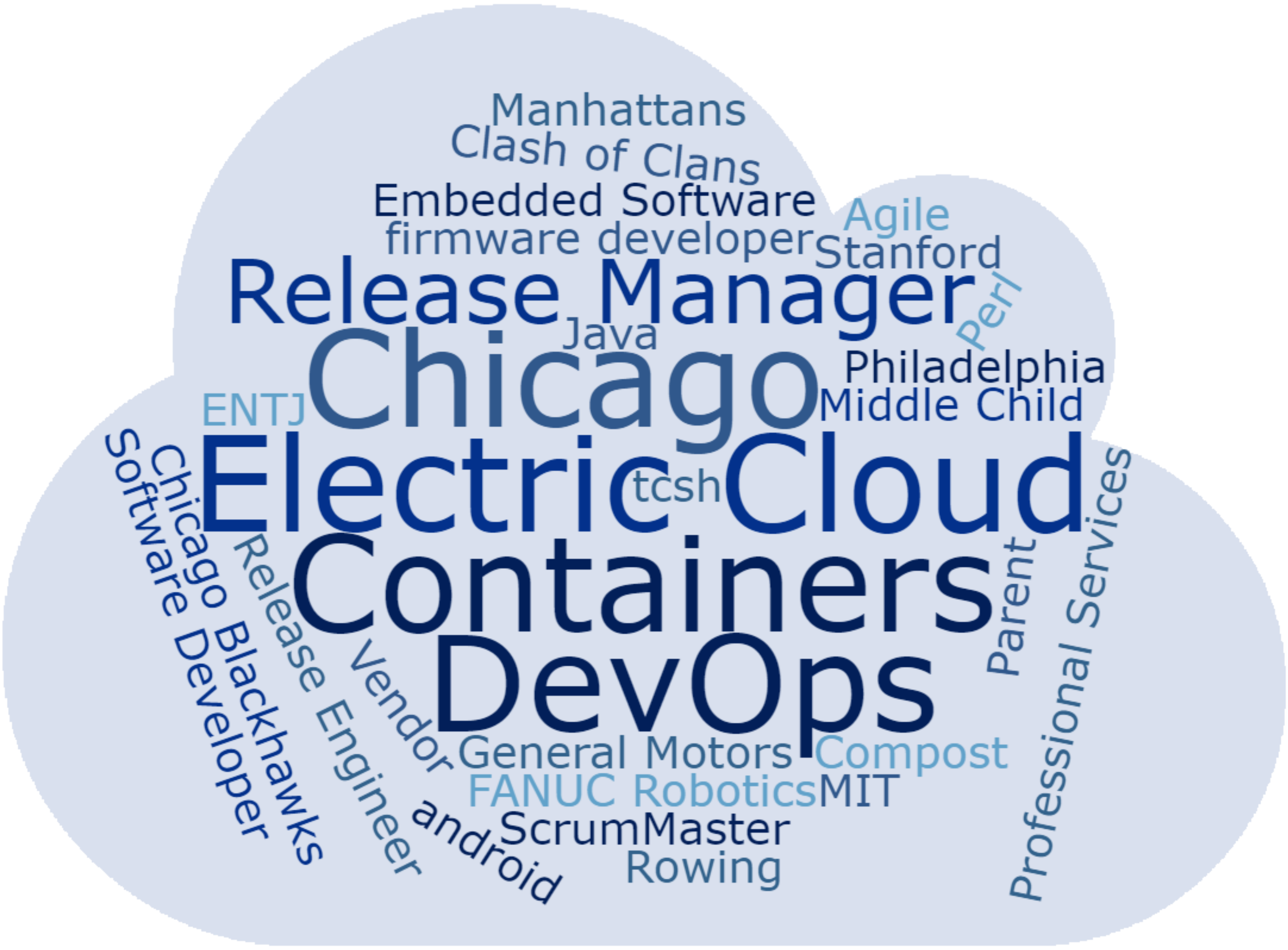
DOES 2017

About Me…Marco Morales

Long-time Developer / build-release Engineer & Manger/ Agile / DevOps participant

Experienced in DevOps tools, processes, behaviors, eating habits, etc.

mrmarcoamorales @ Twitter | LinkedIn

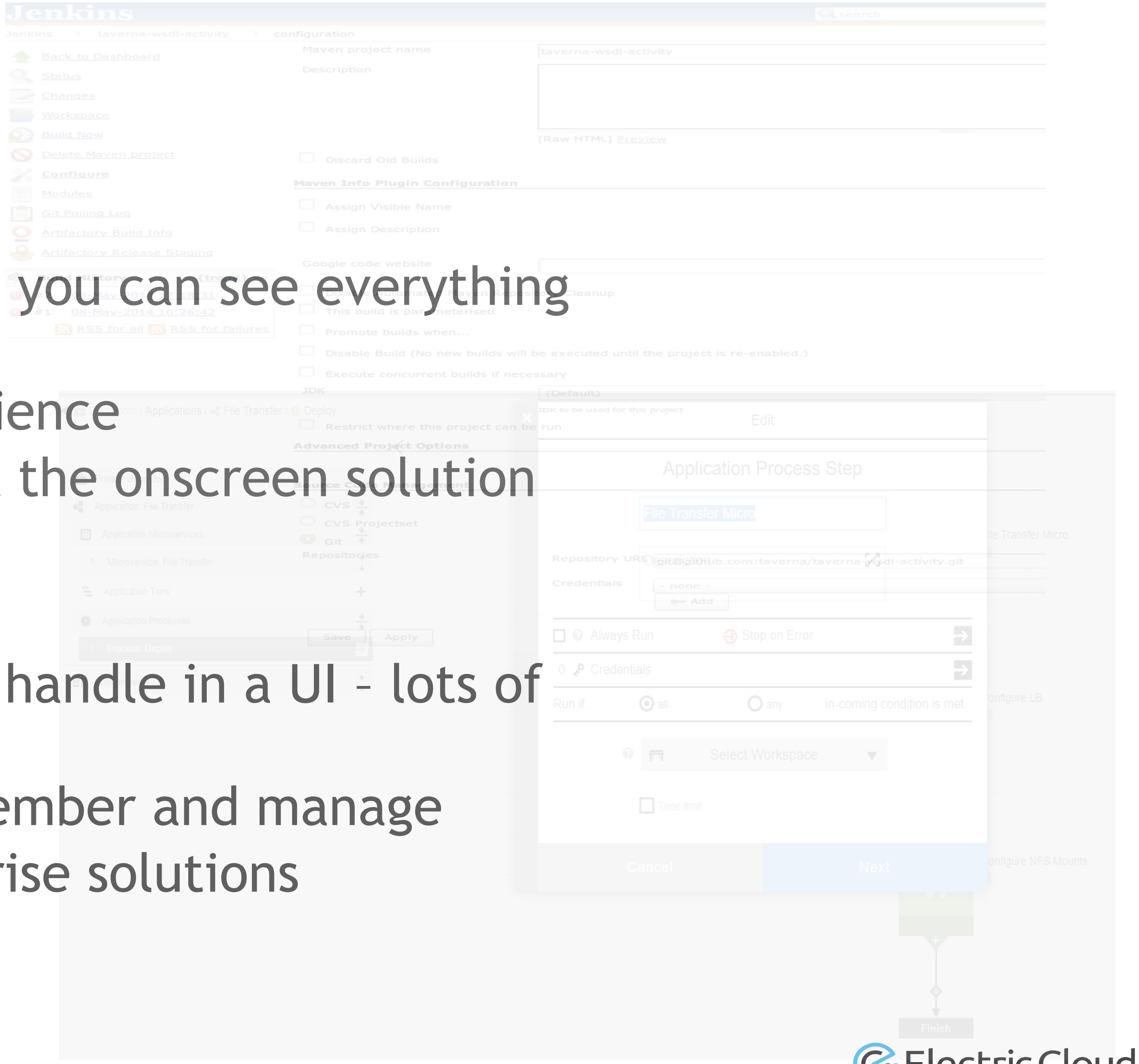


Problem Description

As a user, I wish to define my DevOps **processes** quickly and efficiently to handle all my use cases and working with other members of my team

Difficult to address these simultaneously

Why is it difficult?



The User Interface

- Well-suited for problem where you can see everything on one screen
- Click-through and fill-in experience
- Help & guidance to implement the onscreen solution

However –

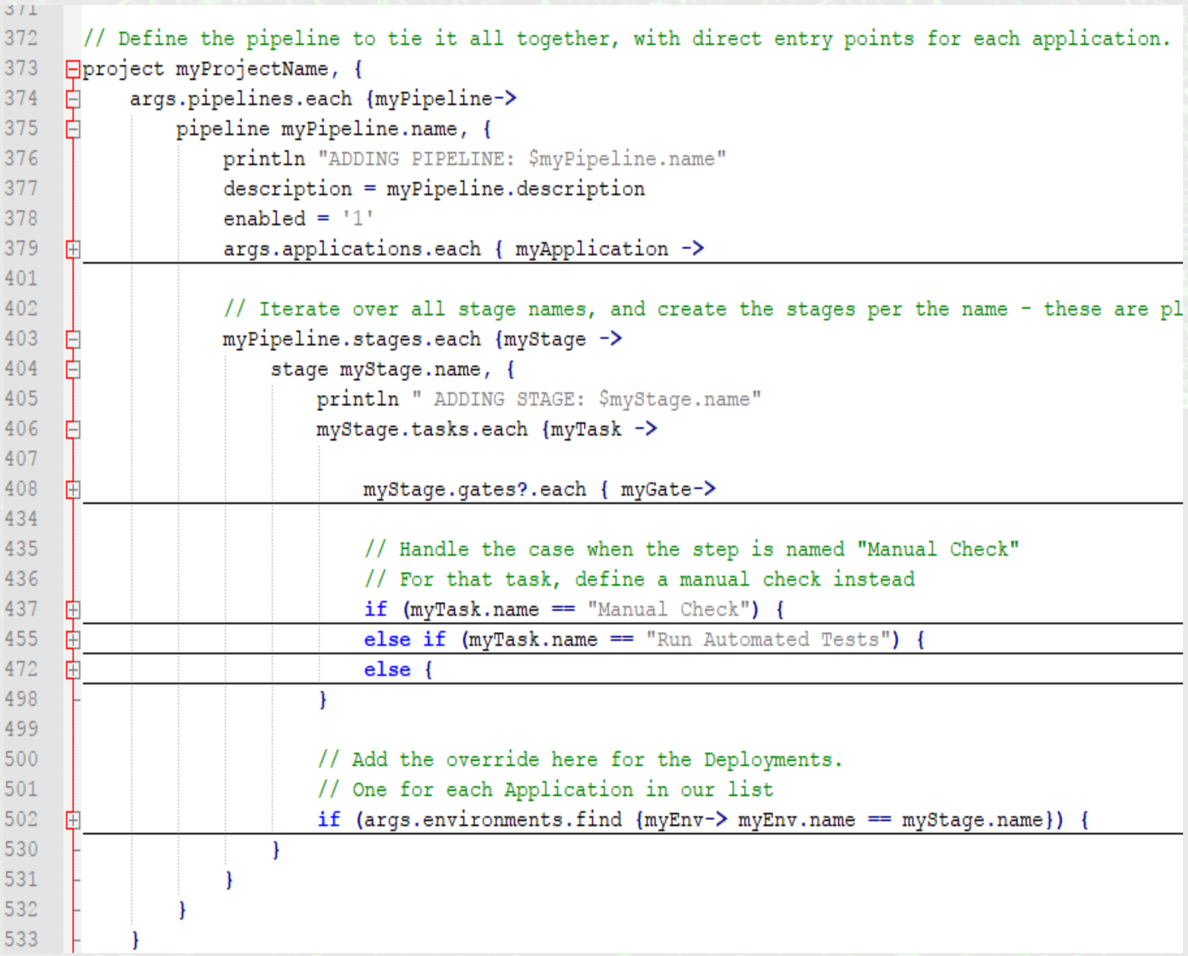
- Large numbers are difficult to handle in a UI – lots of clicks, cut-paste-and-modify
- Variations are difficult to remember and manage
- Difficult to co-develop enterprise solutions

Process as Code as a Solution

In DevOps, Process-as-Code (PaC) really means treating your automation processes as a software development project

- Your code is a product and protected asset
- Your team follow software development disciplines and techniques
- Easier to achieve desired behaviors, such as sharing ideas and increased collaboration

Your processes are *designed* to better meet the needs of your organization



```
// Define the pipeline to tie it all together, with direct entry points for each application.
project myProjectName, {
    args.pipelines.each {myPipeline->
        pipeline myPipeline.name, {
            println "ADDING PIPELINE: $myPipeline.name"
            description = myPipeline.description
            enabled = '1'
            args.applications.each { myApplication ->

            // Iterate over all stage names, and create the stages per the name - these are pl
            myPipeline.stages.each {myStage ->
                stage myStage.name, {
                    println " ADDING STAGE: $myStage.name"
                    myStage.tasks.each {myTask ->

                        myStage.gates?.each { myGate->

                        // Handle the case when the step is named "Manual Check"
                        // For that task, define a manual check instead
                        if (myTask.name == "Manual Check") {
                        else if (myTask.name == "Run Automated Tests") {
                        else {
                    }

                    // Add the override here for the Deployments.
                    // One for each Application in our list
                    if (args.environments.find {myEnv-> myEnv.name == myStage.name}) {
                }
            }
        }
    }
```

Real-World Examples

- Large US Bank – Hundreds of applications managed entirely from DSL in version control - <u>Everything</u>
- Large Brokerage Firm – All applications onboarded from DSL templates, no custom code
- US-based retailer – Dozens of applications, hundreds of environments, spreadsheet-driven
- US-HQ Online marketplace– 200 microservices purely from DSL. Data-driven example cited here

Assumptions and Prerequisites

You are working with a system that supports Process-as-Code concepts

- This usually means a **Domain Specific Language (DSL)** supporting your system
- Your DSL supports objects and attributes in your system
- You can work with a declarative model
- Import & Export

You are working on problems that require scale (servers, users, processes, microservices, etc.)

Best Practice: Use a Version Control System

If you work in an enterprise, chances are you have a code repository

Treat your code as a product development

- Deliverable to other people
- Include documentation (README.MD, instructions)
- Installation script

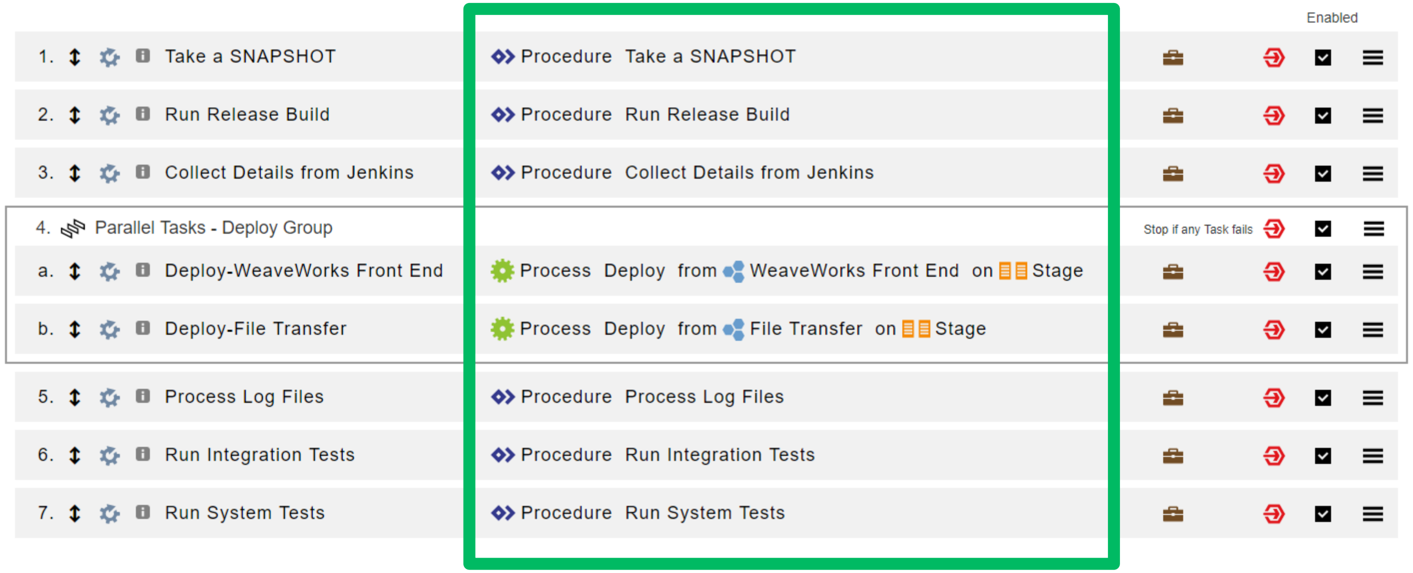
If you only treat the repository as your personal file storage, you are probably doing it wrong

Start with Stubs in the UI

- HUH? Counter-intuitive!
- Starting with the UI is a great way to get the initial **data and process models**
- You might even be able to mock-up your entire pipeline

Stubs are really helpful because when you run them, their status=SUCCESS!

```
echo “Hello world
from $[/myJob/name].
You supplied $[input
parameter]”
```



				Enabled
1.	Take a SNAPSHOT	Procedure Take a SNAPSHOT		☑
2.	Run Release Build	Procedure Run Release Build		☑
3.	Collect Details from Jenkins	Procedure Collect Details from Jenkins		☑
4.	Parallel Tasks - Deploy Group		Stop if any Task fails	☑
a.	Deploy-WeaveWorks Front End	Process Deploy from WeaveWorks Front End on Stage		☑
b.	Deploy-File Transfer	Process Deploy from File Transfer on Stage		☑
5.	Process Log Files	Procedure Process Log Files		☑
6.	Run Integration Tests	Procedure Run Integration Tests		☑
7.	Run System Tests	Procedure Run System Tests		☑

These are all stubs

Take the stubs and export them

You get an initial representation of your models

- Observe structure
- Figure out where to use variable references – for loops
- Figure out where to optimize – remove nulls and NOPs

```
step 'echo file', {
  description = 'Placeholder for future
implementation'
  command = 'echo Hello world from $[/myJob/name].
You supplied $[input parameter]'
}
```

Anti-Pattern - exports are not PaC

Exports provide machine-generated verbose output

Common initial assumption: an export is Process-as-code

- **Not correct.**

Exports are a data-dump – definition and state

When processes are instantiated, you create State information

- Build numbers
- Artifact versions
- Stored name/value

PaC is about specifying the ***definition*** of your processes

Separate Infrastructure and State information

Setup initial conditions separately from your processes

Infrastructure and pre-conditions

- Resources – the endpoints you touch
- Integrations – credentials, tokens
- Artifacts – the versioned objects you are working with
- Initial build numbers

Solving these problems helps solve the “first time in” problem

Create Your Data Model and a Test Harness

- Your data model replaces your UI data-entry experience
- Your test harness verifies your data model quickly and easily
- Same spirit as Test-Driven-Development
- You should have test and production data sets

Solving these problems helps reduce “special sauce” (i.e. snowflakes)

```
args.applications.each { myApplication ->
    println "APPLICATION : $myApplication.name"
    myApplication.services.each { service ->
        println " SERVICE : $service.name"
        service.containers.each {container ->
            println "  CONTAINER : $container.name"
...
args.environments.each {myEnvironment ->
    println "ENVIRONMENT: $myEnvironment.name"
    myEnvironment.clusters.each { myCluster ->
        println " CLUSTER: $myCluster.name"
...
args.pipelines.each {myPipeline ->
    println "PIPELINE: $myPipeline.name"
    myPipeline.stages.each {myStage ->
        println " STAGE: $myStage.name"
        myStage.tasks.each {myTask ->
            println "  TASK: $myTask.name"
```

Iterate through small software changes

- Work and rework your models

- Small changes

- Test along the way, perform trial runs

- Commit early, Commit often

```
git add your-model.groovy
git commit –m “describe small change”
```

WASH

RINSE

TEST

REPEAT

Real-world example - hundreds of servers with 28 lines

Problem: Onboard hundreds of servers and as environments

Data source: a spreadsheet, converted to JSON

JSON File

- Great for arrays and lists of data
- Great for scaling to large numbers
- Test data used 5-10, real data used hundreds
- Hundreds of servers – 27 seconds

Snippet: Excerpt from a 28-line script

```
args.elements.each { element ->
    def elementName = element.Element
        resource elementName, {
            description = element.Description
            hostName = element.Hostname
            zoneName = 'default'
        }
    project args.projName, {
        environment elementName, {
            environmentEnabled = '1'
            projectName = args.projName
            environmentTier args.envTier, {
                resourceName = elementName
```

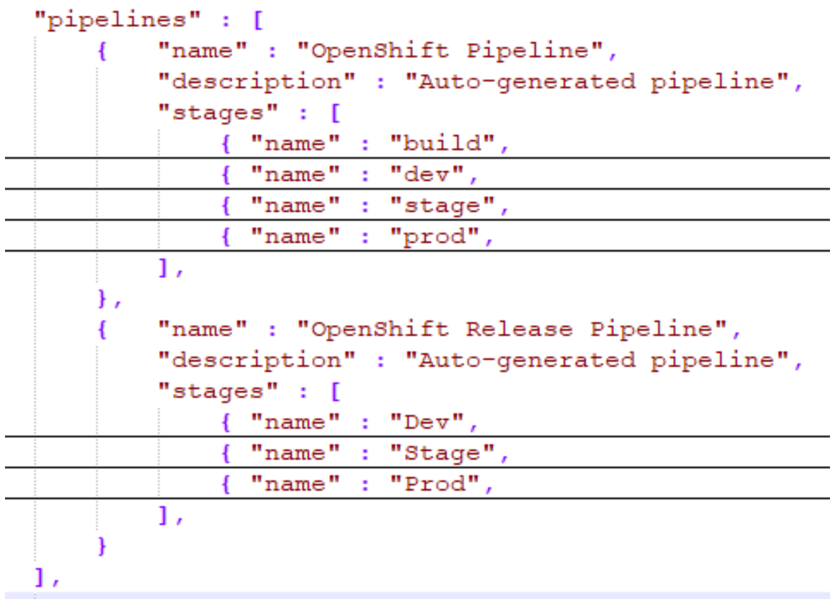
See also: https://github.com/electric-cloud/electricflow-examples/tree/master/CreateLotsOfResources

Separate top-level definitions from detailed definitions

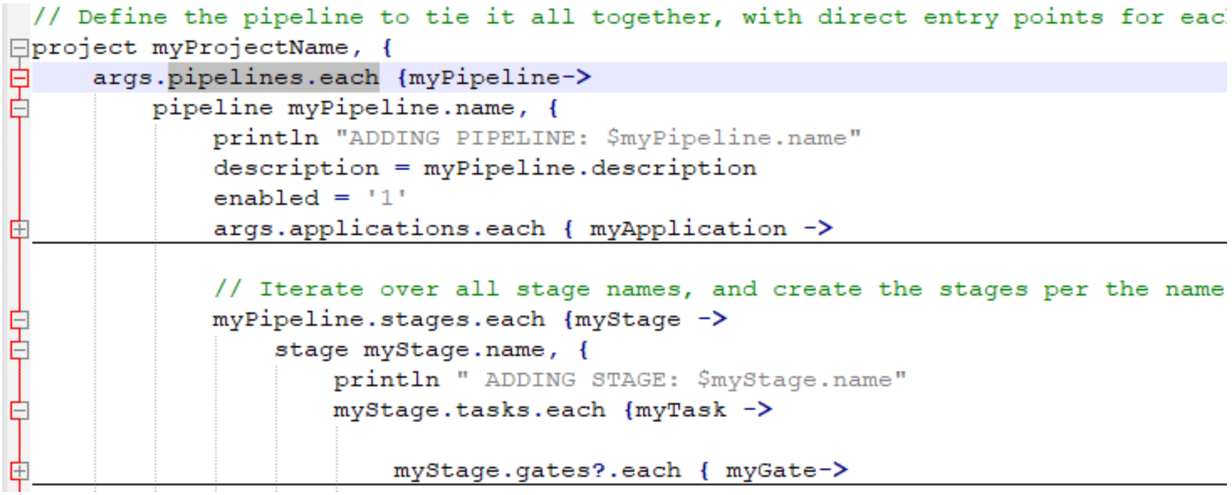
You will notice top-level structures are fairly static

- Environment Names (DEV/QA/SIT/PROD)
- Application Names (Storefront, Shopping Cart, Business Logic, etc.)
- Pipeline Stages (Dev, Test, Staging, pre-Prod, Prod)

As boilerplate entries, define them in a separate section or file



```
"pipelines" : [
    {    "name" : "OpenShift Pipeline",
         "description" : "Auto-generated pipeline",
         "stages" : [
              { "name" : "build",
              { "name" : "dev",
              { "name" : "stage",
              { "name" : "prod",
         ],
    },
    {    "name" : "OpenShift Release Pipeline",
         "description" : "Auto-generated pipeline",
         "stages" : [
              { "name" : "Dev",
              { "name" : "Stage",
              { "name" : "Prod",
         ],
    }
],
```



```
// Define the pipeline to tie it all together, with direct entry points for eac
project myProjectName, {
    args.pipelines.each {myPipeline->
        pipeline myPipeline.name, {
            println "ADDING PIPELINE: $myPipeline.name"
            description = myPipeline.description
            enabled = '1'
            args.applications.each { myApplication ->

            // Iterate over all stage names, and create the stages per the name
            myPipeline.stages.each {myStage ->
                stage myStage.name, {
                    println " ADDING STAGE: $myStage.name"
                    myStage.tasks.each {myTask ->

                        myStage.gates?.each { myGate->
```

Real-world example - 200 microservices

Helped a team onboard 200 microservices

Did NOT want to walk through a UI 200 times

The result was a data-entry exercise for a ~500 line groovy script

My biggest challenges were mapping JSON data structures into Groovy arrays and maps

Solving these programming problems lets us handle arbitrary data (size and length)

```
"serviceClusterMapping" : {"actualParameters" : [
{ "name" : "requestType", "text" : "update" },
{ "name" : "serviceType", "text" : "NodePort" } ],

serviceClusterMapping scmName, {
actualParameter =
myMap.serviceClusterMapping?.actualParameters?.
collectEntries {aParam->
[ (aParam.name) : aParam.text, ] }

-------------------------
"serviceMapDetail" : [
    {"name" : "cpuCount", "text" : "1"},
    {"name" : "cpuLimit", "text" : "2"}
]

myMap.serviceClusterMapping?.serviceMapDetail?.each
{ entry ->
    println "  ADD Detail: $entry.name = $entry.text"
    this[entry.name] = entry.text
}
```

Recognize loops and implement them

Previous examples were about scale

DevOps problems contain a lot of loops –

- Pipeline Stages
- Tasks/Steps
- Servers
- Etc.

When your solutions have an object hierarchy, you will find lists of those objects or elements

```
At work,
• For every team
  • For every application
    ▪ For every component
      ○ Define its processes
        (build/test/deploy)
    ▪ Define its pipelines
  • For every environment
    ▪ Define its
      configuration
```

Burn things to the ground and rebuild

You are in *great* shape if you can…

- destroy and rebuild with no loss
- specify a different project and it still works
- add or remove applications, servers, containers at will

```
rm –rf /path/to/output/*
```

```
deleteProject
“JPetStore”
```

In Summary

Process as Code (PaC) is a set of behaviors and disciplines your enterprise team should consider following if –

They need to define their DevOps processes quickly, efficiently, and collaboratively

Especially if –

You have problems of scale

Links

Github:
https://github.com/electric-cloud/electricflow-examples

References
https://www.linkedin.com/pulse/how-many-endpoints-electricflow-using-dsl-marco-morales

https://www.linkedin.com/pulse/why-you-should-use-process-code-domain-specific-language-morales

Q&A

Electric Cloud