**DevOps Automation Project:**

In order to achieve the desired result, I’d like to classify our project into many sub-tasks.

1. Creating the Microsoft Azure VM or Amazon EC2 instance
2. Have a centralized Version control system such as GIT to commit the code that the DevOps Engineers develop on their local work stations over to a centralized GIT repository.
3. Then, use the Git hooks – Pre-commit and Post-Receive hooks, to push the code into a puppet master and then, when the puppet agents check with the master for their respective catalogs, it will push the changes that were made to the puppet manifests.

My project will have a scope of delivering the project from the perspective of Automation and scalability using Puppet, Git Repository and Instances or VMs on a cloud or On-premises infrastructure. Please look at the picture in the second page which has a work flow of how I wanted to achieve the task.

Description of the picture below:

All of this workflow happens in a matter of seconds when a DevOPS engineer pushes the code out from his Local workstation to the Git Repository to getting the code updated on the Puppet master. When the puppet agents check in, the code will get applied to the nodes that we want to apply the puppet code to. Here’s a description that goes into a little more detail

Step 1:

The Devops engineers will work on and develop a puppet code on their **workstations**. They can spin up **vagrant** virtual boxes on their workstations and configure a local git repository to track their changes. Once they are satisfied with their code, they will commit it and push it into the Centralized repository server (**Git Repository**). The user authentication is done via Gitolite. We then use Pre-commit hooks on the GIT Server to do the validation such as syntax checking and accept the commit only when the Syntax is Valid. If it’s not, it will give them a detailed description as to where it went wrong so they can correct it and re-commit the code.

Step 2:

We will configure a couple of Post-Receive hooks which can be programmed to Send an email out to the DevOps Email Distribution keeping them informed of the code changes. It is very important and useful when you want to have a second set of eyes to do the code review when something goes wrong and you’d want to troubleshoot where it went wrong. For that you have **GitWeb**, where you can view the content of the committed file or code, see what changes were made, and track the versions from a Web User Interface.

Step 3:

We will then configure the second-post receive hook –basically a bash script, to rsync the code over to the **puppet master** whenever a change is detected. Then, when the **puppet agents** check in, they will get a catalog along with the code that was modified by the DevOPS Engineers applied to the puppet nodes

Here’s a detailed description of our DevOPs project and how I’m going to achieve the task.

**1. The EC2 Instance Part**

* Deploying an AWS EC2 instance can be achieved easily from the AWS console, Navigating via the EC2 Launch menu selecting the right instance types and filling out all the specifics needed and click Launch.
* Same goes with Creating Microsoft Azure Virtual machines where you’d login to the Azure portal, Create Virtual machine in either the resource manager or classic portal (old practice), filling out all the specifics, choosing the right size and clicking finish.
* After you create an Amazon EC2 instance, you are going to install puppet agent on it, and make it a client to the puppet master by pointing it to the puppet CA Master server by specifying it in agent section with the ‘**server’** Directive, bootstrap puppet agent so it can pull the catalog from puppet master.
* For all of these to work, you need to be able make the puppet agent communicate over to the puppet master on port **8140**. The puppet master receives inbound requests from agents on port 8140 and responds to them over it. So, make sure that the **Network Security group** assigned to the Puppet master is configured to have allow-rules for the In-bound traffic on port 8140 and same goes with puppet agents. You can configure additional layer of security with **Network ACLs** but make sure the Inbound/Egress traffic is allowed to go over port 8140.
* If the objective is achieve scalability, they can make use of **Auto-Scaling** groups in AWS and spin up virtual instances as they need depending on the application workload.

**2. The GIT Repository Part from the perspective of Automation and Scalability**

-> We have a version control system such as GIT which we use as a source for Puppet code where we can create all the manifests/modules we need, make changes to them on local workstations, keep track of changes and commit them to the GIT Centralized repository.

-> Then, we have a few **Pre-commit** and **Post-Receive GIT hooks** as described in the description of the picture drawn above to publish/rsync the code over to the puppet master.

-> The code changes can be done on a continuous basis as the application evolves. The DevOps Engineers can modify the puppet code depending on how they want to configure the application. They can make changes to **index.html** as much as they want, or create another **vhost** file in /etc/httpd/conf/ directory and call it **vhost.conf** file or make changes to the **puppet code** as in manifests and modules they want and then commit it to the GIT Repository server.

**3. Apply the code over to the nodes from Puppet Master**

When the agents check with the puppet master, the code that will be a part of the catalog that the puppet agent is going to fetch every 30 mins ( default) or however long that you configure your interval in puppet.conf gets applied.

-> Generate a Puppet module for httpd. The task of it is to install httpd package resource, generate an index.html under Docroot /var/www/html and start/restart the httpd service whenever there’s a change to the index.html.

-> Do a Regex match to apply the puppet code over to the agent nodes that we want the puppet code applied on from the Puppet Dashboard.

The below code is just one way of writing a puppet manifest and I’m not dividing it into separate sub-classes in order to keep it relevant to our project. Off-course, we can create a separate puppet module for it and scale it as you like but the crux or core of the code looks like this :

**A simple Puppet Code to be applied**:

Class httpd {

package { 'httpd':

ensure => installed,

before => File[‘/etc/httpd/conf/httpd.conf’],

before => File[‘/var/www/html/index.html’],

}

file { '/etc/httpd/conf/httpd.conf':

ensure => file,

source => 'puppet:///modules/httpd/httpd.conf/',

owner => 'root',

group => 'root',

mode => '644',

notify => Service['httpd'],

require => Package['httpd'],

}

file { '/var/www/html/index.html':

ensure => file,

source => 'puppet:///modules/httpd/index.html/',

owner => 'root',

group => 'root',

mode => '755',

notify => Service['httpd'],

require => Package['httpd'],

}

service { 'httpd':

ensure => running,

enable => true,

hasstatus => true,

hasrestart => true,

subscribe => File [‘/etc/httpd/conf/httpd.conf’],

subscribe => File [‘/var/www/html/index.html’],

}

}

The index.html file template that the above puppet code is going to pull from is below as expected.

**index.html :**

**<html>**

**<head>**

**<title>Hello World</title>**

**</head>**

**<body>**

**<h1>Hello World!</h1>**

**</body>**

**</html>**

All of the Steps I mentioned above only go over it from the DevOPS project that was asked of me. However, that’s just a trailer ( if you call it that) and there’s more to it. If you really want any of the applications scalable, we can get it done with the continuous integration and continous deployment systems such as **Jenkins** and **Hudson** CI tools.