- If you don't define stages and just write a yml file like the following:

```
build laptop:
    image: alpine
    script:
        - echo "Building a laptop"
        - mkdir build
        - touch build/computer.txt
        - echo "Mainboard" >> build/computer.txt
        - cat build/computer.txt
        - echo "Keyboard" >> build/computer.txt
        - cat build/computer.txt

test laptop:
    image: alpine
    script:
        - test -f build/computer.txt
```

    - (these 2 are **jobs**)
    - It'll fail because there is no execution order of the jobs. Any job can run first. So, it'll fail.
    - You need to mention the stages like

```
stages:
    - build
    - test

build laptop:
    image: alpine
    stage: build
    script:
        - echo "Building a laptop"
        - mkdir build
        - touch build/computer.txt
        - echo "Mainboard" >> build/computer.txt
        - cat build/computer.txt
        - echo "Keyboard" >> build/computer.txt
        - cat build/computer.txt

test laptop:
    image: alpine
    stage: test
    script:
        - test -f build/computer.txt
```

        - It'll fail because the docker image will be destroyed after the first job. The second job (test laptop) is pulling one more docker image and testing there. So, it failed.
    - Its solution is **artifacts**.
        - It is like a file/folder produced by a job that GitLab saves and makes available for the next job.

```
stages:
    - build
    - test

build laptop:
    image: alpine
    stage: build
    script:
        - echo "Building a laptop"
        - mkdir build
        - touch build/computer.txt
        - echo "Mainboard" >> build/computer.txt
        - cat build/computer.txt
        - echo "Keyboard" >> build/computer.txt
        - cat build/computer.txt
    artifacts:
        paths:
            - build


test laptop:
    image: alpine
    stage: test
    script:
        - test -f build/computer.txt
```

- Now it'll pass

```
Uploading artifacts for successful job
Uploading artifacts...
build: found 2 matching artifact files and direc
Uploading artifacts as "archive" to coordinator.
esponseStatus=201 Created token=68_YRiwt1
```

(building job)

```
Skipping Git submodules setup
$ git remote set-url origin "${CI_REPOSITORY_URL}" || echo 'N
Downloading artifacts
Downloading artifacts for build laptop (10748848816)...
Downloading artifacts from coordinator... ok        correlati
0 OK token=68_6pCVF2
```

(testing job)

- Coordinator means the GitLab server only, after the job get finished, it tells the GitLab server to save the particular file as it'll be used in the next job.

```
build laptop:
    image: alpine
    stage: build
    variables:
        build_file_name: laptop.txt
    script:
        - build_file_name=laptop.txt
```

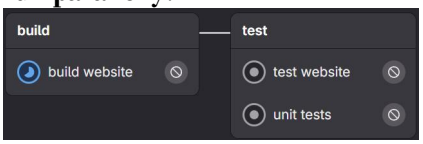
- These are 2 ways of creating variable. NOTE: inside **variables**, its not a list, its kind of key value pair.

```
stages:
  - build
  - test

variables:
    build_file_name: laptop.txt
```

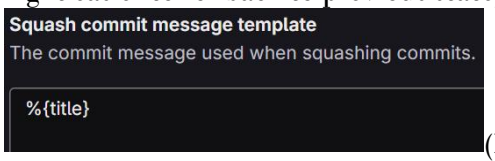
(global variable)

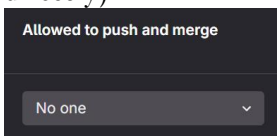
- **Setting >> repository >> Branch rules** (you can set here like no one will be able to push to main branch like things)
- Always select correct **image** so that the dependencies are there inside that image.

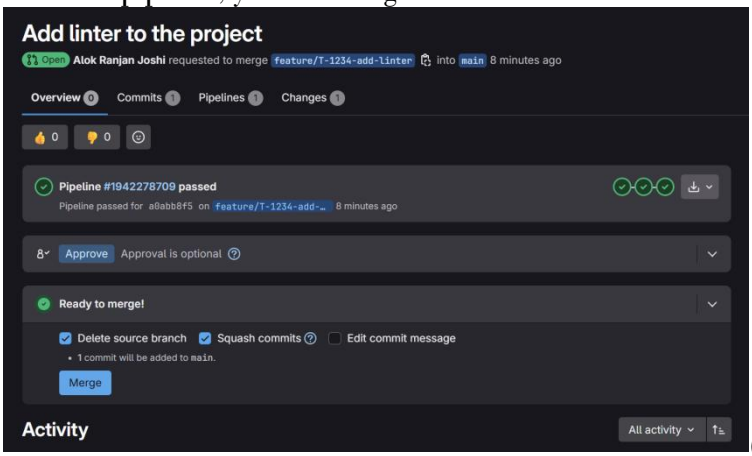
```
stages:
  - build
  - test

build website:
  image: node:lts-alpine
  stage: build
  script:
    - yarn install
    - yarn build
  artifacts:
    paths:
      - build

test website:
  image: alpine
  stage: test
  script:
    - test -f build/index.html

unit tests:
  image: node:lts-alpine
  stage: test
  script:
    - yarn install
    - yarn test
```

- In here, the **build website** is having the stage **build**, so it'll run first.
- After that, **test website** and **unit tests** are having the stage **test**, so they both will run **parallelly**.
- 

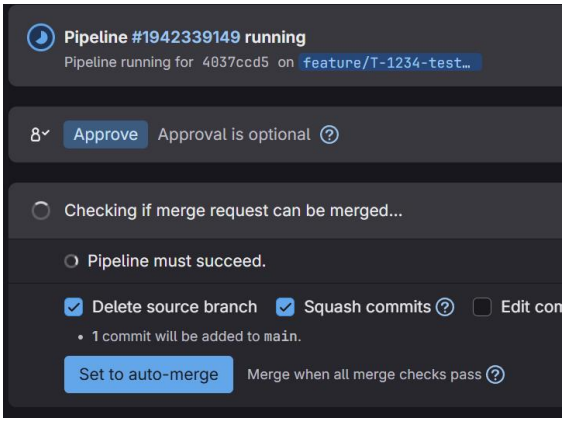
➢ Merge Requests:

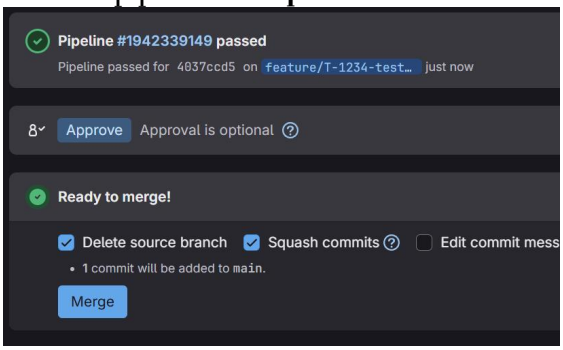
- The changes should not be pushed to the main branch directly by anyone.
- If multiple people are pushing to the main branch, then at some point if the build fails, then no one will be able to continue.
- **Settings >> Merge requests**
  - **Merge method: fast- forward merge** (it directly takes the pointer of the source branch to the destination branch without creating any merge commit leading to a cleaner commit history)
  - **Squash commit where merging: encourage** (it combines all the commits that has been made in the source branch into a single meaningful commit, making it easier to rollback to previous state)
    - 

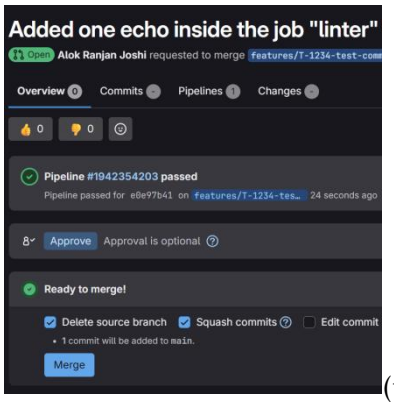
(by default merge request title)
  - **Merge checks: pipeline must succeed** (if the pipeline fails, merge will not happen)
- **Settings >> Repository >> Protected branches**
  - (we want that no one will be able to push the changes to the main branch directly)
  - 

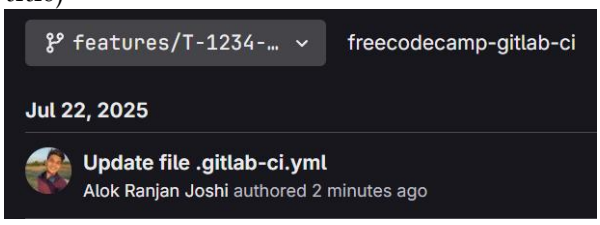
(make it **no one**)
- Now, after changing the rules and settings, try to make some changes and commit to main branch. You'll not be allowed now. So, you have to create one new branch, commit in that new branch & create one MR (merge request).
- The pipeline will always run when you are trying to commit some changes to any branch (provided the rules are not there to run it for any particular branch).
- Then, its depends upon the settings that you have done that even after the failure of pipeline, you can merge or not.
- 

(here its showing that the pipeline passed)

- The people can review that and **approve** this merge request, there might be your team's internal rule that someone has to review this before merging.
- 

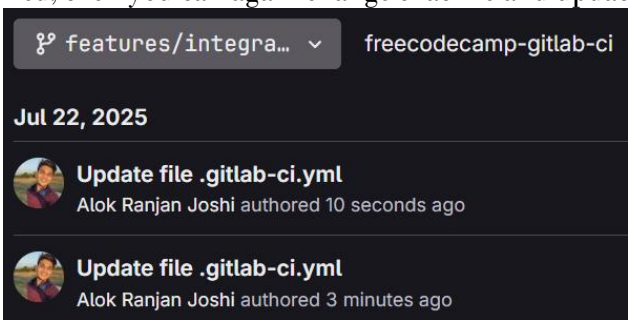
  - As I have set the **pipeline must succeed**, so the merge button is not there. The pipeline must **pass** so that it can be **merged**.
- 

  - After the pipeline passed, the merge button appeared.
- **NOTE**
  - When you do some changes and commit in GitLab, the commit message is by default ***"Update file .gitlab-ci.yml"***.
  - Then when you create MR, you add some comments. That is not the commit, that is the MR title.
  - 

(this is the merge request having the given title)
  - 

(it is the commit which title was automatically added by GitLab)
  -

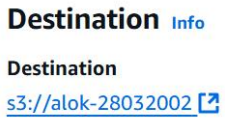
- So, now as the commit was done to the new branch, the pipeline will run once.
- Now, when you merge that change to the destination branch (for me **main**), the title of the **MR** will be the **commit message** (as I have enabled the squash commit) for the destination branch. If the **squash** was not enabled, then the same commit of the source branch would have transferred to the destination branch.
- **However, as one commit happened, so the pipeline will run once again now.**

- Some pre-defined stages are there in GitLab **CI** i.e. **.pre** (setup stage), **build**, **test**, **deploy**, **.post** (cleanup stage).
  - So, even if you don't mention these inside the **stages**, it'll work properly.
- If you have created a MR after changing something, and there are some more changes required, then you can again change that file and update the same MR.
  - 

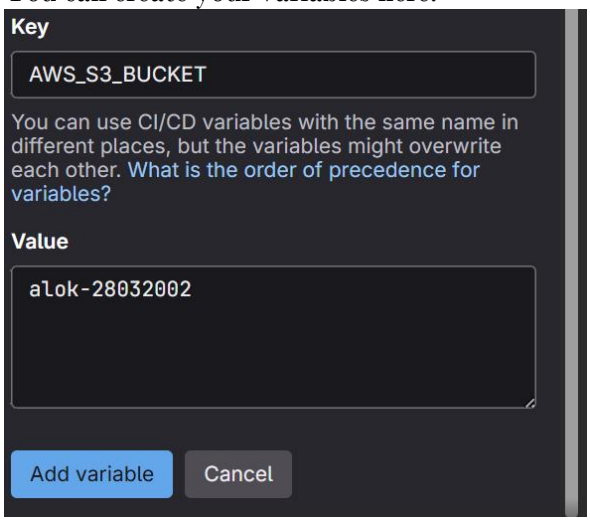
  - Just one more commit will happen to the new branch.
  - The merge request will be same, just the commits will be created in that new branch when you change something and commit.
-
```yaml
test website:
    image: node:lts-alpine
    stage: test
    script:
        - yarn global add serve
        - serve -s build
```
  - There is one issue here, I started the server here which will not stop till the job will be timed out (deffault: 1 hour)
  -
```yaml
test website:
    image: node:lts-alpine
    stage: test
    script:
        - yarn global add serve
        - apk add curl
        - serve -s build &
        - sleep 10
        - curl http://localhost:3000 | grep 'React App'
```
    - **&** means run the command in the background (it is the real game changer)
-
```yaml
deploy to s3:
    stage: deploy
    image:
        name: amazon/aws-cli:2.27.57
        entrypoint: [""]
    script:
        - aws --version
```
  - Here, we have to give the name & entrypoint of the image bcs its default entry point is "aws" (mentioned in the dockerfile)
  - It means, whatever command we'll execute, it'll add **aws** as the prefix.
  - For example, ***echo "hello"*** => ***aws echo "hello"***

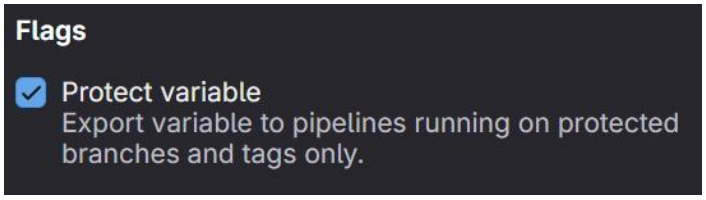
```yaml
deploy to s3:
  stage: deploy
  image:
    name: amazon/aws-cli:2.27.57
    entrypoint: [""]
  script:
    - aws --version
    - echo "Hello S3" > test.txt
    - aws s3 cp test.txt s3://alok-28032002/test.txt
```

- Inside the S3 bucket, click on **upload**. You'll get the destination i.e. **s3://<etc etc>**



- Its better to keep those bucket and all things in a variable, but not inside the **yml** file.
  - **Settings >> CI/CD >> Variables**
    - You can create your variables here.

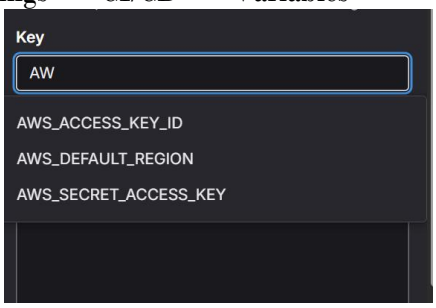


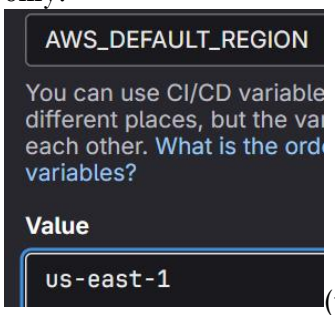


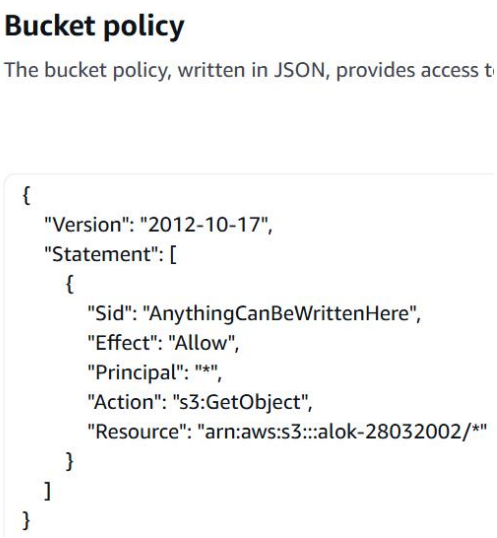
      - If this is enabled, then the variable will only be available for the protected branches.

```yaml
deploy to s3:
  stage: deploy
  image:
    name: amazon/aws-cli:2.27.57
    entrypoint: [""]
  script:
    - aws --version
    - echo "Hello S3" > test.txt
    - aws s3 cp test.txt s3://$AWS_S3_BUCKET/test.txt
```

  - But, still the pipeline will be failed due to the credentials.
- Create one IAM user (without password), add **S3 full access** permission to that.
- Now open that user and create access key for **CLI** and save the **access key id** & **secret access key**.

- Now, we'll add those things (access key id & secret access key) in the variables.
  - Settings >> CI/CD >> Variables
    - 

    - There will be some default variables having these kind of names, you just have to assign the values (access key id, secret access key) to this only.
    - 

(you have to set the region as well)
  - Now, the pipeline will succeed as the authentication will be done now.
  - ```
deploy to s3:
    stage: deploy
    image:
        name: amazon/aws-cli:2.27.57
        entrypoint: [""]
    script:
        - aws --version
        - aws s3 sync build s3://$AWS_S3_BUCKET --delete
        # --delete => if some files are there in dest but not in source
        # then, they'll be deleted from the destination.
```
- 

  - Do this after **enabling** static **web hosting** to give public access. Also **disable** that "block public access" thing.
- There is one issue in our pipeline, everytime when someone changes and commit (in any branch), the website will be deployed in **s3**.
- But it should not be the case, only after the merge, it should happen.
```

```
deploy to s3:
    stage: deploy
    image:
        name: amazon/aws-cli:2.27.57
        entrypoint: [""]
    rules:
        - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
    script:
        - aws --version
        - aws s3 sync build s3://$AWS_S3_BUCKET --delete
        # --delete  => if some files are there in dest but not in source
        # then, they'll be deleted from the destination.
```

- Now, it'll work fine. Only after the merge, the commit branch will be same as the default branch (i.e. **main** in this case). and after that the deployment will happen.

➢ We should add one **staging** stage before the deployment. If the staging passes, then only deployment will happen

- Build -> Test (CI: Continuous Integration)
- Staging -> Production (CD: Continuous Deployment)
- Here, **staging** means also deployment, but to check if the deployment and testing are successful or not, so that if something happened, the actual **production** will not be effected.

```
stages:
    - build
    - test
    - deploy staging
    - post staging
    - deploy production
    - post production

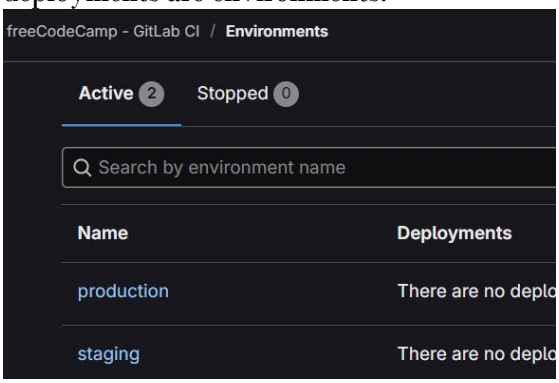
variables:
    APP_BASE_URL_STAGING: http://alok-staging-28032002.s3-website-us-east-1.amazonaws.com
    APP_BASE_URL: http://alok-28032002.s3-website-us-east-1.amazonaws.com

build website:
    image: node:lts-alpine
    stage: build
    script:
        - yarn install
        - yarn lint
        - yarn test
        - yarn build
    artifacts:
        paths:
            - build

test website:
    image: node:lts-alpine
    stage: test
    script:
        - yarn global add serve
        - apk add curl
        - serve -s build &
        - sleep 10
        - curl http://localhost:3000 | grep 'React App'
```

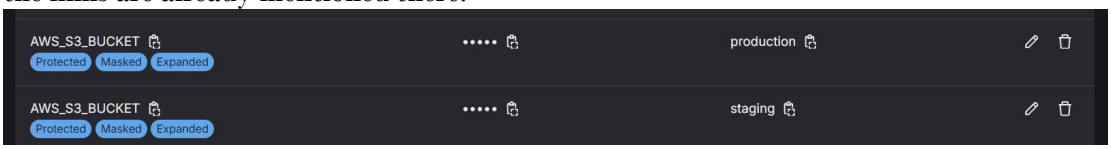
- Created the variables **AWS_S3_BUCKET** (production), **AWS_S3_BUCKET_STAGNIG** (staging)

```yaml
staging to s3:
    stage: deploy staging
    image:
        name: amazon/aws-cli:2.27.57
        entrypoint: [""]
    rules:
        - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
    script:
        - aws --version
        - aws s3 sync build s3://AWS_S3_BUCKET_STAGING --delete

staging tests:
    stage: post staging
    image: alpine/curl:8.14.1
    rules:
        - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
    script:
        - curl $APP_BASE_URL_STAGING
        | grep "React App"

deploy to s3:
    stage: deploy production
    image:
        name: amazon/aws-cli:2.27.57
        entrypoint: [""]
    rules:
        - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
    script:
        - aws --version
        - aws s3 sync build s3://$AWS_S3_BUCKET --delete

production tests:
    stage: post production
    image: alpine/curl:8.14.1
    rules:
        - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
    script:
        - curl $APP_BASE_URL
        | grep "React App"
```

- Now, we can see, so many duplications are there in the code.
  - We can create environments for this **(Operate >> Environments)**
  - Staging is an environment, Deployment is an environment.. means the deployments are environments.
  - 

(given the urls of the static website *[APP_BASE_URL, APP_BASE_URL_STAGING in my case]* inside that environments as the **external url** field)

- From here, you can directly click on the **open** button to open the environment as the links are already mentioned there.

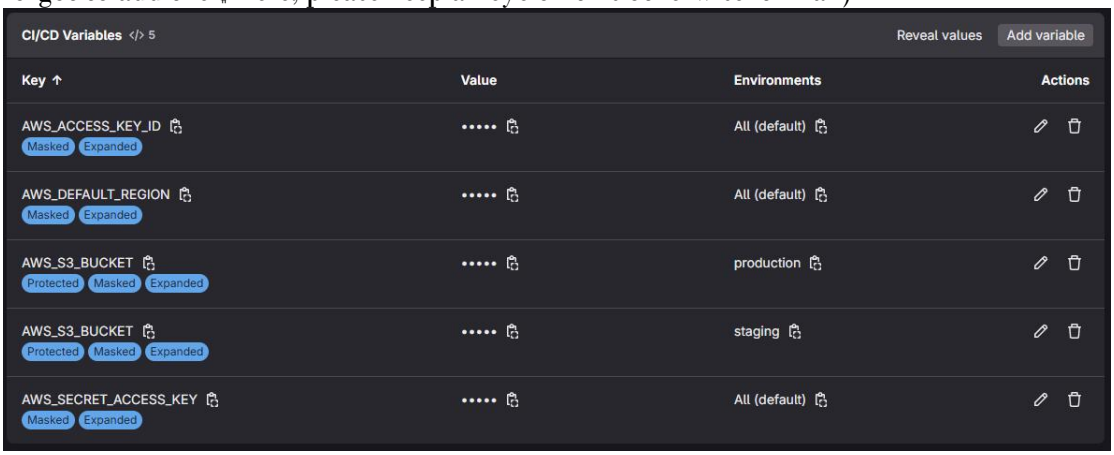


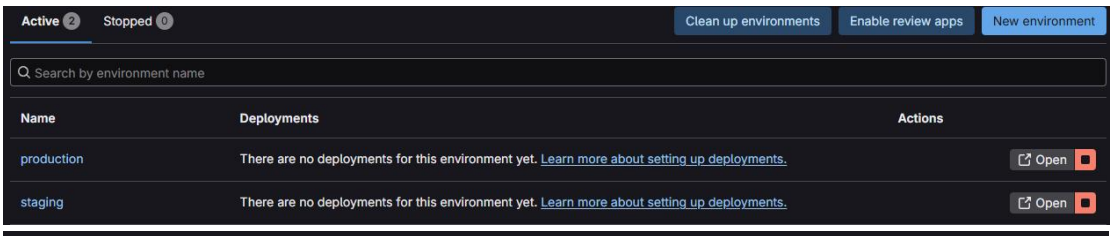
- Now, I edited the variables ***AWS_S3_BUCKET*** & ***AWS_S3_BUCKET_STAGING*** as ***AWS_S3_BUCKET*** and associated them to the specific environments.
- **So, now one variable name, 2 different values. Values are specific to the respective environments.**
- As the environments are linked with the URLs of the static hosted websites, we need to get them to mention inside our code to get rid of those 2 variables ***APP_BASE_URL*** & ***APP_BASE_URL_STAGING***.
- There is one pre-defined variable by gitlab, CI_ENVIRONMENT_URL, it'll extract the URL of the particular environment to which our current **job**(like staging, production) is linked to.

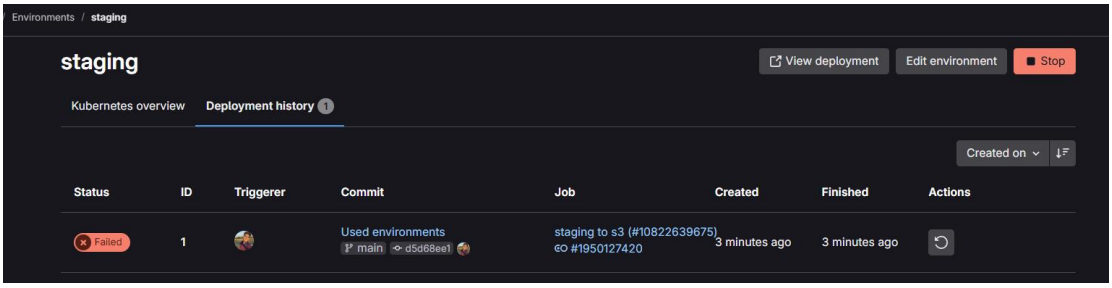
```
stages:
    - build
    - test
    - deploy staging
    - deploy production


build website:
  image: node:lts-alpine
  stage: build
  script:
    - yarn install
    - yarn lint
    - yarn test
    - yarn build
  artifacts:
    paths:
      - build

test website:
  image: node:lts-alpine
  stage: test
  script:
    - yarn global add serve
    - apk add curl
    - serve -s build &
    - sleep 10
    - curl http://localhost:3000 | grep 'React App'
```

```yaml
staging to s3:
    stage: deploy staging
    environment: staging
    image:
        name: amazon/aws-cli:2.27.57
        entrypoint: [""]
    rules:
        - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
    script:
        - aws --version
        - aws s3 sync build s3://AWS_S3_BUCKET --delete
        - curl $CI_ENVIRONMENT_URL
          | grep "React App"

deploy to s3:
    stage: deploy production
    environment: production
    image:
        name: amazon/aws-cli:2.27.57
        entrypoint: [""]
    rules:
        - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
    script:
        - aws --version
        - aws s3 sync build s3://$AWS_S3_BUCKET --delete
        - curl $CI_ENVIRONMENT_URL
          | grep "React App"
```

(I forgot to add the $ here, please keep an eye on this otherwise it'll fail)







(you can see the deployment history of a perticular environment as well)

```yaml
stages:
  - build
  - test
  - deploy staging
  - deploy production


build website:
  image: node:lts-alpine
  stage: build
  script:
    - yarn install
    - yarn lint
    - yarn test
    - yarn build
  artifacts:
    paths:
      - build

test website:
  image: node:lts-alpine
  stage: test
  script:
    - yarn global add serve
    - apk add curl
    - serve -s build &
    - sleep 10
    - curl http://localhost:3000 | grep 'React App'

.deploy:
  image:
    name: amazon/aws-cli:2.27.57
    entrypoint: [""]
  rules:
    - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
  script:
    - aws --version
    - aws s3 sync build s3://AWS_S3_BUCKET --delete
    - curl $CI_ENVIRONMENT_URL
      | grep "React App"

deploy to staging:
  stage: deploy staging
  environment: staging
  extends: .deploy

deploy to production:
  stage: deploy production
  environment: production
  extends: .deploy
```

- We can use **.** (dot) to create re-usable code blocks, like here I did.
- After that, you can use **extends** keyword to use that code-block inside any job.

- AWS Elastic Beanstalk:
  - Allow us to deploy one application in AWS cloud without us worrying about the actual virtual server that runs it.
  - You simply upload your code and Elastic Beanstalk automatically handles the deployment, from capacity provisioning, load balancing, and automatic scaling to web application health monitoring, with ongoing fully managed patch and security updates.
-