Week 13 - Continuous Integration, Instrumentation & Metrics

(Optional)

With your containerized applications, your team is now more easily able to share the individual Flask applications you have made with one another and have made a major step towards deploying your work. You likely noticed, however that the work you did last week was heavily manual and prone to simple terminal errors or mistyping that could force you to start over again. This week, we'll improve upon that process by setting up both a Continuous Integration (CI) tool (GitHub Actions and an automated deployment tool Docker Compose. These tools require configuration files, `.yaml` files to be exact, and automate many of the tedious terminal commands required to build and deploy applications. Once you've automated those processes, we'll extend the capabilites you've built with open source monitoring tools (Prometheus and Grafana).

Learning Objectives

LO1. Automate Container Build Processes using GitHub Actions

LO2. Select relevant metrics to collect from your predictive services

LO3. Configure Instrumentation (Prometheus) and Metrics visualization (Grafana) tools

LO4. Automate multi-container application deployments with Docker compose

Tasks

Note: This week's tasks will require you to use YAML to define multiple files. This language has a very simple syntax requirements, however if you have not used it before or are looking for a quick referesher you should explore Learn Yaml in Y minutes.

1. `[Grp]` Define an initial GitHub Actions workflow. We will go through instructions in detail to build a workflow for a single Flask application as an example and then you can extend this to build similar workflows for your other applications.

    i. Create the directory `.github/workflows` in your capstone base directory. Within that directory create a `.yaml` file for each container image you will need to build. Follow the remaining steps herein for each container image workflow file.

    ii. Start by defining a name for your workflow. This should be something simple and specific that describes what the workflow does. In this case, it will be building and publishing a container image so we could use `Build and Publish nlp_pred`.

    iii. Select the trigger for this workflow. GitHub Actions provides many events that trigger workflows all based around actions your team would execute in GitHub while working on your project. In this case, we will use the action of a push and apply two filters to that action. The first is a branch filter, it should only run when changes occur on the `dev` branch. The second is a paths filter, it should only run when changes occur in the `nlp_pred` directory to a python file, the `requirements.txt` file and the `Dockerfile`. This is because those files are the only ones that should be included in your container images.

    iv. Specify permissions. Because we are going to build and publish a new container, we will need to specifically grant `write` permissions to the `packages` scope and specify `read` permissions to the `contents` scope. We will do this globally for the entire workflow, but you can also do this more specifically for individual jobs.

    v. Define jobs. A GitHub Actions workflow is composed of jobs that in turn have defined steps. A step is analogous to an individual command executed in last week's process of containerizing your applications. Today this will only automate the building and publication of container images, but on a larger project you will also include steps for project unit tests and will automate other repetitive processes (like deployment, too).

    - The first job you create should be the build job. Specify that it `runs-on` os `ubuntu-latest`. It should have steps to do the following things.
    - Checkout the git repository using the `actions/checkout@v3` reusable action.
    - Login to the GitHub Container Registry using the `docker/login-action@v2` action with parameters for the registry, `ghcr.io`, your username and the `GITHUB_TOKEN` secret as the password.
    - Create Docker metadata using the `docker/metadata-action@v4` with an id of `meta` and specifying the images option specifying your container image name from last week (the one that starts with `ghcr.io/AFC-AI2C/...`). Be sure to also specify the tags in `type=raw,value=latest` for our ease of use. Future teams you are a part of may use other rules for container tagging.

  - Build and push the container image using the `docker/build-push-action@v4` with the options of push set to true, the context specified as the directory containing your application and Dockerfile, the tags defined as `${{ steps.meta.outputs.tags }}`$.

  vi. Once you have created your workflow, stage, commit, and push your changes to GitHub.

2. `[Grp]` Create Workflow files for each application. Make small changes to your app files in order to trigger the builds. Verify each build succeeds and a new image was created and pushed to your repository's packages. Depending on previous decisions your team made, you may not have your computer vision model stored in your GitHub repository and therefore would not be able to build your container image in a workflow. What are your options for continuous integration in this case? Add a section in your overall project `README.md` file discussing tradeoffs with different solutions and specifying the solution your team used. How might that translate to other projects in the future? (Note: When using the wildcard syntax in your paths filter for the trigger, you may need to use a slightly different filter for the tsa application than the others so that it matches recursively instead of only on python files in that directory.)

3. `[Grp]` Now that you have automated the process of building your container images, let's do the same for the process of deploying your multi-container application. To do this for local development you will use docker compose (docs here). Start by creating a new file in your project's root directory named `compose.yaml`.

4. `[Grp]` At the root of every docker compose file is a `services` declaration. Below this, you will need to declare each "service" in your multi-container application. In your case, this includes your database, tsa application, cv_pred, marketing_pred, and nlp_pred applications. We'll break these down below with some tips to be aware of.

   i. Database. Be sure to specify the `container-name` so that you can make a network connection to this container once your application is running. In order to pass the `POSTGRES_USER` and `POSTGRES_PASSWORD` environment variables, create a new directory in your project named `db` and include within it a `README.md` file and a `.env` file. Your `.env` file should specify the values for the two required environment variables above and your `README.md` file should specify key pieces of information like the database type and version you are using in this project. Be sure to reference your `.env` file in your `compose.yaml` file properly so the database container can reference these values when it is run.

   ii. For your prediction containers (`cv_pred, nlp_pred, and marketing_pred`), you will need to specify the full image name (including `ghcr.io/AFC-AI2C/...`) that you specified last week for each container. You should also specify a `container_name` for each of these services to enable your `tsa` container to communicate with these services.

   iii. For your `tsa` application, specify the full image name as you did for the prediction containers, map one of your local ports to port 8000 for your debugging access. Specify the `env_file` to reference your tsa `.env` file you referenced last week when running your container. Finally, last week your final question was regarding which containers were dependent on other containers. Your `tsa` application is dependent on your `capstone-db` container, specify a dependency as such in your `compose.yaml` file's service declaration for the `tsa` service. It should use the verbose syntax and specifically await the `service_healthy` state in your database. To do so, you will need to specify a healthcheck in your db service declaration. Do so using the following command as the test, `["CMD", "pg_isready", "-U", "postgres", "-d", "postgres"]`. Select the appropriate healthcheck parameters when you do so.

   The definitions above are enough to run your multi-container application now with a single command.

   ```
   docker compose up -d
   ```

   (Note: the `-d` flag runs these in the background. You can access the logs in the terminal in the same manner you did previously in week 12.)

   Now is a good time to stage, commit, and push your changes to GitHub if you haven't done so already.

5. `[Ind]` Once your team has properly configured docker compose, each team member run your team's application locally and test out its functionality. If you run into an issue with your containers communicating with one another, consider how you might use the `container-name` values you set in the docker compose file to enable local addressing between containers.

6. `[Grp]` Update each application's `Dockerfile` and your `compose.yaml` to enable faster development. So far you've created very simple container images, but you may start to wonder how you can easily update the code within each application. After all, the container's have the code copied into them at build time, so based on the proces you've used so far you would need to rebuild your container each time you wanted to test a change. Instead you will update each application's `Dockerfile` slightly and in your `compose.yaml` file you will specify a bind mount for each application's code. This is a really powerful capability where you will link a directory from your host machine to the container's file system.

   i. To enable this capability we will make one change to each `Dockerfile` to specify the ”working directory”. Add the line `WORKDIR /code` after you add labels in each `Dockerfile` in your application. This command sets the specified directory as the ”working directory” similar to using ”cd” locally. If that directory does not exist, it creates it for you. Your application code is now copied into this directory instead of into the root directory of the container’s file system.
   ii. In each service declaration in your `compose.yaml` file you will also need to add one additional configuration. Specify `volumes` with a record mounting your local directory for that application to the `code` directory in the container. For the computer vision application this may look like the configuration below:

```
volumes:
  - ./cv_pred:/code
```

   iii. After making these changes, rebuild your container images (either by pushing to GitHub and using your GitHub Actions workflows or manually and then pushing to the GitHub Container Registry) and run `docker compose up -d` again. This will restart the necessary containers (those with changes to the image). Test this worked by making changes to the application code for your applications and seeing the changes reflected in your application or in the logs. If you haven’t already, now is a good time to stage, commit, and push your changes to GitHub.

7. `[Grp]` Setup Prometheus.
   i. Pull the container image, `prom/prometheus`: `docker pull prom/prometheus`
   ii. Create a `prometheus` directory in your root project directory. Create two files, `README.md` and `prometheus.yaml`.
   iii. Add a service declaration to your `compose.yaml` file. Be sure to specify the bind mount for the `prometheus.yaml` file to the `/etc/prometheus/prometheus.yml` file on your container and map port 9090 to 9090 on your local machine so you can interact with your metrics endpoint.
   iv. Edit the configuration file to your desired configuration. At a minimum, specify global parameters of the `scrape_interval` and set `external_label` as `team: ’23_capstone_team_<your team number>’`. For an initial test, configure prometheus to monitor itself (you can find an initial guide for this in the prometheus documentation).
   v. Document your configuration choices in the `README.md` file within the `prometheus` directory. Describe why you changed the configurations you did.
   vi. Run `docker compose up -d` and navigate to http://localhost:9090/ to explore the metrics available to you as Prometheus monitors itself. Using the metrics explorer, explore Prometheus’ default metrics offering.

8. `[Ind]` Instrument your prediction services using the Prometheus client library for Python. We instrument our applications to better understand how they are running in production. There are a number of things we may be interested in. Some examples include prediction time (how long does it take to complete the prediction?), counters of which class is predicted, a report of the sentiment classes predicted by your model (how many were positive, neutral, and negative as the primary prediction), and many many more. The way you should come up with the metrics to collect is to consider problems that may occur for your application’s predictive services in production. These could be things like memory or CPU usage or model characteristics like predicted values to monitor for model drift. Once you have identified the questions you want to answer (try to come up with at least two per application) follow the steps below to instrument your application.
   i. Select the type of metric you will use (two per application). In the individual application’s `README.md` file include a description of the metrics you chose to use and why.
   ii. `pip install prometheus-client` and update your application’s `requirements.txt` file accordingly.
   iii. Import necessary packages to your `app.py` file and configure your two metrics in your `app.py` file. To expose your metrics you can either adjust your Flask application’s method to run so your Flask application exposes metrics (we recommend upgrading to use `uWSGI` or `gUnicorn`) or you can update your project’s `Dockerfile` to expose an additional port, `8000` and you can use the `prometheus-client`’s `start_http_server` method.
   iv. Update the `prometheus.yaml` file with a scrape config for your application. You can remove the scrape config for Prometheus itself at this point to make it easier to find your configured metrics.
   v. Once you have finished these configurations and rebuilt your container images, rerun `docker compose up -d` and explore the prometheus metrics UI to identify your metrics. Explore the different query types (like rates) and see how you can interact with your application’s metrics. (Hint: You may need to provide your API with some interaction in order to see metrics appear)

9. `[Grp]` In order to better visualize the metrics coming from Prometheus, setup Grafana to visualize the metrics for you.

i. Start by pulling the container image, `grafana/grafana`.

ii. Create a `grafana` directory in your root project directory. Create a `README.md` file within this directory.

iii. Add a service declaration to your `compose.yaml` file. Specify the dependencies for Grafana, expose its standard port, `3000`, add its `container_name`, and identify the appropriate container dependencies.

iv. Once you have done so, run `docker compose up -d` and navigate to http://127.0.0.1:3000 to open the Grafana UI. The default username and password is admin and admin.

v. In the interface, add a data source referencing your prometheus service.

vi. Next create a new dashboard that visualizes the metrics you created in the previous prometheus configuration section.

vii. Once you have created your dashboard, take a screenshot and include it in your `grafana` directory's `README.md` file.