National Centers for Environmental Prediction (NCEP)

Environmental Equivalence 2 (EE2) Consolidated Document

Completed Under the Supervision of the EE2 Working Group

Version 2.0

December 18, 2018

Table of Contents

I. Introduction

The Environmental Equivalence 2 (EE2) Consolidated Document outlines the technical process for the transition of numerical modeling and analysis systems from research and development to operations in the National Centers for Environmental Prediction (NCEP) Operational Production Suite. This EE2 process document is an upgrade to the original EE1 process (see appendix) and was developed between the Development Organizations (hereby referred to as DevOrg) that are responsible to provide the model upgrades (EMC, NOS, MDL etc) and NCEP Central Operations (NCO), considering input from both development and operational perspectives and personnel, as well as NCEP partners and stakeholders within the National Weather Service (NWS).

The EE2 process document has been organized along three major components of the transition to operations process:

A. Elements of Structure
B. Elements of Testing
C. Elements of Integration

These three categories address and define the process for project kick-off, implementation plan, project management update requirements (regular prescribed quad chart updates and briefings), and the technical transition process, which includes workflow, inter-model dependency considerations, Service Change Notice (SCN, formerly called Technical Information Statement) and Public Information Statement (PNS) requirements, naming conventions, etc.

The intent of this document is to streamline the implementation process through the adoption of a standardized process and clearly define roles and responsibilities expected of both development and production staff, with the goal of reducing delays in the implementation process, as well as creating an implementation process framework that will better support the direction of a unified modeling suite.

A. Elements of Structure

All portions of the production suite shall be divided into self-contained groups of code, scripts and other files, hereafter referred to as a *package*. Packages have associated modulefiles usable by the unix "module" command, that set environment variables needed to install or use the package. These packages shall exist in NCEP Code Repositories in such a way that they can be recompiled from source, and if needed, ported to other platforms. These packages shall follow a specific directory structure and versioning system as described below.
The production suite consists of two types of packages:

1. **MODELS** - a set of jobs to run in a workflow, connected to each other by input and output data, including all called scripts and executables, static (not changing from run to run) input data, and files needed to build the executables.
2. **LIBRARIES** - either a) compiled libraries (dynamic or statically linked) that are intended to be used by models or b) common utilities either installed as production modules by NCO (such as prod_util for date manipulation in scripts or grib_util for GRIB file manipulation), or by developers as needed

Each package must be exactly one of the above; a model cannot be a library and a library cannot be a model. There are two defining differences:

1. A library cannot include jobs as part of a workflow.
2. If a package needs to use any file from another package (for example, if the NAM needs to use the global_chgres binary installed in GFS), then the package's version file must set the version of the other package than is needed (see "Package Installation Area", section A.I.b for details).

A.I Modularity

Each package must exist in a single *vertical structure* with the naming convention:
<*package*>.v<*version*>

where <*version*> is defined in the Versioning section. For a model, the <*package*> is the name given for the packages' NET or RUN environment variable (see section A.IV.a). For a library, it is the library's name.

The directory structure underneath can be found in the section "Package Installation Area".

In general, each model should be able to be built, rebuilt and run without using files outside of that package, except system executables and system libraries. The only exceptions to this rule (external files that can be used) are:

1. Dynamic data, which generally changes from run to run.

2. Third-party libraries are available through modulefiles.
3. Shared code, environment variables, or scripts in library packages as defined and discussed in the Libraries section. Note : On the current NCEP supercomputer, these are available through modulefiles in $NWROOT/modulefiles and $NWROOT/lib/modulefiles (see section A.I.b below for the definition of $NWROOT)

Workflow components shall never be shared between packages. Sharing such components among multiple models has a distinct disadvantage of having to keep and support multiple versions. Therefore, only shared code in a library package that is purportedly backwards-compatible is allowed to be used in other models by simply pointing to the external vertical structure. This is so that an older version can be eliminated during an upgrade of the model using it with minor testing.

In other cases, any shared code or potentially shared code must be copied into the vertical structure of the main model before delivering the model to NCO. If a package contains code copied from another source, then the maintenance of that copy is the responsibility of the package owner or developer, not the originator of the code.

A.I.a Module Files

Each package must include modulefiles that set environment variables necessary to build, run, or link (in the case of libraries) against the system. If a package requires another package (such as g2 requiring g2tmpl to link) then that should be represented by adding a dependency in the modulefiles.

In general, a package may have one or two modulefiles:

1. A modulefile for use while installing the package
2. A modulefile for running the package's programs, using its scripts, or linking against the module's libraries.

Ideally, there should be only one modulefile that does both, but that is not always possible. Variables required in a modulefile are discussed in detail in the Libraries and Models sections.

I**MPORTANT NOTE** : For both code builds and run time modules, the implementation package MUST run "module purge" to unload any loaded modules (like those one may load automatically upon logging into the system). An example of this would be a system developer who loads ics/17.0.1 when logging into WCOSS since it is needed to run a code in their implementation package, but they do not note this in the release notes and when NCO tries to run the code it will fail because the default ics module is different.

A.I.b Package Installation Area

As previously stated, a package must exist in a directory with this name:

<*package*>.v<*version*>

The version number for a production package is set in $**NWROOT/**versions/$RUN.ver, where **NWROOT** is the root production directory (currently /nwprod on NCEP Phase 1 computer, /nwprod2 on Phase 2, /gpfs/hps/nco/ops/nwprod on the Cray, /gpfs/dell1/nco/ops/nwprod on the Dell). For the current operational NAM as of March 2017, the nam.ver file looks like this:

export grib_util_ver=v1.0.4

export util_shared_ver=v1.0.5

export nam_ver=v4.1.1

export radmon_shared_ver=v2.0.4 # used for satellite radiance monitoring job

export ics_ver=15.0.3

export netcdf_ver=4.2

Note that not only is the package version number be defined in this file, but also the version of any utilities or other packages that the NAM needs to use is set here.

Within the $NWROOT/<*package*>*.v*<*version*> directory, there may be several other subdirectories depending on the purpose of the package (library or model):

jobs/ - location of the package's top-level scripts for each system component, commonly called "j-jobs" since they start with uppercase "J" and are all upper case (such as "JNAM_FORECAST"). These "j-jobs" usually run the high-level system scripts in the "scripts" directory that do most of the work. Additional details are below.

scripts/ - high-level logic scripts, usually one per j-job

ush/ - low-level logic scripts, scripting libraries, or utility scripts

doc/ - documentation

exec/ - compiled, executable programs (not scripts; those go in ush or scripts)

sorc/ - package source code for packages with compiled components

modulefiles/ - unix module files

dictionaries/ - input dictionary files (NOTE: Only relevant for decoders)

parm/ - input text files

parm/wmo : Specific subdirectory under ./parm for WMO GRIB headers

fix/ - input binary files

ecf/ - ECflow files

gempak/ - fixed files and utility scripts needed for GEMPAK file processing. Examples include GRIB2 parameter tables, package-specific scripts/files for NAWIPS meta graphics file creation.

lib/*<name>*/*<version>*/ - compiled library *<name>* files for version *<version>*

(see Compiled Libraries for details on the meanings of *<italics>* here)

lib*<name>*<*ver*><*precision*>.<*extension*> - library for the default build target

<compiler_[OPT]>/ - subdirectories with libraries for other build targets

include/ - files needed from library at compile time

data/ - data files needed for libraries, such as during unit testing

src/ - source code for the library

NOTE : Any library that is placed inside a package should only be a package-specific library that is created and used only by that application (an NCEP operational example is the library created when HYSPLIT is built).

A.I.c J-Job Files

The jobs directory contains what are called "*j-job" files* which are the top-level, package-specific driver script in each of the tasks of a model's workflow. Since this is only for the model's workflow, a library must never have a jobs/ directory. The j-jobs files should follow this naming convention:

J*<MODEL>*_*<TASK>*

where:

<MODEL> - is the NET or RUN for the model

<TASK> - is the capitalized name of the task the j-job runs.

In the rare case of j-job files that are shared among multiple tasks (such as ensembles) this should be a name typical of or descriptive of the tasks that will use it.
(Ref: http://www.nco.ncep.noaa.gov/idsb/implementation_standards/)

A.I.d Source Code

The subdirectory structure within **sorc/** is allowed to vary. However, all source code for a given executable must be in one directory named as described below. The only exception is code

that from a compiled library linked to the executable should be in that library's source directory. All source code for executables must be in:

sorc/…subdirectories.../<*package*><*progname*>.<*language*>d/

or

sorc/...subdirectories../<progname>.<language>d/

where <*package*> is the model package name, <*progname*> is the name of the program within the exec directory, and <*language*> is "c" for C or C++ programs and "f" for Fortran programs. As with the executable naming convention discussed in the next section, since the package name is part of the higher-level directory path, having it in the source code name is optional. If the program uses a mix of various languages, the choice of "c" versus "f" is at the developer's discretion, but should be descriptive of the dominant language used in the program. If there are multiple programs built from a source directory, the directory name should be descriptive of the programs within.

A.I.e Executables

Executables within a package's directory must follow one of these naming conventions:

exec/*<package>_<progname>, exec/<progname>, or exec/<progname>.x*

where <*package*> is the name of the overall package, and <*progname*> is the name of the program within the package.

A.II Versioning

A.II.a Production or Release Versions

Production releases use the following naming convention:

<package>.vX.Y.Z

Where

- "X" is the major version number
- "Y" is the minor version number
- "Z" is the release number

Generally, the following guidelines are followed for the means by which the version numbers are incremented:

- Major version numbers (X) should be incremented only for changes that fundamentally change the nature of the model or could cause major differences in the forecast. Examples are adding a new coupled component or doubling the resolution.
- Minor version numbers (Y) should be incremented for substantive changes that should be considered by downstream users due to possible change in forecast skill, delivery time or other important considerations.
- Release numbers (Z) should be incremented for changes that do not change the intent of the system, but may fix problems that prevented the model from achieving that intent (such as bug fixes).

Any hand-off of code to NCO that has undergone a formal evaluation by the Configuration Control Board (CCB) is considered substantive and mandates, depending upon the change, either a major or minor revision number change. That is, any hand-off version must have a release number of 0.

Some examples from the last NAM upgrade:

1) Major upgrade : in August 2014, NCEP implemented NAM version 3. In September 2016, EMC was ready to handoff a major upgrade to the NAM (increases in model resolution, revamped data assimilation system), so EMC's subversion tag for code delivery was called nam.v4.0.0, which was implemented in March 2017.

2) Bug fix: soon after the NAM version 4 was implemented, a NWS forecast noted a wind direction error in an output grid that the code developer did not catch. The developer fixed the error and delivered the change to NCO, it was implemented into production as nam.v4.0.1

3) Minor upgrade : the changes implemented in the July 2017 GFS upgrade necessitated making significant changes to the generation of lateral boundary conditions for the NAM and the processing of ensemble data from the GDAS in the NAM's hybrid GSI analysis. Thus the NAM release was considered a minor upgrade and was implemented as nam.v4.1.0.

Details on version number standards can be found on this online document on the NCO web site at http://www.nco.ncep.noaa.gov/pmb/version_numbers/Version_Numbering_Standard.pdf

A.II.b Code Repository Usage

All deliverables to NCO with the exception of binary files are to be done via code repositories. NCO and the system DevOrg maintain two separate repositories and this section highlights the interaction between the two.

For developers, this is usual configuration of the implementation package in the repository :

1. Trunk : the most recent version with the system with mature changes all developers agree upon. (Note: In GIT the terminology is slightly different but the functionality will be the same)
2. Branches : primarily used by different system developers before the package frozen prior to hand-off to NCO.
3. Tags : “Snapshots” of the trunk version at various stages of development. Once the final configuration of the package is agreed upon by the developers, the changes should be committed to the trunk and a tag is made from the final version which will be delivered to NCO

Only developer branches should contain intermediate changes that are candidates for inclusion in the package but not yet fully mature.

NCO uses production or pre-production branches to archive executables and other build process intermediate output. This is dangerous as it can lead to the NCO SPA accidentally not recompiling any codes that were changed, and has led to operational failures in the past. This EE2 document recognizes the complexity of switching to a more suitable binary file archiving system, and allows such a practice to continue despite the risks involved. The build process rules, described later in this document, mitigates this risk by requiring the developer to provide a “clean build” option to eliminate all output and intermediate files created during the build process.

Code can be handed off and stored in the code repository in two directions:

1. Code given to the SPA from the code manager, and
2. Updates from the SPA given back to the code manager to allow development to track production needs.

During hand-off and pre-implementation work, code and scripts shall exist in two evolving directories (branches or trunks) in NCEP repositories: one for NCO and one for developers handing off code.

Hand-off shall be from tags made from the final development trunk version.

At a minimum the following must be done as an iterative process (loop):

1. The developers shall store the pre-implementation model in the development repository (branch or trunk) that will track the progress of that model towards implementation. These should not contain any executables or intermediate build system output.
2. For hand-off to NCO, the code manager create a tag of the final version in the developer repository:
3. Upon delivery NCO will install the developer's final version in their code repository. This is now considered the official version of the package. That branch shall be used to track progress towards implementation.

4. NCO shall hand back any significant changes to the code manager by creating a tag of that revision of the branch including the full model name and version number. The version number shall be higher than any used in the past for the same model.
5. The code manager shall take the changes, and merge them into the developer repository, making sure to remove any executables or other build system intermediate output.

A.III Installation Process and Related Documentation

The entire implementation package should be installable from the code repository tag provided by NCO to the code manager or from the code manager to NCO. There are generally seven steps to this installation process:

1. Code repository checkout.
2. Load modulefiles
3. Obtaining large binary files from other locations, such as HPSS.
4. Clean the directory tree.
5. Build the system.
6. Install executables and libraries.
7. Run unit tests to verify installation.

A.III.a Build System

The preferred methods are POSIX sh or POSIX Make, but other systems are allowed. However, those build systems can only use programs that are already installed on the production machine. In addition, any implementation package build system provided must follow these requirements:

1. CLEAN command - The build system must have a “clean” command that will delete all build system intermediate and final outputs. This includes, but is not limited to, executables, libraries, object files, automatically-generated code, Fortran module files, listing files, and anything else produced during the build process.
2. BUILD command - The build system must have a “build” command that will build all executables, libraries or other files required for execution. It must do this in a single command without manual intervention. However, it must also:
    a. Provide a way to recompile a single executable or library or small group of executables or libraries without rerunning the entire “clean” and “build” steps.
    b. Adequate exception handling should exist in the build system so that if one code fails to compile, a simple message should appear on the screen saying something like :
    Fatal error in building (name of code)
    The log file is in (location of log)

It is not necessary to abort the entire umbrella build if a code does not compile."

3. DEBUG BUILD command - Provide a “debug build” option that enables extra checks within the code or via compiler or library options to enable debugging of the package. This can be done via a separate “makefile.debug” file or adding optional debug compile options in the makefile.
4. INSTALL command - The build system must have an “install” command that is separate from the “build” command. This “install” command shall install executables, libraries, module files, and other build system output in their final operational locations. It shall also verify that all installed programs are present and perform minimal verification on contents, such as checking if the files are empty or an incorrect format.
5. TEST command - The build system must have a “test” command that runs any simple unit tests on compiled libraries and simple utility programs. If the package does not contain such libraries or utilities, then it is acceptable for the test command to do nothing.
6. Logging - all steps of the build system must log all modules, significant environment variables and build commands executed. Log files must be accessible by the community.
7. Dependencies - If the build system or running applications in the package are dependent on other packages in the production suite, this must be clearly marked in the modulefile and clearly described in documentation.
8. Modifiability - Any configuration changes required to change optimizations, or modify installation structure, must be clear, and well documented.
9. Determinism - If one runs the clean step, the build step and the install step, in that order, multiple times, the same output must be produced by each iteration of the build system (except for timestamps).

A.III.b Obtaining Binary Files

Code repositories should not be used to store large binary files such as those usually in the “fix” directory in implementation packages. Most numerical systems require large binary files as input that do not change frequently (such as very high-resolution global topography/land-use data). Such large binary files can strain the limits of any repository’s storage capacity and increase server hardware requirements. For this reason developers should not include large binary files in their repositories (ASCII files are not usually a problem). Delivery of large binary files should follow these rules:

1. When delivered to NCO, the implementation package must provide a way of obtaining the large fix files automatically in a single command. The instructions should be specified in the release notes.
2. The implementation package must provide a way of validating the large fix files, such as via a checksum. On all WCOSS platforms the command to do this is called “cksum”, which will give you the appropriate checksum for the file along with the number of bytes for that file. The purpose is to compare files after a transfer to make sure nothing got lost or corrupted. A relevant example for an implementation package is to run this check after

a large binary file is transferred from the NOAA R&D machine to WCOSS, or between the development and production WCOSS machines.

A.IV Models

A.IV.a NET and RUN

The NCEP production suite uses two strings to subcategorize models for dataflow and workflow related reasons: NET and RUN. Examples:

Package	NET	RUN
gdas	gfs	enkf, gdas
gfs	gfs	gfs
narre	rap	narre
rap	rap	rap, rap_e, rap_eh, rap_p
rtma	rtma	akrtma, gurtma, hirtma, prrtma, rtma, rtma2p5
wave_multi_1	wave	multi_1
nam	nam	nam

A.IV.b Data Locations

Production defines a number of directories with the specific purpose of exchanging data between models and to customers. These directories must exist in parallel and developmental workflows but may have different names. However, the same environment variables must be used in the top-level job in each workflow to specify directory locations. Locations in this section are described using POSIX sh style string specifications, relative to variables defined in the NCEP Implementation Standards document.

Models may only write to:

$DATAROOT/$jobid (temporary working directory)
$GESROOT/$envir/DIR
where DIR is any of $NET, $NET.$PDY, or (for models with multiple values of RUN) $RUN, $RUN.$PDY (see also below)
$COMROOT/output/$envir/today (stdin/stderr)
$COMROOT/logs/jlogfile (through pre-approved production utilities)
$COMOUT (see below)

In particular, it is never allowed to write to /dcom (except for the ingest suite) or to the installation directories in $NWROOT (including the model's own directories).

All model output intended to remain after the jobs complete must be written to the model's **$COMOUT** directory, which should default in the J-job to:

$COMROOT/$NET/$envir/$RUN.$PDY

Each model is allowed to write only to directories corresponding to its own values of NET and RUN (strict vertical structure). For example, output containing WMO headers must be written to a "wmo" sub-directory under **$COMOUT**:

$COMOUT/wmo

and all GEMPAK files generated from model output which previously was put in **$COMROOT/nawips/${RUN}** must be put into a "nawips" subdirectory under **$COMOUT**:

$COMOUT/gempak

If a production system needs data from another system (such as the NAM or RAP using GFS data for lateral boundary conditions), it must be done through the use of a variable **$COMIN*model*** defined in the J-job, for example:

COMINgdas=${COMINgdas:-$(compath.py gfs/prod/gdas)}

Note that NCO now requires the use of the above Python utility script ("compath.py") which automatically sets the location of the production system so that future changes are not necessary to the script if that production system is moved. It is obtained by loading the prod_util module on WCOSS.

Similarly, reading from other models' nwges directories is allowed through the use of **$GESIN*model*** variables. Devise alternative definitions of variables in a master file. (i.e. **dev_envir.sh.....export COMINgdas=/marine/save/...**)

All the "ROOT" variables are set in the workflow management system before calling the J-job. They must not be modified in J-jobs and must be used in J-jobs (not in any other place) to define input/output directories. It is not allowed to define these directories explicitly (without using the "ROOT" variables).

A.IV.c Output Filenames

Output in the model's $COMOUT input and output directories and other publicly visible directories must follow NCEP File Naming conventions. Any development parallels or retrospectives must follow the same naming conventions used by the corresponding operational

model, or any new naming proposed in the next operational upgrade. If the NCEP file naming conventions are unsuitable, the developer should propose updates to the NCEP Implementation Standards document.

Development workflows must use the same file names as are used in productions (avoid soft-links) or the proposed new names for the next operational upgrade.

A.V Libraries

Libraries are packages that are meant to be linked to executables at build or run time, or scripting libraries that are used by scripts at runtime. They differ from models in that they do not contain jobs that are run as part of a workflow. This section provides additional rules to ensure usability of libraries. Libraries shall follow all requirements given in the rest of this document unless stated otherwise in this section, including generating a log file to document the build.

In the context of this document, a library is:

1. **COMPILED LIBRARIES** - Compiled code that is linked into an executable either at runtime (static libraries) or at execution time (dynamic libraries). Examples are the g2 (Fortran code), g2c (C code) or esmf (C++ and Fortran code).
2. **GROUPS OF SMALL PROGRAMS** - Groups of small executables intended to be used by shell or scripting languages, such as the executable parts of the “prod_util” module.
3. **SCRIPTING LIBRARIES** - Shell or scripting files intended to be included or sourced by model automation suites. Examples are the Python “produtil” module within HWRF, and the shell parts of the “prod_util” module.
4. **ENVIRONMENT VARIABLE COLLECTIONS** - Collections of environment variables, such as the “prod_envir” module.
5. **A COMBINATION OF THE ABOVE** - Some libraries may contain more than one of the above list of types. For example, prod_util contains a number of small programs and is also a scripting library.

A library package should never contain jobs to be run as part of a workflow. If it does, it is a model, not a library.

A.V.a Library Modulefiles

Library modulefiles must set the environment variables listed below to point to the library’s vertical structure. As much as possible, library modulefiles should be standardized to enable portability. Library modulefiles are grouped into two categories, daily use modulefiles (such as the prod_util library module) and installation use modulefiles (such as GRIB/BUFR libraries).

A.V.b Compiled Libraries

Compiled libraries must provide additional environment variables with the following syntax:

<LIB_NAME>_<TYPE of files><PRECISION><ADD_ATTRIBUTES>

for instance: ‘‘SP_LIBd‘‘ or ‘‘SIGIO_INC4‘‘.

At the very least, at least one of each of these must be provided: “LIB”, “INC” and “SRC” for *<TYPE of files>*.

<LIB_NAME>	Capitalized library name (e.g. BUFR, SP, IP, etc.)
<LIB_NAME>_VER	The version of the library consistent consistent with the release or pre-release number, as described in the Versioning section.
<TYPE of files>	“SRC” - location directory of source code “INC” - location directory of files needed at compile time, such as Fortran module files and C or C++ header files. “LIB” - location of file required at linking time “LNK” - linker flags string
<PRECISION>	Letter or number indicating precision used to compile, if relevant: 4 - 32-bit integers and reals 8 - 64-bit integers and reals d - 64-bit integers and 32-bit reals
<ADD_ATTRIBUTES>	Additional attributes about the library such as library-specific information or future additions to this naming convention.

A.V.c Groups of Small Programs

To specify program locations, these libraries should do at least one of the following:

1. Prepend an executable directory to the standard $PATH variable, if absolutely necessary, or
2. Set a list of environment variables that describe each executable’s location independently. (For example, $WGRIB is the location of the “wgrib” program, whereas “$MPISERIAL” tells where the “mpiserial” program resides.)

In addition, any other environment changes required to use the programs should be applied or required by modulefile dependencies.

A.V.d Scripting Libraries

Scripting libraries should follow the general module scripting requirements. Hence,

$USH<name> - location of utility scripts and scripting libraries for module <*name*>, which should point to a directory with the name "ush"

$EX<name> - location of high-level logic scripts for package <*name*> which should point to a directory with the name "scripts"

Language-specific variables - some languages, such as Perl or Python, have language-specific environment variables to set "search paths" to search for library components. If this is relevant to the library in question, it is acceptable to set such environment variables.

There should never be a JOB<*name*> directory since the j-job level scripts are intended to run a job, something a library should never do.

A.V.e Environment Variable Libraries

Libraries of environment variables only need a modulefile and documentation. An example of such a library is the NCEP prod_envir module, which points to locations of common directories.

A.V.f Third Party Libraries

Third-party libraries are libraries that are not developed by either NCEP Central Operations, the vendor of the NCEP supercomputer, or the production system developers. Common examples are the NetCDF data storage library and the Portable Network Graphics library (libpng).

If possible, the third-party libraries should be installed and maintained by the vendor or support staff of the NOAA computers in question. However, if vendor and system administrators are either unable or unwilling to support the library, the developer may negotiate another option with NCEP Central Operations. If, in the judgement of NCEP Central Operations, the developer of a module is able to support the third party library, then that library may be bundled with the module. For example, one may have a custom LAPACK implementation embedded within an ocean model if that is approved.

The approval process is not described here since it is outside the scope of the Environmental Equivalence standard.

Limitations on Third-Party Libraries are

1. **ACTIVELY DEVELOPED AND SUPPORTED** - Third-party libraries must be actively developed and supported by the responsible development organization. “Dead” projects that have no developers are not allowed, unless the developer transitioning the library to operations is willing and able to take over development and support of the library.
2. **SECURE** - Due to requirements outside the scope of this document, any third-party library deemed by NOAA Security to be insecure cannot be used in operations. For that reason, such third-party libraries shall not be used in any development, until the security issue is resolved.

Naming, versioning, and installation conventions may follow the third-party libraries own conventions if that is deemed necessary for installation or maintenance. However,

1. Libraries with compiled components must have a build system that meets requirements in the Build System section.
2. The third-party library must still have a modulefile.
3. The modulefile must still follow the Modulefile and Environment Requirements, except compiled libraries can follow their own naming conventions (e.g. libnetcdf.a)

This ensures the installation of the library can be redone by another user, vetted by others or ported to another machine.

B. Elements of Testing

B.I Test Plan

The following list of test runs is mandatory before code delivery. The same set of the tests will also be run by NCO.

1) Cold/restart test
   a) Provide the detailed instruction on how to cold start
   b) Optimize the best frequency to write out restart files
   c) Restart the model and check the reproducibility
2) Special case test (e.g. a hurricane model forecast storm crossing the International Dateline or the Greenwich Meridian, or trying to insert an erroneous tropical system into GFS/NAM via the tropical cyclone relocation algorithm in winter)
3) The developer must address all Bugzilla entries, and explain to NCO any entries that cannot be addressed. Both the changes made to resolve Bugzilla tickets and/or the reasons why they could not be addressed should be documented in NCO’s online Bugzilla database
4) Scalability test. The most expensive job, such as the forecast job or the analysis job, needs to be run with different computer resources. The run times for each run need to be recorded and provided to NCO. Suggested test runs are as follows:

a) proposed nodes
b) double proposed nodes
c) half of the proposed resource

5) Reproducibility test
   a) Run the same job twice with the same computer configuration, and check if the results are bitwise identical
   b) Run the same job with different computer configuration, and check if the results are bitwise identical
6) Code stability and memory leak check
   a) Avoid use of large stack arrays. Deallocate head array after they are not needed
   b) NCO suggests to check memory leak by running “valgrind –leak-check=yes myexec”
   c) Compile the source code with “-check bounds” option to ensure no out-of-bound arrays in the source code (the actual option may be different depending upon the compiler)
7) Archiving job test. Make sure all output files are archived in the correct HPSS directories

The retrospective tests (as agreed upon by the developers and the stakeholders) will be carried out by developers and agreed upon metrics shall be used for the scientific evaluation.

B.II Testing and Implementation Timeline

The narrative that follows describes the entire implementation process in chronological order; it is depicted graphically in the flowchart in Figure 1, while Table 1 lists the implementation milestones depicted in the flowchart with an approximate timeline.

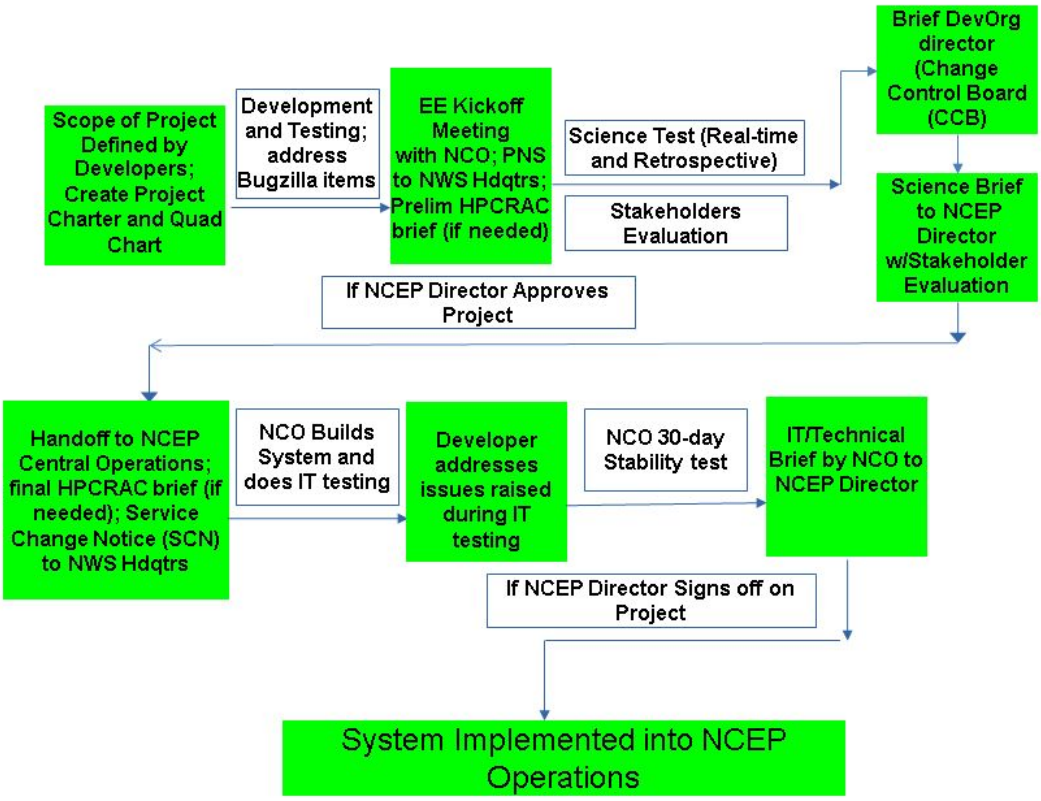

Figure 1: NCEP Implementation Process : Flowchart

From T_{imp} – 18 months to T_{imp} -6 months: – Developers define scope, create Quad Chart and Project Charter – Developers run experimental testing – Developers begin to address NCO Bugzilla tickets in new system
By T_{imp} – 6 months: – EE Coordination Meeting with NCO – Public Information Statement (PNS) to NWS Headquarters for comments – Preliminary HPC Resource Allocation Committee (HPCRAC) briefing (if needed)
At T_{imp} - 6 months to T_{imp} - 4 months – Developers finalize changes and begin final test for field evaluation – Developers contact downstream users and provide them with the data necessary to test their system; downstream developers determine if they need to make any changes to ensure their ops application will run after the target system is implemented.
By T_{imp} – 3 months: – Evaluation by stakeholders ends – EMC only : Developers hand off package for EIB internal review (EE2 compliance) and address any issues found – Developers collect evaluations from stakeholders – Change Control Board (CCB) Meeting with EMC Director (EMC only) – Science Briefing to NCEP Director (all subsequent milestones assume NCEP Director approval for implementation) – At CCB and NCEP OD briefing, get approval for any changes to product delivery times – Final HPCRAC Briefing (if needed) – Developers complete resolution of Bugzilla tickets – Handoff to NCO – Any downstream system changes must be delivered to NCO by this time – Service Change Notice (SCN) to NWS Headquarters for dissemination
From T_{imp} - 13 weeks to T_{imp} - 5 weeks: – NCO builds package, runs IT testing – Developers address any issues found during IT testing. – NCO determines if product delivery times are within 5 minutes of current ops system. If they are > 5 minutes NCEP Director approval is required
At T_{imp} –5 weeks: - NCO starts 30-day IT stability test
At T_{imp} –1 week - IT Briefing by NCO to NCEP Director for final approval
T_{imp} : Operational Implementation

Table 1: NCEP Implementation Process : Timeline

At 6-18 months prior to implementation:

When a upgrade to an existing system or a new system is decided upon by the DevOrg, the first step is for the primary system developer to write a Project Charter which describes the modeling system, reasons for the scope of the planned changes, expected benefits, any risks associated with the project, the test plan, and funding sources. The charter should be written 6-18 months in advance. The NCO standard template for a Project Charter can be found here, and an example of a completed charter can be viewed here.

Based on the information in the Project Charter a Quad chart is created for presentation at the Quarterly Technical Briefing to the NCEP and NCO Directors. This one page Quad chart consists of the following information extracted from the Project Charter:

1. The project leads (typically the code/system manager, their superior, and the head of the NCO Senior Production Analyst (SPA) team).
2. The Project Scope (planned changes) and expected benefits from the changes.
3. Any issues or risks involved in the planned implementation and planned mitigation.
4. The project schedule and estimated dates of completion, from initial coordination with the SPA team to implementation.
5. Computing resource requirements and any funding sources for the project.

The 4 elements of the Quad chart (Project Information, Issues/Risks, Schedule, and Resources) are assigned a color-coded status on the Quad chart: Green (on target), Yellow (Potential Management Attention needed) or Red (Management Attention Required).

The standard NCO template for quad charts can be found at
https://docs.google.com/presentation/d/1iJPitFqDVKvf7y7PFu26dZ_nqEyZtG6DLroq9dazmFY

Once the scope of a project is decided experimental testing begins, ideally with both real-time and retrospective testing. During this period it is expected that developers will start to address NCO Bugzilla items that were not resolved from the previous implementation.

At 6 months prior to implementation:

After experimental testing has reached an advanced enough stage, the first interaction with NCO on the project is established via the initial coordination Environmental Equivalence (EE) or “kickoff” meeting. It is expected that by the time of the EE meeting, development testing of the system should have reached an advanced enough stage for specific resource details to be known. For new systems an existing system upgrade that requires a >=3x increase in computing resources, approval must be obtained from the NCEP High Performance Computing Resource Allocation Committee (HPCRAC), which in EMC is coordinated through the

Engineering and Implementation Branch head. The brief to HPCRAC should be done as early as possible but not later than the time of the EE Kickoff meeting with NCO.

Attending the EE kickoff meeting will be the project development team, NCO SPA team, Dataflow team and (if applicable) a member of the EMC Engineering and Implementation Branch[1] who has been assigned to the project team. In this meeting the developers present to NCO the following information:

1. A brief overview of the new project or the upgrades planned for an existing production system.
2. For new systems, the expected computing resources to be needed; for existing systems, the changes in resources needed compared to the current production system. The resource information provided should include:
    a. Node usage and expected run end-to-end time on the production machine
    b. Total disk space usage per day or per cycle required on the production machine
    c. Total disk space usage per day or per cycle on the operational NCEP FTP/NOMADS server
    d. Any changes to model products that are processed for distribution to customers (inc. AWIPS).
    e. Anticipated changes to output grids (either in GRIB2 or GEMPAK format).

The NCO team at the EE meeting will list all outstanding Bugzilla tickets for the modeling system, with the developers providing information on the extent to which the planned upgrade addresses these issues. All Bugzilla items need to be either addressed by developers by the time of the handoff of the system to NCO. All actions done by developers on Bugzilla tickets (either resolving them or reasons why they could not be addressed) must be documented in the the online Bugzilla database. The development organization is encouraged to discuss code conformity issues with SPAs well in advance of the code hand-off date, including giving NCO an early look at the codes so the SPAs can comment on adherence to standards.

Developers have traditionally been responsible for the major components of the system (forecast model, analysis, post-processing). The difficulty has been in identifying responsibility for needed changes to downstream jobs that are not normally part of development testing, such as

1. AWIPS jobs (GRIB headers, needed script modifications)
2. NAWIPS/GEMPAK grid processing
3. Text products (FOUS bulletins, flight-level winds)
4. Legacy graphical products (such as FAX charts)

[1] Note: this will only apply to the implementations that EMC is responsible for. Other DevOrg's will decide what their representation at this meeting will be.

It is the Principal Investigator's (hereby referred to as the PI and identified as the person from the development organization leading the implementation system) responsibility to identify all the tasks that need to be completed before an end to end system can be delivered. If unsure the PI is encouraged to reach out to NCO to ensure that all tasks are identified. The PI (with NCO's help) will determine which groups need to be involved in the development and testing for each task no later than the EE meeting.

On or around the time of the EE kickoff meeting, a Public Information Statement (PNS) written by the developers and sent to NWS Headquarters for dissemination per National Weather Service Instruction 10-102 for comments from stakeholders. The 30-day comment period described in Section B.II.a applies and must be completed prior to the science briefing to the NCEP Director.

4-6 months prior to implementation :

After the EE meeting, major development is ended and the final version is frozen for the science evaluation test of the system, which is run by the developers. Ideally this test should including both real-time and retrospective forecasts which are to be evaluated by stakeholders inside and outside of NWS. If multiple full season retrospectives are not possible due to resource constraints, stakeholders should be given every opportunity to request specific cases of interest to be rerun. Prior to the start of the science evaluation, the developers write a Request for Evaluation letter for distribution to stakeholders. The evaluation letter describes details of the system changes, what impact these changes should have on analysis / forecast performance. Those who agree to evaluate the package are then notified by the developers when the evaluation starts and ends. The length of the science evaluation period is at least 30 days, but may be considerably longer for major high-profile system upgrades (like the replacement of the NCEP Spectral Model with the FV3 model in the GFS).

At some point during this time developers should contact the points of contact for all operational downstream users of their system and provide them with the parallel data necessary to test their system. NCO can provide these names if needed. This should be done as early as possible after the system is frozen so any downstream applications can be modified and these changes delivered to NCO at the same time as the target system is handed off. NCO maintains a spreadsheet of model dependencies. In this spreadsheet each row lists the systems that the upstream application uses, each column lists the applications that run downstream of the system.

Once the science test is completed and evaluations are collected, the Change Control Board (CCB) meeting for the project will be held, which is essentially a briefing on the project to the responsible DevOrg Director (such as the EMC Director) and other DevOrg managers. At this meeting the project PI will brief the DevOrg Director on the project with an emphasis on the scientific results and the system evaluation by stakeholders during the science test. If the DevOrg Director signs off on the project, the immediate next step is to give the Science briefing to the NCEP Director, during which the PI gives an overview of the planned changes, and the stakeholders discuss their evaluations. If the NCEP Director approves the planned science

changes for implementation, the PI finalizes the Service Change Notice (SCN) and sends it to NWS Headquarters for dissemination, and then the package is handed off to NCO. At the CCB and NCEP Director Briefings, developers must get approval from the DevOrg and NCEP Directors for any changes in product delivery times. If during their IT testing (see next section) if NCO determines that product delivery times are > 5 minutes from the current operational systems, NCEP Director approval is required for them to proceed with the implementation.

As part of the system hand off to NCO, the primary developer will do the following:

1. Create a tag on the developer's repository containing the complete system package except for large binary files. The handoff of large binary files should be coordinated with the NCO SPA assigned to the project but they should not be placed on the repository's server
2. Write the release notes for the new/upgraded system. The release notes give detailed information on resource requirements for various system components.
3. Fill out the code delivery form, which lists the location of the tag, release notes, location of additional files (e.g. binary files that are not part of the repository) and job control flow charts.
4. Prior to Science Briefing, developers should ensure that EE2 standards as laid out in the document are evaluated.

It should be noted that not all implementations have science changes. Some implementations are only technical in nature and do not change results. These implementations do not require a detailed scientific review that needs to be cleared by the NCEP Director. However, the decision to skip the science review will be made by the NCEP Director's office. Without an electronic agreement from the Director all implementations will be subject to a rigorous science review.

From 13 weeks to 5 weeks prior to the implementation:

Once the system is handed off to NCO, the SPA team will examine the package to see if it conforms to WCOSS Implementation Standards and fill out the implementation checklist. They will perform IT testing, the scope of which will vary based on the complexity of the system. This IT testing will usually include both source code checks (warning messages during code compiles, arrays out-of-bounds, memory leaks) and the impact of missing upstream data on the system. After the IT testing NCO informs the developer of any issues found that should be addressed. If NCO deems the package acceptable it will proceed with setup of the parallel production system. If there are sufficient deficiencies in the system, NCO will send it back to the developer with instructions on what issues need to be addressed.

During the setup of the parallel production system the developer will work closely with the SPA(s) assigned to the project. If the SPA needs to make any changes to the system version provided by the developer, the developer should immediately update their tag to keep the development and production systems in sync.

During IT testing NCO determines whether product delivery times are within 5 minutes of the current operational version of the system. If delivery times are > 5 minutes later with the new version then the NCEP Director needs to be informed for approval for the implementation to proceed.

At 5 weeks prior to implementation :

Once the package is fully compliant with NCO Implementation standards and is set up to run in its production configuration, NCO will run a 30-day stability test. If there are any problems during this 30-day test (system code failures, system bugs, issues with downstream products and downstream modeling systems) these problems are addressed by the developer or the SPA, and the 30-day test is restarted. If NCO determines that product delivery times will be > 5 minutes later than the current ops system then the NCEP Director needs to be informed for approval for the implementation to proceed.

At 1 week prior to implementation:

After a successful 30-day stability test, NCO gives the Technical Briefing on the system to the NCEP Director, whose approval will allow the system to be implemented into operations about 1 week later.

B.II.a : Special Instructions for Product Changes on the NWS Satellite Broadcast Network (SBN) for AWIPS and the NCEP NOMADS/FTP server

If any products distributed on the NWS Satellite Broadcast Network (SBN) or the NCEP FTP/NOMADS server are to be shut off as part of an implementation, this must be communicated to NCO as soon as possible, preferably at the EE/kickoff meeting. NCO's Dataflow group will work with the developers to document what products are candidates for termination, which will be described in a Public Information Statement (PNS). This Notice is disseminated to all known users of the package’s products well in advance of the target implementation date. Users will have a fixed period of time (usually 30 days) to comment. The comments will be evaluated and then the decision will be made if some or all of the removals will proceed. If a developer wishes to add/modify existing model products for AWIPS on the SBN, then inform NCO Dataflow at the EE2 meeting what is being added. They will begin the coordination with AWIPS, including facilitating providing sample data to the AWIPS Program office for testing.

If new products are to be added to the NWS SBN, contact the NCO Dataflow Team Lead as soon as possible, as a formal process requiring outside approval within NWS must be followed. In summary, to add products to the SBN, model developers, working with the NCO Dataflow team, must fill out the “SREC Task Submission Form” at https://vlab.ncep.noaa.gov/group/awips-community/srec (Go to the "SREC Task Submission Procedure" link). Once a month, requests for additions to the SBN are prioritized and the SREC Committee votes on what gets the highest priority. On average this process takes 5-6 months.

Removal and/or changes of model output will require notice to be given at least 30 days in advance via the Service Change Notice (SCN). Developers should refer to NWS Directive 10-1805 for the specific time limit. Directives are available at www.nws.noaa.gov/directives

B.II.b : Accelerated Implementations

Circumstances will arise that will necessitate an emergency change to a production system. NCEP Central Operations designates these as “Accelerated Implementations”. It is usually in response to a crisis situation, such as:

1) An operational system failure
2) A change to an upstream product (such as a format change to an observational database) which if not accounted for could lead to system failures and/or degraded products.
3) A directive from higher NCEP/NWS management to implement into production certain changes or items (like a new observation instrument) as soon as possible.

In the event of a production system failure, NCO will usually rerun the failed job to see if the failure is reproducible. If the rerun fails at the same point and attempts to get a successful run do not work, the primary system code manager will be contacted. If the system developers are contacted, they will examine the nature of the failure and either provide a fix or suggest a workaround which will allow the failed production run to finish. The NCO SPA's will save the working directory for the failed run so the system code manager can debug the failure and provide a permanent fix if one was not provided initially.

In the event of an accelerated implementation not associated with a production failure, the system code manager and/or their supervisor will contact the head of NCO's Implementation and Data Services Branch and the NCO SPA Team Lead as soon as possible. For the Code Delivery Form, the primary developer will prepare a repository tag or branch with the required changes and release notes describing why the change is needed and where the repository tag/branch is located. If advance notice to the field needs to be provided, the system code manager or their supervisor will prepare a draft Service Change Notice for NCO Dataflow to review.

B.II.c : Downstream Systems

Output from most major NCEP modeling systems (such as the GFS) is used by many downstream production jobs. These downstream systems may require changes to be implemented simultaneously with the upstream system so they will still work. Code managers of downstream systems should run an end-to-end test with data from the modified upstream system as soon as it becomes available, preferably at least 1-2 months prior to code delivery.

The following criteria must be followed for delivery of these downstream system changes to NCO:

1) They must be only what is minimally necessary for the system to work with the upgraded upstream package. If this criteria is followed, the downstream changes are not subject to the full upgrade process, including rigorous NCO standards and EE2 requirements. Specifically, downstream developers should not try to resolve any Bugzilla tickets as NCO's main focus will be on the primary upstream system implementation.
2) If the delivered downstream changes are more than what is minimally required, it will be subject to the full upgrade process unless otherwise mutually agreed upon by NCO / dev org.

B.III Exception Handling

Exception handling is the process of responding to the occurrence, during computation, of exceptions – anomalous or exceptional conditions requiring special processing – often changing or terminating the normal flow of a job script or a program. During the development of systems for NCEP production, the following tests should be carried out and procedures be followed to handle exceptional and abnormal model failure.

1) Data dependency tests (missing upstream data):

    a) If essential input files are missing or incomplete, the job script should check for the existence of the file and abort (with a "FATAL ERROR" log message) if these files are missing, rather than have the executable abort with a non-descriptive segmentation fault. If the abort does occur in the executable due to a missing input file a descriptive "FATAL ERROR" message should be written.
    b) If missing input files would not lead to a program abort but would degrade the system (such as no available observations for an objective analysis), a clearly defined log message prefaced with "WARNING" should be written to alert developers and NCO personnel.
    c) Downstream data requirement test: if a system's output format or directory paths are changed, all relevant downstream jobs need to be tested before

implementation. These tests should be done once the pre-implementation parallel is running and at least 1-2 months before the system is delivered to NCO.

2) Recovery from code failure or abnormal system failure : Restart capability must be applied to an operational job to save time when recovering from a failure. Long running jobs that have multiple executable calls might be a good candidate to break into two smaller jobs so that if a failure occurs, only the part with the problem needs to be rerun, thus the time to completion is shorter. An example of this would be to submit a separate post-processing job for each forecast hour, so any failure for one forecast hour does not impact others, and can be recovered from quickly. Any job that runs longer than 15 minutes is required to have restart capability built in such that the process picks up where it left off when rerun. For a forecast job, this would involve writing out restart files at fixed intervals during the forecast, from which the model can be restarted. The job scripts must be designed so this restart will happen automatically if the job is rerun. Any products delivered by a restarted production job must not be delayed by more than 15 minutes.

All jobs for the NCEP production suite should use the utilities provided by loading the “prod_util” module for error checking for both code failures and missing essential input, as described in the NCO Implementation Standards document.

For each system, the name of the primary developer and a backup person(s) must be provided to NCO to contact if an emergency system failure occurs and NCO is not able to resolve the failure by themselves. The primary developer or backup (as well as higher-level management) shall be reachable by NCO by both email and phone calls during evening hours and holidays. If a system fails and NCO puts in a fix/workaround, they should document the nature of the failure and the procedure that is used to fix the failure. The document shall be shared with the developers.

If the primary developer is unavailable, both their managers and NCO should be notified so that the backup developers are contacted in case of a system failure.

C. Elements of Integration

C.I Common Workflow Manager

The section on “Elements of Structure” lays out the modeling system framework. During the EE1 process, considerable unification between the developer and NCO layout was achieved. However the two centers still use different workflow systems. NCO exclusively utilizes ECMWF’s ecFlow for their workflow needs while EMC, for example, uses a mix of Unix scripts triggered by cronjobs, ecFlow, or Rocoto.

In order to achieve Environmental Equivalency it is essential that the same ecFlow workflow manager be used by both developers and NCO. This requires setting up an ecFlow development server and providing regular training. In certain situations (e.g. retrospective testing) it may not be feasible to use ecFlow as a workflow manager. In these cases, it is acceptable to use alternative approaches. However, in the last stage of testing before code handoff, the parallel tests should run using the ecFlow workflow system. This will ensure minimum delay in transitioning from development to operations. NCO will begin providing training for ecFlow, once that is completed all users will have access to ecFlow.

C.II Production Dependencies

Model dependencies need to be quantified to provide NCO so that they can adequately assess the impact of actions. If dependencies and changes are not properly tracked and communicated, downstream models could break (worst case) or their performance could change without the developer's knowledge (e.g. due to changes in upstream model physics).

Model dependencies are of two types

1. Upstream dependencies: These include other models, software packages and libraries, as well as data, e.g. satellite data from NESDIS
2. Downstream dependencies: These include models dependent on driving system output, the MAG, National Digital Guidance Database (NDGD) and other downstream graphical displays and NWS supported dissemination servers (e.g.: NOMADS).

It is acknowledged that for a particular modeling system, the upstream dependencies are easier to document than the downstream dependencies. Nevertheless a good faith estimate must be made to document the dependencies as far as possible. Such information must be updated at the EE kick-off meetings approximately three months before code delivery. It is also recommended that an expanded NCO changes page capture the EE presentation and other details about upcoming upgrades at least one quarter in advance. This page can be accessed at http://www.nco.ncep.noaa.gov/pmb/changes/

NCO has also developed a tool to track inter-model dependencies. This tool can be accessed at: http://www2.nco.ncep.noaa.gov/pmb/spatools/modeldeps/

C.II.a. Full Production Suite Test

Full production suite tests are necessary for the models with a large number of downstream dependencies. The “large number of dependencies” determination is subjective but typically driven by the amount of available resources on the production system. If NCO can't run a long term parallel of the upgraded model plus all downstream dependencies then a full production suite test is required. This includes major observation processing upgrades (dump/prep and

satellite ingest) and major upgrades, as defined in NCO version standards, to production systems with many downstream dependencies (such as the GFS, GEFS, or NAM). Others could necessitate a full parallel test depending on the system utilization and as more dependencies are added. Similarly, with the exception of observation processing upgrades, all models needing a full production suite test will also require a production failover in order to complete the implementation.

C.III Validation Products and Delivery Times

C.III.a Validation of Products

Once NCO has the parallel system running and the developer has documented in the draft Service Change Notice (SCN) what products have been changed, added, or removed, NCO Dataflow runs utilities that compares the current production system to the parallel system. If Dataflow discovers changes that have not been documented in the SCN, they inform the developer who will either modify the SCN to match the parallel system, or make changes to the parallel system to match the changes as they were described in the SCN. Developers should make every effort to ensure that the changes described in the SCN match what the model package before it is delivered to NCO, so as to not cause any delays in the implementation schedule and to not adversely impact any operational downstream dependencies.

C.III.b Delivery time

Currently, anything more than a 5 minute change in delivery time will warrant an NCEP Director briefing. Developers run in a random, often over-utilized environment while production is consistent and stable. To enable developers to get consistent/accurate runtimes, in some instances NCO may allow developers to run their package on the production system. If this not possible, NCO would accept the package with only approximate estimates of runtime information (based on testing on the development computer) and work together with the developers on optimization once NCO has the package running routinely. This needs to happen once all science-based changes have been made, and the NCEP Director notified before the start of the 30-day IT parallel.

C.III.c I/O metrics

In addition to the items listed above, I/O metrics for operational implementations need to be documented. Such metrics include:

- The time the code takes to do I/O
- The volume of I/O

- The number of I/O transactions.
- Whether the I/O is synchronous or not, and whether it is sequential (easily buffered) or not.

It is desirable that these I/O metrics should be documented when the code is tested in early development stages, preferably by the time of the EE Kickoff meeting with NCO. Details on the WCOSS Phase 1 and Phase 2 IO profiler tool are described in this WCOSS Wiki page. For WCOSS Cray IOBUF is currently available. As of this writing (6/13/2018) these will be superceded by more up-to-date I/O tool which can be applied on all WCOSS platforms.

APPENDIX A: Outline and Summary of EE1 (For Reference)

Early History

Late January 2011, a small team with members from NCEP's Environmental Modeling Center (EMC) and NCEP Central Operations (NCO) was put together to accelerate progress on a project to improve the NWP system implementation processes at NOAA. Team was set up as a response to the UCAR review in 2009. UCAR concluded that interactions between NCO and EMC were not optimal, and that this was particularly obviously in the implementation process for operational models.

The team was initially integrated by Hendrik Tolman (EMC, lead), Paul van Delst (EMC, svn), Chris Magee (NCO/PMB co-lead), Scott Jacobs (NCO, svn), Simon Hsiao (NCO/SPA), and Rick Hackenberg (NCO, project management). The initial goal of this team was to flesh out a basic technical design for the new NWP implementation process acceptable for EMC and NCO, by modernizing approaches and tools, maximizing similarity in NCO and EMC modeling environments, and optimizing interactions between development and operations. The team was later expanded to include Rich Putt (NCO), Jose-Henrique Alves (EMC), and Arun Chawla (EMC).
Scope:
The scope of the EE1 project was to define CCS Development, Test, NCO Parallel, and NCO Production environment boundaries, identify common and distinct environmental components, identify the desired configuration of common components, develop a plan to implement the desired configuration of all common environmental components, and migrate across environments, and finally, execute the plan.

Key Technical Features

The EE1 project's central objective, was to implement development and test environments within NCEP's central computing system (CCS) that were functionally equivalent to the NCO Parallel and Production environments. Focusing on facilitating the transition to operations of NWP systems, its key technical features were:

- To allow transition of models and other software from development through parallel testing and into production with little/no code modification.
- To unify the directory structures used by developers and NCO.

The new paradigm for directory layout would follow a "vertical structure", where individual models are in individual directories, rather than the previous "horizontal structure" where multiple systems scripts were bundled together in directories according to function (e.g. PREP, POST, etc).

- To allow separate versions of the same NWP system (and libraries) to live side-by-side, by adding version number to basic directory name.
- To enforce maintenance and hand-offs be performed through a code repository system and database (subversion software package).
- To unify scripting, utilities and libraries, product delivery mechanisms, and downstream system interactions.

Overview of the Previously-Proposed Implementation Process (that became EE1)

The proposed process for NWP system upgrades was as follows is provided in Appendix A. An overview of the process is as follows:

- EMC prepares Charter.
- EMC holds kick-off meeting. NCO attendees include SPA and Dataflow Team Leads and SIB representative.

- EMC holds their CCB.
- EMC code manager and Dataflow Team Lead work on TIN. Dataflow Lead submits TIN to NWS for dissemination.
- Pre-RFC email is sent to EMC (see App. B). Upon positive response to the questions in this email, PMB chief will reply with follow-up to EMC and SPA will be assigned and may begin work.
- SPA and developer begin to work together under /nwdev environment on backup CCS using EMC Subversion server. Work consists of single tests validated by EMC, followed by parallel testing (also validated by EMC).
- Single RFC is filed. Includes EMC SVN tag SPA will use to check out upgrade package to /nwpara environment.
- SPA Team Lead schedules briefing to NCEP Director.
- At end of 30-day evaluation period, SPA Team Lead gathers evaluation packets from participants.
- SPA Team Lead pre briefs NCO Director.
- SPA Team Lead and EMC brief NCEP Director.
- RFC is presented at NCO CCB.
- SPA prepares any needed implementation scripts/instructions.
- SPA implements upgrade on scheduled implementation day.
- SPA creates tag on NCO SVN server and gives this to EMC developer so he/she can check implementation version into EMC SVN.

Objectives/Benefits

EE1 had the objective to devise a new environment for EMC that would reproduce the NCO transition to operations (T2O) process as closely as possible, to streamline the handover of systems from research to the T2O step. The assumption was that if developers had the tools to run their codes and proposed upgrades in a manner similar to operations, the number of changes needed to be made by NCO's Production Management Branch (now Implementation and Data Services Branch) to make a code 'production-ready', would be minimized. The proposition also targeted reducing the likelihood of errors being made in the process.
The initial proposal for setting this environment was to create a 'nwdev' environment that developers would use. NCO would provide the tools and training so developers could use the SMS scheduler, then used by the NWS Production Job Suite on the CCS, to run their jobs on a development SMS server. The third foundation of the new process was the use of combined svn code repository servers/database, whereby developers and NCO would be able to share scripts and codes, and a full code repository history from early development, all the way to operational implementation. The latter would add consistency to the process, streamline changes required at either end during the development and T2O stages, and reduce duplication of efforts and the likelihood of mistakes being made and propagated.

As a consequence of the new paradigm, it was expected that EE1 would provide a standard environment for development and production. The main benefit would be a more effective and efficient end-to-end system for operational model upgrades, resulting in reduced time per implementation and an improved T2O process. The expected reduction in time was up to 10 weeks for a major model upgrade. However, the success of the initiative depended heavily on (i) the implementation of a trusted code repository system, shared by both developers and NCO; (ii) the availability of an identical scheduler for developers reproducing the operational scheduler (SMS at the time); and (iii) the advent of a shared implementation environment where systems could be “tweaked” simultaneously by developers and NCO during handover.

Accomplishments

The implementation of EE1 changed significantly the paradigm at NCEP for T2O and implementation. Successful elements of the T2O process that were improved include the adoption of a “vertical structure” for storing scripts and codes under consistent version-controlled directory trees; the establishment of a trusted system for code repository, sharing and storing scripts and codes (based on the Subversion software package), and the establishment of versioned modules that reduced duplication of libraries and shared elements of codes.
Next, we discuss in details the accomplishments and benefits related to each of these successful paradigm changes.

- **Vertical Structure**

  One of the main accomplishments of EE1 was the establishment of the so-called vertical structure, a new way to organize directory trees containing systems at EMC that would be reflected at both development and operational environments. The change was a huge success to move all applications into a top level structure, managed with version cards. Previously EMC and NCO struggled with having a “horizontal structure” where scripts and codes from all different systems were stored in functional directory trees, making it very hard to have a tighter view of the different components of any one system, as well as to update system components more effectively. The vertical structure approach solved these issues, greatly facilitating the transition of new systems and upgrades from research to operations.
  EE1 also provided a great basis for using the vertical structure to separate codes/scripts that were shared across systems, and system-specific components. EE1 was initially rolled out using the wave forecasting systems as its first test case. The opportunity led to having a separation between the wave model codes/scripts (drivers, wrappers for pre- and post-processing), which are shared across systems, and the wave system components (grids, bathymetries, system-specific codes/scripts), which are specific to the application at hand (e.g. Great Lakes model vs global deterministic vs global ensemble). This was beneficial for the latter, as it provided a good basis to handle upgrades that had different time scales for shared or individual resources.

- **Code repository**

  The use of a centralized code repository system, in association with the vertical structure approach, was a great advance to development efforts and their transition to operations. Under EE1, EMC and NCO established jointly a robust code repository system using Subversion. The effort was made more successful due to the development of internal expertise on using the system, particularly spearheaded by Dr. Paul Van Delst. EMC and NCO jointly established a set of procedures that made more effective the versioning of codes, which allowed more clarity and less noise in most of EMC's development efforts, as these are generally the result of many hands collaborating. The use of Subversion also made it possible for EMC to provide NCO with a clearer view of system changes, which were also available in a convenient form via the Trac wiki features integrated to the Subversion servers.

- **Modules**

  A consequence of using a vertical structure to organize codes and scripts under individual NWP systems, associated with a robust code repository system, was the opportunity to develop modules for maintaining and making available system-specific and general-use libraries. The early use of modules in some of NCEP's systems also provided a good transition to the more general use of modules in the most recent operational compute resources.

Problems Encountered

- **Vertical structure**

  The benefits of separating shared and system-specific resources were applied to several systems at NCEP. The approach was largely successful with most systems, as it unfolded from an initial implementation with the operational wave forecasting systems to other NWP suites. However, the paradigm did not address systems with particular needs to manage shared and system-specific resources. An example is the NCEP POST. The details of how to handle shared code in that case were not addressed more extensively, and need to be revisited with a wider scope. Other questions that were not fully answered in the EE1 process include: Is the vertical structure categorized by function or by application? Do we need multiple copies of NCEP POST in the application itself or a single, centralized version? These must be answered in future extensions of the EE proposal.

- **Job Scheduler and NWDEV**

In the midst of EE1, NCO changed its production job scheduler from SMS to ecFlow, because ECMWF (developer of both schedulers) would no longer support SMS. The ecFlow replacement scheduler was not yet released by ECMWF for operational use, and NCO had to send one of its Senior Production Analysts (SPA) over to ECMWF to be trained in ecFlow's use. The NCO SPA team assisted ECMWF in debugging ecFlow and then transitioned the NCEP Production Job Suite to use ecFlow, but this meant that teaching EMC developers to use the job scheduler had to be put on hold, and a developer version of the ecFlow servers also could not be set up in time to leverage the EE1 process. This became a major burden to the end-to-end success of the EE1 paradigm that must be addressed in future extensions of the project.

The concept of nwdev was not fully embrace by NCO. For many at NCO, nwdev never made sense, and as a practice, it was never implemented/used. NCO already had three environments, namely nwtest, nwpara, and nwprod. The need for another one was justified to provide joint access from EMC and NCO, which was in principle a good idea, as it would provide access to EMC to tools such as the ecFlow scheduler. Part of the lack of attention to this important requirement of EE1 was a result of changing job schedulers from SMS to ecFlow, as this required an unexpected overhead from the SPA team, and removed resources from being committed on time to the deployment of the nwdev framework.

A question that remained unanswered and needs to be addressed in future extension of the EE project: is this something that is needed, or would it be more appropriate to provide EMC its own separate ecFlow scheduler? In any case, a solution is still needed to provide developers a solid platform for testing codes at all levels prior to transition to operations, using a job scheduler and compute environment that identically reproduces the operational ones.

- **Code repository**
  Code repository versioning was one of the most successful paradigm changes proposed and implemented by EE1. In contrast to 5 years ago, when most codes and systems were not using a consistent code repository system, today nearly all systems are version-controlled and shared via NCEP's svn servers. Despite the success, EE1 left a few items to be addressed in future extensions of the EE project, as follows. A clear path to management of old versions, and legacy content, as well as the proper way of storing prior versions of systems and libraries must be devised. Too rigid Subversion procedures created problems in unifying the code repository process across all systems. For security reasons, two separate svn servers were created for developers and NCO, which subverted the initial EE1 proposition of having developers and NCO sharing the same svn server, at least during the T2O path. This deviation brought about several malpractices that need also to be addressed in the future, such as most implementations never having SPA and EMC working together in an EMC branch, codes usually not being frozen at that stage to have NCO creating an implementation branch before

implementing etc. One of the main resulting malpractices that needs to be fixed is that now many "small changes" are passed around outside of svn, that become practical nightmares when subsequent upgrades are required but the changes are not recorded in the code repository track.

- **Separate developer and NCO repository servers for security reasons**
  The use of separate developer and NCO repository servers ultimately paralyzed one of the central objectives of the EE1 project, which was to enforce unification of codes and scripts between developers and NCO, using the repository system (then Subversion) as the “enforcer”. The lack of a unified svn server led to the reintroduction of old (bad) practices, where changes would be made on either side of the R2O process without proper communication or record, in a way that made the continuous transition to operations process more difficult over time. This is still a potential issue in EE2 that needs to be addressed.

Future Directions

Future directions are being investigated and proposed via the rolling off of a continuation of EE1, properly called the Environment Equivalence Version 2 (EE2). From the standpoint of lessons learned after EE1 was mostly concluded, EE2 must focus on addressing not only the new requirements set forth by the UMAC, but also the following critical items that either failed, or were incompletely implemented or properly pursued during the lifetime of the EE1.

- The vertical structure approach must be expanded to address systems with particular needs for managing shared and system-specific resources (for example, NCEP POST),
- A set of tools for scheduling jobs and running systems to the ones used by NCO to run, the operational NWP suite at NCEP (e.g. ecFlow), must be made available to developers to ensure a more successful T2O and timely error capturing/fixing,
- A proper environment for shared testing of new codes/libraries and system upgrades involving simultaneously developers and NCO must be investigated, and a solution for in-tandem testing during the T2O stage, envisaged.
- The versioning process must be streamlined in a less-rigid way to ensure all changes are made and communicated, either at the developers’ or NCO’s side, strictly via the svn servers; a process encouraging direct communication must also be envisaged to ensure all stakeholders are aware of changes being made to systems in the T2O stage.
- A unification of repository servers, or a system ensuring consistency between codes maintained at the developer and NCO servers, reinforced by a trusted communication system, must be implemented to ensure upgrades are made consistently and without errors being propagated in subsequent changes (e.g. NCO must ensure changes made during or after the T2O process are reflected in the developers repository etc).