# Running user-defined functions in R on Earth observation data in cloud back-ends

Pramit Ghosh, Florian Lahn, Sören Gebbert, Matthias Mohr and Edzer Pebesma

Institute for Geoinformatics, University of Münster
Heisenbergstraße 2, 48149 Münster, Germany
pramitghosh@uni-muenster.de

## Abstract

Although Earth Observation (EO) data plays a central role in various applications of geospatial sciences, the conventional workflow requiring the presence of the data on the local machine has become a bottleneck for processing large volumes of EO data. Processing it in cloud back-ends instead by transferring the code to the data overcomes bandwidth limitations significantly. Furthermore, it allows for the development of a client-server architecture where client nodes using different programming languages submit processing requests to cloud back-ends, servers with access to EO data. As part of the openEO application programming interface (API), which aims to mediate such workflows, infrastructure to run users' scripts containing custom functions on EO data in cloud back-ends is under development. This paper focuses on an API for running such user-defined functions (UDFs) written in R, and describes three strategies that have been developed with the help of the R package "stars". The first is file-based where the UDF service could be thought of as a part of the backend. The other two are implemented as RESTful web services where the data is transferred between the back-end and the UDF service either in the form of JSON arrays containing pixel values, or base64 encoded strings by embedding it in a JSON file. Containerization using Docker for the reproducibility and testing of the R package which implements these services as well as future directions for its development, particularly to enhance scalability, are also discussed.

Keywords: EO data processing, big geodata, UDFs for EO data, openEO API, r-spatial

## 1. Introduction

Wide range of applications of EO data requires the development of workflows for its analysis and full utilization in geospatial sciences. Suitable EO data, such as satellite images, are identified for an objective based on various factors such as its resolution, sensor type, availability on the date(s) of interest, cloud coverage etc. Subsequently, the data is downloaded to the local machine for further processing. However, with the ever-increasing rate EO data is being generated, this workflow is a serious bottleneck for large data volumes due to limitations in bandwidth and computing power of typical workstations. Therefore, alternative workflows overcoming these limitations are on the rise and cloud-based processing is a viable one (Nativi et al., 2015). Emergence of cloud-based EO data processing platforms such as Google Earth Engine (Gorelick et al., 2017) and Amazon Web Services only proves the necessity of these emerging paradigms.

### 1.1. Cross-platform EO data processing in the cloud

Infrastructure allowing the processing of EO data in the cloud, apart from overcoming the limitations discussed above, also brings the opportunity to make such infrastructure cross-platform. This would allow client nodes using different programming languages to send processing requests to back-ends that could have their own internal processing chains. The potential incompatibility that arises in such a client-server architecture could be solved by a common interface which is comprehensible to both the clients and the back-ends. The openEO project proposes to develop an open application programming interface (API) for client nodes for processing EO data on various cloud-based services provided by its partners that use different technologies internally. The client nodes and the back-ends that are compliant with the openEO API will allow seamless processing of EO data across platforms using native processes available on the back-ends.

### 1.2. User-defined functions (UDFs)

While such infrastructure for cross-platform processing of EO data is a big-step forward, in order for it to be useful by the geospatial community, support for running users' custom scripts is necessary as no particular back-end could provide all the required processes used by the entire EO community natively. The openEO API, therefore, is supplemented by an additional API for running user-defined functions (UDFs) that act as an interface between the back-ends and the UDF services as shown in Figure 1. This paper discusses the development of this UDF API and the strategies used to implement UDFs written in R.

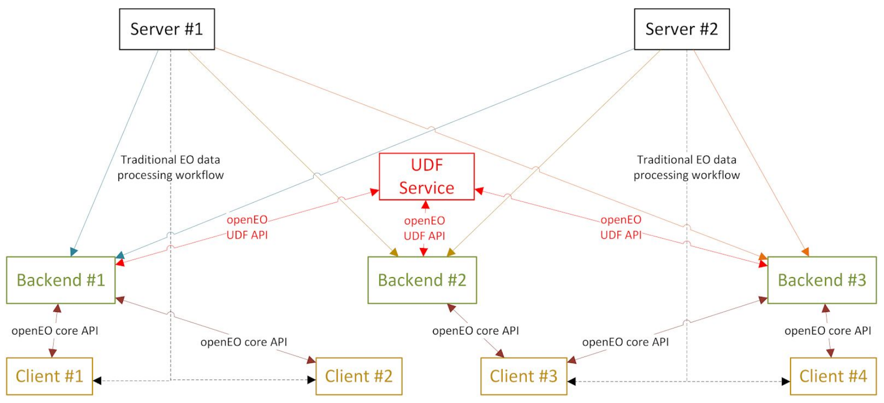

Figure 1. Traditional (dashed lines) vs. openEO-based (solid lines) EO data processing

## 2. Methods

A key aspect of developing infrastructure for UDFs is that the UDF service is not accessible directly to the client nodes, but instead made available through the back-ends. Thus, the UDF API is concerned with the interface between a back-end and the UDF service and not with the client. The internal architecture of the UDF service is, therefore, abstracted and is presented only as a black-box to the clients. However, a challenge here is to create an abstraction while still providing enough flexibility and freedom to the end-users (operating the client nodes) so that the API is of practical use in geospatial research. The client communicates with the back-end using HTTP requests with the body listing the processes to be run ordered by the sequence of their execution in a "process graph" as defined by the core openEO API. Subsequently, the back-end starts executing this "process graph" and on encountering users'

UDFs, it transfers the data and the custom function to the UDF service through various strategies, which converts the received data to a “stars” object (Pebesma, 2018), applies the function on it and transfers the results back to the back-end in the same way it received it.. These strategies have been implemented for UDFs written in R.

## 2.1. UDF SERVICE AS A PART OF THE BACK-END

This is a file-based strategy which assumes that the UDF service is an integral part of the back-end. In this implementation, the intermediate results are written to the disk in the form of GeoTIFF files using a specified directory structure along with a look-up table in the form of a plaintext comma separated values (CSV) file with additional data such as timestamps and band information.After writing this data to disk, the back-end executes the user’s R script that computes the output after converting the data to a “stars” object. Subsequently, the results are written to disk and control returns to the back-end which ingests these files.

## 2.2. RESTFUL UDF WEB SERVICE USING JSON ARRAYS

Instead of assuming the UDF service to be a part of the back-end, here the UDF service is an independent web service with a RESTful interface. The back-end, upon encountering an UDF, instead of writing data to the disk, sends a HTTP POST request to the UDF service with the body containing the user’s script, the data (in the form of JSON arrays representing pixel values of the satellite image) and other relevant data including timestamps, band information, coordinate reference systems (as a PROJ.4 string) etc. The JSON follows a specified schema and the UDF service, upon receiving it, computes the results in memory and sends back a response after converting it to JSON arrays.

Two varieties of this strategy have been implemented. The first one, running at an endpoint /udf, exposes only a particular dimension (e.g. band, time etc.) of the “stars” object created from the JSON arrays to the user’s function. The user’s function, which is to be written in the form of an anonymous R function, only gets to access a set of numbers representing the pixel values along that dimension. This allows for users to run simple aggregation functions such as mean, max, median etc. on that dimension. An R UDF could, then, be defined in the POST body as “function(obj) median(obj)” and then parsed, evaluated and applied on the incoming set of values. If this anonymous function is run on the time dimension, it would calculate the median of the time-series (of the satellite images) of each pixel for all the bands.

The second variety, running at an endpoint /udf/raw provides more freedom to the users as it exposes the actual “stars” object to the user’s function. This allows users to write more complicated code but this freedom comes at the expense of the users’ responsibility to return a “stars” object from their function. For example, the user can write R functions such as “function(obj) st_warp(obj, crs = st_crs(4326))” to change the coordinate reference system of the “stars” object (obj).

## 2.3. RESTFUL UDF WEB SERVICE USING BASE64 ENCODED STRINGS

This strategy is also a web-based service with the data embedded in a JSON and transferred using HTTP POST requests. This is run at the endpoint /udf/binary. The difference between this and the one discussed in the previous section is that the EO data is represented as a base64 encoded string instead of JSON arrays. The base64 encoded string is decoded by the UDF service to a compressed ZIP file and the results are computed subsequently after converting its contents to a “stars” object. The results are converted back to a base64 encoded string which is transferred back to the back-end embedding it in a JSON. This implementation also exposes the entire "stars" object to the user's function.

## 3. Results

These strategies have been implemented as a R package "openEO.R.UDF" (Ghosh and Lahn, 2018) with the source code available publicly as a Github repository. The HTTP requests are handled using the R package "plumber", which exposes a port on the web server. The resulting architecture of the strategies discussed above is shown in Figure 2.

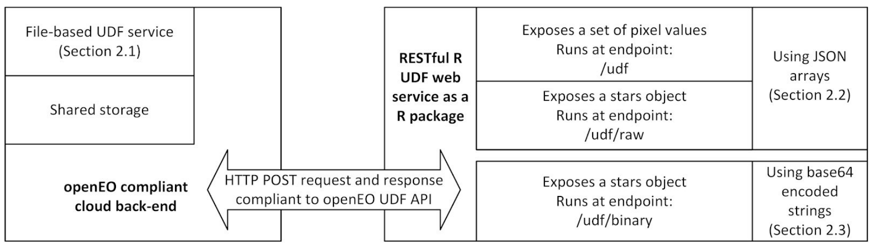
File-based UDF service (Section 2.1)
Shared storage
openEO compliant cloud back-end
HTTP POST request and response compliant to openEO UDF API
RESTful R UDF web service as a R package
Exposes a set of pixel values Runs at endpoint: /udf
Exposes a stars object Runs at endpoint: /udf/raw
Using JSON arrays (Section 2.2)
Exposes a stars object Runs at endpoint: /udf/binary
Using base64 encoded strings (Section 2.3)

Figure 2. Architecture of strategies implemented for R UDF services

In order to promote reproducibility, easier maintenance and testing of the web service, the R package is containerized using Docker (Merkel, 2014) and made available on Docker Hub[1]. Containerization helps to take care of security concerns that are possible for web services such as this that are accessible to the public to run their custom code on the server. Furthermore, Docker containers make sure the whole server does not crash in the event of a system failure - something very plausible while processing large volumes of EO data.

## 4. Conclusion

Infrastructure for running users' functions on EO data in the cloud, as described, is much needed by the geospatial community to tackle such big data challenges. The UDF service facilitates processing requests from practically any back-end compliant with the API. The API's simplicity makes it easy to integrate with commonly-used back-ends. However, the strategies discussed here could be improved for scalability allowing the service to render results efficiently for larger data volumes. Slicing the data into smaller chunks based on the UDF's semantics, sending it to UDF services parallely and stichting the results could be a way to improve scalability. This is an avenue openEO aims to explore in the near future.

## Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 776242. This publication only reflects the author's views; the European Commission is not responsible for any use that may be made of the information it contains.

[1] https://hub.docker.com/r/pramitghosh/openeo.r.udf/

## References

Ghosh, P., and Lahn, F., 2018, openEO.R.UDF: User-defined functions (UDF) in R on Earth observation data in cloud back-ends. R package version 0.0.3. https://github.com/pramitghosh/openEO.R.UDF

Gorelick, N., Hancher, M., Dixon, M., Ilyushchenko, S., Thau, D., & Moore, R., 2017, Google Earth Engine: Planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment*, 202, pp. 18-27.

Merkel, D., 2014, Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014 (239).

Nativi, S., Mazzetti, P., Santoro, M., Papeschi, F., Craglia, M., & Ochiai, O., 2015, Big data challenges in building the global earth observation system of systems. *Environmental Modelling & Software*, 68, pp. 1-26.

Pebesma, E., 2018, stars: Scalable, Spatiotemporal Tidy Arrays. R package version 0.2-0. https://CRAN.R-project.org/package=stars