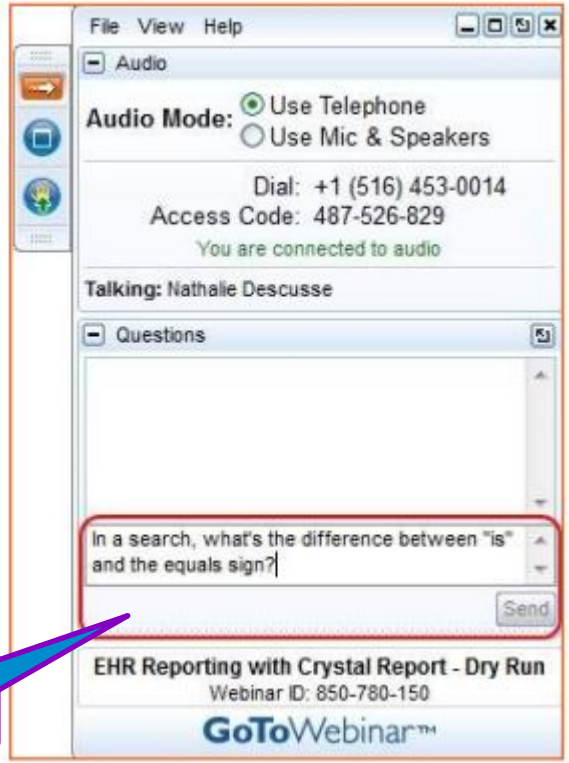
Edinburgh Devops User Group

How to deliver higher quality software, faster.

A step by step guide on creating a CI/CD pipeline

Proceedings

13.00	Open
13.05 - 13:25	CI/CD Presentation
13:25 - 13:40	CI/CD Pipeline Creation Demo
13:40-13:45	Q&A
13:45	Close



Live chat available

John Walker

- Professional Services Team Lead @ CirrusHQ
- APN Ambassador
- 6x AWS Certified
- Technology Enthusiast

- Twitter : @zz9
- I also write about tech at https://blog.johnwalker.tech/

aws partner network

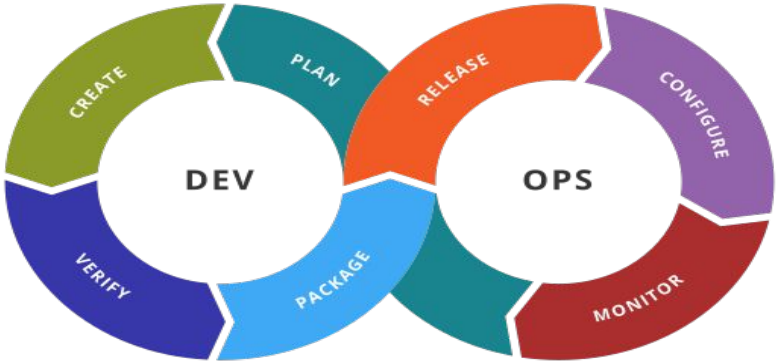
ambassador

What is CI/CD?

CI/CD stands for Continuous Integration / Continuous Deployment (or sometimes delivery, more on that difference later).

CI/CD is a key part of a DevOps lifecycle and aims to allow code or changes to applications to be built, tested and released quicker and safer than standard manual deployment processes.

In the DevOps lifecycle, the aim is to write code, test it, package it up / build it, release or deploy it, then configure and maintain the code, and feed the results back through to planning and development, and use that information to improve your applications.

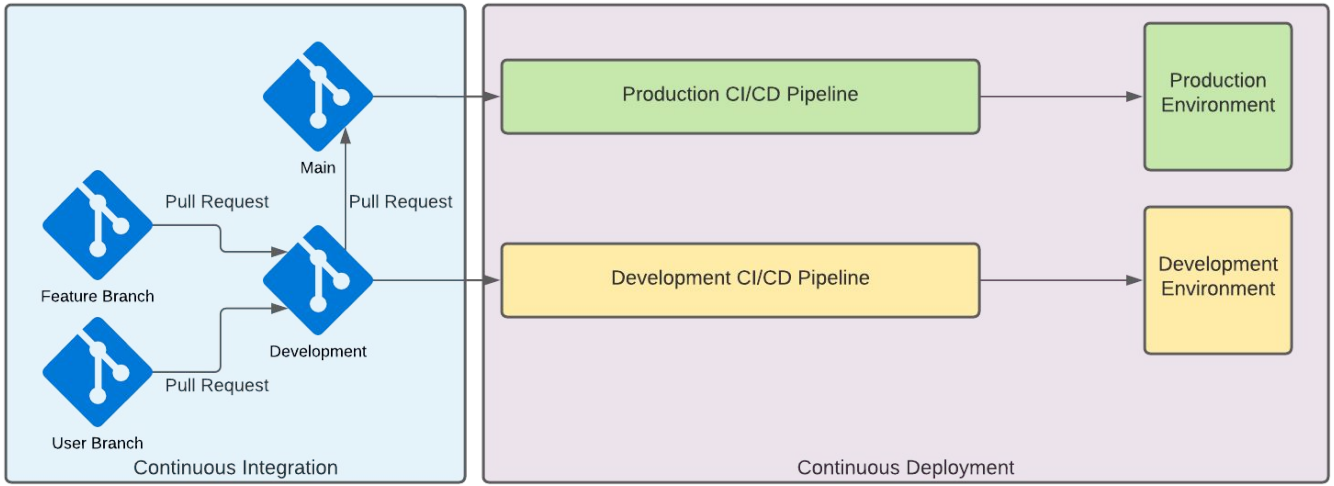


Today we are going to focus on the process from the point you have some code to deploy, through to deployment.

Continuous Integration

The CI part of CI/CD stands for Continuous Integration, which is the process of combining code from multiple contributors into a single combined code base, and then running builds and tests to ensure the code is correct. A core prerequisite for CI is version control. Git is generally the main one used (it has around 70% market share), though Mercurial still has some market share as does SVN and a few others.

In this webinar we'll use Git, and more specifically GitHub. Other hosted Git providers will also work with the setup we are going through here, such as GitLab, BitBucket, and CodeCommit.



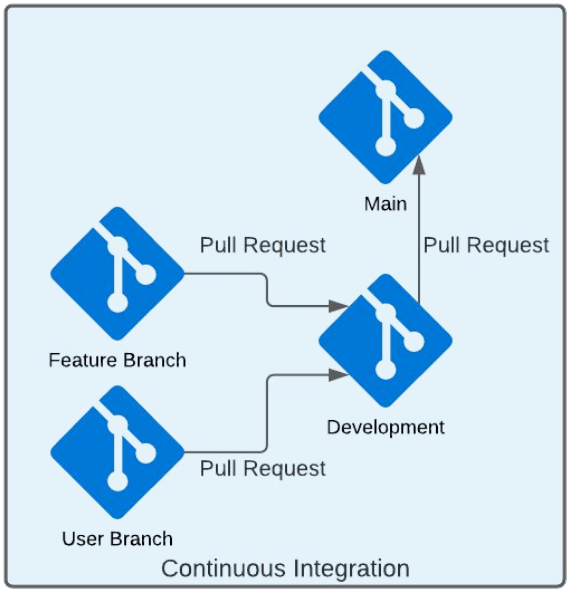
Continuous Integration

If we focus on the CI side of the previous diagram, we can explain a bit more what we mean by Continuous Integration.

There are many ways to manage source code repositories with Git, depending on your code, your team and your goals for software release. We recommend using the GitFlow model, which allows for different features and fixes to be developed and merged back into the main repositories. Atlassian (the makers of Jira, Confluence, Tempo and more) have a good guide on GitFlow here:

https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Whatever method you use to organise your code, frequent integration into the main repository for deployment is key for an agile CI/CD process. In a CI/CD pipeline a branch (main/master or development in this case) is hooked up to a pipeline that automatically deploys the code when a commit is made.

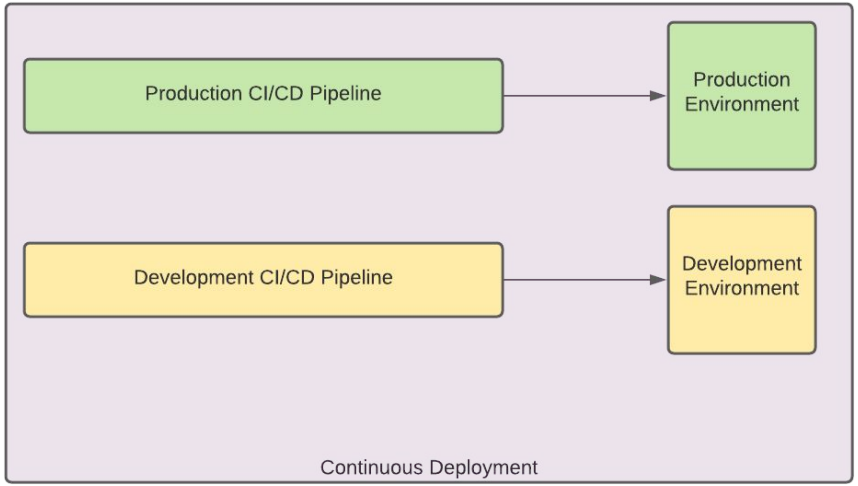


This is a key point to note, and you need to ensure your process for code commits is aligned with this. In a typical setup. any commit to master will kick off a production deployment for example. Pull Requests should be used to review and merge code and integration tests to ensure the code is of sufficient quality should be run as part of the commit process.

Continuous Deployment

The Continuous Deployment part of the CI/CD pipeline involves taking the code after it has been committed and ultimately ending up with the code deployed to your infrastructure. This same process applies whether you are deploying to Virtual Machines, Containers, Serverless compute such as Lambda or any other target.

The CD part of CI/CD sometimes also refers to Continuous Delivery. The main difference between Continuous Delivery and Continuous Deployment is whether the deployment step is automatic or manual.



Continuous Delivery aims to produce software that is ready to be deployed at any time. Continuous Deployment aims to complete that deployment as part of the same process.

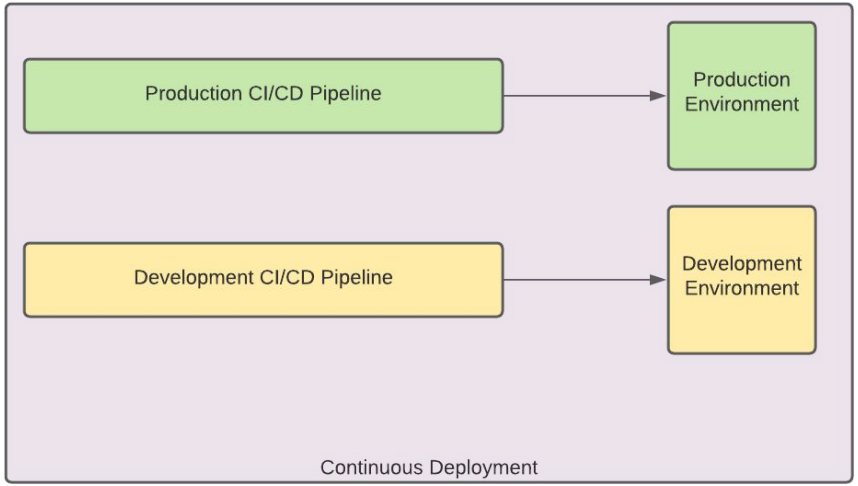
The process is largely the same, and the pipeline we'll show here can have a manual approval step before final deployment, effectively making it a Continuous Delivery pipeline instead.

Continuous Deployment

The continuous deployment process consists of a number of steps as shown in the DevOps image.

- Code is committed.
- Code is fetched from the repository
- Code is packaged up (or compiled)
- Package is uploaded to be deployed
- Package is deployed to Infrastructure

Each of these phases is customisable depending on your use case.



As an example, the deployment step can be used to deploy to a Virtual Machine (EC2), a Lambda function, or the ECS / Fargate service for Docker containers.

In the demo we will do later on, we will show using CodeDeploy to deploy a packaged up application to a specific folder on an EC2 instance, with each step happening automatically after a commit to the GitHub Repository.

What are the benefits of CI/CD?

CI/CD is a key part of the DevOps lifecycle. It allows you to perform regular deployments and ship code faster to your production environment.

Having a CI/CD pipeline also allows you to make smaller, more frequent deployments, avoiding getting caught up in merging hell, where you need to reconcile code that has been developed separately for a long time and has a long list of prerequisites to be done before you can go live.

Other Benefits:

- Faster Releases
- Automation of complex processes
- Focus on code rather than infrastructure and deployment
- Repeatable deployments
- Integration of Automated Testing
- Increased Code Quality due to more regular integration and reviews

CI/CD on AWS

There are many tools that can be used to create CI/CD pipelines, with some applications providing an all-in-one solution for creating CI/CD, and others providing the building blocks so you can customise the solution to your exact needs.

On AWS, the main tools used for creating a CI/CD pipeline are below. You can use other tools (and we'll cover those in a bit) but these are the main AWS provided tools.

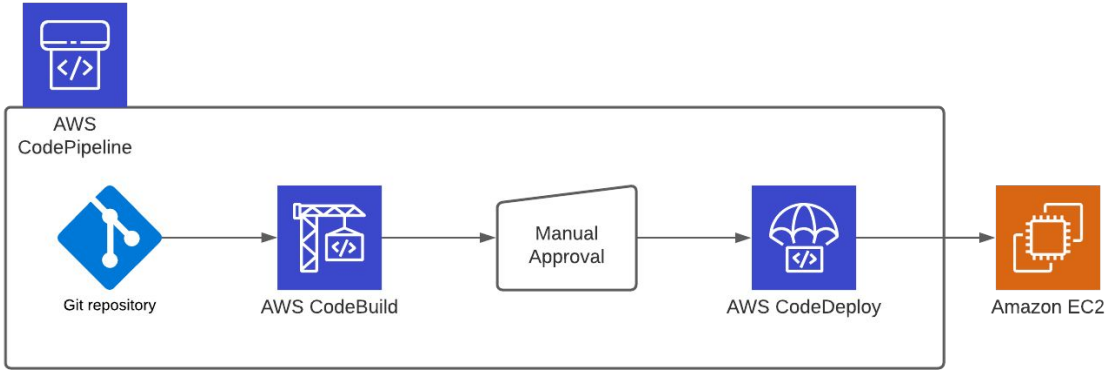
- CodePipeline - Orchestration tool
- CodeBuild - for building and compiling your application
- CodeDeploy - uses an agent to deploy code to EC2, Lambda, ECS and On-Premise instances
- CloudFormation - can be used to deploy code (and infrastructure as code) via change sets
- CodeCommit - hosted Git Repository
- S3 - for storage of artifacts (such as deployment zip files) between stages

CodePipeline is used to link these services together and can be used to define the steps that should be executed in order to deploy your application, and have manual approval steps before releases.

CodePipeline

As mentioned before, AWS CodePipeline is an orchestration tool. What that means is that it's used to organise the other AWS tools and execute them in a step-by-step manner. The CodePipeline keeps each step together as one pipeline and allows you to join up other tools to perform the deployment.

An example of a CodePipeline is shown below:



We'll be deploying something very similar to this in our Demo, just without the Manual Approval step. Some of these components could be swapped out for others, such as using CloudFormation instead of CodeDeploy.

Between each stage you can pass artifacts, such as the output of a compilation of your application. These are stored in S3 by each phase and downloaded by the next phase where required.

CodeBuild

CodeBuild is the AWS service that is used to build your code, run tests, and package your applications up, ready to be deployed.

Each CodeBuild step can be run within a pipeline (or on it's own). CodeBuild will provision, manage and scale resources for you to build out your application, using containers to run operating systems that can be customised with your specific application's build setup.

CodeBuild is also only run for the duration of your application's build process, then it is decommissioned automatically. Meaning you are only paying for the compute resource during the time it is being built, rather than having a build server running all the time.

As CodeBuild is run on standard Operating Systems you can create custom build environments on runtimes such as Node, .Net, Python, and as you have access to the underlying operating system, you can use the different phases of the build process to run commands using Bash, and even install linux packages you require during the build process.

For example, you could do the following:

- Use the .Net runtime environment
- Install the MySQL client
- In the pre_build phase, fetch your login details for a database
- In the build phase build your .Net application and a database migration script
- In the post_build step, run the migration script via the mysql command line client to perform the migration

CodeDeploy

CodeDeploy is one of the ways you can take your code from a previous step and actually deploy it onto some infrastructure.

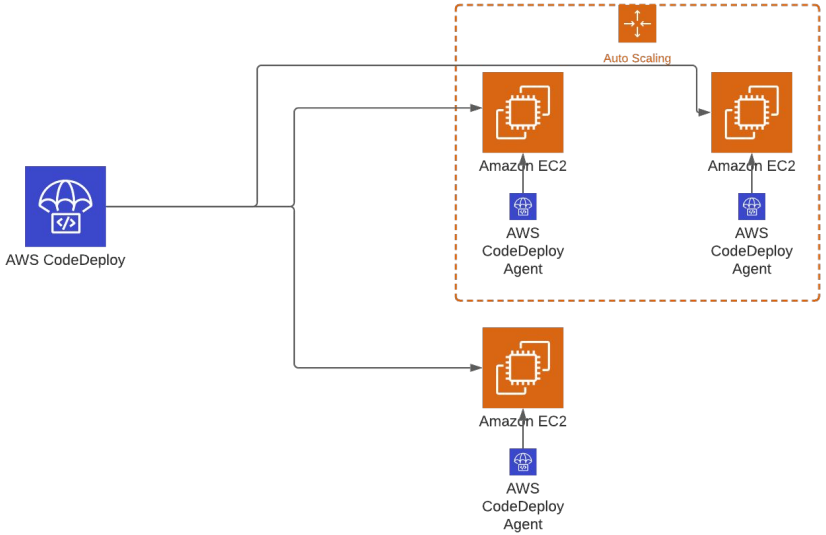
CodeDeploy does this in the case of EC2 or On-Premise hardware by using the CodeDeploy agent. This is a process that runs on the target instance listening out for deployment tasks to be done.

CodeDeploy is controlled by the use of an AppSpec file. This is a YAML or JSON formatted document telling the CodeDeploy Agent what it needs to do to deploy your application.

CodeDeploy can be used to deploy to instances in an AutoScaling Group, or single instances based on ID or Tag.

When deploying, CodeDeploy can also be used with different strategies on when code should be installed.

- AllAtOnce will attempt the deployment on all matching instances at the same time. Can also be used for Blue/Green deployments by deploying to a replacement environment, then switching traffic over.
- HalfAtATime - deploys on half of matching instances first, then the other half.
- OneAtATime - deploys on each matching instance one at a time, waiting for a successful deployment before moving on.



CodeDeploy

The AppSpec file (appspec.yml in our case) consists of a few sections. These vary slightly depending on the target (Lambda for example does not use the os section). For EC2 and On-Premise, we use the following:

- version - always 0.0
- files - contains source and destination tags (single or multiple pairs) showing where code should be deployed into.
    - Keep in mind the working directory for this process will be the CodeDeploy agent install directory, so don't use relative paths
- Hooks - these are stages of the CodeDeploy that are used to perform various tasks. These are different for each type of deployment. EC2 has the following. (I've bolded the ones we'll be using). These tasks can be commands or scripts to run.
    - **ApplicationStop - tasks to run to shut down the application**
    - DownloadBundle
    - **BeforeInstall - tasks to run before the installation of your code happens**
    - Install - This is the step that unzips your code and other files, and places it in the correct folders.
    - **AfterInstall - tasks to run after the installation of your code**
    - **ApplicationStart - tasks to run to start the application**
    - ValidateSerice
    - BeforeBlockTraffic
    - BlockTraffic
    - AfterBlockTraffic
    - BeforeAllowTraffic
    - AllowTraffic
    - AfterAllowTraffic

Other CI/CD Methods

As with many things on AWS, there are multiple ways to achieve a task, and which one to use is dependant on your specific needs for the application you are deploying.

I've listed below some of the common methods of CI/CD that can be used outside CodePipeline (or alongside) as different options, some outside of AWS.

- Jenkins - runs as a standalone server to provide automation for CI/CD
- CircleCI - Hosted option for running pipelines as containers or virtual machines
- Travis CI - Hosted option for pipelines in Virtual Machines
- TeamCity - JetBrains build management and CI / CD server
- Bamboo - Atlassian CI/CD server (Retiring 2024)
- GitLab - CI/CD pipelines in Virtual Machine, Containers or other server
- Buddy - CI/CD software based on Docker containers - aimed at speed of deployment
- CodeShip - hosted CI/CD instances
- GoCD - ThoughtWorks OpenSource CI/CD server
- Wercker - Docker based CI/CD pipelines
- Nevercode - hosted CI/CD aimed at mobile apps
- Spinnaker - CI/CD hosted platform for multi cloud deployments
- Buildbot - Python based CI framework

AWS CDK

As well as all of the CodePipeline and related tools. AWS also has the CDK (Cloud Development Kit) that can be used to automate a lot of the creation of pipelines for CI/CD.

The AWS CDK is a software development framework designed to define cloud infrastructure in code, at a higher level than CloudFormation and allow for more programmatic control, and will generate and deploy CloudFormation templates for you, allowing you to deploy your code to infrastructure you define in the same language. It can use JavaScript, TypeScript, Python, Java, .Net and Go (Go is in preview at the moment and not recommended for production usage).

AWS CDK Pipelines can be used to perform the same steps in the process as the CodePipeline, by fetching your code, building it and then deploying it.

AWS have a good blog post about using CDK Pipelines
https://aws.amazon.com/blogs/developer/cdk-pipelines-continuous-delivery-for-aws-cdk-applications/

AWS SAM

AWS also have AWS SAM, the Serverless Application Model, which can be used to create and manage fully serverless applications (covering AWS tools such as Lambda, API Gateway, Step Functions, DynamoDB and more).

AWS SAM uses the Command Line to deploy applications, either directly to each service, or it can be used to create pipelines, with much the same benefits of CodePipeline.

AWS has a good guide on setting up AWS SAM Pipelines:

https://aws.amazon.com/blogs/compute/introducing-aws-sam-pipelines-automatically-generate-deployment-pipelines-for-serverless-applications/

One difference when using AWS SAM to deploy your serverless applications is that you can use templates for CI/CD pipelines other than CodePipeline. For example you can use a Jenkins, Github Actions or GitLab as the source provider.

EC2 Image Builder

Before we go into the step by step section, I just want to quick mention a drawback of using CodeDeploy and a potential solution that is a bit more complex.

During a deployment, the code from your repository is built and deployed on to the instances that are present at that time. However, what happens if you deploy to an AutoScaling Group where old instances can be removed and new ones added?

During the creation of a new instance, the CodeDeploy agent runs through the same deployment process that happened when your code was last deployed. This means the whole process from downloading the file to installation will be repeated. This can cause some issues however, as you may have a part of your process that isn't available at some point. Say for example you have to download a file from a third-party repository, which was available when you deployed, but during a morning when you are scaling up to meet demand, it's no longer available (or even longer term, what if it gets removed?).

This also has speed implications as your application is deployed from scratch each time a server is spun up, which can cause delays.

EC2 Image Builder is designed to help with these problems. It uses a pipeline and a set of recipes to build atop a base Amazon Machine Image, and deploy your application. It then takes a new Golden Image of the resulting server, and that image can be used as the basis of spinning up new instances.

This means each new server will be the same, and will already have your application installed.

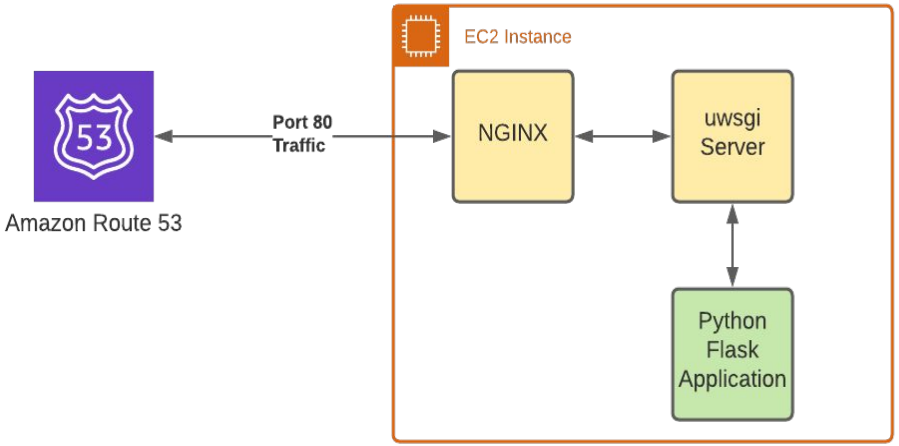
Infrastructure

For this demo, we are going to use a simple application running on an EC2 instance.

The specifics of the application itself aren't the focus here, the deployment process is, however this application is a small Python Flask application, which has a uwsgi server running to deal with traffic routing to the application, and an NGINX server to proxy requests to the application and listen on Port 80 for traffic.

The instance is a single instance, just being directly routed to from Route 53 as it's DNS.

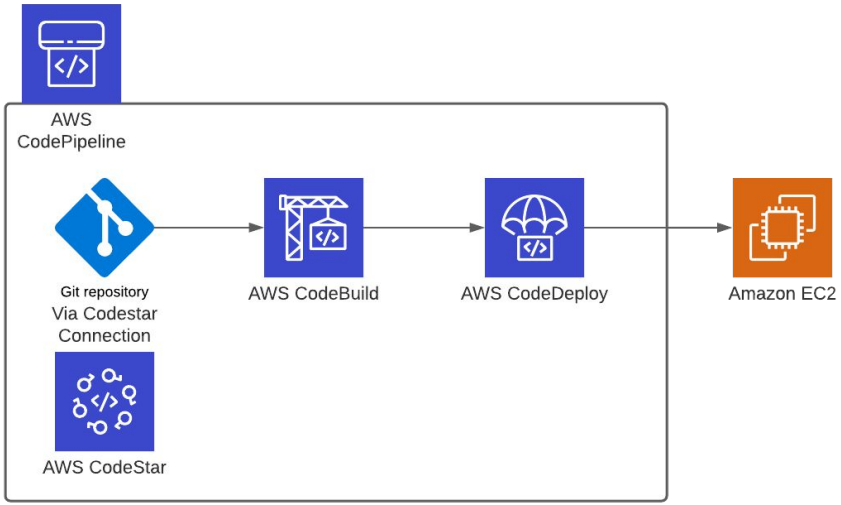
In production there would be more to this infrastructure, such as a Load Balancer and AutoScaling, but this setup will allow us to demonstrate the CI/CD Pipeline



Deployment Pipeline

In this demo we are going to deploy a simple application via a CodePipeline. Below is the architecture of what we are going to set up.

We use a Codestar Connection to connect our account to GitHub, then we will set up a CodePipeline to fetch the source code from GitHub. From there we will run a CodeBuild that will zip up the files we require for our deployment and hand those files over to CodeDeploy. CodeDeploy will then communicate with the agent on the EC2 instance we are using and deploy the code.



Demo Steps

To build our pipeline we will be going through a number of steps in an AWS account to manually create our deployment pipeline and deploy our application.

This demo won't include the setting up of the EC2 instance initially but this will be included in the templates we send out after this webinar.

1. Set up a Codestar Connection to GitHub for our repository
2. Set up a CodeDeploy Application
3. Set up a CodeDeploy Deployment Group
4. Set up CodePipeline
    a. Set up Source Step
    b. Set up CodeBuild Step
    c. Set up Deploy Step
5. Deployment will be initiated
6. Once Deployment is finished, a code change will be made to demonstrate it flowing through automatically

Demo Time

CI/CD Pipeline

Deploying a small application to EC2

Presented by John Walker, CirrusHQ