PODPAC: open-source Python software for enabling harmonized, plug-and-play processing of disparate earth observation data sets and seamless transition onto the serverless cloud by earth scientists

As Published: https://doi.org/10.1007/s12145-020-00506-0

Publisher: Springer Berlin Heidelberg

Persistent URL: https://hdl.handle.net/1721.1/131933

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

PODPAC: Open-source Python software for enabling harmonized, plug-and-play processing of disparate earth observation data sets and seamless transition onto the serverless cloud by earth scientists

Cite this article as: Mattheus P. Ueckermann, Jerry Bieszczad, Dara Entekhabi, Marc L. Shapiro, David R. Callendar, David Sullivan, Jeffrey Milloy, PODPAC: Open-source Python software for enabling harmonized, plug-and-play processing of disparate earth observation data sets and seamless transition onto the serverless cloud by earth scientists, *Earth Science Informatics*, doi: 10.1007/s12145-020-00506-0

1. Title

PODPAC: Open-source Python software for enabling harmonized, plug-and-play processing of disparate earth observation data sets and seamless transition onto the serverless cloud by earth scientists

Authors:

Mattheus P. Ueckermann[1], Jerry Bieszczad[1], Dara Entekhabi[2], Marc L. Shapiro[1], David R. Callendar[1], David Sullivan[1], Jeffrey Milloy[1]

Affiliations:

[1]Creare LLC, Hanover, NH, 03755

[2]Massachusetts Institute of Technology, Cambridge, MA 02139

Corresponding Author:

M. P. Ueckermann (mpu@creare.com)

2. Abstract

In this paper, we present the Pipeline for Observational Data Processing, Analysis, and Collaboration (PODPAC) software. PODPAC is an open-source Python library designed to enable widespread exploitation of NASA earth science data by enabling multi-scale and multi-windowed access, exploration, and integration of available earth science datasets to support analysis and analytics; automatic accounting for geospatial data formats, projections, and resolutions; simplified implementation and parallelization of geospatial data processing routines; standardized sharing of data and algorithms; and seamless transition of algorithms and data products from local development to distributed, serverless processing on commercial cloud computing environments. We describe the key elements of PODPAC's architecture, including *Nodes* for unified encapsulation of disparate scientific data sources; *Algorithms* for plug-and-play processing and harmonization of multiple data source *Nodes*; and *Lambda functions* for serverless execution and sharing of new data products via the cloud. We provide an overview of our open-source code implementation and testing process for development and deployment of PODPAC. We describe our interactive, JupyterLab-based end-user documentation including quick-start examples and detailed use case studies. We conclude with examples of PODPAC's application to: encapsulate data sources available on Amazon Web Services (AWS) Open Data repository; harmonize processing of multiple earth science data sets for downscaling of NASA Soil Moisture Active Passive (SMAP) soil moisture data; and deploy a serverless SMAP-based drought monitoring application for use access from mobile devices. We postulate that PODPAC will also be an effective tool for wrangling and standardizing massive earth science data sets for use in model training for machine learning applications.

Keywords: Data harmonization, plug-and-play algorithms, reproducibility, serverless cloud computing, Python, JupyterLab, SMAP

3. Introduction

NASA requires new solutions to address the ever-growing volume and variety of observational and modeled data products[1,2]. NASA's current and future observational platforms such as NASA-ISRO Synthetic Aperture Radar (NISAR), HyspIRI, JPSS-1, Landsat, and MODIS, as well as model-generated weather and climate data sets, commonly generate data products stored in repositories ranging from

[1] https://cdn.earthdata.nasa.gov/conduit/upload/6803/EOSDIS_Update_Summer_2017.pdf

[2] http://ceos.org/document_management/Working_Groups/WGISS/Meetings/WGISS-42/2_Tuesday%20(9.20)/2016.09.20_14.10_Assessing_Applications_of_Cloud_Computing_to_NASAs_EOSDIS.pdf

terabytes (TB) to petabytes (PB) in size. NASA's Earth Observing System Data and Information System (EOSDIS) archives currently manage ~20 PB of data with a growth rate of ~15 TB per day. Moreover, data storage needs will grow dramatically as the upcoming NISAR satellite mission will require EOSDIS to handle an additional 85 TB of data per day. EOSDIS distributes thousands of distinct scientific data products with significant differences in spatial footprints and resolution, temporal resolution, geospatial projections, data formats, and metadata. The heterogeneity of data products hinders additional product discovery and exploitation, in particular for integrated analysis and analytics across observational and modeled data products.

To address data volume issues, NASA is actively investigating the use of commercial cloud services. For example, NASA's Alaska Satellite Facility (ASF) DAAC, responsible for archiving and serving NISAR data, is evaluating the use of Amazon Web Services (AWS) for data storage under its Getting Ready for NISAR (GRFN) project.[3] GRFN is part of the ongoing EOSDIS efforts to evolve NASA Earth observing data and EOSDIS services to the commercial cloud.

Simply migrating data to cloud-based storage will not address all of NASA's big data needs. NASA also requires a solution which addresses the issue of data variety. In a NASA report by Mehrotra et. al (2014), NASA researchers identified six key use cases for utilization of big observational data sets: (1) produce a derived dataset by processing NASA data, (2) find NASA data relevant to a scientific problem, (3) discover new characteristics/features in a NASA dataset, (4) assess the quality of a simulation dataset, (5) answer a scientific question through analysis or analytics on NASA data, and (6) provide the results of analysis/analytics to others. Existing NASA resources have been developed, such as the EOSDIS, NEX, and openNASA to help address Goal 2 (finding data) and Goal 6 (sharing data). However, the remaining goals, which deal primarily with analysis and analytics objectives, remain unaddressed—not only for "big data" applications, but even for "medium data" earth science problems.

To address data variety challenges, earth science researchers typically employ legacy data analysis approaches that involve ad hoc combinations of data wrangling and custom code development in languages such as FORTRAN, MATLAB®, ENVI, and Python; NASA tools such as Earthdata and Giovanni, commercial and open-source geospatial software libraries such as GDAL, Shapely, PROJ.4, OSR, Rasterio and PostGIS; and GIS platforms such as QGIS, ArcGIS, and GRASS. However, these ad hoc solutions take significant effort to implement, are highly customized to specific tasks, and rarely replicable across different applications and other science programs. To perform even the simplest computation or comparison across two independent geospatial raster data sets, an analyst must use a combination of tools to read the different file formats, reproject the data to a common spatial projection, resample the data to the same resolution and consistent pixel centers, and deal with non-overlapping regions and no-data values.

These unmet big data volume and variety challenges motivated the development of the Pipeline for Observational Data Processing Analysis and Collaboration (PODPAC) Python library. PODPAC is designed to enable data analysis and analytics of large-scale earth observational data in an easy-to-use, plug-and-play manner that is approachable for typical earth science researchers, graduate students, and citizen scientists. It is a structured, modular framework that unifies the capabilities of diverse open-source geospatial software tools, harmonizes disparate geospatial data sources, facilitates collaboration and sharing of earth observation data processing pipelines, and streamlines the transition of geospatial data processing workflows to highly scalable serverless cloud deployments.

Other efforts to create portable and reproducible scientific workflows include CyberGIS-Jupyter (Yin et. al. 2019), geoknife (Read et. al. 2016), geoKepler (Coward et. al. 2015), GeoJModelBuilder (Zhang and Yue 2013), Taverna (Missier et. al. 2010), and VisTrails (Bavoil et. al. 2005).

[3] https://earthdata.nasa.gov/getting-ready-for-nisar

In the following sections, we will describe the design and implementation of PODPAC, present the results from several example applications of PODPAC, and discuss outlooks for its future development and usage.

4. Design and Implementation

4.1. Software Architecture

4.1.1 Overview

Figure 1 shows conceptually how PODPAC addresses the data variety and volume challenges faced by earth scientists. To address data variety challenges, the PODPAC Python library encapsulates data sources enabling automatic harmonization of disparate formats, geospatial projections, and data structures. This data encapsulation allows the development of plug-and-play algorithms where users can substitute a new data source within an established processing algorithm. PODPAC generates lightweight JSON representations that document these algorithms so users can share, publish, and reproduce an analysis. To address data volume challenges, PODPAC integrates tightly with AWS, allowing researchers to seamlessly transition workflows developed locally to the cloud. PODPAC's cloud implementation leverages AWS Lambda functions, a serverless technology that provides massive scalability (up to 1000 concurrent invocation by default) with minimal maintenance, and reduced billing risks due to automatic timeout after a maximum of 15 minutes of computation. Not only can users leverage this cloud deployment for large computations, but it also provides a low-cost, low-effort approach for scientists to share their analysis through RESTful APIs and interactive websites.

In PODPAC, *Nodes* represent the basic unit of computation. They retrieve input data (circles in middle of Figure 1), perform computations (diamonds in middle of Figure 1), and produce outputs (square in middle of Figure 1). *Nodes* have a common interface that allows users to assemble them to form reproducible processing pipelines. Pipelines are evaluated on-demand once a user specifies a set of geospatial coordinates. This pipeline design is modular, allowing scientists to use PODPAC for their applications by combining existing or custom developed *Nodes*.

The following describes *DataSource Nodes* for encapsulating geospatial datasets, *Algorithm Nodes* for building processing pipelines, and the PODPAC *Lambda Node* for transitioning workflows to the AWS serverless cloud processing services.

4.1.2 Nodes for Encapsulating Data Sources

Currently, different scientists and analysts repeat efforts to develop scripts for downloading and interpreting data. Errors in data harmonization are also common due to the complexity of geospatial coordinate reference systems (CRSs). Moreover, users commonly have to interact with datasets on a file-by-file level, instead via a single interface.

The goals of PODPAC *DataSource* nodes are to: (1) eliminate the repetitive task of developing data access and interpretation scripts; (2) automatically harmonize disparate data sources without error; and (3) provide a single, common interface to entire datasets. PODPAC's *DataSource* nodes interface with various geospatial data sources (i.e., HDF5 files, OpenDAP servers, Numpy arrays, *Intake* catalogues, OGC-compliant servers, etc.) that are stored locally or retrieved on demand from remote servers. Developers need to implement methods that tell PODPAC how to retrieve data from the source location and interpret the coordinates of the returned data structure. PODPAC will then automatically retrieve and cache data on-demand, project data into a common CRS, and interpolate data to the same grid. This architecture greatly reduces the data wrangling overhead and repetition common in earth science research.

To accomplish these goals, we carefully developed the evaluation interface used by all PODPAC nodes, implemented a flexible method for specifying coordinates to cover a wide range of data structures, and integrated a wealth of open-source libraries to harmonize disparate datasets.

Interface. In order to access data through PODPAC, users need to evaluate the *DataSource* node as follows:

```
data_source = podpac.DataSource(<parameters>)  # stages evaluation
results = data_source.eval(data_source. coordinates)  # runs evaluation
```

where "<parameters>" are specific to a particular data source, and "*data_source. coordinates*" are the geospatial coordinates that describe the source data. PODPAC will then retrieve the data from its source on-demand, and return it in a coordinate-aware *podpac.UnitsDataArray* container. Our *podpac.UnitsDataArray* object inherits from the *xarray.DataArray* object and contains all of the functionality provided by *xarray* along with basic unit checking capabilities and additional formats for saving results. The above example will retrieve the entire dataset in its native coordinate system, but users can also specify a completely custom set of *Coordinates*.

Coordinate Specification. Scientists can use *Coordinates* to evaluate *Nodes*, and developers use them to define the coordinates of *DataSources*. We designed PODPAC *Coordinates* after *xarray Coordinates*, but ours are restricted to the dimensions ["lat", "lon", "time", "alt"] while also allowing users to create more complex coordinate types. A major advantage of PODPAC lies in the generality of the *Coordinates* object to represent data structures spanning gridded arrays to unstructured spatial points to paths (Figure 2).

PODPAC optimizes the storage of coordinates by taking advantage of structure in the data and the ability to separate dimensions. We can describe any geospatial data structure's coordinates by specifying the ["lat", "lon", "time", "alt"] coordinates for each point. However, this increases the amount of storage and memory: we have to keep track of up to four additional pieces of data for every point. For large datasets, this can be a significant storage burden. Moreover, it is more computationally expensive to interpolate between coordinate systems and to perform basic operations (such as determining if two different sets of coordinates overlap). Figure 2 shows three levels of optimized coordinates storage, starting with 1-D representations of coordinates along axes orthogonal to the coordinate system that describe a grid (least amount of storage), to specifying coordinates for each point describing a path (most amount of storage). The basic interface for specifying coordinates is as follows:

```
c = Coordinates([coords_1, coords_2, …], [name_1, name_2, …], crs="crs")
```

where "coords_*", "name_*", and "crs" are the values, names, and PROJ4 CRS string for the coordinates, respectively. Table 1 shows the interface for creating Coordinates representing the data structures shown in Figure 2. Note, in addition to Python tuples and lists, users can employ Numpy arrays for the coordinate values.

Harmonization. PODPAC harmonizes data by automatically projecting and interpolating data based on the user's requested *Coordinates*. PODPAC's interpolation design is modular, flexible, and extensible, since there is a wide variety of different interpolation methods (and implementations of those methods) that users may want to use. Users can specify the interpolation method broadly, or with great specificity. With a basic specification (such as "nearest" or "bilinear"), PODPAC automatically decides which interpolation implementation to use. Alternatively, users can specifically control the interpolation method for each coordinate dimension for each *DataSource Node* by specifying a dictionary of parameters. PODPAC natively supports a number of *Interpolators* covering a wide range of use cases to interpolate between grid and point data using various methods. Internally, PODPAC leverages interpolation methods from *xarray*, *Scipy*, and *Rasterio*.

4.1.3 Algorithms for Automated Harmonization and Plug-and-Play Processing

Sharing, reproducing, and adapting analyses by earth scientists is challenging due to fragile processing scripts, undocumented data preprocessing procedures, and barriers to data access. The goal of PODPAC *Algorithm* nodes is to document geospatial data processing pipelines in a lightweight manner to address these challenges, describing the complete process, from raw data retrieval to producing the final product.

With data stored in the cloud or on publically accessible servers, this approach promises to make earth science more reproducible in general. In conjunction with automated data harmonization, it also enables plug-and-play substitution of new data into existing processing pipelines. Finally, the lightweight representation enables users to distribute PODPAC pipeline evaluation to heterogeneous compute architectures such as the cloud or local workstations.

In Python, users assemble processing pipelines by instantiating *Node* objects and settings these as inputs to *Algorithm Nodes*. Internally, PODPAC keeps track of this assembly by recursively inspecting each *Algorithm*'s input *Node* and keeping track of the *Node* definitions. A set of tagged attributes define each *Node*, and our custom encoder serializes these to text using the JSON format. Moreover, users can create PODPAC *Nodes* from properly formatted JSON text using the *Node.from_json* method. As such, PODPAC serializes complete processing pipelines to human-readable JSON text, which provides a lightweight mechanism to share analysis. Because PODPAC can recreate *Nodes* from this JSON representation, the same computation can be distributed and run on heterogeneous compute architectures, such as the cloud.

PODPAC *Algorithm Nodes* may manipulate the evaluation *Coordinates* or process the data directly. The former is useful for computing climatologies over complex time windows (such as the 8th week of the year over the past 10 years). PODPAC implements a number of algorithms that include: (1) signal processing / convolution operations; (2) reduction algorithms for computing statistics such as mean, mode, and variance; (3) ways to modify coordinates such as expanding the evaluated coordinates to cover a wider region or substituting a different value for a dimension; (4) and generic algorithms that can evaluate arbitrary equations point-wise or run arbitrary Python code. Developers can also extend PODPAC's capabilities by creating new classes that inherit from our *Algorithm* node. They then need to implement the *algorithm(inputs)* method, which takes a dictionary of harmonized input data sources as *UnitsDataArrays*. With this functionality, scientists create a wide range of processing pipelines that are easily sharable and reproducible across PODPAC installations.

Our implementation of *Algorithm Node*s supports multi-threading. While Python does not support native parallel computation via threads due to the Global Interpreter Lock (GIL), it does support parallel IO. As shown on the right of Figure 3, this capability ameliorates the main bottleneck associated with data retrieval in *Algorithm Nodes*. As shown on the left of Figure 3, without multi-threading each input *DataSource Node* would incur a latency penalty associated with accessing data (from cloud storage, for example). Our implementation allows users to limit the maximum number of threads spawned by PODPAC. Algorithm Nodes fall back to serial data retrieval once the process reaches the maximum number of threads.

4.1.4 Seamless Serverless Cloud Transition

The need for vendor-specific expertise and knowledge, administrative barriers associated with budgeting, and the non-trivial task of transitioning established workflows are major barriers to adoption of cloud computing by earth scientists. Meanwhile, cloud computing offers significant benefits, such as low cost and effort to access data stored on the cloud, on-demand access to massive compute resources, and lower maintenance costs. Moreover, to disseminate their analyses widely and in a reproducible manner, scientists can use cloud services. The goal of PODPAC's AWS integration is to provide a seamless path to cloud computing that abstracts the major barriers identified above. We do this with a full PODPAC environment for AWS Lambda functions that scientists can easily deploy and use inside a Python console. We also provide an interface for tracking budgets. We implemented this capability within PODPAC's *Lambda Node*.

AWS Lambda functions are a serverless technology. This means that users do not need to provision, setup, maintain, and turn off a cloud server manually. Instead, users develop functions that can be "triggered." When triggering a function, AWS automatically provisions a server, runs the desired function, and turns the server off upon completion. This alleviates the problem of server maintenance and

avoids the problem of budget overruns associated with accidently leaving servers turned on. It is also massively scalable, allowing a parallel computation over 1000 machines, by default. The downside to AWS Lambda functions come from restrictions imposed on the available memory, maximum runtime, operating system used for the deployment, and file size restrictions for dependencies. As a result, it can be very challenging to implement an AWS Lambda function for geospatial applications.

PODPAC's AWS Lambda function integration provides a general PODPAC environment that is easy to set up. We create and maintain this environment using a Docker container that generates the necessary zip files used to create the AWS Lambda function. We make these zip files publically available via an S3 bucket so that PODPAC's *Lambda Node* can automatically set up a user's AWS account with a PODPAC AWS Lambda function. The only manual step required is for users to obtain an AWS login and generate an access key ID and secret access key through the following process:

1. Log into console.aws.amazon.com
2. Go to Services → IAM → users
3. Click on username
4. Click on the "Security credentials" tab
5. Click on "Create access key"
6. Save the "Access key ID" and "Secret access key"

Once this is done, the following Python code will create the S3 Bucket, Lambda function, IAM roles, and API Gateway resources needed to run PODPAC pipelines in the cloud, and set up alarms to track spending:

```
import podpac
from podpac.managers import aws
settings = podpac.settings.copy()  # Make a copy of the default settings
# Add AWS Credentials, budget information, and resource base name to settings
settings["AWS_ACCESS_KEY_ID"] =              #    Users's    AWS    access    key    id
settings["AWS_SECRET_ACCESS_KEY"] = # Users's AWS secret access key
settings["AWS_REGION_NAME"] =                # User's AWS region name
settings["AWS_BUDGET_AMOUNT"] =        # budget for AWS resources in USD/month
settings["AWS_BUDGET_EMAIL"] =         # notification e-mail for budget alarms
settings["FUNCTION_NAME"] =          # unique name of AWS Lambda function

node = aws.Lambda(eval_settings=settings)  # create the Lambda Node
node.describe()                            # view the staged AWS resources
node.build()                               # automatically build AWS resources
node.validate()                            # returns True if AWS resource exist
```

Once a user has created a PODPAC AWS Lambda function, they can use the *Lambda Node* to seamlessly transition from using PODPAC on a workstation to computation in the cloud, while tracking their spending. Figure 4 conceptualizes how PODPAC triggers the AWS Lambda function using the JSON definitions of the processing pipeline. It also shows how the user can optionally store the data in an S3 bucket. As an example, the following code runs a PODPAC pipeline on a local workstation, and then transitions to the cloud:

```
node.eval(coordinates)                     # Runs on local workstation
lambda_node = aws.Lambda(source=node)
lambda_node.eval(coordinates)              # Runs on AWS
```

PODPAC's AWS Lambda function includes a partial implementation of the OGC-compliant WMS/WCS standards. This allows users to display geospatial data using OGC-compliant clients such as the Leaflet JavaScript library. To use this functionality, users need to provide query parameters as part of the URL to define the processing pipeline. Users can either embed the JSON pipeline definition in the URL, or specify a URL to a JSON pipeline definition. Figure 5 shows an example using the WMS capability in a webpage with the Leaflet JavaScript library. Here the API Gateway URL is provided through the PODPAC *Lambda.describe()* method, and the pipeline definition is stored at the indicated URL. This shows how scientists can easily develop a webpage to share their analyses through interactive maps.

4.2. Software Development Approach

PODPAC development involves an object-oriented design with quarterly release cycles, automated unit testing, code coverage reports, and automated documentation builds. It is version controlled using Git, with publically accessible source code available from https://github.com/creare-com/podpac, and documentation available at https://podpac.org.

Releases and Versioning. PODPAC is released under the Apache 2.0 license. Releases are versioned using a major, minor, and bug fix version numbering scheme. Changes in major version mean that interfaces are not backwards compatible, changes in minor version mean new features are developed, and bug fix number indicates the number of critical bugs fixed for the specific major/minor version. Additionally, development versions have the Git hash appended.

Code is released on GitHub, the Python Package Index (PYPI), and as a standalone distribution for Windows hosted on AWS S3. Releases include release notes that summarize major features, bug fixes, and backwards-incompatible changes.

For each release, developers run the Jupyter Notebook examples to ensure they work as expected.

Unit Testing. Unit testing ensures that small units of the code are correctly implemented. Thoroughly testing the code and automatically running tests on freshly installed environments is crucial for an open source project with external developers. It is also useful for preventing regression during major refactoring of the code, or while fixing complex bugs. As such, PODPAC uses "Travis CI" on GitHub with Python 3.5–3.8 Python environments and the latest versions of dependencies to automatically run unit tests after every commit. These unit tests are also supplemented by automated coverage reports using "Coveralls."

Unit tests are mandatory for all new major PODPAC features. As part of code reviews during merge requests, reviewers are expected to ensure there are sufficient and relevant unit tests in place. Moreover, upon discovering bugs in the code, it is expected that a unit test is written to cover this bug as part of fixing the problem.

Code Coverage. Coverage reports aid with writing sufficient unit tests to cover all of the branches in the code. Coverage reports capture the lines of code executed by unit tests, and therefore indicate which lines of code are being tested. While coverage does increase the confidence that there is sufficient testing, it does not guarantee that the quality of testing is sufficient to ensure the results are correct. For example, running a function with a subset of possible inputs without checking the correctness of the result will still increase the coverage. As such, we aimed to write useful unit tests, using the coverage report as a guideline for focusing our efforts. Our goal is to maintain greater than 90% code coverage. Greater coverage than this is impractical due to a number of hard-to-test cases. For example, some of our code handles missing dependencies, but these dependencies are always present in our test environment.

Documentation. Useful documentation and examples are crucial for promoting adoption of open source libraries by the community at large. This includes user documentation describing how to use the software, developer documentation describing how to contribute to the source code, and templates for bug reports and feature requests.

PODPAC developers are expected to document every function and class with Python Docstrings. In addition, we develop user guides using markdown, and use-case examples using Jupyter Notebooks. All of this documentation is automatically committed to a dedicated document repository (<https://github.com/creare-com/podpac-docs>) which is published via GitHub "pages" feature to https://podpac.org.

Dependencies. PODPAC strives to leverage existing open source libraries to the maximum extent. However, not all users need all of PODPAC's functionality. As such, PODPAC has a relatively small set of core dependencies, and additional dependencies are imported on-the-fly using the *Lazy-Import* Python package, which throw errors at runtime as soon as Python tries to use an unavailable package.

One of PODPAC's core dependencies is the Python package *Traitlets*. The main goal behind *Traitlets* is to implement type checking of Python variables. Type checking helps users use the code as intended by raising exceptions whenever a variable is set to an unexpected type. *Traitlets* also implements a number of additional features, including metadata tagging of variables, which PODPAC takes advantage of to automatically serialize processing pipelines into JSON text. Whenever a developer adds "*.tag(attr=True)*" to a *Traitlets* type, PODPAC includes that attribute as part of a *Node* definition.

Besides *Traitlets* PODPAC's core dependencies are *Matplotlib*, *Numpy*, *Pint*, *Scipy*, *Xarray*, *Requests*, *Pyproj*, *Lazy-Import*, and *Psutil*. To deal with different geospatial data sources, optional dependencies include *H5py*, *Pydap*, *Rasterio*, and *Zarr*. For AWS integration, PODPAC uses the *AWSCLI*, *Boto3*, and *s3fs* packages.

Open Source Governance. PODPAC uses an open-source governance model similar to that of Linux, where contributions from the wider community are encouraged, but the ultimate control is maintained by a small group of dedicated developers.

4.3. Training Materials

To facilitate adoption of PODPAC by new users, we have developed Jupyter Notebooks in a separate repository that users can access and run using Binder (https://github.com/creare-com/podpac-examples). Binder is a service to run Jupyter Notebooks without any setup, and serves as a highly accessible way to evaluate PODPAC for specific applications. Jupyter is an open-source project developing a web-based programming user interface that combines code with rich display of text and media. Jupyter Notebooks are gaining popularity in the scientific community because they are easy to share and increase reproducibility. We developed notebooks to guide users through learning PODPAC's concepts using simple examples, and to demonstrate PODPAC's capabilities with use-case examples.

Table 2 summarizes the simple example notebooks available in PODPAC version 1.3.0. In addition to these examples, we also have examples demonstrating data access through various interfaces, example processing pipelines, and detailed examples for developers. Finally, we have a systematic tutorial for deploying the Drought-Monitor application (see Section 5.3) suitable for novice users starting from no PODPAC experience.

5. Results

5.1. Encapsulation of Disparate Data Sources

Version 1.3.0 of PODPAC implements interfaces for accessing: (1) SMAP data via OpenDAP; (2) SMAP data via the Earthdata Gateway Interface (EGI); (3) Global Forecast System (GFS) data stored on AWS S3; (4) digital terrain data stored on AWS S3; and (5) Intake catalogues. SMAP is a soil moisture product from the Soil Moisture Active Passive satellite mission, and the National Snow and Ice Data Center (NSIDC) Distributed Active Archive Center (DAAC) provides the data through the OpenDAP protocol, and the EGI endpoint. NOAA hosts their GFS model data output on Amazon's S3 via a publically available bucket through the Amazon Open Data Registry. Similarly, MapZen provides digital terrain data through Amazon's Open Data Registry.

We implemented access to these sources as part of the *podpac.datalib* module. As a result, PODPAC users can easily access any of these data source in their desired coordinate system and CRS. As an example, the following code will access SMAP and terrain data over a 1 degree by 1 degree region centered over a (latitude, longitude) of (40, -100) on February 1st of 2020:

```
import podpac
from podpac import datalib
coords = podpac.Coordinates(
       [podpac.clinspace(41, 39, 65), podpac.clinspace(-101, -99, 65),
       '2020-02-01'], ['lat', 'lon', 'time'])
smap = datalib.smap_egi.SMAP()
terrain = datalib.terraintiles.TerrainTiles()
smap_out = smap.eval(coords)
terrain_out = terrain.eval(coords)
```

We based these specific data source nodes on the following generic PODPAC *Nodes* shown in Table 3. Both the OpenDAP *SMAP* and *TerrainTiles* sources use PODPAC Compositors to assemble the many different files to give users a single interface into the data. The EGI *SMAP* data source uses the generic *EGI* node, which implements an interface to NASA's API. Finally, the *GFS* data source uses the *Rasterio* node, and manually assembles various files together.

5.2. Plug-and-Play Algorithms with Automated Data Harmonization

To illustrate PODPAC's plug-and-play algorithm development enabled by automatic data harmonization, we implemented a simple soil moisture downscaling model that combines multiple disparate data sources shown in Figure 6. Starting from coarse 9km-scale SMAP data, we compute high resolution 30m soil moisture using an approach based on TOPMODEL (Beven et. al., 1984). TOPMODEL requires the Topographic Wetness Index (TWI) as part of the computation. TOPMODEL also needs the Porosity and the wilting point of the soil. Finally, TOPMODEL needs a tuning parameter κ to specify the complete model:

$$\Theta = \Theta_{SMAP} + \frac{\rho - \Theta_w}{\kappa}\left(\lambda - \bar{\lambda}\right)$$

Where Θ is the downscaled soil moisture, Θ_{SMAP} is the coarse-scale soil moisture, ρ is the soil porosity, Θ_w is the soil wilting point, λ is the fine-scale TWI and $\bar{\lambda}$ is the coarse-scale TWI.

The following code implements this model, and users can evaluate it using arbitrary PODPAC *Coordinates*:

```
import podpac, podpac.datalib

# Access data via OpenDAP from the National Snow and Ice Data Center (NSIDC)
smap = podpac.datalib.smap.SMAP(interpolation='bilinear')
wilt = podpac.datalib.smap.SMAPWilt(interpolation='bilinear')
porosity = podpac.datalib.smap.SMAPPorosity(interpolation='bilinear')

# Access TWI via OGS-compliant WCS endpoint
twi = podpac.data.WCS(
source='https://mobility.crearecomputing.com',
       layer_name='dassp.main_map.topography.elevation' + \
                     '.TopographicWetnessIndexUSGSNED30m',
       interpolation='nearest')

# Re-project high resolution TWI onto SMAP grid. Note, the WCS endpoint
# provides averaged data from pre-computed overviews for coarser requests
twi_bar = podpac.data.ReprojectedSource(
source=twi,
       reprojected_coordinates=smap.shared_coordinates,
       interpolation='bilinear')

# Implement the downscaling algorithm
downscaled_sm = podpac.algorithm.Arithmetic(
    smap=smap, twi=twi, twi_bar=twi_bar, rho=porosity, wilt=wilt,
    eqn='smap + (rho - wilt) / 13.0 * (twi - twi_bar)')

# Evaluate at arbitrary coordinates and plot the results
coordinates = podpac.Coordinates(
[podpac.clinspace(41, 40.9, 916),
podpac.clinspace(-77, -76.9, 916),
'2017-09-03T12:00:00'], dims=['lat', 'lon', 'time'])
out = downscaled_sm.eval(coordinates)
out.plot(cmap='gist_earth_r')
```

Users can share this algorithm using its JSON definition. For example, the above example has the following JSON (where parts have been omitted for publication):

```
{
    "SMAPPorosity": {
        "node": "datalib.smap.SMAPPorosity",
        "attrs": {"datakey": "Land_Model_Constants_Data_clsm_poros"},
        "source": "https://.../SMAP_L4_SM_lmc_00000000T000000_Vv4030_001.h5",
        "interpolation": "bilinear"
    },
    "SMAP_SPL4SMAU": {
        "node": "datalib.smap.SMAP",
        "attrs": {"base_url": "https://n5eil02u.ecs.nsidc.org/opendap/SMAP",
            "product": "SPL4SMAU", "version": 4},
        "interpolation": "bilinear"},
    "TopographicWetnessIndexUSGSNED30m": {
        "node": "core.data.ogc.WCS",
        "attrs": {"crs": "EPSG:4326", "version": "1.0.0",
        "layer_name": "...TopographicWetnessIndexUSGSNED30m"},
        "source": "https://mobility.crearecomputing.com/ogc",
        "interpolation": "nearest"},
    "TopographicWetnessIndexUSGSNED30m_reprojected": {
        "node": "core.data.reprojection.ReprojectedSource",
        "attrs": {"reprojected_coordinates": {"coords": [
                    {"values": [ ... ], "name": "lat"}
                    {"values": [ ... ], "name": "lon"}], "crs": "EPSG:4326"},
            "source_interpolation": "nearest"},
        "lookup_source": "TopographicWetnessIndexUSGSNED30m",
        "interpolation": "bilinear"},
    "SMAPWilt": {
        "node": "datalib.smap.SMAPWilt",
        "attrs": {"datakey": "Land_Model_Constants_Data_clsm_wp"},
        "source": "https://.../SMAP_L4_SM_lmc_00000000T000000_Vv4030_001.h5",
        "interpolation": "bilinear"
    },
    "Arithmetic": {
        "node": "core.algorithm.generic.Arithmetic",
        "attrs": {"eqn": "smap + (rho - wilt) / 13.0 * (twi - twi_bar)"},
        "inputs": {
            "rho": "SMAPPorosity",
            "smap": "SMAP_SPL4SMAU",
            "twi": "TopographicWetnessIndexUSGSNED30m",
            "twi_bar": "TopographicWetnessIndexUSGSNED30m_reprojected",
            "wilt": "SMAPWilt"}}
}
```

Before developing PODPAC, implementing this simple downscaling model required hundreds of lines of code to produce an output for a small set of data. With this PODPAC implementation, not only can users evaluate the node at arbitrary coordinates, they could substitute different soil moisture data, and host an interactive browser application via the PODPAC *Lambda* function implementation. Moreover, users do not have to manually download or stage data for this example to work. As a result, PODPAC-developed geospatial analysis should accelerate science by increasing reuse, reproducibility, and productivity.

5.3. Serverless Cloud Deployment of a New Drought Monitoring Index Based on Remote-Sensed SMAP Soil Moisture Data

Timely monitoring of drought conditions is critical for agriculture, insurance, and government applications. The National Drought Mitigation Center (NDMC) publishes a widely used weekly national drought monitoring index[4] (NDMI) which is authored by experts examining multiple remotely sensed data sources, in situ sensors, and local reports. However, more timely data at a higher spatial resolution would further help to mitigate the socio-economic effects of droughts. To demonstrate the use of PODPAC for deploying such an application using serverless cloud services, we developed a new drought index based on remotely sensed SMAP soil moisture data. SMAP is a satellite platform that estimates soil moisture at 9 km resolution with global coverage achieved approximately every 3 days.

NDMI enumerates drought conditions in terms of five categories:

1) D0 represents abnormally dry conditions (20-30% percentile probability)
2) D1 represents moderate drought conditions (10-20% percentile probability)
3) D2 represents severe drought conditions (5-10% percentile probability)
4) D3 represents extreme drought conditions (2-5% percentile probability)
5) D4 represents exceptional drought conditions (0-2% percentile probability)

We defined analogous drought index categories using SMAP soil moisture estimates. To implement this approach, we defined a PODPAC *Algorithm* pipeline that performs the following computations:

1) For each date of the year, select all available SMAP soil moisture data within a 45-day window
2) Divide SMAP volumetric soil moisture by porosity to convert to relative soil saturation
3) Fit a beta distribution using maximum likelihood estimation to retrieve “a” and “b” parameters of distribution
4) Assign the D0 to D4 drought categories according to their percentiles, and convert back to volumetric soil moisture by multiplying by porosity
5) Convert SMAP soil moisture estimate to drought categories by thresholding using the D0-D4 values

We then deployed our SMAP drought monitoring index (DMI) as a PODPAC Lambda function with an HTTP gateway interface for WMS access from a web browser. For the end-user application, we developed a static webpage which includes an interactive slippy map for display of NDMI, SMAP DMI (left panel of Figure 7), and SMAP soil moisture (right panel of Figure 7). This end-user application is available at https://creare-com.github.io/podpac-drought-monitor/

The top row in Figure 8 illustrates the underlying functionality of the deployed SMAP DMI application. A user accesses the application through a web browser on a desktop or mobile device platform. The application makes RESTful calls to the SMAP DMI Lambda function with the function parameters embedded as PODPAC pipeline JSON definitions. The SMAP DMI Lambda function retrieves the required SMAP data which is stored in a cloud-optimized Zarr format. The SMAP DMI Lambda function then transforms the raw data to drought categories, and depending on the request, returns the result as JSON text or PNG images via WMS. The bottom row in Figure 8 shows the basic mechanism for staging new SMAP data in a cloud-optimized format. Since SMAP data is not yet stored on the cloud, we use an AWS CloudWatch Event to check for new data daily via a second PODPAC Lambda function. This Lambda function downloads and stages the SMAP data in the cloud using a cloud-optimized storage format.

Figure 9 shows a comparison between the new SMAP DMI Monitor with NDMC’s NDMI. The overall trends are similar, but the SMAP DMI provides higher temporal and spatial resolution and is developed using fully reproducible data processing approach using remote sensing SMAP data. The manual creation

[4] https://droughtmonitor.unl.edu/

of NDMC's NDMI uses multiple data sources, and this accounts for some of the differences in the comparison. Additional differences arise because drought category percentiles generated from SMAP use relatively recent data (since 2015). For example, our SMAP DMI may consider recent multi-year droughts as statistically likely leading to a "moderate drought" categorization, whereas an analysis with more years of data would categorize the same event as "extreme drought".

6. Discussion

The current 1.3.0 implementation of PODPAC provides a strong foundation for an extensible geospatial analysis framework. Its data wrangling capability, plug-and-play algorithms, and AWS cloud implementation have accelerated our use and sharing of earth science data.

PODPAC has greatly increased our productivity retrieving and wrangling datasets. Once a dataset was wrapped in PODPAC, we no longer needed to worry about the minutia of how the data was stored, which variables we need to consider, and how to avoid simple interpretation errors. For creating new datasets, we found the zarr format to be very beneficial when storing data in the cloud. It breaks up large datasets in chunks and handles operations such as concurrent writes and appending data seamlessly. We also discovered that remote servers sometimes go down or make unanticipated changes, causing a previously-working dataset to no longer function while throwing a potentially incomprehensible error. In these cases, we improve the error message but we anticipate continued problems like this as community members start contributing *DataSource* nodes. We plan to continue wrapping high-value data sources stored in the cloud for our future applications. To sustain data wrapping activities, we plan to leverage our internal research projects, and we invite community members to contribute as well.

PODPAC's plug-and-play algorithms allow us to answer what-if questions in a matter of minutes instead of hours. Before PODPAC, we were frustrated by the difficulty and amount of coding needed to take a simple difference between two datasets. Now we analyze and visualize PODPAC outputs rapidly using powerful open source analysis libraries such as *xarray* and *Scipy*. We have found the grouping, windowing, and statistical operations by *xarray* particularly helpful, since PODPAC nodes output *xarray*-compatible data containers. For initial development purposes, we found it helpful to retrieve data over a small region using PODPAC and then developing analyses in a Jupyter notebook. Once the analysis looks good for the small region, we found it helpful to go back and develop the full PODPAC processing pipeline to enable rapid processing of large regions or interactive visualization in the cloud.

PODPAC's easy AWS cloud integration has allowed us to iterate rapidly when developing a new applications such as the Drought-Monitor. We find the WMS capability most useful for rapidly analyzing a new data product interactively. Invariably there are aspects that we wish to change and improve, and since it is simple to deploy an AWS Lambda function, iteration takes minutes instead of hours or days. Navigating AWS services was non-trivial, and we found their interface across services was not consistent in surprising ways. To overcome the memory and maximum runtime restrictions of Lambda functions, we had to restrict the size of coordinates in our processing requests. The file size limit associated with the Lambda function was particularly difficult to deal with, and requires us to download additional dependencies on-the-fly. This issue precluded us from using existing libraries that facilitates AWS Lambda function deployments such as Serverless[5] and Python-lambda.[6] However, with the introduction of AWS Lambda Layers, some of our maintenance burden is relieved. At the time of writing, we also found that the traceback logs provided through PODPAC to be much more useable compared with those available from the AWS CloudWatch web console, which required a lot of scrolling and dynamic loading of data. This allowed us to optimize the execution speed of our Lambda function more rapidly for the Drought-Monitor, but this is an ongoing effort. Optimizing on-demand calculations on AWS Lambda

[5] https://github.com/serverless/
[6] https://github.com/nficano/python-lambda

functions for interactive browsing remains challenging, primarily due to application-specific data access patterns that drive specific cloud storage structures.

While PODPAC targets AWS Lambda functions for cloud deployment, PODPAC is not restricted to run on this service with this provider. PODPAC provides a Docker file as a starting point for users to integrate with AWS Elastic Container Services (ECS) or other cloud providers such as Azure and Google Cloud. The difficulty of implementing PODPAC on other cloud architecture depends on the particulars of the application and the available services. The main challenges would involve installing dependencies, managing message interfaces (API endpoints), data access, and managing authentication.

Future capability development of PODPAC includes: (1) increasing the robustness of the code by finding and correcting edge cases; (2) improving the functionality of compositors (for example to composite a folder full of tiled files automatically); (3) wrapping additional high-value data sources; (4) implementing additional algorithms; (5) increasing the capability and types of interpolation methods; (6) developing custom JupyterLab widgets for graphical interaction and rapid dashboard creation; (7) integrating seamlessly with other cloud services/providers; and (8) integrating with machine learning libraries such as TensorFlow. Since formatting and preprocessing data is a major challenge with machine learning applications, PODPAC is a natural candidate for improving and streamlining this process, reproducibly.

7. Conclusions

Widespread analysis and analytics of NASA and non-NASA earth science data currently face significant barriers to entry due to disparate geospatial data formats, large computational demands, and high data storage and bandwidth requirements. To mitigate these barriers to entry, we have developed PODPAC for automated harmonization of earth science datasets, plug-and-play development of algorithmic processing pipelines, and seamless transition of scientific workflows to the serverless cloud by scientists with minimal experience (or interest) in managing the complexity of cloud computing environments. These capabilities are documented, readily demonstrated, and built upon through example JupyterLab notebooks which accompany the software. Moreover, our open-source software approach and explicit documentation of algorithm pipelines and data product provenance greatly facilitates sharable and reproducible scientific research. Finally, providing agile access to earth science data and tools for plug-and-play geospatial analysis can revolutionize the productivity and effectiveness of everyday decision-makers and citizen scientists from a variety of fields such as agriculture, aquaculture, disaster response, forestry, land management, and municipal planning, and outdoor recreation.

8. Availability and Requirements

PODPAC source code, examples, and documentation can be downloaded from the PODPAC website at https://podpac.org and its Git repository at https://github.com/creare-com/podpac. All files and documentation are made freely available under the 2.0 version of the Apache License (https://www.apache.org/licenses/LICENSE-2.0). The PODPAC software can be run on any computational hardware and operating system for which Python 3.6 (or later). This includes most recent versions of the Microsoft Windows, Apple MacOS, and Linux operating systems. Deployment of algorithms to Lambda serverless computing resources from PODPAC requires users to have an active AWS account and credentials.

9. List of Abbreviations

AWS	Amazon Web Services
CRS	Coordinate Reference System

DAAC	Distributed Active Archive Center
DEM	Digital Elevation Mode
JSON	JavaScript Object Notation
NASA	National Aeronautics and Space Administration
NSIDC	National Snow and Ice Data Center
PODPAC	Pipeline for Observational Data Processing Analysis and Collaboration
SMAP	Soil Moisture Active Passive
TWI	Topographic Wetness Index
WCS	Web Coverage Service
WMS	Web Mapping Service

10. Acknowledgements

This research is supported by NASA under SBIR Phase II Contract No. 80NSSC18C0061. We gratefully acknowledge the developers of the open source software libraries on which PODPAC depends, in particular Anaconda, GDAL, JupyterLab, numpy, proj4J, rasterio, scipy and xarray. The SMAP data used in this study can be downloaded from https://nsidc.org/data/smap/smap-data.html. The global digital elevation model (DEM) data used in this study can be downloaded from http://viewfinderpanoramas.org/dem3.html. The TWI data used in this study can be downloaded via WCS from https://mobility.crearecomputing.com/ogc?

11. References

Bavoil L, Callahan S, Scheidegger C, Vo H, Crossno P, Silva C, Freire J (2005) Vistrails: enabling interactive multiple-view visualizations. IEEE visualization 135–142

Beven KJ, Kirkby MJ, Schofield N, Tagg AF. (1984) Testing a physically-based flood forecasting model (TOPMODEL) for three UK catchments. Journal of Hydrology, 10; 69(1-4):119-43.

Cowart, C., Block, J., Crawl, D., Graham, J., Gupta, A., Nguyen, de Callafon, M., R. Smarr, L. and Altintas, I. (2015). geoKepler Workflow Module for Computationally Scalable and Reproducible Geoprocessing and Modeling. AGUFM, 2015, NH43B-1887.

Mehrotra, P., Pryor, LH., Bailey RF. and Cotnoir, M. (2014) NASA Technical Report NAS-2014-02. NASA Ames, Moffett Field, CA. http://www.nas.nasa.gov/assets/pdf/papers/NAS_Technical_Report_NAS-2014-02.pdf. Accessed 12 August 2020.

Missier P, Soiland-Reyes S, Owen S, Tan W, Nenadic A, Dun-lop I, Williams A, Oinn T, Goble CA. (2010) Taverna, reloaded. SSDBM, 471–481.

Read, J. S., Walker, J. I., Appling, A. P., Blodgett, D. L., Read, E. K., & Winslow, L. A. (2016). geoknife: reproducible web-processing of large gridded datasets. Ecography, 39(4), 354-360.

Yin, D., Liu, Y., Hu, H., Terstriep, J., Hong, X., Padmanabhan, A., & Wang, S. (2019). CyberGIS-Jupyter for reproducible and scalable geospatial analytics. Concurrency and Computation: Practice and Experience, 31(11), e5040.

Zhang, M., & Yue, P. (2013). GeoJModelBuilder: A java implementation of model-driven approach for geoprocessing workflows. In 2013 Second International Conference on Agro-Geoinformatics (Agro-Geoinformatics) (pp. 393-397). IEEE.

12. Figures and captions

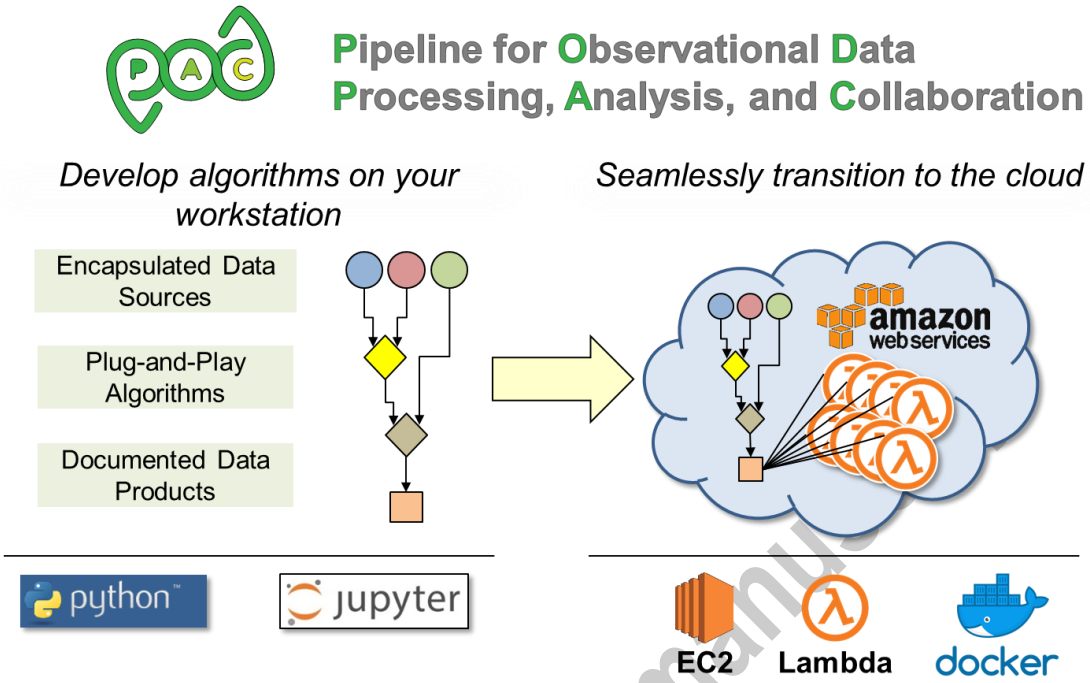

Figure 1. PODPAC's Python library automatically harmonizes encapsulated data sources to enable plug-and-play algorithm development (left). Lightweight JSON representations automatically document Algorithms, allowing them to be processed remotely (right) or shared for reproduction. PODPAC tightly integrates with AWS to provide a seamless transition from running on a workstation to running on the cloud.

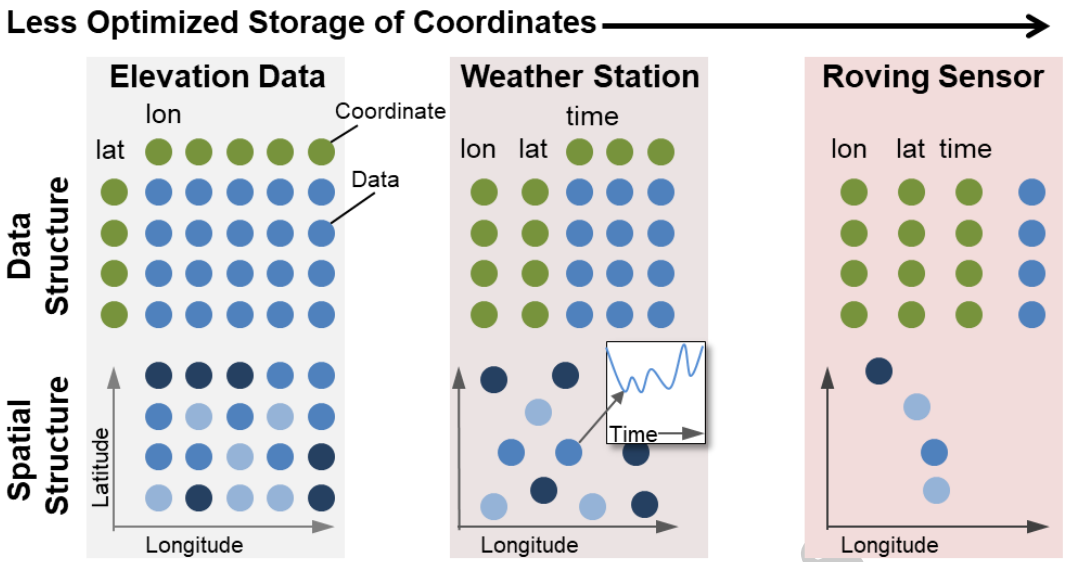

Figure 2. Optimized storage of *Coordinates* in PODPAC. The top row represents the data structure used to store data, with coordinates indicated in green, and data points indicated in blue. The bottom row shows how the data maps spatially with notional data values indicated by various shades of blue. Left shows an example of raster data such as a digital elevation model, middle shows an example of a point collection recording data over time such as weather stations, and right shows data collected along a path from a roving sensor such as pressure recorded by a cellphone while the user is walking around.

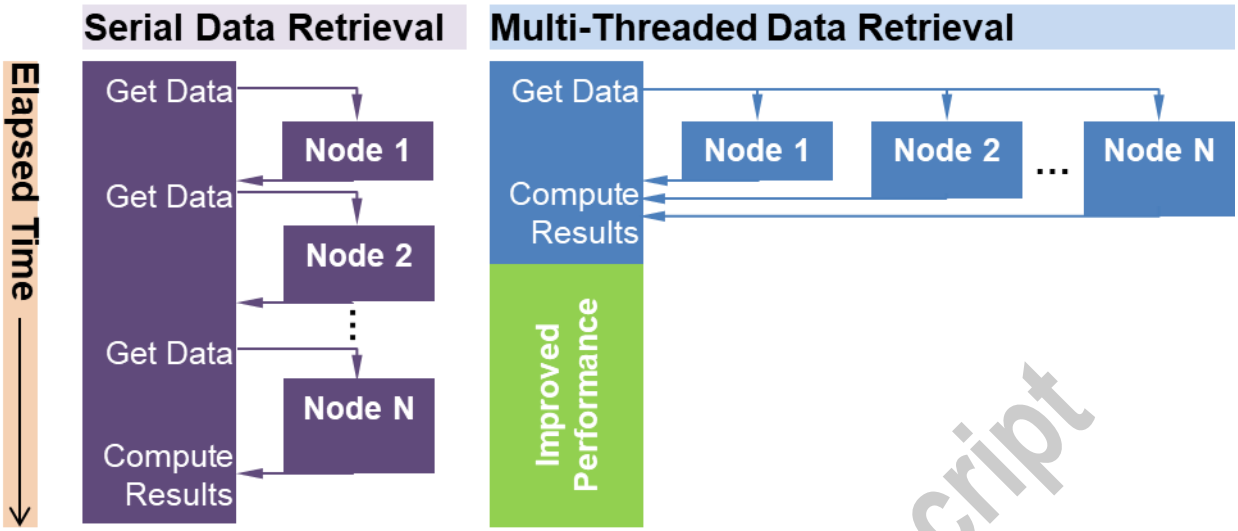

Figure 3. PODPAC Algorithm Nodes are multi-threaded for parallel IO to reduce the time needed for retrieving multiple remotely stored data sources.

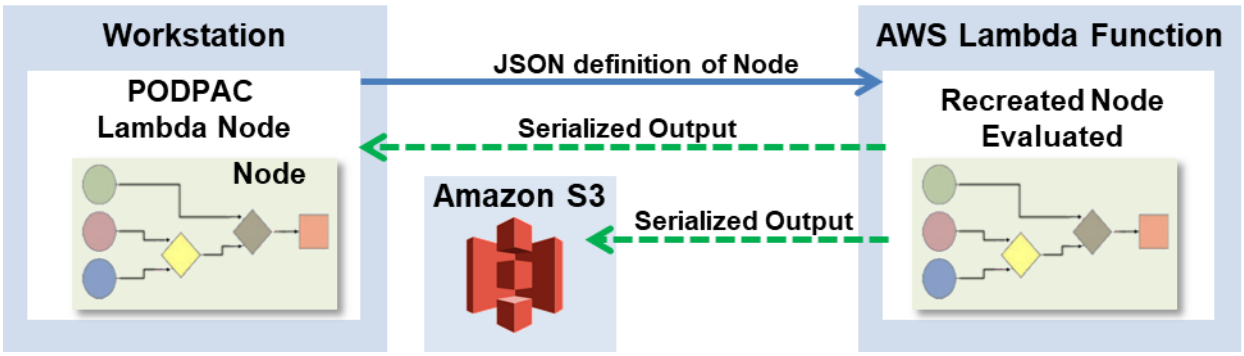

Figure 4. PODPAC Lambda function integration. The *Node* and *Coordinates* created on a user's local workstation are serialized as JSON and used to trigger our generic PODPAC AWS Lambda function. This AWS Lambda function recreates the same *Node* and *Coordinates* from the JSON definition and evaluates it in the cloud. The output can be uploaded to an S3 bucket, or returned to the user's local workstation.

```
<body>
  <!-- Map Page -->
  <div id="map" class="map" style="height:800px;"></div>
  <script>
    // API URL from node._function_api_url in Python
    var SMAPWMS = L.tileLayer.wms(
      "https://ps1dfpoecf.execute-api.us-east-1.amazonaws.com/prod/eval/?",
      {
        transparency: true, opacity: 0.95, format: 'image/png', time: "2020-01-01",
        layers: "https://podpac-drought-monitor-s3.s3.amazonaws.com/pipeline_category.json"
      });
    var map = L.map('map',
    {
      center: [42, -100.0], zoom: 4,
      layers: [SMAPWMS]
    });
  </script>
```

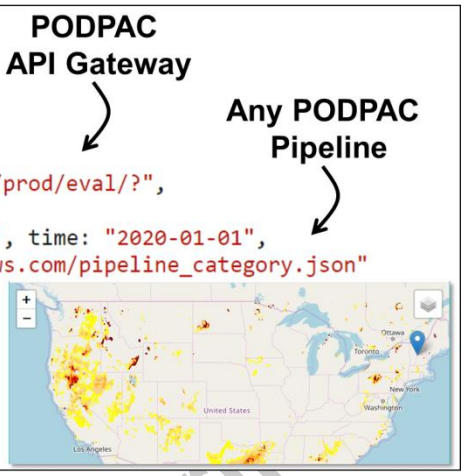

Figure 5. Example WMS usage with PODPAC Lambda function. This JavaScript code creates the overlay in the map shown in the inset.

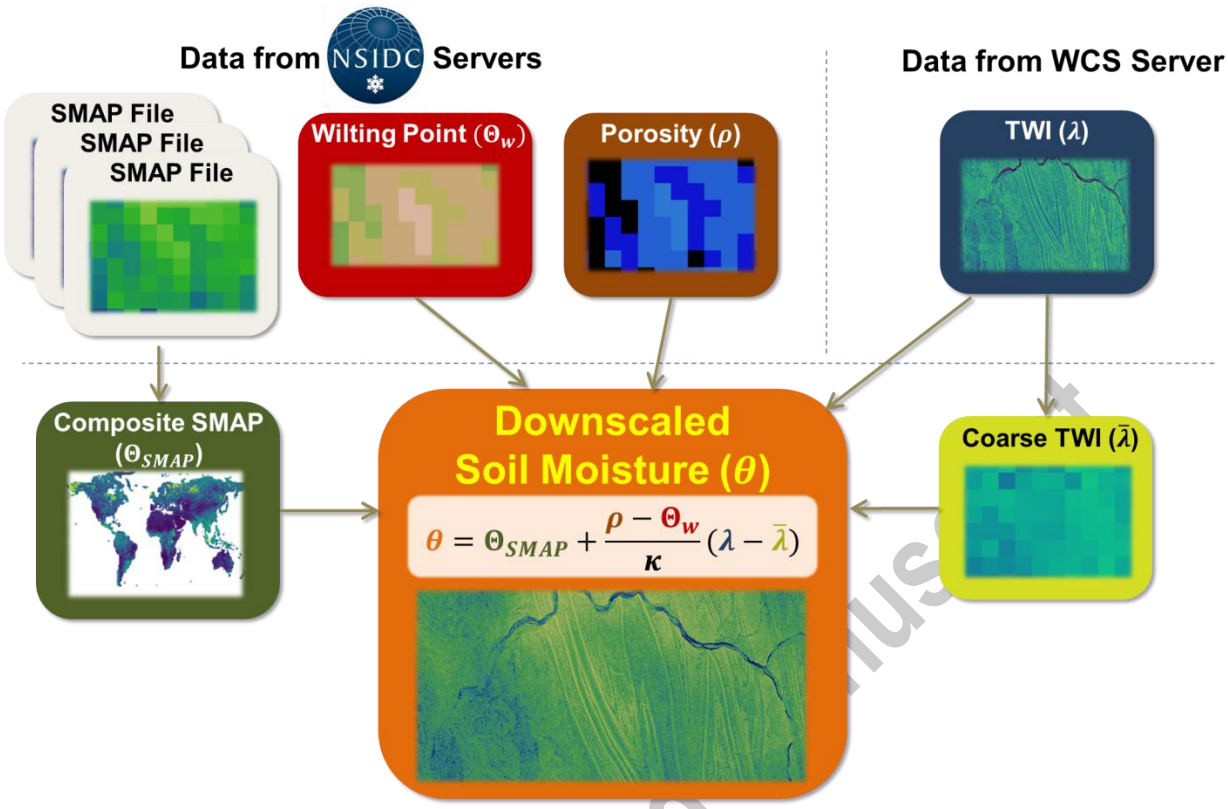

Figure 6. Example plug-and-play algorithm development enabled through automatic data harmonization.

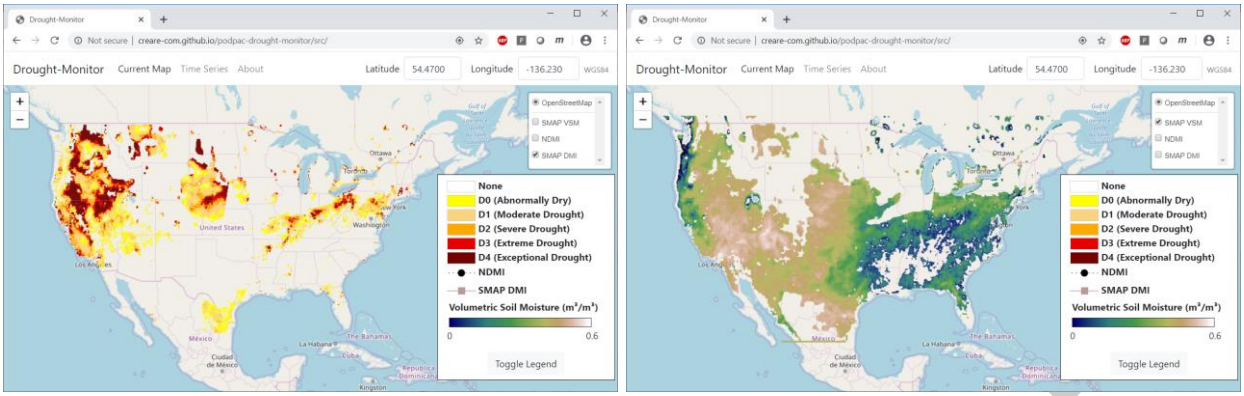

Figure 7. User interface of SMAP Drought Monitoring Index application developed using PODPAC.

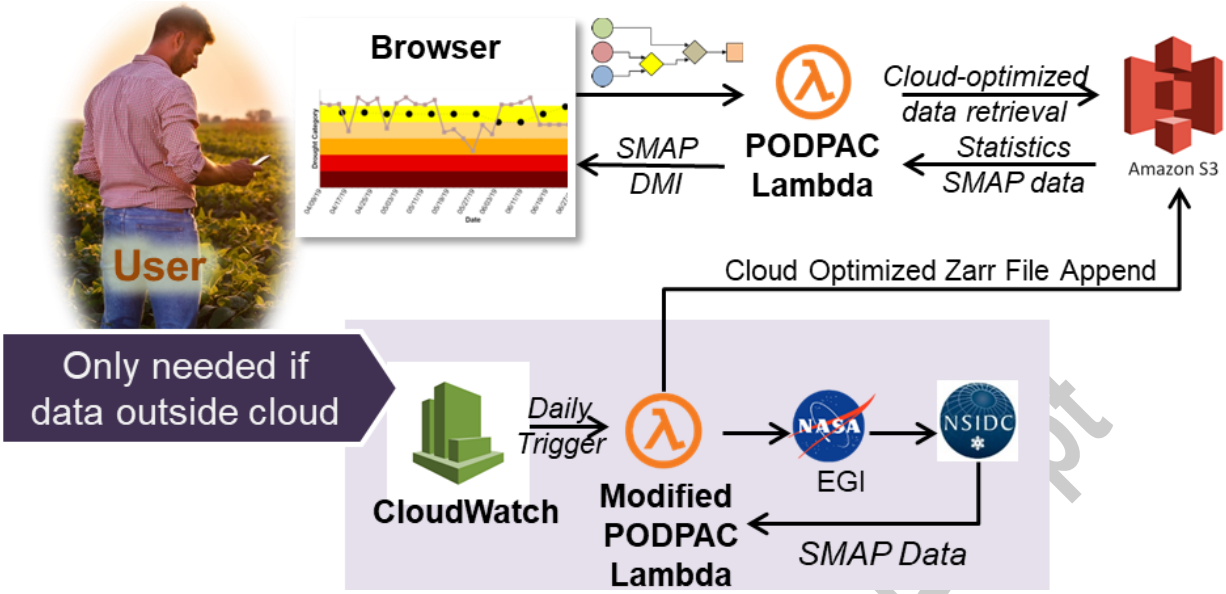

Figure 8. SMAP DMI application architecture deployed using POPDPAC Lambda functions.

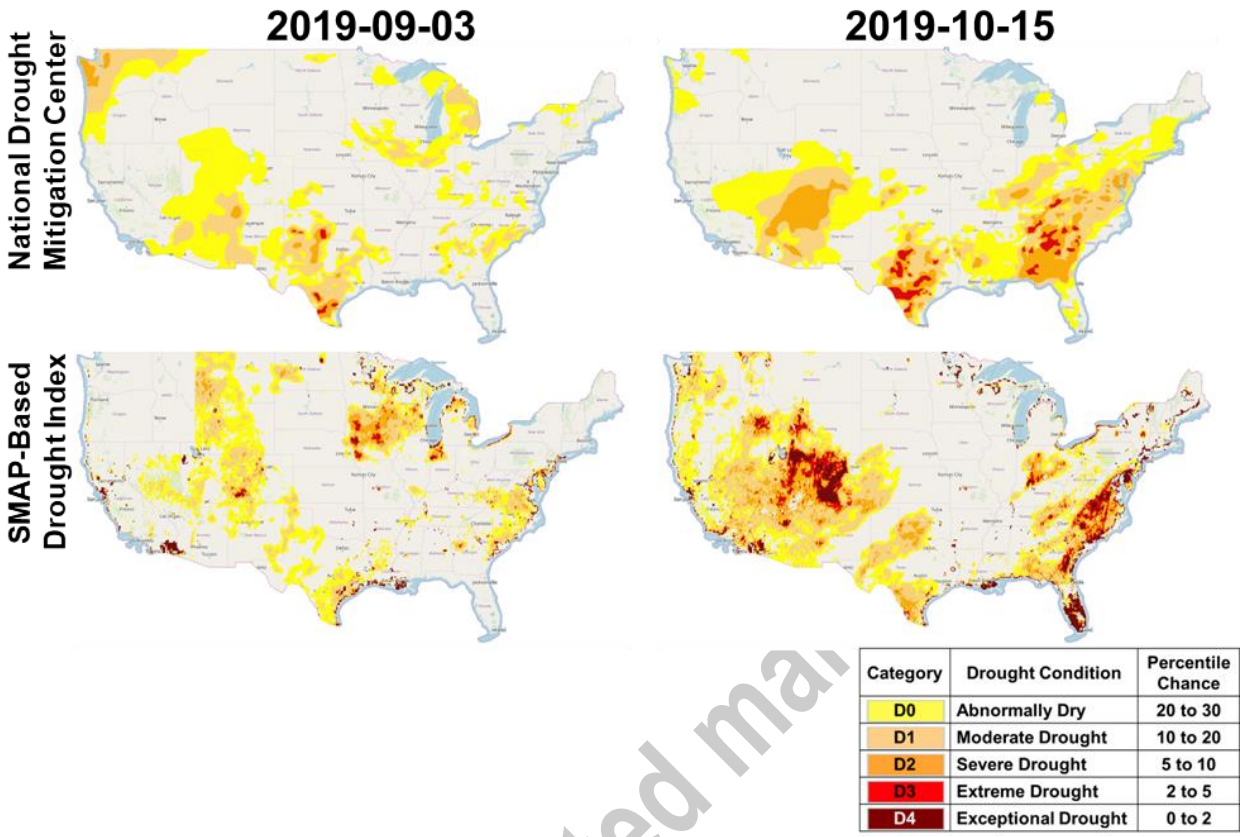

Figure 9. Comparison of SMAP DMI Monitor Index with National Drought Mitigation Center's NDMI.

13. Tables and captions

Table 1. Methods for Coordinate Creation in PODPAC.

Example (grid size)	Python Code
Grid in Space (3, 2)	`Coordinates([(4, 2, 1), (-4, -0)], dims=['lat', 'lon'])`
Time Series at Points (31, 3)	`Coordinates([('2019-01-01', '2019-01-31'), ([1, 2], [2, 3])], dims=['time', ('lat', 'lon')])`
Path in Space-Time (3)	`Coordinates([([4, 2], [1, 2], ['2019-01-01', '2019-01-31'])], dims=[('lat', 'lon', time')])`

Table 2. Basic examples introducing users to PODPAC.

Notebook Example	Description
001-open-raster-file	Open a locally stored file containing raster data
002-open-point-file	Open a .CSV file stored locally
003-combining-data-in-algorithm	Combine local data sources in a processing algorithm
004-load-array-data	Create an in-memory data source
010-retrieving-SMAP-data	Retrieve remotely stored SMAP data from NSIDC DAAC
020-using-coordinates	Create and manipulate PODPAC Coordinates
021-composite-array-datasources	Create a new data source that merges multiple other sources
100-analyzing-SMAP-data	Use remotely stored SMAP data in a processing algorithm
101-working-with-SMAP-Sentinel-data	Discover where SMAP-Sentinel tiles are available
200-running-on-aws-lambda	Run a processing algorithm in the cloud
300-parallel-processing	Run algorithm over multiple processes in parallel

Table 3. Implementation of specific data source in PODPAC

datalib Source	Generic PODPAC Nodes	File Storage Structure
smap_egi.SMAP	datalib.egi.EGI	API query
smap.SMAP	compositor.OrderedCompositor, data.PyDAP	date/globe_at_time.h5
terraintiles.TerrainTiles	compositor.OrderedCompositor, data.Rasterio	tile_format/zoom/x/y.tif
gfs.GFS	data.Rasterio	param/level/date/hour/forecast.grib2

14. Software files

Source code for PODPAC can be downloaded from GitHub at https://github.com/creare-com/podpac.