JOURNAL OF APPLIED CRYSTALLOGRAPHY

ISSN 1600-5767

Received 24 April 2018
Accepted 23 August 2018

Edited by Th. Proffen, Oak Ridge National Laboratory, USA

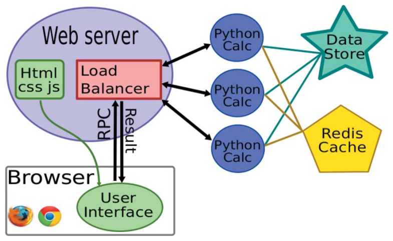
Keywords: neutron reflectometry; X-ray reflectometry; data reduction; computer programs.

reductus: a stateless Python data reduction service with a browser front end

Brian Maranville,* William Ratcliff II and Paul Kienzle

NIST Center for Neutron Research, 100 Bureau Drive, Gaithersburg, MD 20899, USA. *Correspondence e-mail: brian.maranville@nist.gov

The online data reduction service *reductus* transforms measurements in experimental science from laboratory coordinates into physically meaningful quantities with accurate estimation of uncertainties from instrumental settings and properties. This reduction process is based on a few well known transformations, but flexibility in the application of the transforms and algorithms supports flexibility in experiment design, enabling a broader range of measurements than a rigid reduction scheme for data. The user interface allows easy construction of arbitrary pipelines from well known data transforms using a visual data flow diagram. Source data are drawn from a networked, open data repository. The Python back end uses intelligent caching to store intermediate results of calculations for a highly responsive user experience. The reference implementation allows immediate reduction of measurements as they are recorded for the three neutron reflectometry instruments at the NIST Center for Neutron Research, without the need for visiting scientists to install additional software on their own computers.



1. Motivation

The transformation of raw measurements into meaningful interpretable data with attached uncertainties (data reduction) is a ubiquitous task in the experimental sciences. In the case where the workflow is well established and the community is small the most direct way to accomplish this is to develop a custom application to be installed on a limited number of dedicated computers. However, at a scientific user facility a large number of visiting researchers are using the measurement tools on a part-time basis. Thus, it is necessary to make reduction capabilities widely available and flexible. In this case a web-based application is an attractive alternative to distributing dedicated installable executables.

The main benefit of a web application is the almost universal accessibility to potential users. On top of this, a centralized reduction server also benefits from the ability to update the calculation code at any time without requiring all users to update their software, as well as largely eliminating the non-trivial time cost of maintaining an installable application for more than one target platform.

Users can also install the service locally, much as they would if reduction were a desktop application. This allows them to access their own data and develop reduction applications for their own instrumentation.

The specific implementation described herein, *reductus*, was developed to provide reduction for reflectometry instruments, but the system was designed to be extensible and usable for other data processing problems and work is underway to support reduction for off-specular reflectometry, small-angle neutron scattering and triple-axis spectrometry. The approach can be adapted to any type of computational problem, but the level of flexibility and computational complexity supported is driven by the needs of a data reduction engine.

1.1. Data reduction for reflectometry

Neutron reflectometry is a technique for characterizing surfaces, thin films and multilayers by analyzing the intensity of a reflected signal relative to the incident beam. This fraction of number reflected/number incident is called the reflectivity, and it can be modeled and interpreted by using, for example, an optical matrix formalism analogous to the one used in the analysis of electromagnetic reflectivity (Born & Wolf, 1959; Abelès, 1950), but where the relevant potentials arise from interactions between the neutron and nucleus, or the neutron and the internal magnetic field of the sample, rather than electromagnetism. The theory provides a prediction for the reflectivity as a function of momentum transfer (difference of momentum for incident and reflected photon or neutron, in this case).

As an illustrative example, we will describe just one of the many types of reflectometry instrument setups: one with a monochromatic beam and with no polarization analysis.

In order to measure this reflectivity, an apparatus is constructed such that a collimated beam (of known energy, and known angular and energetic resolution) is made incident on a planar sample, and a neutron detector is placed after the sample to record the number of neutrons that are specularly reflected (where the angle of incidence equals the angle of reflectance) in a given time. This measurement is repeated for a number of incident and reflected angles, which correspond to a range of momentum transfers, so that the results can be compared with the prediction from theory as described above. In order to calculate the reflectivity, though, we need to divide the number of reflected neutrons at each momentum transfer by the number of incident neutrons, and so we also have to measure the 'direct-beam' intensity separately, with no sample in the beam, with the instrument collimation set up exactly as it is for the reflection measurement. In our case, the collimation is controlled by the opening of two slits in the beam path before the sample position, so setting the direct beam to the same collimation just requires opening those two slits to the same position they were in during the reflected-beam measurement. We also wish to subtract spurious detection events, such as incoherent scattering from the sample holder, so we perform another set of experiments where the sample is in the beam but the detector is not placed at the specular reflection position (the angle of reflection is intentionally set to be different from the angle of incidence). These measurements we call the 'background'. Often two background measurements are taken, where the angle is offset from the specular condition in either the positive or negative direction (later labeled bg+ and bg− in the text and figures.)

Then in the simplest case, in order to extract the physically meaningful reflectivity (number of neutrons reflected/number of neutrons incident for a given momentum transfer) we first subtract the 'background' signal at the corresponding incident angle from the measured reflected intensity, and then divide the subtracted value by the 'direct-beam' measurement with the same collimation. The incident angles and wavelengths of the measurements are converted to more meaningful reciprocal space Q_z. The generation of error bars for the counts is based on the fact that these are drawn from a Poisson distribution, and instrumental angular and energy resolutions are calculated on the basis of the collimation and optics (monochromator) of the measurements.

Within the application, the process is treated as a series of data transformations, with data flowing from one transformation module into the next. Corrections are chained together in a data flow diagram (Sutherland, 1966), with the final data set saved as the result of the reduction. This yields the

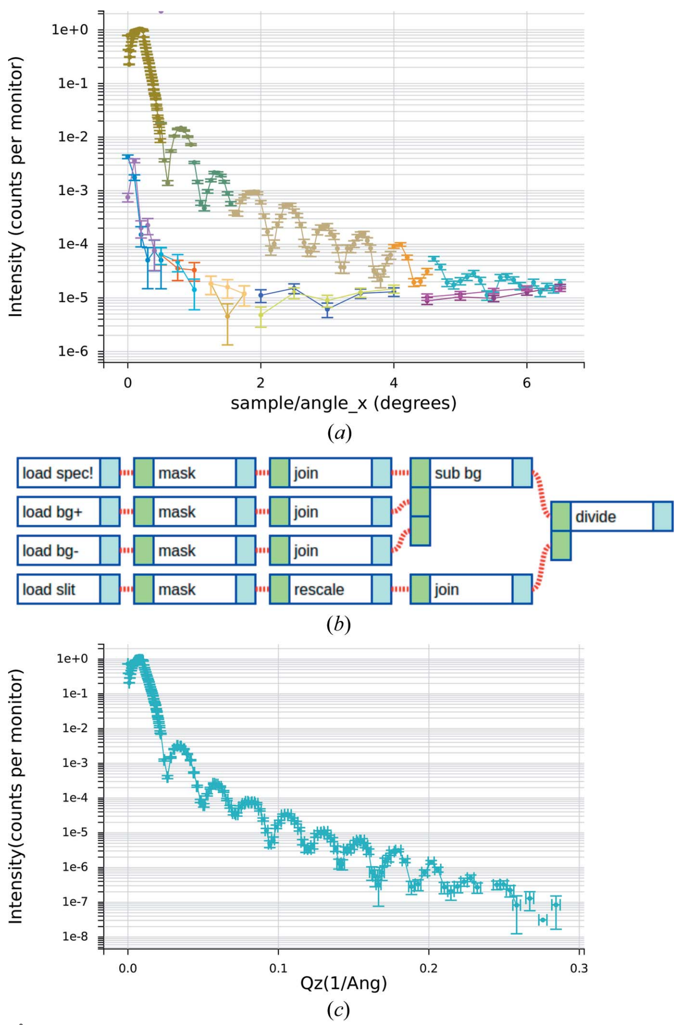


Figure 1
(*a*) Measured signal and background counts. Each color represents an independent data file from our data acquisition system, plotted against the instrument motor that controls the incident beam angle, which in this case is 'sample/angle_x'. The upper curves are the signal and the lower curves are the background. (*b*) Data flow diagram, with the left boxes for each node representing the input data and the right boxes representing output. The experimentalist can select the signal, ±background and slit normalization for the `load` nodes, producing the reduced data as the output from the `divide` node. (*c*) Reduced data in physical coordinates.

estimated reflectivity *versus* the momentum transfer, Q_z (see Fig. 1).

Users must first identify the data files that will be read as the inputs of the 'load' modules at the left of the diagram; then further individual modules may have control parameters, such as a scale factor for a scaling module (needed if there is a beam attenuator in one measurement but not in the others, for example). The measured reflectivity is the output of the 'divide' module at the right, in physical coordinates with instrument-specific details removed.

Other, more complicated setups are in common use for measuring reflectivity, including polychromatic sources (multiple incident energies), polarized-beam components *etc.*, and correction modules are included for many of these cases so that the user can construct a reduction pipeline that matches their experimental setup.

2. Web interface

The user interface for *reductus* is a JavaScript application that runs in a browser. The application relies on a number of advanced JavaScript libraries, and as such it is supported only by browsers that have reasonably standards-compliant implementations of JavaScript (`ECMAScript` version $\geq$5). The application is made up of these key components (see Fig. 2):

(*a*) data source file browser

(*b*) plotting panel to show results

(*c*) panel for setting parameters for individual computation modules

(*d*) visual, interactive data flow diagram

The components and user interactivity thereof are described in the following subsections.

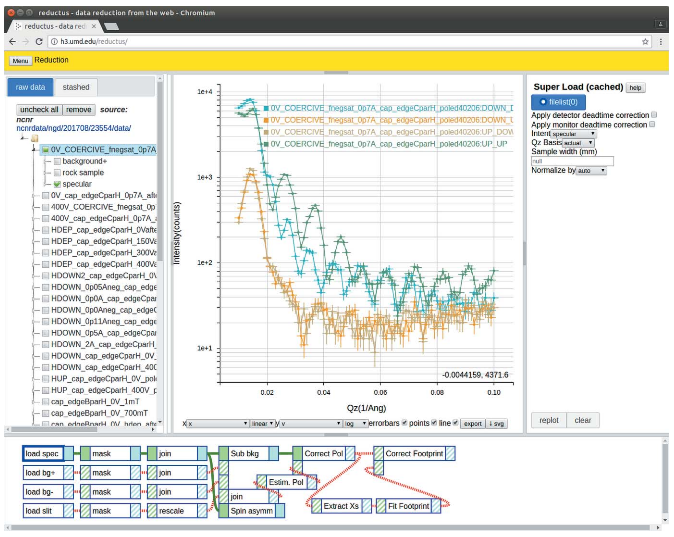


Figure 2
reductus user interface panels: (top left) data source file browser, (top center) plot of the data corresponding to the current module, (top right) current module parameters and (bottom) interactive data flow diagram.

Once connected to the server, the web client requests a listing of the public data stores, the available reduction steps and a set of predefined template diagrams representing the usual reduction procedures for the data. The menus and default options are populated on the basis of this information.

2.1. Data flow diagram

The user interacts with the data flow diagram in order to navigate the reduction chain: by clicking on a module within the diagram to bring up the parameters panel for that module, or by clicking on the input (left) or output (right) terminals to display calculation results. Changes made in the parameters panel are by default immediately applied to the active template, though by un-checking the `auto-accept parameters` box in the `data` menu an additional confirmation step (pressing the 'accept' button in the panel) can be imposed to avoid accidental changing of the active template. In that context, if a user makes a parameter change but then selects a different module without first pressing 'accept', that change is lost.

The 'clear' button removes all values from the active parameters; the client then repopulates the panel with default values for each parameter.

The web client creates a JavaScript object notation (JSON) representation of the data flow diagram along with an indicator of the input or output of interest, and sends the request to the server *via* HTTP POST. The response is a plottable representation of the data for that connector encoded either as JSON or MSGPACK (Furuhashi, 2013), which is then displayed in the plotting panel.

Clicking on any output within the data flow diagram, including the rightmost final output ('Correct Footprint' in Fig. 2), will trigger a calculation of all ancestor results for that result in the diagram on the server, as described in §4.1.1. So, while the user has the option to inspect intermediate calculation steps, for a routine reduction they can simply click on the final output to export the completely reduced data.

2.2. Parameters panel

At the beginning of a reduction, a user chooses an instrument definition for working with the data. As described in §4.1.1, this includes a list of data types and a list of reduction steps (modules) to act on those data types. The rendering of the parameters panel is based on the definition for the chosen module type, using the predefined types for each input field to the module function, mapping the simple known types (`int`, `float`, `bool` *etc.*) to HTML form elements. Some field types have renderers with enhanced interaction with the plot panel, such as an `index` type, which allows clicking on data points to add them to an index list in the parameters panel. Another example of enhanced interactions is the `scale` type, which enables dragging a whole data set on the plot to set the scaling factor in the parameters panel.

When parameters are changed in this panel and committed with 'accept', they will be used in any calculation of data flowing through that module.

2.3. Browser caching of calculations

In addition to the caching provided on the server for avoiding recalculation of identical results, a local browser cache of calculations is provided. This is particularly useful for the initial source data load, in which metadata from all of the files in a source directory are passed to the client for inspection and sorting in the source file browser. Naturally in a data reduction scheme, the quantity of data on the input side (loaders) is much greater than the output result, so caching of the inputs helps tremendously when making small adjustments interactively to the data flow algorithm or parameters.

2.4. Sessions and persistence

The *reductus* server is stateless; the reduction diagrams created by the user are not stored (a unique hash of the template representation may be associated with cached calculations on the server, but no user template is ever stored there). The only state associated with a session is stored in the browser or on the filesystem of the user's computer.

2.4.1. Stashing in-browser. Results of calculations can be 'stashed' in the local persistent memory of the browser. A list of these results can be recalled in the client and reused in two ways: by reloading the entire calculation into the active data flow panel, or by selecting multiple stashed results to directly compare their outputs.

2.4.2. Saving to and loading from filesystem. In addition to the local browser store, the user may download a text version of the data flow diagram in JSON format with `Menu` → `Template` → `Download`, which can be reloaded with `Menu` → `Template` → `Upload`. The file contains the diagram along with any field values, including the names of the input files. The actual data are not included.

The data for the currently selected node can be saved with `Menu` → `Data` → `Export`. This prompts the user for a filename, then produces a tab-delimited column-format text file with the data flow diagram prepended as a comment header. The stored diagram allows a full reduction to be reloaded into the client with `Menu` → `Data` → `Reload` → `Exported` (note that this may trigger recalculation if the raw data have been updated since the reduction was exported).

As a result of security limitations built into all current browsers, the data may only be saved to the user's 'Downloads' folder, while uploads can of course originate from any user-readable folder.

2.4.3. Sharing data among collaborators. The data flow diagram is self-contained. The reduced-data text files produced by the *reductus* system can be shared with others easily by e-mail or portable media, and provide both useful data and a recipe in the header for recreating the data from known sources with known transformations; the chain of logic can be inspected and verified by reloading the data flow into the web client at any time, thus assuring data provenance. This allows for easy collaboration amongst users without the need for accounts or passwords.

3. Data flow diagram as a template for computation

3.1. Data types

The data set flowing between modules has a type associated with it. In order to connect the output of one module to the input of the next module, the type of the output parameter on the first module must match the type of the input parameter on the subsequent module. For each data type, there are methods to store and load the data, to convert the data to display form, and to export the data. The stored format should be complete, so that reloading saved data returns an equivalent data set. The displayed and exported forms might be a subset of the total data.

3.2. Operations

By implementing the transformation modules in Python, the instrument scientist has access to a large library of numerical processing facilities based on *NumPy* and *SciPy* (Oliphant, 2007) and, in particular, libraries which support propagation of uncertainties (Lebigot, 2013). The *reductus* library includes additional facilities for unit conversion, data rebinning, interpolation and weighted least-squares solving as well as its own simplified uncertainties package for handling large data sets.

3.3. Bundles of inputs

It is often the case that many measurements need to be combined, with the same computation steps applied to each data file. Rather than defining a separate chain for each file, *reductus* instead sends bundles of files between nodes. To interpret the bundles, the module parameters are defined as either single or multiple. If the primary input is single, then the module action operates on each of the inputs separately; if multiple, then all inputs are passed to the module action as a single list. For example, if several measurements must be scaled independently then joined together into a single data set, the scale module input would be tagged as single, but the join module input would be tagged multiple. The scale factor would be tagged multiple, indicating a separate scale factor for each input. Outputs can also be single or multiple. Unlike the join module, which produces a single output from multiple inputs, a split module would produce multiple outputs from a single input. A module which rebalances data amongst inputs (*e.g.* to correct for leakage between polarization states in a polarized beam experiment) takes multiple inputs and produces multiple outputs.

3.4. Instrument and module definition

An instrument is a set of data types and the computation modules for working with them. A computation module has a number of properties, including name, description, version, module action and parameters. Each parameter has an ID, a label, a type, a description, and some flags indicating whether the parameter is optional or required, and if it is single or multiple.

Input and output parameters use one of the data types defined for the instrument. Control parameters can have a variety of types, including simple integers, floats or strings, or more complicated types such as indices into the data set or coordinates on the graph, allowing some parameter values to be set with mouse pointer interaction in the user interface.

3.5. Module interface definition

The module interface definition, including the input channels, the controlling parameters and the output channels, is encoded in the model documentation. As a result, the documentation should remain consistent with the user interface to the module. The stylized documentation starts with an overview of the module action. For each input, control and output parameter it gives the data type and units and provides a short description of the parameter which can be displayed as a tool tip in the user interface. Inputs and control parameters are distinguished by examining the action declaration; the positional parameters are inputs that can be wired to another node's output and the keyword parameters are control parameters. After the parameters, the module documentation should define the author and version. The module name is set to the name of the action function.

The module documentation is valid reStructuredText, which means that the standard *docutils* toolset for Python (https://pypi.org/project/docutils/) can be used to convert the documentation string to hypertext markup (HTML) or portable document format (PDF). The conversion to HTML is performed with *Sphinx* (http://www.sphinx-doc.org/), allowing for the creation of an independent user manual for each instrument; it is also done dynamically for each module for display in the user interface. Embedded equations are rendered in HTML using *mathjax* (https://www.mathjax.org/), a TeX equation interpreter for JavaScript.

3.6. Serialization of the diagram

A data flow diagram is represented as a list of nodes numbered from 0 to n, with each node having a computation module, a label, an (x, y) position and values for the control parameters of the computation module. The connections are defined as a list of links, with each link having a source (node number and output parameter name) and a target (node number and input parameter name).

Every diagram can be used as a template, with the configuration values for the nodes packaged separately from the diagram. The computation engine looks first in the configuration for control parameter values for the node, using the value given in the diagram if a specific configuration value is not found. If no value is provided in the configuration or in the diagram then the default parameter value for the module is used.

4. Back end

The back end is composed of two main pieces: a traditional web (HTTP) server providing static resources including HTML, JavaScript and cascading style sheet (CSS) source code for the client application in the user's browser, and a computation engine that handles requests for reduction calculations as remote procedure calls (RPC), as diagrammed in Fig. 3. A pool of shared calculation engines is shown but is only recommended for a production server, as a single engine is sufficient for a single-user test environment. A shared disk-backed computation cache is not required but is strongly recommended for a responsive server, even in a single-user environment (a per-instance non-persistent in-memory cache is the fallback option.)

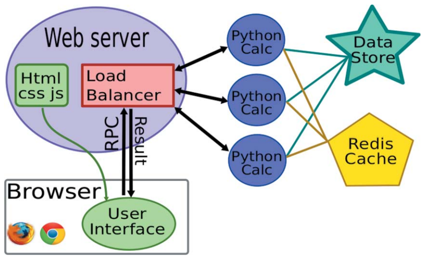


Figure 3
reductus system diagram. Upon receiving a request from the user interface, the load balancer on the web server will find an available Python thread to run the reduction diagram. The first step will be to fetch the requested data files from the data source and save them in the *Redis* cache. Intermediate calculations may also be cached allowing future repeated requests to be returned immediately to the client, trading efficiency against the size of the cache. As demand increases the different parts can be run on different servers to spread the load.

The `Data Store` in Fig. 3 is not part of the server but is an HTTP-accessible source of raw data[1] which is loaded as the first step of a reduction. This arrangement makes it possible to perform data reduction without handling the raw data files on the client computer – the user can download just the reduced data if they wish.

4.1. Computation server

The point of contact for the client is the web server, which serves the static resources (HTML, JavaScript, CSS) as well as being a proxy gateway to the calculation engines through the Python web services gateway interface (WSGI).

4.1.1. Converting the diagram to computations. A data flow diagram is a directed acyclic graph (DAG), with the modules as the nodes in the graph and the connections from outputs to inputs as the links between the nodes. No cycles are allowed, which means that the output of a module cannot be used to compute its own input. Every DAG has topological order, so the nodes can be arranged linearly such that the independent nodes appear first, and each dependent node appears after all of its input nodes. By computing nodes in topological order all inputs are guaranteed to be computed before they are needed. Although there are linear time algorithms for sorting a DAG, the diagram sizes for data reduction are small enough that a naïve $O(n^2)$ algorithm can be used.

[1] For example, the NIST Center for Neutron Research (NCNR) data store is located at https://dx.doi.org/10.18434/T4201B.

4.1.2. Results and source caching. The results of every calculation request are cached on the server. There are several choices in the configuration of the server but the default is to use a *Redis* (Sanfilippo & Noordhuis, 2012) key-value store with a least recently used expiry algorithm and a maximum cache size. This can be started automatically as a system service at startup on a Linux server, allowing worry-free operation between server reboots.

The server cache is very important to the performance of the service: the slowest part of many computations is retrieving the source data files from the data repository to the server. With caching, this typically will only happen once per data file per experiment, and after that the server-local cached version will be used. For data files, it is assumed that they can be uniquely identified by their resource path (URL) and last-modified time (accurate to a second because of the filesystem `mtime` limitations). It is therefore fallible if the file is rewritten in a time span of less than a second, but the data files we are using are updated much more slowly than that.

Each calculation step is identified by a unique id, created by hashing all of the input values along with the version number of the code for the step itself. For inputs which are the result of a previous calculation step, the hash of that step is used to identify the input. Since the calculations on the same data should give the same results each time, the results can be cached using this key and retrieved for future calculation steps or returned to the web client for display. If the calculation parameters change (for example, the scale factor for a scaling step), then the hash will change, and the step will be recalculated and cached with the new key. This will change the input values for the subsequent step, which will change its hash, which will cause it to be recalculated as well. In this way, if there are changes to the source data (timestamp), all reduction steps which depend upon the data will be updated. By including the version number of each step in the cache, changes to the server will automatically cause all future reductions to be recomputed, even if they are already cached.

4.1.3. Data provenance and reproducibility. In a highly interactive environment, where parameters, files and even workflows can be modified and the results saved at any time, it is important to have a record of the inputs as well as the results (Simmhan *et al.*, 2005). Therefore, the reduction template and all its parameters are stored along with the reduced data in each saved file. By loading a file into *reductus* the precise steps required to reproduce the data become visible. The NCNR data source is referenced through a digital object identifier (DOI), with the implicit promise of a stable reference even if the data are moved to a new URL, thus providing long-term reproducibility of the reduction.

The server uses the current version of each reduction step to evaluate the outputs. This effectively acts as a behind-the-scenes update to all steps in the reduction process; any steps that are newer will be recomputed, and the updated results can be re-exported. This is particularly useful for reductions performed during data acquisition. As newer measurements are added the updated timestamp will force recomputation of all subsequent steps.

In order to reproduce an existing reduction, the server version at the time of the reduction must be used. The server source is managed with the *Git* source control system (Torvalds & Hamano, 2010), available on GitHub at https://github.com/reductus/reductus. *Git* creates a unique hash of the entire source tree for each commit which is stored as part of the template. To reproduce the data from a specific reduction this hash must be used to retrieve the source code and run it as a local server. The specific versions of the dependencies (*SciPy*, *NumPy*, *uncertainties*) can be recorded in the source tree as well to protect against changing interfaces. Because users can easily revert to older versions of the software, developers are free to modify the code at will and maintain a clean code base.

4.1.4. Statelessness. The computation engine maintains no state. The user interface manages the interactions of the end user with the engine and keeps a working copy of the active data flow template(s); the browser session is the only real context. This has distinct operational advantages for the compute engine – it can be restarted at any time with close to zero impact on the availability and continuity of the service. The cache is persistent between engine restarts but can be completely wiped if necessary without destroying user data (performance will suffer temporarily as the calculations are re-cached).

4.2. Server configurations

The system is designed to be modular, allowing a number of possible configurations of the needed resources.

4.2.1. Simple single-computer configuration. The simplest configuration is to run the web server, calculation nodes and cache on the same computer. A server implementation using the Python `flask` package is provided, which can simultaneously serve the static client resources and the RPC calculation requests. This server is suggested for use when running a private reduction service.

A private server is required for processing data sets stored locally; since the service is stateless, providing neither data upload nor storage of reduction diagrams and reduced data, there is no other way to reduce data that are not present in the public data repositories.

Local file access is only enabled for servers running on the private 'localhost' network interface. Such servers should not allow external connections since access to local files is a security risk.

Similarly, a private server is required for custom reduction steps that are not part of the standard service since the standard service does not allow the execution of arbitrary Python code. Users at facilities that do not allow external access to the web will need to copy all the data files to a local machine and reduce the data with a private server.

4.2.2. Container-based configuration. The service can be run using a Linux container environment. This allows for reproducible environments and eases development so that new developers do not have to learn how to install of the required packages for a modern web application. A recipe for

Docker (https://www.docker.com/) (the particular container technology that we used) is provided in the source code for running the application as three coordinated containers: one for the web server, one for the Python calculation engine and one for the *Redis* cache. The current snapshot of the source code in the user's directory is copied into the containers as part of the build step, so this is a useful setup for development and testing. The user need only install *Docker* and *Docker Compose*. The supporting tools, including Python, *Redis* and all dependent libraries, are pulled in by the *Docker Compose* recipe. This greatly eases the ability of users to extend the project and to test new features.

4.2.3. Scalable production server configuration. For production (public facing) servers, the static files can be copied from the `/static` subfolder of the source repository to a web server such as *Apache* or *nginx*, where requests to the `/RPC2` path are forwarded to a pool of Python calculation engines [*e.g.* using *uWSGI* (https://uwsgi-docs.readthedocs.io/) to run the calculations with Apache `mod_uwsgi_proxy` acting as the load balancer], sharing a *Redis* instance for caching.

An elastic on-demand service could be built from these components as well, with multiple (replicated) *Redis* caches and no limit on the number of worker Python calculation engines that can be employed by a high-availabilty web server setup. The statelessness of the server means that no complicated synchronization is required between server components.

5. Conclusions

The *reductus* system is an interesting experiment in providing stateful web services with a stateless server. Although users lose the convenience of cloud services for managing their data, they are free from the inconvenience of maintaining yet another user ID and password. Files can be stored and shared using familiar tools such as e-mail. The server is easy to adapt and install locally for the rare user that needs more than the rigid set of functions and data sources provided in the remote web service; this is no more complex than adapting and installing the equivalent desktop application would be. For the developer, the stateless server needs very little maintenance. There are no database migrations needed and no backups required, and moving the service to a different computer is as simple as installing the software and redirecting the domain name service (DNS) to an alternative IP. JavaScript provides a flexible environment for interactive applications, with a rich and growing ecosystem of libraries that work across most browser platforms. The Python back end provides an ecosystem for rapid development of numerical code. The web services middleware gives us scalability with no additional effort.

Making the data flow graph visible and modifiable increases flexibility without increasing complexity. Users with simple reduction problems can enter their data on the left of the graph (see Fig. 2) and retrieve their results on the right, ignoring all steps in between. If there are problems, they can examine inputs and outputs at each step and identify the cause. Although 85% of reduction is performed on-site during the experiment, over 150 external users were able to access the system from across the United States and Europe in 2017. Without the need to install a local version of the software we have far fewer support requests; now they only occur for unusual data sets.

Feedback from the users has been overwhelmingly positive, and the new system has completely supplanted our old reduction software for the three neutron reflectometry instruments at the NCNR, with work underway to adapt the system to several new instrument classes at the facility.

Acknowledgements

We are grateful to Professor Robert Briber of the Department of Materials Science and Engineering at the University of Maryland for providing significant technical support and operating the servers for early deployments of the *reductus* service. The data flow display and editor, plotting routines, and parameter-setting panels were implemented using the *D3.js* visualization library (Bostock *et al.*, 2011). The identification of commercial products or services in this paper does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the equipment or service used is necessarily the best available for the purpose.

Funding information

Significant contributions to the early code were made by Brendan Rowan, Alex Yee, Ophir Lifshitz and Elakian Kanakaraj as part of the NIST Summer High School Internship Program (SHIP) and Joseph Redmon as part of the Summer Undergraduate Research Fellowship (SURF), supported by NSF CHRNS grant DMR-1508249.

References

Abelès, F. (1950). *J. Phys. Radium*, **11**, 307–309.
Born, M. & Wolf, E. (1959). *Principles of Optics: Electromagnetic Theory of Propagation, Interference and Diffaction of Light*. Oxford: Pergamon Press.
Bostock, M., Ogievetsky, V. & Heer, J. (2011). *IEEE Trans. Visualization Comput. Graph.* **17**, 2301–2309.
Furuhashi, S. (2013). *MessagePack Specification*, https://github.com/msgpack/msgpack/blob/master/spec.md.
Lebigot, E. O. (2013). *Uncertainties*, http://pythonhosted.org/uncertainties.
Oliphant, T. E. (2007). *Comput. Sci. Eng.* **9**, 10–20.
Sanfilippo, S. & Noordhuis, P. (2012). *Redis: The Definitive Guide: Data Modeling, Caching, and Messaging*. Sebastopol: O'Reilly and Associates.
Simmhan, Y. L., Plale, B. & Gannon, D. (2005). *SIGMOD Rec.* **34**, 31–36.
Sutherland, W. R. (1966). PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.
Torvalds, L. & Hamano, J. (2010). *Git*, http://git-scm.com.