A Definition of Data Management

"The development, execution and supervision of plans, policies, programs, and practices that control, protect, deliver, and enhance the value of data and information assets."

Organizations need a future-proof solution to managing data and its surrounding infrastructure.
What is a Policy

A Definition of Policy

A set of ideas or a plan of what to do in particular situations that has been agreed to officially by a group of people...

So how does iRODS do this?
The reflection of real world data management decisions in computer actionable code.

(a plan of what to do in particular situations)
Possible Policies

- Data Movement
- Data Verification
- Data Retention
- Data Replication
- Data Placement
- Checksum Validation
- Metadata Extraction
- Metadata Application
- Metadata Conformance
- Replica Verification
- Vault to Catalog Verification
- Catalog to Vault Verification
- ...
The iRODS Data Management Model

Core Competencies

Policy

Capabilities

Patterns

Core Competencies
Some Questions

- How can we help new users get started?
- How can we make policy reusable?
- How can we simplify policy development?
- How can we provide a cook book of deployments?
- How do we get from Policy to Capabilities?
Consider Policy as building blocks towards Capabilities

Follow proven software engineering principles:
  Favor composition over monolithic implementations

Rules and Dynamic Policy Enforcement Points can be overloaded and fall through

Implement or configure several rule bases or rule engine plugins to achieve complex use cases
The Original Approach

Assuming there was even a provided policy enforcement point for the desired event...

```c
acPostProcForPut() {
    if($rescName == "demoResc") {
        # extract and apply metadata
    } else if($rescName == "cacheResc") {
        # async replication to archive
    } else if($objPath like "/tempZone/home/alice/*" && $rescName == "indexResc") {
        # launch an indexing job
    } else if(.xyz) {
        # compute checksums ...
    } else {
        # and so on ...
    }
```

In `/etc/irods/core.re`...
Our second approach

Expanding policy implementation across rule bases

For example: `pep_data_obj_put_post(...)`

- Metadata extraction and application
- Asynchronous Replication
- Initiate Indexing
- Apply access time metadata
- Asynchronous checksum computation

Rather than one monolithic implementation, separate the implementations into individual rule bases, or plugins, and allow the rule(s) to fall through
Expanding policy across rule bases

Separate the implementation into several rule bases:

```
/etc/irods/metadata.re

    # metadata extraction and application code
    RULE_ENGINE_CONTINUE
}
```

```
/etc/irods/checksum.re

    # checksum code
    RULE_ENGINE_CONTINUE
}
```

```
/etc/irods/access_time.re

    # access time application code
    RULE_ENGINE_CONTINUE
}
```
Within the Rule Engine Plugin Framework, order matters

```json
"rule_engines": [
    {
        "instance_name": "irods_rule_engine_plugin-irods_rule_language-instance",
        "plugin_name": "irods_rule_engine_plugin-irods_rule_language",
        "plugin_specific_configuration": {
            "re_rulebase_set": [
                "metadata",
                "checksum",
                "access_time",
                "core"
            ]
        }
    },
    {
        "instance_name": "irods_rule_engine_plugin-cpp_default_policy-instance",
        "plugin_name": "irods_rule_engine_plugin-cpp_default_policy",
        "plugin_specific_configuration": {
            "shared_memory_instance": "irods_rule_language_rule_engine"
        }
    }
]```
Consider Storage Tiering as a collection of policies:

- Data Access Time
- Identifying Violating Objects
- Data Replication
- Data Verification
- Data Retention
Policies composed by a monolithic framework plugins

Policy delegated by naming convention:

- `irods_policy_access_time`
- `irods_policy_data_movement`
- `irods_policy_data_replication`
- `irods_policy_data_verification`
- `irods_policy_data_retention`

Each policy may be overridden by another rule engine, or rule base to customize to future use cases or technologies
The New Approach

Continue to separate the concerns:

- **When**: Which policy enforcement points
- **What**: The policy to be invoked
- **Why**: What are the conditions necessary for invocation
- **How**: Synchronous or Asynchronous

Write simple policy implementations

- Not tied to a Policy Enforcement Point
- Do one thing well
- How it is invoked is of no concern

Each policy may now be reused in a generic fashion, favoring configuration over code.
The When
A Rule Engine Plugin for a specific Class of events

- Data Object
- Collection
- Metadata
- User
- Resource

The Events are specific to the class of the handler

The handler then invokes policy based on its configuration
A Rule Engine Plugin for data creation and modification events

- Create
- Read
- Replication
- Unlink
- Rename
- Register

Policy invocation is configured as an array of json objects for any given combination of events

**Unifies the POSIX and Object behaviors into a single place to configure policy**
Example: Synchronous Invocation

```
{
  "instance_name": "irods_rule_engine_plugin-event_handler-data_object_modified-instance",
  "plugin_name": "irods_rule_engine_plugin-event_handler-data_object_modified",
  "plugin_specific_configuration": {
    "policies_to_invoke": [
      {
        "active_policy_clauses": ["post"],
        "events": ["create", "write", "registration"],
        "policy_to_invoke": "irods_policy_access_time",
        "configuration": {}
      },
      {
        "active_policy_clauses": ["pre"],
        "events": ["replication"],
        "policy_to_invoke": "irods_policy_example_policy",
        "configuration": {}
      }
    ]
  }
}
```

Note that order still matters if more than one policy needs to be invoked for a given event.
The What
Basic policies that are leveraged across many deployments and capabilities:

- irods_policy_access_time
- irods_policy_query_processor
- irods_policy_data_movement
- irods_policy_data_replication
- irods_policy_data_verification
- irods_policy_data_retention

The library will continue to grow, with a cookbook of usages.
Standardized JSON interface : parameters, and configuration

iRODS Rule Language

```
1  irods_policy_example_policy_implementation(*parameters, *configuration) {
2  }
```

Python Rule Language

```
1  def irods_policy_example_policy_implementation(rule_args, callback, rei):
2     # Parameters     rule_args[1]
3     # Configuration rule_args[2]
```

Policy can also be implemented as fast and light C++ rule engine plugins
Policy may be invoked using one of three different conventions:

- Direct Invocation: a JSON object
- Query Processor: a JSON array of parameters
- Event Handler: a JSON object

Each invocation convention defines its interface by contract.
Direct Invocation: Parameters passed as a JSON object

```javascript
my_rule() {
    irods_policy_access_time("{"object_path" : "/tempZone/home/rods/file0.txt"}", "");
}
```

Parameters may also be configured statically

```javascript
{
    "policy" : "irods_policy_execute_rule",
    "payload" : {
        "policy_to_invoke" : "irods_policy_storage_tiering",
        "parameters" : {
            "object_path" : "/tempZone/home/rods/file0.txt"
        }
    }
}
```
Query Processor Invocation

Serializes results to JSON array and passed to the policy via the parameter object as "query_results"

```json
1 {
    "policy_to_invoke": "irods_policy_enqueue_rule",
    "parameters": {
        "delay_conditions": "<PLUSET>1s</PLUSET>",
        "policy_to_invoke": "irods_policy_execute_rule",
        "parameters": {
            "policy_to_invoke": "irods_policy_query_processor",
            "parameters": {
                "query_string": "SELECT USER_NAME, COLL_NAME, DATA_NAME, RESC_NAME WHERE COLL_NAME like '/tempZone/home/rods%'",
                "query_limit": 10,
                "query_type": "general",
                "number_of_threads": 4,
                "policy_to_invoke": "irods_policy_engine_example"
            }
        }
    }
}
```

For example the invoked policy would receive a row:
['rods', '/tempZone/home/rods/', 'file0.txt', 'demoResc']
Event Handler Invocation

Serializes `dataObjInp_t` and `rsComm_t` to a JSON object

```json
{
    "comm": {
        "auth_scheme": "native", "client_addr": "152.54.8.141", "proxy_auth_info_auth_flag": "5", "proxy_auth_info_auth_scheme": "", "proxy_auth_info_auth_str": "", "proxy_auth_info_host": "", "proxy_auth_info_ppid": "0", "proxy_rods_zone": "tempZone", "proxy_sys_uid": "0", "proxy_user_name": "rods", "proxy_user_other_info_user_comments": "", "proxy_user_other_info_user_create": "", "proxy_user_other_info_user_info": "", "proxy_user_type": "", "user_auth_info_auth_flag": "5", "user_auth_info_auth_scheme": "", "user_auth_info_auth_str": "", "user_auth_info_host": "", "user_auth_info_ppid": "0", "user_rods_zone": "tempZone", "user_sys_uid": "0", "user_user_name": "rods", "user_user_other_info_user_comments": "", "user_user_other_info_user_create": "", "user_user_other_info_user_info": "", "user_user_other_info_user_modify": "", "user_user_type": "",
        "cond_input": {
            "dataIncluded": "", "dataType": "generic", "destRescName": "ufs0", "noOpenFlag": "", "openType": "1", "recursiveOpr": "1", "resc_hier": "ufs0", "selObjType": "dataObj", "translatedPath": ""
        }
    },
    "create_mode": "33204", "data_size": "1", "event": "CREATE", "num_threads": "0", "obj_path": "/tempZone/home/rods/test_put_gt_max_sql_rows/junk0083", "offset": "0", "open_flags": "2", "opr_type": "1", "policy_enforcement_point": "pep_api_data_obj_put_post"
}
```

Which is also passed in as the parameter object
Configuration

Any additional statically set context passed into the policy

```
{
  "policy_to_invoke": "irods_policy_access_time",
  "configuration": {
    "attribute": "irods::access_time"
  }
}
```

May be "plugin_specific_configuration" from a rule engine plugin or "configuration" from within the event framework.
The Why
Each invoked policy may set a conditional around each noun within the system which gates the invocation

- Data Object
- Collection
- Metadata
- User
- Resource

Leverages boost::regex to match any combination of logical_path, metadata, resource name, etc.
Matching a logical path for replication policy invocation

```json
1 { 
2    "instance_name": "irods_rule_engine_plugin-event_handler-data_object_modified-instance",
3    "plugin_name": "irods_rule_engine_plugin-event_handler-data_object_modified",
4    "plugin_specific_configuration": { 
5        "policies_to_invoke": [ 
6        { 
7            "conditional": { 
8                "logical_path": "\tempZone.*"
9            },
10            "active_policy_clauses": ["post"],
11            "events": ["put"],
12            "policy_to_invoke": "irods_policy_data_replication",
13            "configuration": { 
14                "source_to_destination_map": { 
15                    "demoResc": ["AnotherResc"]
16                }
17            }
18        },
19        ... 
20    ]
21    }
22 }
23 }
```
Matching metadata for indexing policy invocation

```json
{
  "instance_name": "irods_rule_engine_plugin-event_handler-data_object_modified-instance",
  "plugin_name": "irods_rule_engine_plugin-event_handler-data_object_modified",
  "plugin_specific_configuration": {
    "policies_to_invoke": [
      {
        "active_policy_clauses": ["post"],
        "events": ["put", "write"],
        "conditional": {
          "metadata_exists": {
            "recursive": "true",
            "attribute": "irods::indexing::index",
            "entity_type": "collection"
          }
        }
      }
    ],
    "policy_to_invoke": "irods_policy_indexing_full_text_index_elasticsearch",
    "configuration": {
      "hosts": ["http://localhost:9200/"],
      "bulk_count": 100,
      "read_size": 1024
    }
  }
}
```
The How
The cpp_default rule engine plugin in 4.2.8+ will now support two new policies:

- irods_policy_enqueue_rule
- irods_policy_execute_rule

The enqueue rule policy will push a job onto the delayed execution queue. The "payload" object holds the rule which is to be executed.
How - Asynchronous Execution

The execute rule policy invokes a policy engine either from the delayed execution queue or as a direct invocation.

```json
1 { 
  "policy_to_invoke" : "irods_policy_enqueue_rule",
  "parameters" : {
    "comment" : "Set the PLUSET value to the interval desired to run the rule",
    "delay_conditions" : "<PLUSET>10s</PLUSET><EF>REPEAT FOR EVER</EF><INST_NAME>irods_rule_engine_plugin-cpp_default_policy-instance</INST_NAME>",
    "policy_to_invoke" : "irods_policy_execute_rule",
    "parameters" : {
      "policy_to_invoke" : "irods_policy_filesystem_usage",
      "parameters" : {
        "source_resource" : "demoResc"
      }
    }
  }

INPUT null
OUTPUT ruleExecOut
```
The New Approach

- When: Which policy enforcement points
- What: The policy to be invoked
- Why: What are the conditions necessary for invocation
- How: Synchronous or Asynchronous
Examples
```json
{
    "instance_name": "irods_rule_engine_plugin-event_handler-data_object_modified-instance",
    "plugin_name": "irods_rule_engine_plugin-event_handler-data_object_modified",
    "plugin_specific_configuration": {
        "policies_to_invoke": [
            {
                "active_policy_clauses": ["post"],
                "events": ["put", "get", "create", "read", "write", "rename",
                            "register", "unregister", "replication", "checksum",
                            "copy", "seek", "truncate"],
                "policy_to_invoke": "irods_policy_access_time",
                "configuration": {
                ...
            }
        ]
    }
}
```
Synchronous Replication

```json
{
  "instance_name": "irods_rule_engine_plugin-event_handler-data_object_modified-instance",
  "plugin_name": "irods_rule_engine_plugin-event_handler-data_object_modified",
  "plugin_specific_configuration": {
    "policies_to_invoke": [
      {
        "active_policy_clauses": ["post"],
        "events": ["create", "write", "registration"],
        "policy_to_invoke": "irods_policy_data_replication",
        "configuration": {
          "source_to_destination_map": {
            "source_resource_0": ["destination_resource_0a", "destination_resource_0b"],
            "source_resource_1": ["destination_resource_1a"],
          }
        }
      },
      {
        "active_policy_clauses": ["post"],
        "events": ["create", "write", "registration"],
        "policy_to_invoke": "irods_policy_data_replication",
        "configuration": {
          "destination_resource": "destination_resource_3"
        }
      }
    ]
  }
}
```
Asynchronous Replication

```json
{
  "instance_name": "irods_rule_engine_plugin-event_handler-data_object_modified-instance",
  "plugin_name": "irods_rule_engine_plugin-event_handler-data_object_modified",
  "plugin_specific_configuration": {
    "policies_to_invoke": [
      {
        "active_policy_clauses": ["post"],
        "events": ["create", "write", "registration"],
        "policy_to_invoke": "irods_policy_enqueue_rule",
        "parameters": {
          "delay_conditions": "<ET>PLUSET 1</ET>",
          "policy_to_invoke": "irods_policy_execute_rule",
          "parameters": {
            "policy_to_invoke": "irods_policy_data_replication",
            "configuration": {
              "source_to_destination_map": {
                "source_resource_0": ["destination_resource_0a", "destination_resource_0b"],
                "source_resource_1": ["destination_resource_1a"],
              }
            }
          }
        }
      }
    ]
  }
}
```
Synchronous Retention

```json
1 {
2   "instance_name": "irods_rule_engine_plugin-event_handler-data_object_modified-instance",
3   "plugin_name": "irods_rule_engine_plugin-event_handler-data_object_modified",
4   "plugin_specific_configuration": {
5     "policies_to_invoke": [
6       {
7         "active_policy_clauses": ["post"],
8         "events": ["replication"],
9         "policy_to_invoke": "irods_policy_data_retention",
10        "configuration": {
11          "mode": "trim_single_replica",
12          "source_resource_list": ["source_resource_1", "source_resource_2"],
13        }  
14       }
15     ]
16   }
17 }```
{ 
  "policy_to_invoke": "irods_policy_enqueue_rule",
  "parameters": {
    "delay_conditions": ":<EF>REPEAT FOR EVER</EF>",
    "policy_to_invoke": "irods_policy_execute_rule",
    "parameters": {
      "policy_to_invoke": "irods_policy_query_processor",
      "parameters": {
        "query_string": "SELECT USER_NAME, COLL_NAME, DATA_NAME, RESC_NAME WHERE COLL_NAME like '/tempZone/home/rods%' AND RESC_NAME IN ('source_resource_1', 'source_resource_2')",
        "query_limit": 10,
        "query_type": "general",
        "number_of_threads": 4,
        "policy_to_invoke": "irods_policy_data_retention",
        "configuration": {
          "mode": "trim_single_replica",
          "source_resource_list": ["source_resource_1", "source_resource_2"]
        }
      }
    }
  }
}
The type of verification to perform is stored as metadata on the resource:

- catalog
- filesystem
- checksum
The type of verification to perform is stored as metadata on the resource

- catalog
- filesystem
- checksum
Policy Composed Capabilities
Storage Tiering Overview

Periodically, the storage tiering policy discovers data objects in violation via a default query and schedules their migration to the next tier group.

Unified Namespace

After 1800 seconds, any data objects in violation are automatically replicated to tier 1, and then once at rest, they are trimmed from tier 0.

After 9000 seconds, any data objects in violation are automatically replicated to tier 2, and then once at rest, they are trimmed from tier 1.

Your Organization

Tier 0 (Fast)
Tier 1 (Intermediate)
Tier 2 (Slow)

The default query that determines which data objects are in violation can be overridden by adding a new metadata attribute irods::storage_tier_query with a value that defines the custom query.

Federate Securely

Other Organization

Data Virtualization (Unified Namespace)
Data Discovery (Metadata)
Workflow Automation (Rule Engine)
Secure Collaboration (Federation)
Policy Composed Storage Tiering

- Asynchronous Discovery
- Asynchronous Replication
- Synchronous Retention
- Resource associated metadata
- Identified by 'tiering groups'
Asynchronous Discovery and Replication

```json
{
  "policy_to_invoke": "irods_policy_execute_rule",

  "parameters": {
    "policy_to_invoke": "irods_policy_query_processor",

    "configuration": {
      "query_string": "SELECT META_RESC_ATTR_VALUE WHERE META_RESC_ATTR_NAME = 'irods::storage_tiering::group'",
      "query_limit": 0,
      "query_type": "general",
      "number_of_threads": 8,
      "policy_to_invoke": "irods_policy_event_generator_resource_metadata",

      "configuration": {
        "conditional": {
          "metadata_exists": {
            "attribute": "irods::storage_tiering::group",
            "value": "(0)"
          }
        }
      }
    }

    "policies_to_invoke": [
      {
        "policy_to_invoke": "irods_policy_query_processor",
        "configuration": {
          "query_string": "SELECT META_RESC_ATTR_VALUE WHERE META_RESC_ATTR_NAME = 'irods::storage_tiering::query' AND RESC_NAME = 'IRODS_TOKEN_SO",
          "default_results_when_no_rows_found": ["SELECT USER_NAME, COLL_NAME, DATA_NAME, RESC_NAME WHERE META_DATA_ATTR_NAME = 'irods::access_tiering::query'"
          "query_limit": 0,
          "query_type": "general",
          "number_of_threads": 8,
          "policy_to_invoke": "irods_policy_query_processor",
          "configuration": {
            "conditional": {
              "metadata_exists": {
                "attribute": "irods::storage_tiering::group",
                "value": "(0)"
              }
            }
          }
        }
      },

      {
        "policy_to_invoke": "irods_policy_data_replication",
        "configuration": {
          "comment": "source_resource, and destination_resource supplied by the resource metadata event generator"
        }
      }
    ]
  }
}
```

INPUT null

OUTPUT ruleExecOut
Synchronous Configuration for Storage Tiering

```json
{
"instance_name": "irods_rule_engine_plugin-event_handler-data_object_modified-instance",
"plugin_name": "irods_rule_engine_plugin-event_handler-data_object_modified",
"plugin_specific_configuration": {
  "policies_to_invoke": [
    {
      "active_policy_clauses": ["post"],
      "policy_to_invoke": "irods_policy_access_time",
      "configuration": {
        "log_errors": "true"
      }
    },
    {
      "active_policy_clauses": ["post"],
      "events": ["read", "write", "get"],
      "policy_to_invoke": "irods_policy_data_restage",
      "configuration": {}
    },
    {
      "active_policy_clauses": ["post"],
      "events": ["replication"],
      "policy_to_invoke": "irods_policy_tier_group_metadata",
      "configuration": {}
    },
    {
      "active_policy_clauses": ["post"],
      "events": ["replication"],
      "policy_to_invoke": "irods_policy_data_verification",
      "configuration": {}
    },
    {
      "active_policy_clauses": ["post"],
      "events": ["replication"],
      "policy_to_invoke": "irods_policy_data_retention",
      "configuration": {
        "mode": "trim_single_replica",
        "log_errors": "true"
      }
    }
  ]
}
```
Possible Metadata Driven Restage for Storage Tiering

```json
{
  "instance_name": "irods_rule_engine_plugin-event_handler-metadata_modified-instance",
  "plugin_name": "irods_rule_engine_plugin-event_handler-metadata_modified",
  "plugin_specific_configuration": {
    "policies_to_invoke": [
      {
        "active_policy_clauses": ["post"],
        "events": ["set", "add"],
        "attribute": "irods::storage_tiering::restage",
        "value": "*.*",
        "policy_to_invoke": "irods_policy_data_restage",
        "configuration": {}
      }
    ]
  }
}
```
iRODS provides eight packaged capabilities, each of which can be selectively deployed and configured. These capabilities represent the most common use cases as identified by community participation and reporting.

The flexibility provided by this model allows an organization to address its immediate use cases. Additional capabilities may be deployed as any new requirements arise.

A pattern represents a combination of iRODS capabilities and data management policy consistent across multiple organizations. Three common patterns of iRODS deployment have been observed within the community:

- Data to Compute
- Compute to Data
- Synchronization
Capabilities become easily configured recipes.

A Policy GUI is now a possibility with simple manipulation of server side JSON.
Data management should be data-centric and metadata driven.

Future-proof automated data management requires open formats and open source.