EPIC Code Fest

Unit Testing for Unified Forecast System (UFS)

April 4-7, 2023

Overview of Unit Testing Framework for the Unified Forecast System

Yi-Cheng Teng, Ph.D. yi-cheng.teng@tomorrow.io
Stylianos Flampouris, Ph.D. stylianos.flampouris@tomorrow.io
NOAA/EPIC, Tomorrow.io

April 4, 2023

EPIC's mission:

NOAA created the Earth Prediction Innovation Center (EPIC) to improve operational weather and climate forecast systems through scientific and technical innovation via model co-development with the Weather Enterprise — government, industry and academia.

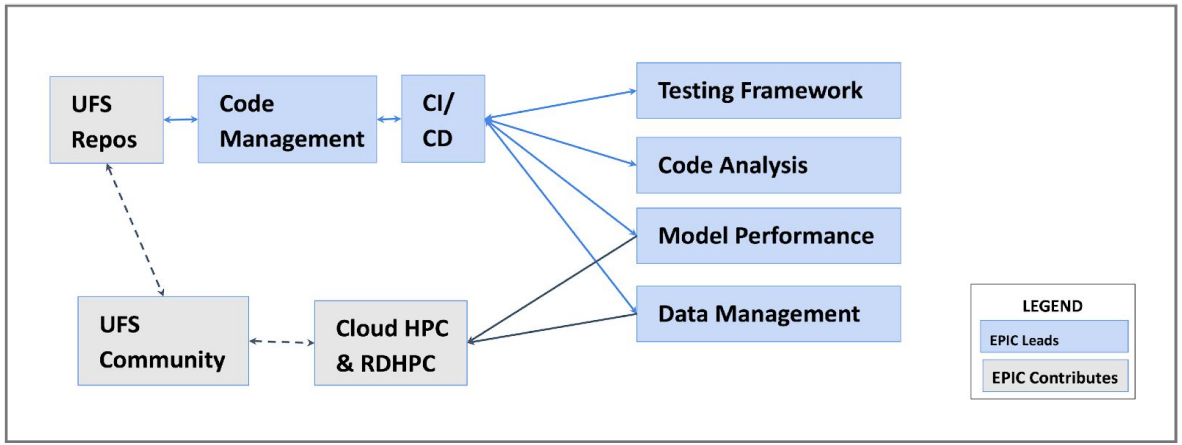


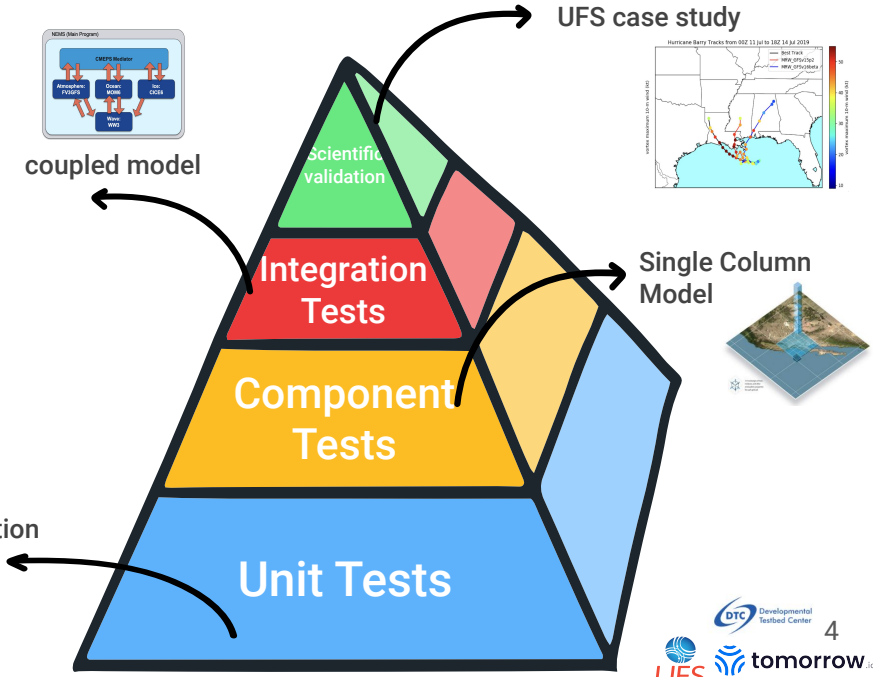
Figure 1. Simplified diagram of the main components and infrastructure developmental activities of the EPIC within the broader framework of the UFS.

EPIC Infrastructure Focuses on:

- Develop/provide necessary Software Infrastructure;
- Manage, maintain, test, and evaluate the UFS WM and Apps source code;
- Develop and maintain the appropriate frameworks;
- Support the transition of the UFS WM to Cloud-based HPC

Modern Software testing Principles

- A common infrastructure;
- Hierarchical Testing (Multi-level testing);
- Optimized testing (Reduction of cost);
- Simplification of CM;
- User-friendliness;



UFS HTF - Engineering component

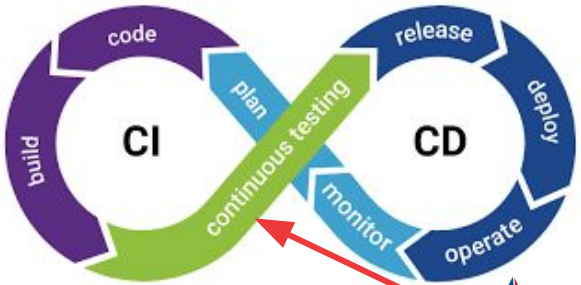
- **EPIC is developing a testing infrastructure based on Ctest and container technology**

Pros of CTest:

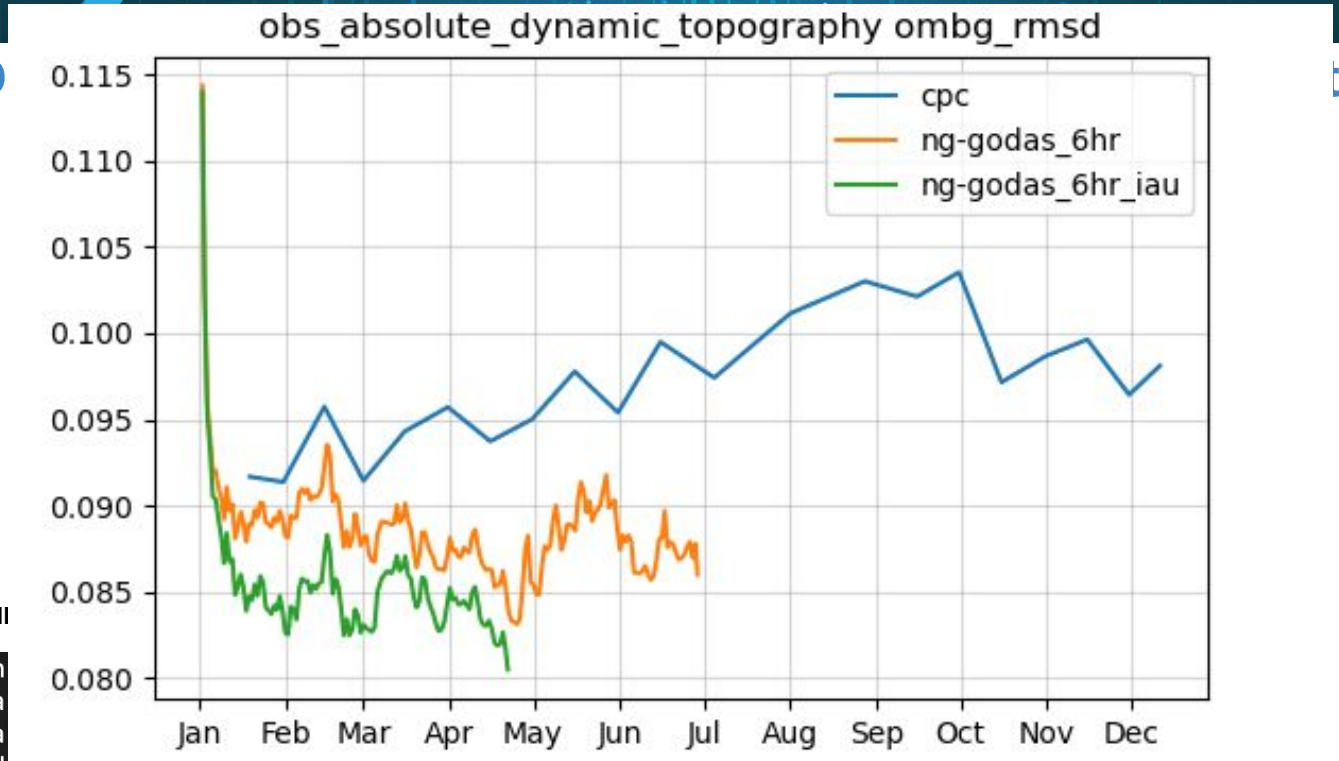
- Ctest is the CMake test driver;
- Easy to add new tests in Ctest;
- Has potential to integrate with JEDI DA Ctest framework directly.

Pros of Container:

- portability;
- No need worry about library dependencies;
- Ensure bitwise identical results, regardless of different computing systems.



DEMO: D ...tion



= 154.317682
= 956432

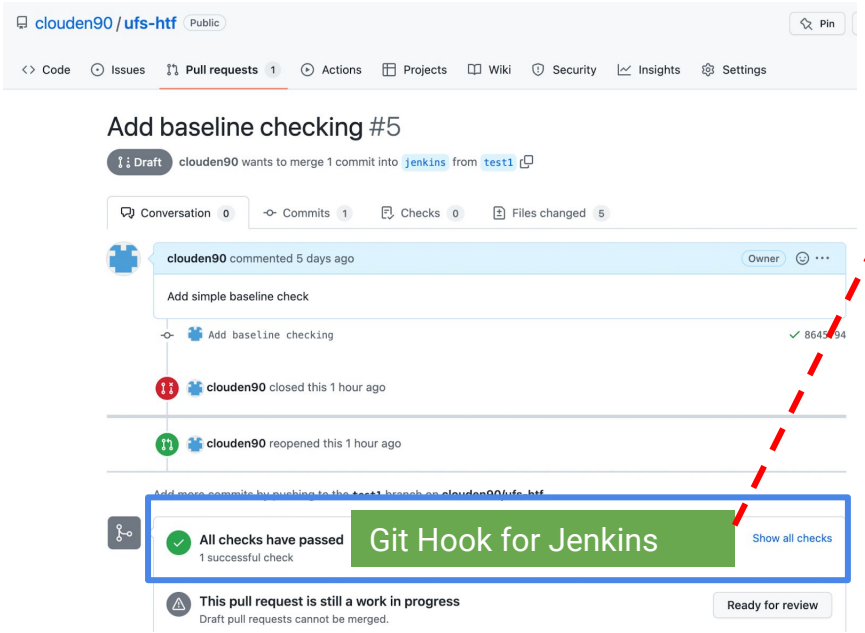
Unit Test: IAU fu...

```
1: NOTE: Run in
1: NOTE: initia
1: NOTE: initia
1: NOTE: incupd
```

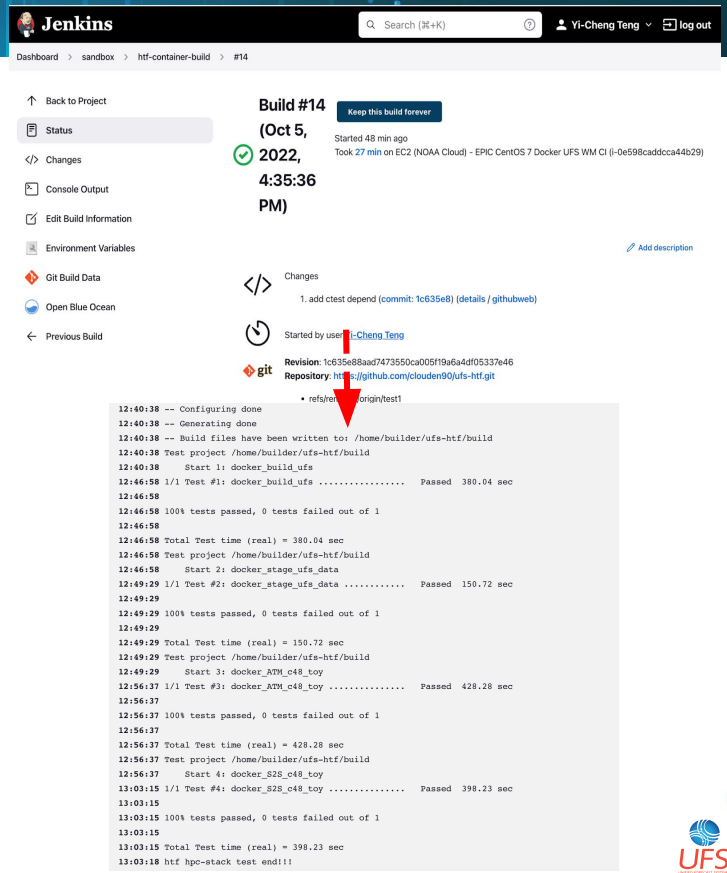
EARTH 2015-01-15 00Z to 2015-12-15 00Z

JCSDA ioda-plots

UFS HTF: integration with Jenkins CI



Git Hook for Jenkins



Summary

- In EPIC, we continue analyzing the testing needs, experimenting and evaluating potential solutions to address the immediate engineering and HPC resources issues.
- A prototype testing infrastructure based on Ctest and container technology has been developed and tested with UFS WM and SRW apps.
- We will evaluate the effectiveness of our approach by utilizing the testing framework for an end-to-end example from innovation to integration in the UFS-WM.
- At the same time, we are looking for solutions to integrate scientific testing in collaboration and support of the UFS community.

Acknowledgments

- DTC
- UFS Community
  - NOAA/NCEP/EMC
  - NOAA/GSL
  - NOAA/EPIC

Thank you!
Questions?

Contact: yi-cheng.teng@tomorrow.io
stylianos.flampouris@tomorrow.io

Setting up and Running Unit Testing

Dr. Yi-Cheng Teng

EPIC
Tomottow.io

April 4, 2023

Overview

- build Demo source code with unit testing framework
- Run unit testing
- Describe the process for setting up a unit testing
- Take Home message & Exercise!

Objective: I can use the unit testing framework to add, modify, and run unit tests.

Prerequisites - Windows System (Use container)

- disk space: ~40GB
- At least 6 CPU cores
- Installation of Docker (*)
- Start Docker (CMD)

($ = command-line prompt)

```
$ systemctl start docker
```

- In CMD, Pull Docker image (This step may take a while since the image is quite large)

```
$ docker pull ufscommunity/ufs-noahmp_landa:unit_test_demo
```

- Start Docker container with interactive shell

```
sudo docker run --platform linux/amd64 --rm -it ufscommunity/ufs-noahmp_landa:unit_test_demo
bash --login
cd /root && source /root/.bashenv
```

- git clone demo source code

```
$ git clone -b unit_test_example https://github.com/yichengt90/land-apply_jedi_incr.git
```

(*) - Software container is a virtualization approach implemented at the operating system level. Software container basics and their implementations, e.g., *Singularity* or *Docker,* however, are beyond the scope of this course. The users are encouraged to explore the basics of container approach on their own.

Prerequisites - Mac System (Use Homebrew)

($ = command-line prompt)

- disk space: ~40GB
- At least 6 CPU cores
- Use Homebrew to Install fortran compiler (e.g. gfortran), cmake, python v3.x, netcdf, nefcdf-fortran,openmpi
- git clone ecbuild

```
$ git clone https://github.com/ecmwf/ecbuild.git
$ export PATH=$PWDecbuild/bin:$PATH
```

- git clone pFUint and build

```
$ git clone https://github.com/Goddard-Fortran-Ecosystem/pFUnit.git
$ mkdir -p pFUnit/build; cd pFUnit/build; ecbuild ..; make -j2; make install
$ export PFUNIT_DIR=${YOUR PATH}/pFUnit/build/installed
```

- git clone demo source code

```
$ git clone -b unit_test_example https://github.com/yichengt90/land-apply_jedi_incr.git
```

Prerequisites - NOAA RDHPCS (Hera & Orion)

($ = command-line prompt)

- git clone demo source code

```
$ git clone -b unit_test_example https://github.com/yichengt90/land-apply_jedi_incr.git
```

- Load machine specific modules

```
$ module use land-apply_jedi_incr/modulefiles
$ module load landda_orion.intel
```

- git clone pFUint and build

```
$ git clone https://github.com/Goddard-Fortran-Ecosystem/pFUnit.git
$ mkdir -p pFUnit/build; cd pFUnit/build; ecbuild ..; make -j2; make install
$ export PFUNIT_DIR=${YOUR PATH}/pFUnit/build/installed
```

Build Unit testing Demo (# = comment, $ = command prompt)

 # Switch to source code folder

```
$ cd land-apply_jedi_incr
```

 # Now make build folder and do build process:

```
$ mkdir build && cd build
```

```
$ ecbuild .. && make -j4
```

This should not take too long to build. Once it's done, try the following command:

```
$ ctest -N
```

Run unit tests (# = comment, $ = command prompt)

The following command should list all existing unit tests in demo:

```
$ ctest -N
```

```
Test project /Users/yi-chengteng/epic/sandbox/land/example/build
  Test #1: test_jediincr_module
  Test #2: test_apply_jediincr

Total Tests: 2
```

Run unit tests (# = comment, $ = command prompt)

Use the following command to run all unit tests:

```
$ ctest --stop-on-failure
```

```
Test project /Users/yi-chengteng/epic/sandbox/land/example/build
    Start 1: test_jediincr_module
1/2 Test #1: test_jediincr_module .............   Passed    0.06 sec
    Start 2: test_apply_jediincr
2/2 Test #2: test_apply_jediincr ..............   Passed    0.80 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.87 sec
```

Run unit tests (# = comment, $ = command prompt)

Use the following command to run specific test

```
$ ctest -R test_jediincr_module
```

```
bash-5.2$ ctest -R test_jediincr_module
Test project /Users/yi-chengteng/epic/sandbox/land/land-apply_jedi_incr/build
    Start 1: test_jediincr_module
1/1 Test #1: test_jediincr_module .............   Passed    0.07 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.07 sec
```

Use the following command to run specific test with verbose output

```
$ ctest -V -R test_jediincr_module
```

How to set unit testing

Ref: [Creating and running tests with CTest]

Open land-apply_jedi_incr/test/**CMakeLists.txt** with a text editor:

add_test

- Name
- COMMAND
- WORKING DIRECTORY

```
# test for adding jedi increment
add_test(NAME test_apply_jediincr
         COMMAND ${PROJECT_SOURCE_DIR}/test/apply_jedi_incr.sh ${PROJECT_BINARY_DIR} ${PROJECT_SOURCE_DIR}
         WORKING_DIRECTORY ${PROJECT_BINARY_DIR}/test)
set_tests_properties(test_apply_jediincr
    PROPERTIES
    DEPENDS "test_jediincr_module; test_python_compare"
    ENVIRONMENT "LANDDA_INPUTS=$ENV{LANDDA_INPUTS};
                 PYTHONPATH=${PROJECT_SOURCE_DIR}/test:$ENV{PYTHONPATH};
                 TOL=$ENV{TOL}")
```

Set unit testing for Fortran

Open /opt/land-offline_workflow/test/**CMakeLists.txt** with a text editor:

add_pfunit_test

- Name
- Source code of unit test
- Link to library

```
# test fortran jedi incr module
add_pfunit_ctest(test_jediincr_module
   TEST_SOURCES test_jediincr.pf
   LINK_LIBRARIES jediincr
   )
```

```
module NoahMPdisag_module

  private

  public noahmp_type
  public UpdateAllLayers

! Note: in GFS with fractional grid cells, the swe and snow_depth saved in noahmp_type are the
!       values over the land fraction only. The restarts have a separate variable with the grid
!       cell average. All other varibales here (except temperature_soil) are Noah-MP specific,
!       and are defined only over land.

  type noahmp_type
    double precision, allocatable :: swe                (:)
    double precision, allocatable :: snow_depth         (:)
    double precision, allocatable :: active_snow_layers (:)
    double precision, allocatable :: swe_previous       (:)
    double precision, allocatable :: snow_soil_interface(:,:)
    double precision, allocatable :: temperature_snow   (:,:)
    double precision, allocatable :: snow_ice_layer     (:,:)
    double precision, allocatable :: snow_liq_layer     (:,:)
    double precision, allocatable :: temperature_soil   (:)
    character(len=10)      :: name_snow_depth
    character(len=10)      :: name_swe
  end type noahmp_type

contains

  subroutine UpdateAllLayers(vector_length, increment, noahmp)
```

Ref: [Installation of pFUnit and pFUnit demos]

[cesm unit test tutorial]

Now let us try to fail a test, open test/test_jediincr.pf and edit line 23: change 10.95 to 10.93. Then rebuild source code. Then run

```
$ ctest --stop-on-failure
```

You will find out you fail the test! Check Testing/Temporary/LastTest.log for more details

```
[test_jediincr.pf:60]
AssertRelativelyEqual failure:
           Expected: <10.930000000000000>
             Actual: <10.949999999999999>
    Rel. difference: <0.18298261665141422E-2> (greater than tolerance of 0.99999999999999995E-6)

 FAILURES!!!
Tests run: 1, Failures: 1, Errors: 0
```

Add new unit testing for Python

Ref: Python unittest

Assume we write a python script for file comparison (check python scripts under test) and want to add it to unit testing framework. Open land-apply_jedi_incr/test/**CMakeLists.txt** with a text editor and add the following lines:

```
# test python compare function using ctest

add_test(NAME test_python_compare

         COMMAND ${PROJECT_SOURCE_DIR}/test/test_compare.py -v

         WORKING_DIRECTORY ${PROJECT_BINARY_DIR}/test)
```

Add new unit testing for Python

Ref: [Python unittest]

After adding the new test. Re-build our source codes

```
$ ctest --stop-on-failure
      Test project /Users/yi-chengteng/epic/sandbox/land/example/build
          Start 1: test_jediincr_module
      1/3 Test #1: test_jediincr_module .............   Passed    0.11 sec
          Start 2: test_python_compare
      2/3 Test #2: test_python_compare ..............   Passed    0.40 sec
          Start 3: test_apply_jediincr
      3/3 Test #3: test_apply_jediincr ..............   Passed    1.09 sec

      100% tests passed, 0 tests failed out of 3

      Total Test time (real) =   1.61 sec
```

Exercise

- Our #3 test_apply_jediincr only check if the program generate outputs. It would be good to go further step to compare with baseline files so it will be a more useful test.
- Try to implement the python file comparison script to the #3 test_apply_jediincr (test/apply_jedi_incr.sh). Hit: The references(baseline files) can be found under test/testref

General guidelines for writing unit tests (FIRST)

- Fast (order milliseconds or less)
  - This means that, generally, they should not do any file i/o. Also, if you are testing a complex function, test it with a simple set of inputs - not a 10,000-element array that will require a few seconds of runtime to process.
- Independent
  - This means that test Y shouldn't depend on some global variable that was created by test X. Dependencies like this cause problems if the tests run in a different order, if one test is dropped, etc.
- Repeatable
  - This means, for example, that you shouldn't generate random numbers in your tests.
- Self-verifying
  - This means that you shouldn't write a test that writes out its answers for manual comparison. Tests should generate an automatic pass/fail result.
- Timely
  - This means that the tests should be written before the production code (Test Driven Development), or immediately afterwards - not six months later when it's time to finally merge your changes onto the trunk, and have forgotten the details of what you have written. Much of the benefit of unit tests comes from developing them alongside the production code.