Integration of the Java Image Science Toolkit with E-Science Platforms

Release 3.0

Stephen M. Damon[1], Sahil Panjwani[1], Shunxing Bao[1], Peter Kochunov[2] and Bennett A. Landman[1]

January 11, 2016

[1]Vanderbilt University, Department of Electrical Engineering and Computer Science

[2] University of Maryland, Maryland Psychiatric Research Center

Abstract

Medical image analyses rely on diverse software packages assembled into a "pipeline". The Java Image Science Toolkit (JIST) has served as a standalone plugin into the Medical Image Processing Analysis and Visualization (MIPAV). We addressed shortcomings that previously prevented deeper integration of JIST with other E-science platforms. First, we developed an interface for integrating externally compiled packages (similar to the interfaces in NiPy) such that the application can become a "draggable module" in the module tree. This allows for connection of inputs and outputs to other JIST modules while maintaining external processing and monitoring. Second, we develop an integration interface with the Neuroimaging Informatics Tools and Resources Clearinghouse Cloud Environment (NITRC-CE). User can launch and terminate pre-configured nodes to utilize computational resources of the Amazon cloud. Finally, we define a new external data source, which can connect to the eXtensible Neuroimaging Archive Toolkit (XNAT) to query and retrieve remote data using XNAT's REST API. Specifically, we define dataflow for files that can readily be converted into volumes and collections of volumes to interface with any JIST module that expects volumetric image data as an input. Users now have the ability to run their pipelines from a well-defined external data source and no longer are required to already have data on the disk. With these upgrades we have extended JIST's capabilities outside of complied java source code and enhanced capabilities to seamlessly interface with E-science platforms.

Contents

1 Introduction **2**

2 Integration with external CLI packages **3**

3 Integration with the NITRC CE **6**

4 Integration with XNAT **7**

5 Demonstration of new features **8**

6 Conclusions **9**

1 Introduction

The Java Image Science Toolkit (JIST)[1-3] is a modular and open-source resource for neuroimaging development and pipelining developed in the Java programming language[4]. JIST is a plugin to the National Institutes of Health Center for Information Technology (NIH/CIT) Medical Image Processing Analysis and Visualization (MIPAV)[5] which handles viewing and reading and writing of medical image file formats as well as processing and analysis tools. JIST has been downloaded over 13,400 times and remains listed in the top downloads and most active projects on NITRC's page.

JIST supports parallel processing and management of many different types of neuroimaging algorithms. However, as neuroimaging studies continue grow in complexity in terms of sequences and data collected, local hard drive storage is no longer sufficient to logically store and maintain file and processed data integrity. Additionally, as storage needs continue to grow, so does the need to process study data. Large scale neuroimaging storage and management platforms such as the eXtensible Neuroimaging Archive Toolkit (XNAT)[6], aim to simplify management of several to hundreds of projects in a logical storage management system while maintaining all information in a database such as PostgreSQL[7]. Additionally, a webapp allows users to log in and control user access, view processing results, set quality control (QC) statuses of both processing as well as input (i.e., raw scan) data.

With the advent and rapid adaptation of archival and management tools, such as XNAT, processing needs are now increasing and more computationally intensive processes are being run. Cloud computing technology gives users the ability to run massively parallel processing without the need to set-up, monitor, and maintain a grid. Furthermore, as computation time continues to drop in price, it now is feasible to run entire studies for several hundred dollars at once rather than running everything serially on local machines.

While both the initial JIST release ("JIST-I") and second release ("JIST-II") were well received, one missing gap is that JIST does not interface with E-science database systems or software platforms.

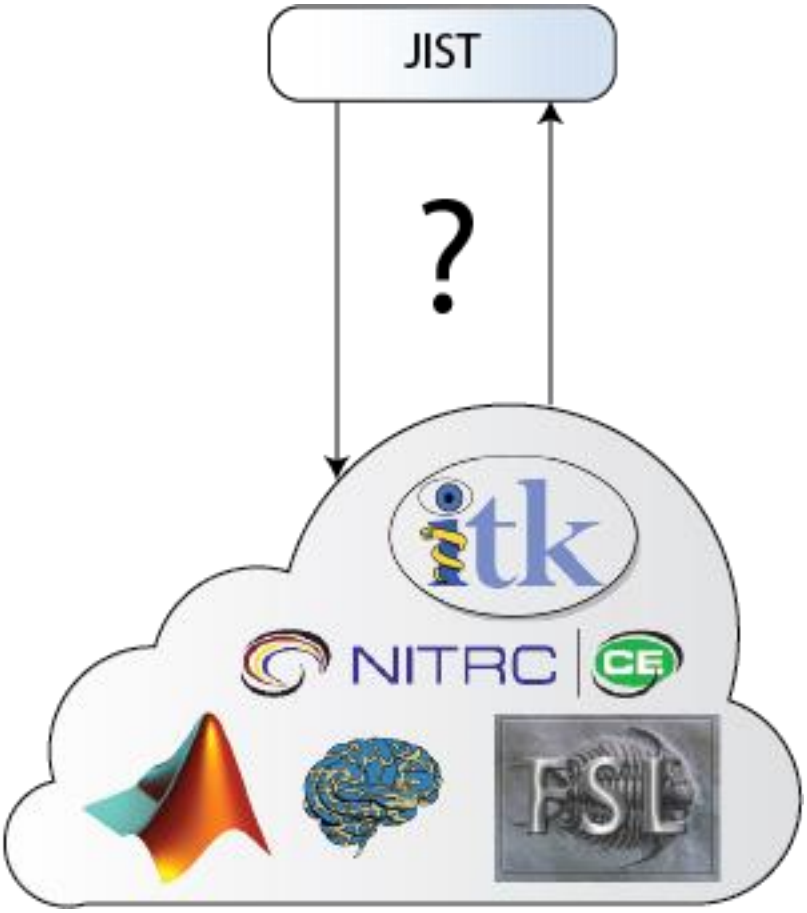


Figure 1: Depicting a missing link between JIST and running processing on the cloud.

In this manuscript, we discuss steps of integrating JIST with E-Science technology. More specifically:

1. We propose a generalized interface wrapper method (similar to interfaces in NiPy[8]) that allow the wrapping of any well-document CLI program to become a "draggable module" in the JIST Algorithm Tree.

    a. Many external programs have been developed to handle image processing tasks such as image registration, intensity normalization, and specialized image processing methods (like ITK)[9]. In the previous versions of JIST, these would have to be wrapped into complicated CLI calls to first run modules in JIST, then feed the outputs into the external programs and then feed the outputs back in to the final modules in JIST. This process was cumbersome and left room for improvement. By converting external CLI calls to modules, users are now able to seamlessly integrate external CLI packages as part of their pre-existing layouts and pipelines.

2. We discuss an interface into XNAT which allows for querying through the RESTful API to retrieve scan data remotely and inject into the JIST pipeline process.

   a. As neuroimaging studies grow, the need for logical and well-documented storage is required. Navigating through the XNAT web interface can be a bit complicated for new users or those just using publicly available datasets. Also, with a simple configuration (host URL, username, and password) users are able to query through XNAT without the need to code CURL commands or using PyXNAT methods. A user-friendly search UI lets the users query entire "subjects" or "projects", or even "sessions" in XNAT and download specific "resources" such as NIfTI images to run through JIST.

3. We discuss methods and implementation for running JIST on the NITRC-CE environment hosted by Amazon's AWS.

   a. With the growing collection of open-source multi-modal image repositories, the need to fork off processing into cloud-based environments is rapidly growing. Computationally demanding processing steps such as whole-brain probabilistic tractography may overwhelm locally available computational resources. Utilizing the power of the NITRC-CE hosted by Amazon's AWS, we are able to distribute each "Execution Context" (process) on a separate cloud node. Users are able to specify a grid size, which will automatically boot up and install JIST into MIPAV and set up all requirements for processing. All monitoring is managed through the standard JIST Process Manager. The I/O and data transfer process are automatically changed to maintain a streamlined flow in the cloud.

2 Integration with external CLI packages

To wrap an external CLI command into a "JIST external module", users need to define a directory containing the module. Under the "Global Preferences" pane, there is a text window for "External JSON Library Location". This, by default, will be called ext-jist-lib in the user's home directory unless otherwise specified. In order for these modules to be recognized, users need to wrap the CLI call into a JavaScript Object Notation[10] file (JSON). JSON modules should all end with a "json" file extension to be appropriately recognized. The module highlighted in yellow in Figure 2 is an example of the requirements for a module to be successfully compiled. In the expanded view, the JSON code that encapsulates this module is displayed for reference.

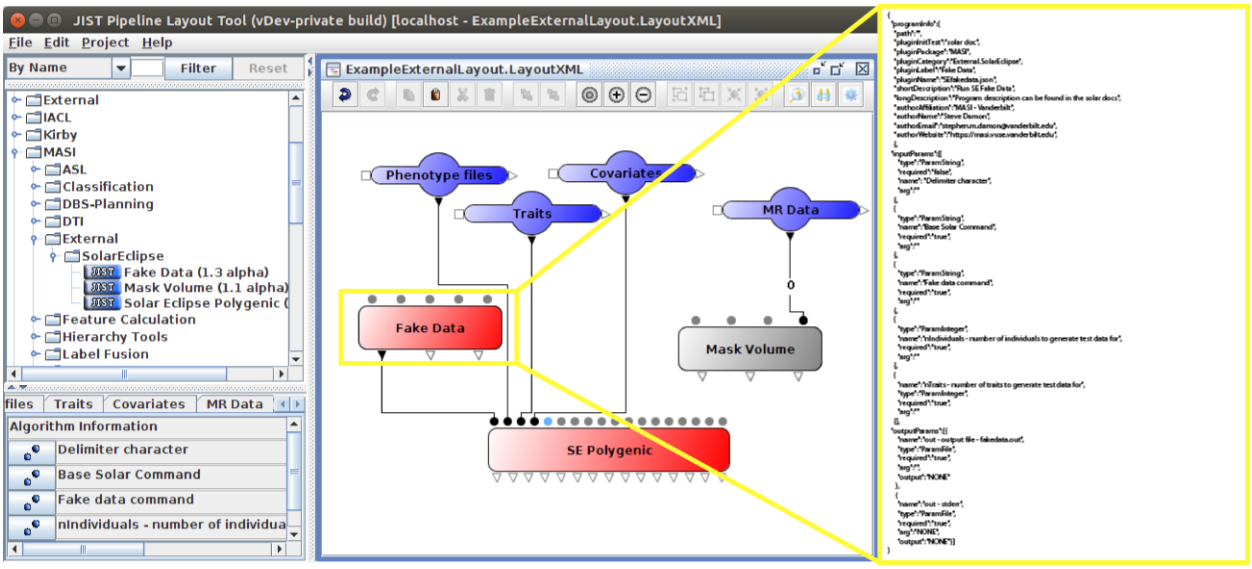


Figure 2: The JIST Pipeline Layout Tool showing some externally linked modules (left) and JSON module encapsulation (right).

As seen in the JSON code, "programInfo" provides high-level information about the module (i.e., author, email, and name for the module as it should appear in the algorithm tree). Other required information is enumerated below:

1. pluginInitTest –a simple method used to test if the program is in the system's path. Suggested methods would include a command like "flirt" (from FSL[11]) or "solar doc" for Solar Eclipse (https://www.nitrc.org/projects/se_linux/). During the module build process, this command will be executed so it is a good idea to make sure it is a quick test. If the command has a non-zero exit status, the module will be colored gray (see Figure 2) to indicate that it is unable to run. If the exit status is zero, then the color will be red, like other modules, indicating that it can run.

2. pluginPackage – the highest level in the module tree. As seen in Figure 2, this would appear under IACL, Kirby, and MASI etc. If the desired package does not exist, it will automatically be generated for the user(s).

3. pluginCategory – used to specify sub directories branching off from pluginPackage. If these folders do not exist under jist-lib, they will automatically be generated (Figure 2: See MASI, External). If multiple subdirectories are desired, the sub directories should be separated by periods (e.g., External.SolarEclipse).

4. path – can be used to give an absolute path to the executable if adding to path is not desired. Note that this is required, but can be blank. If path is included, it will be added on to your command. For example, if for some reason FSL was not set up in your path and you wanted to use bet, path would be "/usr/local/fsl/bin" and one of the input params would be ParamString with the value of "bet".

Each argument to the call is required to be a pin for the module. Many of the JIST param types are supported. Each of the inputs must encapsulate at least the following four parameters of information:

1. type – the type of parameter that it should be. Types include: PramString, ParamFile, ParamVolume, ParamInteger, ParamBoolean, ParamFloat, and ParamLong.

2. required – true/false (case sensitive). This specifies whether this specific pin is required to run the CLI call. Just like all other JIST modules, if the requirements are not met during execution, the module(s) will show up as "Not Ready" in the process manager.

3. name – a short description that will be used to label the input in the module. When user's click on the input pin, this text will be displayed just like all other JIST modules.

4. arg – this should be the argument with flag(s) (if needed). Some CLI programs do not require flags but require positional arguments. By making use of JSON arrays, there is implicit ordering for arguments so users should consider this when developing modules.

Outputs are required to encapsulate the following five parameters of information:

1. type – just like for the inputs, the outputs must also be specified. In this case, it really only makes sense for CLI programs to return a file or a volume so ParamFile and ParamVolume are supported.

2. name – a helpful name to give so when users click on the output port, the know what the output should be.

3. required – same as input. Specification of requirement for successful completion.

4. arg – same as input. Some CLI programs require args for outputs. If not specified, none will be used

5. output – the full path of the expected output file. If just a file name is given, the current output directory is assumed to be the path where the file will be located. This allows users to pass a "-d" or "–directory" argument which many CLI programs support.

Two additional outputs will automatically be added to your file (you are not required to add them in as output params in your JSON module). These are external.out and external.err, which are different from the standard debug.err and debug.out files that JIST provides. The external.* files contain STDOUT and STDERR of only the external CLI command call. This debug information is additionally injected into JIST's Debug Viewer, but are stored as files as well.

After a JSON module has been written, it should be placed in the ext-jist-lib directory. In order to turn this JSON definition into a module, users should rebuild the plugin library and the module will be written out and placed in a directory according to the pluginPackage and pluginCategory (with the root directory being jist-lib). Any errors during this pseudo-compilation process will be written out in the terminal that MIPAV was launched from. Compilation is performed on a module-to-module basis. Thus, one broken module will not cause a domino effect on the remaining modules that need to be compiled. Common errors include syntactical issues such as generating non well-formed JSON syntax. Type is also case sensitive as it is used to reference JIST class files.

3 Integration with NITRC-CE

Cloud Dashboard Panel interface was added to configure, launch and terminate an Amazon AWS NITRC-CE cloud instance with predefined machine info (See "The Cloud Dashboard panel" in Figure 3). Users are required to gain an access key ID as well as a secret access key which gives them access to the cloud. This process cannot be done automatically as payment is required for S3 storage and non-free EC2 instance. Once the AWS account and region validation are completed, the user can create a new S3 bucket or import their existing S3 storage bucket. The bucket is used to save all relevant files to install MIPAV, JIST plugins and configuration files for installing necessary packages on each AWS instances. The user's project will also submit and store into this selected S3 bucket (See "Create a new bucket?" in Figure 3). Additionally, a standard cryptographic key pair is required to log in. If users already have a key pair generated from previous uses of the AWS cloud, these keys can be imported and used to connect to the remote instance. If users do not already have a key pair, one can be created by automatically (See "Create a new key pair?" in Figure 3)

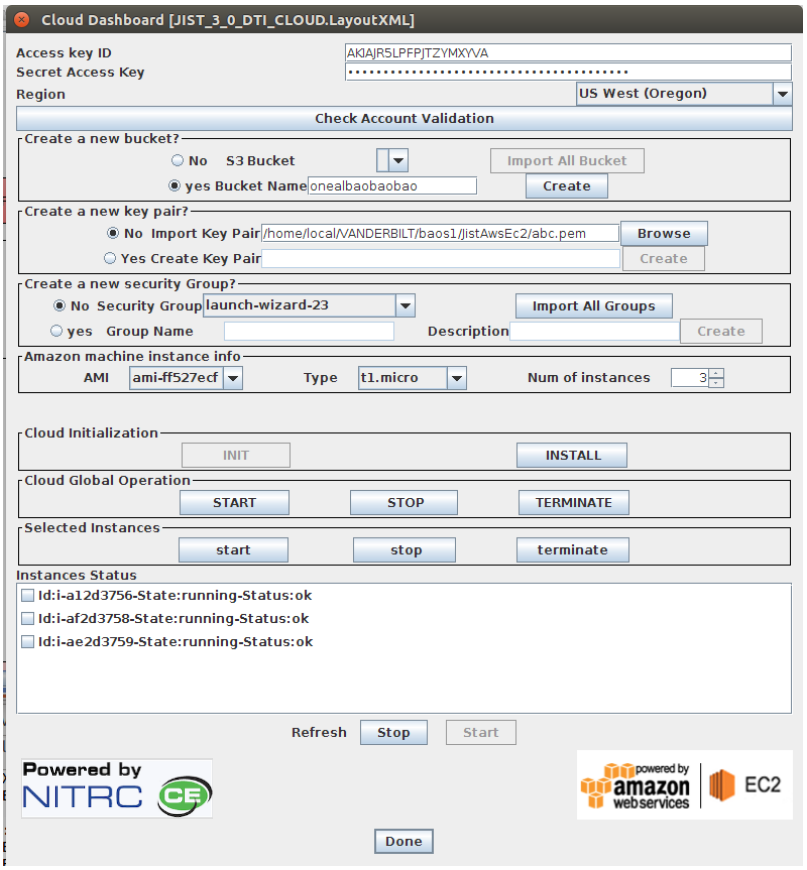


Figure 3: The Cloud Dashboard panel.

AWS also allows users to generate custom security groups on cloud instances (See Figure 3 "Create a new security Group?"). By default, there are no restrictions, though this is user-configurable. After security groups are set, users can select several different types of pre-configured amazon machines and any number of identical machines to boot (See Figure 3 "Amazon machine instance info". Note that the

defaults are set as the recommended usage case.). After all of this information is set up, it is saved such that if the instance crashes, or the window is opened again, users don't have to re-enter all of the configuration information. Any changes made to the current configuration panel will overwrite current settings.

Once all of the settings look correct, users need to please check steps as follows.

1. In users system home directory, check if there is a folder existed as "JistAwsEc2". If not, users need to create one.

2. In system-home/JistAwsEc2 folder, check if folder "tmp" exits. If not, users need to create one.

3. In system-home/JistAwsEc2 folder, check if folder "DoNotDelete" exits. If not, users need to create one.

4. Users should download MIPAV installation file from MIPAV official site, we recommend version that best matches the JIST being run on the users' local machine. The file needs to save in the folder system-home/JistAwsEc2/DoNotDelete.

5. Find MIPAV source folder directory "mipav" in users' workstation, find plugins folder and compress it as "plugins.zip". Then users move the zip file into the folder system-home/JistAwsEc2/DoNotDelete.

6. In MIPAV source folder, users could find a file "mipav.preferences". Please copy this file to folder system-home/JistAwsEc2/DoNotDelete.

7. Please find JIST library folder "jist-lib", compress it as "jist-lib.zip", and move it to folder system-home/JistAwsEc2/DoNotDelete.

8. In JIST library, please locate the JIST preference file "JistUserPreference.xml", copy this file and place it to folder system-home/JistAwsEc2/DoNotDelete.

All steps above the users would just need to do once when using the cloud AWS service.

Once preparing all files done, users can click the "INIT" button (See Figure 3 "Cloud Initialization"). The init process handles the entire login on each of the machines as well as uploading necessary files for MIPAV and JIST to S3 bucket that users just selected. If the init process fails, users should check all of their keys as well as the machine info that they have set. The init process should take approximately 5 minutes per cloud machine (though this depends on internet connection speed). When all instances state are running with their status are ok, users can click the "INSTALL" button to install all necessary packages including mount S3 storage as a local folder on each machines. There is a new window will be opened for cloud install process, users should click button "install" to check the installation progress. (See Figure 4 "Cloud install progress"). The processes of copying and installing JIST and MIPAV will complete as well once the installation done. Users have the ability to start and stop as well as terminate all or selected machine instances. If JIST crashes, users should connect to the AWS web client to manually terminate all of the machines before restarting a new instance. JIST maintains no mechanism to interface with the AWS webapp.

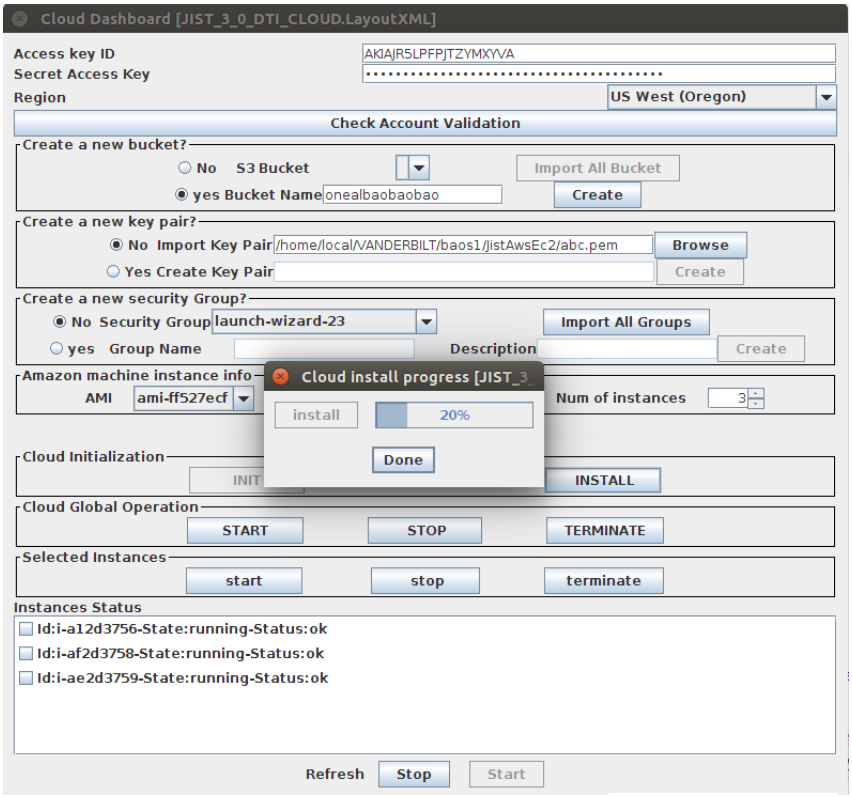


Figure 4: Cloud install process.

JIST will automatically send necessary project to cloud once they are opened the process manager. The uploading time depends on the size of project, and all project files are saved in S3 bucket with same project structure (See Figure 5 "Send project to cloud"). Once all project files are saved in cloud successfully, the users would get a notice of completion. All processing is handled in the cloud environment, though the monitoring process is handled on the host (local machine). Each execution context (i.e., a single process in the process manager) is run on a separate cloud node while maintaining JIST's graphical process flow. The process manager periodically polls all running nodes and jobs in queue for jobs that have completed, jobs that have errored out, and jobs that are ready to run. Once a process has completed, it will be marked as "Complete" in the process manager, just as if the job had been run locally. Once a status of complete has been reached, all files that have been generated during this execution have been copied back to the local machine and paths have been changed from those point to the cloud node to those on the local machine. Thus, even though processing was handled on the cloud, outputs are ready to be further processed on the local machine.

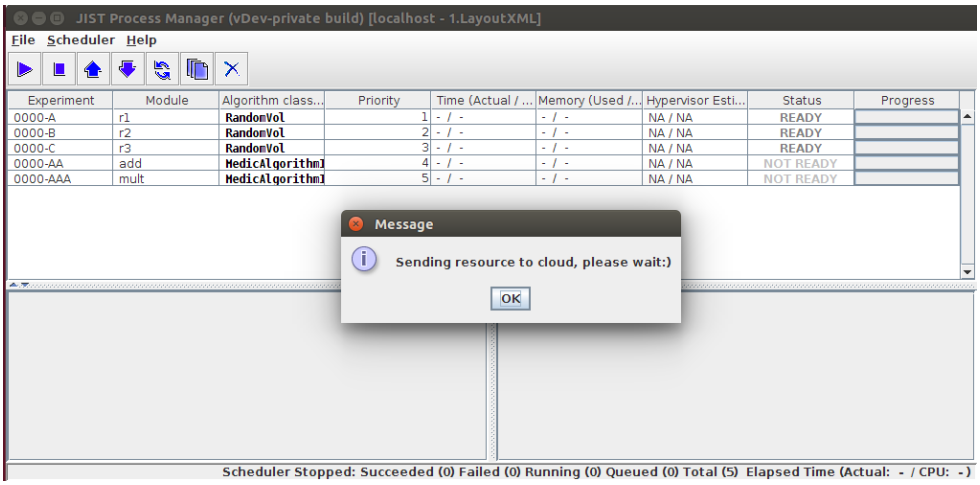


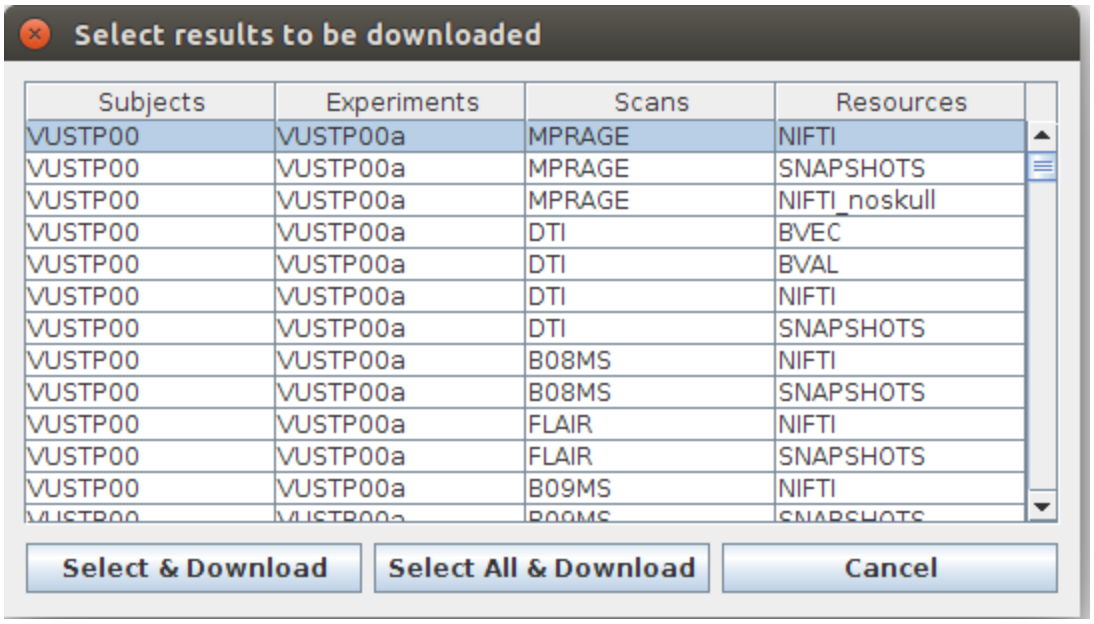
Figure 5: Sending project to the cloud.

4 Integration with XNAT

XNAT's ability to store large scale imaging studies and experiments coupled with JIST's semantic process naming and now, massively parallel cloud processing engine, couple together for the ability to process neuroimaging studies in a single unified environment. Exposing the REST API in a similar fashion to PyXNAT, JIST allows users to log into an XNAT instance, which is simply defined by a host URL, user name, and password. This connection gives the user an server defined time-limited JSESSION. While connected via a JSESSION token, users have the ability to search projects (studies in XNAT), which they have been given access to (or created), subjects (a study participant in XNAT), experiments (a single imaging modality session including but not limited to MR and CT), scans (a single acquisition as part of a series), and resources (file types e.g., DICOM, NIfTI and PAR/REC). Users can enter full or partial search strings for each of these fields or just leave them blank to get everything they have access to.

User must login once into the XNAT on starting JIST after which the JSESSION stays active for as long as XNAT is being used or when the admin-defined timeout is hit. During each query, an HTTP "keep-alive" is used to maintain connection. Users can either explicitly logout or will be logged out automatically on exiting JIST.

Once connected with appropriate credentials, users can drag the XNAT source and connect it to any of the modules that accept ParamFileCollections. Users also have the ability to use "File Collection to Volume Collection" module to convert the source to a ParamVolumeCollection, which may be useful for other modules. Searching may take some time depending on database load as well as the complexity of the query (e.g., thousands of subjects looking for just T1 scans and a DICOM resource). After the query is complete, search information is cached and can be viewed or used to re-download the files. If XNAT is updated (e.g., new sessions are added or new resources are generated), the search needs to be re-executed.



The files are downloaded using a HTTP GET request and can be seen as a list in the input view allowing the user to ensure that all the files have been downloaded. Also, to avoid unnecessary downloading a check for prior existence is performed in the output directory. After completion of the data pull, each of the experiment(s) as part of the current layout will be listed as READY in the process manager and execution can begin. It is important to note that if the pipe breaks, users need not re-run their search, instead they can re-download the files using the saved query.

Currently JIST only supports resources that are considered to be "raw data" (i.e., non-processed data). These resources all must follow the following REST schema /project/<PROJECT_ID>/subject/<SUBJECT_ID>/experiments/<EXPERIMENT_ID>/scans/<SCAN_ID>/resources/<RESOURCE_ID> where everything in capital letters is user-selectable. XNAT maintains a simply extensible REST API (https://wiki.xnat.org/display/XNAT16/XNAT+REST+API+Directory) which can easily be extended to pull results from previous processing ("assessors" in XNAT terms, previously "reconstructions"). If that is the case, the API would need to be minorly extended to /project/<PROJECT_ID>/subject/<SUBJECT_ID>/experiments/<EXPERIMENT_ID>/out/assessors/<ASSESSOR_ID>/resources/<RESOURCE_ID>.

5 Demonstration of new features

We have pre-packaged a virtual machine running Ubuntu 14.04 LTS and our release of JIST 3.0, which is available on our NITRC page (https://www.nitrc.org/projects/jist). On the Desktop is a folder called "Examples". In it is a layout called Example_XNAT_Source.layoutXML. This layout is setup such that users can connect to our XNAT instance located at (www.vandyxnat.org/xnat). Users are encouraged to make an account, which will give them public access to open data resources such as IXI (https://wiki.xnat.org/display/XNAT16/XNAT+REST+API+Directory), the Multi-Modal Reproducibility

Study (Kirby21 on XNAT) [12], and ABIDE (https://www.nitrc.org/projects/fcon_1000). A table of expected outputs is shown below:

Project	Subject	Session	Scan	Resources	Output
IXI	IXI026-Guys-0696	IXI026-Guys-0696-1	*	NIFTI	4 NIFTI volumes
IXI	*	*	T1	NIFTI	581 NIFTI volumes
Kirby21	*	*	ASL	NIFTI	42 NIFTI volumes
Kirby21	Kirby21_492	*	T2w	NIFTI	2 NIFTI volumes

Already included in the ext-jist-lib are three .json module files. SEfakedata.json can be used to run the Solar fakedata command (https://nitrc.org/projects/se_linux). FSLFlirt.json can be used to run FSL's FLIRT registration method. Finally there is an MRICron.json file that is intentionally set up to not work so users can see how it will appear in the module tree. All modules appear under the MASI package under the EXTERNAL sub category. Though all of these packages are external CLI packages, they are included in layout files and usable. The file Example_External_CLI.layoutXML is provided on the desktop to demonstrate how these new features work. The FLIRT module is set up to register an MPRAGE scan to 1mm MNI space from subject space. The fakedata command is set up to generate a 1000 family x 5 traits testing file using Solar. MRICron is expected to stay as "Not Ready" in the process manager because MRICron is not included in the path.

Finally, we have included a complex layout for processing DTI data that makes use of the XNAT Source. Three input datasets from the Kirby21 project are used with over 30 separate processes. Users can connect to the cloud using our pre-configured set of test credentials. The cloud size will be 10 machines, which will all run a separate process. Though the processing is executed on the cloud, users should expect to see all outputs on their local disk, just as if execution was performed on the local machine.

6 Conclusions

JIST 3.0 builds on our previous releases by integrating E-Science platforms. Users now have the ability to effortlessly interface with large scale cloud processing and neuroimaging archiving and storage platforms such as XNAT. External CLI packages can now be wrapped in to JIST modules to further simplify more complicated pipelines into one coherent layout. With these new functionalities less time needs to be spent organizing and downloading data for large scale processing and more time can be spent innovating new post-processing methods and pipelines.

Acknowledgements

This work is supported by NIH R01EB15611.

References

1 Li, B., Bryan, F., and Landman, B.A.: 'Next Generation of the Java Image Science Toolkit (JIST): Visualization and Validation', The insight journal, 2012, 2012, pp. 1-16

2 Lucas, B., Landman, B., Prince, J., and Pham, D.: 'MAPS: a free medical image processing pipeline', Organization for Human Brain Mapping, 2008

3 Lucas, B.C., Bogovic, J.A., Carass, A., Bazin, P.L., Prince, J.L., Pham, D.L., and Landman, B.A.: 'The Java Image Science Toolkit (JIST) for rapid prototyping and publishing of neuroimaging software', Neuroinformatics, 2010, 8, (1), pp. 5-17

4 Boisvert, R.F., Moreira, J., Philippsen, M., and Pozo, R.: 'Java and numerical computing', Computing in Science & Engineering, 2001, 3, (2), pp. 18-24

5 McAuliffe, M.J., Lalonde, F.M., McGarry, D., Gandler, W., Csaky, K., and Trus, B.L.: 'Medical image processing, analysis and visualization in clinical research', in Editor (Ed.)^(Eds.): 'Book Medical image processing, analysis and visualization in clinical research' (IEEE, 2001, edn.), pp. 381-386

6 Marcus, D., Olsen, T., Ramaratnam, M., and Buckner, R.: 'XNAT: a software framework for managing neuroimaging laboratory data', in Editor (Ed.)^(Eds.): 'Book XNAT: a software framework for managing neuroimaging laboratory data' (2006, edn.), pp.

7 Stonebraker, M., and Rowe, L.A.: 'The design of Postgres' (ACM, 1986. 1986)

8 Taylor, J., and Perez, F.: 'nipy: Neuroimaging in Python', in Editor (Ed.)^(Eds.): 'Book nipy: Neuroimaging in Python' (2011, edn.), pp.

9 Ibanez, L., Schroeder, W., Ng, L., and Cates, J.: 'The ITK software guide', 2003

10 Bray, T.: 'The JavaScript Object Notation (JSON) Data Interchange Format', 2014

11 Jenkinson, M., Beckmann, C.F., Behrens, T.E., Woolrich, M.W., and Smith, S.M.: 'Fsl', Neuroimage, 2012, 62, (2), pp. 782-790

12 Landman, B.A., Huang, A.J., Gifford, A., Vikram, D.S., Lim, I.A., Farrell, J.A., Bogovic, J.A., Hua, J., Chen, M., Jarso, S., Smith, S.A., Joel, S., Mori, S., Pekar, J.J., Barker, P.B., Prince, J.L., and van Zijl, P.C.: 'Multi-parametric neuroimaging reproducibility: a 3-T resource study', Neuroimage, 2011, 54, (4), pp. 2854-2866