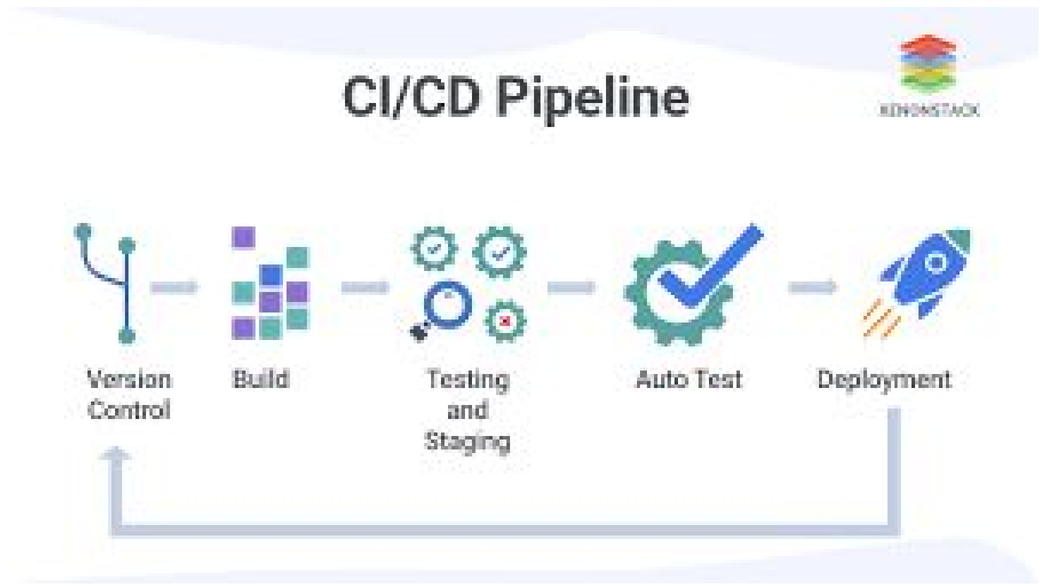
Lecture Notes
November 19, 2020

“Assignment T4: Initial Demo” due Monday. https://courseworks2.columbia.edu/courses/104335/assignments/491014 Schedule your demo slot with your IA mentor asap!

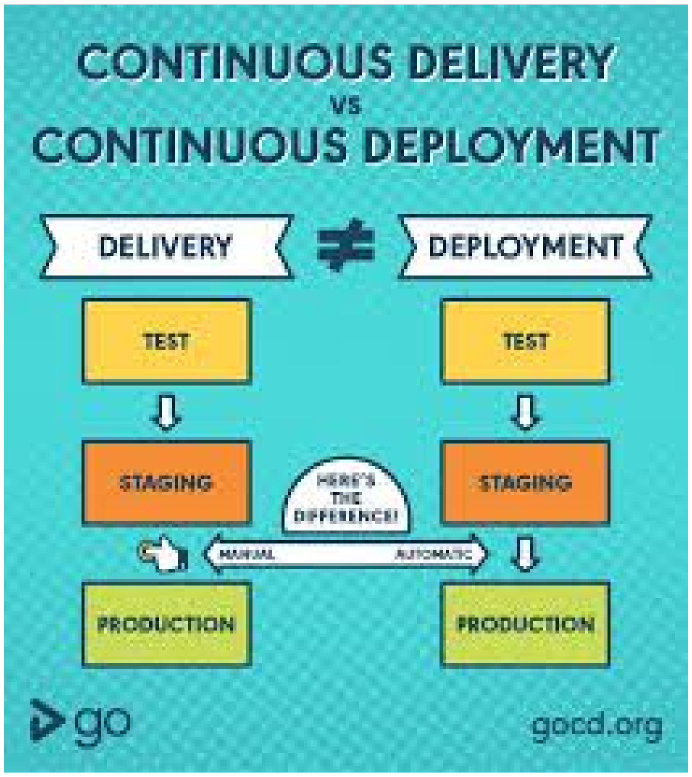
Volunteer demos: Alchemist, GUO, Advance Engineers

Working as a team: where do test assertions come from?

Continuous integration (CI) tools hook a version control system with a build tool to automatically re-run the build after every commit to the shared repository, so errors can be detected quickly.



Continuous Delivery/Deployment (CD) adds automatic delivery and/or deployment to continuous integration = Installs new build (after testing!) on the production server(s) and/or produces a patch for customers.



Continuous deployment has no human in the loop!

This requires that testers be able to use automated testing tools, with automatically checked assertions, for <u>all</u> testing, including end-to-end system testing and acceptance testing - not just for unit and integration testing.

Test assertions, at the full system level or otherwise, usually compare actual outputs to expected outputs, meaning the expected outputs must be known in advance. But how do the testers know in advance what the expected output should be?

For some problems, it’s much easier to check whether a given answer is correct than to produce an answer, so some test assertions might perform this kind of checking rather than hardwire a specific answer. But how do the testers know in advance how to write assertions to do these checks?

Testers need an oracle!

Possible test oracles:

The human tester "knows" and can write it down. This is much easier for unit testing that for full system testing, where there may be many side-effects like network traffic, file changes, database updates, device control, etc,

Naive - if it doesn't crash, hang, or produce obviously nonsensical output, it works!

Detailed prose specification - common for networking, database, file system and other standard protocols necessary for interoperability, rare for user-facing functionality of business/consumer applications

Formal specifications -

Wait until users complain that the software didn't do what they expected, then write test cases to check for what they say they expected 😁

Pseudo-oracles may be possible in some cases, where one test case acts as an oracle for another test case. You may not know whether the result of either test case is right, but if the results are inconsistent you know (at least) one of them is wrong.

Regression testing is the most common form of pseudo-oracle. We know what the test results were for the old version and we re-run the tests to see if they have the same or different results on the new version.

```
program_v1(input)->output_v1

program_v2(input)->output_v2

output_v1 =? output_v2
```

Some test cases *should* change results on the new version. For example, when we try to fix a bug that caused a test case to fail on the old version, we want that test case to pass on the new version.

But (apparently) unrelated test cases should not be affected - if they are, that indicates a hidden dependency and (probably) a new bug.

"Differential testing" is another kind of pseudo-oracle, which requires a second *independently developed* implementation of the same functionality.

Practical mostly for standard protocols, e.g., there are many different implementations of SSL/TLS and HTTP

```
program_1(input)->output_1

program_2(input)->output_2

output_1 =? output_2
```

- ➢If they disagree, one or both is wrong
- ➢If they agree, that does not guarantee both are right, but does provide some confidence that they are right

It's critical that the implementations are truly independent, with no common third-party libraries or other common factors

What might be examples of common factors that could make it likely that the "independent" developers made the same mistakes?

“Metamorphic testing” is another approach to pseudo-oracles that is less widely used, but works for only-one-implementation.

Say we have a machine learning application that is supposed to learn from a series of traffic images how to control the steering of a self-driving car, so it obeys traffic laws and doesn’t cause accidents.

How do we know what is “correct” steering for a given real-world driving scenario? How would we write test cases?

Show DeepTest images

Synthetic but realistic changes to driving images such as adding snow, fog, rain, more sunlight, less sunlight, etc. should not (in most cases) change automobile steering

Metamorphic testing starts with an original input and its original output. We do not need to know whether the original output is correct, we just know that it was produced by the program.

We create a new test case by *deriving* a new input from the original input, and *predicting* the expected new output from the original output - the prediction is usually that the output is the same, nearly the same, or changed in a simple way. If the actual new output deviates too much from the predicted output, there is (probably) a bug.

```
program(input_1)->output_1

deriveInput(input_1)->input_2

predictOutput(input_1, input_2, output_1) ->
predicted_output_2

program(input_2)->output_2

check(predicted_output_2, output_2) -> ??
```

Example metamorphic properties for a sorting program (lowest first) applied to an array of numbers

```
sort(5,2,3,1,4) -> 1,2,3,4,5
```

Permutative: if we shuffle the order of the input array, the sorted output array should be the same

```
sort(3,4,1,2,5) -> 1,2,3,4,5 😀
```

Additive: if we add N to every element of the input array, the sorted output array should be in the same order

```
sort(25,22,23,21,24) -> 21,22,23,24,25  😀
```

Multiplicative: if we multiply every element of the input array by (positive) N, the sorted output array should be in the same order

```
sort(15,6,9,3,12) -> 3,6,9,12,15  😀
```

Invertive: if we multiply every element of the input array by (negative) N, the sorted output array should be in the opposite order

```
sort(-5,-2,-3,-1,-4) -> -5,-4,-3,-2,-1  😀
```

Inclusive: if we add one new element to the input array, the sorted output array should be the same except for the placement of that new element

sort(5,2,42,3,1,4) -> 1,2,3,4,5,42 😀

Exclusive: if we remove one element from the input array, the sorted output array should be the same except it is missing that one element

sort(5,3,1,4) -> 1,3,4,5 😀

None of the metamorphic test cases above found a bug. How would we know if one of them did find a bug?

Original:

sort(5,2,3,1,4) -> 1,2,3,4,5

Permutative: if we shuffle the order of the input array, the sorted output array should be the same

sort(3,4,1,2,5) -> 2,4,1,5,3 😞

Additive: if we add N to every element of the input array, the sorted output array should be in the same order

sort(25,22,23,21,24) -> 25,25,25,25,25 😳

Multiplicative: if we multiply every element of the input array by (positive) N, the sorted output array should be in the same order

sort(15,6,9,3,12) -> 6,12,3,15,9 😟

Invertive: if we multiply every element of the input array by (negative) N, the sorted output array should be in the opposite order

sort(-5,-2,-3,-1,-4) -> -1,-4,-3,-2,-5 😕

Inclusive: if we add one new element to the input array, the sorted output array should be the same except for the placement of that new element

sort(5,2,42,3,1,4) -> ⌛ 😖

Exclusive: if we remove one element from the input array, the sorted output array should be the same except it is missing that one element

sort(5,3,1,4) -> 🌋 😳

Metamorphic testing can be very useful for machine learning applications, like DeepTest, where it would be challenging to write assertions: For most road conditions, different human drivers will not necessarily steer at exactly the same angle and the same human driver will not necessarily steer at the same angle every time.

But there are also some anomalies when metamorphic testing is applied to model training software: If the order of a large dataset is permuted, the resulting model may not be exactly the same. Try it!

Assignment T5: Second Iteration:
https://courseworks2.columbia.edu/courses/104335/assignments/490853

Assignment T6: Final Demo:
https://courseworks2.columbia.edu/courses/104335/assignments/491047

Extra Credit: Optional Demo Video
https://courseworks2.columbia.edu/courses/104335/assignments/543277