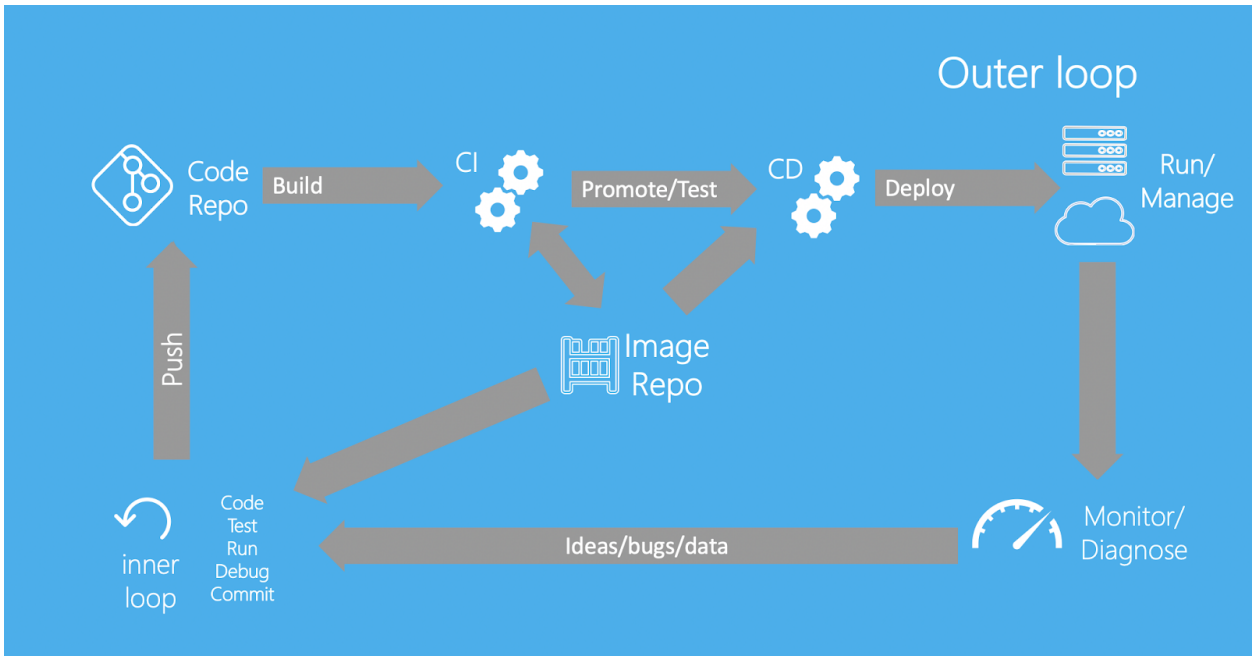
ECGR 4090/5090 Cloud Native Application Architecture
Lab 12 : CI/CD with GitHub Action

Continuous Integration (CI) is a practice of frequently committing code to a shared repository, as well as building and testing it. CI advantages include detecting errors sooner, reducing the amount of code that needs to be debugged, and managing changes made by multiple members of a software development team. The alternative to CI was the inefficient “big bang” approach to code integration where all components are integrated and tested at once to build a complicated system.

Building and testing a code requires a server. In CI, the server monitors change events in the repository either by getting notifications of change events from the repository (push) or polling the repository for change events. Examples of change events are code pushing, and pull requests. The server could be run locally or in the cloud.

Continuous Deployment (CD) is a software release strategy where the code that passes the test phase, is automatically released to production (that is visible to users). In the cloud native world, this would involve creating a Docker container for the new version of the application, uploading it to a container registry, and performing a rolling update of the application with Kubernetes.

The overall flow looks like this -



(From docker.com)

Jenkins is a CI/CD tool that has been used for the last 10 years. However, Jenkins is pre-cloud native, and is a complex software that often requires a dedicated operational team for operation. Other tools in this space include Travis CI, and Circle CI. However, recently popular code

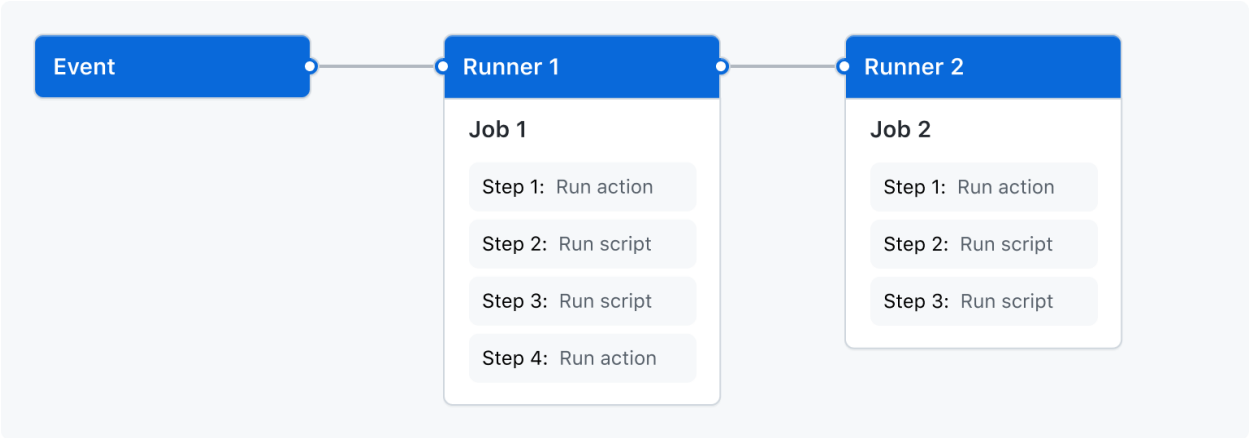
repositories GitHub and GitLab have introduced a CI/CD workflow integrated onto their platforms. A big win with the repo hosted CI/CD tools from the application developers perspective, is that the hosting and maintenance is done by the repo vendor.

Github Actions

In this lab, we will explore Github's CI/CD workflow known as Github Actions to get hands-on experience with CI/CD workflow.
For more information, refer to the Github Actions docs
https://docs.github.com/en/actions

The following figure from GitHub shows the basic working of Actions

An event (for example push code into repo) automatically triggers the *workflow*, which contains a *job*. The job then uses *steps* to control the order in which *actions* are run. These actions are the commands that automate your software building, and testing



Components of Github actions

Workflows
A workflow is a configurable automated process that will run one or more jobs. Workflows are defined by a YAML file checked in to your repository and will run when triggered by an event in your repository, or they can be triggered manually, or at a defined schedule.

You can have multiple workflows in a repository, each of which can perform a different set of steps. For example, you can have one workflow to build and test pull requests, another workflow to deploy your application every time a release is created, and still another workflow that adds a label every time someone opens a new issue.

Events
An event is a specific activity in a repository that triggers a workflow run. For example, activity can originate from GitHub when someone creates a pull request, opens an issue, or pushes a commit to a repository. You can also trigger a workflow run on a schedule, by posting to a REST API, or manually.

Jobs
A job is a set of steps that execute on the same runner (run time server). By default, a workflow with multiple jobs will run those jobs in parallel. You can also configure a workflow to run jobs sequentially to capture dependencies between jobs.

Actions
An action is a custom application for the GitHub Actions platform that performs a complex but frequently repeated task. Use an action to help reduce the amount of repetitive code that you write in your workflow files. An action can pull your git repository from GitHub, set up the correct toolchain for your build environment, or set up the authentication to your cloud provider.

You can write your own actions, or you can find actions to use in your workflows in the GitHub Marketplace.

Runners
A runner is a server that has the GitHub Actions runner application installed. You can use a runner hosted by GitHub, or you can host your own. A runner listens for available jobs, runs one job at a time, and reports the progress, logs, and results back to GitHub. GitHub-hosted runners are based on Ubuntu Linux, Microsoft Windows, and macOS, and each job in a workflow runs in a fresh virtual environment. As of March 2021, GitHub Free account gets 500 MB of storage, and a monthly 2,000 minutes of runner time.

GitHub Actions uses YAML syntax to define the events, jobs, and steps. These YAML files are stored in your code repository, in a directory called .github/workflows

Example workflow file for automatically running a Go test

```
name: hello-world
on: [push, pull_request]
name: Test
jobs:
```

```
  test:
    strategy:
      matrix:
        go-version: [1.15.x, 1.16.x]
        os: [ubuntu-latest]
    runs-on: ${{ matrix.os }}
    steps:
    - name: Install Go
      uses: actions/setup-go@v2
      with:
        go-version: ${{ matrix.go-version }}
    - name: Checkout code
      uses: actions/checkout@v2
    - name: Test
      run: go test ./<your package name>
```

Explanations of YAML

name - optional
on - Specify the event that triggers the workflow
jobs - defines the job
test - name of job
strategy - specifies how the job will be run. Here we plan to use multiple Go versions, and potentially multiple OS versions.
matrix: allows you to create multiple jobs by performing variable substitution in a single job definition
steps: groups together all the steps that run in the *test* job.
uses (setup-go@v2): This action installs the Go on the runner
uses(checkout@v2): tells the job to retrieve v2 of the community action named actions/checkout@v2. This is an action that checks out your repository and downloads it to the runner, allowing you to run actions against your code
run: run the go test command

Continuous Integration

Create a new Github repo (name it *github-action*). Let's create a Go REST microservice that returns "Hello World". Note the use of the *httptest* package to test the webserver.
Now create a local folder under lab12 with the following Go code (server.go and server_test.go) in a directory */microservice*

```
// Hello world microservice - server.go
package microservice
```

```
import (
    "fmt"
    "net/http"
)

func NewServer(host string, port string) *http.Server {
    addr := fmt.Sprintf("%s:%s", host, port)

    mux := http.NewServeMux()
    mux.HandleFunc("/", handler)

    return &http.Server{
        Addr:    addr,
        Handler: mux,
    }
}

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World!\n")
}

// Test - server_test.go
package microservice

import (
    "bytes"
    "net/http"
    "net/http/httptest"
    "testing"
)

var (
    port = "8000"
)

// Unit Tests
```

```
func TestHandler(t *testing.T) {
    expected := []byte("Hello World!\n")

    req, err := http.NewRequest("GET", buildUrl("/"), nil)
    if err != nil {
        t.Fatal(err)
    }

    res := httptest.NewRecorder()

    handler(res, req)

    if res.Code != http.StatusOK {
        t.Errorf("Response code was %v; want 200", res.Code)
    }

    if bytes.Compare(expected, res.Body.Bytes()) != 0 {
        t.Errorf("Response body was '%v'; want '%v'", expected,
res.Body)
    }
}

// Helper functions

func buildUrl(path string) string {
    return urlFor("http", port, path)
}

func urlFor(scheme string, serverPort string, path string) string {
    return scheme + "://localhost:" + serverPort + path
```

In the microservice directory run
$ go test
Test should pass.

Now create a main.go file outside the package directory to invoke the server.
Note: adjust the microservice path according to your go mod

```
// Launch microservice server- main.go
package main

import (
    "github-action/microservice"
    "log"
)

func main() {
    s := microservice.NewServer("", "8000")
    log.Fatal(s.ListenAndServe())
}
```

Create a .github/workflows directory with the YAML file listed at the beginning of the lab (call it main.yaml). Your directory structure looks like this

github-action
- *microservice*
- *.github*
- *main.go*

Your .github directory looks like this

.github
- *workflows*
  - *main.yaml*

Commit and push the code to Github.
Note: Github requires you to use a Personal Access Token to push code from CLI. Be sure to enable the Workflow scope of your access token.
https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token

On your Github page of the repo, click on the Actions tab. Click on the workflow to examine the result of the CI actions that were triggered by the code push. You should be able to see the test running. Eventually the test should pass. Make a change to the code (change the greeting message), and push the code to the repo. Check to see if the tests were run on Github.

Note1: The CI action is automatically triggered by the code push event.
Note2: There might be a delay in the pipeline running on Github if the servers are busy. You might want to try during other times of the day if that happens.

In the Cloud Native workflow, developers push code to the repo automatically triggering tests. Note that the test server (runner) could run in your own cloud environment rather than on Github. Also, you could specify a more complex build process using a build tool such as the *make* utility. Languages may come with their own build tools (for example, Maven for Java)

Continuous Deployment

Our first step is to containerize the application with Docker and test locally. Copy and paste the following into a Dockerfile

```
FROM golang:1.15-alpine AS build

WORKDIR /src
COPY go.mod .
RUN go mod download
COPY main.go .
ADD microservice ./microservice
RUN CGO_ENABLED=0 go build -o /bin/helloserver

FROM scratch
COPY --from=build /bin/helloserver /bin/helloserver
ENTRYPOINT ["/bin/helloserver"]
```

Build the container

```
$ sudo docker image build -t helloserver .
```

Run the container

```
sudo docker container run -p 8000:8000 helloserver
```

Open another terminal and curl

```
$ curl http://localhost:8000
```

You should see the “Hello World!” message

We are now ready to push the Docker image to DockerHub using Github action. DockerHub is used to host Docker images. The goal is - whenever code is pushed to the GitHub repository, it should trigger a new Docker image build, followed by pushing the image to DockerHub. The image can then be used in a cluster.

Head to hub.docker.com and create a free account.

Head back to GitHub. Click on repo (the one you are using for this lab) settings (gear icon), and click on secrets (on the left). Add the the following

DOCKER_USERNAME - your DockerHub username
DOCKER_PASSWORD - your DockerHub password

Copy and paste (append) the following to *.github/workflow/main.yaml*. Note that *docker* is another job, and should be on the same column as *test*. We are specifying a dependency that the *docker* job needs the *test* job.
Note: Replace the tag with your Docker username

```
  docker:
    needs: test
    runs-on: ubuntu-latest
    steps:
    - name: Login to DockerHub
      uses: docker/login-action@v1
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}
    - name: Build and push
      id: docker_build
      uses: docker/build-push-action@v2
      with:
        push: true
        tags: arunravindran/helloserver:latest
    - name: Image digest
      run: echo ${{ steps.docker_build.outputs.digest }}
```

Commit and push the the yaml to the Github repo. This should automatically trigger the test, Docker build, and Docker push actions. Check the actions tab of your repo on Github.

If the execution was successful, your Dockerhub should have a new *helloserver* repo.

Next step
A Kubernetes (or similar) cluster can now pull this image from Dockerhub and perform a rolling update.

Here's how to do this for the AWS Elastic Container Service. Instead of DockerHub AWS Elastic Container Registry is used for the Docker images.
https://docs.github.com/en/actions/guides/deploying-to-amazon-elastic-container-service

To do -

Change the greeting in microservice/server.go (say to "Howdy World!\n"). Leave the test unchanged.

Next trigger the CI/CD using Github actions, and see if the tests were run. Also check DockerHub to see if a new image was pushed to DockerHub.