GitLab-CI

GitLab-CI is our preferred deployment pipeline technology for AWS deployments. Octopus Deploy will remain for pet servers. Pace of migration of Octopus to GitLab-CI follows pace of migration from pets to AWS cattle.

How to Enable Your Project to Utilize GitLab-CI for Deployment

To *enable* your project for GitLab-CI deployment you'll have to add your repository to the whitelist here. Adding your project to this whitelist will enable the automation that was created for applying the settings required to meet our SOC auditing standards. This automation accomplishes the following things:

- Enables merge request approvals if not already set, requiring at least 1 approver for merge
- Prevents overriding the approvals
- Prevents push to any protected branches
- Enables the project on the production GitLab-CI runners.

The automation runs via the Nomad job cis-gitlab-protect-whitelist-projects on a schedule of once per hour.

What is Required to Meet Our Auditing Standards

A report can be run on demand to pull a list of jobs deployed to our production environment and the following outlines what is required for us to be able to do deployments through GitLab-CI.

- All changes deployed to production must have gone through a merge request with at least one approver
- The settings applied by the whitelist automation ensures that your only way to get changes into a protected branch are merged and approved
  - Additionally the whitelist itself requires a merge with one approver to ensure any project added to it has been reviewed
- Additionally there is an alarm setup in CloudWatch to notify our auditing team when project settings are set to inappropriate levels
- Another alarm was created in SumoLogic to notify the auditing team when a new protected branch is created to stop un-reviewed changes from possibly making it to production without

notice

- You must fill out a Change Request Form (CR) and get PO approval before proceeding with any changes to production
- Releaser: This needs to be the person running the jobs
- Deployment Procedure: This needs to have a link to the pipeline(s) and/or job(s) you will be running
- Recovery Procedure: This needs to have rollback instructions, please note a "roll forward" situation
- Verification Procedure: You should provide sufficient instructions for verifying your change was successful
- Note it is the 'releasers' responsibility to move the CR into a released state with the appropriate notes
- In the case of an Emergency or Roll Forward Scenario you must get verbal/text PO approval before proceeding
- Any change done during this time needs to be documented in a CR, marked as Emergency, and then submitted by the end of the next business day
- Your original change also needs to be moved to the 'Released With Issues' or 'Not Released With Issues' with an appropriate note and link to the new emergency CR

Why GitLab-CI > Octopus/Jenkins

While Octopus deploy was a fine solution for our on-premises deployments to pet servers, our strategy of cattle servers for cloud development has made working with Octopus cumbersome. Specifically:

- Deployments of AWS infrastructure and code don't go to the target servers, they go to a deployment bastion server which actually does the deployment
- Use of on-premises GitLab and Jenkins were not previously in scope of our SOC audit. With infrastructure as code now deploying server permissions, this would have brought GitLab on-premises and Jenkins into scope for SOC audit. We built the new GitLab with this in mind to replace both on-premises GitLab and Jenkins.
- Deployment scripts aren't versioned along side the rest of the system code.
- Systems running in AWS can run in IAM roles to get the permissions they need. Deploying from on-premises would mean that we would have to deploy/cycle sensitive AWS credentials to deployment agents.
- GitLab-CI builds can run in containers meaning that teams can isolate their build dependencies. Jenkins, e.g., requires most dependencies be installed at the OS level. There have been previous discussions about teams maintaining their own build servers and this is a much lighter-weight solution to this.

Gitlab CI Runners

GitLab CI runners are where your build / deploy jobs actually run. There are many different runner configurations that can be explored in GitLab's documentation: (https://docs.gitlab.com/ee/ci/runners/)

Gitlab CI Runners Guidance on Usage

GitLab CI pipelines are *only* to be used to deploy infrastructure and software.

- Make the CI/CD pipeline the only way to deploy changes to production.
- Maintain parity between pipelines for production, testing and sandbox whenever possible.
- Do not create a pipeline to execute a batch process or report. These should be automated in a batch process.
- Production should only be deployed when test stages are successful.
- Only allow production to be deployed to from a protected and whitelisted branch.
- *Never* abuse a pipeline to experiment/test in production.
- Testing can be done against local, Sandbox or Testing account.

Runner Tagging

CI jobs MUST target runners based on tags. Depending on the development tier and required build dependencies, different runners should be targeted.

All runners are configured with one of EACH of the following tags:

- AWS account: `sandbox` , `testing` , `production`
- Region: `us-east-1` , `us-west-2`
- Type: `docker` , `windows-shell` (see Executors below)

In your `.gitlab-ci.yml` , configure the runner for each job. Example:

```
plan_sandbox
  stage: deploy
  script:
    - cd accounts/$ACCOUNT/
    - terraform init  -input=false
    - terraform plan -input=false -auto-approve
  only:
    - master
  tags:
    - sandbox
    - us-east-1
    - docker
```

Executors

Each runner is set up as a specific type of executor. To control the type of executor your build runs on, use the following tags. If a tag is not specified, then any runner may be used.

- `docker` - Docker executor
- `windows-shell` - Shell executor on Windows

Generally, the `docker` executor should be used for most builds. This gives you an isolated and customizable build environment for your CI pipeline. The deployment bastions in each account/region are setup with a runners configured as a Docker executor.

The `windows-shell` executor is also available for some scenarios, specifically building Windows Docker containers. It also provides support for the following tools and activities:

- Git
- MSBuild v15
- Nuget
- Docker
- Pushing to ECR

See this repository for an up-to-date list of runner features.