Fermilab HEPCloud Facility

Decision Engine Design

Version 1.0
March 6, 2017

CS-doc-XXXX

PREPARED BY:

Mine Altunay, Jim Kowalkowski, Parag Mhashilkar, Steven Timm, Anthony Tiradani

Revision Log

Revision	Description	Effective Date
0.5	Rough Draft	03/06/2017
1.0	Initial Version	04/07/2017

Table of Contents

1. Introduction

1.1 Document Purpose

The Decision Engine is a critical component of the HEP Cloud Facility. It provides the functionality of resource scheduling for disparate resource providers, including those which may have a cost or a restricted allocation of cycles. Along with the architecture, design, and requirements for the Decision Engine, this document will provide the rationale and explanations for various design decisions. In some cases, requirements and interfaces for a limited subset of external services will be included in this document.

This document is intended to be a high level design. The design represented in this document is not complete and does not break everything down in detail. The class structures and pseudo-code exist for example purposes to illustrate desired behaviors, and as such, should not be taken literally. The protocols and behaviors are the important items to take from this document. This project is still in prototyping mode so flaws and inconsistencies may exist and should be noted and treated as failures.

1.2 Scope

For the most part, this document will be limited in scope to the architecture, design, and requirements specific to the Decision Engine. However, in certain cases the document may extend itself to include interactions between the Decision Engine and external services. The included interactions will be used to clarify functionality, to provide expected interfaces and requirements to external services, and/or to set boundaries on the responsibilities of the Decision Engine.

1.3 Rationale

Commercial Cloud and allocation-based HPC resources both have explicit costs that must be considered when deciding to provision these resources. The Decision Engine incorporates these costs, quantity, and capability requirements in the decision making process. This allows Fermilab to provision these new kinds of computing resources in a more efficient and cost-effective way, incorporating “elasticity”. The facility responds to demand peaks without local overprovisioning, using a more cost-effective mix of local and remote resources.

The Decision Engine is the HEP Cloud subsystem that executes administrator- and management-defined policies to create resource scheduling requests on behalf of the Facility. It is responsible for ensuring that policies are executed in a reliable, traceable and consistent manner. The policies that are executed result in resource requests on behalf of the facility, and ensure that those requests are to resource providers that match the job requirements. The goal is to minimize the execution time of the project subject to the constraint of available funding.

1.4 Terminology

Term	Definition
Decision Engine	The Decision Engine is the HEP Cloud subsystem that executes administrator- and management-defined policies to create resource scheduling requests on behalf of the Facility.
Decision Engine Framework	The parts of the Decision Engine which define and configure Decision Channels.
Decision Engine Standard Library	The Decision Engine Standard Library contains all the modules, functions, logic statements, and classes that are used to build Decision Channels.
Decision Engine Policy	A Decision Engine Policy is a configuration which makes use of decisions made by one or more Decision Channels. A policy can be viewed as a high level decision algorithm that aggregates multiple decision channels together to arrive at a resource provisioning request.
Decision Channel	A configuration that ties together various modules and business rules to form a workflow to make a decision. Decision channels are dedicated to one specific decision point, such as the optimal spot price for a given region, zone, and virtual machine type in Amazon. Typically, a channel does not implement an entire policy by itself. Channels are broken up into four discrete functions. Sources obtain data, Transforms feed the data to algorithms which transform the data into new data products, Logic Engines which provide decisions based on the transformed data, and Publishers which push the decision to external systems.
Data Space	The Data Space is a time sensitive data store that contains the complete state of the Decision Engine. The Data Space manages DataBlocks on behalf of the Decision Channels.
DataBlock	The DataBlocks contain all data gathered by the modules that make up a Decision Channel and all data required for traceability, debugging, and logging. There is a one-to-one relationship between a DataBlock and a Decision Channel.
Decision Engine Module	A module of code that runs within the Decision Engine Framework. This code fulfills one of the four functions for a Decision Engine Policy: Source, Transform, Logic Engine, Publisher. **Source:** Sources are the interface for the Decision Engine

		to query external systems for data. Sources are responsible for querying, formatting, and registering data within the Data Store.
	Source Proxy:	A special Source that is used to retrieve data from a DataBlock belonging to a different Decision Channel.
	Transform:	Transforms contain algorithms and heavy lifting code that is responsible for consuming and correlating the data from produced by sources.
	Logic Engine:	Logic Engines evaluate logic expressions that are derived from the data provided by the Transforms to ultimately trigger Publishers.
	Publisher:	Publishers are responsible for formatting, and pushing decisions via the proper APIs to an external system or the Data Store.
Decision Document	**Administrative Parameters:**	Parameters set by management that provide limits or boundary conditions for the policies. An example of a parameter might be spending rates for cloud resources. These may be stored in the Parameter Datastore.
	Business Rules:	Business rules tie together specific Sources, Transforms, Logic Engines, and Publishers into a Decision Channel. Additionally, they provide logic statements that the Transforms and Logic Engines incorporate into their algorithms. This way, algorithms can be tweaked without having to release new code.
Decision Cycle:	Triggered by the state change of a Source caused by new data being inserted, this cycle contains the execution of the Transforms, Logic Engines, and Publishers.	
System-wide Configuration	Information that lets the decision engine framework know where user defined modules that are dynamically loaded based on configuration options are, and system-level parameters that	

	control the basic functioning of the Decision Engine Framework.
Resource Provider	An organization that possesses resources and wishes to make them accessible to the Fermilab HEP Cloud Facility.
Provisioner	The Provisioner accepts requests from the Decision Engine and translates those requests into requests that Remote Sites understand. The provisioner is responsible for managing the lifecycle of the resources. Fermilab HEPCloud Instance will be using the glideinWMS Factory as its provisioner.
Compute Resource	A provisioned resource such as a physical machine, a virtual machine, or a Container where a job can run.
Resource Request States	"Normalized" set of states for a resource request so that policies do not have to be aware of different provisioner states.

Pending	A resource request which has been made, but not fulfilled yet by the resource provider.
Active	A resource request that is in the fulfillment process, but has not been completely fulfilled yet.
Fulfilled	A resource request that has been satisfied by the resource provider and is in use.
Complete	A resource request that was satisfied by the resource provider and is no longer in use.
Failed	A resource request that was not fulfilled. This could be due to the resource provider denying the request or due to an error condition.

HEPCloud Resource	A representation of a Site/Resource provider combination that the HEPCloud Provisioner understands.
HEPCloud Resource States	

Available	The HEPCloud Resource is available for use
Offline	The HEPCloud Resource is unavailable. No requests will be made.
Degraded	Capacity is at or near limit.

	Error	The Provisioner is receiving known understand error states from the resource provider.
	Unknown	HEP Cloud Resource status has not been received within some configured timeframe.

2. Overview

The Decision Engine provides a framework to execute one or more Decision Engine Policies. A policy consists of one or more Decision Channels. Decision Channels are "Source -> Transform -> Logic Engine -> Publisher" workflows dedicated to making a specific decision. A channel may be created to load Spot Pricing data from AWS or to determine the health of a particular Resource Provider. Policies use the same "Source -> Transform -> Logic Engine -> Publisher" workflow to combine these channel decisions into high level decisions such as a provisioning request to a particular Resource Provider.

Large decisions such as deciding to send workflows to the cloud require many smaller intermediate decision results as inputs. The Decision Engine breaks down the decision making process into Decision Channels. Each Decision Channel is composed of modules that correspond to a "Source -> Transform -> Logic Engine -> Publisher" model. The Decision Channel triggers the execution of Sources. Upon completing of a Source the Decision Channel will trigger a "Decision Cycle". The decision cycle is composed of one or more Transforms, a Logic Engine which could cascade to multiple Logic Engines, then finally one or more Publishers. Sources are responsible for information acquisition from entities external to the Decision Channel. These external entities may be DataBlocks from other Decision Channels, or they may be systems external to the Decision Engine itself. Transforms contain the algorithms for specific decision tasks. Logic Engines operate on boolean facts derived from the data provided by the Transforms to follow rules that express a decision. Publishers are responsible for pushing the resulting decision information out to external systems. At the highest level, there will be a publisher responsible for pushing out specific requests to the Provisioner.

Base classes that define the interfaces and minimal functionality that a module must implement are contained by the Core Framework. The standard library contains a set of approved, tested, and released modules which adhere to the interfaces defined by the core framework. These modules may be contributed from interested stakeholders. User libraries contain modules that are not part of the released standard library nor are they part of the core framework. These modules must use the provided interface standards from the core framework. The modules in the user libraries may provide functionality for specific types of decisions, or they could simply be modules that are being tested prior to release.

The Decision Engine Factory/Builder takes a combination of module parameters and the location of the module code within the libraries to create a runnable instance of a Decision Channel. The module parameters provide all information to bootstrap a module instance.

Additional "tweakable" parameters are defined that allow management input into the decision making process within fixed boundaries. An example of a "tweakable" parameter might be the overall budget allocated to Cloud spending. Another might be the rate at which the Facility is allowed to spend down the given budget.

All interactions between different Modules and even other Decision Channels is mediated through the use of DataBlocks. All state and all relevant data is kept within a DataBlock dedicated to the Decision Channel. Sources acquire data from from external systems, format or derive the required information from the acquired data, and store the results in the DataBlock. Transforms and Logic Engines only know how to query the DataBlock for required information. The resulting Decisions are then stored in the DataBlock as well. Publishers query the DataBlock for the decisions that need to be pushed out to external systems and use the external system interfaces to communicate the decisions. In addition to the data acquisition and communication processes, each Module regularly updates the DataBlock with its state. In this manner, everything about the Data Channel is encapsulated in the DataBlock. Each decision can be traced because all inputs, all state, and all outputs are kept.

Operators interact with the Decision Engine from two perspectives. The system Decision Engine will provide the standard tools that required for service management. These are the primary tools that system administrators will interact with. Additional tools will be provided that will:

- Allow dynamic updates to the "tweakable" parameters
- Create new Decision Channels
- Take Decision Channels off-line
- Query the state of Decision Channels
- Provide an API for monitoring hooks

3. Requirements and Constraints

3.1 Requirements

- Operational Requirements
    - Must be highly available (HA)
    - Must scale horizontally
- Compatible with the Provisioner Protocol
    - Facility expansion
        - Understands protocol used by a provisioner to describe a HEP Cloud resource. For the FNAL deployment, this will be the glideinWMS Factory.
        - Supports protocol used by the provisioner to request compute resources in one or more HEP Cloud resources
        - Supports protocol used by the provisioner to terminate compute resources provisioned in one or more HEP Cloud resources

        - Supports protocol used by the provisioner to cancel compute resource requests in one or more HEP Cloud resources
        - Understands the protocol used by the provisioner to describe status of the resource requests
    - Operations
        - Supports custom validation scripts
        - Supports custom periodic scripts
    - Security
        - Can manage submission credentials
        - Can manage credentials required by provisioned resources to authenticate with the pool
        - Must securely pass credentials to the provisioner
- Data Space
    - Must be able to save a configurable amount of history of each DataBlock for audit purposes, configurable on a DataBlock by DataBlock basis.
- Source
    - Must query a given information source to get its data
    - Must validate the data retrieved from an information source
    - Must timestamp the data retrieved from an information source
    - Must store the validated and timestamped data into a DataBlock
- Publisher
    - Must retrieve the data to be published from the DataBlock
    - Must be able to deal with stale data
    - Must deal with error conditions gracefully
    - Must publish the data to a given endpoint
- Transform
    - Must provide audit logs for financial decisions wherever applicable
    - Must provide audit logs for debugging purposes
    - Must provide audit logs detailing decisions made with regards to facility expansion and the inputs that lead to each decision
    - Must be flexible enough to allow for input of arbitrary mathematical expressions including but not limited to cost considerations, performance measurements, time deadlines, and budget compliance.
    - Must be able to deal with stale data
    - Must deal with error conditions gracefully
- Parameter Datastore
    - Must provide storage for static or partially static information
    - Must provide means to store information
    - Must provide data access to a source

3.2 Constraints

The Decision Engine Policies will interface with many external systems such as Experiment specific Data Movement systems. The Decision Engine will not exercise any API call other than specifically exposed APIs for these systems. The Decision Engine will not duplicate or replace any component or function of the external systems.

Specifically, the Decision Engine does not:

- Perform authentication and/or authorization
- Perform Data Movement
- Execute Jobs
- Match individual jobs to individual resources. This is the function of the HTCondor Negotiator.

There are several technology choices that have already been made for the HEPCloud Facility. These choices imply certain protocols and APIs. The following are choices made prior to the design of the Decision Engine:

- Provisioner selected for Fermilab HEPCloud Instance is the glideinWMS Factory. This component communicates via HTCondor classads.
- The Fermilab HEPCloud Facility batch computing service is HTCondor. There is a command-line tool or a python API available to query for jobs or the state of the batch system.
- The Fermilab HEPCloud Facility batch computing service is HTCondor. Is is assumed that provisioned resources will be delivered in the form of a container (of some type, e.g. glidein pilot, shifter container, vm, etc.) that runs an HTCondor StartD daemon which joins the Facility HTCondor pool.
- The Fermilab HEPCloud Facility already utilizes monitoring suites provided by the Landscape project and Check_MK. These tools have their own custom APIs
- GRACC is the accounting system used by the Fermilab HEPCloud Facility. There is a Restful API available.

3.3 The Major Inputs and Output

The proposed Class Diagram of the Decision Engine Framework specifies the types of inputs and output that each component of the framework. Most of the interaction with other systems of HEPCloud and the external providers is done not by the Decision Engine Framework itself but by the Source and Publisher modules that it runs. The candidate inputs and outputs are based on examples of modules we intend to run.

Component	Input(s)	Output(s)
Source	Information from external sources or from other HEPCloud subsystems which include but are not limited to external cloud providers, billing system, local batch clusters, and facility interfaces.	Formatted, timestamped data to DataBlock
Proxy Source	Information from another DataBlock	Information to local DataBlock

Transform	Information from local DataBlock	Information to local DataBlock
Logic Engine	Information from local DataBlock	Information to local DataBlock
Publisher	Info from local DataBlock	Info to other subsystems including (but not limited to) Provisioner, Monitoring, Facility Interface.
Task Manager	A list of Sources, Transforms, Logic Engines, and Publishers that form each Decision Channel	Status of whether the Decision Channel was executed successfully
Factory	One or more Decision Documents containing Business Rules and Configuration	A machine-readable description of routines to be executed by the Task Manager and parameters.

3.4 Behavioral requirements (use cases)

These are three generic tasks each of which corresponds to a Decision Channel above. They are abstracted from the use case in appendix A. They cover the major tasks.

Task	Resource Identification
Level	Continuously_recurring
Goal	Find all available resources that match the current job requests
Actor	Daemon
Trigger	Continuously at short interval
Preconditions	System configuration information, budget information
Post-conditions	List of potential resources to submit.
Description	Find all resources where the user is authorized to run and has budget.
Nonstandard Flow	Inputs out of date, or calls to external resources fail.
Comments	Used as input to resource selection.

Task	Resource Selection

Level	Continuously_recurring
Goal	Rank available resource based on occupancy and price-performance
Actor	Daemon
Trigger	Continuously at short interval
Preconditions	List of available resources
Post-conditions	List of potential resources to submit ranked by price-performance
Description	Find all resources where the user is authorized to run and has budget.
Nonstandard Flow	Occupancy or price information is stale, performance info or resource list is missing.
Comments	Used as input to resource request

Task	Resource Request
Level	Continuously_recurring
Goal	Request resources from optimum resources
Actor	Daemon
Trigger	Continuously at short interval
Preconditions	Priority list of resources ranked by price performance
Post-conditions	Request for how many of each type of resource
Description	Request an appropriate number of each type of resource
Nonstandard Flow	Burn rate info is stale, resource list is stale
Comments	Final output goes to the provisioner.

4. Architectural Overview

This section describes the architecture of the Decision Engine, its components and their relationship to other components within the Decision Engine and external systems.

4.1 Decision Engine Overview

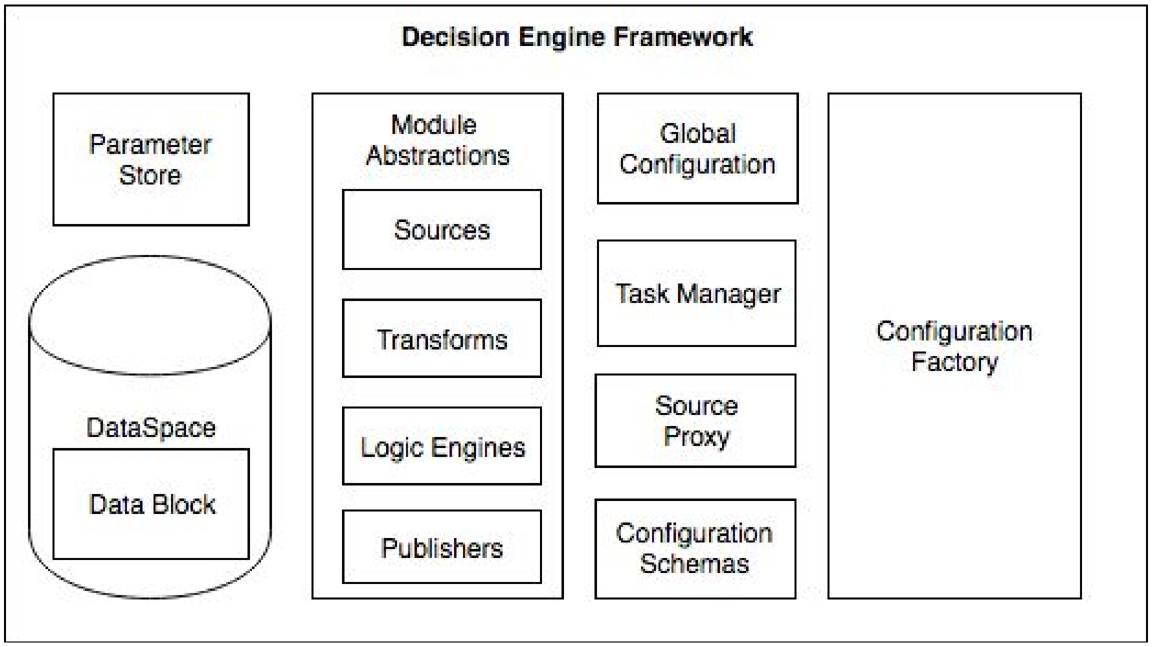


Figure 1: Decision Engine Design Overview

The core purpose of the Decision Engine is to reliably process Decision Channel workflows. The Decision Engine accomplishes this by utilizing a) a time sensitive Data Space which expires data when it reaches an age threshold and b) a standard library and Modules library to provide the necessary functions which are used by business rules that go into a Decision Channel. Additionally, the Decision Engine contains a Configuration Factory and a Scheduler. The Configuration Factory compiles the business rules into a Decision Channel which is run by the Task Manager. The Scheduler is responsible for triggering the execution of Decision Channels.

4.2 Decision Channel vs Policy

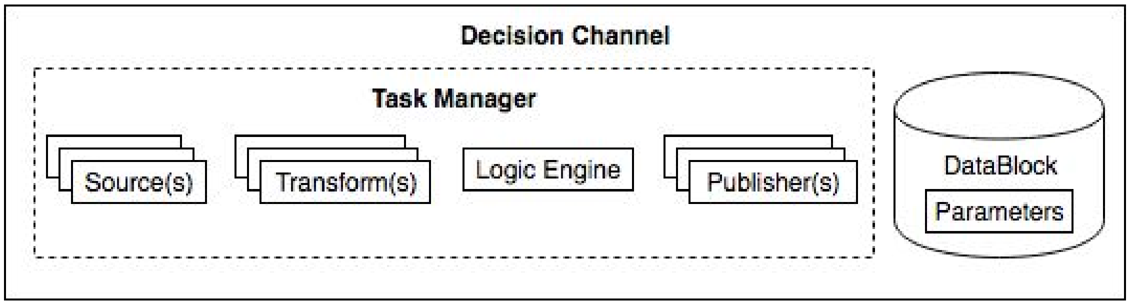


Figure 2: Decision Channel

A Decision Channel is a conceptual construct used to define a discrete workflow that is executed by the Decision Engine. A Decision Channel minimally consists of a Source, a Transform, a Logic Engine and a Publisher. The Decision Channel may one or more Sources, Transforms, and Publishers. The Decision Channel is configured with only one Logic Engine, but that Logic Engine may cascade to new Logic Engines.

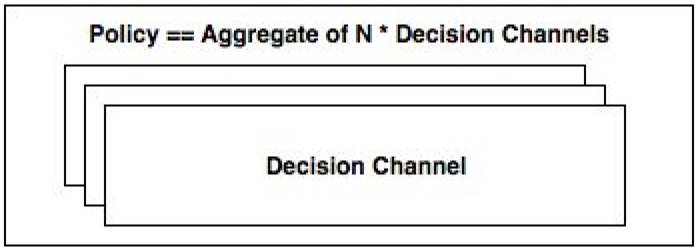


Figure 3: Decision Engine Policy

Conceptually, a Policy is an aggregation of multiple Decision Channels. However, the Policy is implemented in the same "Source -> Transform -> Logic Engine -> Publisher" model that makes up a Decision Channel. The Sources for a Policy are the published results of a Decision Channel. Additionally, the Logic Engine rules may be more complicated than a "lower level" Decision Channel.

5. The Core Components

5.1 Data Space

The Data Space is a time sensitive data store that contains the complete state of the Decision Engine. All data management, archiving, and time management is accomplished through the management of self-contained private databases called DataBlocks. These DataBlocks are assigned to Decision Channels and contain all data gathered by Sources, all data generated by Transforms, all results from Logic Engines, and all data required for traceability, debugging, and logging.

Interactions between modules are only permitted through data stored in a DataBlock. The DataBlock defines the data access protocol, the allowable data types that can be made available, and the metadata that must be present for each stored item. The DataSpace manages the versioning and archiving of DataBlocks. When a Decision Channel starts a decision cycle, the complete DataBlock is copied for archiving. The decision cycle always ends up using t-1 version, while the Sources always use t=0 where t=0 is always the current version. This way future analysis tools can walk backwards through the cycles and and analyze exactly what occurred during each cycle. When a copy for archiving operation starts, any incoming put operations must be queued up until the copy is finished. This ensures consistency of data.

There is no fixed schema for the data that will be inserted into the DataBlock. Therefore, the DataSpace must provide a set of tools for managing a user-defined data model. This includes a schema repository which can be used to store the specific schemas a particular

Source might produce. The pointer to the schema is the only dependent data that will be propagated along with any data products.

Additionally, the DataBlock must provide tools for defining data retention policies for DataBlocks. For example, if a Decision Channel is removed from the system, there must be tools available to set how long a DataBlock will be retained for archival or forensics purposes.

The DataSpace must also provide query tools that allow administrators to query the state and content of DataBlocks, update DataBlocks, and remove DataBlocks. These tools can be command line tools or APIs.

Data access protocol:

```
      DataBlock {
            Value get(Key) # user
            put(Key, Value, Header, ...) # implies a transaction,
e.g., no versioning may occur until all puts are complete, no
puts may start until versioning completes, etc.
            Header get_header(Key) # admin
            Metadata get_metadata(Key) # admin
      }
```

Where:
Key = a unique string identifying this piece of data
Value = A JSON object (Python Dictionary) containing a required descriptive Header.

```
      Header {
            DataBlockID
            CreateTime # Timestamp
            ExpirationTime # Timestamp
            ScheduledCreateTime # Timestamp
            Creator # name
            SchemaID # An identifier that points to a schema definition
            # any other fixed information required for system management and tracking
      }

      Metadata {
            DataBlockID
            # might be internally generated so the user never sees this
            State # system defined assessment of the state of this information
            GenerationID # which write to this name is this one?
            GenerationTimestamp # ?
            MissedUpdateCount # ?
            # any other information required for system management and tracking
      }

      DataSpace {
            # example set of functions that all DataBlocks support
            checkpoint(a_datablock_id) # cause DataBlock to be archived as is
            mark_expired(current_timestamp)
            mark_demented(which_values)
```

```
        # any other functions required for management of the system
    }
```

Notes:

- The Value must be read-only once placed into a DataBlock
- Any "updates" result in a new record inserted into the DataBlock
- Metadata manipulations (updates) must be tracked
- The expectations on a completed "put" operation is:
    - A data product object is generated
    - A header object is generated
    - A metadata object is generated
    - The DataBlock contains the data, header, and metadata generated by the put
    - The put function will return these objects
- Once a Decision Cycle is triggered, the DataBlock will be completely copied to a new version. The original DataBlock is now version t-1. The new DataBlock is version t=0. Sources always insert into version t=0. The copy procedure puts the t-1 DataBlock in the "BeingProcessed" state shown in Figure 4. Once the Decision Cycle is finished, a metadata update is performed and the final state, "archived", of the t-1 DataBlock is triggered.

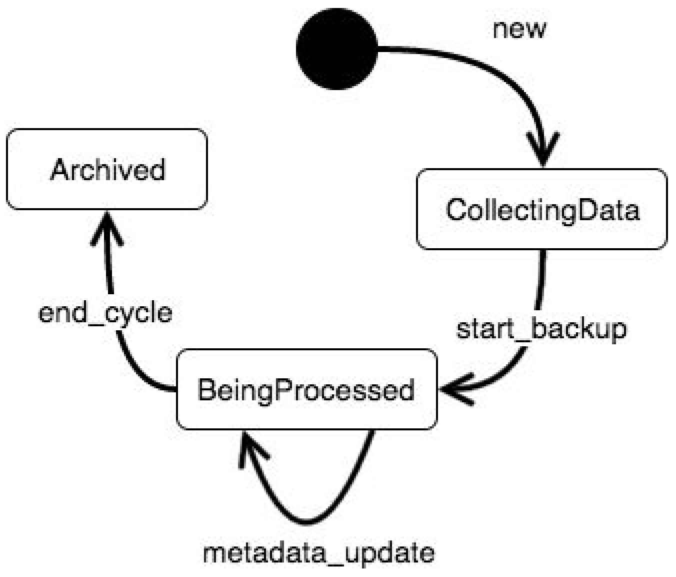


Figure 4: Data Space State Diagram - Use of a DataBlock within the steady state of the TaskManager

5.2 Modules

The framework defines the concept of a Module. Modules are objects that are "plugged" into Decision Channels to perform particular tasks. These encapsulate algorithms, decision actions, and the read/writing of data to and from external sources. Modules have the following features:

- Use DataBlocks to operate on data that is active within the system
- Are written (coded) to the standard protocol defined by the framework
- Are runtime configurable: they appear in a running application because a configuration file indicates that they are needed. The configuration file also supplies runtime parameters for the module to operate within a given Decision Channel
- Exist in libraries that are accessible at runtime to the application.
- Are constructed as objects within the running framework application. Their lifetime is controlled by the framework.
- Are contributions to the core application. This means they are only known to the framework via the standard protocol. They are expected to come from either the standard library (defined in the next section), or from private user libraries.
- Ability to specify default values for parameters such as execution times

The core framework of the decision engine defines what it means to be a module: how a module interacts with data within the system, when a module is scheduled to do its work, and how a module is configured i.e. how it comes into existence. Modules come in four flavors, depending on the service they provide: Sources, Transforms, Logic Engines, and Publishers. These interfaces define the protocol.

A Task Manager will instantiate the modules that it is configured to manage. Once instantiated, a module enters “idle” state. When the Task Manager calls the action function for the module type (defined below), the module moves into “working” state. Upon appropriate updates to the DataBlock and completion of the action function, the module moves back to “idle” state. The module instance ends when the Task Manager is stopped or the Task Manager configuration is changed.

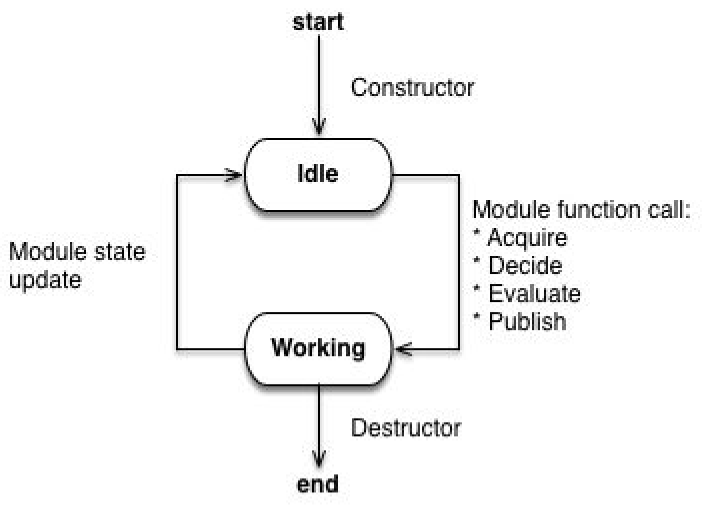


Figure 5: Module States

All modules must implement the follow interfaces depending on the module type:

```
    Module {
        # my_data_block: pointer to the Task Manager’s
        #    DataBlock.  (Required)
        # set_of_parameters: any and all parameters required
        #    for the module to function.  These come from the
        #    configuration
        constructor(set_of_parameters, my_data_block)
    }
    Source (derived from Module) {
        # name_schema_id_list: a list of dictionaries
containing
        #    the data product name and a pointer to a schema
        produces(name_schema_id_list)


        # acquire:  The action function for a source.  Will
        #    retrieve data from external sources and issue a
        #    DataBlock “put” transaction.
        acquire()
    }
    Transform (derived from Module) {
        # name_list:  A list of the data product names that
    the
             Transform will consume
        consumes(name_list)


        # name_schema_id_list: a list of dictionaries
containing
        #    the data product name and a pointer to a schema
        produces(name_schema_id_list)


        # decide:  The action function for a Transform.  Will
        #    retrieve from the DataBlock the data products
        #    listed by consumes and performs algorithmic
        #    operation on them.  The Transform will issue a
        #    DataBlock “put” transaction for each of the data
        #    products promised in the produces list
        bool decide()
    }
    LogicEngine (derived from Module) {
         Bool evaluate()
    }
    Publisher (derived from Module) {
         consumes(name_list)
         publish()
    }
```

All module interactions will be handled through the DataBlock. The sources are intended to be scheduled periodically by the Task Manager. The Transforms, Logic Engines, and Publishers make up a "decision cycle" A decision cycle is a scheduling concept that the Task Manager implements and is described in detail below. Transforms always run before Logic Engines, and Logic Engines trigger Publishers. The completion of a Source triggers the decision cycle.

5.2.1 Sources

As the name suggests, Source is a module that is responsible for communicating with external systems via the native APIs to gather data that acts as input to the system. For example, to gather data from AWS, the Source would utilize the EC2 Query API to communicate with AWS to acquire required information like running VMs, billing and accounting information etc. Each Source is independent from other Sources and is responsible for gathering its own data. Every Source is scheduled to run periodically by the Task Manager based on it's configuration. Data gathered by a Source is stored into the DataBlock associated with the Task Manager along with any manifest. When a Source stores new information into the DataBlock, this represents change of state in the TaskManager and it will trigger a decision cycle. A Source can be configured with a timeout. If a source does not return within the time period specified by the timeout, an error condition should be generated.

5.2.2 Transforms

Just like a Source, Transform is a module that produces data that is stored in the DataBlock associated with the TaskManager. However, unlike Source, Transform does not interact with external systems but instead consumes the data produced by one or more Sources. Transforms will perform algorithmic manipulation of the data and store the results in the DataBlock for use in the Logic Engines and Publishers. A Transform must produce the data products specified during its instantiation.

Upon a state change of a source, i.e. new data, the Task Manager will trigger a decision cycle. The first stage of the decision cycle is to run all configured Transforms.

5.2.3 Logic Engines

Logic Engine operates on the data produced by one or more Transforms or Sources. Facts are expressions which evaluate to a boolean "true" or "false and depend on the data products in the DataBlocks. The logic rules are constructed using the facts to make decisions on how to fulfill the policy being expressed by the Decision Channel. The Logic Engine will process all Facts to obtain the boolean values, then process all rules to obtain the decision. A Rule consists of a condition made up of facts and boolean operators and an action. Actions are triggered when the rule evaluates to boolean "True". Example:

```
Available Facts:
    # value is determined by configuration and stored in the
    # DataBlock as parameters
    cloud_enabled


    # budget_available is a value stored in the DataBlock by a
    # Source estimated_cost is a calculation done by a
Transform,
    # the value is also stored in the DataBlock
    cost_ok = (budget_available - estimated_cost) > 0


Rule:
    cloud_enabled && cost_ok -> request_cloud_resources
```

There are two distinct types of action that a Logic Engine executes when a rule is triggered. A publish-action triggers Publishers and logic-actions trigger subsequent Logic Engines. Logic Engines are said to be cascaded when a logic-action triggers another Logic Engine. This allows for the compartmentalization of rules into discrete blocks. An action will store the results and/or state of the action in the DataBlock. Facts can reference actions as booleans. By default an action referenced in a Fact returns a False value. This indicates that an action has not been triggered. If a return value of True indicates that the action was triggered.

When a Decision Cycle is triggered, the Task Manager runs all the Transforms prior to invoking the Logic Engine. Logic Engines never trigger any Transforms and all Transforms need to happen before the Logic Engine is triggered. The Logic Engine then processes all Facts and applies Rules to produce result that is stored in the DataBlock. There is no inherent ordering implied in the Rules. The data product in the DataBlock can in turn be used by another Logic Engine when multiple Logic Engines are cascaded or the data is used by the publisher to publish information to external systems.

5.2.4 Publishers

A Publisher is the inverse of a Source. Publishers read data products from the DataBlock and format them according to the formats that external systems expect. Publishers use the remotely exposed APIs to push the data to the external systems. Publishers must publish all the data products that they promise or generate an error condition. Publishers are triggered by Logic Engine Actions.

5.3 Task Manager

A Task Manager manages the execution and life cycles of a configuration of Modules and Logic Engines. The Task Manager is assigned a unique DataBlock by the Configuration Factory which holds all the current state which is operated on by the Modules and Logic Engines managed by the Task Manager.

The Task Manager maintains a source -> transform -> logic engine -> publish workflow. To accomplish the workflow, the Task Manager schedules Sources to run periodically. The period of execution is determined by the configuration used to instantiate the Task Manager instance. Sources may be executed in parallel. Once a Source changes state, or in other words, updates the value of advertised data to a new, different value, the Task Manager triggers the rest of the workflow. All configured Transforms are executed before the Logic Engine(s). As with Sources, Transforms may be executed in parallel. In this manner, all required data is available in the DataBlock prior to making any decisions. Logic Engines are are executed next. All Logic Engines, embedded or not, have access to the same DataBlock. The Logic Engine stage begins with one and only one Logic Engine. Logic Engine actions may trigger subsequent Logic Engines. This allows Rule and Fact writers to compartmentalize specific rules. At some point the Logic Engine rules will lead to a decision. The Logic Engine decision action will trigger a configured set of Publishers to run. This document does not describe the mechanism by which Publishers are executed. This is deliberately left to the discretion of the implementer.

The "Decision Cycle" is defined as the execution of Transforms, Logic Engines, and Publishers. A Decision Cycle is triggered by a state change in any of the configured sources. However, there must be a configurable minimum delay between Decision Cycles to avoid decision collisions. It also prevents over scheduling due to jitters in the system. This configuration item should provide a sensible default value. The start of a Decision Cycle should cause the TaskManager to trigger the DataBlock to initiate a version (history) change. The Decision Cycle will then use the t-1 version. All Sources use the t-0 version of the DataBlock.

The TaskManager must guarantee reproducibility of results. Given the same module scheduling parameters and a replaying of the same data used as input earlier to the system, the system must make the same decisions, and form the same intermediate results.

A Task Manager has the following states:

- **Boot**: This state begins when a Task Manager is started. A Task Manager is in boot state until all configured sources have populated the DataBlock with the data products they guarantee and all Transforms have successfully executed at least once.
- **Steady State**: This is the main operational state of the Task Manager. If an unrecoverable error is encountered, the Task Manager is moved to the "Offline" state. An administrator may shutdown a Task Manager which will move the Task Manager into the Shutdown state.
- **Offline State**: This state occurs when an unrecoverable error is encountered that needs administrator action, or an administrator manually takes an Task Manager offline. Once an administrator fixes an error state or chooses to manually restart an

offline Task Manager, the Task Manager moves to the Boot state. When a Task Manager enters the offline state, all data products in the DataBlock are marked as inaccessible.

- **Shutdown State**: A Task Manager in Shutdown state will complete a Decision Cycle if one is ongoing, then trigger an archive operation on the DataBlock to preserve the history of the Task Manager. All Source Execution will cease immediate to ensure a new Decision Cycle will not be triggered. Just prior exiting, the Task Manager trigger a new version in the DataBlock and then take the DataBlock offline. At the end of the Shutdown state the Task Manager will exit.

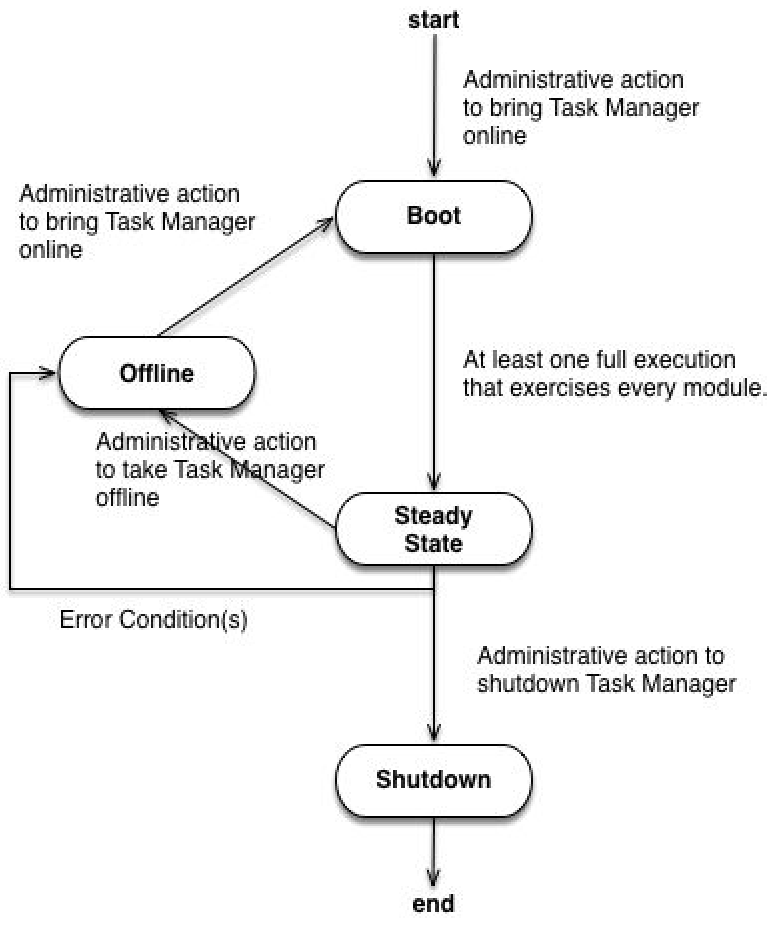

start
Administrative action to bring Task Manager online
Boot
Administrative action to bring Task Manager online
Offline
At least one full execution that exercises every module.
Administrative action to take Task Manager offline
Steady State
Error Condition(s)
Administrative action to shutdown Task Manager
Shutdown
end

Figure 6: Task Manager State

Notes:

A Task Manager that is in any state other than "Steady State" is not expected to provide valid, reliable results. Another Task Manager may have a Source Proxy that reads data from this Task Manager. Therefore, the Task Manager must have access to the headers for all data products that its configured Sources promise to produce. The Task Manager will manage state for these data products such that data in the Task Manager's DataBlock will be marked as inaccessible to Source Proxies when the Task Manager is not in "Steady State".

A Source Proxy may not read data from a foreign Task Manager's DataBlock if the foreign Task Manager is not in Steady State. Sources and Source Proxies define a valid lifetime for data that they produce. If a Task Manager that produces data used by another Task Manager is not in Steady State, the dependent Task Manager may continue to execute on the data it has until it is expired. Once the data is expired, the Task Manager must report an error state and move to an offline state.

If a Task Manager is Shutdown or effectively permanently disabled, all dependent Task Managers will remain in an offline state until the dependency is fixed. If an offline Task Manager has not been manually taken offline, i.e. it went offline due to a dependency issue, that Task Manager should periodically check to see if the issue has been addressed and automatically cycle to boot state when it has.

Consider the following scenario:

```
(source1 <put1>)(start decision cycle 1 <do versioning> do
work)
    (source2 active <put2>)
                                                (source3 active <put3>)
                                 (source4 active <put4>)
```

Put operations put3 and put4 will be delayed until the versioning operation is complete. Put2 will be added to the DataBlock prior to the versioning operation, however, any trigger that might have occurred due to put2 is considered invalid and no trigger should happen since this data is being considered by decision cycle 1.

During the Decision Cycle, the Task Manager must manage dependencies between transforms. If transform A requires data produced by transform B, then the Task Manager must manage the execution of the transforms such that transform B finishes its execution before transform A is executed.

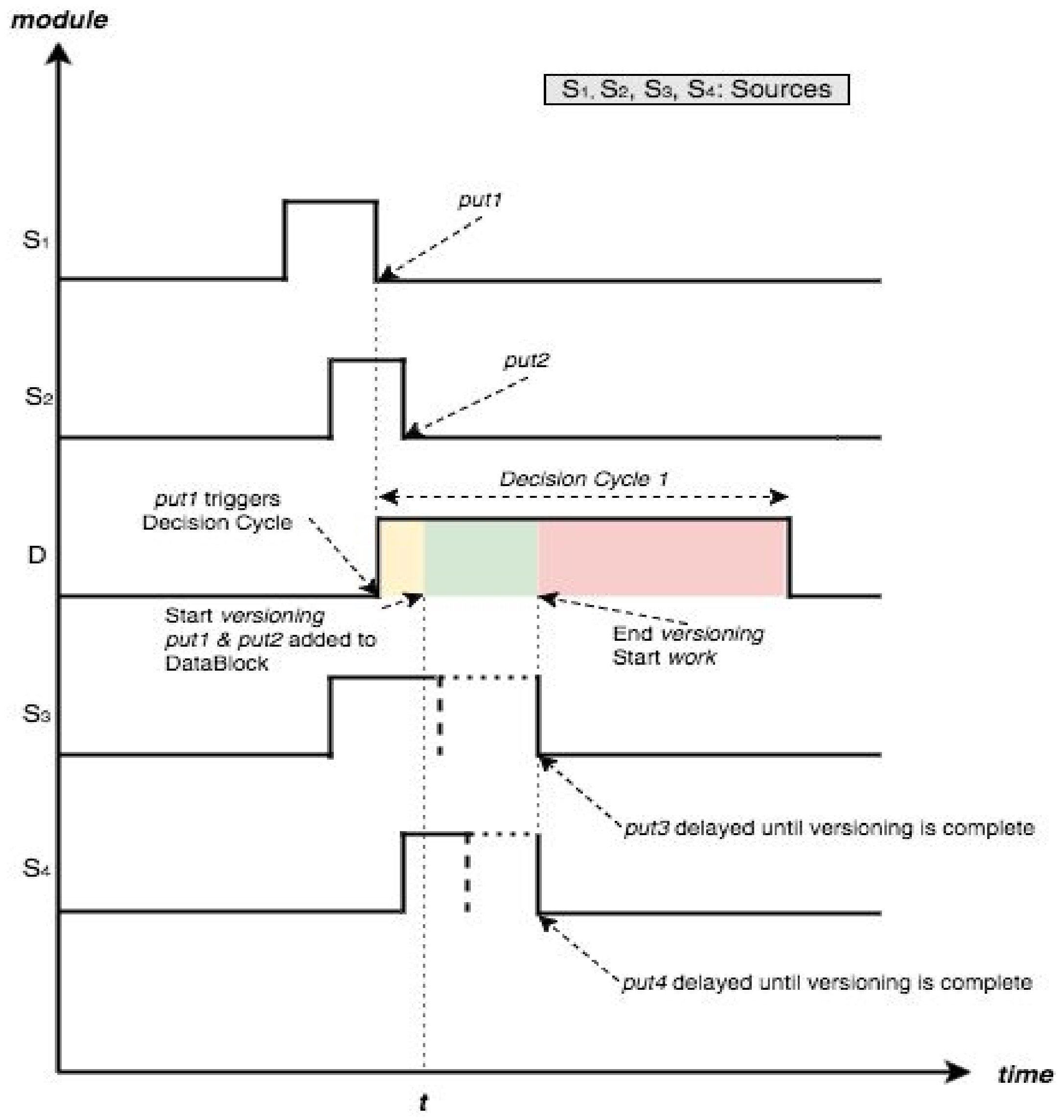

module
S1, S2, S3, S4: Sources
S1
put1
S2
put2
D
Decision Cycle 1
put1 triggers Decision Cycle
Start versioning put1 & put2 added to DataBlock
End versioning Start work
S3
put3 delayed until versioning is complete
S4
put4 delayed until versioning is complete
time
t

Figure N: Task Manager Timing Diagram

5.4 Configuration Management

Configuration management contains three important components necessary for constructing a consistent, running applications using dynamically loaded modules.

- Configuration document definitions and handling for describing the functionality that will be available in the running application and all the runtime settable parameters
- A Factory for loading and constructing Module objects in the running application

- A Builder that uses Configuration objects and the factory to construct Task Managers

Module configuration appears in two different contexts: Facility and Operator. The basic requirement is that Facility-configured modules be capable of overriding the decisions of Operator-configured modules.

Module configuration must permit three kinds of module parameters:

- Module-specific constants - parameters necessary to instantiate a module instance, where the values are the same for the duration of the module's lifetime.
- Module-specific adjustables - parameters that are permitted to change within pre-established bounds.
- Module management - parameters that must be specified for a module to run correctly within a Task Manager, but not specific to a given module. An example is the periodic scheduling value.

System-wide configuration must contain the following types of parameters:

- Workflow - decision channel definitions, modules that will be active within each decision channel, version information for the modules, scheduling and error handling policies and parameters, any required configurable ordering of work, connections between decision channels.
- Component - connection definitions for access to remote components such as the data space, pointers to credentials.

The Engine will provide a runtime environment for the configuration factory to operate within. The runtime environment supplies the search paths for the standard library, user libraries, the logic engine facts and rules, and the locations for the system-wide configurations.

The configuration factory translates a Decision Channel Document into a functional workflow definition and performs any required validation steps. The configuration factory also ensures that the information necessary for the Task Manager to schedule source execution is available to the instantiated Task Manager. The factory passes the workflow definition to a "builder" and triggers a build. The Factory will store history of configuration changes for traceability, auditing, and debugging purposes.

5.5 Decision Engine Service

The Decision Engine Service manages Task Managers and their states, the tweakable parameter connection, and connections (if any) to the DataSpace implementation. Additionally, the Decision Engine Service facilitates the validation of the tweakable parameters prior to handing them over to the Task Managers for delivery to the modules.

At a minimum, the Decision Engine Service should provide tools to the Operators and Facility that allow the following functionality:

- List Decision Channels
- Return Decision Channel status
- start/stop/load specified Decision Channels
- start/stop/reload all

- Global error handling
- Create an audit trail detailing the execution paths and state changes of a specified Decision Channel execution cycle

6. Standard Library

The Decision Engine Standard Library is a collection of implemented Modules that have been deemed useful for operating a HEPCloud/DecisionEngine facility. All the modules in this library conform to decision engine standard interfaces, and therefore can be found, loaded, and configured by the running decision engine system. The library also contains typical and standard configuration files for operating these modules, and all the validation tests that show that the modules are operating correctly.

The Standard Library is expected to contain versioned releases that include the module sources, module configurations, unittests, data schemas, and examples for the modules.

Example standard library structure:

```
/library base path/version1.1.1/lib
/library base path/version1.1.1/lib/unittests
/library base path/version1.1.1/config
/library base path/version1.1.1/config/schemas
/library base path/version1.1.1/examples

/library base path/version1.2.1/lib
/library base path/version1.2.1/lib/unittests
/library base path/version1.2.1/config
/library base path/version1.2.1/config/schemas
/library base path/version1.2.1/examples
```

Example logic engine facts and rules:

```
/logic base path/version2.1.1/facts
/logic base path/version2.1.1/rules
/logic base path/version2.1.1/examples

/logic base path/version3.1.1/facts
/logic base path/version3.1.1/rules
/logic base path/version3.1.1/examples
```

Modules are minimally validated via unittests that encompass all expected functionality. Regression and integration testing requires infrastructure outside the standard library, but should be considered best practices.

At a minimum, we expect the groups of modules to be found in this library:

- AWS access - all the modules necessary to monitor and control AWS using the AWS protocols and convert all the data to the DE standard formats
- Google access - all the modules that allow us to work with Google as a cloud provider
- NERSC - all the modules that are specific to working with NERSC
- Algorithms - standard modules for deciding how to best utilize resources for use access providers.
- Modules for dealing with laboratory, government, and division regulations and policies, especially monitoring and control of spending
- Modules for checking data integrity and reporting problems with data

We further expect the standard library to define a majority of the data model (the definitions of the data structures that will exist within DataBlocks), since the algorithms and modules for accessing providers and resources are contained here.

User supplied libraries must conform to the same standards as the Standard Libraries.

7. Runtime Environment

The runtime environment for the Decision Engine will provide the following:

- STD_LIB_PATH - Standard Library search path
- USER_LIB_PATH - User Library search path
- LOGIC_LIB_PATH - Logic Engine facts and rules search path
- PYTHON_PATH - complete python path to all python modules and packages
- CONFIG_PATH - path to the configuration files used to bootstrap a functioning engine, to create all the Decision Channel instances, and keep histories of configuration changes
- Any other required environment variables

The file system layout Config path

- ${config_base_path}/decisionengine - global configuration, this may include connection details for the Data Space, parameter store details,
- ${config_base_path}/decisionengine/config.d - this is where individual Decision Channel configs are stored
- ${config_base_path}/decisionengine/config.d/history - this is where the configuration factory stores the history of configuration changes

A Operator should be allowed to take individual Decision Channels on and offline at will. This should have no effect on the Engine as a whole. However, any Decision Channels that depend on the channel that was just taken offline, will also change state to offline when the dependent data expires. Decision Channels that have went offline due to dependency issues should automatically come back online when the dependency issues have been addressed.

When a Facility administrator chooses to stop the Engine, the expected behavior is that all Decision Channels are instructed to stop. The Decision Channels should halt all scheduling of Sources and allow any running sources to finish. Any running decision cycles should be allowed to complete within a configurable timeout. Decision Channels will exit after the Sources and decision cycles have finished. When all Decision Channels have exited, the Engine should clean up any configured external connections, such as connections to the Data Space. Finally, the Engine is free to exit.

8. Roles

There are two different categories of human roles that encompass the configuration and operation of the Decision Engine, Facility roles and Service Operator roles. There is nothing that prohibits a person from spanning the two categories or even multiple sub categories. The split is conceptual to allow tasks to be assigned according to skill sets.

Facility roles:

1. **System Administrators**: Administrators maintain the service as a whole. They guarantee that the accepted configurations are in place and provide first level troubleshooting of problems.
2. **Facility Operators**: Facility operators are responsible for ensuring that the accepted management policies are encoded correctly in the Decision Engine configurations. They have the ability to override any configuration or parameter that a Service Operator may have provided as input.

Service Operator roles:

1. **API Experts**: API experts develop Sources and Publishers since these modules interact with external systems via the native APIs.
2. **Algorithm experts**: Algorithm experts develop Transforms.
3. **Data Analyst**: Data Analysts develop Facts that are evaluated by the Logic Engines. Facts require a sound understanding of both the data products that are produced by the Transforms and the potential uses.
4. **Service Operator**: Service operators are the closest role the Decision Engine has to an end user. Service Operators create rule sets that use Facts to make decisions.

9. Testing Scenarios

DataBlock Testing:

- Create new DataBlock
- “Put” data, including metadata, header, and data product
- “Get” data, verify that the metadata, header, and data product are the same as the original put

- Update data, verify that the data product is versioned and metadata is updated appropriately
- Version DataBlock
    - Ensure puts are queued until version operation completes
    - Ensure queued puts only operate on DataBlock t-0
    - Ensure DataBlock t-1 is returned for use by the Decision Cycle

Logic Engine Testing:

- Test Facts return expected values
- Test rules use test facts to make decisions
    - Changes to the test facts cause appropriate changes to the decision

Decision Engine Testing:

- Successful installation and configuration of an “empty” engine
- Successful, repeatable, configuration of a sample decision channel
- Successful execution of sample decision engine
- Successfully replay a previous execution such that the replay reproduces the exact same results
- Successfully produce an audit trail showing the execution path and decisions made during the execution path

Future testing considerations:

- Financial tests: The engine must show that it correctly stops sending jobs to paid resources when there is no remaining budget or when the burn rate is too high. This can be accomplished by starting a test with budget and then adjusting the budget numbers downward.
- Load test: The engine must show that it can correctly perform under a load of 100,000 simultaneous jobs and execute a timely ramp-up and ramp-down.

10. Discussion

10.1 Architecture

One of the goals during the DE design was to make the architecture flexible and reusable to support a Prediction Engine in the future. The architecture should support the capability to use different implementations of key components. This enables different deployments of the Decision engine to be extensible as it can interface with different subsystems like different flavors of monitoring and information systems, job submissions and management systems, accounting systems, etc. New types of components in the decision channel (sources, transform, logic engines, and publishers) can be added to extend the functionality of the decision engine and interface it with different systems as required. This architecture philosophy makes core components of the decision engine agnostic to the changes in components of the decision channel.

10.2 Component Design

10.2.1 Decision Channel

The Configuration Factory takes as its input the information provided in the configuration schema to identify different sources, transforms, logic engines, and publishers required in a decision channel. It then uses the configuration to define their interaction with each other through DataBlock. The Task Manager uses a Scheduler to schedule sources periodically. When a source completes with a data state change, a Decision Cycle is triggered by the Task Manager. This cycle contains the transforms, logic engines, and publishers. A decision cycle will create and publish a decision.

Each Decision Channel will contain a single Task Manager responsible for working with the Scheduler to schedule and manage individual tasks. The architecture supports forming Decision Channels based on any arbitrary policies provided they can be supported by the underlying modules, components and plugins of the Decision Engine. As a recommended implementation, each Decision Channel will always act on behalf a single VO. However, a single VO may have more than one decision channels active at a time. This recommendation is an implementation recommendation that will allow for better control and management of the resource requests made by the DE.

10.2.2 DataBlock as the Communication Dashboard

Global namespaces are required to access DataBlocks universally. Data can be stored in distributed databases that provide High Availability. Data in the DataBlock can be unstructured from the Database point of view. However, modules and/or Decision Channels can impose application specific structure. A Dictionary (aka Hash Map) is sufficient to describe the data. Data should provide minimally required fields to be considered valid. Data should be timestamped and have a valid lifetime. It should provide its type and identifier. Beyond its valid lifetime data is considered stale and unusable for decision making process. For scalability reasons, access to most recent data from the components of Decision Channel should be instantaneous, however, access to old and archived data in the DataBlock is useful for debugging and audits and does not have same instantaneous access restrictions. Valid data in DataBlock is expected to be in the order of 10s of GigaBytes. Actual size may vary based on the complexity and number of Decision Channels. Interfaces provided by unconditional databases like UcondDB maybe able to satisfy our requirements as a DataBlock. This needs to be further investigated.

DataBlocks and global namespaces act as a medium for different components in a decision channel to communicate with each other. A DataBlock is unique to a specific instantiation of a Task Manager and contains information representing current state of the system as known to Decision Channels and the modules that are contained within. DataBlocks can be used to provide audit trails.

In order for the Decision Cycle to make accurate and consistent decisions, it is important that the Decision Channel acts upon most recent information from the DataBlock. When new information is acquired and stored in the DataBlock by sources periodically, it invalidates old data. Any failures to acquire data in a reasonable timeframe would result in the Decision Channel making less than optimal decisions. To avoid such scenarios, data in the DataBlock should have a valid lifetime. If this data is not refreshed within its lifetime, it should be considered invalid and the Decision Cycles should handle this appropriately. Data from different sources can have different life span.

10.2.3 Provisioner

The HEPCloud architecture allows for using different provisioners provided the components of the Decision Channel follow same communication protocol. HEPCloud at Fermilab will use a GlideinWMS Factory as its provisioner. GlideinWMS has a well defined communication protocol for requesting resources from the different resource providers through HTCondor. As a result, the Decision Engine should be able to communicate with the GlideinWMS factory using the GlideinWMS protocol through HTCondor classad mechanism.

10.3 Use Case

This use case description is written with the FNAL deployment as an assumption. Therefore much of the terminology used will be HTCondor and glideinWMS based. This use case is presented in two parts. The first part is a description of the use case as it was executed, and the second part is a translation into a configuration and pseudo-code for the Decision Engine as designed in this document.

10.3.1 Use Case: As-is Description

The Amazon Web Services optimization use case has been selected because it represents the most complex workflow implemented to date. The workflow was prototyped in a “proto-decision engine” in late 2015/early 2016. The prototyped workflow had three phases: a) resource identification, b) resource selection, and c) resource request.

During the resource identification phase, the provisioner is queried to select all available factory entries. For the Amazon use case, a factory entry corresponds to a Region/Zone/AMI combination. These combinations are explicitly defined in the factory. The job queue is queried to obtain a list of jobs to be considered for overflow into the cloud. The remaining budget is queried to ensure that additional requests do not use more budget than is available. For the purposes of this use case, authorization checks are assumed to be inherent in the queries. If you are not authorized, the queries fail.

The resource selection phase queries for price-to-performance ratios for the entries, estimated cost for resources requests, the latest pricing information, and a figure of merit for ranking entries. This phase ranks the factory entries in order of desirability while removing entries if they are unworkable or completely undesirable.

The resource request phase runs through the remaining entries in order of desirability and calculates how many resources to request for each entry. This request is then published to the provisioner.

10.3.2 Use Case: Decision Engine Translation

The as-is use case assumed that the VO was already authorized to request cloud resources, and all the jobs submitted were going to run on the cloud. This translation adds a higher level logic flow to select resources to provision.

Sources:

Source Name	Source Type	Data Label	Description
s_job_query	HTCondor Query	d_jobs	Job information for jobs in the queue
s_local_capacity	HTCondor Query	d_local_capacity	Number of “slots” available locally to run jobs
s_hpc_allocation	Accounting Query	d_hpc_allocation	Hours left in HPC allocation
s_cloud_budget	Accounting Query	d_cloud_budget	Dollars left to spend on Cloud resources
p_overflow_threshold	Parameter Query	p_overflow_threshold	Quantity of idle jobs required to trigger overflow
p_overflow	Parameter Query	p_overflow	Boolean: allow overflow
p_overflow_hpc	Parameter Query	p_overflow_hpc	Boolean: allow HPC
p_overflow_cloud	Parameter Query	p_overflow_cloud	Boolean: allow cloud

Transforms:

Transform Name	Data Label	Description
t_osg_requests	d_osg_requests	Performs the algorithm to spread requests out among acceptable OSG entries.

t_hpc_requests	d_hpc_requests	Performs the algorithm to spread requests over time and over HPC sites.
t_cloud_requests	d_cloud_requests	Performs the algorithm described above to optimize the requests across entries, contain costs, etc.

Resource Type Selection Logic Engine:

Fact Name:	**Fact**
jobs_present	(len(jobs) > 0)
overflow_condition	((len(jobs) - local$slots) > params$threshold)
overflow_permitted	(params$overflow_permitted)
overflow_hpc_permitted	(params$overflow_hpc_permitted)
overflow_cloud_permitted	(params$overflow_cloud_permitted)
hpc_sufficient_allocation	(hpcinfo$hours_available > SUM(jobs$time))
cloud_sufficient_budget	(cloudinfo$available_budget > SUM(jobs$estimated_cost))

Rule:	**Description:**
(jobs_present & overflow_permitted & overflow_condition) -> [use_osg, handle_overflow]	If jobs are present, overflow is permitted, and we are in an overflow condition: action = use OSG and set a fact that enables overflow actions. The use_osg action will trigger a publisher that publishes OSG resource requests to the provisioner.
(jobs_present & !(overflow_permitted & overflow_condition)) -> [use_local]	If jobs are present and overflow is not permitted or not in overflow condition: action = use local
(overflow_hpc_permitted & handle_overflow & hpc_sufficient_allocation) -> use_hpc	If HPC overflow is permitted and in overflow condition and there is sufficient HPC allocation available: action = use HPC

(overflow_cloud_permitted & !use_hpc & cloud_sufficient_budget) -> use_cloud	If cloud overflow is permitted and use_hpc didn't trigger and there is sufficient cloud budget, action = use cloud. For this use case, use_cloud triggers another logic engine with its own facts and rules.
(!use_cloud & !use_hpc) -> use_local	This is a catch all rule that says if none of the other rules triggered, action = use local

Cloud Resource Selection Logic Engine:

Fact Name:	Fact:
good_total_estimated_budget	ds$budget - (match_table.filter(still_good == True).mutate(req_cost=number_to_request*burn_rate).summarize(SUM(req_cost))) > 0
good_total_burn_rate	ds$targetburn <= (match_table.filter(still_good == True).summarize(SUM(burn_rate)))

Rule:	Description:
Good_total_estimated_budget & good_total_burn_rate -> publish_cloud_request	If total estimated budget and burn rates are good, publish the calculated cloud requests to the provisioner and publish all required information to the relevant HEPCloud subsystems.
!(Good_total_estimated_budget & good_total_burn_rate) -> publish_cloud_refusal	If both total estimated budget and burn rates are not good, publish the a refusal to request cloud resources to the appropriate logging facility.

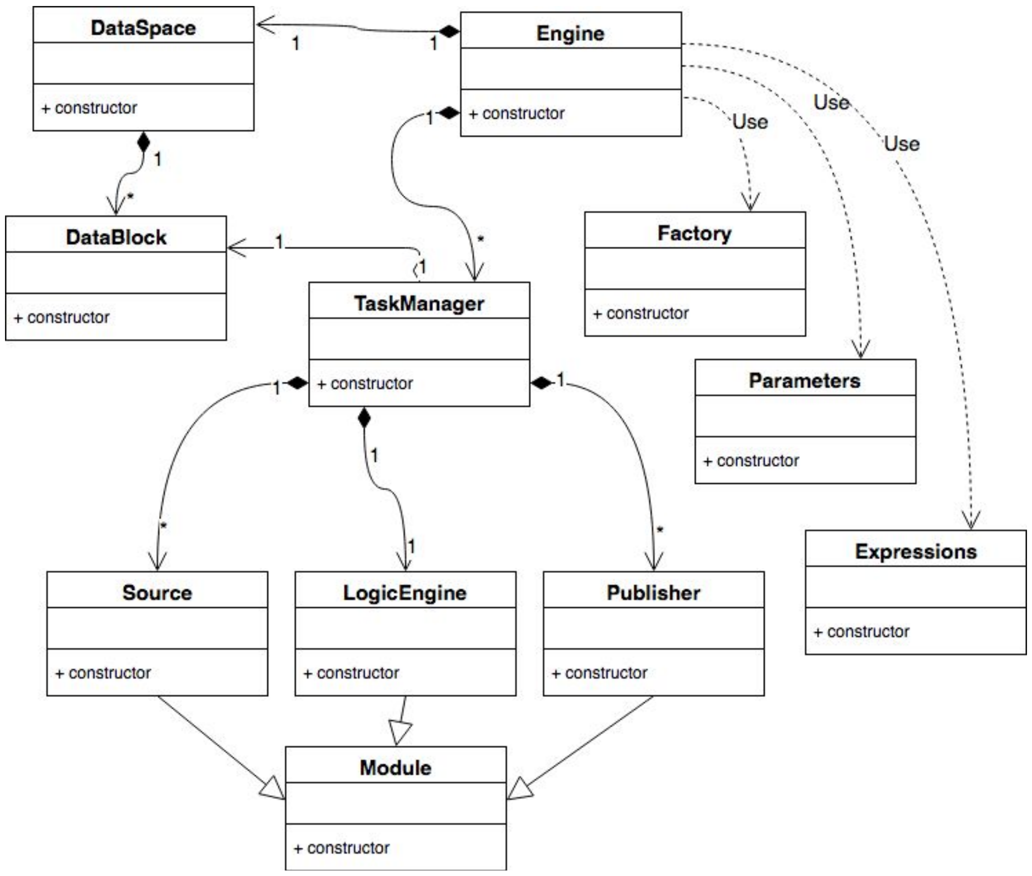
Publishers(s):

This example only lists publishers for the Cloud use case.

Publisher Name:	Description:
pub_cloud_req	Publish the cloud requests to the provisioner
pub_cloud_req_monitoring	Publish the request to the HEPCloud

	monitoring subsystem.
pub_cloud_req_accounting	Publish the requests to the HEPCloud accounting subsystem
pub_cloud_req_refusal	Publish the refusal to request new cloud resources, the reason, and all required supporting data to the proper logging subsystems.

Appendix A



This is a simplified class diagram of the system, showing essential elements and their relationships. Note that everything is not included in this diagram. Expressions are used by the LogicEngine. Parameters are used in construction of Task Managers, Sources, and Publishers. The interface is left out of this diagram. The relevant interfaces are described directly in document sections.

Appendix B

A sketch of the interaction between task manager scheduling functions and the data block for handling timing issues that may occur between decision cycles, backups, and concurrent puts to data blocks.

```
DataBlock {
 put() { }
 get() { }
}

Backup(db_curr) { …    return db_prev }

TaskManager {

 do_backup() {
  lock(my.cycle_lock)
  while(! my.active_cycle)
  {
    condition_wait(my.cycle_cv,
     my.cycle_lock)
  }

  db_prev = Backup(my.data_block)
  my.active_cycle=false
  release(my.cycle_lock)
  return db_prev
 }
```

```
 do_cycle() {
  db_prev = do_backup()
  my.run_transforms(db_prev)
  my.run_logic_engine(db_prev)
  my.run_publishers(db_prev)
 }

 do_one_source(src) {
  src.run()
  lock(my.cycle_lock)
  my.active_cycle=true
  condition_signal(my.cycle_cv,
    my.cycle_lock)
  release(my.cycle_lock)
  reschedule_source(function() {
do_one_source(src) }, now+src.period)
 }

 start_all_sources() {
   map(source_list, function(s) {
    reschedule_source(s, now) } )
 }

 run_cycles() {
   while(online) do_cycle()
 }
}
```