Automatic Deployments: Create a CodePipeline

We'll cover the following

Objective

- Automatically update our application when a change gets pushed to GitHub.

Steps

- Create a CodePipeline.

Defining our pipeline

The pipeline comes in three stages:

1. The *Source* stage pulls the latest code from GitHub.
2. The *Build* stage builds the latest code with CodeBuild according to our `buildspec.yml` file.
3. The *Deploy* stage deploys the build artifacts from CodeBuild to the EC2 instances referenced in the deployment group, and starts the application according to our `appspec.yml` file.

```
Pipeline:
  Type: AWS::CodePipeline::Pipeline
  Properties:
    Name: !Ref AWS::StackName
    ArtifactStore:
      Location: !Ref CodePipelineBucket
      Type: S3
    RoleArn: !GetAtt DeploymentRole.Arn
    Stages:
      - Name: Source
        Actions:
          - Name: Source
            ActionTypeId:
```

```
       Category: Source
       Owner: ThirdParty
       Version: 1
       Provider: GitHub
      OutputArtifacts:
       - Name: Source
      Configuration:
       Owner: !Ref GitHubOwner
       Repo: !Ref GitHubRepo
       Branch: !Ref GitHubBranch
       OAuthToken: !Ref GitHubPersonalAccessToken
       PollForSourceChanges: false
      RunOrder: 1
   - Name: Build
    Actions:
     - Name: Build
      ActionTypeId:
       Category: Build
       Owner: AWS
       Version: 1
       Provider: CodeBuild
      InputArtifacts:
       - Name: Source
      OutputArtifacts:
       - Name: Build
      Configuration:
       ProjectName: !Ref BuildProject
      RunOrder: 1
   - Name: Staging
    Actions:
     - Name: Staging
      InputArtifacts:
       - Name: Build
      ActionTypeId:
       Category: Deploy
       Owner: AWS
       Version: 1
       Provider: CodeDeploy
      Configuration:
       ApplicationName: !Ref DeploymentApplication
       DeploymentGroupName: !Ref StagingDeploymentGroup
      RunOrder: 1
```

main.yml

Line #25: We don’t need to poll for changes because we’ll set up a webhook to trigger a deployment as soon as GitHub receives a change.

Now, let’s create the webhook that will trigger our pipeline as soon as a change is pushed to GitHub.

```
PipelineWebhook:
 Type: AWS::CodePipeline::Webhook
 Properties:
  Authentication: GITHUB_HMAC
  AuthenticationConfiguration:
   SecretToken: !Ref GitHubPersonalAccessToken
```

```
    Filters:
      - JsonPath: $.ref
        MatchEquals: 'refs/heads/{Branch}'

    TargetPipeline: !Ref Pipeline
    TargetAction: Source
    Name: !Sub 'webhook-${AWS::StackName}'
    TargetPipelineVersion: !GetAtt Pipeline.Version
    RegisterWithThirdParty: true
```

main.yml

We also need to make some changes to our EC2 instance to get the CodeDeploy agent installed on it.

```
Instance:
  Type: AWS::EC2::Instance
  CreationPolicy:
    ResourceSignal:
      Timeout: PT5M
      Count: 1
  Metadata:
    AWS::CloudFormation::Init:
      config:
        packages:
          yum:
            ruby: []
        files:
          /home/ec2-user/install:
            source: !Sub "https://aws-codedeploy-${AWS::Region}.s3.amazonaws.com/latest/install"
            mode: "000755" # executable
        commands:
          00-install-cd-agent:
            command: "./install auto"
            cwd: "/home/ec2-user/"
  Properties:
    ImageId: !Ref EC2AMI
    InstanceType: !Ref EC2InstanceType
    IamInstanceProfile: !Ref InstanceProfile
    Monitoring: true
    SecurityGroupIds:
      - !GetAtt SecurityGroup.GroupId
    UserData:
      # ...
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

main.yml

Line #12: The CodeDeploy agent requires `ruby`.

Line #14: Downloads the CodeDeploy agent install script to `/home/ec2-user/install` and makes it executable.

Line #18: Installs the CodeDeploy agent.

Line #29: See the next code listing for how to fill in this part.

Let's update the `UserData` section next. We need to remove the bits where we were downloading our application from GitHub because CodeDeploy will do that for us now.

```
UserData:
  Fn::Base64: !Sub |
    #!/bin/bash -xe

    # send script output to /tmp so we can debug boot failures
    exec > /tmp/userdata.log 2>&1

    # Update all packages
    yum -y update

    # Get latest cfn scripts; https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-
    yum install -y aws-cfn-bootstrap

    cat > /tmp/install_script.sh << EOF
      # START
      echo "Setting up NodeJS Environment"
      curl https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh | bash

      # Dot source the files to ensure that variables are available within the current shell
      . /home/ec2-user/.nvm/nvm.sh
      . /home/ec2-user/.bashrc

      # Install NVM, NPM, Node.JS
      nvm alias default v12.7.0
      nvm install v12.7.0
      nvm use v12.7.0

      # Create log directory
      mkdir -p /home/ec2-user/app/logs
    EOF

    chown ec2-user:ec2-user /tmp/install_script.sh && chmod a+x /tmp/install_script.sh
    sleep 1; su - ec2-user -c "/tmp/install_script.sh"

    # Have CloudFormation install any files and packages from the metadata
    /opt/aws/bin/cfn-init -v --stack ${AWS::StackName} --region ${AWS::Region} --resource Instance
    # Signal to CloudFormation that the instance is ready
    /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} --region ${AWS::Region} --resource Ins
```

main.yml

And with all of that done, we can deploy our infrastructure updates. But first, we need to delete our stack from the CloudFormation console, because the changes we've made will not trigger CloudFormation to tear down our EC2 instance and start a new one. So, let's delete our stack, and recreate it by running the `deploy-infra.sh` script.

```
./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://ec2-3-93-145-152.compute-1.amazonaws.com:8080"
]
```

terminal

> **NOTE:** Let’s run the code and also push all our infrastructure changes to our GitHub repository. Check out the `github.sh` file for that.

This code requires the following API keys to execute:

username	Not Specified...
AWS_ACCESS_KE...	Not Specified...
AWS_SECRET_AC...	Not Specified...
AWS_REGION	us-east-1
Github_Token	Not Specified...

```
aws configure --profile awsbootstrap  set aws_access_key_id {{AWS_ACCESS_KEY_ID}}
aws configure --profile awsbootstrap  set aws_secret_access_key {{AWS_SECRET_ACCESS_KEY}}
aws configure --profile awsbootstrap  set region {{AWS_REGION}}
```

At this point, our EC2 instance should be up and running with the CodeDeploy agent running on it. But the CodeDeploy agent doesn’t automatically deploy the application when it gets installed. For now, we can trigger the first deployment manually by hitting *Release Change* in the CodePipeline console. When we get to the Scaling section, we will have our EC2 instances deploy the application automatically as soon as they start.

As soon as the deployment completes, we should be able to see the “Hello World” message when we visit the URL we got after running `deploy-infra.sh`.

We can now test our automatic deployments by making a change to the “Hello

We can now test our automatic deployments by making a change to the "Hello World" message in our application. Let's change it to "Hello Cloud" and push the changes to GitHub.

```
const message = 'Hello Cloud\n';
```

server.js

```
git add server.js
git commit -m "Change Hello World to Hello Cloud"
git push
```

terminal

> **NOTE:** We have added `server.js` this time too. So before running the code, do change the message from `Hello World\n` to `Hello Cloud\n` on **Line #3**. Then you can `curl` to see if it works.

This code requires the following API keys to execute:

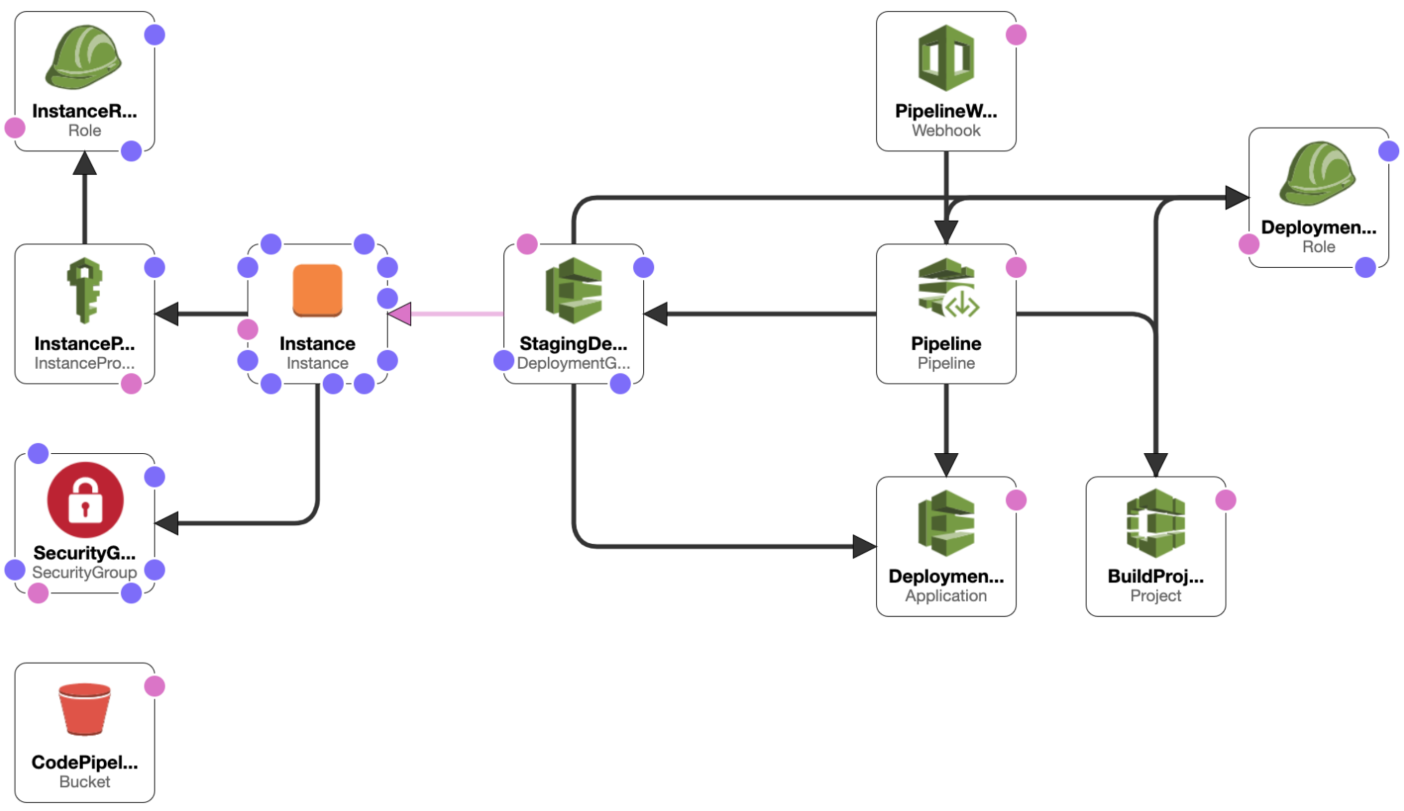
username	Not Specified...
AWS_ACCESS_KE...	Not Specified...
AWS_SECRET_AC...	Not Specified...
AWS_REGION	us-east-1
Github_Token	Not Specified...

```
const { hostname } = require('os');
const http = require('http');
const message = 'Hello World\n';
const port = 8080;
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(message);
});
  server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname()}:${port}/`);
});
```

As soon as we push the changes to GitHub, we can watch the deployment progress in the CodePipeline console. As soon as the deployment reaches the *Staging* phase, we should see "Hello Cloud" when we refresh the URL.

Our application is now getting updated automatically as soon as a change gets pushed to GitHub. And since we're now using GitHub access tokens, we can also pushed to GitHub. And since we're now using GitHub access tokens, we can also mark our repository as private.

In order to get a pictorial view of our developed cloudformation stack so far, below is the design view which shows the resources we created and their relationships.



Automatic Deployments

In the next lesson, we will run our application on more than one EC2 instance.