Utrecht University

06/09/2020

Yoda and the iRODS Python rule engine plugin

Chris Smeele
c.j.smeele@uu.nl

Lazlo Westerhof
l.r.westerhof@uu.nl

Yoda: 'FAIR' Research Data Management

Research

Collaborate safely as a group

Vault

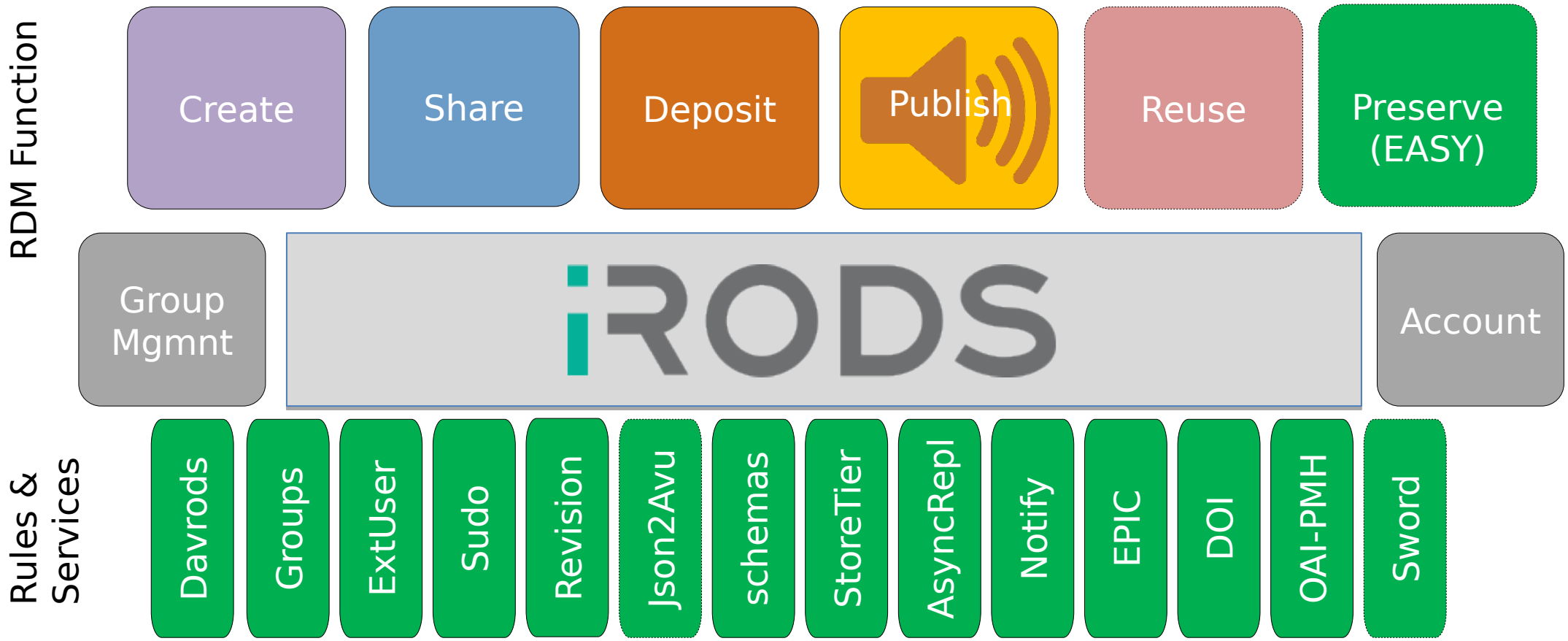
Maintain integrity, deposit a folder in the vault

Allow FAIR reuse, publish a data package

Yoda milestones

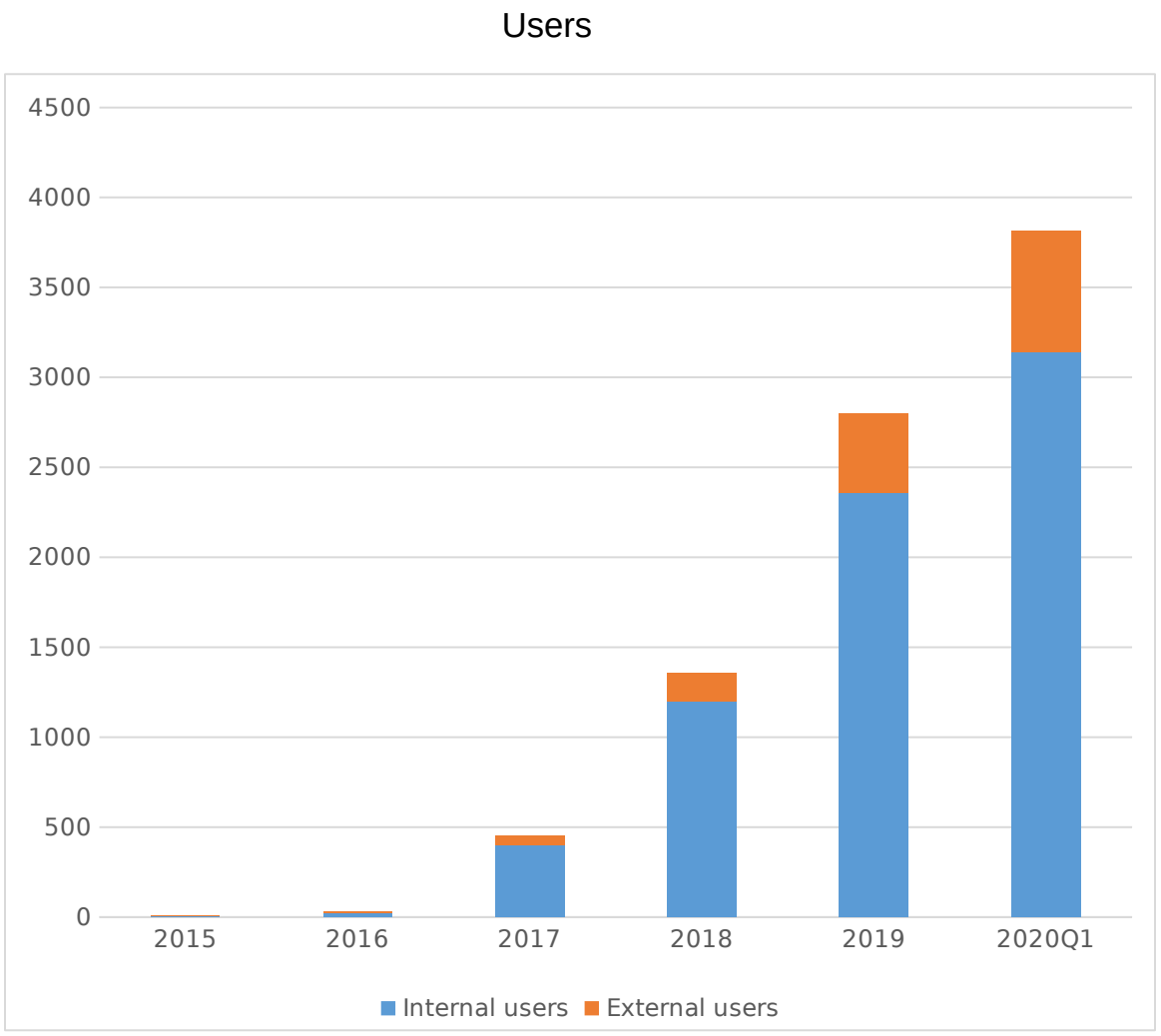
2015	• Yoda Portal and Intake module for Youth project, iRODS 3.3 based • Groupmanager module	Yoda v0.4 – v0.9
2016	• Yoda Disk (DavRods module) • Yoda Portal supports dynamic Plug-in modules, iRODS 4.0	Yoda v0.9.7
2017	• Research Workspace, revisions, metadata form • Vault archive, deposit workflow, statistics, data publication workflow (DOI)	Yoda v1.0 – v1.3
2018	• Vault metadata operations, EPIC PID, External user provisioning • (de/re)publication workflows, OAI-PMH harvestable, iRODS 4.1	Yoda v1.4
2019	• Metadata-schema management, dynamic metadata forms rendering • Metadata form based on JSON schema, file up/download in Portal, iRODS 4.2.6	Yoda v1.5
2020	• Metadata format changed from XML to JSON, with JSON-AVU • Python rules engine, iRODS 4.2.8	Yoda v1.6

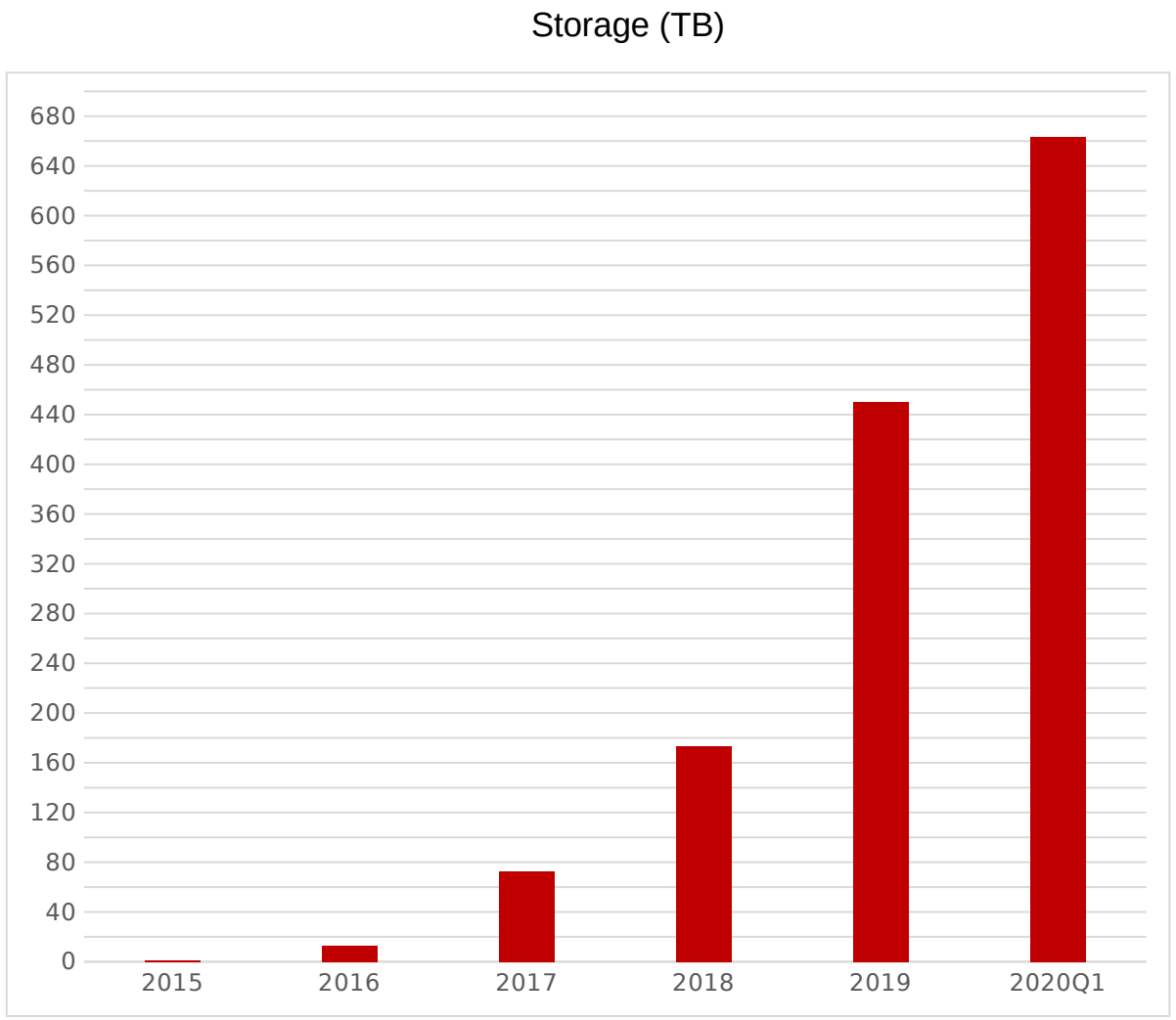
iRODS implementation for RDM



https://github.com/UtrechtUniversity/yoda

Yoda in numbers

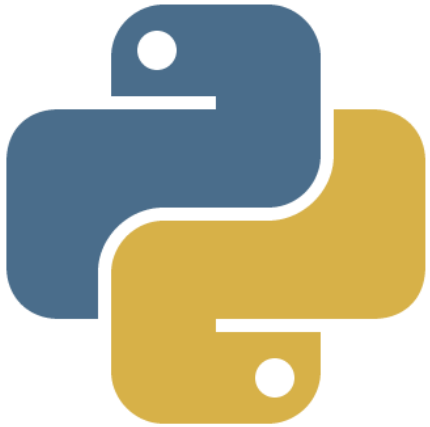




Utrecht University

Why switch to Python?

- Maintainability
- Performance
- Readability
- Learning curve
- Libraries and frameworks
- Available tooling
- Web development



python

Tidying up our rules with Python

- iRODS rule language lines of code decreased with a third
- PHP lines of code cut in half

- More maintainable code
- Improved readability
- Easier and faster development
- Better performance
- Fewer lines of code

Modular approach

- Where do we start?
  - core.py (vs. core.re)
- Our RE ruleset was a set of concatenated rule files
- How would a Python programmer approach this?

core.py

```
from rules_uu import *
# ...
```

Modular approach

- Where do we start?
  - core.py (vs. core.re)
- Our RE ruleset was a set of concatenated rule files
- How would a Python programmer approach this?
  - Modules and packages
  - core.py imports other packages as rulesets
  - Allows namespacing and private helper functions

```
/etc/irods
├   (...)
├── core.py           imports * from rules_uu
└── rules_uu/         imports * from modules
    ├── __init__.py
    ├── datacite.py   exports via '__all__'
    ├   (...)
```

Converting a rule to Python

- ‘rule_args’ calling convention
  - Boilerplate
  - Non-pythonic
  - Difficult to interface from Python functions
- Can we make this easier?

```
# iRODS rule language.
concat(*x, *y, *foo) {
  *foo = *x ++ *y;
}
```

```
# Equivalent Python rule.
def concat(rule_args, callback, rei):
    x, y = rule_args[0:2]
    rule_args[2] = x + y
```

```
# Can we not simplify it like this?
def concat(callback, x, y):
    return x + y
```

Converting a rule to Python

- ‘rule_args’ calling convention
  - Boilerplate
  - Non-pythonic
  - Difficult to interface from Python functions
- Can we make this easier?
  - Delegate argument and return value handling to a decorator
  - Support any mix of in, out, in/out args
  - Support writing returned value to stdout

```python
# Old Python rule.
def concat(rule_args, callback, rei):
    x, y = rule_args[0:2]
    rule_args[2] = x + y

# … simplified:
from rules_uu.util import rule

@rule.make(inputs=[0,1], outputs=[2])
def concat(callback, x, y):
    return x + y
```

Converting a rule to Python

- ‘rule_args’ calling convention
  - Boilerplate
  - Non-pythonic
  - Difficult to interface from Python functions
- Can we make this easier?
  - Delegate argument and return value handling to a decorator
  - Support any mix of in, out, in/out args
  - Support writing returned value to stdout

```
# Arguments are in/out by default:
@rule.make()
def uppercase2(callback, x, y):
    return x.upper(), y.upper()
```

Interfacing with rules

- How can we call rules from our web portal?
  - With structured inputs and outputs?
- RE approach: JSON microservices
  - Creating structured data is cumbersome
  - Manually handle JSON in/out for each rule
- In Python, can we do better?

```
foo(*x, *result) {
    *x.y = "abc";
    *x.z = "123";
    uuKvpList2JSON(*x, *result);
}
```

Interfacing with rules

- How can we call rules from our web portal?
  - With structured inputs and outputs?
- RE approach: JSON microservices
  - Creating structured data is cumbersome
  - Manually handle JSON in/out for each rule
- In Python, can we do better?
  - @api decorator
    - JSON input → Python args
    - Python return value → JSON output
  - Checks required/optional function args
  - Supports dicts, lists...
  - Standardizes error handling
  - Abstract away!

```python
from rules_uu.util import api

@api.make()
def api_uu_concat(callback, foo, bar):
    return foo + bar
```

```javascript
// Callable from frontend JavaScript:
let str = await
      Yoda.call('uu_concat',
                {'foo': 'test', 'bar': '123'});
```

Additional challenges

- Genquery support limited
  - Improved and merged!
- Microservice error handling (no `errorcode`)
  - Wrapped microservices, custom exceptions
- Python2 EOL
  - Work around until it’s upgraded

Future work

- Packaging (`pip install irods_ruleset_uu`)
- Python3 support?
  - Removes unicode cruft
  - Type hints & type checking
  - Modern libraries

Side note

- Davrods, our WebDAV → iRODS bridge
  - 1.5 released for iRODS 4.2.8
  - Ticket support
  - Apache conditional configuration support

- All mentioned code available at
  - https://github.com/UtrechtUniversity/irods-ruleset-uu/
  - https://github.com/UtrechtUniversity/davrods/

```
$ iexit
```

Utrecht University

DISCLAIMER

The information in this presentation has been compiled with the utmost care,
but no rights can be derived from its contents.