BIDS INCREMENTAL, BIDS RUN, AND BIDS ARCHIVE: INTEGRATING THE BRAIN IMAGING DATA STRUCTURE WITH RT-CLOUD, A REAL-TIME FMRI CLOUD PLATFORM

STEPHEN POLCYN

ADVISOR: PROFESSOR KEN NORMAN (PNI)

PROFESSOR KAI LI (COS), SECOND READER

GRANT WALLACE (COS), PROJECT MENTOR

APRIL 2021

Abstract

Real-time fMRI is a powerful neurofeedback technique that enables novel research and medical treatments using hardware already common in research institutions and hospitals. This thesis excerpt presents the implementation of data structures that integrate the Brain Imaging Data Structure (BIDS), the leading neuroscience data standard, with RT-Cloud, a real-time fMRI cloud platform under active development at Princeton University. Taken together, these data structures enable researchers to easily collaborate on experiments and datasets, more quickly understand RT-Cloud, and connect the system with familiar, BIDS-compatible software packages, all while meeting real-time fMRI's strict performance requirements.

Contents

Abstract **ii**

1 Implementation of BIDS Streaming: BIDS Incremental and BIDS Run **1**

1.1 Rejected Approaches 1

1.1.1 DICOM Streaming 1

1.1.2 BIDS Standard Modification 1

1.1.3 BIDS Archive Packing 2

1.2 BIDS Incremental: Design 2

1.3 BIDS Incremental: Implementation 4

1.3.1 Data Properties 4

1.3.2 Metadata API 5

1.3.3 Archive Emulation API 5

1.3.4 Archive Writing API 5

1.4 BIDS Run: Design 6

1.5 BIDS Run: Implementation 7

2 Implementation of BIDS Appending and Querying: BIDS Archive **8**

2.1 Rejected Approaches 9

2.2 BIDS Archive: Design 10

2.3 BIDS Archive: Implementation 10

2.3.1 Querying the BIDS Archive 10

2.3.2 Getting BIDS Runs 11

2.3.3 Appending BIDS Runs 12

2.3.4 Integration with PyBids' BIDS Layout 13

Contents

1. Implementation of BIDS Streaming: BIDS Incremental and BIDS Run

Streaming BIDS data is a key part of all three workflows, whether from the MRI machine to the analysis model and long-term BIDS archive storage or from an existing BIDS archive to an analysis model while testing. This section first summarizes approaches considered for streaming BIDS data, then details the design and implementations of the BIDS Incremental and BIDS Run data structures I created to meet the functional and performance requirements outlined above.

1.1. Rejected Approaches

1.1.1. DICOM Streaming The approach the current generation of RT-Cloud uses is to stream the DICOM images to the ProjectServer in the cloud. While this approach has the advantage of simplicity and does meet the network transport performance requirement, it fails to meet the other requirements for the next generation of RT-Cloud.

1. **Data**: DICOM holds raw image data and can hold substantial amounts of metadata. However, it is not BIDS-aware and does not hold dataset metadata.
2. **Compatibility**: DICOM is not compatible with a BIDS Archive until it is converted to NIfTI and the file is renamed according to BIDS naming conventions.
3. **On-Disk Interactions**: External tools and manual input are needed to convert a DICOM to a BIDS Archive, which consists of DICOM to NIfTI conversion and substantial metadata processing to figure out where to place the data in an existing BIDS Archive.
4. **Common Currency**: DICOM is not supported by BIDS apps, and lacks the BIDS awareness needed by all components of the analysis pipeline.
5. **Network Transport**: DICOM can be efficiently transported.

1.1.2. BIDS Standard Modification Another approach considered was to modify the BIDS standard to create a streaming format for real-time fMRI data. This would, by definition, meet all the requirements after it was eventually merged into the standard. However, modifying an at-rest

standard to support streaming is a fundamental change that would take some time, which would hinder RT-Cloud's goal of quickly making real-time fMRI accessible to a wide variety of users. Additionally, just modifying the standard would not provide an implementation, so the software would still need to be designed even after the modification process was completed, further delaying deployment.

1.1.3. BIDS Archive Packing A final approach considered was simply taking BIDS Archives, collecting the pieces to be streamed into a new directory on disk, and then compressing the directory and sending it to the downstream destination. At the destination, it could be decompressed to disk and the pipeline component could work on it like a standard, on-disk BIDS Archive. Additional methods would be created for merging these smaller, incremental BIDS Archives with the larger BIDS Archives that are the final output of the experiment, for extracting the smaller, incremental BIDS Archives from existing datasets for streaming, and for packaging the DICOMs coming off the scanner into a BIDS Archive containing a single image.

The primary disadvantage of this approach is in performance. The amount of time needed to copy files to new directories, compress the directories, send them over the network, then decompress them and write them back to disk before doing any operations on them would substantially reduce the time available for analysis. However, the central idea of a single-volume BIDS Archive is the idea that guided the approach selected.

1.2. BIDS Incremental: Design

To take advantage of the benefits of standardization already offered by the BIDS Archive format, like clear formatting requirements and a large ecosystem of tools to interact with BIDS datasets, I implemented "BIDS Incremental," a data structure that efficiently encapsulates a single-volume BIDS archive. This data structure is used for single-image streaming in the real-time fMRI workflows, such as when an image is output by the MRI machine during a scanning run, or when a single image is extracted from a BIDS archive to be used in a testing workflow. A graphical summary of BIDS Incremental's design and interaction modes is shown in Figure 1.

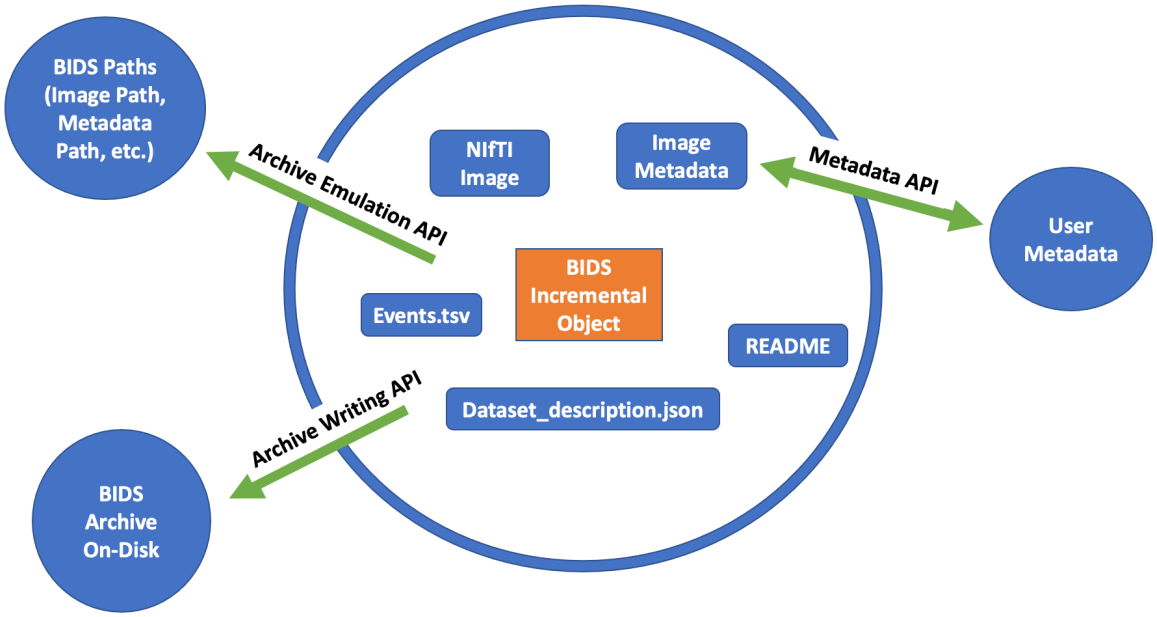

Figure 1: Diagram of the BIDS Incremental's data members and typical client interaction modes.

The data structure contains a NIfTI image, an image metadata dictionary describing the image, and several supporting data structures that map to files in a BIDS archive, namely the events file, the README, and the dataset_description.json, the details of which are not important here, other than that each file is required by the BIDS standard. All of this data is stored in memory to enable fast access and easy packing/unpacking on either side of a network connection, with no disk access required. To enable the BIDS Incremental to emulate a BIDS archive, I designed the Archive Writing API, which enables the BIDS Incremental's data to be written out to disk as a fully compliant BIDS archive, and the Archive Emulation API, which creates the names and paths that correspond to the names and paths the data in the BIDS Incremental would have if they were in an on-disk BIDS archive. I also included a Metadata API which helps users to modify the BIDS Incremental's metadata by checking modifications to ensure the BIDS Incremental's metadata stays compliant with the BIDS standard.

1.3. BIDS Incremental: Implementation

As Python is widely used both in computational neuroscience and outside the research community, I followed the RT-Cloud convention and implemented all data structures in Python, including BIDS Incremental, to maximize their accessibility, This choice improves the ability of researchers and clinicians to quickly get up to speed on and begin using the RT-Cloud framework. A class diagram of the BIDS Incremental object implementation is shown in Figure 2, and the reminder of this section describes the class's properties and methods in more detail.

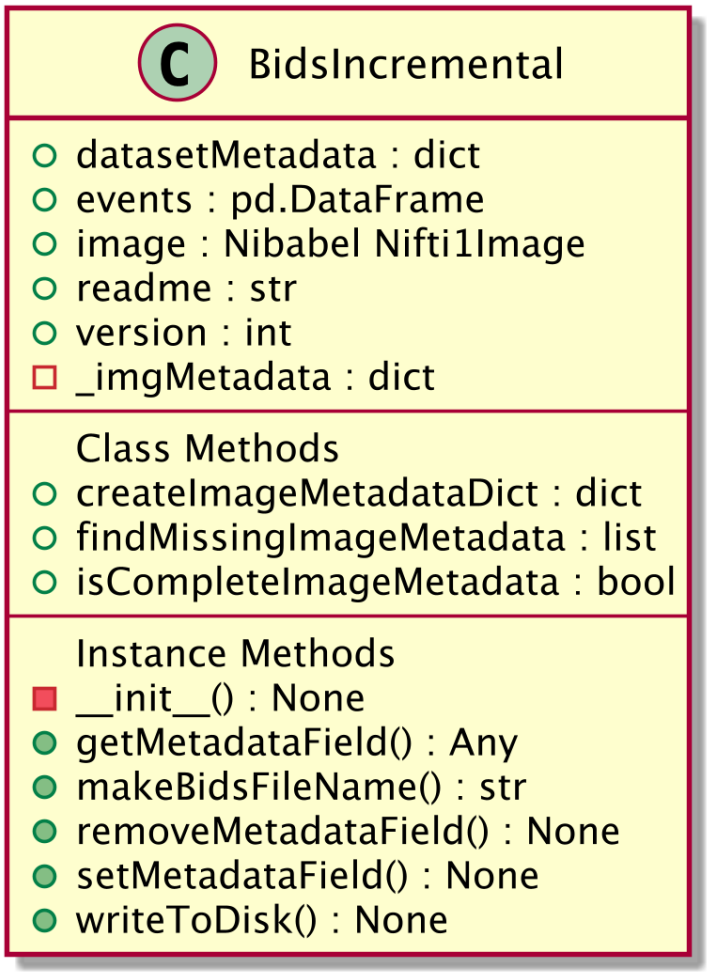

Figure 2: Class diagram of the BIDS Incremental data structure.

1.3.1. Data Properties The BIDS Incremental object has five primary data properties, which are listed and described here:

1. **NIfTI Image**: The NIfTI image is managed using Nibabel, which provides an expansive set of tools for managing neuroimaging formats in Python. It provides efficient memory-mapped I/O for on-disk image files (using Numpy) with a built-in caching mechanism, serialization and deserialization of images to a byte stream, and intuitive access to many properties of the image data and its header.
2. **Metadata**: Metadata is stored in a Python dictionary for efficient modification.
3. **Events File**: The events file is a tab-separated file on disk. Thus, it is stored as a Pandas DataFrame in the BIDS Incremental object for easy modification by users.
4. **README**: The README is stored as a simple text string.
5. **dataset_description.json**: The dataset description is stored as a Python dictionary for efficient modification, and it is converted to JSON only when the BIDS Incremental is written out to disk.

1.3.2. Metadata API The Metadata API is composed of *get, set,* and *remove* methods to add, modify, and delete existing metadata in the BIDS Incremental. These methods validate user changes to metadata to ensure the BIDS Incremental always has all required metadata to maintain standards compliance with BIDS.

1.3.3. Archive Emulation API The Archive Emulation API provides methods to get on-disk filenames and paths for the data in the BIDS Incremental, even when it's stored in memory. For example, querying the BIDS Incremental for the path of the image with *makeBidsFileName* might return a string like "sub-01/ses-02/func/sub-01_ses-02_task-test_bold.nii", indicating the path of the image in a BIDS archive if the data were written out to disk. The path's BIDS entities state that the image belongs to the first subject's second session, is a functional imaging run using BOLD contrast, and was for the 'test' task. The paths and filenames are computed automatically from the BIDS Incremental's metadata dictionary, enabling them to reflect any changes made to the BIDS Incremental's metadata by users during runtime. This API helps users compare the BIDS Incremental object to on-disk archives, and it is also used internally by the Archive Writing API.

1.3.4. Archive Writing API The Archive Writing API provides a single method, *writeToDisk*, which writes the data in the BIDS Incremental out to a provided location on disk in BIDS archive

format. This method is the primary link between a BIDS archive and a BIDS Incremental, and enables the BIDS Incremental to link with any other software system that expects on-disk BIDS data.

1.4. BIDS Run: Design

BIDS Run is a data structure based on the insight that real-time fMRI users typically work one scanning run at a time, and thus certain performance optimizations can be made without restricting the platform's capabilities. Concretely, the primary motivation for my creating this second data structure for streaming is avoiding disk operations during the time-sensitive critical path of the real-time fMRI workflows. Relying solely on BIDS Incremental to fulfill the streaming portion of the workflows would require a disk operation during each TR, either for appending a BIDS Incremental to a BIDS Archive object (described in Section 2), or for getting a BIDS Incremental from a BIDS Archive object. Appendix **??** explores the negative performance implications of using this method. In short, the performance issue is that using just BIDS Incremental requires re-creating an increasingly large NIfTI file on-disk during each TR, which results in the runtime for append operations being proportional to the number of BIDS Incrementals processed so far rather than constant. This would severely limit the size of datasets RT-Cloud could work with.

BIDS Run entirely solves this performance issue by keeping all data in-memory and batching expensive operations to minimize disk I/O. Additionally, it enables certain expensive operations (like writing or reading a run's data from disk) to be done before or after the time-sensitive section of a real-time fMRI workflow, massively increasing the dataset size RT-Cloud can work with.

At a high-level, BIDS Run is a data structure that efficiently stores a full run's worth of BIDS Incrementals in-memory and in a deduplicated fashion (during a run, BIDS Incrementals typically share most of their metadata). Its key operations enable adding a BIDS Incremental to the BIDS Run data structure and retrieving a BIDS Incremental from the data structure. The workflow for creating a BIDS Run is shown in Figure 3, and the opposite workflow for getting BIDS Incrementals is simply the reverse of what the diagram shows.

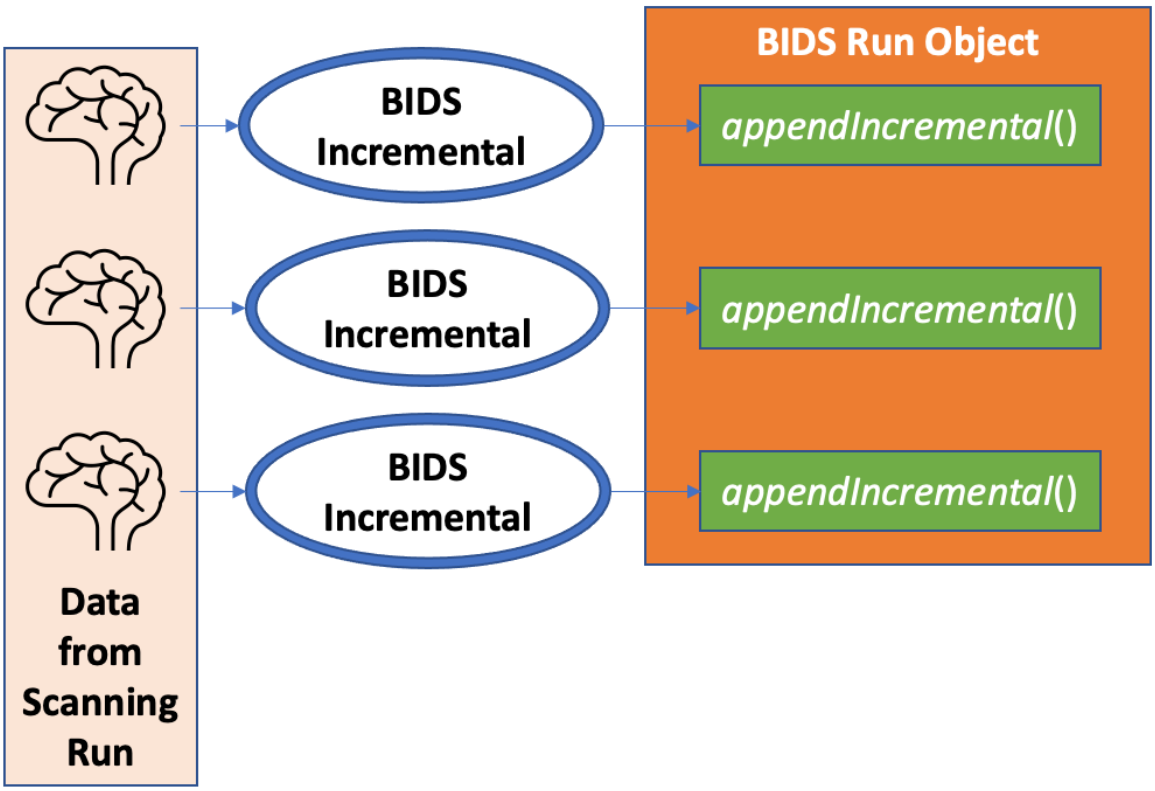

Figure 3: Diagram showing the process for creating a BIDS Run. BIDS Incrementals are created as the scanner outputs brain data, and each BIDS Incremental is then appended to the BIDS Run as it arrives over the network.

1.5. BIDS Run: Implementation

The BIDS Run API is shown in Figure 4, and it foreshadows the BIDS Archive API presented in Section 2. As mentioned previously, the BIDS Run data structure stores only one copy of each metadata element, due to the duplication of metadata across BIDS Incrementals in the same run. The image data, which is always different between BIDS Incrementals, is stored as a Python list of Numpy arrays, enabling efficient, copy-free appends to the BIDS Run data structure, regardless of how many arrays of image data are added. Since the metadata data is stored in de-duplicated form and the image data in raw Numpy array form, when *getIncremental* is called, a BIDS Incremental object is quickly created that packages the appropriate deduplicated metadata and data array. To add data to a BIDS Run, *appendIncremental* is used. This method takes care of extracting the relevant raw data, as well as checking that the metadata matches the existing data in the BIDS

Run to ensure users don't accidentally mix their datasets. To coalesce all the data in a BIDS Run into a single BIDS Incremental (e.g., for writing a full BIDS Run to one file or a BIDS Archive), *asSingleIncremental* efficiently concatenates all the raw data arrays together, then packages this overall data array with the metadata into a BIDS Incremental.

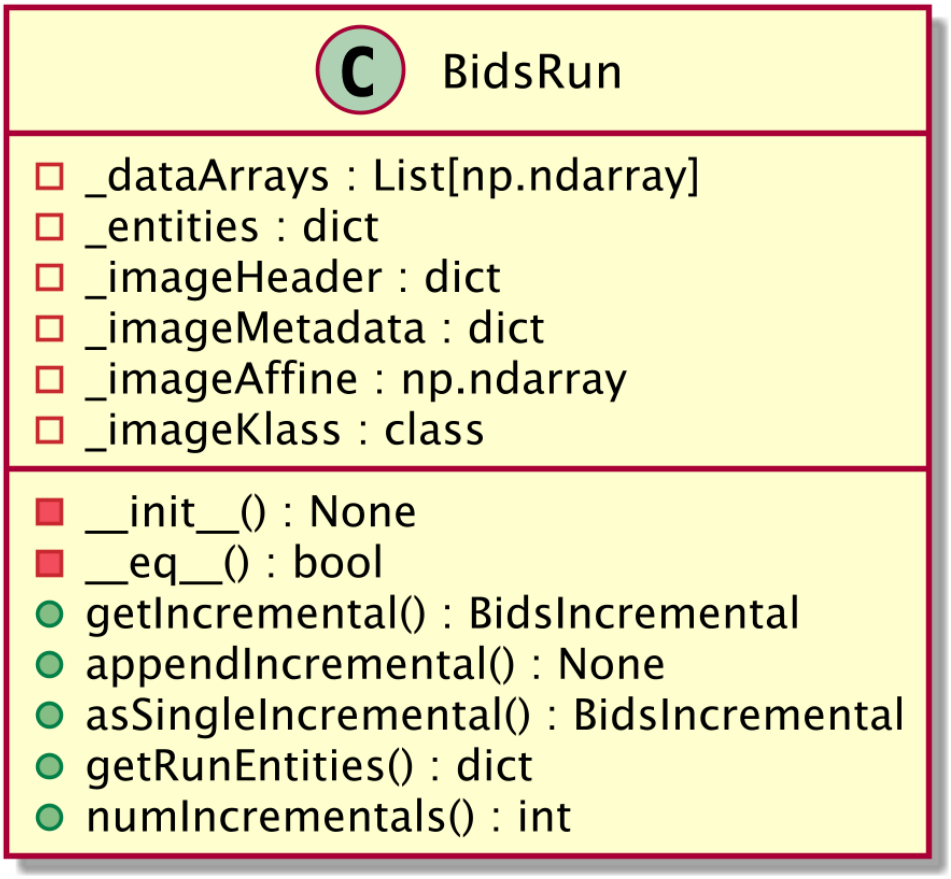

Figure 4: Class diagram of the BidsRun API.

2. Implementation of BIDS Appending and Querying: BIDS Archive

To support the appending and querying operations for the real-time fMRI workflows, I created BIDS Archive, a data structure that provides an easy-to-use API for interacting with on-disk BIDS archives and enables efficient movement between the streaming data structure BIDS Run and the on-disk BIDS archive.

2.1. Rejected Approaches

This section motivates my solution to implementing the Appending and Querying operations by describing the approaches I considered and rejected for interacting with BIDS data on-disk.

1. **Get/Append Only Data Structure**: The primary need motivating a data structure to interact with BIDS archives on-disk is the need to get and append streaming BIDS data from them. Thus, a data structure that implemented only these operations, or even including these operations in the streaming data structure, would minimize the amount of new code required. However, all modules of RT-Cloud by design frequently interact with BIDS data, and the get/append operations themselves require in-depth interactions with BIDS data (e.g., searching for images and metadata to append to or to return in streaming form), so a more full-featured data structure was required.
2. **PyBIDS' BIDSLayout Class**: PyBIDS is a software package produced by the BIDS Standard maintainers that provides a set of utilities for interacting with and manipulating BIDS data [2, 1]. The package includes a BIDSLayout class designed for interacting with BIDS archives. While BIDSLayout has an expansive API that provides many functions for querying an on-disk BIDS dataset, our goal is to provide a limited and quick-to-learn API to real-time fMRI users. Unsurprisingly, PyBIDS does not support getting or appending streaming data structures from a BIDS archive, so this approach also would have required significant extension.
3. **Custom BIDS Archive Class**: A final option was to implement a custom BIDS Archive class from scratch that presented a limited API and also supported the get and append operations. While this would, by definition, meet all of the requirements, the engineering work required to build, test, and maintain such a major set of functions was infeasible. Additionally, much of the functionality required already had solutions within PyBIDS, which indicated the best solution would take advantage of the existing work and focus on new functionality.

2.2. BIDS Archive: Design

The design I chose for BIDS Archive merges the last two approaches, and it is a custom class that employs the software design technique of composition to take advantage of the functionality provided by PyBIDS' BIDSLayout while also adding the new API methods *getBidsRun* and *appendBidsRun*. The high-level API methods and class interactions are shown in Figure 6, and the key parts are as follows:

1. **General Query of BIDS Archive Data**: Methods are provided for common user queries, like getting a set of images from the BIDS Archive, or getting the images' accompanying metadata.
2. **Query and Append of Streaming Data Structures**: Methods are provided to retrieve the BIDS Archive's data in BIDS Run form and add the data contained in a BIDS Run to the BIDS Archive.
3. **BIDS Layout Integration**: The BIDS Archive class integrates seamlessly with the PyBIDS BIDSLayout class, enabling users to directly call BIDSLayout methods from a BIDS Archive object, thereby giving users access to all functionality of the BIDSLayout object, both present and future.

The next section describes the implementation of each of these features in more detail.

2.3. BIDS Archive: Implementation

2.3.1. Querying the BIDS Archive Querying the BIDS Archive is done using the same general approach as PyBIDS to ensure users can easily switch between RT-Cloud classes and PyBIDS classes, and users already familiar with PyBIDS can quickly start using RT-Cloud. The approach is to use "BIDS Entities" (as described earlier in Section **??**) to search the files present in the BIDS Archive. As a reminder, a BIDS Entity is a piece of metadata that describes a file (like which subject, session, or run the file corresponds to), and all filenames in a BIDS Archive are formed from BIDS entities.

To support querying the BIDS Archive using BIDS Entities, a method accepts a list of entities to filter the files in the BIDS Archive with. For example, calling *getImages(sub='01')* will return all the files in the BIDS Archive that are related to Subject #01. Additional entities can be added

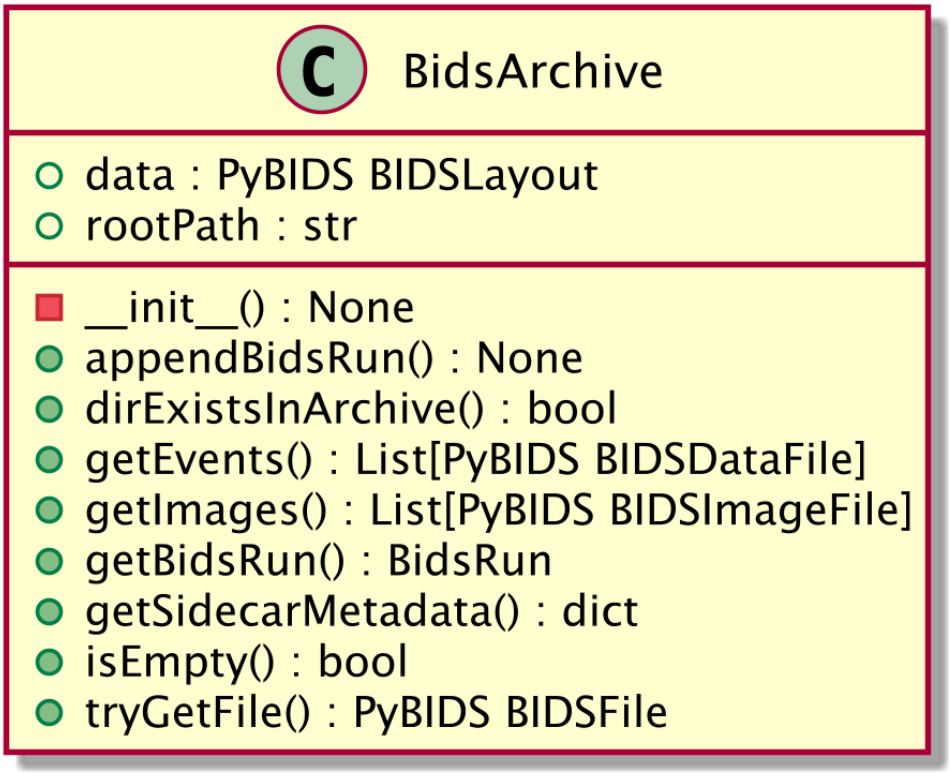

Figure 5: Class diagram for the BidsArchive class showing the primary properties and API methods. PyBIDS is a Python package written and maintained by the BIDS Standard maintainers that provides several useful classes for interacting with BIDS data.

in a function call to further specify the target files. For example, calling *getImages(sub='01', task='language', run=2, datatype='func', suffix='bold')* will return only fMRI BOLD data for the second run of Subject #01 performing the language task. Because all files in the BIDS Archive conform to the BIDS standard and use BIDS entities in their filenames, this method is applicable to all BIDS Archives, and the filtering approach allows a wide degree of specificity in queries.

2.3.2. Getting BIDS Runs Getting data from a BIDS Archive in BIDS Run format has a very similar API to querying the BIDS Archive. Users supply a set of BIDS entities in exactly the same way as when querying the BIDS Archive for other data. For example, calling *getBidsRun(sub='01', task='language', run=1)* will return all the data for Subject #01's first run of the language task. This enables all disk operations to be concentrated at one time so that future operations, done on the BIDS Run, can all be in-memory and fast.

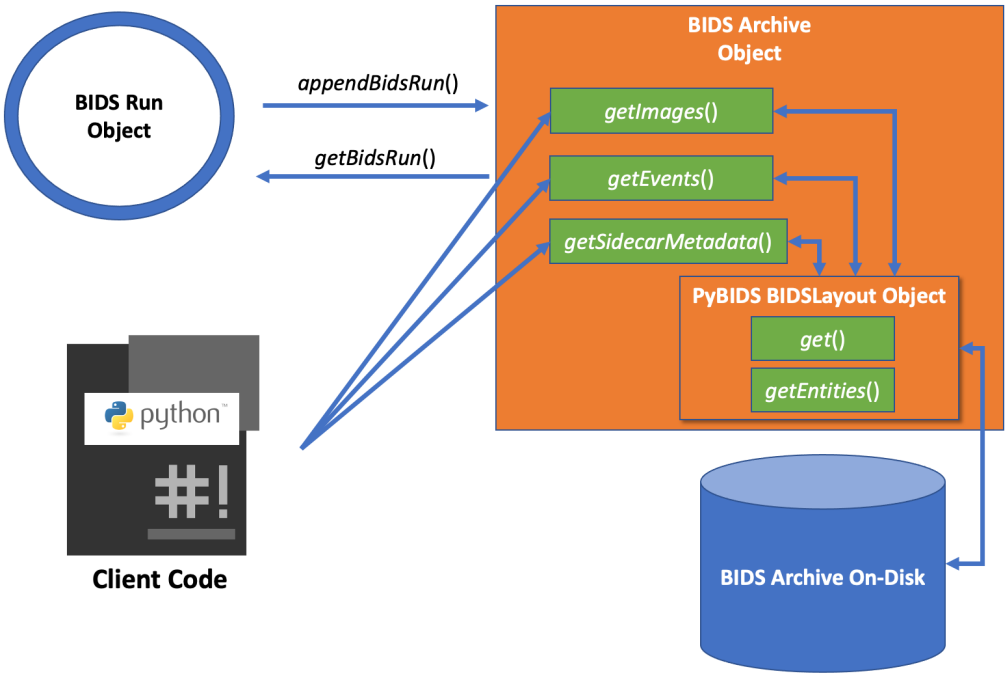

Figure 6: Design of the BIDS Archive API. Using *appendBidsRun()*, a BIDS Run can be added to the BIDS Archive, and using *getBidsRun()*, a BIDS Run can be extracted from the BIDS Archive. The BIDS Archive object also presents several methods for retrieving other common data, like images, event files, and image sidecar metadata files, all of which can be called by the experimenter or clinician's code. Under the hood, some of these operations rely on a BIDSLayout object (included in the PyBIDS package), which the BIDS Archive maintains a reference to.

2.3.3. Appending BIDS Runs When appending a BIDS Run to a BIDS Archive, there are two cases that can occur:

1. **No file exists corresponding to this BIDS Run**: In this case, no previous data matching the BIDS Run has been added to the archive, so the BIDS Run's data is written to the BIDS Archive as new NIfTI and metadata files. Afterwards, the underlying BIDSLayout object is updated with the existence of the new files.
2. **File exists corresponding to this BIDS Run**: This happens less frequently, but it's possible for a user to append data to an existing run. For example, the BIDS Run may have data for Subject 01's first run of the 'language' task, which the BIDS Archive already has data for. To execute the append, validation is first performed to ensure the metadata in the BIDS Run matches the data on disk to prevent accidental mixing of datasets. Next, the data on disk is combined with the data in the BIDS Run to form a single NIfTI file. Finally, everything is written back to disk.

Appending a BIDS Run, like getting a BIDS Run, concentrates all disk I/O in a single operation, ensuring that minimal copying and read/write from disk is done when adding data to a dataset. Figure 7 shows the full workflow for using a BIDS Run with a BIDS Archive.

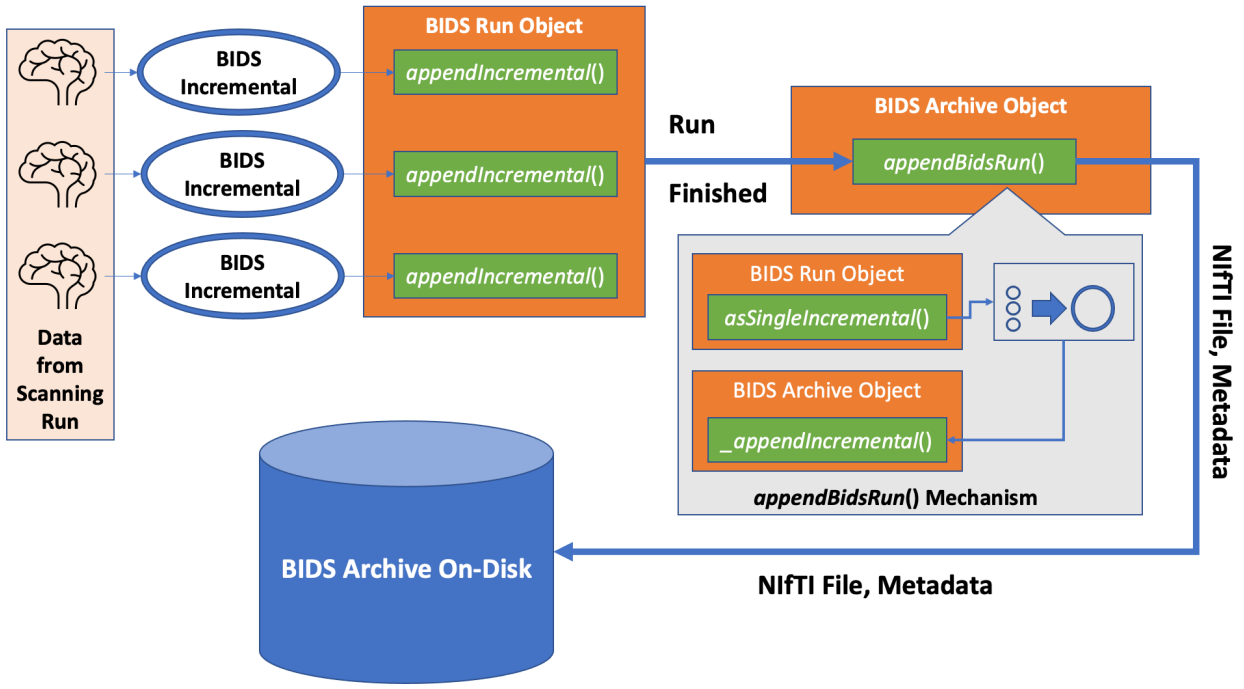

Figure 7: Diagram showing the BIDS Run workflow. BIDS Incrementals are created from the brain data obtained during the scanning run. Then, the BIDS Incrementals are appended to the BIDS Run as they arrive. Then, upon conclusion of the run, the BIDS Run is appended in one shot to the BIDS Archive and written to disk. Under the hood, the append mechanism is to use the output of BIDS Run's *asSingleIncremental* method as input to a private BIDS Archive helper method called *_appendIncremental*, which then executes the append and disk write.

2.3.4. Integration with PyBids' BIDS Layout The PyBIDS BIDSLayout object provides many useful methods for interacting with BIDS datasets on disk. While some of those methods have analogues in the BIDS Archive (e.g., BIDS Archive's *getImages* is implemented using the highly general *get* method from BIDSLayout), there are many other methods that BIDS Archive doesn't provide a convenient wrapper for. Instead, BIDS Archive provides a remapping functionality so that when a user tries to call a BIDSLayout method using a BIDS Archive, the BIDS Archive realizes it doesn't implement the requested method and transparently redirects the call to the underlying

BIDSLayout object. This enables users to always use the BIDS Archive object directly, even when accessing BIDSLayout features.

This seamless remapping functionality was used to design a BIDS Archive class that maximizes use of already-written PyBIDS code, while also adding the *getBidsRun* and *appendBidsRun* functionality. Additionally, using this approach enables the initialization of an empty BIDS Archive which can then be populated using *appendBidsRun*, as a BIDSLayout cannot be created with an empty dataset. This feature is useful at the beginning of an experiment when no data has been collected yet, as the user can open a BIDS Archive where they want to store the upcoming data, and *appendBidsRun* takes care of initializing and updating the BIDSLayout under the hood when the first data arrives.

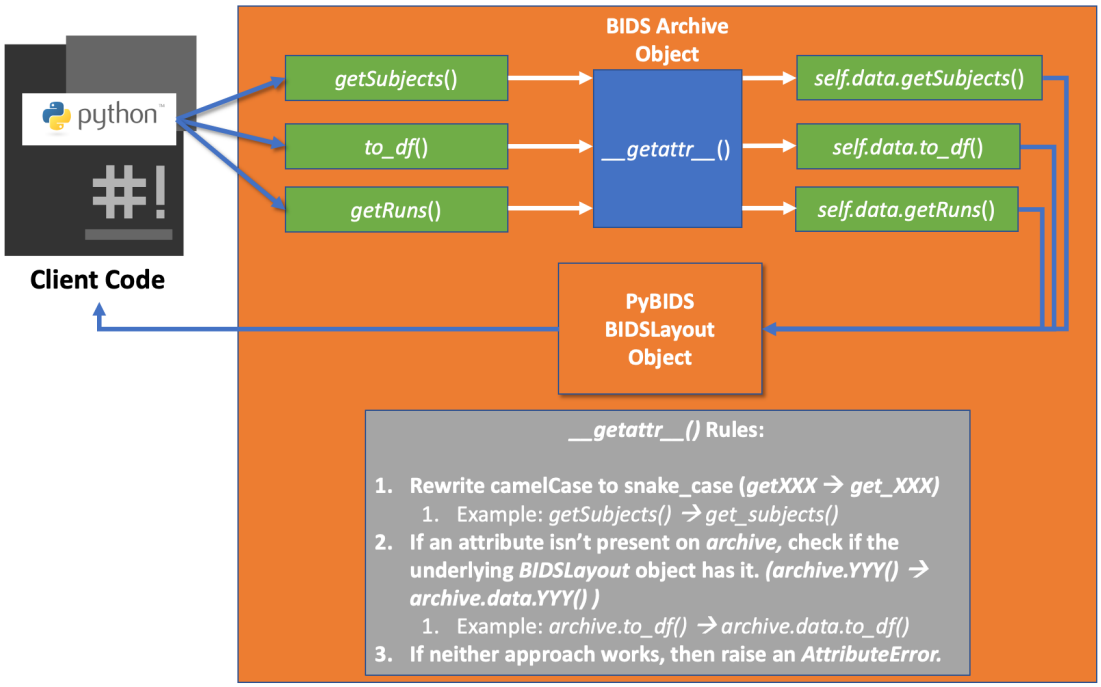

Figure 8: The remapping functionality of BIDS Archive, enabling clients to seamlessly call methods that BIDS Archive doesn't implement, but that the underlying BIDSLayout does.

References

[1] T. Yarkoni, C. J. Markiewicz, A. de la Vega, K. J. Gorgolewski, Y. O. Halchenko, T. Salo, Q. McNamara, K. DeStasio, J.-B. Poline, D. Petrov, V. Hayot-Sasson, D. M. Nielson, J. Carlin, G. Kiar, K. Whitaker, A. Wagner, E. DuPre, S. Appelhoff, A. Ivanov, J. Wennberg, L. S. Tirrell, O. Esteban, M. Jas, M. Hanke, R. Poldrack, C. Holdgraf, I. Staden, A. Rokem, B. Thirion, C. Boulay, D. F. Kleinschmidt, E. W. Dickie, J. A. Lee, M. Visconti di Oleggio Castello, M. P. Notter, P. Roca, and R. Blair, "Bids-standard/pybids: 0.9.3," Zenodo, Aug. 2019.

[2] T. Yarkoni, C. J. Markiewicz, A. de la Vega, K. J. Gorgolewski, T. Salo, Y. O. Halchenko, Q. McNamara, K. DeStasio, J.-B. Poline, D. Petrov, V. Hayot-Sasson, D. M. Nielson, J. Carlin, G. Kiar, K. Whitaker, E. DuPre, A. Wagner, L. S. Tirrell, M. Jas, M. Hanke, R. A. Poldrack, O. Esteban, S. Appelhoff, C. Holdgraf, I. Staden, B. Thirion, D. F. Kleinschmidt, J. A. Lee, M. V. O. di Castello, M. P. Notter, and R. Blair, "PyBIDS: Python tools for BIDS datasets," *Journal of Open Source Software*, vol. 4, no. 40, p. 1294, Aug. 2019.