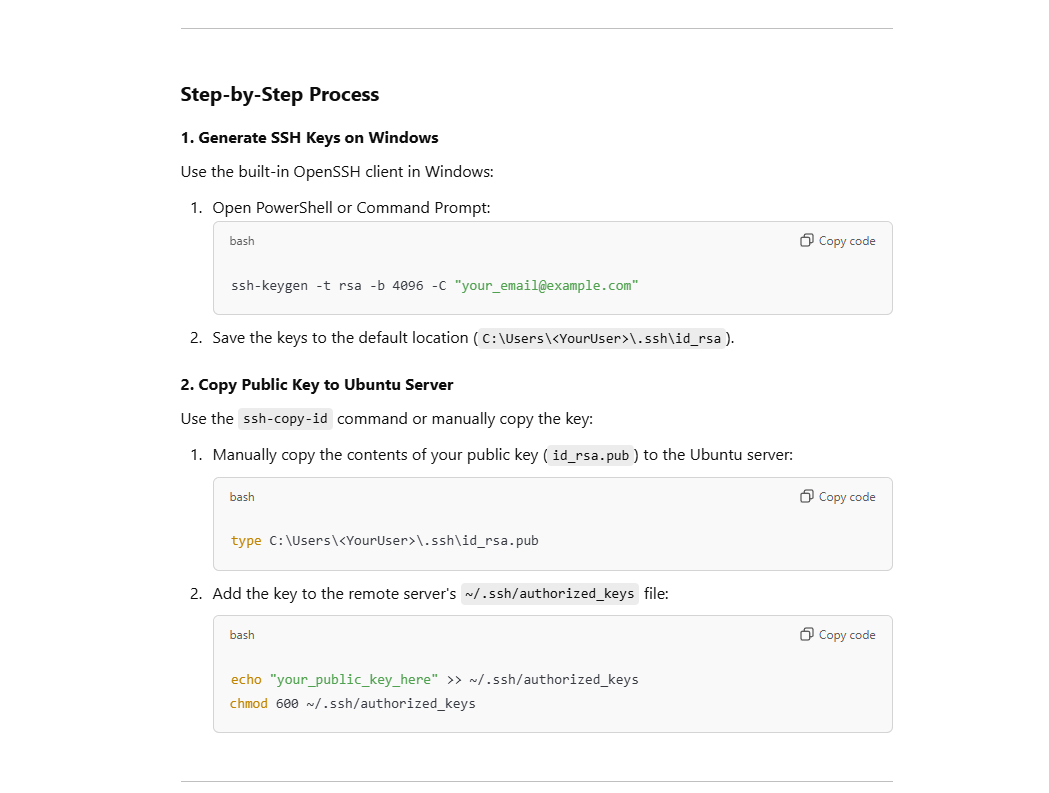
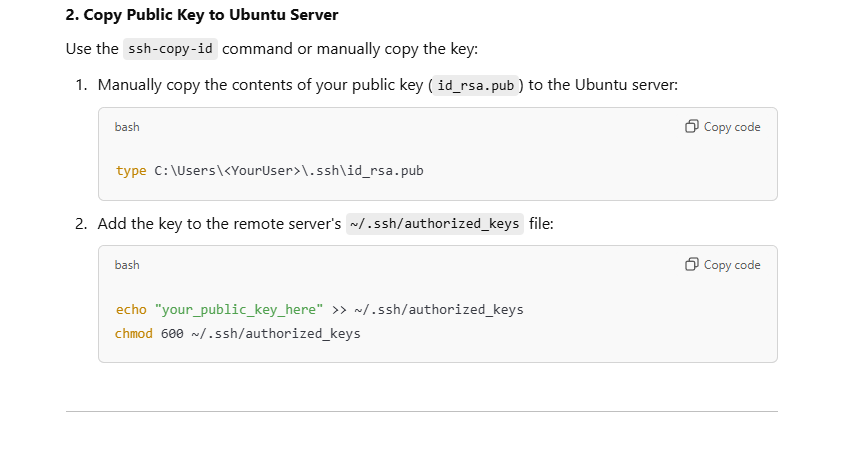
# How to upload code to github then server:



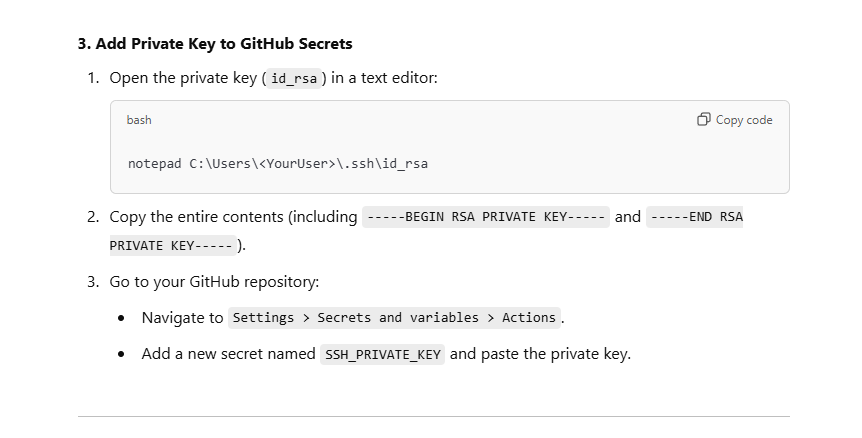
On your local windows machine

1. ssh-keygen -t rsa -b 4096 -C [your\_email@example.com](mailto:your_email@example.com)
2. search using program ‘everything’ for ‘.pub’ program

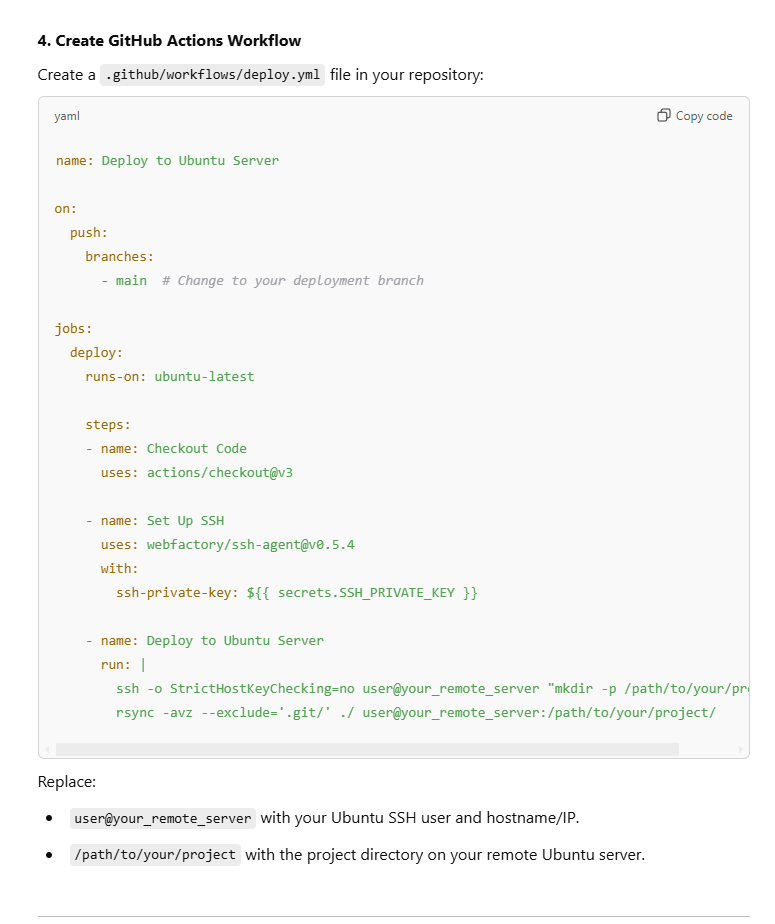
On ubuntu machine accept the windows ssh key to make connection between two on ssh



On GitHub, on your repository u freshly created



On your current project, create the following yml file before pushing code to repository



Now you must upload code to GitHub using git add ., git push

Now u have multiple choices, if the machine (ubuntu server on remote server) is protected by a firewall or proxy or VPN, you have two choices:

1. make a GitHub runner instance run locally on the end machine (ubuntu server located in Europe), this way, GitHub runner is talking directly from inside the machine as listener to events from global GitHub
   1. u can set the yml file to a place on the network on another machine currently on network with the ubuntu server existing, and u need to setup the configuration as in the figure above of ‘.yml’ file

name: Deploy to Ubuntu Server

on:

  push:

    branches:

      - main  # Change to your deployment branch

jobs:

  deploy:

    runs-on: self-hosted

    steps:

    - name: Checkout Code

      uses: actions/checkout@v3

    - name: Set Up SSH

      uses: webfactory/ssh-agent@v0.5.4

      with:

        ssh-private-key: ${{ secrets.xxxx }}

    - name: Deploy to Ubuntu Server

      run: |

        ssh -o StrictHostKeyChecking=no test@192.168.x.x "mkdir -p /home/test/test\_react"

        rsync -avz --exclude='.git/' ./ test@192.168.x.x:/home/test/test\_react

* 1. u can set the GitHub runner in the same server as the ubuntu machine holding the code, this way you need to modify the ‘.yml’ file, U DO NOT NEED SSH HERE, it is direct, you can simplify the GitHub Actions workflow since no SSH or rsync is required. Instead, you can directly execute commands on the machine itself.

name: Deploy to Local Server

on:

  push:

    branches:

      - main  # Replace with the branch you want to deploy from

jobs:

  deploy:

    runs-on: self-hosted  # Use the self-hosted runner on the same machine

    steps:

    - name: Checkout Code

      uses: actions/checkout@v3

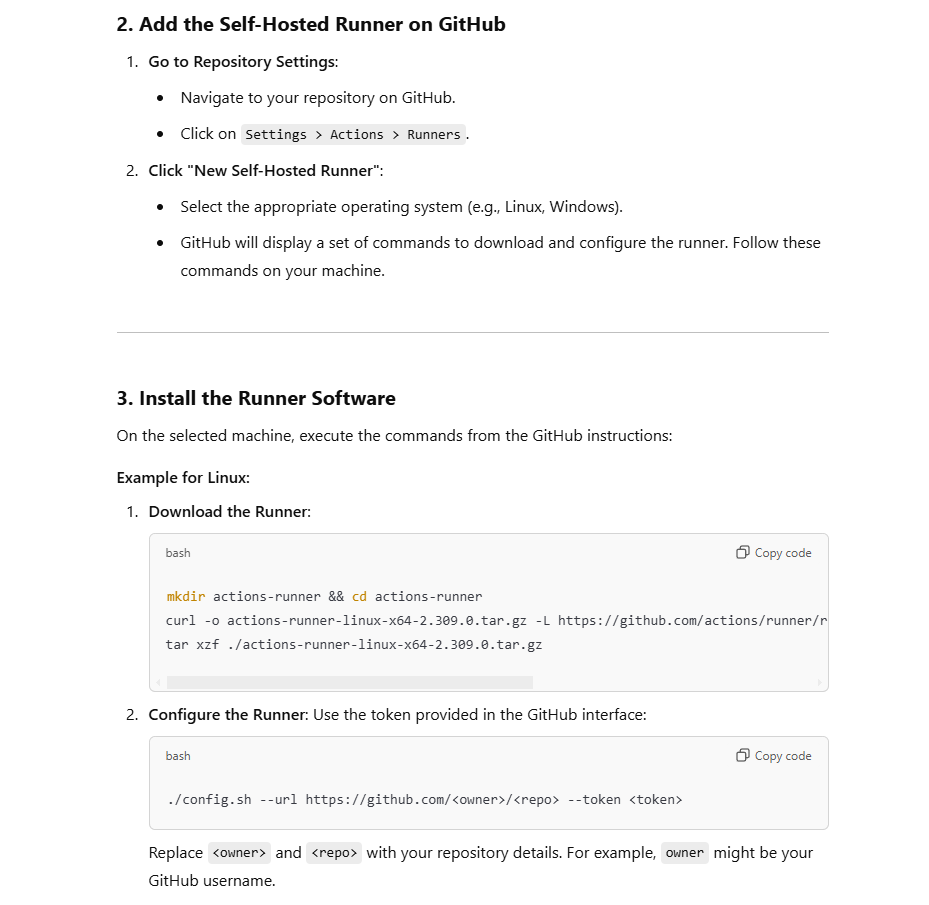
    - name: Deploy Application

      run: |

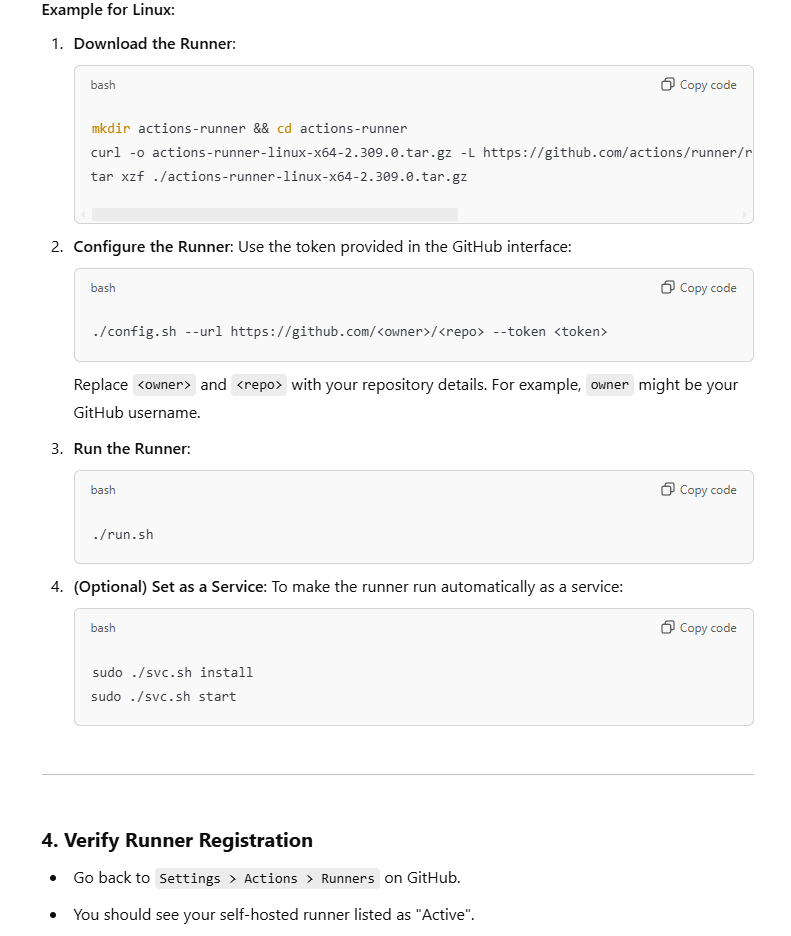
        mkdir -p /home/test/test\_react

        rsync -avz --exclude='.git/' ./ /home/test/test\_react/

U should also register a GitHub runner in github account



https://github.com/repository\_owner/repository\_name/settings/actions/runners/new?arch=x64&os=linux



Still need to make the runner as a service , using the optional command in the picture above, after navigating to the directory containing the runner directory that is installed on ubuntu

NOW, it will run on machine boot, so it is always running

1. The other choice is to use public ip, provided by the admin of company and allow a specific port to access ssh for this specific machine, so its kinda gateway,in ssh, u need to include the -P command or other way , search for it

# How to perform docker compose delete container and image and create new one when a release is done

I found another way, which is to use the tag clause, which is used to create a release in GitHub, so if u use tag in your code and then push the tag to GitHub, it will add a release to the releases page with the version u defined as name, for example xxx\_1.3

1-must make docker non sudo

sudo groupadd docker

sudo usermod -aG docker $USER

newgrp docker

sudo systemctl restart docker

restart ubuntu remote machine

this is needed to not encounter sock error

sudo chmod 666 /var/run/docker.sock

name: Deploy on Tag with Image Retention (Self-Hosted Runner)

on:

  push:

    tags:

      - '[0-9]\*.[0-9]\*' # Matches tags like 0.1, 1.2, etc.

jobs:

  deploy:

    runs-on: self-hosted

    steps:

      - name: Checkout code

        uses: actions/checkout@v3

      - name: Deploy application

        run: |

          IMAGE\_NAME=test-react-img

          CONTAINER\_NAME=test-react-container

          # Stop and remove the old container

          docker ps -a -q --filter "name=$CONTAINER\_NAME" | xargs -r docker rm -f

          # Remove older images beyond the retention limit (keep 3 older + current)

          docker images $IMAGE\_NAME --format "{{.Repository}}:{{.Tag}} {{.CreatedAt}}" | \

          sort -rk 2 | \

          awk 'NR>3 {print $1}' | xargs -r docker rmi -f

          # Build the new image

          docker build -t $IMAGE\_NAME:${GITHUB\_REF\_NAME} .

          # Start the new container

          docker run -d --name $CONTAINER\_NAME -p 82:82 $IMAGE\_NAME:${GITHUB\_REF\_NAME}

        env:

          GITHUB\_REF\_NAME: ${{ github.ref\_name }}

Docker file is this :   
# Use an official Node.js runtime as a parent image

FROM node:14

# Set the working directory

WORKDIR /usr/src/app

# Copy the current directory contents into the container at /usr/src/app

COPY . .

# Install any needed packages

RUN npm install

# Make port 80 available to the world outside this container

EXPOSE 82

# Define build argument

# -----------------------------------------------

# Define environment variable

ENV NAME World

# Run app.js when the container launches

CMD ["node", "app.js"]

Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1. 0.1). A tag is like a branch that doesn't change. Unlike branches, tags, after being created, have no further history of commits.

Git add . (add changes u want to last version)

Git commit -m “xzcxzc”

Git push ( to upload the code to the server with latest updates by the first github action which deploys code to server)

-------------wait 30 seconds for deploying code github action----(max)

git tag 1.5

git push origin 1.5

----------------- initiating creating docker service github action---------------

In this code, it will run the docker file in the base directory when a release is made(u made a git tag branch that is a release)and keep 2 old images from the tag of the base image

When using docker compose :

name: Deploy Docker on git release Tag with Image Retention (Self-Hosted Runner)

on:

  push:

    tags:

      - '[0-9]\*.[0-9]\*' # Matches tags like 0.1, 1.2, etc.

jobs:

  deploy:

    runs-on: self-hosted

    steps:

      - name: Checkout code

        uses: actions/checkout@v3

      - name: Deploy application

        run: |

          IMAGE\_NAME=react\_test

          CONTAINER\_NAME=test\_container

          # Stop and remove the old container

          docker ps -a -q --filter "name=$CONTAINER\_NAME" | xargs -r docker rm -f

          # Remove older images beyond the retention limit (keep 3 older + current)

          docker images $IMAGE\_NAME --format "{{.Repository}}:{{.Tag}} {{.CreatedAt}}" | \

          sort -rk 2 | \

          awk 'NR>3 {print $1}' | xargs -r docker rmi -f

          export TAG=${GITHUB\_REF\_NAME}

          TAG=${GITHUB\_REF\_NAME} docker compose build

          TAG=${GITHUB\_REF\_NAME} docker compose up -d --force-recreate

        env:

          GITHUB\_REF\_NAME: ${{ github.ref\_name }}

This will get the GitHub ref name from pass git tag clause and pass it to docker compose

Will build the image name and container name by the tag of release

Here we are sending the tag to the docker compose

          export TAG=${GITHUB\_REF\_NAME}

          TAG=${GITHUB\_REF\_NAME} docker compose build

          TAG=${GITHUB\_REF\_NAME} docker compose up -d --force-recreate

version: '3'

services:

  react-ui:

    container\_name: react-ui

    build:

      context: .

      args:

        IMAGE\_TAG: ${TAG} # Dynamically passed from GitHub Actions

    image: react\_test:${TAG} # Set the dynamic tag for the image

    ports:

      - "3000:3000"

    restart: unless-stopped

in docker compose , we are passing the tag to the docker file to add to image name

      args:

        IMAGE\_TAG: ${TAG} # Dynamically passed from GitHub Actions

# Use an official Node.js runtime as a parent image

FROM node:14

# Set the working directory

WORKDIR /usr/src/app

# Copy the current directory contents into the container at /usr/src/app

COPY . .

# Install any needed packages

RUN npm install

# Make port 80 available to the world outside this container

EXPOSE 82

# Define build argument

# in case u are using docker-compose, u add these 2 , else not needed

# -----------------------------------------------

ARG IMAGE\_TAG

ENV IMAGE\_TAG=${IMAGE\_TAG}

# -----------------------------------------------

# Define environment variable

ENV NAME World

# Run app.js when the container launches

CMD ["node", "app.js"]

in these two lines here of the docker file , we are receiving the IMAGE\_TAG from docker compose which is receiving it from github action which is receiving it from git tag clause version

ARG IMAGE\_TAG

ENV IMAGE\_TAG=${IMAGE\_TAG}