UNIVERSITY OF CAMBRIDGE

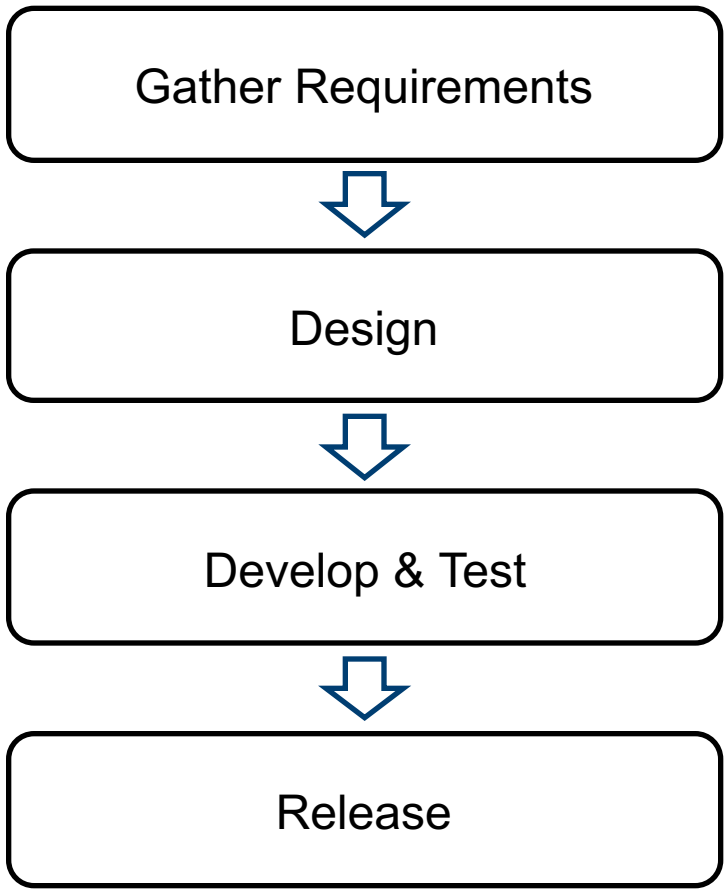
hpcs Cambridge HPC service

Continuous Integration and Numerical Regression Testing for ~~HPC~~ Scientific Codes

Filippo SPIGA[1,2] <fs395@cam.ac.uk>

[1] High Performance Computing Service, University of Cambridge
[2] Quantum ESPRESSO Foundation

How *we* build software for science?



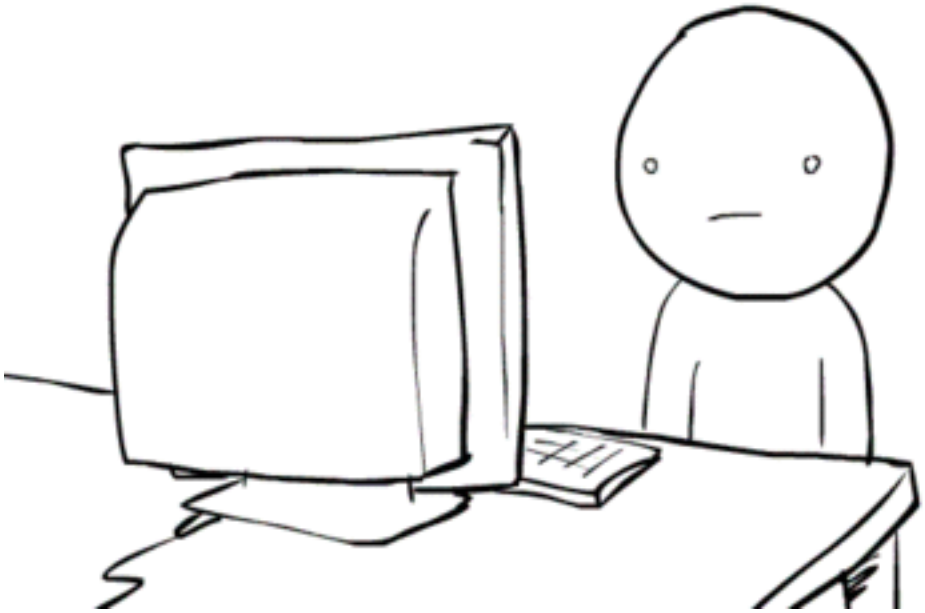
→ *I know what is needed, I know what Me and my group need to compute*

→ *I think this is the correct way because this is what interests me*

→ *I developed the code and It works for me!*

→ *I pass the software to another collaborator, he/she will figure out how to compile/run/…*

This happens… lot of times!



Because everybody …

The (invalid) argument


Why bother about all of this, we always did “stuff” in this way and it was good enough…


WRONG


Bottom line

BETTER SOFTWARE

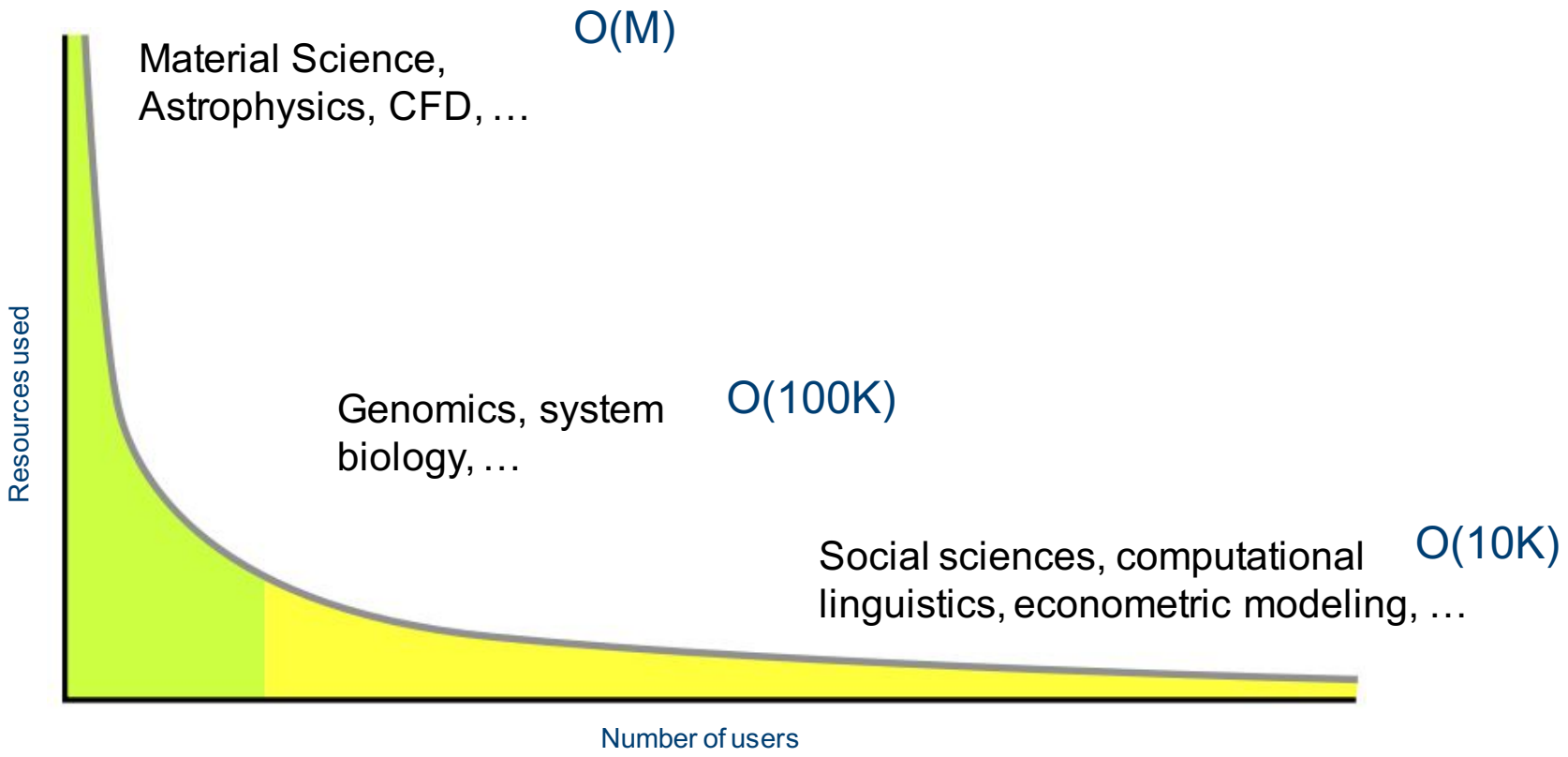
BETTER RESEARCH

Challenges in High Performance Computing

3 main challenges in High Performance Computing

- the **performance** challenge: build High Performance Computing systems (and the convergence of HPC and Big Data)
- the **programming** challenge: programming High Performance Computing systems, achieving scalability and good efficiency without sacrifice portability
- the **prediction** challenge: developing truly predictive complex application codes, explore new science and push the boundaries

Long Tail of Science



Impact *everything*: system design, scheduling policies, data policies, data management, cost models, code efficiency vs scalability, TTS vs ETS, …

Verification vs Validation

Fact: a computational simulation is only a **model** of reality. Such models may not accurately reflect the phenomena of interest.

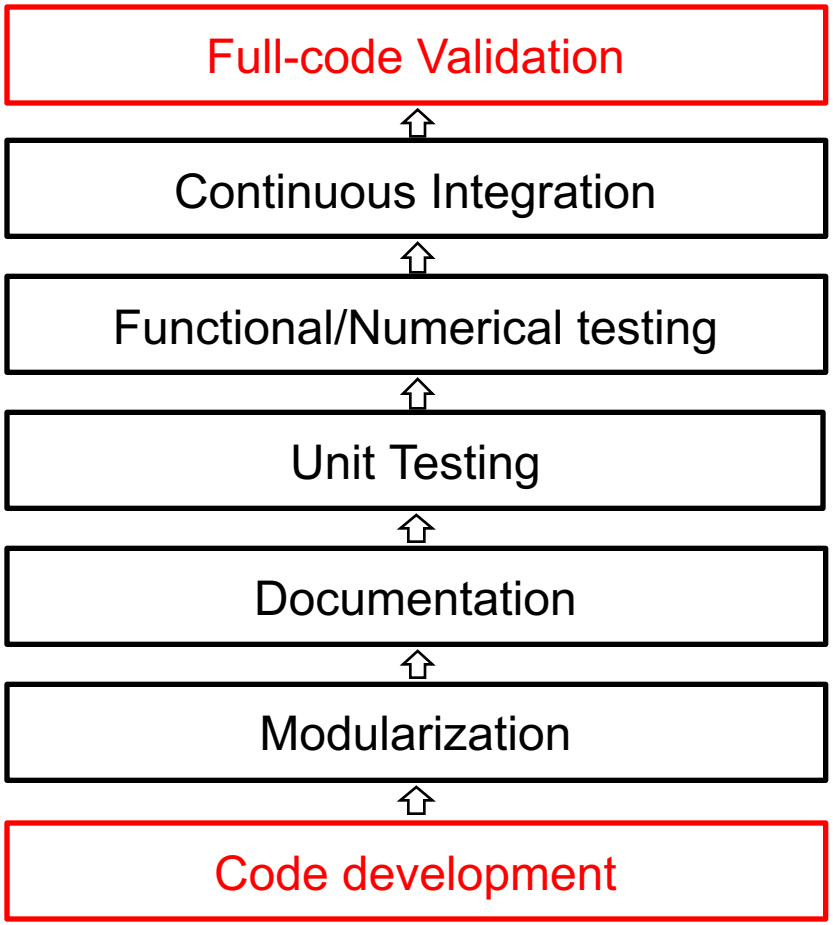
- **Verification**: determine that the code solves the chosen model correctly
- **Validation**: determine that the model itself captures the essential physical phenomena with adequate fidelity

Everybody understands that

- without adequate verification and validation, computational results are not credible
- the bigger and more complex the code, the harder it is to verify and validate
- diligence and alertness are far from a guarantee that the code is free of defects/bugs

however …

What happen in most ~~HPC~~ Research Computing communities…



Very few…

Gaining popularity after long time

Not common at all, sophistication rarely pays back

Lot of new "in house" codes lack of these

Software Testing

Software testing is an investigation conducted to provide stakeholders with information about the **quality** of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. **Test techniques include the process of executing a program or application with the intent of finding software bugs** (errors or other defects).

What a *software bug* looks like...

- The code **crashes** for some input cases
- The code returns **wrong results**
- The code **misbehave** if a particular library is used or architecture is changed
- The code generate **unreproducible results**

Software Testing tecniques

- **Static** (no execution): examination of the documentation, source code…
  - code do not need to be executed
- **Functional** (“Black Box”): based on functionality of the software
  - no knowledge of the interior workings of the application
- **Structural** (“White Box”): base on the structure of the software
  - Required knowledge the internal workings of the code.

Regression Testing

Regression testing is a type of software testing that seeks to uncover new software bugs, or regressions, in existing functional and non-functional areas of a system after changes such as enhancements, patches or configuration changes, have been made to them. The purpose of regression testing is to ensure that changes have not introduced new faults.

Regression testing can be used not only for testing the **correctness** of a program, but often also for tracking the **quality** of its output. Regression tests can be broadly categorized as functional tests or unit tests. **Functional tests exercise the complete program with various inputs.**

- A tradeoff between *Coverage* and *Complexity*
- Being *test-driven,* not *bug-driven*

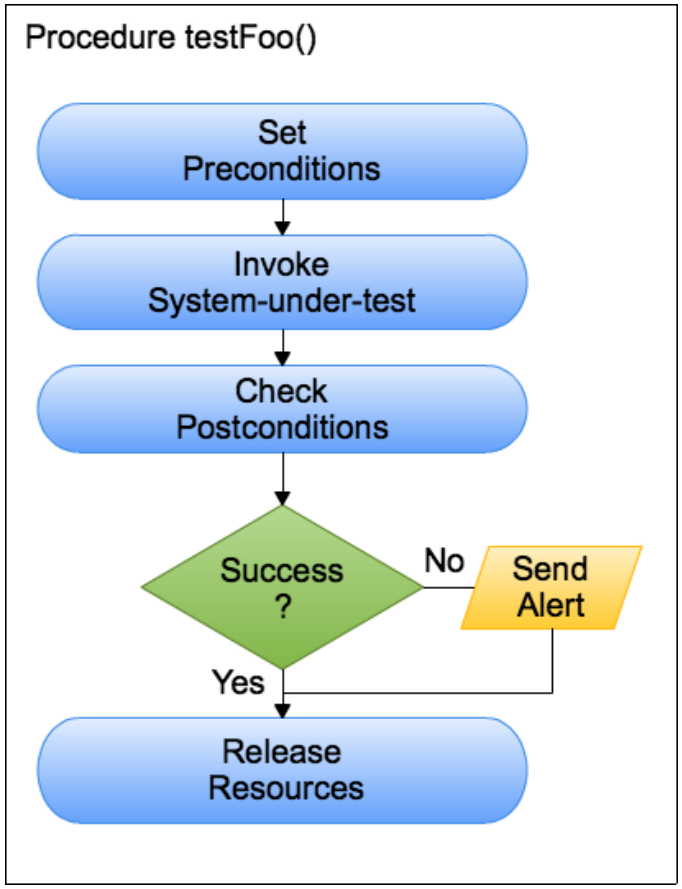
Unit-testing

Unit testing is a software testing method by which **individual units of source code**, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to **determine whether they are fit for use**.

Unit tests are short code fragments created by *programmers* or occasionally by *testers* during the development process.

- Unit tests can exercise individual functions, subroutines, or object methods. (core functionalities must be unit-tested)
- There is not universal recipe… you must know what is worth to test!
- Know the internal mechanism of the application helps…
- …but Unit Tests can be design as black-box (look at reference/trusted outputs)

Anatomy of an Unit Test



Practicing Testing: Must Have *vs* Should Have

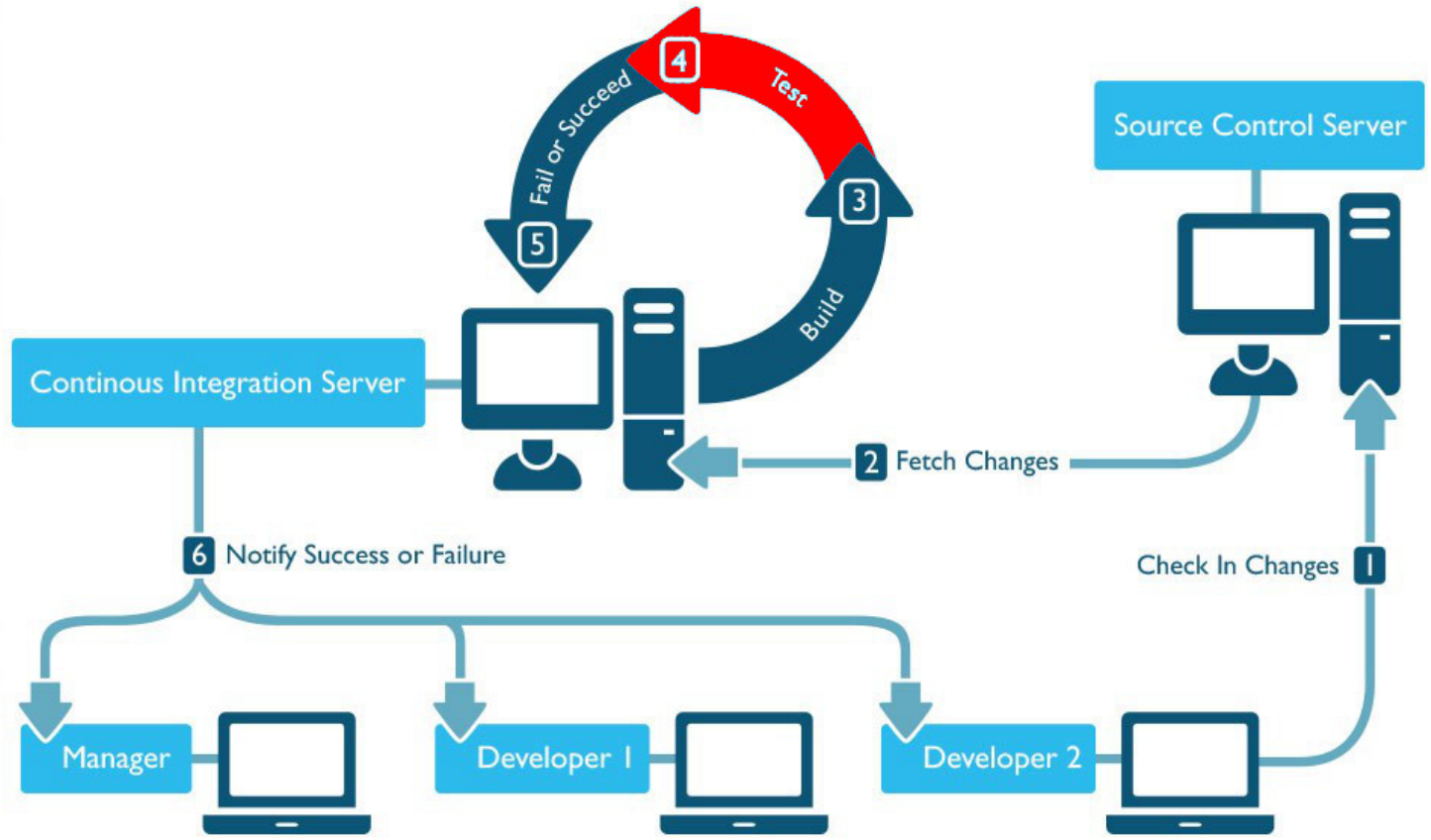
Must have

- A set of input cases that cover most of the functionalities
- A set of reference outputs (run on a set of reference architectures) to verify your results
- A set of *historical results* to *compare* variability across different versions and *ensure* continuity in correctness
- Broader view about how all the package works and components interact one to each other

Should have

- Something that run tests for you (→ *Continuous Integration*)
- Framework that compares all the possible tests for you (→ e.g. *Test-suite*)
- Routines that implement core functionalities (e.g. computing observable quantities, derivates/integrals, …) constantly under control during simulations
- Specific domain expertise in the field to be able to *set tolerances* and *validate outlier*

Continuous Integration (CI)



Continuous Integration SW

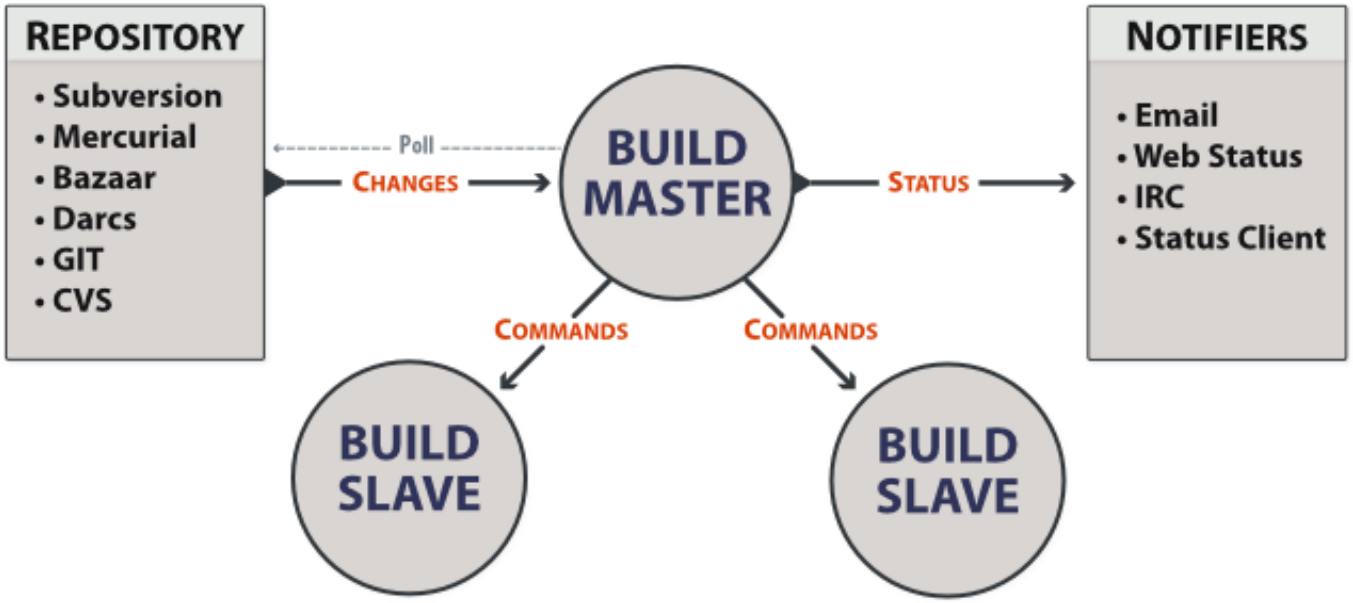
- Tens of different software package… a jungle!
- Most CI designed for enterprise software development... not scientific software development!
- There is no *standard de-facto* for CI in Research Computing
- CI is usually a “separate service”, you need to maintain and upgrade it regularly (and it can be a pain)

For an extensive list: https://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software

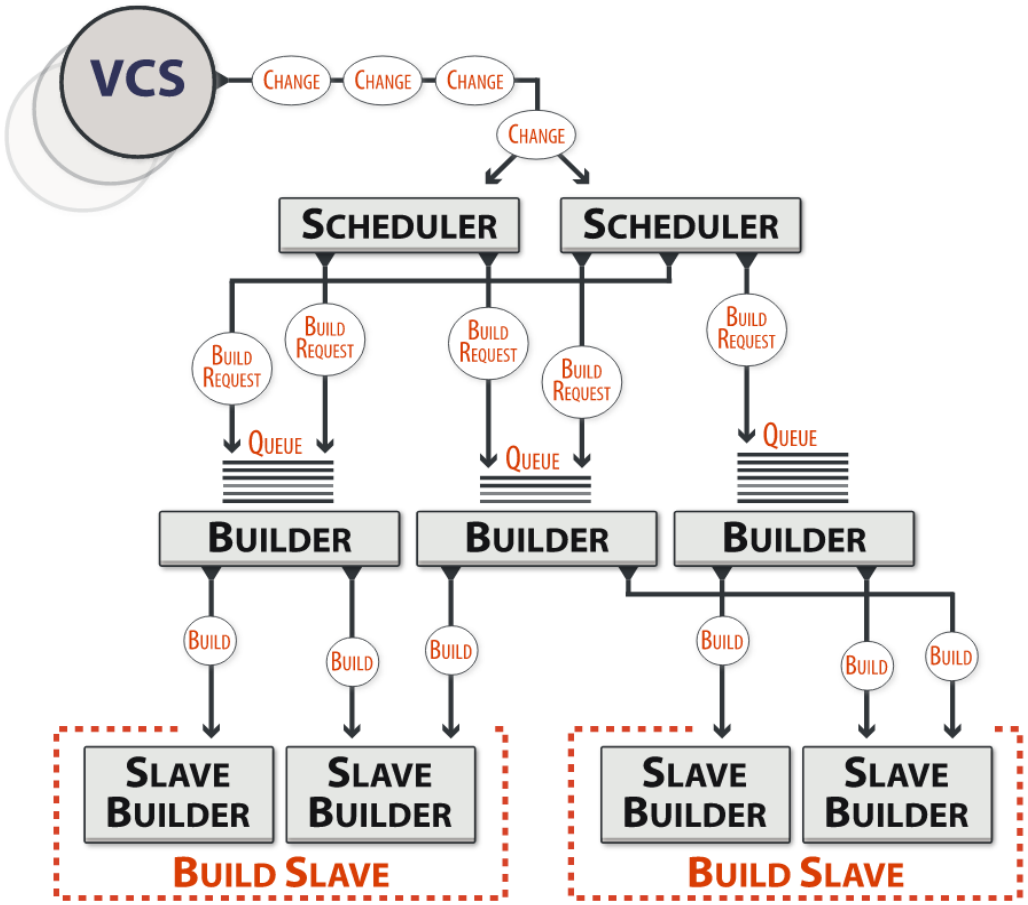
Builbot

- Buildbot is a build and test automation framework (CI).
- Very popular in many open source projects
- Configuration by editing a file
- Based on Python/Twisted
- Easy to customize

BuildBot -- architecture



Scheduling, building, notifying



What buildbot does for me…

- Upon branch update or at specific time interval, a **build** is created and test suite is run
- Developer is alerted (via UI or email) when a test fails, can submit fix, and re-launch test (even manually)
- Upload test results or compiled applications to an external server

Essential configuration elements

- List of **Slaves**
- One or multiple **Factories** (composed by list of **Steps**)
- One or multiple **Builders** (which accept lst of Factories)
- One or multiple **Schedulers** (which trigger list of Builders)

How do I deploy a BuildBot *master*?

```
mkdir -p $HOME/my_buildbot_master
cd $HOME/my_buildbot_master
virtualenv --no-site-packages buildbot_sandbox
source buildbot_sandbox/bin/activate
pip install buildbot
buildbot create-master master
edit “master/master.cfg”
buildbot start master
```

How do I deploy a BuildBot *slave*?

```
mkdir -p $HOME/my_buildbot_slave
cd $HOME/my_buildbot_slave
virtualenv --no-site-packages buildbot_sandbox
source buildbot_sandbox/bin/activate
pip install buildbot
buildslave create-slave my_slave\
            <buildbot-master-IP>:9989 my_slave <password>
buildbot start my_slave
```

Live demo…

Test-code

- Project initiated by James Spencer (ICL)
- Python module for testing for regression errors in numerical (principally scientific) software.
- It runs a set of calculations, and compares the output data to that generated by a previous calculation (which is regarded to be "correct").
- It is designed to be lightweight and highly portable.
- It can run a set of tests and check the calculated data is within a desired tolerance of results contained in previous reference output
- The programs to be tested can be run in serial and in parallel and tests can be run in either locally or submitted to a compute cluster.

https://github.com/jsspencer/testcode

Test-code, configuration files

Two configuration files:

- **jobconfig** defines the tests to run
- **userconfig** defines a program to be tested

Test-code, capabilities

- **compare** = compare set of test outputs from a previous testcode run against the benchmark outputs.
- **diff** = diff set of test outputs from a previous testcode run against the benchmark outputs.
- **make-benchmarks** = create a new set of benchmarks and update the *userconfig* file with the new benchmark id. Also runs the 'run' action.
- **recheck** = compare set of test outputs from a previous testcode run against benchmark outputs and rerun any failed tests.
- **run** = run a set of tests and compare against the benchmark outputs.
- **tidy** = remove files from previous testcode runs from the test directories.

Live demo…

Online resources

- BuildBot homepage : http://buildbot.net
- BuildBot documentation: http://docs.buildbot.net/current/manual/index.html (v0.8.12)
- BuildBot examples
  - ONETEP: http://www.cmth.ph.ic.ac.uk/buildbot/onetep/waterfall
  - ABINIT: http://buildbot.abinit.org/waterfall
  - DCO : http://www.stce.rwth-aachen.de/buildbot/dco/waterfall
- Test-code source code (GitHub): https://github.com/jsspencer/testcode
- Test-code Documentation: https://testcode.readthedocs.org/en/latest/

Best Practices (1/3)

1. **Write programs for people, not computers**
   - Make names consistent, distinctive, and meaningful.
   - Make code style and formatting consistent.
2. **Let the computer do the work**
   - Make the computer repeat tasks.
   - Use a build tool to automate workflows.
3. **Make incremental changes**
   - Work in small steps with frequent feedback and course correction.
   - Use a version control system.
   - Put everything that has been created manually in version control

Best Practices (2/3)

4. **Do not repeat yourself (or others)**
   - Every piece of data must have a single authoritative representation in the system.
   - Modularize code rather than copying and pasting.
   - Re-use code instead of rewriting it.
5. **Plan for mistakes**
   - Add assertions to programs to check their operation.
   - Use an off-the-shelf unit testing library.
   - Turn bugs into test cases.
6. **Optimize software only after it works correctly**

Best Practices (3/3)

7. **Document design and purpose, not mechanics**
   - Document interfaces and reasons, not implementations.
   - Refactor code in preference to explaining how it works.
   - Embed the documentation for a piece of software in that software.
8. **Collaborate**
   - Use pre-merge code reviews.
   - Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.
   - Use an issue tracking tool.

Closing remarks…

But everybody can make mistakes…