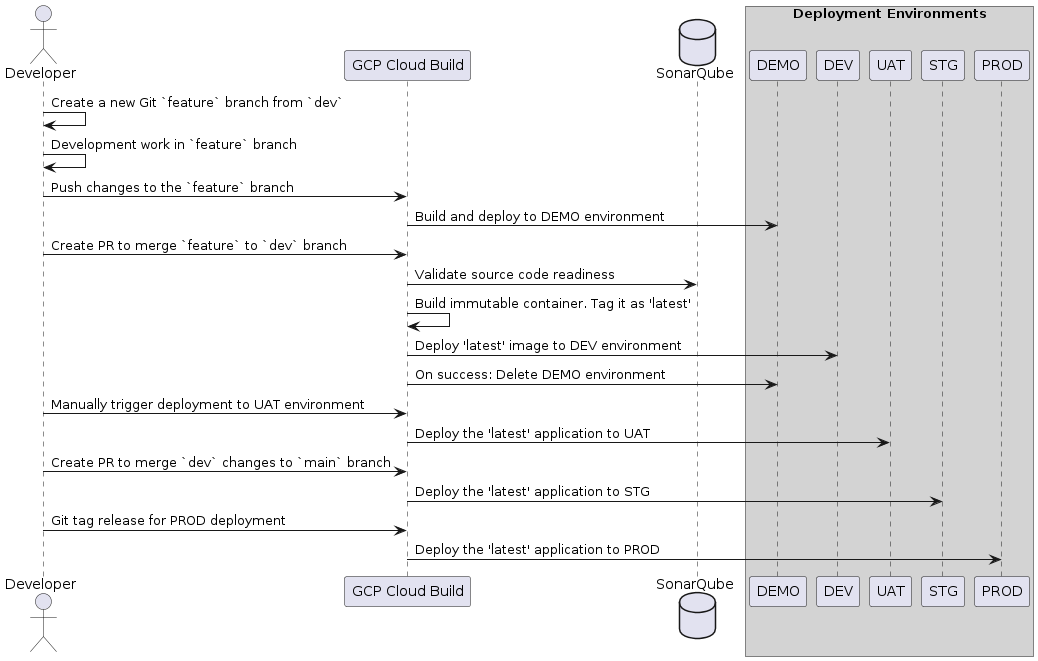
# Getting started with Google Cloud Build

## Code to Release Workflow

This diagram represents implemented “Code to Release” Workflow.



Releasing Workflow

## Create initial source-code project

### Authenticate with GitHub

gh auth login  
  
# or alternatively use the environment GITHUB\_TOKEN  
  
export GITHUB\_TOKEN='ghp\_xxx'

### Configure Git

git config --global user.email "your\_email@example.com"  
git config --global user.name "Your Name"

### Obtain the source code

#### Create it

gcloud storage cp -r gs://pls-resource-bucket/tldr-everything .  
cd tldr-everything

#### Clone it

git clone https://github.com/larkintuckerllc/hello-nodejs-typescript.git  
  
cd hello-nodejs-typescript  
rm -rf .git

### Initialize local Git repo

git init  
git checkout -b main  
git add .  
git commit -m "Initial commit"

### Create new private repository on GitHub

gh repo create hello-nodejs-typescript --private

### Push the new branch upstream (‘-u’) to the Remote Repository

Replace my username (dzivkovi), GitHub repo and Git project specific info by yours:

git remote add origin https://github.com/dzivkovi/hello-nodejs-typescript.git  
git branch -M main  
git push -u origin main

## Define your Git Workflow

### Switch to the main branch, to ensure we’re starting from the right place

git checkout main

### Create the Development branch for development work and push it to GitHub

git checkout -b dev  
git push -u origin dev

~~### Create the branch for User Acceptance Testing and push it to GitHub~~

~~sh~~ ~~git checkout -b uat~~ ~~git push -u origin uat~~ ~~~~

~~### Create the branch for the Staging environment and push it to GitHub~~

~~sh~~ ~~git checkout -b stg~~ ~~git push -u origin stg~~ ~~~~

## Development Work

~~### Get the latest code from ‘dev’ branch~~

~~sh~~ ~~git checkout dev~~ ~~git pull # Ensure you have the latest changes~~ ~~~~

### Create a new branch for your feature

git checkout -b feature/my-new-feature

#### Note

This command is correct for creating a new feature branch from the current branch, which will be dev if the previous steps were followed correctly. However, if there is any concern about the user possibly not being on dev (due to manual errors or interruptions), explicitly specifying the parent branch can avoid such issues:

git checkout -b feature/my-new-feature dev

### Make sure you have the latest code from the ‘dev’ branch

Rebase option ensures you changes (if any) are kept on top:

git pull origin dev --rebase

### Do your work

### Committing changes in feature branch

Add your changes and commit them:

git add .  
git commit -m "Add my new feature"

Push your feature branch to the remote repository:

git push -u origin feature/my-new-feature   
# The '-u' flag sets the upstream branch

### Creating a Pull Request

Create a pull request (PR) from your feature branch to dev using the GitHub CLI (or GUI):

gh pr create --base dev --head feature/my-new-feature \  
 --title "My New Feature" \  
 --body "Description of my new feature"

#### Review and Merge Pull Requests

After your pull request is reviewed and approved, merge it through the GitHub website. Choose the appropriate merge strategy (Merge commit, Squash and merge, Rebase and merge).

### Post-Merge Cleanup

After the pull request is merged:

1. **Update Your Local ‘dev’ Branch**: Switch to the dev branch and pull the latest changes.

* git checkout dev  
  git pull

1. **Delete the Local and Remote Feature Branches**: Clean up your branches after the merge.

* # Delete the local branch  
  git branch -d feature/my-new-feature  
    
  # Delete the remote branch  
  git push origin --delete feature/my-new-feature

Deleting the feature branch locally and remotely after the merge helps keep the repository clean and manageable. Ensure that the pull request is fully merged before attempting to delete the branches to avoid losing work.

### Continue Development Work

Ready to start on the next feature:

git checkout dev  
git pull # Ensure you have the latest changes  
# Optionally start a new feature branch  
git checkout -b feature/my-next-feature

## Development Branch Protection Rules

To prevent direct changes to the dev branch, you would need to set up branch protection rules in your Git repository. The process for setting up these rules varies depending on the platform you’re using (GitHub, GitLab, Bitbucket, etc.). Here’s a general guide on how to do it:

### GitHub

1. Go to your repository and click on “Settings”.
2. Click on “Branches” in the left sidebar.
3. In the “Branch protection rules” section, click “Add rule”.
4. In the “Branch name pattern” field, enter dev.
5. Check “Require pull request reviews before merging”.
6. Click “Create” to save the rule.

### GitLab

1. Go to your project and click on “Settings”.
2. Click on “Repository” in the left sidebar.
3. Scroll down to “Protected Branches”.
4. In the “Branch” field, select dev.
5. In the “Allowed to push” and “Allowed to merge” fields, select the roles you want to allow.
6. Click “Protect” to save the rule.

### Bitbucket

1. Go to your repository and click on “Settings”.
2. Click on “Branch permissions” under “Workflow”.
3. Click “Add a branch permission”.
4. In the “Branch name or pattern” field, enter dev.
5. Under “Write access”, select the users or groups you want to allow.
6. Click “Save” to save the rule.

## Build/Deploy Process

### Prepare the GCP environment

One-off activities, that might have already been performed by your GCP or Project admins:

# Enabled GCP Services  
gcloud services enable \  
 cloudbuild.googleapis.com artifactregistry.googleapis.com \  
 run.googleapis.com eventarc.googleapis.com \  
 logging.googleapis.com  
  
# Create the repository for docker images   
export REPOSITORY=r2d2  
gcloud artifacts repositories create ${REPOSITORY} \  
 --repository-format=Docker --location ${REGION}

### Set up the environment

# Set the project ID and region  
export PROJECT\_ID=$(gcloud config get-value project)  
export PROJECT\_NUMBER="$(gcloud projects describe ${PROJECT\_ID} --format='get(projectNumber)')"  
  
export REGION=$(gcloud config get-value compute/region)  
if [ "$REGION" = "(unset)" ] || [ -z "$REGION" ]; then  
 export REGION="us-central1"  
 gcloud config set compute/region $REGION  
fi  
echo "Region set to: $REGION"  
  
# See artifact registry repository content  
export REPOSITORY=r2d2  
gcloud artifacts docker images list \  
 $REGION-docker.pkg.dev/$PROJECT\_ID/$REPOSITORY \  
 --include-tags

### Use Cloud Build Service account (SA)

See the “Using minimal IAM permissions” section in the [Deploying to Cloud Run using Cloud Build](https://cloud.google.com/build/docs/deploying-builds/deploy-cloud-run#continuous-iam) page.

#### Default SA

SERVICE\_ACCOUNT=cloudbuild  
  
gcloud projects add-iam-policy-binding ${PROJECT\_ID} \  
 --member=serviceAccount:${PROJECT\_NUMBER}@${SERVICE\_ACCOUNT}.gserviceaccount.com \  
 --role=roles/artifactregistry.writer

#### Create a new SA

If you don’t have permission to update Cloud Build Service Account, create a new one:

SERVICE\_ACCOUNT=my-cloudbuild-sa  
  
gcloud iam service-accounts create ${SERVICE\_ACCOUNT} \  
 --description="Temporary Service Account" \  
 --display-name="${SERVICE\_ACCOUNT}"  
  
gcloud projects add-iam-policy-binding ${PROJECT\_ID} \  
 --member="serviceAccount:${SERVICE\_ACCOUNT}@${PROJECT\_ID}.iam.gserviceaccount.com" \  
 --role="roles/iam.serviceAccountUser" \  
 --role="roles/cloudasset.owner" \  
 --role="roles/storage.admin" \  
 --role="roles/logging.logWriter" \  
 --role="roles/artifactregistry.admin" \  
 --role="roles/compute.admin"

## Delivery Pipeline Setup

### Setup Branch Triggers

We enable developers to demo their work via a Cloud Run deployment named after their Git branches. Push to any branch not named ^(main|master|dev|develop)$ will trigger a branch-named deployment.

* Set up the triggers:
* # Push to a branch event to build and deploy DEMO environments  
  gcloud beta builds triggers import --source=.ci/trigger-branch.yaml --region=$REGION

### Triggers to create DEV, UAT, STG, PROD environments

* Pull request event to build immutable image and deploy it DEV environment:
* gcloud beta builds triggers import --source=.ci/trigger-pr-dev.yaml --region=$REGION
* Manual trigger to promote ‘latest’ image to UAT:
* gcloud beta builds triggers import --source=.ci/trigger-manual.yaml --region=$REGION
* This step requires approval.
* Pull request event to deploy pre-production STG environment:
* gcloud beta builds triggers import --source=.ci/trigger-pr-main.yaml --region=$REGION
* Push new tag event to deploy PROD release from tested image:
* gcloud beta builds triggers import --source=.ci/trigger-tag-prod.yaml --region=$REGION
* It also requires approval.

### Review the triggers

gcloud beta builds triggers list \  
 --format="table(name, id, filename, description)" --region $REGION

or by going to the [Cloud Build Triggers page](https://console.cloud.google.com/cloud-build/triggers) in the Cloud Console.

## Handling Secrets

Cloud build now supports [Secret Manager](https://cloud.google.com/cloud-build/docs/securing-builds/use-secrets) for accessing sensitive data. This is the recommended way to store and access.

### Prepare the environment

* Before you can use secrets in your builds, you need to grant the Cloud Build service account the necessary permissions to access the secrets:
* gcloud projects add-iam-policy-binding $PROJECT\_ID --member=serviceAccount:$PROJECT\_NUMBER@cloudbuild.gserviceaccount.com --role=roles/secretmanager.secretAccessor
* Create your secrets. E.g. this POC will need a SonarQube API key used:
* echo -n "$SONAR\_TOKEN" | gcloud secrets create SONAR\_TOKEN --replication-policy=automatic --data-file=-

### Validate your secret is in the Secret Manager

gcloud secrets versions access latest --secret=SONAR\_TOKEN

## Manually Triggering Builds

CI/CD Pipeline is triggered by pushing changes to the repository, pull requests, and release tagging, but you can also run/validate individual steps from the command line.

1. Feature branch validation and demo deployments:

* gcloud builds submit --config .ci/cloudbuild-dev.yaml

1. Test the code, build the immutable container image, and deploy it to the DEV environment:

* gcloud builds submit --config .ci/cloudbuild-dev.yaml

1. UAT environment deployment:

* gcloud builds submit --config .ci/cloudbuild-uat.yaml

1. Staging environment deployment:

* gcloud builds submit --config .ci/cloudbuild-stg.yaml

1. Production environment deployment:

* gcloud builds submit --config .ci/cloudbuild-prod.yaml

## Automated Pipeline Flow

1. When you push changes to the repository, the pipeline will be triggered automatically. The pipeline will build and deploy the application to the DEMO environment.
2. Once happy with the changes, create a pull request to merge the changes to the dev branch. The pipeline will build and deploy the application to the DEV environment.
3. When the QA team is ready to test the DEV release, execute the manual trigger to deploy the application to the UAT environment:

* git checkout dev  
  git pull  
    
  gcloud builds submit --config .ci/cloudbuild-uat.yaml

1. Once the QA team has tested the application, the code is ready to be merged from dev to main branch. Create a pull request to merge the changes to the main branch. The pipeline will build and deploy the application to the STG environment. E.g.:

* gh pr create --base main --head dev \  
   --title "Ready for Staging" \  
   --body "Describe why the app is ready to be merged into 'main' branch and deployed to the STG environment"

1. After the STG environment is tested, create a tag to trigger the deployment to the PROD environment. E.g.:

* git checkout main  
  git pull  
    
  git tag -a v1.0.0 -m "Your message about this release"  
  git push origin v1.0.0

## Testing Deployments

After the build completes, you can test the deployment by accessing the application URL, which is based on the branch name. For example:

* the URL for the feature/my-new-feature branch would be https://appname-feature-my-new-feature-something.run.app,
* the URL for the dev branch would be https://appname-dev-something.run.app.

When Cloud Run deployments do not use --allow-unauthenticated flag (which is recommended), you need to authenticate to access the URL. Here is an example using curl and the gcloud CLI to authenticate and access the URL:

# Here, the Appname is composed of Cloud Run service ('java-11-hw'), hyphenated ('-') with sanitized branch name ('dev'): java-11-hw-dev  
APP\_URL=$(gcloud run services describe java-11-hw-dev \  
 --platform managed --region $REGION --format 'value(status.url)')  
echo $APP\_URL  
curl -X GET -H "Authorization: Bearer $(gcloud auth print-identity-token)" $APP\_URL/books

## Learn more

* [Cloud Native Automation with Google Cloud Build](https://www.packtpub.com/product/cloud-native-automation-with-google-cloud-build/9781801816700)
* [Code examples used in the official Cloud Build documentation](https://github.com/GoogleCloudPlatform/cloud-build-samples)