Feature Specification - Migrating from CDAT

Target Release	**TBD (v3.0.0?)**
Task Timeline	~~**End 2023**~~ **UPDATE 9/26/23 - We are far behind schedule due to limited staff and resource constraints. We are extending the timeline for this task to the end of 2024.**
GitHub Issue(s)	https://github.com/E3SM-Project/e3sm_diags/discussions/607 https://github.com/E3SM-Project/e3sm_diags/issues/626 https://github.com/E3SM-Project/e3sm_diags/issues/628
Document Status	**Task In Progress**
Document Owner	Tom Vo
Developers/Collaborators	Tom Vo, Jill Zhang, Ryan Forsyth, Chris Golaz
QA	

This is the feature specification for refactoring CDAT code, which outlines its purpose, assumptions and constraints, algorithm design, and test cases.

Objective

CDAT's EOL is planned for December/2023, which means actively developed and used packages must migrate away from CDAT to other libraries. For information, refer to:
FY23 ISCP Proposal - Modernizing Open-Source LLNL Developed Tools for Robust Climate Data A…

Assumptions

Technical Constraints

- Limited FTEs and funding available to work on this task
    - Many of us work on several efforts that take a significant portion of our time.
    - We need to find ways to distribute refactoring work and squeeze in time efficiently. This means having measurable progress through iterative development cycles
- Limited knowledge of codebase
    - There is an overhead with learning large codebases

    - Potential difficulties understanding code from past developers. One of the original developers (Zeshawn) is no longer on the project and he developed a lot of the core functionality which has since been duplicated or extended over the past 4 years.
- Adding new features using the legacy software architecture
    - This will make refactoring difficult because we have to refactor new features too (adds to the technical debt)

Software Architecture Challenges

- Each diagnostic has its own set of modules defined in driver/, parameter/, plot/, and viewer/, which all vary in complexity
    - Not a generally scalable practice as new diagnostics are added over time.
    - This makes refactoring and unit testing challenging.
- Majority of the codebase is not covered by unit tests
    - Refactoring without unit tests can easily introduce undesired regressions in outputs, side-effects, and bugs
    - ***Have to be extra careful while refactoring***
- Nested subclassing of CDAT modules
    - For example, CoreParameter subclasses CDPParameter and CoreParser subclasses CDPParser.
    - CoreParameter and CoreParser are then subclassed by diagnostic modules (e.g., area_mean_time_series_parser.py)
- ~~Can't step through the code using iPython debugger at the moment when using CDPParameter/CDParser bug – which is required for more efficient debugging and refactoring~~
    - ~~These CDAT classes subclass the Python standard lib `argparse`, which can easily introduce unexpected bugs~~ Resolved by #632
- Downstream e3sm_diags functions operate on CDAT classes
    - e.g., cdutil.tvariable.TransvientVariable, cdutil.fvariable.FileVariable
    - These functions also require refactoring to operate on another data structure like the xarray.DataArray or xarray.Dataset

Technical Overview

There are 3 phases for this refactoring effort: (1) Refactor how e3sm_diags is configured and run, (2) Refactor diagnostics sets individually (modules, tests, docs), (3) Phase 3: Clean up and documentation

Phase 1: Refactor How Diagnostics Are Configured and Ran – **Done**

Refactoring can be done at the module level since it is not coupled to specific diagnostic runs.

- ☑ ~~Replace cdp.cdp_parameter.CDPParameter (GitHub Issue)~~
- ☑ ~~Replace cdp.multiprocessing (GitHub comment and Issue)~~

- ☑ ~~Replace cdp.cdp_parser.CDPParser (GitHub Issue)~~

Phase 2: Refactor Diagnostic Sets – ***In Progress***

Overview for Refactoring

- We should dedicate one PR per diagnostic set to make the development cycle modular and iterative.
    - Run a regression on the master and development branch and compare changes
        - Compare plots, `.nc` files, metric checker
    - Write generalizable functions that can be shared between diagnostics
        - Adopt these functions for other diagnostics.
    - Write unit tests as we go
- Use this to help find alternative APIs to use: CDAT API Dependency Map

Components of a Set

1. Driver (*driver/*)
    a. Utilities (*driver/utilities/)*
        i. *dataset.py* – a centerpiece for reading in data as a class Dataset object
        ii. climo.py – computes climatology for a variable for the given season (xCDAT supports this)
        iii. *diurnal_cycle.py* – Computes the composite diurnal cycle for var for the given season
        iv. *general.py* – stores general utilities such as getting variable names, years, converting pressure coordinates to pressure levels, selecting regions, selecting points, regridding to lower resolution, masking, and saving variables and files to NetCDF.
    b. Each set driver has a `run_diags` function with modifications based on need
    c. Most set drivers have unique custom functionality which cannot be generalized (or easily at least)
    d. Some set drivers have a `create_metrics()` function
2. Plotter (*plot/*)
    a. Each set plot files has a `plot()` function that might have similar composition between one another, or not at all
    b. Most of the plot functions extract variables such as longitude/latitude and the associated data, which gets transformed into matplotlib plots
    c. I think we can keep a large majority of the plotting code as is and we just refactor the process of extracting the data needed for the plots. Plotting functions should work without significant refactoring if they operate on `numpy` arrays (which can easily be extracted from xarray objects).

3. Viewer (*viewer/cartopy*)
    a. Refactor cdp.viewer.OutputView(GitHub issue)
    b. There is root `default_viewer.py` file with a generic `create_viewer()` function for viewing sets
    c. Other sets have their own specific viewer module with a custom `create_viewer()` function
4. Utility functions used by diagnostic(s)
    a. **These will be identified as we work through refactoring**
5. Integration Tests (*tests/integration/*)

How to Refactor a Set

1. Setup a `run_e3sm_diags.py` script with a diagnostic run
    a. We have example scripts that we can use
2. Setup breakpoints for areas of code we are trying to refactor ("What we need to Refactor" above)
3. Execute the script with debugger
4. Use the Bubble Context mixed with Test-Driven Development to replace legacy code
    a. Create a new bubble of clean code (a namespace, a package …)
    b. Write baseline unit tests for new functions (TDD)
        i. https://philippe.bourgau.net/incremental-software-development-strategies-for-large-scale-refactoring-number-2-baby-steps/#mikado-method
    c. Rewrite a piece of legacy code in the bubble
    d. From the legacy code, delegate to the bubble
    e. Make unit tests pass
    f. Repeat until the legacy code is not used anymore
5. Make sure integration tests pass for that specific set

Regression Testing

1. Run `run_e3sm_diags.py` script on `main` and `dev` branches
2. Compare output plots and `.nc` files to make sure they align
3. A metrics checker is available to compare new results against a baseline, *but this works only for the lat-lon set.*
4. Existing integration tests should also continue passing

Phase 3: Clean Up and Documentation

1. Update examples
    a. In the root `examples/` directory, there are example run scripts that use the legacy way of configuring and running E3SM diagnostics. They should all be refactored to reflect the latest changes.

2. Update documentation

Overview of Sets

	Set	Core Components	Complexity	Original Contributor(s)
1	**lat_lon**	lat_lon_driver lat_lon_plot default_viewer	**Hard** – tackle this one first since it is the most complete set in terms of utilities it touches	Jill, Zeshawn
2	**lat_lon_land**	lat_lon_land_driver lat_lon_land_plot default_viewer	**Trivial -** subclasses lat_lon	Jill, Tom (reviewer)
3	**lat_lon_river**	lat_lon_river_driver lat_lon_river_plot default_viewer	**Trivial** - subclasses lat_lon	Jill
4	**polar**	polar_driver polar_plot default_viewer	**Easy** - driver is short, reuse logic from lat_lon set	Jill, Zeshawn
5	**area_mean_time_series**	area_mean_time_series_driver area_mean_time_series_plot area_mean_time_series_viewer	**Easy** - driver is short, reuse logic from lat_lon set	Jill, Zeshawn
6	**cosp_histogram**	cosp_histogram_driver cosp_histogram_plot default_viewer	**Easy** - driver is short, reuse logic from lat_lon set	Jill, Zeshawn
7	**zonal_mean_xy**	zonal_mean_xy_driver zonal_mean_xy_plot default_viewer	**Easy – reuse logic from lat_lon set**	Jill, Zeshawn
8	**annual_cycle_zonal_mean**	annual_cycle_zonal_mean_driver annual_cycle_zonal_plot annual_cycle_zonal_mean_viewer	**Easy** - Driver is pretty short and has unique `_create_annual_cycle` function	Jill
9	**zonal_mean_2d**	zonal_mean_2d_driver zonal_mean_2d_plot mean_2d_viewer	**Medium - reuse logic from lat_lon set**	Jill, Zeshawn

10	**zonal_mean_2d_stratosphere**	zonal_mean_2d_stratosphere_drive zonal_mean_2d_stratosphere_plot mean_2d_viewer	**Trival** - subclasses zonal_mean_2d	Jill, Tom (reviewer)
11	**meridional_mean_2d**	meridional_mean_2d_driver meridional_mean_2d_plot mean_2d_viewer	**Medium**	Jill, Zeshawn
12	**enso_diags**	enso_diags_driver enso_diags_plot enso_diags_viewer	**Hard** - Driver has unique nino and regression calculation functions	Ryan
13	**qbo**	qbo_driver qbo_plot qbo_viewer	**Hard** - Driver has many unique functions	Ryan
14	**streamflow**	streamflow_driver streamflow_plot streamflow_viewer	**Hard** - Driver has many unique functions	Ryan
15	**diurnal_cycle**	diurnal_cycle_driver diurnal_cycle_plot default_viewer	**Medium** - Driver uses unique `diurnal_cycle.py` functions	Jill
16	**arm_diags**	arm_diags_driver arm_diags_plot arm_diags_viewer	**Hard** - Driver has many unique functions	Jill
17	**tc_analysis**	tc_analysis_driver tc_analysis_plot tc_analysis_viewer	**Hard** - Driver has many unique functions	Jill
18	**aerosol_aeronet**	aerosol_aeronet_driver aerosol_aeronet_plot default_viewer	**Easy** - Driver is pretty short and has unique `interpolate_model_output_tobs_site`	Jill
19	**aerosol_budget**	aerosol_budget_driver aerosol_budget_plot aerosol_budget_viewer	**Medium** - Driver uses unique `calc_column_integral` and `global_integral` functions	Jill, Tom

Legend for “Complexity” column:

- Trivial – The code for the set is straightforward to refactor using other APIs, mainly just drop in replacements of CDAT code

- Easy – The set has a low amount of unique logic and has a minimal level of dependency on CDAT, mostly uses generalized utilities
- Medium – The set has a moderate amount of unique logic and has a moderate level of dependency on CDAT (e.g., utilities to manipulate CDAT objects like TransientVariables)
- Hard – The set has significant unique logic and significant level of dependency on CDAT (e.g., utilities to manipulate CDAT objects like TransientVariables

Progress and Milestones

- **USE THIS** **Project Kanban Board**

Open Questions

Question	Answer	Date Answered
Can we extend CDAT dependency maintenance specifically for E3SM Unified for FY24? We don't need to advertise this to users. cdat_info cdms2 cdtime cdutil e3sm_unified recipe		
Will it require a lot of work to update these CDAT packages to support Python 3.11?		

Out of Scope

List of functionalities or behaviors of this feature that have been discussed, but are out of scope or might be revisited in a later release.

-

Resources

- Incremental software development cycle
    - Review progress bi-weekly
    - https://philippe.bourgau.net/incremental-software-development-techniques-for-large-scale-refactorings/#the-real-problems
    - https://www.reddit.com/r/Python/comments/pzscwb/whats_your_strategy_on_refactoring/
    - https://github.com/97-things/97-things-every-programmer-should-know/blob/master/en/thing_06/README.md
- https://realpython.com/python-refactoring/