# Pluggable Rule Engine Architecture

**Hao Xu**
DICE Center
University of North Carolina at Chapel Hill,
NC 27599, USA
xuh@email.unc.edu

**Jason Coposky**
Renaissance Computing Institute (RENCI)
100 Europa Drive Suite 540 Chapel Hill, North Carolina 27517
jasonc@renci.org

**Ben Keller**
Renaissance Computing Institute (RENCI)
100 Europa Drive Suite 540 Chapel Hill, North Carolina 27517
kellerb@renci.org

**Terrell Russell**
Renaissance Computing Institute (RENCI)
100 Europa Drive Suite 540 Chapel Hill, North Carolina 27517
unc@terrellrussell.com

## ABSTRACT

We describe a new development in the next release of iRODS. The pluggable rule engine architecture allows us to easily create new rule engines as plugins and run multiple rule engines concurrently. The pluggable rule engine architecture allows easy implementation and maintenance of rule engine plugin code and offers significant performance gains in some use cases. The pluggable rule engine architecture enables modular incorporation of features from other programming languages, allows efficient auditing of interactions between user-defined rules and the iRODS system, and supports full interoperability between rules and libraries written in different languages. This design allows us to easily incorporate libraries designed for different programming languages, for example, Python, C++, etc., into the policy sets, significantly enhancing the capabilities of iRODS without syntactic overhead. This new architecture enables a wide range of important applications including auditing, indexing, and modular distribution of policies. We demonstrate how to create the Python rule engine plugin and how to create user defined policy plugins.

### Keywords

Pluggable Policy, Rule Engine, Plugin Architecture

## INTRODUCTION

In this paper, we are going to describe a new development in the next release of iRODS. The pluggable rule engine architecture allows us to easily create new rule engines as plugins and run multiple rule engines concurrently. The pluggable rule engine architecture allows easy implementation and maintenance of rule engine plugin code and offers significant performance gains in some use cases. The pluggable rule engine architecture enables modular incorporation of features from other programming languages, allows efficient auditing of interactions of user-defined rules and the iRODS system, and supports full interoperability between rules and libraries written in different languages. This design allows us to easily incorporate libraries designed for different programming languages, for example, Python, C++, etc., into the policy sets, significantly enhancing the capabilities of iRODS without syntactic overhead. This new architecture enables a wide range of important applications including auditing, indexing, and modular distribution of policies.

Users of iRODS have expressed the following areas of improvement:

*iRODS UGM 2015* June 10-11, 2015, Chapel Hill, NC

- customization of error handling in pre and post PEPs.
- calling microservices written in other languages directly.
- native performance for event tracking rules.
- modular distribution of policies.
- full auditing of data access operations.
- reduce manual change when upgrading.
- new policy enforcement points.

The pluggable rule engine architecture addresses these challenges.

## THE DESIGN

In this section, we overview the key designs in the pluggable rule engine architecture.

### iRODS Features

iRODS supports a wide range of plugin types. This allows the core iRODS to be independent from the components that it uses. For example, the database plugin allows iRODS to use different databases without changing the core code. Each plugin has a set of defined operations that it has to provide. The core interacts with plugins only through those operations. One benefit of this design is that we can easily capture all state changing operations by looking at plugin operations. And we can show that such capture is complete in the following sense. If we want to capture all database operations, we only need to look at database plugin operations. Because of the ignorance of the underlying implementation of these operations, the core cannot perform any additional operations than those provided by the plugin architecture. Therefore, if we capture all operations in the plugin architecture, we capture all state changing operations.

In iRODS, a pair of pre and post PEPs are automatically generated for every defined plugin operation. This way we ensure that all policy enforcement points are present. Having the capability to write policies for every state changing operation, we make the complete information about each operation available to the PEPs by making the argument and environment in which the operation is called available to the PEPs. This way the PEPs can determine what to do based on this information.

Formally speaking, let $Op$ denote the set of plugin operations, and $Act$ denote the set of actions, with

$$Op \subset Act$$

Let $f$ denote the function that generates an action from a plugin operation. For example, given a plugin operation, the plugin architecture generates a PEP-added action $Act$ comprising of pre and post operations PEPs as follows:

$$f : Op \to Act$$

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

Here the sequential combination operator can be thought of as the monadic *bind* operator. This formalism can be used to adopt a wide-range of applications. One of the disadvantages of this design is that the semantics of $f$ must be fixed in an iRODS implementation, for example, how the error is handled. And the particular form $f$ lacks principal error handling semantics, i.e., one which fits all of our users' use cases by just varying pre and post PEPs. For example, should we make $op$ to be skipped if $pre_{op}$ fails? Should we still call $post_{op}$? This problem can be solved by providing a generalization that can be customized by plugins.

### Pluggable Rule Engine Architecture

The pluggable rule engine architecture generalizes the current design and is fully backward compatible. The design provides a global policy enforcement that can be further customized for different semantics.

An example is that you can have error handling semantics encapsulated in a plugin, and by installing that plugin, you enable those error handling semantics. This requires the plugin architecture to load multiple rule engine plugins at the same time, and in a way that one plugin may provide semantics for another plugin.

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

$$f : Op \rightarrow Act$$

$$f[op(args, env)] = pep(op, args, env)$$

To recover the default behavior, we can define $pep$ as

$$pep : Op \times Args \times Env \rightarrow Act$$

$$pep[op, args, env] = pre_{op}(args, env); op(args, env); post_{op}(args, env)$$

We can define different error handling semantics as follows:

Skip $post_{op}$ if $pre_{op}$ fails:

$$pep_1 : Op \times Args \times Env \rightarrow Act$$

$$pep[op, args, env] = if(pre_{op}(args, env) >= 0)\{op(args, env); post_{op}(args, env)\}$$

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

$$pep_2 : Op \times Args \times Env \rightarrow Act$$

$$pep[op, args, env] = if(pre_{op}(args, env) >= 0)\{op(args, env)\}; post_{op}(args, env)$$

This way the rule engines form a hierarchy, with rule engines gradually refining the semantics of plugin operations. We can define such a hierarchy so that it is fully compatible with the current semantics, with the current rule engine at the bottom of the hierarchy, so that all existing rules run as expected. We can also, when new use cases arise, define a different set of plugins that implement different semantics, without changing the core code. This gives our users the flexibility to implement their policies.

Another challenge is the inter-rule-engine-call (IREC). Each rule engine provides a set of rules that it defines. Rules defined in one rule engine should be able to call rules defined in another rule engine. This is done through a universal callback function. The universal callback function is the only point of entry from the rule engine plugin to the iRODS core system. It handles all operations including accessing state information, accessing session variables, and the IREC. The general format of a callback is

$$fn(args)$$

where $fn$ is a callback name and $args$ is a list of arguments. In the case of IREC, $fn$ is the name of the rule and $args$ are the arguments to the rule. Compared to exposing a server API to the rule engine plugin, this approach has several advantages: First, this enables calling functions written in other programming languages as if they are microservices. Second, it allows us to add new APIs without changing the rule engine plugin interface. Third, we can add a pair of PEPs to this operation, which is sufficient for monitoring all interactions from the rule engine back to the core.

### IMPLEMENTATION

The rule engine plugin architecture allows loading of multiple types of rule engine plugins, and multiple instances of each type of rule engine plugin. All instances share the same plugin object, but with different contexts. This way we don't have to load a rule engine plugin multiple times.

The rule engine contains the following four operations given in C++:

```
template<typename T>
irods:error start(T&);
template<typename T>
irods::error stop(T&);
template<typename T>
irods::error rule_exists(std::string, T&, bool&);
template<typename T>
irods::error exec_rule(std::string, T&, std::list<boost::any>&, callback);
```

The `start` function is called when the rule engine plugin is started. This happens when an iRODS process starts. The `stop` function is called when the rule engine is stopped. This happens when an iRODS process stops. The parameter is an object that can be used to pass data to and from these functions as well as other functions in the plugin operation. It can be thought of as the context. In fact, the state information can only be stored in this object. When the rule engine plugin manager loads more than one instance of the same plugin, the only object that is newly created is this object.

The `rule_exists` function accepts a rule name, a context, and writes back whether the rule exists in this plugin.

The `exec_rule` function accepts a rule name, a context, a list of arguments, and a callback object. The list of arguments are boxed by boost::any, and stored in a std::list container. This allows us to load the function in a dynamically linked library. The callback object is a C++ Callable, with the following interface method:

```
template<typename ...As>
irods::error operator()(std::string, As&&...);
```

The first parameter is $fn$. The second, third, etc. parameters are $args$.

One may have noticed that the callback interface expects raw values whereas the `exec_rule` function expects a list of values boxed by boost::any. Why do we design them like this? Ideally we would like to always use raw values to maximize efficiency, but this would require templates. We can accept raw parameters for the callback interface because it is statically compiled. But to allow the `exec_rule` to be loaded from a dynamic library, we cannot use templates. Because C++ templates are expanded at compile time, we cannot put a template function in a dynamically linked library that is linked to the main program at runtime. Wouldn't this be inefficient if the rule engine plugin simply wants pass the list of incoming arguments to the callback? The answer is to use the `unpack` construct as follows:

```
irods::error exec_rule(std::string _rn, T& _re_ctx, std::list<boost::any>& _ps, callback _cb) {
    cb(rn2, irods::unpack(_ps));
}
```

The `unpack` constructor is implemented so that the time complexity is $O(1)$.

The default implementation comes with a default rule engine. The default rule engine only has the *pep* rule and provides an implementation of the generalized PEP. It provides extended namespace support for the translation

to the default semantics. Formally speaking, it implements the following function, given a list of $n$ namespaces $ns_i, i \in \{1, \ldots, n\}$ (configured in `server_config.json`)

$$\begin{aligned} pep &: Op \times Args \times Env \to Act \\ pep[op, args, env] &= ns_1 pre_{op}(args, env); \ldots ns_n pre_{op}(args, env); \\ &\quad op(args, env); ns_n post_{op}(args, env); \ldots ns_1 post_{op}(args, env) \end{aligned}$$

Here, for simplicity, we omitted error handling semantics.

By default, we have only one namespace which is $ns_1 =$ "", which implements the default semantics. We can implement different semantics outlined in the previous section by changing this plugin. We can add more namespaces and keep the default semantics. For example, we can add in another namespace for auditing $ns_2 =$ "$audit\_$" or indexing $ns_3 =$ "$index\_$". The rules listen to the *audit* namespace. For example pre and post file read PEPs are provided as follows:

$$audit\_pep\_resource\_read\_pre$$

$$audit\_pep\_resource\_read\_post$$

Rule engine plugins can be written to listen to those namespaces and provide the specific functionalities in a modular fashion. When a set of specialized plugins are installed, we can switch on/off a feature by just changing which namespaces are available.

## APPLICATIONS

### Python Rule Engine

We have created a proof of concept Python rule engine plugin. It allows users to implement PEPs directly in Python. This provides an avenue for the rapid expansion in the functionality of iRODS deployments, by taking advantage of the vast ecosystem of existing Python libraries as well as the large community of Python developers.

The plugin translates calls to `exec_rule` into calls to Python functions, whose implementations are loaded from `/etc/irods/core.py`, a Python code file. Users of the plugin are only required to write Python code, and are able to use all features of the Python programming language, including importing arbitrary Python modules.

Because of the pluggable rule engine architecture, this means iRODS users will be able to implement all PEPs directly in Python, or to call out to Python from other rule engine plugins, e.g. to extend the functionality of existing iRODS rules.

### 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. Because the PEPs are dynamically generated, it supports any future plugin operation automatically. It allows the log to be sent to a remote system and processed asynchronously[1].

The rules listen to the *audit* namespace. To illustrate the implementation, a pre and post file read rule can be provided as follows:

[1] currently under development

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

In our implementation, these rules are implemented directly in C++ and therefore incur minimum overhead over normal operations.

## CONCLUSION

We described a new development in the next release of iRODS. The pluggable rule engine architecture allows us to easily create new rule engines as plugins and run multiple rule engines concurrently. The pluggable rule engine architecture allows easy implementation and maintenance of rule engine plugin code and offers significant performance gains in some use cases. The pluggable rule engine architecture enables modular incorporation of features from other programming languages, allows efficient auditing of interactions of user-defined rules and the iRODS system, and supports full interoperability between rules and libraries written in different languages. This design allows us to easily incorporate libraries designed for different programming languages, for example, Python, C++, etc., into the policy sets, significantly enhancing the capabilities of iRODS without syntactic overhead. This new architecture enables a wide range of important applications including auditing, indexing, and modular distribution of policies.