Improving type safety in Rucio

CERN Summer Student Programme 2022

Author:
Aksel Lunde Aase <aksel.lunde.aase@{cern.ch,gmail.com}>

Under the supervision of:
Martin Barisits <martin.barisits@cern.ch>
Mario Lassnig <mario.lassnig@cern.ch>

August 12, 2022

Abstract

This report covers my contribution to the Rucio project during my eight weeks at CERN from June 20th to August 12th 2022. Rucio is an open source distributed data management system initially built to serve the ATLAS experiment's needs, but is now also used in other scientific experiments like CMS and DUNE. Rucio is written in Python, and in June 2022 ended support for Python 2.7. This allows the introduction of Python type annotations throughout the codebase, which was part of my work. My focus has primarily been to improve the developer experience for developers working on and using Rucio.

1 Introduction

Rucio [1] is a software tool to manage the distribution of experimental and simulation data to multiple datacenters around the world in an efficient and resilient manner. For LHC Run 2 alone, ATLAS recorded nearly 18 petabytes of data which has to be stored securely and at the same time be readily available for physics analysis all over the world [2]. As of 2021, Rucio managed over 500 PB of data in total for the ATLAS experiment [3], and these amounts of data require dedicated and specialized solutions to serve the users' needs.

The Rucio project is a Distributed Data Management (DDM) system that handles distribution and replication of data between data centers, ensuring that data is available where it is needed and safely stored in case of data loss at specific sites. It is developed as an open-source project and hosted on GitHub [4], with a core development team situated at CERN.

Rucio is built on a client-server architecture, and allows integration into experiment-specific tooling through its client-side Python API. There is therefore a separation between developers of Rucio which contribute to both client- and server-side code, and users of Rucio, which utilize the Python API to build their own tooling. To simplify both development and usage of Rucio, there has recently been a flux of improvements to the Rucio Documentation website [5]. Good documentation can greatly simplify a new developer or user's first experience with a project. With regards to documentation, there is always room for improvement, and this summer I have tried to improve the experience of working with the Rucio codebase for both developers and users.

2 Contribution

Given the recent removal of Python 2.7 support, my project was to utilize this opportunity to begin implementing type annotations throughout the codebase. Type annotations serve as documentation for both developers and users, and will also help prevent and catch errors by introducing additional automatic tooling to review code.

2.1 Static type checking

Rucio is written in Python, which is a dynamically typed, interpreted language. This implies that less checks are performed against the code at development time than in a statically typed and compiled language, instead postponing eventual errors until runtime. The consequences of this is that certain kinds programming errors are more likely to affect the end user, unless special care is taken to prevent this.

To mitigate this problem and to help developers catch errors earlier, different tools exist to perform static code analysis before the software is released to the end user. Examples of such tools are 'MyPy' and 'Pyright', both of which perform static type checking to uncover potential type confusion in the software.

Part of my work has been to integrate 'Pyright' into the Rucio development process. Introducing a new tool to an existing codebase poses some difficulties which are not encountered when using the tool from the beginning. Static code analyzers typically output a report containing a list of errors, and for a legacy codebase these errors have had time to accumulate before the introduction of the tool.

In the case of Pyright, running the tool for the first time produced a list of 3382 potential errors, a list which is far to large to be manageable for a human. As a developer, filtering through this list to find diagnostic messages which are relevant for the code one is currently working on is not feasible, thus rendering the tool largely useless. To combat this, a separate tool called 'run-pyright' was developed that compares two 'Pyright'-reports from before and after some specific code changes, and then only informs the developer about new errors present in the second report but not in the first report. These new errors have to be related to the recently changed code, thus eliminating the noise from unrelated code.

The output of 'run-pyright' is designed to be as helpful as possible in helping the developer spot their mistake. To do this, errors are grouped together by which file they are introduced in, and the line numbers where the errors appear are clearly visible. The full error message from 'Pyright' is also available to help the developer understand their error and realize a potential solution. Listing 1 is an example of the output after comparing two 'Pyright' reports from different points in the code history (after the introduction of two new

typing errors). Instead of thousands of lines of error messages, the tool only shows the newly introduced errors.

```
Found 2 new problems in /home/runner/work/rucio/rucio/lib/rucio/api/authentication.py
  - 2 errors with message """Argument of type "InternalAccount" cannot be assigned to
                             parameter "account" of type "str" in function "get_auth_oidc"
                               "InternalAccount" is incompatible with "str"""".
    Candidates: line 98, 100, 102
Summary:
    2 new errors.
    0 new warnings.
```

Listing 1: Demonstration of output from 'run-pyright' after comparing reports from two different points in history. Instead of thousands of lines of error messages, the tool only shows the newly introduced errors.

2.2 Automated static type checking

Following the creation of the 'run-pyright'-tool, the next step was to integrate it into the GitHub Continuous Integration (CI) pipeline. CI is a feature of GitHub which allows running arbitrary checks every time code is uploaded, thus one can automate the generation of a code analysis report for every change to the Rucio codebase. The automatically generated report is then available during code review, and the GitHub user interface clearly notifies the reviewer if there are any new typing errors introduced in the pull request.

Two main pull requests cover this work, which are:

#5750 Add the Pyright type checker.
Adds the 'run-pyright' tool and implements a GitHub CI check that generates a comparison report whenever a new pull request to Rucio is submitted. This depends on the 'rucio/containers#197' PR to install necessary requirements in the Rucio development container.

#5765 Pyright type checking: Generate and compare reports for specific commits.
Extends the 'run-pyright' tool by adding the 'compare-with-commit' subcommand that greatly simplifies the local workflow for finding new errors in recent commits. This allows running the tool locally instead of pushing code to GitHub and waiting on the CI pipeline to generate the report.

2.3 Reducing the amount of errors

The goal of the automatic checking for typing errors is to prevent the number of errors to increase. Still, a lot of errors are already present in the codebase, which requires manual effort to fix. As there are currently over 3000 potential errors, the process of reaching close to zero is a long term project which requires some planning and evaluation. These already existing errors may cause problems for users of Rucio, therefore improving static typing in the public interface like the Python and REST API could have a large impact on users' experience of using Rucio.

To test the grounds of how these improvements could be made, I defined types and implemented type annotations for the 'ReplicaClient' class. The Rucio codebase extensively uses dictionaries to pass structured data to functions, which are impossible to statically analyze for correctness due to their dynamic nature. Ideally, one would define specialized data structures to hold information about the various entities Rucio contains and substitute these for all the dictionaries, but this is impossible without breaking backwards compatibility of the public facing Python API. Fortunately, using 'TypedDict's, one can explicitly specify which keys and values should be present in a dictionary, while still allowing backwards compatibility with existing code due to 'TypedDict's behaving as normal dictionaries at runtime. This enables 'Pyright' to verify that the dictionaries contain the expected keys and values of correct type, without having to rewrite existing code using the Rucio clients.

The main benefits of introducing 'TypedDict's are seen from a developers point of view. As demonstrated in Listing 2, tools like 'Pyright' can help spot errors like spelling mistakes, misremembering dictionary keys, and so on. The developers' text editor can also use the typing information to suggest autocompletion results, which further reduces the risk of spelling errors, and also reduces demand on the developer to remember key names.

```python
from typing import TypedDict

class DidDict(TypedDict):
    scope: str
    name: str

# Accepted by type checker
data: DidDict = {'scope': 'test', 'name': 'file1'}

# Rejected by type checker due to missing key 'scope',
# wrong value type for key 'name', and unexpected key 'scop'.
data: DidDict = {'scop': 'test', 'name': 4}

# The text editor can autocomplete which keys exists in the dictionary,
# and infer the correct type 'str' for the variable 'name'.
name = data['name']
```

Listing 2: Example illustrating how 'TypedDict's enable the development tooling to assist the developer.

The main pull request to cover this work is **#5770 Clients: Type annotations for 'ReplicaClient'.**

2.4 Other work

During the start of this project, I was introduced to the code base by working on issues marked as "good first issue". These were issues which could be resolved mostly by working on isolated parts of the code base without requiring a deep understanding of the full project. Later issues gradually grew in scope, which gave exposure to the rest of the project. This enabled me to better understand the requirements which determines how type annotations could be implemented. Following is a short summary of other pull requests submitted during this summer.

#5657 Database: Add poolclass configuration option for database engine.
Added a configuration option to the 'rucio.cfg' configuration file which allows specifying which kind of database connection pooling should be used.

#5669 Core: Fix RSE 'supported_checksums' attribute parsing.
Added type annotations to prevent an error where a list of strings would be passed to a function which expected a single string, and tests to verify the correct behavior of the function.

#5706 Protocols: Migrate WebDAV XML parsing to Python's ElementTree.
The Rucio WebDAV protocol did not support XML namespaces when parsing the response from WebDAV servers, which caused it to malfunction when connection to some WebDAV servers utilizing XML namespaces. Solved by utilizing Python's 'xml.etree.ElementTree' class, which properly handles namespaces.

#5718 Ensure consistent units in Prometheus timer monitoring.
During operation, Rucio logs metrics to Prometheus about system performance. In some cases, there was confusion whether the units of time were in milliseconds or seconds, which caused large errors in the reported metrics. Fix by unifying all timing reporting through a class which takes care of measuring and reporting in correct units.

#5740 Core & Internals: Log errors during email notifications.
In one case where Rucio attempts to send an email notification to the user, the email would fail to send without the server logging the error. Fix by introducing logging statements in case of errors.

#5785 Fix usages of 'datetime.now()' instead of 'datetime.utcnow()'.
The Rucio server internally expects all 'datetime' objects to be naïve and in the UTC timezone. To get the current date in this format, 'datetime.utcnow()' can be used. In some cases, 'datetime.now()' was used instead, which returns the server local time. Fixed by reviewing and replacing usages of 'datetime.now()'.

References

[1] E. O. for Nuclear Research (CERN), *Rucio - Scientific Data Management*. URL: https://rucio.cern.ch/.

[2] A. Experiment, *Bringing new life to ATLAS data*. URL: https://atlas.cern/updates/news/reprocessing-Run2-data.

[3] M. Barisits et al., *Rucio: Scientific Data Management*, Computing and Software for Big Science **3** (2019), p. 11. URL: https://doi.org/10.1007/s41781-019-0026-3, doi:10.1007/s41781-019-0026-3.

[4] E. O. for Nuclear Research (CERN), *rucio/rucio: Rucio - Scientific Data Management*. URL: https://github.com/rucio/rucio.

[5] E. O. for Nuclear Research (CERN), *Welcome to Rucio's documentation*. URL: https://rucio.cern.ch/documentation/.