DSC Capstone Sequence

Lecture 08

Long Running Jobs

Lecture Outline

- Remote Job Submission
  - Running Long Running Jobs)
- Test Data
  - Developing Long Running Jobs
- Logging
  - Debugging Long Running Jobs

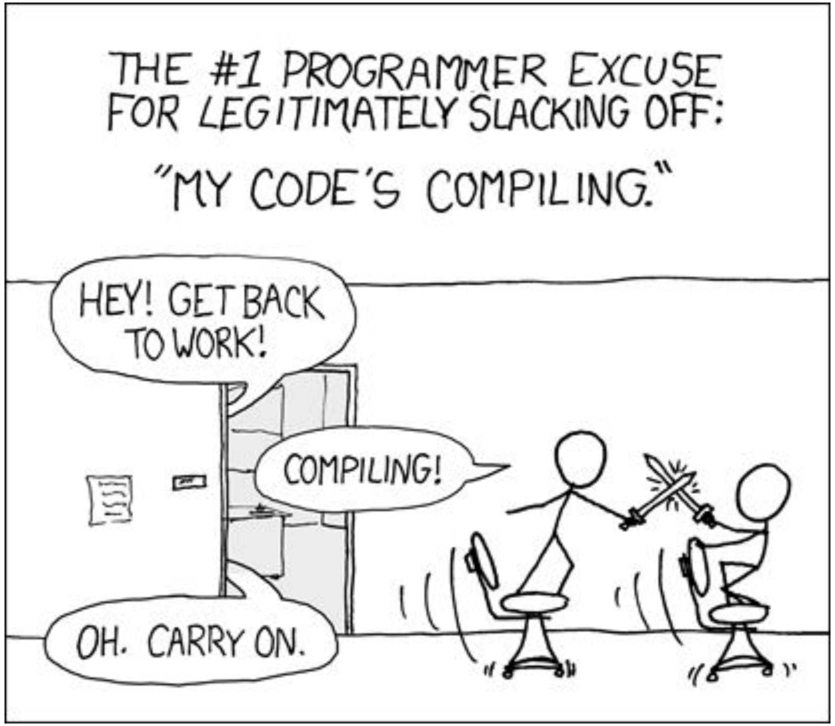
Remote Job Submission

Long Running Jobs

Why watch your 3HR job run, when you can:

- Log onto a server
- Start your job (build script)
- Log off server
- [[...]]
- Return when job is finished

Compute time != Keyboard time



Requirements for Long Running Jobs

Effective use of running "hands-off" project execution requires:

- The server to remain running after logout
- Non-interactive code that saves output to file
  - e.g. a build script with targets
- Writing code that's largely correct
  - It is a *long* running job, afterall!
- Good logging
  - Debugging is more difficult and more removed.

DSMLP and Background Pods

- DSMLP is a "Kubernetes Cluster" that manages compute resources (pods).
- User requests a pod (e.g. using a launch script) and Kubernetes allocates resources for you (a pod running a container with requested memory).
- Kubernetes has a command-line tools for interacting with Pods:
  - kubectl is the kubernetes "control" tool -- allowing you to list/delete your running pods.
  - kubesh is the kubernetes "shell" -- connects you to the shell in a specified pod.

Kubernetes command-line tools allow you to connect, disconnect, and reconnect to pods without shutting them down!

Connecting to DSMLP Background Pods

- To launch a background pod:
    - `launch-180.sh -G <group> -b`
- Text displayed to the terminal lists your `pod-ID with instructions.`
- Connect to your pod using the command:
    - `kubesh <pod-ID>`
- After starting job, disconnect from pod (`ctrl-d`)
- Check the pod is still running:
    - `kubectl get pods`
- **Delete your pod when finished**
    - `kubectl delete pods`

Running Jobs on Background Pods

- You can now attach/detach to/from a running pod, without shutdown.
- From within the pod, how do you run a job "in the background" so that you can exit the pod with the job still running?

Running a job in the background:

- `python run.py target &`
- `&` tells shell to start the process in the background
    - Terminal remains free

To stop a job, use `kill <PID>` where PID is the Process ID.

- Find your job's PID using the `jobs` command, or `ps` command.

Advice for Long Running Jobs

- Be sure code is *correct* before running (else you waste *a lot* of time!)
- Develop code on small data w/small resources before committing to running the big job
- Use a lot of logging; they are difficult to debug.
- Increase the timeout on a POD by editing the launch script:
    - Pods shutdown automatically after ~3 hours
    - Edit the launch script as advised here
    - Change the K8S_TIMEOUT_SECONDS variable (up to 12 hours; longer with permission).

Test Data

Developing on Test Data

- Create small "realistic" test data on which to develop code
    - Speeds up code development iterations
    - Using real (large) data naturally takes longer...
- Integrate test data into doctests and/or unittests
    - Test changes to your code against this data to make debugging easier.
- Test data works well to check that the steps of your project "fit together" properly.

What is Test Data?

Small made-up data, that is realistic enough to *test* code (quickly)

- It is NOT "real" data; the developer creates it.
  - Test data should NOT be "a small sample" of real data
  - Test data is designed to test correctness of the code.
  - Developer should design each line of test data to test a specific attribute!
- Should not contain sensitive information.
- A test-data file should only be a few lines.
  - You should be able to test the correctness of code on this data BY HAND.
- The statistics of the test data may not look realistic, but the schema should.

Test Data and Versioning (Git)

- Test data *should* be versioned!
- It is developed, and changes, to test code.
    - This meets the criteria for versioning: changing material that is not built from a static data source and versioned code.
- Place test data in a testing directory, like `test/testdata`.
    - `test` is where developers place their unit-tests.

Test data is not real data; **never version control data**.

How to use test data

- Test data tests the correctness of code *before running it on real data.*
  - Run the test data on a small container (small RAM; fast), before starting a large container and letting it run overnight on the "real" data.
- Particularly useful for testing the steps of your pipeline fit together.
- Substitute test-data for real data in your build script to quickly run tests.

`test` Target

- Implement a `test-data` target that runs the same code as `data`, only on your test data.
- Then, to train a model on your test data, you might run:
  - `python run.py test-data features model`
- The target `test` is a standard target that implements the `all` target on test data. That is, it builds your entire project on test data.

How Project Code is Graded

Your project will be run on test-data on the DSMLP server, in a clean location:

1. `launch-180.sh -i <student-dockerhub-repo>`
2. `git clone <student-repo>` (or copy code from gradescope)
3. `python run.py test`
4. `(run script that checks generated files)`

In Step 3, the test target should generate all your project output, run on test data that's versioned in your repository. It should take no more than a few minutes.

Logging

Logging

When running build scripts without interaction:

- Hard to tell why something failed
- Takes a long time to try and retry
- You have no access to terminal output
  - Must log messages to file, for later reading

Print Debugging

- Simple: log processes with print statements
- You can "redirect" standard output (stdout) using `>`
    - `python run.py all > log.txt &`
    - This redirects all terminal output to the file `log.txt`
- Disadvantage: hard to tell which statement came from where and when.

`logging` Module

- logging module comes in the python standard library
- Automatically records:
  - when logging happened (can use this to time your steps!)
  - which file/function the logging occurred in
- Can set "logging levels" to restrict output based on context
  - debug/info/warn/critical
  - use debug only when debugging
  - use info to record performance (e.g. time for each step)

Using the logging module

```
logging.basicConfig(filename=logname, filemode='a',
                    format='%(asctime)s,%(msecs)d %(name)s %(levelname)s %(message)s',
                    datefmt='%H:%M:%S',
                    level=logging.DEBUG)
logging.info("Log info Here")
```