Pluggable Rule Engine Architecture

Hao Xu[1]

Jason Coposky[2] Ben Keller[2] Terrell Russell[2]

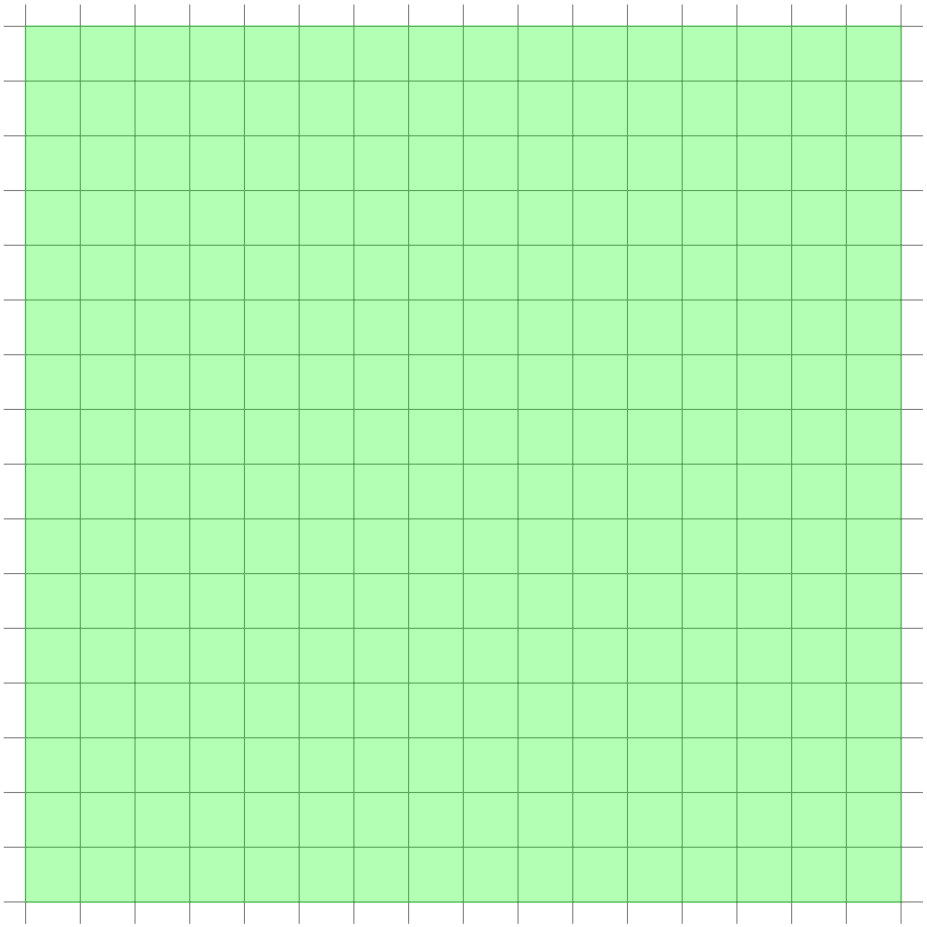
[1]DICE Center

[2]RENCI

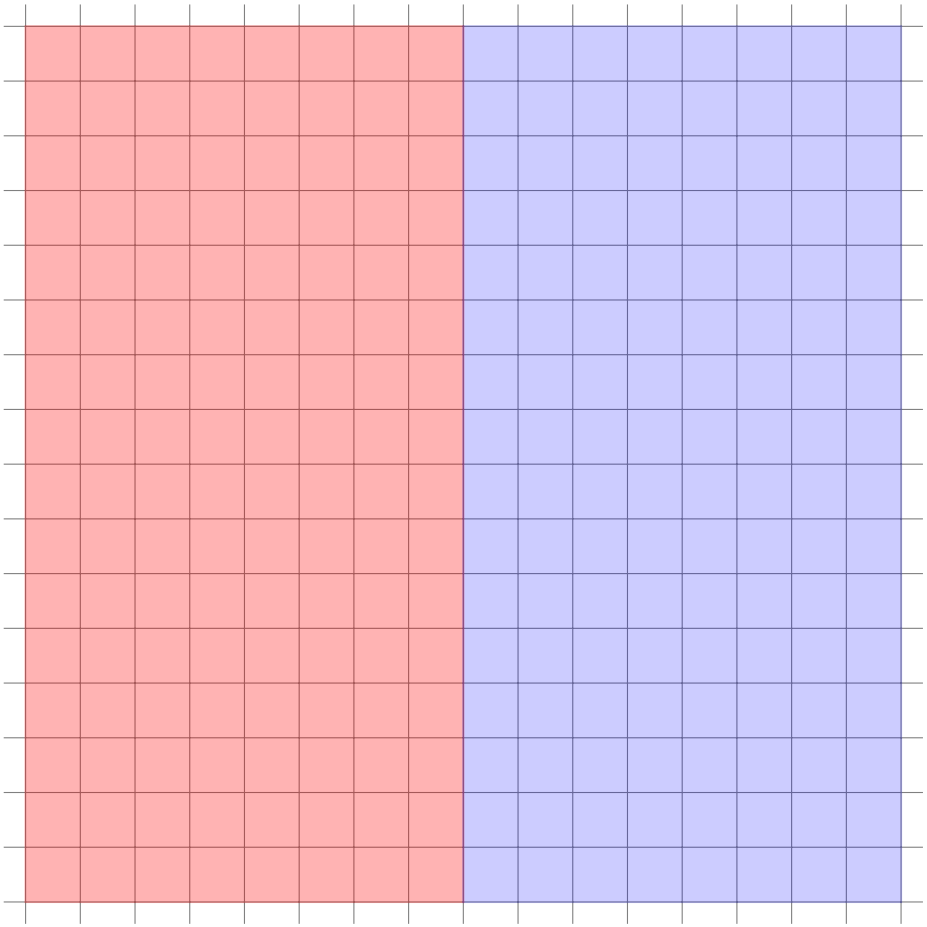
iRODS User Group Meeting, 2015

Table of Contents

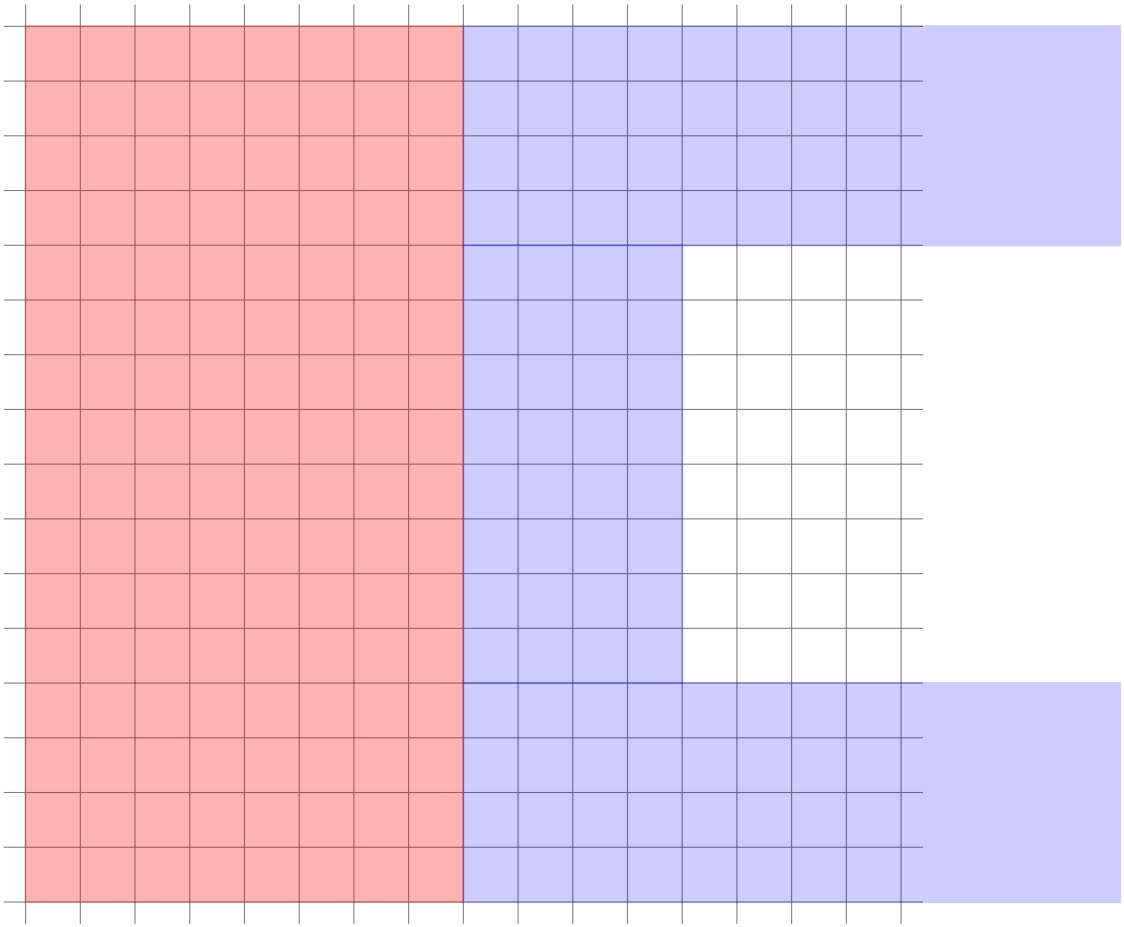
One piece of software



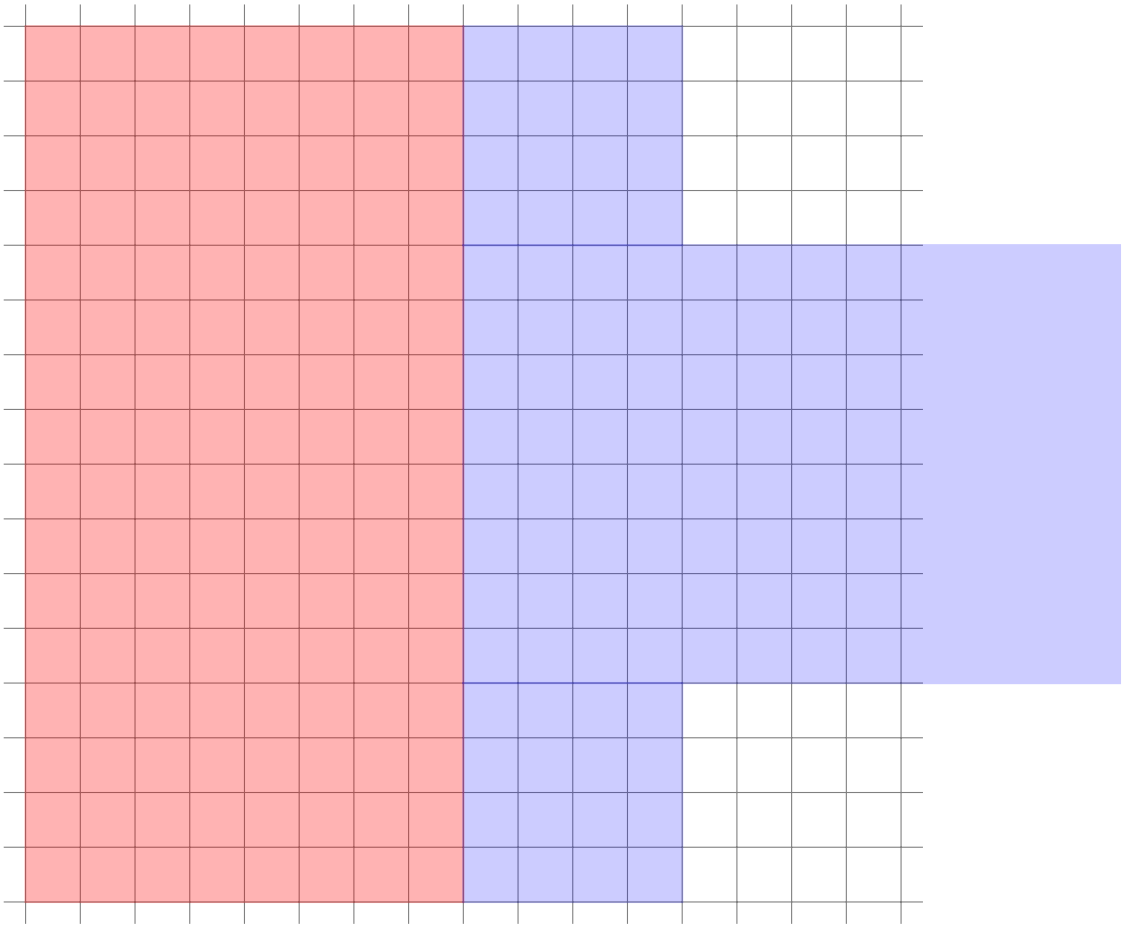
Divide it into plugins



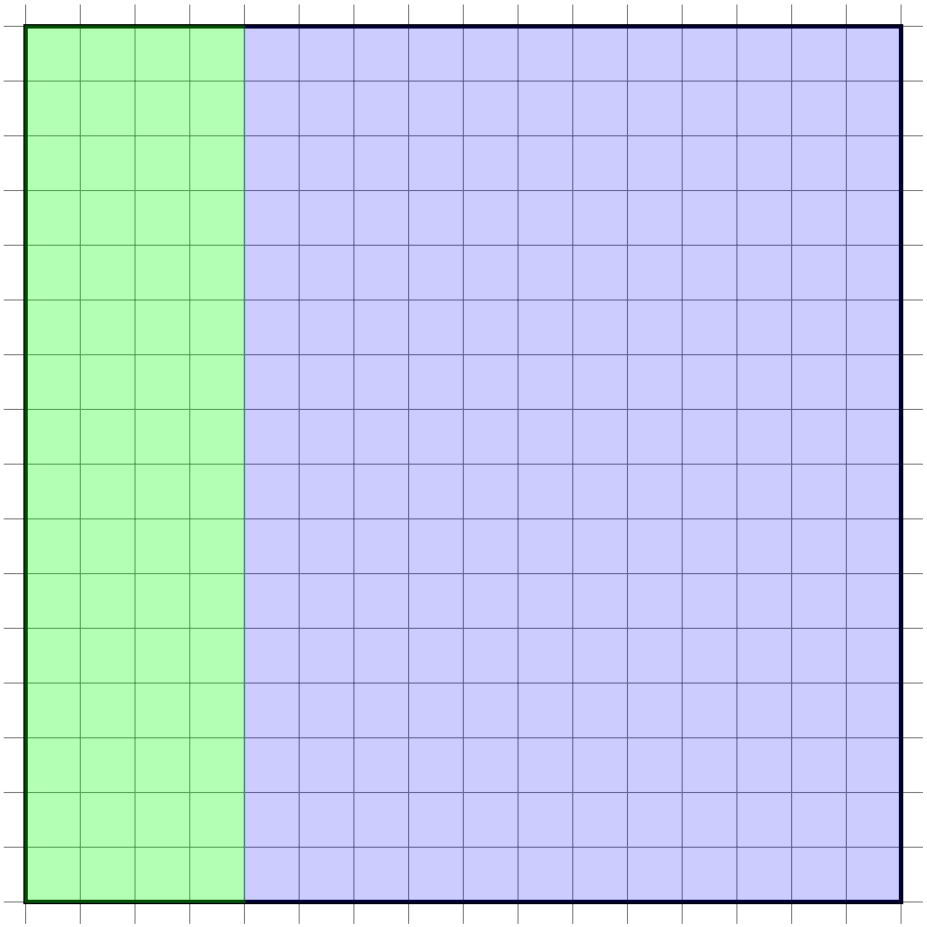
Simple



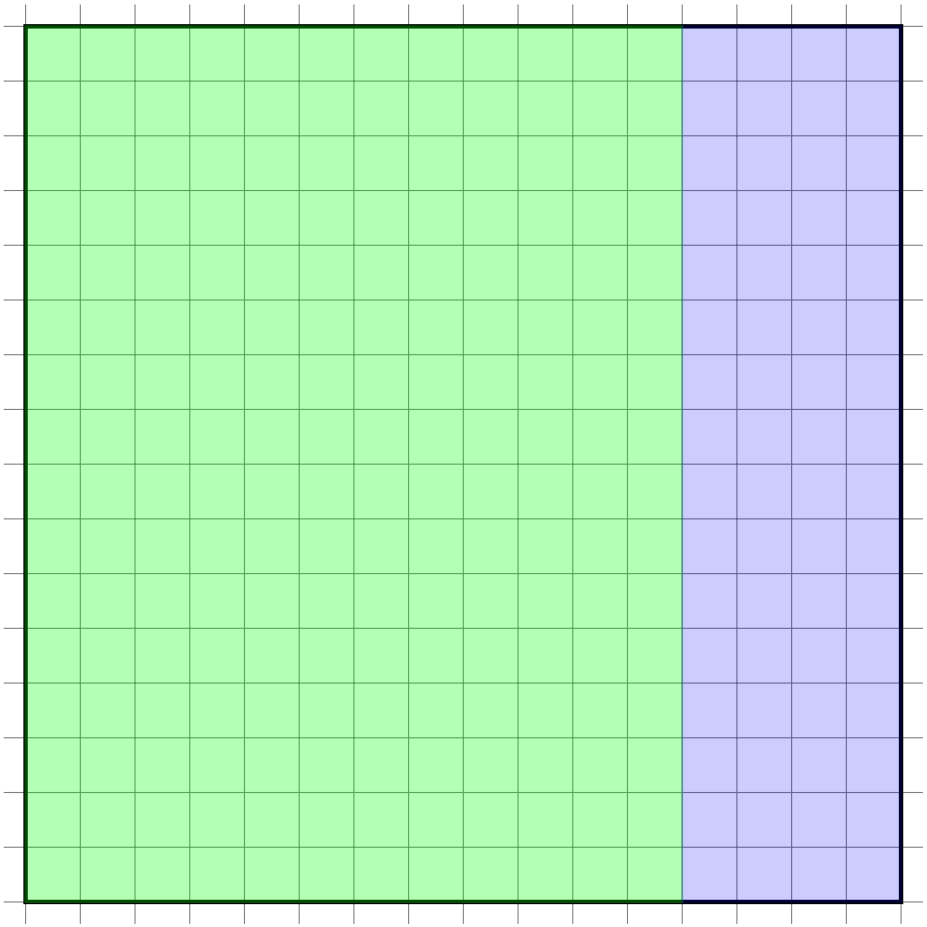
Flexible



Not simple



Unflexible



Introduction

From our users

- Calling microservices written in other languages directly.
- Modular distribution of policies.
- Reduce manual change when upgrading.
- Customization of error handling in pre and post PEPs.
- New policy enforcement points.
- Full auditing of data access operations.
- Native performance for event tracking rules.

In addition

- Allows us to easily create new rule engine plugins (100 LOC)
- Run multiple rule engines concurrently, full backward compatibility
- Interoperablity with other programming languages, Python, C++, etc.

What do we talk about?

Pluggable rule engine architecture

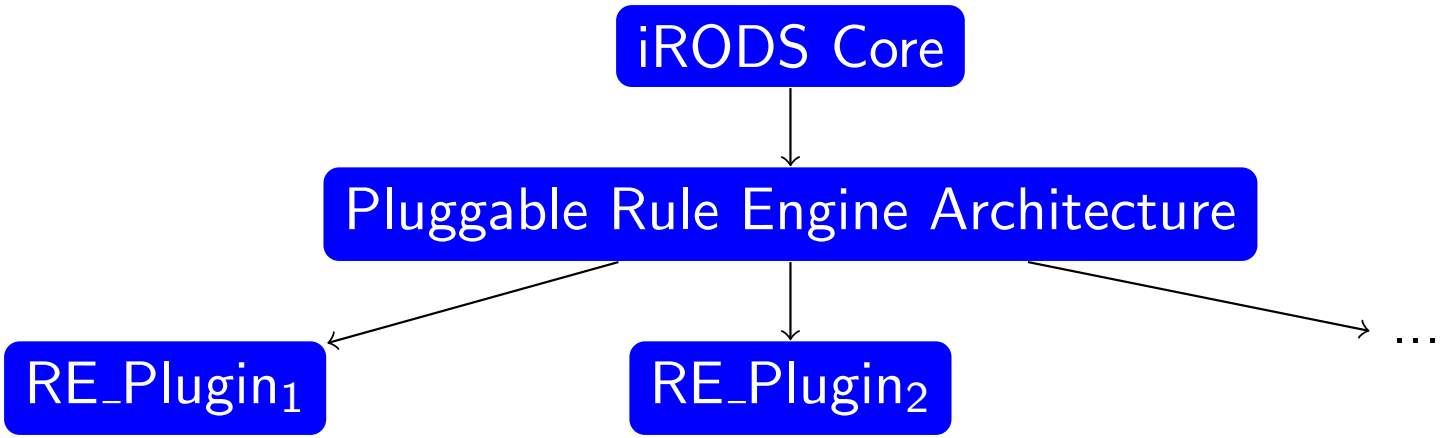
- Rule engine plugin type
- Rule engine plugin operation
- Rule engine plugins

Generalized PEP

- Policy about policy enforcement
- Namespace enables modular distribution of policy sets
- Auditing

Table of Contents

One slide summary



All you need to know to implement a rule engine plugin

```
start, stop, rule_exists, exec_rule
```

- The `callback` object is the only point of entry from the rule engine plugin to the iRODS core system.
- Rules defined in one rule engine may call rules defined in another rule engine.

Applications

libre	???
libre-audit	???
libre-irods	iRODS Rule engine is converted to a plugin
libre-python	We have created a proof of concept Python rule engine plugin. 100 LOC.
libre-v8	Javascript

Write specialized plugins with fixed policies (e.g. auditing, indexing)

The plugins combined allow rules written in the iRODS rule language to call Python and Javascript functions and Python and Javascript code to call rules written in the iRODS rule language.

Easier to upgrade

Example

user.re

```
acPostProcForPut {
    delay("<PLUSET>0s</PLUSET>") {
        remote("computing-resource", "") {
            pythonFunc($objPath);
        }
    }
}
```

user.py

```
def pythonFunc(objPath, irods):
    irods.writeLine("serverLog", "do␣Python␣stuff")
```

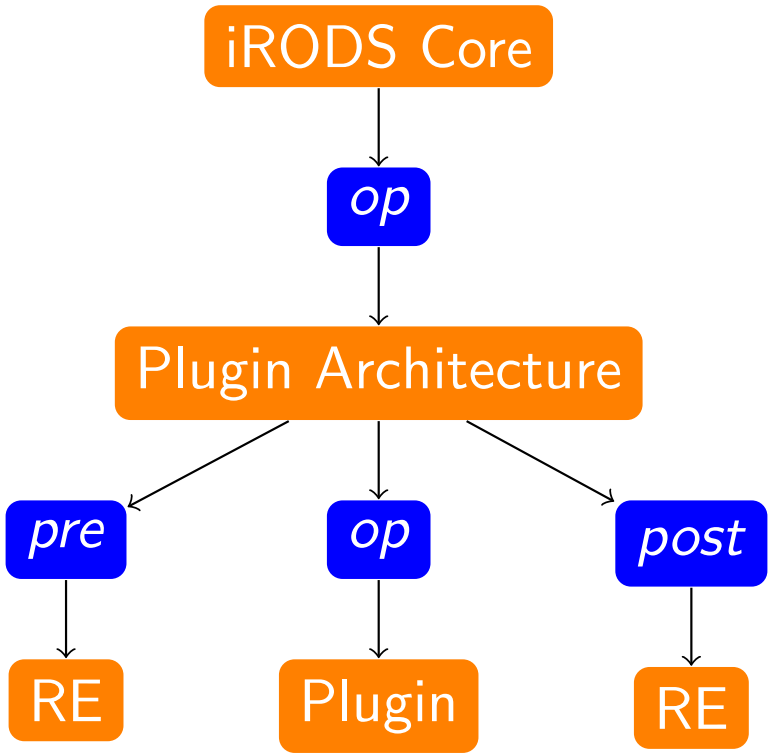
Table of Contents

iRODS PEP

Policy-based Distributed Data Management Systems (Arcot Rajasekar, Reagan Moore, Mike Wan, Wayne Schroeder, TDL 2010)
Dynamic PEP (Jason Coposky, iRODS 4.0)
Building an Extensible File System via Policy-based Data Management (Xu et al, 2014) Link

$$f[op(args, env)] = pre_{op}(args, env) >> op(args, env) >> post_{op}(args, env)$$

Dynamic PEP



Pros and cons

- This form can be used to adopt a wide-range of applications.
- The semantics of f must be fixed in an iRODS implementation
- Conundrum for iRODS devs: For example, should we make op to be skipped if pre_{op} fails? Should we still call $post_{op}$?
- We want to ensure that all policy enforcement semantics are configurable. configurability + strong default
- The key: the capability to write policies for policy enforcement.

Generalized PEP

Given a set of plugin operations, the pluggable rule engine architecture generates a PEP-added action as follows:

$$f_{generalized}[op(args, env)] = pep(op, args, env)$$

pep is a higher-order function

Run *op* if pre_{op} fails

```
pep(op, args, env) {
    pre_op(args,env);
    op(args,env);
    post_op(args,env);
}
```

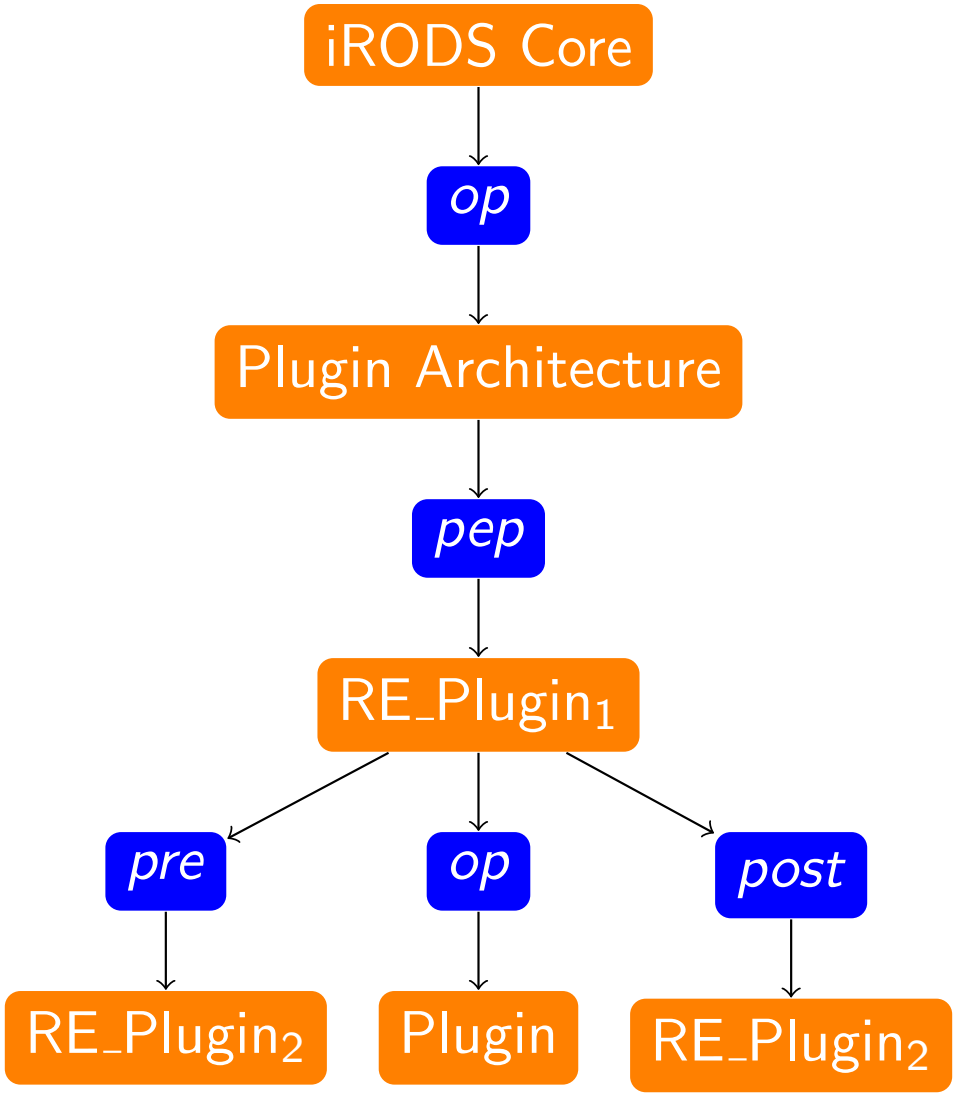
Skip *op* and $post_{op}$ if pre_{op} fails

```
pep(op, args, env) {
    if(pre_op(args,env) >= 0) {
        op(args,env);
        post_op(args,env);
    }
}
```

Run $post_{op}$ if pre_{op} fails

```
pep(op, args, env) {
    if(pre_op(args,env) >= 0) {
        op(args,env);
    }
    post_op(args,env);
}
```

Specialization



Namespaces

- RE_Plugin$_1$ provides extended namespace support for the translation to the default semantics.

$$
\begin{aligned}
pep[\ldots] \quad = \quad & ns_1 pre_{op}(args, env) >> \ldots ns_n pre_{op}(args, env) >> \\
& op(args, env) >> \\
& ns_n post_{op}(args, env) >> \ldots ns_1 post_{op}(args, env)
\end{aligned}
$$

- By default, we have namespace $ns_1 =$ "".
- We can add more namespaces. For example, for auditing $ns_2 =$ "*audit_*" or indexing $ns_3 =$ "*index_*". For the *audit_* namespace, pre and post file read PEPs:

```
audit_pep_resource_read_pre
audit_pep_resource_read_post
```

Event tracking

- The `audit` plugin provides an asynchronous tracking mechanism for every operation and their arguments and environments in iRODS, thereby providing a complete log.
- It runs at native code speed.
- It supports any future plugin operation automatically.
- The rules listen to the *audit_* namespace. a pre and post file read rule can be provided as follows (showing iRODS rule language equivalence to the C++ implementation):

```
audit_pep_resource_read_pre (...) {
    writeLine("serverLog", ...);
}
audit_pep_resource_read_post (...) {
    writeLine("serverLog", ...);
}
```

Example

server_config.json

```
...
{
    "instance_name": "re-audit-instance",
    "plugin_name": "re-audit"
}
...
{
    "namespace": "audit_"
}
...
```