Notes:

Module 6.1 : Testing Python code with pytest

In this section, we will work on the streaming code in the 04-deployment section. We will make the code better from an engineering pov by adding tests.

Creating and activate virtual environment:

1. Go to the correct directory by:

```
cd mlops-zoomcamp/06-best-practices/code
```

2. Activate any existing conda virtual evironment:

```
. /opt/homebrew/anaconda3/bin/activate && conda activate
/opt/homebrew/anaconda3/envs/mlops-zoomcamp-venv;
```

3. Create a new virtual environment: `pipenv install`
4. Activate it by `pipenv shell`
5. Install pytest with `pipenv install --dev pytest`. We use the dev argument cause we want pytest only in the dev environment and not in the production environment.
6. Find the location of your virtual environment by typing `pipenv --venv`. You'll get the path `/Users/aasth/.local/share/virtualenvs/code-JCzC6QQn` to the venv. Copy the path.
7. We need to set up our python envionment in VSCode. Hit `Cmd+Shift+P` -> `Select Python Interpreter` and paste the path of the venv that you copied in step 6.
8. We will configure the python tests. Click on the `Testing` tab which is lcoated on the left panel of VSCode. Click om the `Configure Python Tests` button. Select `pytest` and the `test` directory.

Testing if Docker works

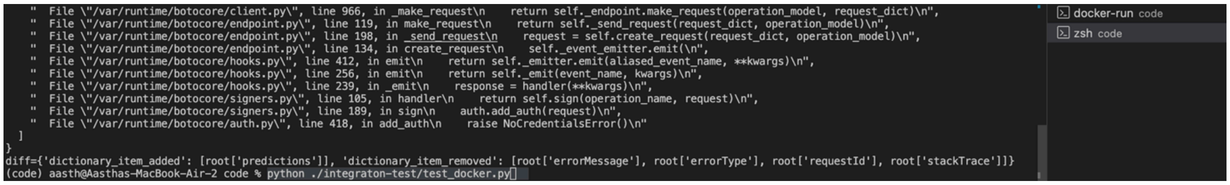
1. Open your Docker app and in the terminal with the `code` environment activated, run `docker build -t stream-model-duration:v2 .`.

Note: If you already have a previous docker container running, it might be exposed to *the same port that we will use now. A good practice is to use the `docker ps` command to lists all active Docker containers along with their respective port mappings. You can stop the previous docker containers in case you don't need it.*

2. Now run in the same terminal:

```
docker run -it --rm \
    -p 8080:8080 \
    -e PREDICTIONS_STREAM_NAME="ride_predictions" \
    -e RUN_ID="e1efc53e9bd149078b0c12aeaa6365df" \
    -e TEST_RUN="True" \
    -e AWS_DEFAULT_REGION="eu-west-1" \
    stream-model-duration:v2
```

3. Then open up a new terminal with the `code` directory (`cd mlops-zoomcamp/06-best-practices/code`) and activate the `code` evnrionment. Run `python ./integraton-test/test_docker.py` . You should see this:



```
    "  File \"/var/runtime/botocore/client.py\", line 966, in _make_request\n    return self._endpoint.make_request(operation_model, request_dict)\n",
    "  File \"/var/runtime/botocore/endpoint.py\", line 119, in make_request\n    return self._send_request(request_dict, operation_model)\n",
    "  File \"/var/runtime/botocore/endpoint.py\", line 198, in _send_request\n    request = self.create_request(request_dict, operation_model)\n",
    "  File \"/var/runtime/botocore/endpoint.py\", line 134, in create_request\n    self._event_emitter.emit(\n",
    "  File \"/var/runtime/botocore/hooks.py\", line 412, in emit\n    return self._emitter.emit(aliased_event_name, **kwargs)\n",
    "  File \"/var/runtime/botocore/hooks.py\", line 256, in emit\n    return self._emit(event_name, kwargs)\n",
    "  File \"/var/runtime/botocore/hooks.py\", line 239, in _emit\n    response = handler(**kwargs)\n",
    "  File \"/var/runtime/botocore/signers.py\", line 105, in handler\n    return self.sign(operation_name, request)\n",
    "  File \"/var/runtime/botocore/signers.py\", line 189, in sign\n    auth.add_auth(request)\n",
    "  File \"/var/runtime/botocore/auth.py\", line 418, in add_auth\n    raise NoCredentialsError()\n"
  ]
}
diff={'dictionary_item_added': [root['predictions']], 'dictionary_item_removed': [root['errorMessage'], root['errorType'], root['requestId'], root['stackTrace']]}
(code) aasth@Aasthas-MacBook-Air-2 code % python ./integraton-test/test_docker.py
```

Running the unit tests

In the code folder with the `code` environment activated, run `pytest tests/` and you should get:

```
=================================================== test
session starts
===================================================
platform darwin -- Python 3.9.6, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/aasth/Desktop/Data analytics/MLOps/datatalks-
zoomcamp/mlops-zoomcamp/06-best-practices/code
collected 4 items

tests/model_test.py ....
[100%]

==================================================== 4 passed
in 1.06s ====================================================
```

Module 6.2 : Integration tests with docker-compose

Integration tests:

Unit tests just test partial of the code, we still need to test the whole code and we will do that using integration test. We will convert our `test_docker.py` file into an intergration test.

We can do that by adding assertions and the `DeepDiff` library. The `test_docker.py` file returns a dictionary and we use deepdiff to see the difference between the expected dictionary and the returned dictionary.

Load the model first from local env and remove the dependency in S3

We were loading our model from S3 in the model.py file. We can remove that dependency by adding `get_model_location()` function to the model file. It will check if the user has specified a local path where the model is downnloaded and load it from that path. If the user doesn't specify a path, then it will load the model from S3.

Then in the `code` directory with the `code` environment activated run:

```
docker run -it --rm \
    -p 8080:8080 \
    -e PREDICTIONS_STREAM_NAME="ride_predictions" \
    -e RUN_ID="Test123" \
    -e MODEL_LOCATION="/app/model" \
    -e TEST_RUN="True" \
    -e AWS_DEFAULT_REGION="eu-west-1" \
    -v "$(pwd)/model:/app/model" \
    stream-model-duration:v2
```

Then in another terminal (in the `code` directory with the `code` environment activated), run `python ./integraton-test/test_docker.py` to test the model which has been downloaded in local env.

Automating tests

Right now to run the latest version of the tests, we need to build the docker container first and then run the docker run command and in another terminal run the test. We can automate this:

1. Create a new file named `run.sh` under integration-test and changes its permissions by running `chmod +x ./integraton-test/run.sh`. `chmod +x` on a file (your script) only means, that you'll make it executable.

Run.sh file:

- The first line is `#!/usr/bin/env bash` which basically means that we are going to use the bash command.
- The `cd "$(dirname "$0")"` takes us to the directory of our script (the model directory)
- The

```
LOCAL_TAG=date +"%Y-%m-%d-%H-%M"
export LOCAL_IMAGE_NAME="stream-model-duration:${LOCAL_TAG}"
```

line maintains the build version

- The `docker build -t ${LOCAL_IMAGE_NAME} ..` builds the image
- `docker compose up -d`: start docker compose
- `sleep 1`: give the container some time to start so we make the program sleep for sometime
- `ERROR_CODE=$?`: reads the exit status of the last command executed. The error code will be 0 if the script executes successfully.
- `if [${ERROR_CODE} != 0]; then docker compose logs fi`: When you see a non-zero error code then, print the docker logs
- `docker compose down`: stops containers and removes containers

2. Create docker-compose.yaml based on the format of Compose file reference
3. Open a terminal in a non-virtual env, run ./run.sh.

Module 6.3 : Testing cloud services with LocalStack

We wrote unit tests to test our function and intergation test to test our service but we didn't test Kinesis. We need to test the Kinesis connection or the function that puts the responses to the Kinesis stream with LocalStack. LocalStack helps us to develop and test AWS applications locally.

1. Add the kinesis service to the docker-compose.yaml file.
2. First `cd integraton-test` and then run `export LOCAL_IMAGE_NAME=123`

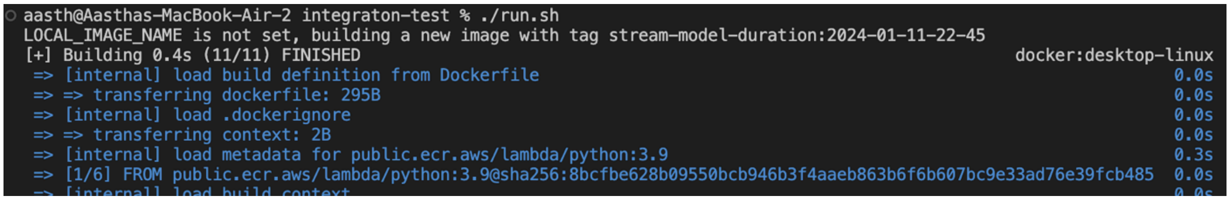
and then `export PREDICTIONS_STREAM_NAME=ride_predictions` and finally `docker-compose up kinesis` in the `code` environment.

3. We will create a stream locally using localstack. In another terminal, in the `code` directory and in the `code` envrionment,run:

```
aws --endpoint-url=http://localhost:4566 \
    kinesis create-stream \
    --stream-name ride_preditions \
    --shard-count 1
```

4. We add the `create_kinesis_client()` function in our model.py file where we use a similar logic that we used when we removed the dependency from S3. We check if the local path to kinesis is set, then we can use that path otherwise we actually call the AWS Kinesis client.

5. Stop the `docker-compose up kinesis` (by pressing control+C) and then in a new terminal in a **non-virtual envrionment**, run `cd mlops-zoomcamp/06-best-practices/code/integraton-test` and then run `./run.sh`:



```
○ aasth@Aasthas-MacBook-Air-2 integraton-test % ./run.sh
  LOCAL_IMAGE_NAME is not set, building a new image with tag stream-model-duration:2024-01-11-22-45
  [+] Building 0.4s (11/11) FINISHED                                                   docker:desktop-linux
   => [internal] load build definition from Dockerfile                                                 0.0s
   => => transferring dockerfile: 295B                                                                 0.0s
   => [internal] load .dockerignore                                                                    0.0s
   => => transferring context: 2B                                                                      0.0s
   => [internal] load metadata for public.ecr.aws/lambda/python:3.9                                    0.3s
   => [1/6] FROM public.ecr.aws/lambda/python:3.9@sha256:8bcfbe628b09550bcb946b3f4aaeb863b6f6b607bc9e33ad76e39fcb485  0.0s
   => [internal] load build context                                                                    0.0s
```

6. We will now check the content in the stream. Get the shard ID present in the stream by `aws --endpoint-url=http://localhost:4566 kinesis list-shards --stream-name ride_preditions`. It will return the shard ID as "shardId-000000000000". Use this to set the SHARD variable.

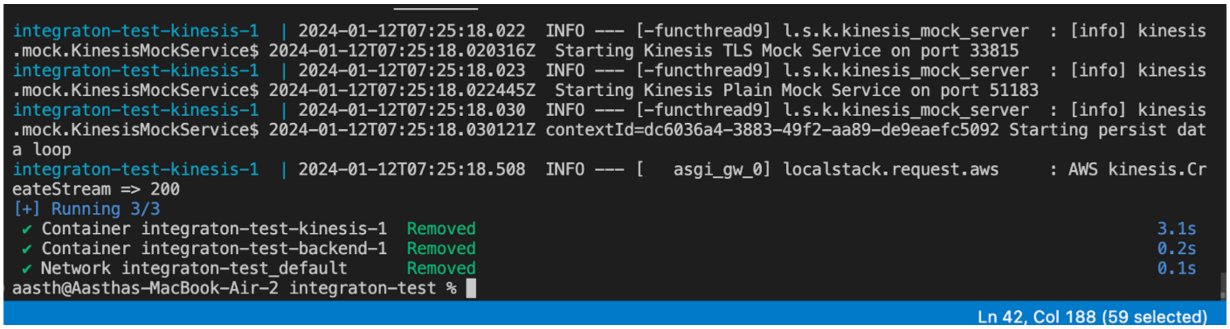
```
export SHARD="shardId-000000000000"
export PREDICTIONS_STREAM_NAME=ride_predictions
aws  --endpoint-url=http://localhost:4566 \
    kinesis     get-shard-iterator \
    --shard-id ${SHARD} \
    --shard-iterator-type TRIM_HORIZON \
    --stream-name ${PREDICTIONS_STREAM_NAME} \
    --query 'ShardIterator'
```

_Note: Step 6 didn't work for me. It says `An error occurred (ResourceNotFoundException) when calling the GetShardIterator operation: Stream arn arn:aws:kinesis:us-east-1:000000000000:stream/ride_predictions not found`__

7. Create the test_kinesis.py file

8. Edit the logic in run.sh to incorporate the test_kinesis.py as well (similar logic to the test_docker.py file that we used in run.sh)

9. Stop any docker containers that are running. In a new terminal in a **non-virtual envrionment**, run `cd mlops-zoomcamp/06-best-practices/code/integraton-test` and then run `./run.sh`. You can see that the kinesis service has finished running:



```
integraton-test-kinesis-1  | 2024-01-12T07:25:18.022  INFO --- [-functhread9] l.s.k.kinesis_mock_server  : [info] kinesis
.mock.KinesisMockService$ 2024-01-12T07:25:18.020316Z  Starting Kinesis TLS Mock Service on port 33815
integraton-test-kinesis-1  | 2024-01-12T07:25:18.023  INFO --- [-functhread9] l.s.k.kinesis_mock_server  : [info] kinesis
.mock.KinesisMockService$ 2024-01-12T07:25:18.022445Z  Starting Kinesis Plain Mock Service on port 51183
integraton-test-kinesis-1  | 2024-01-12T07:25:18.030  INFO --- [-functhread9] l.s.k.kinesis_mock_server  : [info] kinesis
.mock.KinesisMockService$ 2024-01-12T07:25:18.030121Z contextId=dc6036a4-3883-49f2-aa89-de9eaefc5092 Starting persist dat
a loop
integraton-test-kinesis-1  | 2024-01-12T07:25:18.508  INFO --- [   asgi_gw_0] localstack.request.aws     : AWS kinesis.Cr
eateStream => 200
[+] Running 3/3
 ✔ Container integraton-test-kinesis-1  Removed                                                                    3.1s
 ✔ Container integraton-test-backend-1  Removed                                                                    0.2s
 ✔ Network integraton-test_default      Removed                                                                    0.1s
aasth@Aasthas-MacBook-Air-2 integraton-test % 
```

Ln 42, Col 188 (59 selected)

Module 6.4: Code quality - linting and formatting

In Python, it is recommended to follow the PEP8 guidelines. This ensures clean, standard code formats and helps code readability as well as minimizing diffs. To do this automatically, we use Formatters

In addition to following style guides, we also want our code to be free of bad practices and deprecated language features.

Linting:

We may want styled code, however, conforming to the PEP8 style manually may be cumbersome. So we can use Linters instead. Linters are pieces of software that make sure the code conforms to a certain style with minimal hinderence to the developer. For Python, a common linter is pylint. Pylint ensures that our code follows the PEP8 style guide and follows good coding practices.

Installing pylint:

1. Go to the correct directory by: `cd mlops-zoomcamp/06-best-practices/code`
2. Activate any existing conda virtual evironment: `.`

`/opt/homebrew/anaconda3/bin/activate && conda activate /opt/homebrew/anaconda3/envs/mlops-zoomcamp-venv;`

3. Install pylint and create a new virtual envrinment: `pipenv install --dev pylint`. We use the dev argument cause we want pytest only in the dev environment and not in the production environment.
4. Activate it by `pipenv shell`

You can check the code quality of a file with `pylint file_name.py`, or all the files under the current folder with `pylint **/*.py`.

It might be cumbersome to go through the output of the pylint for each file. You can also enabe the linter in VSCode so that the linter will warn you about good coding practice while you are coding. To do this, Hit `Cmd+Shift+P` -> `Python: Select Linter` -> `pylint`. Now when you open any python file, you'll see that the linter will underline lines where there is a code quality issue.

To ignore certain errors raised by pylint:

For example, let's say you want to ignore the `missing-function-docstring` error that is raised by pylint. You can do so by creating a `pyproject.toml` file in the directory where you will run pylint and include the following in the `pyproject.toml` file:

`pyproject.toml` file:

```
[tool.pylint.messages_control]

disable = [
    "missing-function-docstring"]
```

Formatting & Imports:

For formatting the code (like for example, removing trailing whitespaces), you can use a tool called `black`. To fix formatting import statements, use the tool called `isort`.

Use `pipenv install --dev black isort` to install both the tools. (Follow the same steps as we did for installing pylint).

It's good practice to commit your code before running the black and isort library on any of the files because these libraries will change the contents of your files. So if you commit your files before running black&isort, you can roll back to the previous version of the file easily.

You can use `black --diff .` to see what changes Black would've made to the files but Black won't actually make any changes to the files. If you are happy with its suggested changes and want to apply it to the files, you can use `black .`. If you want Black to ignore certain changes, you can modify the `pyproject.toml` file like we did for pylint.

Similarly, you can run isort by `isort --diff .` to see what changes isort would have made to files. To actually apply these changes, do `isort .`. If you want isort to ignore certain changes, you can modify the `pyproject.toml` file like we did for pylint.

We can configure to run pytests,pylint,black and isort automatically before committing or using CI/CD. We will look at that later.

Module 6.5 : Git pre-commit hooks

Git has a way to fire off custom scripts when certain important actions occur using something called git hooks. An example of a git hook is a git post-commit hook that runs after the commit process is completed.

We are going to look into the pre-commit hooks. These run first, before you even type in a commit message.

Installing pre-commit hooks:

1. Go to the correct directory by: `cd mlops-zoomcamp/06-best-practices/code`
2. Activate any existing conda virtual evironment: `. /opt/homebrew/anaconda3/bin/activate && conda activate /opt/homebrew/anaconda3/envs/mlops-zoomcamp-venv;`
3. Install pylint and create a new virtual envrinment: `pipenv install --dev pre-commit`. We use the dev argument cause we want pytest only in the dev environment and not in the production environment.
4. Activate it by `pipenv shell`

Let's pretend that our existing `code` directory is a new repo that we just cloned from github. In the `code` directory, do `git init` to initialize an empty git repository in the code folder. You'll see a `.git` folder being created after you execute the `git init` command. Type `ls -a` in the command line to verify that the .git folder was created.

Then we need to create a `.pre-commit-config.yaml` file before running the `pre-commit` command.

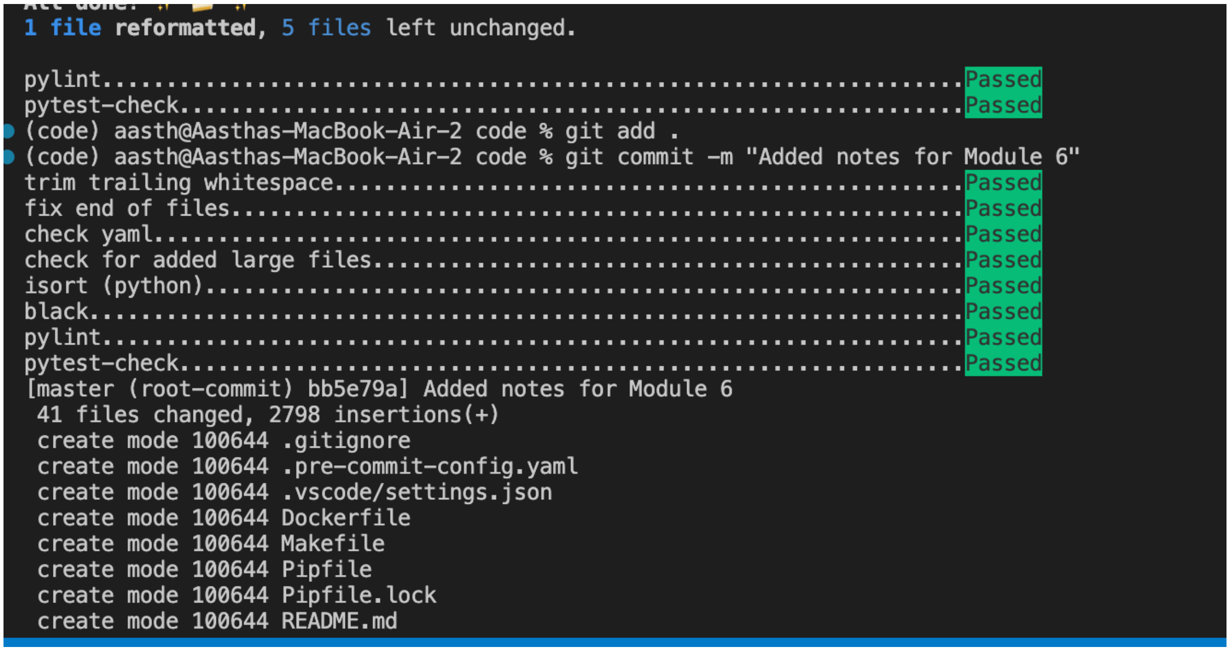
Type `pre-commit sample-config` (in the `code` directory with the `code` environment activated) to see see a sample content for the `.pre-commit-config.yaml` file. You can copy this content and then create a `.pre-commit-config.yaml` file and paste the contents in this file.

We can add more hooks to this file by googling "pylint pre-commit" to see the pre-commit config file for pylint. Usually online repos will provide a config like:

```
-   repo: https://github.com/pycqa/isort
        rev: 5.10.1
        hooks:
        - id: isort
          name: isort (python)
```

Run `pre-commit install` in the command line. This creates a pre-commit folder in the `.git` folder. The `.git` folder isn't committed to git. The `.git` folder is a local folder that is created which means whenever you clone a repo, you need to run `pre-commit install` first to create the pre-commit folder.

Now, proceed to do `git add .` and `git commit -m "some message"` and you will see that some hooks may say "Failed" but it will have a "files were modified by this hook" message which means the hook modified the file to make it pass. So the next time you do `git add .` and `git commit`, you'll notice that same hook will now Pass. In the below screenshot, the "End Of Files" hook Failed earlier but in my second commit, it passed as the file was modified by the hook:



```
All done! ✨ 🍰 ✨
1 file reformatted, 5 files left unchanged.

pylint...................................................................Passed
pytest-check.............................................................Passed
(code) aasth@Aasthas-MacBook-Air-2 code % git add .
(code) aasth@Aasthas-MacBook-Air-2 code % git commit -m "Added notes for Module 6"
trim trailing whitespace.................................................Passed
fix end of files.........................................................Passed
check yaml...............................................................Passed
check for added large files..............................................Passed
isort (python)...........................................................Passed
black....................................................................Passed
pylint...................................................................Passed
pytest-check.............................................................Passed
[master (root-commit) bb5e79a] Added notes for Module 6
 41 files changed, 2798 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 .pre-commit-config.yaml
 create mode 100644 .vscode/settings.json
 create mode 100644 Dockerfile
 create mode 100644 Makefile
 create mode 100644 Pipfile
 create mode 100644 Pipfile.lock
 create mode 100644 README.md
```

Module 6.6 : Makefiles and make

Note: Mac already installed make

Make is a tool for automating various steps for production.

`make` looks for the `Makefile`, which contains the steps needed to automate the build.

`make` makes use of aliases, like:

```
run:
    echo 123
```

here `run` is an alias. Now when `make run` is executed, `echo 123` is executed as a result.

We can also make aliases depend on other aliases

```
test:
    echo test
run: test
    echo 123
```

here `run` depends on `test`, and when `make run` is executed, both `echo test` and `echo 123` are executed in that order.

In our case, we want to run many things before running the program or commiting or deploying to AWS Lambda/GCP Functions/.... To do so we can make use of `Makefile`. We want to run:

1. Tests (Unit tests and integration tests): using `pytest`
2. Quality checks: `pylint`, `black`, `isort`

An example on how `Makefile` can be used in our case:

```
LOCAL_TAG:=$(shell date +"%Y-%m-%d-%H-%M")
LOCAL_IMAGE_NAME:=stream-model-duration:${LOCAL_TAG}

test:
        pytest tests/

quality_checks:
        isort .
        black .
        pylint --recursive=y .
```

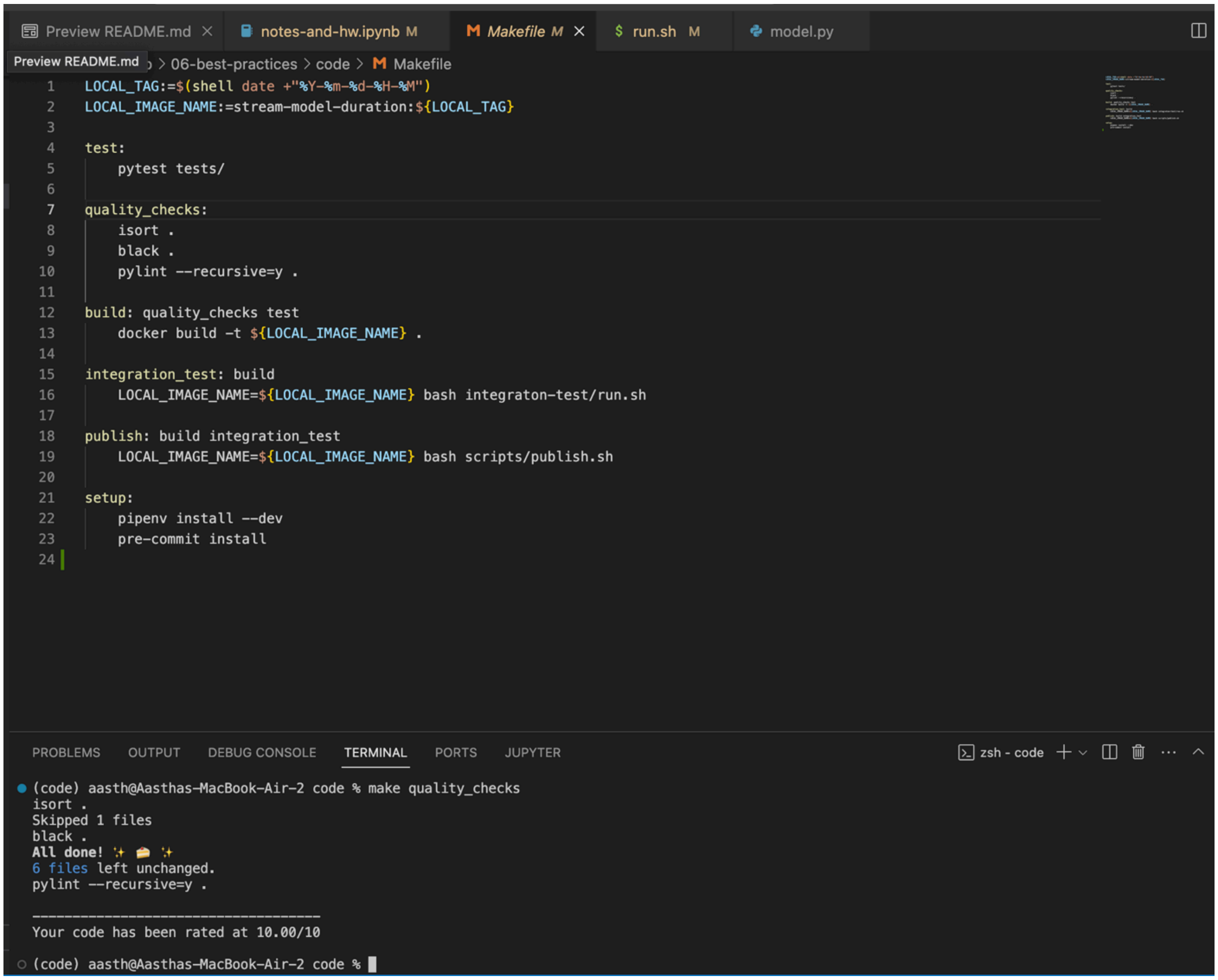
```
build: quality_checks test
	docker build -t ${LOCAL_IMAGE_NAME} .

integration_test: build
	LOCAL_IMAGE_NAME=${LOCAL_IMAGE_NAME} bash integraton-test/run.sh

publish: build integration_test
	LOCAL_IMAGE_NAME=${LOCAL_IMAGE_NAME} bash scripts/publish.sh

setup:
	pipenv install --dev
	pre-commit install
```

To run any command, you can type `make test` or `make quality_checks`, etc:



```
LOCAL_TAG:=$(shell date +"%Y-%m-%d-%H-%M")
LOCAL_IMAGE_NAME:=stream-model-duration:${LOCAL_TAG}

test:
    pytest tests/

quality_checks:
    isort .
    black .
    pylint --recursive=y .

build: quality_checks test
    docker build -t ${LOCAL_IMAGE_NAME} .

integration_test: build
    LOCAL_IMAGE_NAME=${LOCAL_IMAGE_NAME} bash integraton-test/run.sh

publish: build integration_test
    LOCAL_IMAGE_NAME=${LOCAL_IMAGE_NAME} bash scripts/publish.sh

setup:
    pipenv install --dev
    pre-commit install

```

```
● (code) aasth@Aasthas-MacBook-Air-2 code % make quality_checks
isort .
Skipped 1 files
black .
All done! ✨ 🍰 ✨
6 files left unchanged.
pylint --recursive=y .

------------------------------------
Your code has been rated at 10.00/10

○ (code) aasth@Aasthas-MacBook-Air-2 code %
```

Module 6b.5: CI/CD (WIP)

Introduction:

CI/CD is an import DevOps practice which is helpful for shortening the delivery of our software applications.

- CI: This involves developing, testing, and packaging code in a structured manner.
- CD: This is responsible for delivering the integrated code to various dependent applications.

For our use case, we will be implementing a complete CI/CD pipeline using GitHub Actions. This pipeline will automatically trigger jobs to build, test, and deploy our service for every new commit or code change to our repository.