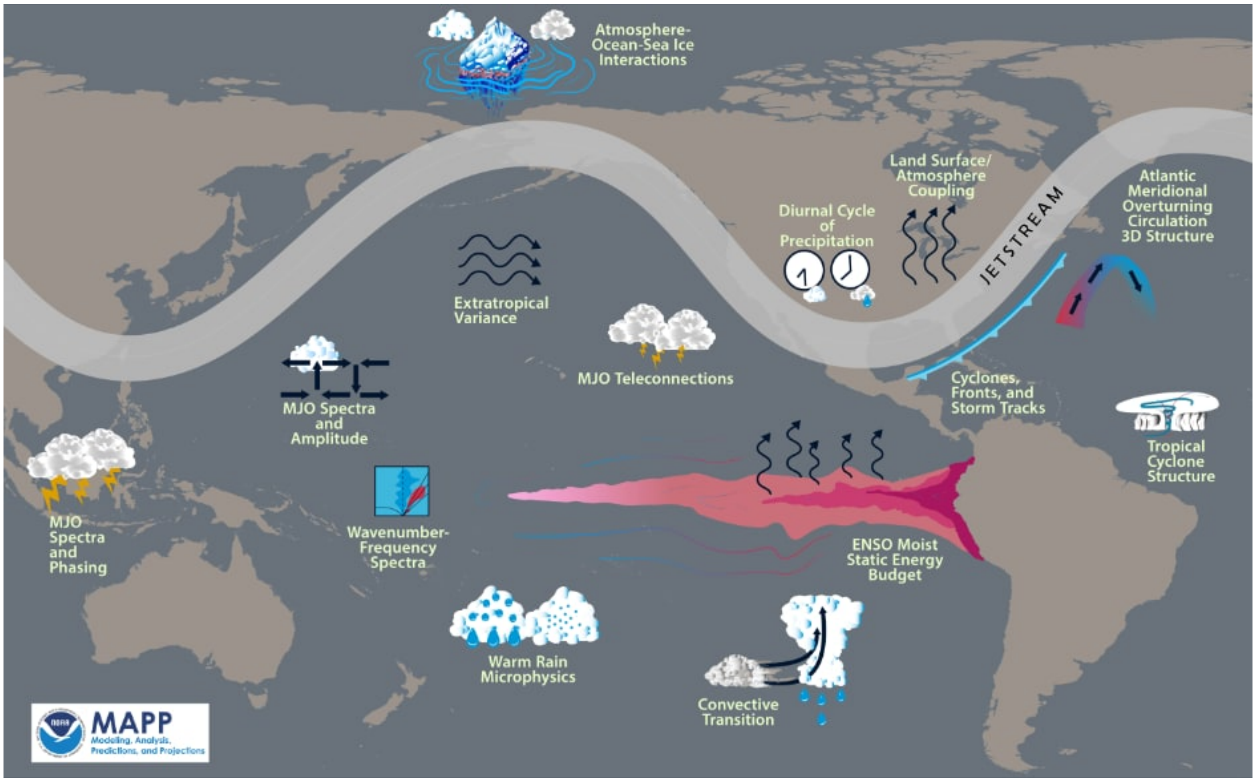

Atmosphere-Ocean-Sea Ice Interactions
Land Surface/Atmosphere Coupling
Diurnal Cycle of Precipitation
JETSTREAM
Atlantic Meridional Overturning Circulation 3D Structure
Extratropical Variance
MJO Teleconnections
Cyclones, Fronts, and Storm Tracks
MJO Spectra and Amplitude
Tropical Cyclone Structure
MJO Spectra and Phasing
Wavenumber-Frequency Spectra
ENSO Moist Static Energy Budget
Warm Rain Microphysics
Convective Transition
MAPP
Modeling, Analysis, Predictions, and Projections

MDTF Developer's Walkthrough

Release 3.0

Jessica Liptak[a] Dani Coleman[b]
Thomas Jackson Aparna Radhakrishnan[c]
Andrew Gettelman[b] J. David Neelin[a]
Eric Maloney[d] John Krasting[c]

(a: UCLA; b: NCAR; c: GFDL; d:CSU)

Nov 04, 2022

CONTENTS

CHAPTER

ONE

DEVELOPER INFORMATION

1.1 Introduction for POD developers

This walkthrough contains information for developers wanting to contribute a process-oriented diagnostic (POD) module to the MDTF framework. There are two tracks through the material: one for developers who have an existing analysis script they want to adapt for use in the framework, and one for developers who are writing a POD from scratch.

Section 1.4 provides instructions for setting up POD development, in particular managing language and library dependencies through conda. For developers already familiar with version 2.0 of the framework, Section 1.2 summarizes changes from v2.0 to facilitate migration to v3.0. New developers can skip this section, as the rest of this walkthrough is self-contained.

Section 1.3 Provides a list of instructions for submitting a POD for inclusion in the framework. We require developers to submit PODs through GitHub[1]. See Git-based development workflow (page 28) for how to manage code through the GitHub website.

Section 1.5 provides overall guidelines for POD development. Section 1.6 is a reference for the POD's settings file format. In Section 1.7, we walk the developers through the workflow of the framework, focusing on aspects that are relevant for the operation of individual PODs, and using the Example Diagnostic POD[2] as a concrete example to illustrate how a POD works under the framework Section 1.8 provides coding best practices to address common issues encountered in submitted PODs..

1.1.1 Scope of a process-oriented diagnostic

The MDTF framework imposes requirements on the types of data your POD outputs and takes as input. In addition to the scientific scope of process-oriented diagnostics, the analysis that you intend to do needs to fit the following model:

Your POD should accept model data as input and express the results of its analysis in a series of figures, which are presented to the user in a web page. Input model data will be in the form of one NetCDF file (with accompanying dimension information) per variable, as requested in your POD's settings file (page 15). Because your POD may be run on the output of any model, you should be careful about the assumptions your code makes about the layout of these files (eg, the range of longitude or the positive[3] convention for vertical coordinates). Supporting data may be in any format and will not be modified by the framework (see next section).

[1] https://github.com/NOAA-GFDL/MDTF-diagnostics

[2] https://github.com/NOAA-GFDL/MDTF-diagnostics/tree/main/diagnostics/example

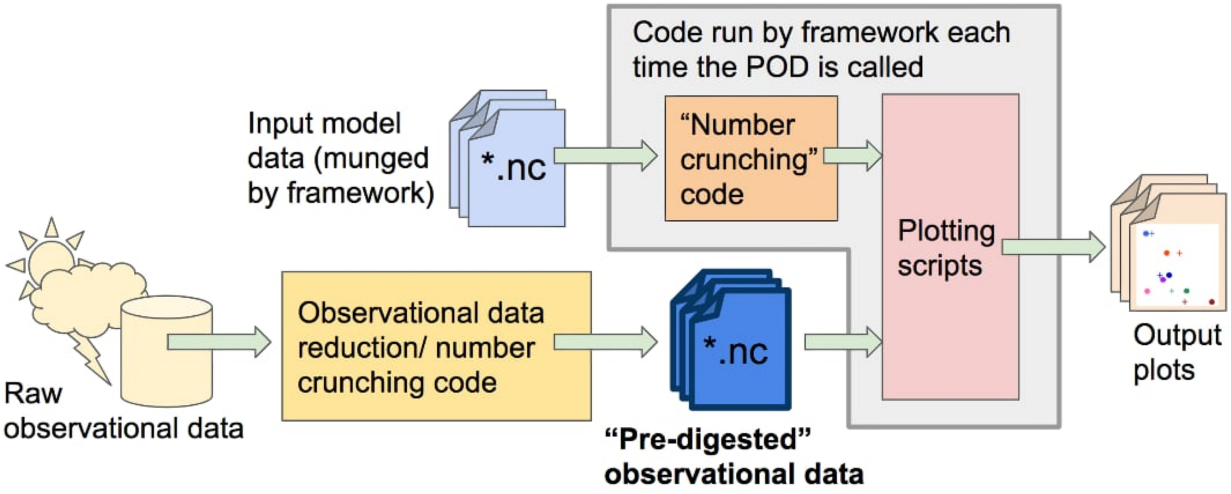
[3] http://cfconventions.org/faq.html#vertical_coords_positive_attribute

The above data sources are your POD's only input: your POD should not access the internet or other networked resources. You may provide options in the settings file for the user to configure when the POD is installed, but these cannot be changed each time the POD is run.

To achieve portability, the MDTF cannot accept PODs written in closed-source languages (eg, MATLAB or IDL). We also cannot accept PODs written in compiled languages (eg, C or Fortran): installation would rapidly become impractical if users had to check compilation options for each POD.

The output of your POD should be a series of figures in vector format (.eps or .ps). Optionally, we encourage POD developers to also save relevant output data (e.g., the output data being plotted) as netcdf files, to give users the ability to take the POD's output and perform further analysis on it.

1.1.2 POD code organization and supporting data


Code run by framework each time the POD is called
Input model data (munged by framework)
*.nc
"Number crunching" code
Plotting scripts
Output plots
Raw observational data
Observational data reduction/ number crunching code
*.nc
"Pre-digested" observational data

In order to make your code run faster for the users, we request that you separate any calculations that don't depend on the model data (e.g., pre-processing of observational data), and instead save the end result of these calculations in data files for your POD to read when it is run. We refer to this as "digested observational data," but it refers to any quantities that are independent of the model being analyzed. For purposes of data provenance, reproducibility, and code maintenance, we request that you include all the pre-processing/data reduction scripts used to create the digested data in your POD's code base, along with references to the sources of raw data these scripts take as input (yellow box in the figure).

Digested data should be in the form of numerical data, not figures, even if the only thing the POD does with the data is produce an unchanging reference plot. We encourage developers to separate their "number-crunching code" and plotting code in order to give end users the ability to customize output plots if needed. In order to keep the amount of supporting data needed by the framework manageable, we request that you limit the total amount of digested data you supply to no more than a few gigabytes.

In collaboration with PCMDI, a framework is being advanced that can help systematize the provenance of observational data used for POD development. This section will be updated when this data source is ready for public use.

1.2 Migration from framework v2.0

In this section we describe the major changes made from v2.0 to v3.0 of the framework that are relevant for POD developers. The scope of the framework has expanded in version 3.0, which required changes in the way the PODs and framework interact. New developers can skip this section, as the rest of this documentation is self-contained.

1.2.1 Getting Started and Developer's Walkthrough

A main source of documentation for v2.0 of the framework were the "Getting Started" and "Developer's Walkthrough" documents. Updated versions of these documents are:

- Getting Started v3.0 (PDF)[4]
- Developer's Walkthrough v3.0 (PDF)[5]

Note: These documents contain a subset of information available on this website, rather than new material: the text is reorganized to be placed in the same order as the v2.0 documents, for ease of comparison.

1.2.2 Checklist for migrating a POD from v2.0

Here we list the broad set of tasks needed to update a POD written for v2.0 of the framework to v3.0.

- Update settings and varlist files: In v3.0 these have been combined into a single `settings.jsonc` file. See the settings file guide (page 15), reference (page 37), and example[6] for descriptions of the new format.
- Update references to framework environment variables: See the table below for an overview, and the reference (page 48) for complete information on what environment variables the framework sets. Note that your POD should not use any hard-coded paths or variable names, but should read this information in from the framework's environment variables.
- Resubmit digested observational data: To minimize the size of supporting data users need to download, we ask that you only supply observational data specifically needed for plotting (preferably size within MB range), as well as any code used to perform that data reduction from raw sources.
- Remove HTML templating code: Version 2.0 of the framework required that your POD's top-level driver script take particular steps to assemble its HTML file. In v3.0 these tasks are done by the framework: all that your POD needs to do is generate figures of the appropriate formats and names in the specified folders, and the framework will convert and link them appropriately.

[4] https://mdtf-diagnostics.readthedocs.io/en/latest/_static/MDTF_getting_started.pdf

[5] https://mdtf-diagnostics.readthedocs.io/en/latest/_static/MDTF_walkthrough.pdf

[6] https://github.com/NOAA-GFDL/MDTF-diagnostics/tree/main/diagnostics/example/settings.jsonc

1.2.3 Conversion from v2.0 environment variables

In v3.0, the paths referred to by the framework's environment variables have been changed to be specific to your POD. The variables themselves have been renamed to avoid possible confusion. Here's a table of the appropriate substitutions to make:

Table 1: Environment variable name conversion
[t][T][T][T]

Path Description	v2.0 environment variable expression	Equivalent v3.0 variable
Top-level code repository	`$DIAG_HOME`	No variable set: PODs should not access files outside of their own source code directory within `$POD_HOME`
POD's source code	`$VARCODE`/<pod name>	`$POD_HOME`
POD's observational/supporting data	`$VARDATA`/<pod name>	`$OBS_DATA`
POD's working directory	`$variab_dir`/<pod name>	`$WK_DIR`
Path to requested NetCDF data file for <variable name> at date frequency <freq>	Currently unchanged: `$DATADIR`/<freq>/`$CASENAME`.<variable name>.<freq>.nc	
Other v2.0 paths	`$DATA_IN`, `$DIAG_ROOT`, `$WKDIR`	No equivalent variable set. PODs shouldn't access files outside of their own directories; instead use one of the quantities above.

1.3 POD development checklist

This section lists all the steps that need to be taken in order to submit a POD for inclusion in the MDTF framework.

1.3.1 Code and documentation submission

The material in this section must be submitted though a pull request[7] to the NOAA-GFDL GitHub repo[8]. This is described in Git-based development workflow (page 28).

The example POD[9] should be used as a reference for how each component of the submission should be structured.

The POD feature must be up-to-date with the NOAA-GFDL main branch, with no outstanding merge conflicts. See Git-based development workflow (page 28) for instructions on syncing your fork with NOAA-GFDL, and pulling updates from the NOAA-GFDL main branch into your feature branch.

[7] https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests

[8] https://github.com/NOAA-GFDL/MDTF-diagnostics

[9] https://github.com/NOAA-GFDL/MDTF-diagnostics/tree/main/diagnostics/example

POD source code

All scripts should be placed in a subdirectory of `diagnostics/`. Among the scripts, there should be 1) a main driver script, 2) a template html, and 3) a `settings.jsonc` file. The POD directory and html template should be named after your POD's short name.

- For instance, `diagnostics/convective_transition_diag/` contains its driver script `convective_transition_diag.py`, `convective_transition_diag.html`, and `settings.jsonc`, etc.
- The framework will call the driver script, which calls the other scripts in the same POD directory.
- If you need a new Conda environment, add a new .yml file to `src/conda/`, and install the environment using the `conda_env_setup.sh` script as described in the Getting Started.

POD settings file

The format of this file is described in POD settings file summary (page 15) and in more detail in Diagnostic settings file format (page 37).

POD html template for output

- The html template will be copied by the framework into the output directory to display the figures generated by the POD. You should be able to create a new html template by simply copying and modifying the example templates from existing PODs even without prior knowledge about html syntax.

Preprocessing scripts for digested data

The "digested" supporting data policy is described in Section 1.1.2.

For maintainability and provenance purposes, we request that you include the code used to generate your POD's "digested" data from raw data sources (any source of data that's permanently hosted). This code will not be called by the framework and will not be used by end users, so the restrictions and guidelines concerning the POD code don't apply.

POD documentation

- The documentation for the framework is automatically generated using sphinx[10], which works with files in reStructured text[11] (reST, `.rst`) format. In order to include documentation for your POD, we require that it be in this format.
  - Use the example POD documentation[12] as a template for the information required for your POD, by modifying its .rst source code[13]. This should include a one-paragraph synopsis of the POD, developers' contact information, required programming language and libraries, and model output variables, a brief summary of the presented diagnostics as well as references in which more in-depth discussions can be found.

[10] https://www.sphinx-doc.org/en/master/index.html

[11] https://docutils.sourceforge.io/rst.html

[12] https://mdtf-diagnostics.readthedocs.io/en/latest/sphinx_pods/example.html

[13] https://raw.githubusercontent.com/NOAA-GFDL/MDTF-diagnostics/main/diagnostics/example/doc/example.rst

  - The .rst files and all linked figures should be placed in a `doc` subdirectory under your POD directory (e.g., `diagnostics/convective_transition_diag/doc/`) and put the .rst file and figures inside.
  - The most convenient way to write and debug reST documentation is with an online editor. We recommend https://livesphinx.herokuapp.com/ because it recognizes sphinx-specific commands as well.
  - For reference, see the reStructured text introduction[14], quick reference[15] and in-depth guide[16].
  - Also see a reST syntax comparison[17] to other text formats you may be familiar with.
- For maintainability, all scripts should be self-documenting by including in-line comments. The main driver script (e.g., `convective_transition_diag.py`) should contain a comprehensive header providing information that contains the same items as in the POD documentation, except for the "More about this diagnostic" section.
- The one-paragraph POD synopsis (in the POD documentation) as well as a link to the full documentation should be placed at the top of the html template (e.g., `convective_transition_diag.html`).

Preprocessing script documentation

The "digested" supporting data policy is described in Section 1.1.2.

For maintainability purposes, include all information needed for a third party to reproduce your POD's digested data from its raw sources in the `doc` directory. This information is not published on the documentation website and can be in any format. In particular, please document the raw data sources used (DOIs/versioned references preferred) and the dependencies/build instructions (eg. conda environment) for your preprocessing script.

1.3.2 Sample and supporting data submission

Data hosting for the MDTF framework is currently managed manually. The data is hosted via anonymous FTP on UCAR's servers.

Digested observational or supporting data

Create a directory under `inputdata/obs_data/` named after the short name of your POD, and put all your digested observation data in (or more generally, any quantities that are independent of the model being analyzed). The "digested" data policy is described in Section 1.1.2.

- Requirements - Digested data should be in the form of numerical data, not figures. - The data files should be small (preferably a few MB) and just enough for producing figures for model comparison. If you really cannot reduce the data size and your POD requires more than 1GB of space, consult with the lead team. - Include in the directory a "README.txt" description file with original source info. - Include in the directory any necessary licensing information, files, etc. (if applicable)

[14] http://docutils.sourceforge.net/docs/user/rst/quickstart.html
[15] http://docutils.sourceforge.net/docs/user/rst/quickref.html
[16] http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html
[17] http://hyperpolyglot.org/lightweight-markup

- Create a tar file of your obs_data directory: - Use the –hard_dereference flag so that all users can read your file. - Naming convention: $pod_name.yyyymmdd.tar, where yyyymmdd is the file creation date. Alternatively, you may use some other version tag to allow the framework to check compatibiity between the POD code and data provided. - Create the tar file from the inputdata directory so the file paths start with obs_data. - Example (c-shell):

  ```
  set pod_name = MJO_suite
  set tartail  =  `date +'%Y%m%d'`
  cd inputdata/obs_data
  tar cfh $pod_name.$tartail.tar --hard-dereference $pod_name
  ```

  – To check:

  ```
  % tar tf $pod_name.$tartail.tar
  MJO_suite/
  MJO_suite/ERA.v200.EOF.summer-0.png
  MJO_suite/ERA.u200.EOF.summer-1.png
  ```

After following the above instructions, please refer to the GitHub Discussion on transfering obs_data[18] or email Dani Coleman at bundy at ucar dot edu or contact your liason on the MDTF Leads Team.

Files will be posted for Guest/anonymous access : ftp://ftp.cgd.ucar.edu/archive/mdtf/obs_data_latest/{$pod_name}.latest.tar with 'latest' pointing to the date-or-version-tagged tar file

Note that, previous to version 3, obs_data from all PODs was consolidated in one tar file. To assist in usability as the number of PODs grow, they will now be available individually, with the responsiblity for creating the tar files on the developer.

Sample model data

For PODs dealing with atmospheric phenomena, we recommend that you use sample data from the following sources, if applicable:

- A timeslice run of NCAR CAM5[19]
- A timeslice run of GFDL AM4[20] (contact the leads for password).

1.4 Developer quickstart guide

This section contains instructions for beginning to

[18] https://github.com/NOAA-GFDL/MDTF-diagnostics/discussions/125
[19] https://www.earthsystemgrid.org/dataset/ucar.cgd.ccsm4.NOAA-MDTF.html
[20] http://data1.gfdl.noaa.gov/MDTF/

1.4.1 Developer installation instructions

To download and install the framework for development, follow the instructions for end users given in start_install, with the following developer-specific modifications:

Obtaining the source code

POD developers should create their branches from the main branch[21] of the framework code

```
git checkout -b feature/[POD name] main
```

This is the “beta test” version, used for testing changes before releasing them to end users

Developers may download the code from GitHub as described in ref-download, but we strongly recommend that you clone the repo in order to keep up with changes in the main branch, and to simplify submitting pull requests with your POD’s code. Instructions for how to do this are given in Git-based development workflow (page 28).

Installing dependencies via conda

Regardless of development language, we strongly recommend that developers use conda to manage their language and library versions. Note that Conda is not Python-specific, but allows coexisting versioned environments of most scripting languages, including, R[22], NCL[23], Ruby[24], PyFerret[25], and more.

Python-based PODs should be written in Python 3.10 or newer. We provide a developer version of the python3_base environment (described below) that includes Jupyter and other developer-specific tools. This is not installed by default, and must be requested by passing the `--all` flag to the conda setup script:

```
% cd $CODE_ROOT
% ./src/conda/conda_env_setup.sh --all --conda_root $CONDA_ROOT --env_dir $CONDA_ENV_DIR
```

1.4.2 POD development using existing Conda environments

To prevent the proliferation of dependencies, we suggest that new POD development use existing Conda environments whenever possible, e.g., python3_base[26], NCL_base[27], and R_base[28] for Python, NCL, and R, respectively.

In case you need any exotic third-party libraries, e.g., a storm tracker, consult with the lead team and create your own Conda environment following instructions (page 9) below.

[21] https://github.com/NOAA-GFDL/MDTF-diagnostics/tree/main
[22] https://anaconda.org/conda-forge/r-base
[23] https://anaconda.org/conda-forge/ncl
[24] https://anaconda.org/conda-forge/ruby
[25] https://anaconda.org/conda-forge/pyferret
[26] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/conda/env_python3_base.yml
[27] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/conda/env_NCL_base.yml
[28] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/conda/env_R_base.yml

Python

The framework provides the _MDTF_python3_base[29] Conda environment (recall the _MDTF prefix for framework-specific environment) as the generic Python environment, which you can install following the instructions. You can then activate this environment by running in a terminal:

```
% source activate $CONDA_ENV_DIR/_MDTF_python3_base
```

where $CONDA_ENV_DIR is the path you used to install the Conda environments. After you've finished working under this environment, run `% conda deactivate` or simply close the terminal.

Other languages

The framework also provides the _MDTF_NCL_base[30] and _MDTF_R_base[31] Conda environments as the generic NCL and R environments.

1.4.3 POD development using a new Conda environment

If your POD requires languages that aren't available in an existing environment or third-party libraries unavailable through the common conda-forge[32] and anaconda[33] channels, we ask that you notify us (since this situation may be relevant to other developers) and submit a YAML (.yml) file[34] that creates the environment needed for your POD.

- The new YAML file should be added to `src/conda/`, where you can find templates for existing environments from which you can create your own.
- The YAML filename should be `env_$your_POD_short_name.yml`.
- The first entry of the YAML file, name of the environment, should be `_MDTF_$your_POD_short_name`.
- We recommend listing conda-forge as the first channel to search, as it's entirely open source and has the largest range of packages. Note that combining packages from different channels (in particular, conda-forge and anaconda channels) may create incompatibilities.
- We recommend constructing the list of packages manually, by simply searching your POD's code for `import` statements referencing third-party libraries. Please do not exporting your development environment with `% conda env export`, which gives platform-specific version information and will not be fully portable in all cases; it also does so for every package in the environment, not just the "top-level" ones you directly requested.
- We recommend specifying versions as little as possible, out of consideration for end-users: if each POD specifies exact versions of all its dependencies, conda will need to install multiple versions of the same libraries. In general, specifying a version should only be needed in cases where backward compatibility was broken (e.g., Python 2 vs. 3) or a bug affecting your POD was fixed (e.g., postscript

[29] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/conda/env_pythone3_base.yml
[30] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/conda/env_NCL_base.yml
[31] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/conda/env_R_base.yml
[32] https://conda-forge.org/feedstocks/
[33] https://docs.anaconda.com/anaconda/packages/pkg-docs/
[34] https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#creating-an-environment-file-manually

font rendering on Mac OS with older NCL). Conda installs the latest version of each package that's consistent with all other dependencies.

Framework interaction with conda environments

As described in ref-execute, when you run the `mdtf` executable, among other things, it reads `pod_list` in `default_tests.jsonc` and executes POD codes accordingly. For a POD included in the list (referred to as $POD_NAME):

1. The framework will first try to look for the YAML file `src/conda/env_$POD_NAME.yml`. If it exists, the framework will assume that the corresponding conda environment `_MDTF_$POD_NAME` has been installed under `$CONDA_ENV_DIR`, and will switch to this environment and run the POD.
2. If not, the framework will then look into the POD's `settings.jsonc` file in `$CODE_ROOT/diagnostics/$POD_NAME/`. The `runtime_requirements` section in `settings.jsonc` specifies the programming language(s) adopted by the POD:

   a). If purely Python 3, the framework will look for `src/conda/env_python3_base.yml` and check its content to determine whether the POD's requirements are met, and then switch to `_MDTF_python3_base` and run the POD.

   b). Similarly, if NCL or R is used, then `NCL_base` or `R_base`.

Note that for the 6 existing PODs depending on NCL (EOF_500hPa, MJO_prop_amp, MJO_suite, MJO_teleconnection, precip_diurnal_cycle, and Wheeler_Kiladis), Python is also used but merely as a wrapper. Thus the framework will switch to `_MDTF_NCL_base` when seeing both NCL and Python in `settings.jsonc`.

The framework verifies PODs' requirements via looking for the YAML files and their contents. Thus if you choose to selectively install conda environments using the `--env` flag (ref-conda-install), remember to install all the environments needed for the PODs you're interested in, and that `_MDTF_base` is mandatory for the framework's operation.

- For instance, the minimal installation for running the `EOF_500hPa` and `convective_transition_diag` PODs requres `_MDTF_base` (mandatory), `_MDTF_NCL_base` (because of b), and `_MDTF_convective_transition_diag` (because of 1). These can be installed by passing `base`, `NCL_base`, and `convective_transition_diag` to the `--env` flag one at a time (ref-conda-install).

Testing with a new Conda environment

If you've updated an existing environment or created a new environment (with corresponding changes to the YAML file), verify that your POD works.

Recall how the framework finds a proper Conda environment for a POD. First, it searches for an environment matching the POD's short name. If this fails, it then looks into the POD's `settings.jsonc` and prepares a generic environment depending on the language(s). Therefore, no additional steps are needed to specify the environment if your new YAML file follows the naming conventions above (in case of a new environment) or your `settings.jsonc` correctly lists the language(s) (in case of updating an existing environment).

- For an updated environment, first, uninstall it by deleting the corresponding directory under `$CONDA_ENV_DIR`.

- Re-install the environment using the `conda_env_setup.sh` script as described in the installation instructions, or create the new environment for you POD:

```
% cd $CODE_ROOT
% ./src/conda/conda_env_setup.sh --env $your_POD_short_name --conda_root
↪$CONDA_ROOT --env_dir $CONDA_ENV_DIR
```

- Have the framework run your POD on suitable test data.
  1. Add your POD's short name to the `pod_list` section of the configuration input file (template: `src/default_tests.jsonc`).
  2. Prepare the test data as described in start_config.

1.5 POD development guidelines

1.5.1 Admissible languages

The framework itself is written in Python, and can call PODs written in any scripting language. However, Python support by the lead team will be "first among equals" in terms of priority for allocating developer resources, etc.

- To achieve portability, the MDTF cannot accept PODs written in closed-source languages (e.g., MATLAB and IDL; try Octave[35] and GDL[36] if possible). We also cannot accept PODs written in compiled languages (e.g., C or Fortran): installation would rapidly become impractical if users had to check compilation options for each POD.
- Python is strongly encouraged for new PODs; PODs funded through the CPO grant are requested to be developed in Python. Python version >= 3.8 is required. Official support for Python 2 was discontinued as of January 2020.
- If your POD was previously developed in NCL or R (and development is not funded through a CPO grant), you do not need to re-write existing scripts in Python 3 if doing so is likely to introduce new bugs into stable code, especially if you're unfamiliar with Python.
- If scripts were written in closed-source languages, translation to Python 3.8 or above is required.

1.5.2 Preparation for POD implementation

We assume that, at this point, you have a set of scripts, written in languages consistent with the framework's open source policy, that a) read in model data, b) perform analysis, and c) output figures. Here are 3 steps to prepare your scripts for POD implementation.

We recommend running the framework on the sample model data again with both `save_ps` and `save_nc` in the configuration input `src/default_tests.jsonc` set to `true`. This will preserve directories and files created by individual PODs in the output directory, which could come in handy when you go through the instructions below, and help understand how a POD is expected to write output.

[35] https://www.gnu.org/software/octave/

[36] https://github.com/gnudatalanguage/gdl

- Give your POD an official name (e.g., Convective Transition; referred to as `long_name`) and a short name (e.g., convective_transition_diag). The latter will be used consistently to name the directories and files associated with your POD, so it should (1) loosely resemble the long_name, (2) avoid space bar and special characters (!@#$%^&*), and (3) not repeat existing PODs' name (i.e., the directory names under `diagnostics/`). Try to make your POD's name specific enough that it will be distinct from PODs contributed now or in the future by other groups working on similar phenomena.
- If you have multiple scripts, organize them so that there is a main driver script calling the other scripts, i.e., a user only needs to execute the driver script to perform all read-in data, analysis, and plotting tasks. This driver script should be named after the POD's short name (e.g., `convective_transition_diag.py`).
- You should have no problem getting scripts working as long as you have (1) the location and filenames of model data, (2) the model variable naming convention, and (3) where to output files/figures. The framework will provide these as environment variables that you can access (e.g., using `os.environ` in Python, or `getenv` in NCL). DO NOT hard code these paths/filenames/variable naming convention, etc., into your scripts. See the complete list of environment variables supplied by the framework.
- Your scripts should not access the internet or other networked resources.

1.5.3 An example of using framework-provided environment variables

The framework provides a collection of environment variables, mostly in the format of strings but also some numbers, so that you can and MUST use in your code to make your POD portable and reusable.

For instance, using 3 of the environment variables provided by the framework, `CASENAME`, `DATADIR`, and `pr_var`, the full path to the hourly precipitation file can be expressed as

```
MODEL_OUTPUT_DIR = os.environ["DATADIR"]+"/1hr/"
pr_filename = os.environ["CASENAME"]+"."+os.environ["pr_var"]+".1hr.nc"
pr_filepath = MODEL_OUTPUT_DIR + pr_filename
```

You can then use `pr_filepath` in your code to load the precipitation data.

Note that in Linux shell or NCL, the values of environment variables are accessed via a `$` sign, e.g., `os.environ["CASENAME"]` in Python is equivalent to `$CASENAME` in Linux shell/NCL.

1.5.4 Relevant environment variables

The environment variables most relevant for a POD's operation are:

- `POD_HOME`: Path to directory containing POD's scripts, e.g., `diagnostics/convective_transition_diag/`.
- `OBS_DATA`: Path to directory containing POD's supporting/digested observation data, e.g., `inputdata/obs_data/convective_transition_diag/`.
- `DATADIR`: Path to directory containing model data files for one case/experiment, e.g., `inputdata/model/QBOi.EXP1.AMIP.001/`.
- `WK_DIR`: Path to directory for POD to output files. Note that this is the only directory a POD is allowed to write its output. E.g., `wkdir/MDTF_QBOi.EXP1.AMIP.001_1977_1981/convective_transition_diag/`.

1. Output figures to `$WK_DIR/obs/` and `$WK_DIR/model/` respectively.
2. `$WK_DIR/obs/PS/` and `$WK_DIR/model/PS/`: If a POD chooses to save vector-format figures, save them as `EPS` under these two directories. Files in these locations will be converted by the framework to `PNG` for HTML output. Caution: avoid using `PS` because of potential bugs in recent `matplotlib` and converting to PNG.
3. `$WK_DIR/obs/netCDF/` and `$WK_DIR/model/netCDF/`: If a POD chooses to save digested data for later analysis/plotting, save them in these two directories in `NetCDF`.

Note that (1) values of `POD_HOME`, `OBS_DATA`, and `WK_DIR` change when the framework executes different PODs; (2) the `WK_DIR` directory and subdirectories therein are automatically created by the framework. Each POD should output files as described here so that the framework knows where to find what, and also for the ease of code maintenance.

More environment variables for specifying model variable naming convention can be found in the `data/fieldlist_$convention.jsonc` files. Also see the list of environment variables supplied by the framework.

1.5.5 Guidelines for testing your POD

Test before distribution. Find people (eg, nearby postdocs/grads and members from other POD-developing groups) who are not involved in your POD's implementation and are willing to help. Give the tar files and point your GitHub repo to them. Ask them to try running the framework with your POD following the Getting Started instructions. Ask for comments on whether they can understand the documentation.

Test how the POD fails. Does it stop with clear errors if it doesn't find the files it needs? How about if the dates requested are not presented in the model data? Can developers run it on data from another model? Here are some simple tests you should try:

- Move the `inputdata` directory around. Your POD should still work by simply updating the values of `OBS_DATA_ROOT` and `MODEL_DATA_ROOT` in the configuration input file.
- Try to run your POD with a different set of model data.
- If you have problems getting another set of data, try changing the files' `CASENAME` and variable naming convention. The POD should work by updating `CASENAME` and `convention` in the configuration input.
- Try your POD on a different machine. Check that your POD can work with reasonable machine configuration and computation power, e.g., can run on a machine with 32 GB memory, and can finish computation in 10 min. Will memory and run time become a problem if one tries your POD on model output of high spatial resolution and temporal frequency (e.g., avoid memory problem by reading in data in segments)? Does it depend on a particular version of a certain library? Consult the lead team if there's any unsolvable problems.

1.5.6 Other tips on implementation

1. Structure of the code package: Implementing the constituent PODs in accordance with the structure described in earlier sections makes it easy to pass the package (or just part of it) to other groups.
2. Robustness to model file/variable names: Each POD should be robust to modest changes in the file/variable names of the model output; see Getting Started regarding the model data filename structure, An example of using framework-provided environment variables (page 12) and POD development checklist (page 4) regarding using the environment variables and robustness tests. Also, it would be easier to apply the code package to a broader range of model output.
3. Save digested data after analysis: Can be used, e.g., to save time when there is a substantial computation that can be re-used when re-running or re-plotting diagnostics. See Step 5: Output and cleanup (page 23) regarding where to save the output.
4. Self-documenting: For maintenance and adaptation, to provide references on the scientific underpinnings, and for the code package to work out of the box without support. See POD development checklist (page 4).
5. Handle large model data: The spatial resolution and temporal frequency of climate model output have increased in recent years. As such, developers should take into account the size of model data compared with the available memory. For instance, the example POD precip_diurnal_cycle and Wheeler_Kiladis only analyze part of the available model output for a period specified by the environment variables `FIRSTYR` and `LASTYR`, and the convective_transition_diag module reads in data in segments.
6. Basic vs. advanced diagnostics (within a POD): Separate parts of diagnostics, e.g, those might need adjustment when model performance out of obs range.
7. Avoid special characters (`!@#$%^&*`) in file/script names.

See ref-execute and :doc:` framework operation walkthrough <dev_walkthrough>` for details on how the package is called. See the command line reference for documentation on command line options (or run `mdtf --help`).

Avoid making assumptions about the machine on which the framework will run beyond what's listed here; a development priority is to interface the framework with cluster and cloud job schedulers to enable individual PODs to run in a concurrent, distributed manner.

1.6 POD settings file summary

This page gives a quick introduction to how to write the settings file for your POD. See the full documentation (page 37) on this file format for a complete list of all the options you can specify.

1.6.1 Overview

The MDTF framework can be viewed as a “wrapper” for your code that handles data fetching and munging. Your code communicates with this wrapper in two ways:

- The settings file is where your code talks to the framework: when you write your code, you document what model data your code uses and what format it expects it in. When the framework is run, it will fulfill the requests you make here (or tell the user what went wrong).
- When your code is run, the framework talks to it by setting environment variables (page 48) containing paths to the data files and other information specific to the run.

In the settings file, you specify what model data your diagnostic uses in a vocabulary you’re already familiar with:

- The CF conventions[37] for standardized variable names and units.
- The netCDF4 (classic) data model, in particular the notions of variables[38] and dimensions[39] as they’re used in a netCDF file.

1.6.2 Example

```
// Any text to the right of a '//' is a comment
{
  "settings" : {
    "long_name": "My example diagnostic",
    "driver": "example_diagnostic.py",
    "realm": "atmos",
    "runtime_requirements": {
      "python": ["numpy", "matplotlib", "netCDF4"]
    }
  },
  "data" : {
    "frequency": "day"
  },
  "dimensions": {
    "lat": {
      "standard_name": "latitude"
    },
    "lon": {
      "standard_name": "longitude"
    },
    "plev": {
```

(continues on next page)

[37] http://cfconventions.org/

[38] https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcVars.html

[39] https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcDims.html

(continued from previous page)

```
      "standard_name": "air_pressure",
      "units": "hPa",
      "positive": "down"
    },
    "time": {
      "standard_name": "time",
      "units": "day"
    }
  },
  "varlist" : {
    "my_precip_data": {
      "standard_name": "precipitation_flux",
      "path_variable": "PATH_TO_PR_FILE",
      "units": "kg m-2 s-1",
      "dimensions" : ["time", "lat", "lon"]
    },
    "my_3d_u_data": {
      "standard_name": "eastward_wind",
      "path_variable": "PATH_TO_UA_FILE",
      "units": "m s-1",
      "dimensions" : ["time", "plev", "lat", "lon"]
    }
  }
}
```

1.6.3 Settings section

This is where you describe your diagnostic and list the programs it needs to run.

`long_name`:
Display name of your diagnostic, used to describe your diagnostic on the top-level index.html page. Can contain spaces.

`driver`:
Filename of the driver script the framework should call to run your diagnostic.

`realm`:
One or more of the eight CMIP6 modeling realms (aerosol, atmos, atmosChem, land, landIce, ocean, ocnBgchem, seaIce) describing what data your diagnostic uses. This is give the user an easy way to, eg, run only ocean diagnostics on data from an ocean model.

`runtime_requirements`:
This is a list of key-value pairs describing the programs your diagnostic needs to run, and any third-party libraries used by those programs.

- The key is program's name, eg. languages such as "python[40]" or "ncl[41]" etc. but also any utilities such as "ncks[42]", "cdo[43]", etc.

[40] https://www.python.org/

[41] https://www.ncl.ucar.edu/

[42] http://nco.sourceforge.net/

[43] https://code.mpimet.mpg.de/projects/cdo

- The value for each program is a list of third-party libraries in that language that your diagnostic needs. You do not need to list built-in libraries: eg, in python, you should to list numpy[44] but not math[45]. If no third-party libraries are needed, the value should be an empty list.

1.6.4 Data section

This section contains settings that apply to all the data your diagnostic uses. Most of them are optional.

`frequency`:
: The time frequency the model data should be provided at, eg. “1hr”, “6hr”, “day”, “mon”, …

1.6.5 Dimensions section

This section is where you list the dimensions (coordinate axes) your variables are provided on. Each entry should be a key-value pair, where the key is the name your diagnostic uses for that dimension internally, and the value is a list of settings describing that dimension. In order to be unambiguous, all dimensions must specify at least:

`standard_name`:
: The CF standard name[46] for that coordinate.

`units`:
: The units the diagnostic expects that coordinate to be in (using the syntax of the UDUnits library[47]). This is optional: if not given, the framework will assume you want CF convention canonical units[48].

In addition, any vertical (Z axis) dimension must specify:

`positive`:
: Either `"up"` or `"down"`, according to the CF conventions[49]. A pressure axis is always `"down"` (increasing values are closer to the center of the earth).

1.6.6 Varlist section

This section is where you list the variables your diagnostic uses. Each entry should be a key-value pair, where the key is the name your diagnostic uses for that variable internally, and the value is a list of settings describing that variable. Most settings here are optional, but the main ones are:

`standard_name`:
: The CF standard name[50] for that variable.

`path_variable`:
: Name of the shell environment variable the framework will use to pass the location of the file containing this variable to your diagnostic when it’s run. See the environment variable documentation (page 48) for details.

[44] https://numpy.org/

[45] https://docs.python.org/3/library/math.html

[46] http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html

[47] https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax

[48] http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html

[49] http://cfconventions.org/faq.html#vertical_coords_positive_attribute

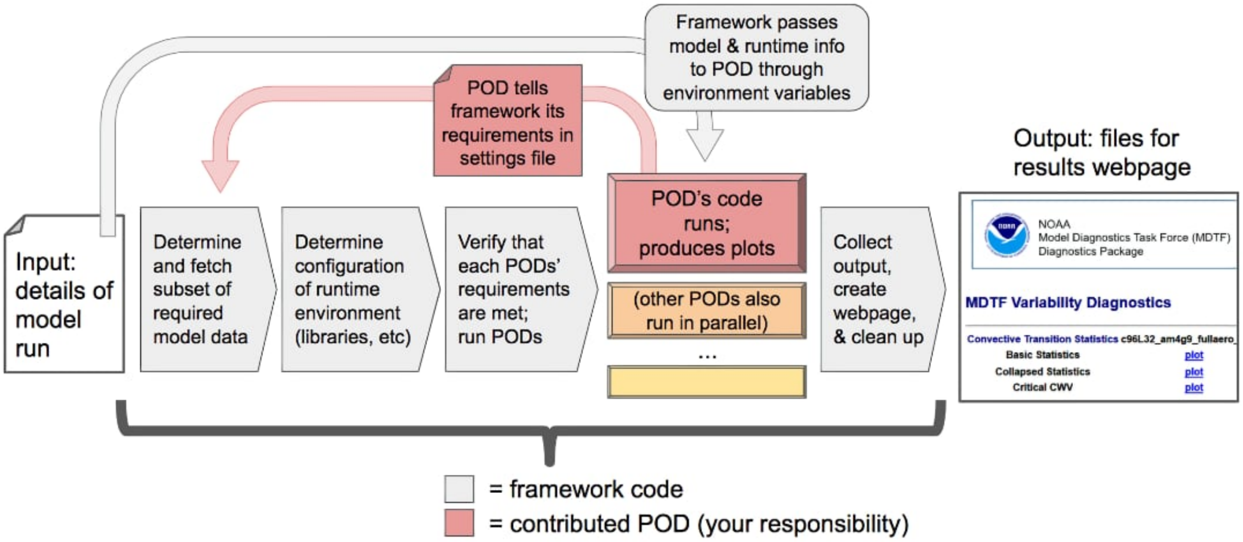
[50] http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html

`units`:
: The units the diagnostic expects the variable to be in (using the syntax of the UDUnits library[51]). This is optional: if not given, the framework will assume you want CF convention canonical units[52].

`dimensions`:
: List of names of dimensions specified in the "dimensions" section, to specify the coordinate dependence of each variable.

1.7 Walkthrough of framework operation

In this section, we describe the actions that are taken when the framework is run, focusing on aspects that are relevant for the operation of individual PODs. The Example Diagnostic POD[53] (short name: `example`) is used as a concrete example here to illustrate how a POD is implemented and integrated into the framework.


Framework passes model & runtime info to POD through environment variables
POD tells framework its requirements in settings file
Output: files for results webpage
Input: details of model run
Determine and fetch subset of required model data
Determine configuration of runtime environment (libraries, etc)
Verify that each PODs' requirements are met; run PODs
POD's code runs; produces plots
(other PODs also run in parallel)
...
Collect output, create webpage, & clean up
NOAA
Model Diagnostics Task Force (MDTF)
Diagnostics Package
MDTF Variability Diagnostics
Convective Transition Statistics c96L32_am4g9_fullaero_
Basic Statistics plot
Collapsed Statistics plot
Critical CWV plot
= framework code
= contributed POD (your responsibility)

We begin with a reminder that there are 2 essential files for the operation of the framework and POD:

- `src/default_tests.jsonc`: configuration input for the framework.
- `diagnostics/example/settings.jsonc`: settings file for the example POD.

To setup for running the example POD, (1) download the necessary supporting[54] and NCAR-CAM5.timeslice sample data[55] and unzip them under `inputdata/`, and (2) open `default_tests.jsonc`, uncomment the whole `NCAR-CAM5.timeslice` section in `case_list`, and comment out the other cases in the list. We also recommend setting both `save_ps` and `save_nc` to `true`.

[51] https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax
[52] http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html
[53] https://github.com/NOAA-GFDL/MDTF-diagnostics/tree/main/diagnostics/example
[54] ftp://ftp.cgd.ucar.edu/archive/mdtf/obs_data.example.tar
[55] ftp://ftp.cgd.ucar.edu/archive/mdtf/model.NCAR-CAM5.timeslice.tar

1.7.1 Step 1: Framework invocation

The user runs the framework by executing the framework's main driver script `$CODE_ROOT/mdtf`, rather than executing the PODs directly. This is where the user specifies the model run to be analyzed, and chooses which PODs to run via the `pod_list` section in `default_tests.jsonc`.

- Some of the configuration options can be input through command line, see the command line reference or run `% $CODE_ROOT/mdtf --help`.

At this stage, the framework also creates the directory `$OUTPUT_DIR/` (default: `mdtf/wkdir/`) and all subdirectories therein for hosting the output files by the framework and PODs from each run.

- If you've run the framework with both `save_ps` and `save_nc` in `default_tests.jsonc` set to `true`, check the output directory structure and files therein.

Note that when running, the framework will keep collecting the messages relevant to individual PODs, including (1) the status of required data and environment, and (2) texts printed out by PODs during execution, and will save them as log files under each POD's output directory. These `log` files can be viewed via the top-level results page `index.html` and, together with messages printed in the terminal, are useful for debugging.

Example diagnostic

Run the framework using the `NCAR-CAM5.timeslice` case. After successful execution, open the `index.html` under the output directory in a web browser. The `plots` links to the webpage produced by the example POD with figures, and `log` to `example.log` including all example-related messages collected by the framework. The messages displayed in the terminal are not identical to those in the log files, but also provide a status update on the framework-POD operation.

1.7.2 Step 2: Data request

Each POD describes the model data it requires as input in the `varlist` section of its `settings.jsonc`, with each entry in `varlist` corresponding to one model data file used by the POD. The framework goes through all the PODs to be run in `pod_list` and assembles a list of required model data from their `varlist`. It then queries the source of the model data (`$DATADIR/`) for the presence of each requested variable with the requested characteristics (e.g., frequency, units, etc.).

- The most important features of `settings.jsonc` are described in the settings documentation (page 15) and full detail on the reference page (page 37).
- Variables are specified in `varlist` following CF convention[56] wherever possible. If your POD requires derived quantities that are not part of the standard model output (e.g., column weighted averages), incorporate necessary preprocessing for computing these from standard output variables into your code. PODs are allowed to request variables outside of the CF conventions (by requiring an exact match on the variable name), but this will severely limit the POD's application.
- Some of the requested variables may be unavailable or without the requested characteristics (e.g., frequency). You can specify a backup plan for this situation by designating sets of variables as alternates if feasible: when the framework is unable to obtain a variable that has the `alternates` attribute in `varlist`, it will then (and only then) query the model data source for the variables named as alternates.

[56] http://cfconventions.org/

- If no alternates are defined or the alternate variables are also unavailable, the framework will skip executing your POD, and an `error log` will be presented in `index.html`.

Once the framework has determined which PODs are able to run given the model data, it prepares the necessary environment variables, including directory paths and the requested variable names (as defined in `data/fieldlist_$convention.jsonc`) for PODs' operation.

- At this step, the framework also checks the PODs' observational/supporting data under `inputdata/obs_data/`. If the directory of any of the PODs in `pod_list` is missing, the framework would terminate with error messages showing on the terminal. Note that the framework only checks the presence of the directory, but not the files therein.

Example diagnostic

The example POD uses only one model variable in its varlist[57]: surface air temperature, recorded at monthly frequency.

- In the beginning of `example.log`, the framework reports finding the requested model data file under `Found files`.
- If the framework could not locate the file, the log would instead record `Skipping execution` with the reason being missing data.

1.7.3 Step 3: Runtime environment configuration

The framework reads the other parts of your POD's `settings.jsonc`, e.g., `pod_env_vars`, and generates additional environment variables accordingly (on top of those being defined through `default_tests.jsonc`).

Furthermore, in the `runtime_requirements` section of `settings.jsonc`, we request that you provide a list of languages and third-party libraries your POD uses. The framework will check that all these requirements are met by one of the Conda environments under `$CONDA_ENV_DIR/`.

- The requirements should be satisfied by one of the existing generic Conda environments (updated by you if necessary), or a new environment you created specifically for your POD.
- If there isn't a suitable environment, the POD will be skipped.

Note that the framework's information about the Conda environments all comes from the YAML (.yml) files under `src/conda/` (and their contents) by assuming that the corresponding Conda environments have been installed using (thus are consistent with) the YAML files.

- The framework doesn't directly check files under `$CONDA_ENV_DIR/`, where the Conda environments locate.
- Therefore, it's imperative that you keep the Conda environments and the YAML files consistent at all time so the framework can properly function.

[57] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/diagnostics/example/settings.jsonc#L46

Example diagnostic

In its `settings.jsonc`, the example POD lists its requirements[58]: Python 3, and the matplotlib, xarray and netCDF4 third-party libraries for Python. In this case, the framework assigns the POD to run in the generic python3_base[59] environment provided by the framework.

- In `example.log`, under `Env vars:` is a comprehensive list of environment variables prepared for the POD by the framework. A great part of them are defined as in data/fieldlist_CMIP.jsonc[60] via setting `convention` in `default_tests.jsonc` to `CMIP`. Some of the environment variables are POD-specific as defined under pod_env_vars[61] in the POD's `settings.jsonc`, e.g., `EXAMPLE_FAV_COLOR`.
- In `example.log`, after `--- MDTF.py calling POD example`, the framework verifies the Conda-related paths, and makes sure that the `runtime_requirements` in `settings.jsonc` are met by the python3_base environment via checking env_python3_base.yml[62].

1.7.4 Step 4: POD execution

At this point, your POD's requirements have been met, and the environment variables are set. The framework then activates the right Conda environment, and begins execution of your POD's code by calling the top-level driver script listed in its `settings.jsonc`.

- See Relevant environment variables (page 12) for most relevant environment variables, and how your POD is expected to output results.
- All information passed from the framework to your POD is in the form of Unix/Linux shell environment variables; see reference for a complete list of environment variables (another good source is the log files for individual PODs).
- For debugging, we encourage that your POD print out messages of its progress as it runs. All text written to stdout or stderr (i.e., displayed in a terminal) will be captured by the framework and added to a log file available to the users via `index.html`.
- Properly structure your code/scripts and include error and exception handling mechanisms so that simple issues will not completely shut down the POD's operation. Here are a few suggestions:
  A. Separate basic and advanced diagnostics. Certain computations (e.g., fitting) may need adjustment or are more likely to fail when model performance out of observed range. Organize your POD scripts so that the basic part can produce results even when the advanced part fails.

  B. If some of the observational data files are missing by accident, the POD should still be able to run analysis and produce figures for model data regardless.

  C. Say a POD reads in multiple variable files and computes statistics for individual variables. If some of the files are missing or corrupted, the POD should still produce results for the rest (note that the framework would skip this POD due to missing data, but PODs should have this robustness property for ease of workarounds or running outside the framework).

[58] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/diagnostics/example/settings.jsonc#L38
[59] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/conda/env_python3_base.yml
[60] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/data/fieldlist_CMIP.jsonc
[61] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/diagnostics/example/settings.jsonc#L29
[62] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/conda/env_python3_base.yml

- The framework contains additional exception handling so that if a POD experiences a fatal or unrecoverable error, the rest of the tasks and POD-calls by the framework can continue. The error messages, if any, will be included in the POD's log file.

In case your POD requires derived quantities that are not part of the standard model output, and you've incorporated necessary preprocessing into your code (e.g., compute column average temperature from a vertically-resolved temperature field), one might be interested in saving these derived quantities as intermediate output for later use, and you may include this functionality in your code.

- Here we are referring to derived quantities gridded in a similar way to model output, instead of highly-digested data that is just enough for making figures.
- Save these as NetCDF files to the same directory containing the original model files. One file for one variable, following the filename convention spelled out in Getting Started.
- You must provide an option so that users can choose not to save the files (e.g., because of write permission, disk space, or files are accessed via soft links). Include this option through `pod_env_vars` in your POD's `settings.jsonc`, with "not to save" as default. You can remind users about this option by printing out messages in the terminal during runtime, or include a reminder in your POD documentation.

Example diagnostic

The framework activates the `_MDTF_python3_base` Conda environment, and calls the driver script example-diag.py[63] listed in `settings.jsonc`. Take a look at the script and the comments therein.

`example-diag.py` performs tasks roughly in the following order:

1) It reads the model surface air temperature data at `input_path`,
2) computes the model time average,
3) saves the model time averages to `$WK_DIR/model/netCDF/temp_means.nc` for later use,
4) plots model figure `$WK_DIR/model/PS/example_model_plot.eps`,
5) reads the digested data in time-averaged form at `$OBS_DATA/example_tas_means.nc`, and plots the figure to `$WK_DIR/obs/PS/example_obs_plot.eps`.

Note that these tasks correspond to the code blocks 1) through 5) in the script.

- When the script is called and running, it prints out messages which are saved in `example.log`. These are helpful to determine when and how the POD execution is interrupted if there's a problem.
- The script is organized to deal with model data first, and then to process digested observations. Thus if something goes wrong with the digested data, the script is still able to produce the html page with model figures. This won't happen if code block 5) is moved before 4), i.e., well-organized code is more robust and may be able to produce partial results even when it encounters problems.

In code block 7) of `example-diag.py`, we include an example of exception handling by trying to access a non-existent file (the final block is just to confirm that the error would not interrupt the script's execution because of exception-handling).

- The last few lines of `example.log` demonstrate the script is able to finish execution despite an error having occurred. Exception handling makes code robust.

[63] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/diagnostics/example/example_diag.py

1.7.5 Step 5: Output and cleanup

At this point, your POD has successfully finished running, and all remaining tasks are handled by the framework. The framework converts the postscript plots to bitmaps according to the following rule:

- `$WK_DIR/model/PS/filename.eps` → `$WK_DIR/model/filename.png`
- `$WK_DIR/obs/PS/filename.eps` → `$WK_DIR/obs/filename.png`

The html template for each POD is then copied to `$WK_DIR` by the framework.

- In writing the template file all plots should be referenced as relative links to this location, e.g., "`<A href=model/filename.png>`". See templates from existing PODs.
- Values of all environment variables referenced in the html template are substituted by the framework, allowing you to show the run's `CASENAME`, date range, etc. Text you'd like to change at runtime must be changed through environment variables (the v3 framework doesn't allow other ways to alter the text of your POD's output webpage at runtime).
- If `save_ps` and `save_nc` are set to `false`, the `.eps` and `.nc` files will be deleted.

Finally, the framework links your POD's html page to the top-level `index.html`, and copies all files to the specified output location (`OUTPUT_DIR` in `default_tests.jsonc`; same as `WK_DIR` by default).

- If `make_variab_tar` in `default_tests.jsonc` is set to `true`, the framework will create a tar file for the output directory, in case you're working on a server, and have to move the file to a local machine before viewing it.

Example diagnostic

Open the html template `diagnostics/example/example.html` and the output `$WK_DIR/example.html` in a text editor, and compare. All the environment variables in the template have been substituted, e.g., `{EXAMPLE_FAV_COLOR}` becomes `blue` (defined in `pod_env_vars` in settings.jsonc).

1.8 POD coding best practices

In this section we describe issues we've seen in POD code that have caused problems in the form of bugs, inefficiencies, or unintended consequences.

1.8.1 All languages

- PS vs. EPS figures: Save vector plots as .eps (Encapsulated PostScript), not .ps (regular PostScript).

  Why: Postscript (.ps) is perhaps the most common vector graphics format, and almost all plotting packages are able to output postscript files. Encapsulated Postscript[64] (.eps) includes bounding box information that describes the physical extent of the plot's contents. This is used by the framework to generate bitmap versions of the plots correctly: the framework calls ghostscript[65] for the conversion, and if not provided with a bounding box ghostscript assumes the graphics use an entire sheet of (letter or A4) paper. This can cause plots to be cut off if they extend outside of this region.

[64] https://en.wikipedia.org/wiki/Encapsulated_PostScript

[65] https://www.ghostscript.com/

Note that many plotting libraries will set the format of the output file automatically from the filename extension. The framework will process both *.ps and *.eps files.

1.8.2 Python: General

- Whitespace: Indent python code with four spaces per indent level.

  Why: Python uses indentation to delineate nesting and scope within a program, and indentation that's not done consistently is a syntax error. Using four spaces is not required, but is the generally accepted standard.

  Indentation can be configured in most text editors, or fixed with scripts such as `reindent.py` described here[66]. We recommend using a linter[67] such as `pylint` to find common bugs and syntax errors.

  Beyond this, we don't impose requirements on how your code is formatted, but voluntarily following standard best practices (such as described in PEP8[68] or the Google style guide[69]) will make it easier for you and others to understand your code, find bugs, etc.

- Filesystem commands: Use commands in the os[70] and shutil[71] modules to interact with the filesystem, instead of running unix commands using `os.system()`, `commands` (which is deprecated), or `subprocess`.

  Why: Hard-coding unix commands makes code less portable. Calling out to a subprocess introduces overhead and makes error handling and logging more difficult. The main reason, however, is that Python already provides these tools in a portable way. Please see the documentation for the os[72] and shutil[73] modules, summarized in this table:

Table 2: Recommended python functions for filesystem interaction

[t]|l|l|

Task	Recommended function
Construct a path from dir1, dir2, ..., filename	os.path.join[74](dir1, dir2, ..., filename)
Split a path into directory and filename	os.path.split[75](path) and related functions in os.path[76]
List files in directory dir	os.scandir[77](dir)
Move or rename a file or directory from old_path to new_path	shutil.move[78](old_path, new_path)
Create a directory or sequence of directories dir	os.makedirs[79](dir)
Copy a file from path to new_path	shutil.copy2[80](path, new_path)
Copy a directory dir, and everything inside it, to new_dir	shutil.copytree[81](dir, new_dir)
Delete a single file at path	os.remove[82](path)
Delete a directory dir and everything inside it	shutil.rmtree[83](dir)

[66] https://stackoverflow.com/q/1024435

[67] https://books.agiliq.com/projects/essential-python-tools/en/latest/linters.html

[68] https://www.python.org/dev/peps/pep-0008/

[69] https://github.com/google/styleguide/blob/gh-pages/pyguide.md

[70] https://docs.python.org/3.7/library/os.html

[71] https://docs.python.org/3.7/library/shutil.html

[72] https://docs.python.org/3.7/library/os.html

[73] https://docs.python.org/3.7/library/shutil.html

In particular, using os.path.join[84] is more verbose than joining strings but eliminates bugs arising from missing or redundant directory separators.

1.8.3 Python: Arrays

To obtain acceptable performance for numerical computation, people use Python interfaces to optimized, compiled code. NumPy[85] is the standard module for manipulating numerical arrays in Python. xarray[86] sits on top of NumPy and provides a higher-level interface to its functionality; any advice about NumPy applies to it as well.

NumPy and xarray both have extensive documentation and many tutorials, such as:

- NumPy's own basic[87] and intermediate[88] tutorials; xarray's overview[89] and climate and weather examples[90];
- A demonstration[91] of the features of xarray using earth science data;
- The 2020 SciPy conference has open-source, interactive tutorials[92] you can work through on your own machine or fully online using Binder[93]. In particular, there are tutorials for NumPy[94] and xarray[95].
- Eliminate explicit for loops: Use NumPy/xarray functions instead of writing for loops in Python that loop over the indices of your data array. In particular, nested for loops on multidimensional data should never need to be used.

  Why: For loops in Python are very slow compared to C or Fortran, because Python is an interpreted language. You can think of the NumPy functions as someone writing those for-loops for you in C, and giving you a way to call it as a Python function.

  It's beyond the scope of this document to cover all possible situations, since this is the main use case for NumPy. We refer to the tutorials above for instructions, and to the following blog posts that discuss this specific issue:

[74] https://docs.python.org/3.7/library/os.path.html?highlight=os%20path#os.path.join
[75] https://docs.python.org/3.7/library/os.path.html?highlight=os%20path#os.path.split
[76] https://docs.python.org/3.7/library/os.path.html?highlight=os%20path
[77] https://docs.python.org/3.7/library/os.html#os.scandir
[78] https://docs.python.org/3.7/library/shutil.html#shutil.move
[79] https://docs.python.org/3.7/library/os.html#os.makedirs
[80] https://docs.python.org/3.7/library/shutil.html#shutil.copy2
[81] https://docs.python.org/3.7/library/shutil.html#shutil.copytree
[82] https://docs.python.org/3.7/library/os.html#os.remove
[83] https://docs.python.org/3.7/library/shutil.html#shutil.rmtree
[84] https://docs.python.org/3.7/library/os.path.html?highlight=os%20path#os.path.join
[85] https://numpy.org/doc/stable/index.html
[86] http://xarray.pydata.org/en/stable/index.html
[87] https://numpy.org/doc/stable/user/absolute_beginners.html
[88] https://numpy.org/doc/stable/user/quickstart.html
[89] http://xarray.pydata.org/en/stable/quick-overview.html
[90] http://xarray.pydata.org/en/stable/examples.html
[91] https://rabernat.github.io/research_computing/xarray.html
[92] https://www.scipy2020.scipy.org/tutorial-information
[93] https://mybinder.org/
[94] https://github.com/enthought/Numpy-Tutorial-SciPyConf-2020
[95] https://xarray-contrib.github.io/xarray-tutorial/index.html

  - "Look Ma, no for-loops[96]," by Brad Solomon;
  - "Turn your conditional loops to Numpy vectors[97]," by Tirthajyoti Sarkar;
  - "'Vectorized' Operations: Optimized Computations on NumPy Arrays[98]", part of "Python like you mean it[99]," a free resource by Ryan Soklaski.

- Use xarray with netCDF data:

  Why: This is xarray's use case. You can think of NumPy as implementing multidimensional matrices in the fully general, mathematical sense, and xarray providing the specialization to the case where the matrix contains data on a lat-lon-time-(etc.) grid.

  xarray lets you refer to your data with human-readable labels such as 'latitude,' rather than having to remember that that's the second dimension of your array. This bookkeeping is essential when writing code for the MDTF framework, when your POD will be run on data from models you haven't been able to test on.

  In particular, xarray provides seamless support for time axes[100], with support[101] for all CF convention calendars through the `cftime` library. You can, eg, subset a range of data between two dates without having to manually convert those dates to array indices.

  See the xarray tutorials linked above for more examples of xarray's features.

- Memory use and views vs. copies: Use scalar indexing and slices[102] (index specifications of the form start_index:stop_index:stride) to get subsets of arrays whenever possible, and only use advanced indexing[103] features (indexing arrays with other arrays) when necessary.

  Why: When advanced indexing is used, NumPy will need to create a new copy of the array in memory, which can hurt performance if the array contains a large amount of data. By contrast, slicing or basic indexing is done in-place, without allocating a new array: the NumPy documentation calls this a "view."

  Note that array slices are native Python objects[104], so you can define a slice in a different place from the array you intend to use it on. Both NumPy and xarray arrays recognize slice objects.

  This is easier to understand if you think about NumPy as a wrapper around C-like functions: array indexing in C is implemented with pointer arithmetic, since the array is implemented as a contiguous block of memory. An array slice is just a pointer to the same block of memory, but with different offsets. More complex indexing isn't guaranteed to follow a regular pattern, so NumPy needs to copy the requested data in that case.

  See the following references for more information:

  - The NumPy documentation[105] on indexing;
  - "Numpy Views vs Copies: Avoiding Costly Mistakes[106]," by Jessica Yung;

[96] https://realpython.com/numpy-array-programming/
[97] https://towardsdatascience.com/data-science-with-python-turn-your-conditional-loops-to-numpy-vectors-9484ff9c622e
[98] https://www.pythonlikeyoumeanit.com/Module3_IntroducingNumpy/VectorizedOperations.html
[99] https://www.pythonlikeyoumeanit.com/
[100] http://xarray.pydata.org/en/stable/time-series.html
[101] http://xarray.pydata.org/en/stable/weather-climate.html
[102] https://numpy.org/doc/stable/reference/arrays.indexing.html#basic-slicing-and-indexing
[103] https://numpy.org/doc/stable/reference/arrays.indexing.html#advanced-indexing
[104] https://docs.python.org/3.7/library/functions.html?highlight=slice#slice
[105] https://numpy.org/doc/stable/reference/arrays.indexing.html
[106] https://www.jessicayung.com/numpy-views-vs-copies-avoiding-costly-mistakes/

  - "How can I tell if NumPy creates a view or a copy?[107]" on stackoverflow.

- MaskedArrays instead of NaNs or sentinel values: Use NumPy's MaskedArrays[108] for data that may contain missing or invalid values, instead of setting those entries to NaN or a sentinel value.

  Why: One sometimes encounters code which sets array entries to fixed "sentinel values" (such as 1.0e+20 or NaN[109]) to indicate missing or invalid data. This is a dangerous and error-prone practice, since it's frequently not possible to detect if the invalid entries are being used by mistake. For example, computing the variance of a timeseries with missing elements set to 1e+20 will either result in a floating-point overflow, or return zero.

  NumPy provides a better solution in the form of MaskedArrays[110], which behave identically to regular arrays but carry an extra boolean mask to indicate valid/invalid status. All the NumPy mathematical functions will automatically use this mask for error propagation. For example[111], trying to divide an array element by zero or taking the square root of a negative element will mask it off, indicating that the value is invalid: you don't need to remember to do these sorts of checks explicitly.

1.8.4 Python: Plotting

- Use the 'Agg' backend when testing your POD: For reproducibility, set the shell environment variable `MPLBACKEND` to `Agg` when testing your POD outside of the framework.

  Why: Matplotlib can use a variety of backends[112]: interfaces to low-level graphics libraries. Some of these are platform-dependent, or require additional libraries that the MDTF framework doesn't install. In order to achieve cross-platform portability and reproducibility, the framework specifies the `'Agg'` non-interactive (ie, writing files only) backend for all PODs, by setting the `MPLBACKEND` environment variable.

  When developing your POD, you'll want an interactive backend – for example, this is automatically set up for you in a Jupyter notebook. When it comes to testing your POD outside of the framework, however, you should be aware of this backend difference.

- Pass the cartopy CRS to plotting functions: See cartopy's documentation[113]. A coordinate reference system (CRS) must be passed as a `projection` argument when plot axes are created. This should be passed to subsequent functions that set the plot range (`crs` argument of `set_extent`: avoid the use of `set_xlim`/`set_ylim`) and to plotting functions (`transform` argument).

Note that this applies even to simple lat/lon plots, for which the appropriate CRS is `PlateCarree()`. Not specifying a CRS in this case will give rise to subtle errors, e.g. when trying to set longitude ranges of [-180,180] or [0, 360] in which the bounds map to the same location.

[107] https://stackoverflow.com/questions/11524664/how-can-i-tell-if-numpy-creates-a-view-or-a-copy
[108] https://numpy.org/doc/stable/reference/maskedarray.generic.html
[109] https://en.wikipedia.org/wiki/NaN
[110] https://numpy.org/doc/stable/reference/maskedarray.html
[111] https://numpy.org/doc/stable/reference/maskedarray.generic.html#numerical-operations
[112] https://matplotlib.org/tutorials/introductory/usage.html#backends
[113] https://scitools.org.uk/cartopy/docs/latest/tutorials/understanding_transform.html

1.8.5 NCL

- Large file support: By default, NCL cannot read netCDF files larger than 2gb. To drop this limitation, call setfileoption[114] with the following arguments in every script before any file operations:

  ```
  setfileoption("nc", "Format", getenv("MDTF_NC_FORMAT"))
  ```

  "netCDF4" can also be used as the requested format in the above call.

- Deprecated calendar functions: Check the function reference[115] to verify that the functions you use are not deprecated in the current version of NCL[116]. This is especially necessary for date/calendar functions[117].

  Why: The framework uses a current version of NCL[118] (6.6.x), to avoid plotting bugs that were present in earlier versions. This is especially relevant for calendar functions: the ut_* set of functions have been deprecated in favor of counterparts beginning with cd_ that take identical arguments (so code can be updated using find/replace). For example, use cd_calendar[119] instead of the deprecated ut_calendar[120].

  This change is necessary because only the cd_* functions support all calendars defined in the CF conventions, which is needed to process data from some models (eg, weather or seasonal models are typically run with a Julian calendar.)

1.9 Git-based development workflow

1.9.1 Steps for brand new users:

1. Fork the MDTF-diagnostics branch to your GitHub account (Creating a fork of the MDTF-diagnostics repository (page 29))
2. Clone (Cloning a repository onto your machine (page 29)) your fork of the MDTF-diagnostics repository (repo) to your local machine (if you are not using the web interface for development)
3. Check out a new branch from the local main branch (Working on a brand new feature (page 30))
4. Start coding
5. Commit the changes in your feature branch (Working on a brand new feature (page 30))
6. Push the changes to the copy of the feature branch on your remote fork (Working on a brand new feature (page 30))
7. Repeat steps 4–6 until you are finished working
8. Submit a pull request to the NOAA-GFDL repo for review (Submitting Pull Requests (page 30)).

[114] https://www.ncl.ucar.edu/Document/Functions/Built-in/setfileoption.shtml
[115] https://www.ncl.ucar.edu/Document/Functions/index.shtml
[116] https://www.ncl.ucar.edu/
[117] https://www.ncl.ucar.edu/Document/Functions/date.shtml
[118] https://www.ncl.ucar.edu/
[119] https://www.ncl.ucar.edu/Document/Functions/Built-in/cd_calendar.shtml
[120] https://www.ncl.ucar.edu/Document/Functions/Built-in/ut_calendar.shtml

1.9.2 Steps for users continuing work on an existing feature branch

1. Create a backup copy of the MDTF-Diagnostics repo on your local machine
2. Pull in updates from the NOAA-GFDL/main branch to the main branch in your remote repo (Updating your remote and local main branches (page 31))
3. Pull in updates from the main branch in your remote fork into the main branch in your local repo (Updating your remote and local main branches (page 31))
4. Sync your feature branch in your local repository with the local main branch using an interactive rebase (Updating your feature branch by rebasing it onto the main branch (preferred method) (page 32)) or merge (Updating your feature branch by merging in changes from the main branch (page 33)). Be sure to make a backup copy of of your local MDTF-diagnostics repo first, and test your branch after rebasing/merging as described in the linked instructions before proceeding to the next step.
5. Continue working on your feature branch
6. Commit the changes in your feature branch
7. Push the changes to the copy of the feature branch in your remote fork (Pushing to your remote feature branch on your fork (page 30))
8. Submit a pull request (PR) to NOAA-GFDL/main branch when your code is ready for review (Submitting Pull Requests (page 30))

1.9.3 Creating a fork of the MDTF-diagnostics repository

- If you have no prior experience with GitHub[121], create an account first.
- Create a fork of the project by clicking the `Fork` button in the upper-right corner of NOAA's MDTF GitHub page[122]. This will create a copy (also known as repository, or simply repo) in your own GitHub account which you have full control over.

1.9.4 Cloning a repository onto your machine

Before following the instructions below, make sure that a) you've created a fork of the project, and b) the `git` command is available on your machine (installation instructions[123]).

- Clone your fork onto your computer: `git clone git@github.com:<your_github_account>/MDTF-diagnostics.git`. This not only downloads the files, but due to the magic of git also gives you the full commit history of all branches.
- Enter the project directory: `cd MDTF-diagnostics`.
- Git knows about your fork, but you need to tell it about NOAA's repo if you wish to contribute changes back to the code base. To do this, type `git remote add upstream git@github.com:NOAA-GFDL/MDTF-diagnostics.git`. Now you have two remote repos: `origin`, your GitHub fork which you can read and write to, and `upstream`, NOAA's code base which you can only read from.

[121] https://github.com/

[122] https://github.com/NOAA-GFDL/MDTF-diagnostics

[123] https://git-scm.com/download/

Another approach is to create a local repo on your machine and manage the code using the `git` command in a terminal. In the interests of making things self-contained, the rest of this section gives brief step-by-step instructions on git for interested developers.

1.9.5 Working on a brand new feature

Developers can either clone the MDTF-diagnostics repo to their computer, or manage the MDTF package using the GitHub webpage interface. Whichever method you choose, remember to create your feature/[POD name] branch from the main branch, not the main branch. Since developers commonly work on their own machines, this manual provides command line instructions.

1. Check out a branch for your POD

```
git checkout -b feature/[POD name]
```

2. Write code, add files, etc…

3. Add the files you created and/or modified to the staging area

```
git add [file 1]
git add [file 2]
...
```

4. Commit your changes, including a brief description

```
git commit -m "description of my changes"
```

5. Push the updates to your remote repository

```
git push -u origin feature/[POD name]
```

1.9.6 Pushing to your remote feature branch on your fork

When you are ready to push your updates to the remote feature branch on your fork

1. Type `git status` to list the file(s) that have been updated
2. Repeat steps 3–5 of section (Working on a brand new feature (page 30))

1.9.7 Submitting Pull Requests

The pull request (PR) for your branch is your proposal to the maintainers to incorporate your feature into NOAA's repo. Your changes will not affect the official NOAA's repo until the PR is accepted by the lead-team programmer. Note that if any buttons are missing, try `CRTL` + `+` or `CRTL` + `-` to adjust the webpage font size so the missing buttons may magically appear.

To submit a PR :

1. Click the Contribute link on the main page of your MDTF-diagnostics fork and click the Open Pull Request button

2. Verify that your fork is set as the base repository, and main is set as the base branch, that NOAA-GFDL is set as the head repository, and main is set as the head branch

3. Click the Create Pull Request button, add a brief description to the PR header, and go through the checklist to ensure that your code meets that baseline requirements for review

4. Click the Create Pull Request button (now in the lower left corner of the message box).

Note that you can submit a Draft Pull Request if you want to run the code through the CI, but are not ready for a full review by the framework team. Starting from step 3. above

1. Click the arrow on the right edge of the Create Pull Request button and select Create draft pull request from the dropdown menu.
2. Continue pushing changes to your feature branch until you are ready for a review (the PR will update automatically)

3. When you are ready for review, navigate to the NOAA-GFDL/MDTF-Diagnostics *Pull requests*[124] page, and click on your PR

4. Scroll down to the header that states "this pull request is still a work in progress", and click the ready for review button to move the PR out of draft mode

1.9.8 Updating your remote and local main branches

Method 1: Web interface+command line

See the MDTF Best Practices Overview[125] presentation for instructions with figures.

1. Click the Fetch Upstream link on the main page of your MDTF-diagnostics fork, then click the Open Pull Request button
2. Verify that your fork is set as the base repository, and main is set as the base branch, that NOAA-GFDL is set as the head repository, and main is set as the head branch
3. Create a title for your PR, add a description if you want, then click Create pull request
4. Click Merge pull request

Your remote main branch is now up-to-date with the NOAA-GFDL/main branch.

5. On your machine, open a terminal and check out the main branch

```
git checkout main
```

6. Fetch the updates to the main branch from your remote fork

```
git fetch
```

7. Pull in the updates from the remote main branch.

```
git pull
```

Your local main branch is now up-to-date with the NOAA-GFDL/main branch.

[124] https://github.com/NOAA-GFDL/MDTF-diagnostics/pulls

[125] https://docs.google.com/presentation/d/18jbi50vC9X89vFbL0W1Ska1dKuW_yWY51SomWx_ahYE/edit?usp=sharing

Method 2: Command line only

This method requires adding the NOAA-GFDL/MDTF-diagnostics repo to the .git/config file in your local repo, and is described in the GitHub discussion post Working with multiple remote repositories in your git config file[126].

1.9.9 Updating your feature branch by rebasing it onto the main branch (preferred method)

Rebasing is procedure to integrate the changes from one branch into another branch. `git rebase` differs from `git merge` in that it reorders the commit history so that commits from the branch that is being updated are moved to the tip of the branch. This makes it easier to isolate changes in the feature branch, and usually results in fewer merge conflicts when the feature branch is merged into the main branch. 1. Create a backup copy of your MDTF-diagnostics repo on your local machine

2. Update the local and remote main branches on your fork as described in Updating your remote and local main branches (page 31), then check out your feature branch

```
git checkout feature/[POD name]
```

and launch an interactive rebase of your branch onto the main branch:: git rebase -i main 3. Your text editor will open in the terminal (Vim by default) and display your commit hashes with the oldest commit at the top

```
pick 39n3b42 oldest commit
pick 320cnyn older commit
pick 20ac93c newest commit
```

You may squash commits by replacing pick with squash for the commit(s) that are newer than the commit you want to combine with (i.e., the commits below the target commit). For example

```
pick 39n3b42 oldest commit
squash 320cnyn older commit
pick 20ac93c newest commit
```

combines commit 320cnyn with commit 29n3b42, while

```
pick 39n3b42 oldest commit
squash 320cnyn older commit
squash 20ac93c newest commit
```

combines 20ac93c and 320cnyn with 39n3b42.

Note that squashing commits is not required. However, doing so creates a more streamlined commit history.

4. Once you're done squashing commits (if you chose to do so), save your changes and close the editor `ESC + SHIFT + wq` to save and quit in Vim), and the rebase will launch. If the rebase stops because there are merge conflicts and resolve the conflicts. To show the files with merge conflicts, type

```
git status
```

[126] https://github.com/NOAA-GFDL/MDTF-diagnostics/discussions/96

This will show files with a message that there are merge conflicts, or that a file has been added/deleted by only one of the branches. Open the files in an editor, resolve the conflicts, then add edited (or remove deleted) files to the staging area

```
git add file1
git add file2
...
git rm file3
```

5. Next, continue the rebase

```
git rebase --continue
```

The editor will open with the modified commit history. Simply save the changes and close the editor (`ESC+SHIFT+wq`), and the rebase will continue. If the rebase stops with errors, repeat the merge conflict resolution process, add/remove the files to staging area, type `git rebase --continue`, and proceed.

If you have not updated your branch in a long time, you'll likely find that you have to keep fixing the same conflicts over and over again (every time your commits collide with the commits on the main branch). This is why we strongly advise POD developers to pull updates into their forks and rebase their branches onto the main branch frequently.

Note that if you want to stop the rebase at any time and revert to the original state of your branch, type

```
git rebase --abort
```

6. Once the rebase has completed, push your changes to the remote copy of your branch

```
git push -u origin feature/[POD name] --force
```

The `--force` option is necessary because rebasing modified the commit history.

7. Now that your branch is up-to-date, write your code!

1.9.10 Updating your feature branch by merging in changes from the main branch

1. Create a backup copy of your repo on your machine.
2. Update the local and remote main branches on your fork as described in Updating your remote and local main branches (page 31).

3. Check out your feature branch, and merge the main branch into your feature branch

```
git checkout feature/[POD name]
git merge main
```

4. Resolve any conflicts that occur from the merge

5. Add the updated files to the staging area

```
git add file1
git add file2
...
```

6. Push the branch updates to your remote fork

```
git push -u origin feature/[POD name]
```

Reverting commits

If you want to revert to the commit(s) before you pulled in updates:

1. Find the commit hash(es) with the updates, in your git log

```
git log
```

or consult the commit log in the web interface

2. Revert each commit in order from newest to oldest

```
git revert <newer commit hash>
git revert <older commit hash>
```

3. Push the updates to the remote branch

```
git push origin feature/[POD name]
```

1.9.11 Set up SSH with GitHub

- You have to generate an SSH key[127] and add it[128] to your GitHub account. This will save you from having to re-enter your GitHub username and password every time you interact with their servers.
- When generating the SSH key, you'll be asked to pick a passphrase (i.e., password).
- The following instructions assume you've generated an SSH key. If you're using manual authentication instead, replace the "`git@github.com:`" addresses in what follows with "`https://github.com/`".

1.9.12 Some online git resources

If you are new to git and unfamiliar with many of the terminologies, Dangit, Git?![129] provides solutions in plain English to many common mistakes people have made.

There are many comprehensive online git tutorials, such as:

- The official git tutorial[130].
- A more verbose introduction[131] to the ideas behind git and version control.
- A still more detailed walkthrough[132], assuming no prior knowledge.

[127] https://help.github.com/en/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent
[128] https://help.github.com/en/articles/adding-a-new-ssh-key-to-your-github-account
[129] https://dangitgit.com/
[130] https://git-scm.com/docs/gittutorial
[131] https://www.atlassian.com/git/tutorials/what-is-version-control
[132] http://swcarpentry.github.io/git-novice/

1.9.13 Git Tips and Tricks

- If you are unfamiliar with git and want to practice with the commands listed here, we recommend you to create an additional feature branch just for this. Remember: your changes will not affect NOAA's repo until you've submitted a pull request through the GitHub webpage and accepted by the lead-team programmer.
- GUI applications can be helpful when trying to resolve merge conflicts.Git packages for IDEs such as VSCode and Eclipse often include tools for merge conflict resolution. You can also install free versions of merge-conflict tools like P4merge[133] and Sublime merge[134].
- If you encounter problems during practice, you can first try looking for plain English instructions to fix the situation at Dangit, Git?![135].
- A useful command is `git status` to remind you what branch you're on and changes you've made (but have not committed yet).
- `git branch -a` lists all branches with `*` indicating the branch you're on.
- Push your changes to your remote fork often (at least daily) even if your changes aren't "clean", or you are in the middle of a task. Your commit history does not need to look like a polished document, and nobody is judging your coding prowess by your development branch. Frequently pushing to your remote branch ensures that you have an easily accessible recent snapshot of your code in the event that your system goes down, or you go crazy with `rm -f *`.
- A commit creates a snapshot of the code into the history in your local repo.
  - The snapshot will exist until you intentionally delete it (after confirming a warning message). You can always revert to a previous snapshot.
  - Don't commit code that you know is buggy or non-functional!
  - You'll be asked to enter a commit message. Good commit messages are key to making the project's history useful.
  - Write in present tense describing what the commit, when applied, does to the code – not what you did to the code.
  - Messages should start with a brief, one-line summary, less than 80 characters. If this is too short, you may want to consider entering your changes as multiple commits.
- Good commit messages are key to making the project's history useful. To make this easier, instead of using the `-m` flag, To provide further information, add a blank line after the summary and wrap text to 72 columns if your editor supports it (this makes things display nicer on some tools). Here's an example[136].
- To configure git to launch your text editor of choice: `git config --global core.editor "<command string to launch your editor>"`.
- To set your email: `git config --global user.email "myemail@somedomain.com"` You can use the masked email github provides if you don't want your work email included in the commit log message. The masked email address is located in the Primary email address section under Settings>emails.

[133] https://www.perforce.com/products/helix-core-apps/merge-diff-tool-p4merge

[134] https://www.sublimemerge.com/

[135] https://dangitgit.com/

[136] https://github.com/NOAA-GFDL/MDTF-diagnostics/commit/225b29f30872b60621a5f1c55a9f75bbcf192e0b

- When the feature branch is no longer needed, delete the branch locally with `git branch -d feature/<my_feature_name>`.

  If you pushed the feature branch to your fork, you can delete it remotely with `git push --delete origin feature/<my_feature_name>`. - Remember that branches in git are just pointers to a particular commit, so by deleting a branch you don't lose any history.

- If you want to let others work on your feature, push its branch to your GitHub fork with `git push -u origin feature/<my_feature_name>`.
- For additional ways to undo changes in your branch, see How to undo (almost) anything with Git[137].

[137] https://github.blog/2015-06-08-how-to-undo-almost-anything-with-git/

CHAPTER

TWO

FRAMEWORK REFERENCE

2.1 Diagnostic settings file format

The settings file is how your diagnostic tells the framework what it needs to run, in terms of software and model data.

Each diagnostic must contain a text file named `settings.jsonc` in the JSON[138] format, with the addition that any text to the right of `//` is treated as a comment and ignored (sometimes called the "JSONC" format).

2.1.1 Brief summary of JSON

We'll briefly summarize subset of JSON syntax used in this configuration file. The file's JSON expressions are built up out of items, which may be either

1. a boolean, taking one of the values `true` or `false` (lower-case, with no quotes).
2. a number (integer or floating-point).
3. a case-sensitive string, which must be delimited by double quotes.

In addition, for the purposes of the configuration file we define

4. a "time duration": this is a string specifying a time span, used e.g. to describe how frequently data is sampled. It consists of an optional integer (if omitted, the integer is assumed to be 1) and a units string which is one of `hr`, `day`, `mon`, `yr` or `fx`. `fx` is used where appropriate to denote time-independent data. Common synonyms for these units are also recognized (e.g. `monthly`, `month`, `months`, `mo` for `mon`, `static` for `fx`, etc.)

   In addition, the string `"any"` may be used to signify that any value is acceptable.
5. a "CF unit": this is a string describing the units of a physical quantity, following the syntax[139] of the UDUNITS2[140] library. `1` should be used for dimensionless quantities.

Items are combined in compound expressions of two types:

6. arrays, which are one-dimensional ordered lists delimited with square brackets. Entries can be of any type, e.g. `[true, 1, "two"]`.

[138] https://en.wikipedia.org/wiki/JSON#Data_types_and_syntax

[139] https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax

[140] https://www.unidata.ucar.edu/software/udunits/udunits-current/doc/udunits/udunits2.html

7. objects, which are un-ordered lists of key:value pairs separated by colons and delimited with curly brackets. Keys must be strings and must all be unique within the object, while values may be any expression, e.g. `{"red": 0, "green": false, "blue": "bagels"}`.

Compound expressions may be nested within each other to an arbitrary depth.

2.1.2 File organization

```
{
  "settings" : {
    <...properties describing the diagnostic..>
  },
  "data" : {
    <...properties for all requested model data...>
  },
  "dimensions" : {
    "my_first_dimension": {
      <...properties describing this dimension...>
    },
    "my_second_dimension": {
      <...properties describing this dimension...>
    },
    ...
  },
  "varlist" : {
    "my_first_variable": {
      <...properties describing this variable...>
    },
    "my_second_variable": {
      <...properties describing this variable...>
    },
    ...
  }
}
```

At the top level, the settings file is an object (page 37) containing four required entries, described in detail below.

- settings (page 39): properties that label the diagnostic and describe its runtime requirements.
- data (page 40): properties that apply to all the data your diagnostic is requesting.
- dimensions (page 42): properties that apply to the dimensions (in netCDF[141] terminology) of the model data. Each distinct dimension (coordinate axis) of the data being requested should be listed as a separate entry here.
- varlist (page 45): properties that describe the individual variables your diagnostic operates on. Each variable should be listed as a separate entry here.

[141] https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcDims.html

2.1.3 Settings section

This section is an object (page 37) containing properties that label the diagnostic and describe its runtime requirements.

Example

```
"settings" : {
  "long_name" : "Effect of X on Y diagnostic",
  "driver" : "my_script.py",
  "realm" : ["atmos", "ocean"],
  "runtime_requirements": {
    "python": ["numpy", "matplotlib", "netCDF4", "cartopy"],
    "ncl": ["contributed", "gsn_code", "gsn_csm"]
  },
  "pod_env_vars" : {
    // RES: Spatial Resolution (degree) for Obs Data (0.25, 0.50, 1.00).
    "RES": "1.00"
  }
}
```

Diagnostic description

`long_name`:

String, required. Human-readable display name of your diagnostic. This is the text used to describe your diagnostic on the top-level index.html page. It should be in sentence case (capitalize first word and proper nouns only) and omit any punctuation at the end.

`driver`:

String, required. Filename of the top-level driver script the framework should call to run your diagnostic's analysis.

`realm`:

String or array (page 37) (list) of strings, required. One of the eight CMIP6 modeling realms (aerosol, atmos, atmosChem, land, landIce, ocean, ocnBgchem, seaIce) describing what data your diagnostic uses. If your diagnostic uses data from multiple realms, list them in an array (e.g. `["atmos", "ocean"]`). This information doesn't affect how the framework fetches model data for your diagnostic: it's provided to give the user a shortcut to say, e.g., "run all the atmos diagnostics on this output."

Diagnostic runtime

`runtime_requirements`:

object (page 37), required. Programs your diagnostic needs to run (for example, scripting language interpreters) and any third-party libraries needed in those languages. Each executable should be listed in a separate key-value pair:

- The key is the name of the required executable, e.g. languages such as "python[142]" or "ncl[143]"

[142] https://www.python.org/

[143] https://www.ncl.ucar.edu/

etc. but also any utilities such as “ncks[144]”, “cdo[145]”, etc.

- The value corresponding to each key is an array (page 37) (list) of strings, which are names of third-party libraries in that language that your diagnostic needs. You do not need to list standard libraries or scripts that are provided in a standard installation of your language: eg, in python, you need to list numpy[146] but not math[147]. If no third-party libraries are needed, the value should be an empty list.

In the future we plan to offer the capability to request specific versions[148]. For now, please communicate your diagnostic’s version requirements to the MDTF organizers.

`pod_env_vars`:

object (page 37), optional. Names and values of shell environment variables used by your diagnostic, in addition to those supplied by the framework. The user can’t change these at runtime, but this can be used to set site-specific installation settings for your diagnostic (eg, switching between low- and high-resolution observational data depending on what the user has chosen to download). Note that environment variable values must be provided as strings.

2.1.4 Data section

This section is an object (page 37) containing properties that apply to all the data your diagnostic is requesting.

Example

```
"data": {
  "format": "netcdf4_classic",
  "rename_dimensions": false,
  "rename_variables": false,
  "multi_file_ok": true,
  "frequency": "3hr",
  "min_frequency": "1hr",
  "max_frequency": "6hr",
  "min_duration": "5yr",
  "max_duration": "any"
}
```

[144] http://nco.sourceforge.net/

[145] https://code.mpimet.mpg.de/projects/cdo

[146] https://numpy.org/

[147] https://docs.python.org/3/library/math.html

[148] https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/pkg-specs.html#package-match-specifications

Example

`format`:

String. Optional: assumed `"any_netcdf_classic"` if not specified. Specifies the format(s) of model data your diagnostic is able to read. As of this writing, the framework only supports retrieval of netCDF formats, so only the following values are allowed:

- `"any_netcdf"` includes all of:
  - `"any_netcdf3"` includes all of:
    - `"netcdf3_classic"` (CDF-1, files restricted to < 2 Gb)
    - `"netcdf3_64bit_offset"` (CDF-2)
    - `"netcdf3_64bit_data"` (CDF-5)
  - `"any_netcdf4"` includes all of:
    - `"netcdf4_classic"`
    - `"netcdf4"`
- `"any_netcdf_classic"` includes all the above except `"netcdf4"` (classic data model only).

See the netCDF FAQ[149] for information on the distinctions. Any recent version of a supported language for diagnostics with netCDF support will be able to read all of these. However, the extended features of the `"netcdf4"` data model are not commonly used in practice and currently only supported at a beta level in NCL, which is why we've chosen `"any_netcdf_classic"` as the default.

`rename_dimensions`:

Boolean. Optional: assumed `false` if not specified. If set to `true`, the framework will change the name of all dimensions (page 42) in the model data from the model's native value to the string specified in the `name` property for that dimension. If set to `false`, the diagnostic is responsible for reading dimension names from the environment variable. See the environment variable documentation (page 48) for details on how these names are provided.

`rename_variables`:

Boolean. Optional: assumed `false` if not specified. If set to `true`, the framework will change the name of all variables (page 45) in the model data from the model's native value to the string specified in the `name` property for that variable. If set to `false`, the diagnostic is responsible for reading dimension names from the environment variable. See the environment variable documentation (page 48) for details on how these names are provided.

`multi_file_ok`:

Boolean. Optional: assumed `false` if not specified. If set to `true`, the diagnostic is signalling that it's able to accept data for a single variable that may be spread out in multiple files, to be aggregated along the time dimension (e.g. through the use of xarray[150].) Aggregation along the time dimension is the only type of aggregation the diagnostic will need to consider.

If `false`, the framework will ensure all data for a single variable is presented as a single netCDF file. This may lead to large file sizes if your diagnostic uses high-frequency data, in which case you should consider setting a limit via `max_duration`.

[149] https://www.unidata.ucar.edu/software/netcdf/docs/faq.html#How-many-netCDF-formats-are-there-and-what-are-the-differences-among-them
[150] http://xarray.pydata.org/en/stable/generated/xarray.open_mfdataset.html

`min_duration`, `max_duration`:

Time durations (page 37). Optional: assumed `"any"` if not specified. Set minimum and maximum length of the analysis period for which the diagnostic should be run: this overrides any choices the user makes at runtime. Some example uses of this setting are:

- If your diagnostic uses low-frequency (e.g. seasonal) data, you may want to set `min_duration` to ensure the sample size will be large enough for your results to be statistically meaningful.
- On the other hand, if your diagnostic uses high-frequency (e.g. hourly) data, you may want to set `max_duration` to prevent the framework from attempting to download a large volume of data for your diagnostic if the framework is called with a multi-decadal analysis period.

The following properties can optionally be set individually for each variable in the varlist section (page 45). If so, they will override the global settings given here.

`dimensions_ordered`:

Boolean. Optional: assumed `false` if not specified. If set to `true`, the framework will ensure that the dimensions of each variable's array are given in the same order as listed in `dimensions`. If set to false, your diagnostic is responsible for handling arbitrary dimension orders: e.g. it should not assume that 3D data will be presented as (time, lat, lon).

`frequency`, `min_frequency`, `max_frequency`:

Time durations (page 37). Time frequency at which the data is provided. Either `frequency` or the min/max pair, or both, is required:

- If only `frequency` is provided, the framework will attempt to obtain data at that frequency. If that's not available from the data source, your diagnostic will not run.
- If the min/max pair is provided, the diagnostic must be capable of using data at any frequency within that range (inclusive). The diagnostic is responsible for determining the frequency from the data file itself if this option is used.
- If all three properties are set, the framework will first attempt to find data at `frequency`. If that's not available, it will try data within the min/max range, so your code must be able to handle this possibility.

2.1.5 Dimensions section

This section is an object (page 37) contains properties that apply to the dimensions of model data. "Dimensions" are meant in the sense of the netCDF data model[151], and "coordinate dimensions" in the CF conventions: informally, they are "coordinate axes" holding the values of independent variables that the dependent variables are sampled at.

All dimensions (page 47) and scalar coordinates (page 47) referenced by variables in the varlist section must have an entry in this section. If two variables reference the same dimension, they will be sampled on the same set of spatial values. Different time values are specified with the `frequency` attribute on varlist entries.

Note that the framework currently only supports the (simplest and most common) "independent axes" case of the CF conventions[152]. In particular, the framework only deals with data on lat-lon grids.

[151] https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcDims.html

[152] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#_independent_latitude_longitude_vertical_and_time_axes

Example

```
"dimensions": {
  "lat": {
      "standard_name": "latitude",
      "units": "degrees_N",
      "range": [-90, 90],
      "need_bounds": false
  },
  "lon": {
      "standard_name": "longitude",
      "units": "degrees_E",
      "range": [-180, 180],
      "need_bounds": false
  },
  "plev": {
      "standard_name": "air_pressure",
      "units": "hPa",
      "positive": "down",
      "need_bounds": false
  },
  "time": {
      "standard_name": "time",
      "units": "days",
      "calendar": "noleap",
      "need_bounds": false
  }
}
```

Latitude and Longitude

`standard_name`:

Required, string. Must be `"latitude"` and `"longitude"`, respectively.

`units`:

Optional, a CFunit (page 37). Units the diagnostic expects the dimension to be in. Currently the framework only supports decimal `degrees_north` and `degrees_east`, respectively.

`range`:

Array (page 37) (list) of two numbers. Optional. If given, specifies the range of values the diagnostic expects this dimension to take. For example, `"range": [-180, 180]` for longitude will have the first entry of the longitude variable in each data file be near -180 degrees (not exactly -180, because dimension values are cell midpoints), and the last entry near +180 degrees.

`need_bounds`:

Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ensure that bounds are supplied for this dimension, in addition to its midpoint values, following the CF conventions[153]: the `bounds` attribute of this dimension will be set to the name of another netCDF variable containing the bounds information.

[153] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries

`axis`:

String, optional. Assumed to be `Y` and `X` respectively if omitted, or if `standard_name` is `"latitude"` or `"longitude"`. Included here to enable future support for non-lat-lon horizontal coordinates.

Time

`standard_name`:

Required. Must be `"time"`.

`units`:

String. Optional, defaults to "day". Units the diagnostic expects the dimension to be in. Currently the diagnostic only supports time axes of the form "<units> since <reference data>", and the value given here is interpreted in this sense (e.g. settings this to "day" would accommodate a dimension of the form "[decimal] days since 1850-01-01".)

`calendar`:

String, Optional. One of the CF convention calendars[154] or the string `"any"`. Defaults to "any" if not given. Calendar convention used by your diagnostic. Only affects the number of days per month.

`need_bounds`:

Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ensure that bounds are supplied for this dimension, in addition to its midpoint values, following the CF conventions[155]: the `bounds` attribute of this dimension will be set to the name of another netCDF variable containing the bounds information.

`axis`:

String, optional. Assumed to be `T` if omitted or provided.

Z axis (height/depth, pressure, ...)

`standard_name`:

Required, string. Standard name[156] of the variable as defined by the CF conventions[157], or a commonly used synonym as employed in the CMIP6 MIP tables.

`units`:

Optional, a CFunit (page 37). Units the diagnostic expects the dimension to be in. If not provided, the framework will assume CF convention canonical units[158].

`positive`:

String, required. Must be `"up"` or `"down"`, according to the CF conventions[159]. A pressure axis is always `"down"` (increasing values are closer to the center of the earth), but this is not set automatically.

`need_bounds`:

Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ensure that bounds are supplied for this dimension, in addition to its midpoint values, following the CF conventions[160]:

[154] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#calendar
[155] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries
[156] http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html
[157] http://cfconventions.org/
[158] http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html
[159] http://cfconventions.org/faq.html#vertical_coords_positive_attribute
[160] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries

the `bounds` attribute of this dimension will be set to the name of another netCDF variable containing the bounds information.

`axis`:
String, optional. Assumed to be `Z` if omitted or provided.

Other dimensions (wavelength, ...)

`standard_name`:
Required, string. Standard name[161] of the variable as defined by the CF conventions[162], or a commonly used synonym as employed in the CMIP6 MIP tables.

`units`:
Optional, a CFunit (page 37). Units the diagnostic expects the dimension to be in. If not provided, the framework will assume CF convention canonical units[163].

`need_bounds`:
Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ensure that bounds are supplied for this dimension, in addition to its midpoint values, following the CF conventions[164]: the `bounds` attribute of this dimension will be set to the name of another netCDF variable containing the bounds information.

`modifier`: String, Optional. Used to distinguish variables that are defined on a vertical level that is not a pressure level (e.g., 2-meter temperature) from variables that are defined on pressure levels. Modfiers are defined in data/modifiers.jsonc. MDTF-diagnostics currently supports atmos_height.

2.1.6 Varlist section

This section is an object (page 37) contains properties that apply to the model variables your diagnostic needs for its analysis. "Dimensions" are meant in the sense of the netCDF data model[165]: informally, they are the "dependent variables" whose values are being computed as a function of the values stored in the dimensions.

Note that this includes "auxiliary coordinates" in the CF conventions terminology and similar ancillary information. If your diagnostic needs, eg, cell areas or volumes, orography data, etc., each piece of data should be listed as a separate entry here, even if their use is conventionally implied by the use of other variables.

Each entry corresponds to a distinct data file (or set of files, if `multi_file_ok` is `true`) downloaded by the framework. If your framework needs the same physical quantity sampled with different properties (e.g. slices of a variable at multiple pressure levels), specify them as multiple entries.

[161] http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html
[162] http://cfconventions.org/
[163] http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html
[164] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries
[165] https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcVars.html

Varlist entry example

```
"u500": {
    "standard_name": "eastward_wind",
    "path_variable": "U500_FILE",
    "units": "m s-1",
    "dimensions" : ["time", "lat", "lon"],
    "dimensions_ordered": true,
    "scalar_coordinates": {"pressure": 500},
    "requirement": "optional",
    "alternates": ["another_variable_name", "a_third_variable_name"]
}
```

Varlist entry properties

The key in a varlist key-value pair is the name your diagnostic uses to refer to this variable (and must be unique). The value of the key-value pair is an object (page 37) containing properties specific to that variable:

`standard_name`:

String, required. Standard name[166] of the variable as defined by the CF conventions[167], or a commonly used synonym as employed in the CMIP6 MIP tables (e.g. "ua" instead of "eastward_wind").

`path_variable`:

String, optional but recommended. Name of the shell environment variable the framework will set with the location of this data. This is the only currently supported method for communicating the location of model data to your diagnostic. If omitted, set to `<key>_FILE`, where `<key>` is the key to the varlist entry (case-sensitive). See the environment variable documentation (page 48) for details.

- If `multi_file_ok` is `false`, `<path_variable>` will be set to the absolute path to the netcdf file containing this variable's data.
- If `multi_file_ok` is `true`, `<path_variable>` will be a single path or a colon-separated list of paths to the files containing this data. Files will be listed in order of the dates of their contents.
- If the variable is listed as `"optional"` or `"alternate"` or has `alternate` variables listed, `<path_variable>` will be defined but set to the empty string if the framework couldn't obtain this data from the data source. Your diagnostic should test for this possibility. (If the variable is required but the framework couldn't obtain data, an error will be logged and your diagnostic will not run).

`use_exact_name`:

Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ignore the model's naming conventions and only look for a variable with a name matching the key of this entry, regardless of what model or data source the framework is using. The only use case for this setting is to give diagnostics the ability to request data that falls outside the CF conventions: in general, you should rely on the framework to translate CF standard names to the native field names of the model being analyzed.

`units`:

Optional, a CFunit (page 37). Units the diagnostic expects the variable to be in. If not provided, the

[166] http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html

[167] http://cfconventions.org/

framework will assume CF convention canonical units[168].

`dimensions`:

Required. List of strings, which must be selected the keys of entries in the dimensions (page 42) section. Dimensions of the array containing the variable's data. Note that the framework will not reorder dimensions (transpose) unless `dimensions_ordered` is additionally set to `true`.

`dimensions_ordered`:

Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ensure that the dimensions of this variable's array are given in the same order as listed in `dimensions`. If set to false, your diagnostic is responsible for handling arbitrary dimension orders: e.g. it should not assume that 3D data will be presented as (time, lat, lon). If given here, overrides the values set globally in the `data` section (see description (page 42) there).

`scalar_coordinates`:

object (page 37), optional. This implements what the CF conventions refer to as "scalar coordinates[169]", with the use case here being the ability to request slices of higher-dimensional data. For example, the snippet at the beginning of this section shows how to request the u component of wind velocity on a 500 mb pressure level.

- keys are the key (name) of an entry in the dimensions (page 42) section.
- values are a single number (integer or floating-point) corresponding to the value of the slice to extract. Units of this number are taken to be the `units` property of the dimension named as the key.

In order to request multiple slices (e.g. wind velocity on multiple pressure levels, with each level saved to a different file), create one varlist entry per slice.

`frequency`, `min_frequency`, `max_frequency`:

Time durations (page 37). Optional. Time frequency at which the variable's data is provided. If given here, overrides the values set globally in the `data` section (see description (page 42) there).

`requirement`:

String. Optional: assumed `"required"` if not specified. One of three values:

- `"required"`: variable is necessary for the diagnostic's calculations. If the data source doesn't provide the variable (at the requested frequency, etc., for the user-specified analysis period) the framework will not run the diagnostic, but will instead log an error message explaining that the lack of this data was at fault.
- `"optional"`: variable will be supplied to the diagnostic if provided by the data source. If not available, the diagnostic will still run, and the `path_variable` for this variable will be set to the empty string. The diagnostic is responsible for testing the environment variable for the existence of all optional variables.
- `"alternate"`: variable is specified as an alternate source of data for some other variable (see next property). The framework will only query the data source for this variable if it's unable to obtain one of the other variables that list it as an alternate.

`alternates`:

Array (page 37) (list) of strings, which must be keys (names) of other variables. Optional: if provided,

[168] http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html

[169] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#scalar-coordinate-variables

specifies an alternative method for obtaining needed data if this variable isn't provided by the data source.

- If the data source provides this variable (at the requested frequency, etc., for the user-specified analysis period), this property is ignored.
- If this variable isn't available as requested, the framework will query the data source for all of the variables listed in this property. If all of the alternate variables are available, the diagnostic will be run; if any are missing it will be skipped. Note that, as currently implemented, only one set of alternates may be given (no "plan B", "plan C", etc.)

2.2 MDTF Environment variables

This page describes the environment variables that the framework will set for your diagnostic when it's run.

2.2.1 Overview

The MDTF framework can be viewed as a "wrapper" for your code that handles data fetching and munging. Your code communicates with this wrapper in two ways:

- The settings file (page 15) is where your code talks to the framework: when you write your code, you document what model data your code uses (not covered on this page, follow the link for details).
- When your code is run, the framework talks to it by setting shell environment variables[170] containing paths to the data files and other information specific to the run. The framework communicates all runtime information this way: this is in order to 1) pass information in a language-independent way, and 2) to make writing diagnostics easier (you don't need to parse command-line settings).

Note that environment variables are always strings. Your script will need to cast non-text data to the appropriate type (e.g. the bounds of the analysis time period, `FIRSTYR`, `LASTYR`, will need to be converted to integers.)

Also note that names of environment variables are case-sensitive.

2.2.2 Paths

`OBS_DATA`:

Path to the top-level directory containing any observational or reference data you've provided as the author of your diagnostic. Any data your diagnostic uses that doesn't come from the model being analyzed should go here (i.e., you supply it to the framework maintainers, they host it, and the user downloads it when they install the framework). The framework will ensure this is copied to a local filesystem when your diagnostic is run, but this directory should be treated as read-only.

`POD_HOME`:

Path to the top-level directory containing your diagnostic's source code. This will be of the form `../MDTF-diagnostics/diagnostics/<your POD's name>`. This can be used to call sub-scripts from your diagnostic's driver script. This directory should be treated as read-only.

[170] https://en.wikipedia.org/wiki/Environment_variable

WK_DIR:

Path to your diagnostic's working directory, which is where all output data should be written (as well as any temporary files).

The framework creates the following subdirectories within this directory:

- $WK_DIR/obs/PS and $WK_DIR/model/PS: All output plots produced by your diagnostic should be written to one of these two directories. Only files in these locations will be converted to bitmaps for HTML output.
- $WK_DIR/obs/netCDF and $WK_DIR/model/netCDF: Any output data files your diagnostic wants to make available to the user should be saved to one of these two directories.

2.2.3 Model run information

CASENAME:

User-provided label describing the run of model data being analyzed.

FIRSTYR, LASTYR:

Four-digit years describing the analysis period.

2.2.4 Locations of model data files

These are set depending on the data your diagnostic requests in its settings file (page 15). Refer to the examples below if you're unfamiliar with how that file is organized.

Each variable listed in the varlist section of the settings file must specify a path_variable property. The value you enter there will be used as the name of an environment variable, and the framework will set the value of that environment variable to the absolute path to the file containing data for that variable.

From a diagnostic writer's point of view, this means all you need to do here is replace paths to input data that are hard-coded or passed from the command line with calls to read the value of the corresponding environment variable.

- If the framework was not able to obtain the variable from the data source (at the requested frequency, etc., for the user-specified analysis period), this variable will be set equal to the empty string. Your diagnostic is responsible for testing for this possibility for all variables that are listed as optional or have alternates listed (if a required variable without alternates isn't found, your diagnostic won't be run.)
- If multi_file_ok is set to true in the settings file, this environment variable may be a list of paths to multiple files in chronological order, separated by colons. For example, /dir/precip_1980_1989.nc:/dir/precip_1990_1999.nc:/dir/precip_2000_2009.nc for an analysis period of 1980-2009.

2.2.5 Names of variables and dimensions

These are set depending on the data your diagnostic requests in its settings file (page 15). Refer to the examples below if you're unfamiliar with how that file is organized.

For each dimension:

> If <key> is the name of the key labeling the key:value entry for this dimension, the framework will set an environment variable named `<key>_coord` equal to the name that dimension has in the data files it's providing.
>
> - If `rename_dimensions` is set to `true` in the settings file, this will always be equal to <key>. If `rename_dimensions` is `false`, this will be whatever the model or data source's native name for this dimension is, and your diagnostic should read the name from this variable. Your diagnostic should only use hard-coded names for dimensions if `rename_dimensions` is set to `true` in its settings file (page 37).
>
> If the data source has provided (one-dimensional) bounds for this dimension, the name of the netCDF variable containing those bounds will be set in an environment variable named `<key>_bnds`. If bounds are not provided, this will be set to the empty string. Note that multidimensional boundaries (e.g. for horizontal cells) should be listed as separate entries in the varlist section.

For each variable:

> If <key> be the name of the key labeling the key:value entry for this variable in the varlist section, the framework will set an environment variable named `<key>_var` equal to the name that variable has in the data files it's providing.
>
> - If `rename_variables` is set to `true` in the settings file, this will always be equal to <key>. If `rename_variables` is `false`, this will be whatever the model or data source's native name for this variable is, and your diagnostic should read the name from this variable. Your diagnostic should only use hard-coded names for variables if `rename_variables` is set to `true` in its settings file (page 37).

2.2.6 Simple example

We only give the relevant parts of the settings file (page 37) below.

```
"data": {
  "rename_dimensions": false,
  "rename_variables": false,
  "multi_file_ok": false,
  ...
},
"dimensions": {
  "lat": {
    "standard_name": "latitude",
    ...
  },
  "lon": {
    "standard_name": "longitude",
    ...
  },
  "time": {
```

(continues on next page)

(continued from previous page)

```
    "standard_name": "time",
    ...
  }
},
"varlist": {
  "pr": {
    "standard_name": "precipitation_flux",
    "path_variable": "PR_FILE"
  }
}
```

The framework will set the following environment variables:

1. `lat_coord`: Name of the latitude dimension in the model's native format (because `rename_dimensions` is false).
2. `lon_coord`: Name of the longitude dimension in the model's native format (because `rename_dimensions` is false).
3. `time_coord`: Name of the time dimension in the model's native format (because `rename_dimensions` is false).
4. `pr_var`: Name of the precipitation variable in the model's native format (because `rename_variables` is false).
5. `PR_FILE`: Absolute path to the file containing `pr` data, e.g. `/dir/precip.nc`.

2.2.7 More complex example

Let's elaborate on the previous example, and assume that the diagnostic is being called on model that provides precipitation_flux but not convective_precipitation_flux.

```
"data": {
  "rename_dimensions": true,
  "rename_variables": false,
  "multi_file_ok": true,
  ...
},
"dimensions": {
  "lat": {
    "standard_name": "latitude",
    ...
  },
  "lon": {
    "standard_name": "longitude",
    ...
  },
  "time": {
    "standard_name": "time",
    ...
  }
},
```

(continues on next page)

(continued from previous page)

```
"varlist": {
  "prc": {
    "standard_name": "convective_precipitation_flux",
    "path_variable": "PRC_FILE",
    "alternates": ["pr"]
  },
  "pr": {
    "standard_name": "precipitation_flux",
    "path_variable": "PR_FILE"
  }
}
```

Comparing this with the previous example:

- `lat_coord`, `lon_coord` and `time_coord` will be set to "lat", "lon" and "time", respectively, because `rename_dimensions` is true. The framework will have renamed these dimensions to have these names in all data files provided to the diagnostic.
- `prc_var` and `pr_var` will be set to the model's native names for these variables. Names for all variables are always set, regardless of which variables are available from the data source.
- In this example, `PRC_FILE` will be set to `''`, the empty string, because it wasn't found.
- `PR_FILE` will be set to `/dir/precip_1980_1989.nc:/dir/precip_1990_1999.nc:/dir/precip_2000_2009.nc`, because `multi_file_ok` was set to `true`.

CHAPTER

THREE

ACKNOWLEDGEMENTS

Development of this code framework for process-oriented diagnostics was supported by the National Oceanic and Atmospheric Administration[171] (NOAA) Climate Program Office Modeling, Analysis, Predictions and Projections[172] (MAPP) Program (grant # NA18OAR4310280). Additional support was provided by University of California Los Angeles[173], the Geophysical Fluid Dynamics Laboratory[174], the National Center for Atmospheric Research[175], Colorado State University[176], Lawrence Livermore National Laboratory[177] and the US Department of Energy[178].

Many of the process-oriented diagnostics modules (PODs) were contributed by members of the NOAA Model Diagnostics Task Force[179] under MAPP support. Statements, findings or recommendations in these documents do not necessarily reflect the views of NOAA or the US Department of Commerce.

3.1 Disclaimer

This repository is a scientific product and is not an official communication of the National Oceanic and Atmospheric Administration, or the United States Department of Commerce. All NOAA GitHub project code is provided on an ‘as is’ basis and the user assumes responsibility for its use. Any claims against the Department of Commerce or Department of Commerce bureaus stemming from the use of this GitHub project will be governed by all applicable Federal law. Any reference to specific commercial products, processes, or services by service mark, trademark, manufacturer, or otherwise, does not constitute or imply their endorsement, recommendation or favoring by the Department of Commerce. The Department of Commerce seal and logo, or the seal and logo of a DOC bureau, shall not be used in any manner to imply endorsement of any commercial product or activity by DOC or the United States Government.

[171] https://www.noaa.gov/

[172] https://cpo.noaa.gov/Meet-the-Divisions/Earth-System-Science-and-Modeling/MAPP

[173] https://www.ucla.edu/

[174] https://www.gfdl.noaa.gov/

[175] https://ncar.ucar.edu/

[176] https://www.colostate.edu/

[177] https://www.llnl.gov/

[178] https://www.energy.gov/

[179] https://cpo.noaa.gov/Meet-the-Divisions/Earth-System-Science-and-Modeling/MAPP/MAPP-Task-Forces/Model-Diagnostics-Task-Force