Paper SAS2280-2018

Model Life Cycle Automation Using REST APIs

Glenn Clingroth, Wenjie Bao, and Robert Chu, SAS Institute Inc.

ABSTRACT

SAS has created a new release of its model life cycle management offerings. This release provides an open, RESTful API that enables easier and more customizable integrations. Applications, including third-party applications, can communicate with the model life cycle management system through the RESTful web service APIs.

RESTful integration has become the new standard for web service integration. SAS® APIs conform to the OpenAPI specification, so they are well documented and easy to understand. This article walks readers through the major steps of the model management life cycle: build, register, compare, test, compare, approve, publish, monitor, and retrain. Models that can participate in the model management life cycle are created in SAS® Enterprise Miner, SAS® Visual Data Mining and Machine Learning, or in open-source modeling tools such as the Python scikit-learn toolkit or Google TensorFlow.

Using the latest versions of SAS® Model Manager, SAS® Workflow Manager, and SAS® Workflow Services, this article shows how to create custom model life cycles that integrate with various SAS and open-source services. After reading this article, readers will have a clear understanding of model life cycle management and how to start creating their own integrated custom model life cycles by calling SAS REST APIs.

INTRODUCTION

Model management of analytical and statistical models is more than just version control for model code. A model consists of the model's code, variables, metadata, analysis criteria, performance history, supporting documentation, and so on. This information isn't static; changes can occur slowly or in days or hours and it is important to know when these changes occurred and who made the changes. As a given model continues to be used, its performance naturally degrades. As the performance degrades, new models are produced to replace it. These models must be tested and compared with the current champion model and when a successor is selected, it is used when analyzing the production data.

SAS Model Manager provides all of these functions in a web application. But as it is a web application, it is inherently manual and requires users to log on in order to enter, view, and review data. In some cases, this is appropriate. But often organizations have processes that are better served by automated interactions with the models and their metadata. Also, many functions don't require manual intervention. For example, models are evaluated against known metrics. Model evaluation can be an automated, iterative process that notifies the users when the model performance no longer meets expectations.

To meet these needs, SAS Model Manager provides a set of REST APIs that are used to customize and manage the model management life cycle. These REST APIs cover the full range of model management functions, and an analyst or developer has access to the same APIs that are used by the SAS Model Manager web application.

The examples in this paper are presented using Python. Python provides a clean interpreted syntax and is used by developers and analysts in many organizations. The paper also provides an example that uses the SAS Workflow service to show how a workflow is used to automate the life cycle.

SAS Model Manager is complex software and the SAS® Viya®release adds some new concepts that the reader needs to understand. For more information, see the SAS Model Manager 15.1: User's Guide.

MODEL MANAGEMENT LIFE CYCLE

Each organization has its own model management life cycle. Models can be managed by one person or be vetted through several organizations, and the process can change based on the types of models. But most model life cycles incorporate these events: build, registration, test, comparison, approve, publish, monitor, and retrain.

While creating a proper model is arguably the most crucial step in the life cycle, it is also the most complex, and for the purposes of this article we assume that models have already been built and will focus on the remaining tasks.

SAS Model Manager has supported life cycle management since its early versions and it does not prescribe any specific life cycle. But for the purposes of this paper we will use the life cycle that is described in Figure 1.

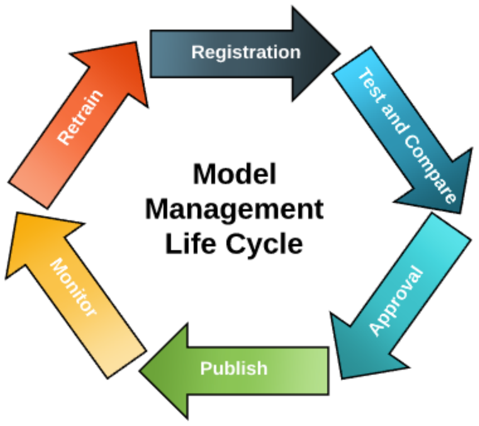


Figure 1: Model Management Life Cycle

MODEL MANAGEMENT LIFE CYCLE USE CASE

The Mycology department of the University of the Amazon has received hundreds of new mushroom species. They have been told that there are potentially thousands more species and they could create hundreds of jobs harvesting edible mushrooms throughout the Amazon basin. But they don't want to waste time harvesting the wrong mushrooms, so first each species must be classified as poisonous or edible.

Analyzing each species individually will take months. But they have a database of characteristics for 10,000 known species to assist with the job. Data analysts have produced five different models based on standard characteristics and feel that gathering the mushrooms can begin based on the predictions from the best of those models. First the different models need to be tested to validate that the code is accurate and executes properly against the data set. Now those models need to be compared and analyzed against each other so that the best one is used.

While mushrooms are being gathered, mycologists test each species to determine the correctness of the model. As the true results are determined, they are compared against the predicted results to determine if the model properly selected the mushrooms that should be harvested. If more than 4% of the results are incorrect, the data analysts retrain the model including the new information and use the new champion model in future analyses.

APIS FOR MODEL LIFE CYCLE MANAGEMENT

SAS has provided a comprehensive set of APIs for use with the SAS Viya solutions. These APIs are implemented using OpenAPI standards for documentation and provide a standard syntax and access point for integration with the SAS platform.

Model management functions are separated into three APIs:

- Model Management API
- Model Publish API
- Model Repository API

This paper provides example usage for the SAS Model Manager APIs, but does not document all possible usages. To learn more, refer to Getting Started with SAS® Viya® REST APIs.

When describing an API call, we provide the required information in the following format:

```
HTTP Method: URL
Headers: Required HTTP Headers
Body: call body in JSON format, if needed
```

For example:

```
POST: /modelManagement/reports
Headers:
   "Accept": "application/vnd.sas.models.report.comparison.model+json",
   "Authorization": "Bearer <Token>"
Body:
{
    "name":"compareModels",
    "description":"",
    "modelUris":[
          "/modelRepository/models/{model_1_id}",
          "/modelRepository/models/{model_2_id}"
    ]
}
```

MODEL REPOSITORY API

The Model Repository API provides the core RESTful API for model repository management and provides support for registering, organizing, and managing models. The API is accessed with the root context of /modelRepository and provides functions for repositories, projects, and models.

MODEL MANAGEMENT API

The Model Management API provides a RESTful API that works with the model repository. The API is accessed with the root context of /modelManagment and provides functions for model performance and comparison, workflow management, and code retrieval for scoring and publishing.

MODEL PUBLISH API

The Model Publish service API provides a RESTful API that supports publishing models for use in production data servers such as CAS, Hadoop, SAS Micro Analytic Service, and Teradata. The API is accessed with the root context of /modelPublish and provides functions to publish models and to manage publish destinations.

WORKING WITH MODELS AND PROJECTS IN A MODEL REPOSITORY

A model in SAS Model Manager can be as simple as a name or a single file, or it can be a complex collection of files, variables, and other metadata. It is most common that a model is a complex collection, but there are use cases for each style of model. A model does not need to be a member of a project, but they are in our examples. A model project is a collection of models that are divided into analysis groups that are termed “versions”. A model project, similar to a model, can be simple or complex with only a single model or many models.

Models and projects are stored in a hierarchical folder structure. The top-level of the hierarchy is a collection of repositories. Repositories are used to organize models and projects, and provide a root level for assigning permissions.

Other SAS web applications such as Model Studio directly integrate with the model repository to create projects and register the models. Model Studio automates model creation and analysis and can create models using multiple algorithms on the same data set. These projects and models are then viewed and edited in SAS Model Manager. The core examples in this paper use Model Studio to create, import, and retrain models.

It is also possible to create and import all of the content through the Model Repository API. The examples provided in this section are included to show how to interact with the Model Repository API and are instructive for how to use any of the SAS Viya RESTful APIs.

GETTING AN AUTHORIZATION TOKEN

The first step in using the API is authorization. SAS uses an OAuth2 authorization framework with OpenID Connect extensions. To authorize directly, it is necessary to post data to the following uri: http://<server>:<port>/SASLogon/oauth/token. The body of the post contains the userId and password in clear text.

The following Python program performs the authorization and returns an authorization token in the “myToken” variable that is used with all subsequent examples. Note that the token expires after a period of inactivity, so it is necessary at times to request a new token.

```
import requests
import json

protocol='http'
server='sas.myorganization.com'
authUri='/SASLogon/oauth/token'
user="sasuser"
password="password"

headers = {
   'Accept': 'application/json',
   'Content-Type': 'application/x-www-form-urlencoded'
}
payload= 'grant_type=password&username=' + user + '&password=' + password
authReturn = requests.post(protocol + '://' + server + authUri ,
                           auth=('sas.ec', ''),
                           data=payload,
                           headers=headers)
myToken = authReturn.json()['access_token']
```

RETRIEVE THE REGISTERED PROJECT

Our life cycle starts with analysts using Model Studio pipelines to generate a collection of models to solve a given problem. Some of these generated models are selected to be registered into the model repository where SAS Model Manager accesses them. To retrieve the newly registered SAS Model Manager project, ”MushroomAnalysis”, from the model repository use the following code:

```
# The following variables were defined in the above code: protocol, server,
myToken
headers={
  'Accept': 'application/vnd.sas.collection+json',
  'Authorization': 'Bearer ' + myToken
}
projectName = 'MushroomAnalysis'
url = protocol+'://'+server+'/modelRepository/projects?name='+projectName
projectResult = requests.get(url, headers = headers)
myProj = projectResult.json()['items'][0]
projectId = myProj['id']
```

This retrieves a summary of the project. The summary contains key details about the project as seen below:

```
'id': '7a669a50-c3cc-458b-88ac-75f182b4f48b',
'name': 'MushroomAnalysis',
'createdBy': 'edmdev',
'creationTimeStamp': '2018-01-24T21:01:02.607Z',
'folderId': '7bdb5885-c6d8-4181-abde-bbde423967df',
'latestVersion': 'Version 1',
'modifiedBy': 'edmdev',
'modifiedTimeStamp': '2018-01-24T21:01:04.030Z',
'repositoryId': '3db982df-229e-4b32-9e41-329102d9eb09'
```

As seen in the code above, values are referenced from the result object using the property name. So, to retrieve the project identifier, use this code *projectId = myProj['id']*. The pattern is used for all properties.

All results also contain a list of links. The links are urls that can guide usage of the object and contain references to associated data and potential functions.

```
'links': [
{'href': '/modelRepository/projects/7a669a50-c3cc-458b-88ac-75f182b4f48b',
   'method': 'GET',
   'rel': 'self',
   'type': 'application/vnd.sas.models.project',
   'uri': '/modelRepository/projects/7a669a50-c3cc-458b-88ac-
75f182b4f48b'},
  {'href': '/modelRepository/projects/7a669a50-c3cc-458b-88ac-
75f182b4f48b',
   'method': 'PUT',
   'rel': 'update',
   'type': 'application/vnd.sas.models.project',
   'uri': '/modelRepository/projects/7a669a50-c3cc-458b-88ac-75f182b4f48b'}
]
```

The links are specific to the current object or collection and reflect the security associated with the object, that is, if the user can’t perform a function then the link will not be provided. The ‘method’ corresponds to the http method and the ‘type’ corresponds to the type specified in the ‘Accept’ and ‘Content-type’ headers of the http call.

To retrieve the ‘update’ uri from the collection shown above, use the following syntax:

```
myRepo['links'][1]['uri']
```

RETRIEVE AND UPDATE MODELS

When Model Studio registers a project, it also registers one or more models. Models are retrieved for a project with the following code:

```
# The following variables were defined in the above code: protocol, server,
myToken, projectId

headers={
  'Accept': 'application/vnd.sas.collection+json',
  'Authorization': 'Bearer ' + myToken
}
url= protocol+'://'+server+'/modelRepository/projects/'+projectId+'/models'
modelResult = requests.get(url, headers = headers)
myModelList = modelResult.json()
myModel = myModelList['items'][0]
modelId = myModel['id']
```

The above code retrieves all of the models from the project and creates a reference to the first model and stores the ID of the model as the value of the “modelId” parameter. To update the model, use the ‘update’ link from the “myModel” object variable. The code below updates the model description property:

```
# The following variables were defined in the above code: protocol, server,
myToken, projectId, modelId

headers={
  'Accept': 'application/vnd.sas.models.model+json',
  'Authorization': 'Bearer ' + myToken
}
url = protocol + '://' + server + '/modelRepository/models/' + modelId
modelResult = requests.get(url, headers = headers)
myModel = modelResult.json()
modelEtag = modelResult.headers['ETag']
myModel['description'] = 'Mushroom Model'

headers={
  'Content-type': 'application/vnd.sas.models.model+json',
  'If-match': modelEtag,
  'Authorization': 'Bearer ' + myToken
}
url = protocol + '://' + server + '/modelRepository/models/' + modelId
modelResult = requests.put(url, data=json.dumps(myModel), headers=headers)
myUpdatedModel = modelResult.json()
```

The above code highlights the optimistic locking mechanism used by the SAS Viya API calls. Each object that can be updated generates a hash code and returns it in the ETag header. When an update is submitted using the PUT method, the value of the ETag header is read from the response header and is passed in the If-match header of the update request. The update request is rejected if the If-match value does not match the expected ETag.

These examples are provided as a quick primer for SAS Viya REST API usage. Deeper examples are included in the example code that is available with this paper.

COMPARE AND TEST MODELS

After models are imported into the common model repository, comparison reports are run for the models. After review of the reports, the models are tested to validate that they execute properly and produce the expected results.

Both SAS Model Manager and Model Studio have the ability to display model comparisons in their user interfaces. Model Studio provides a set of fit statistics and graphs that are also displayed in SAS Model Manager. Model Studio provides a large number of standard statistics such as the Kolmogorov-Smirnov statistic (KS) and Gini coefficient. The output graphs include the Receiver Operating Curve (ROC) and Lift. The model comparison information is created by Model Studio and imported into SAS Model Manager.

COMPARING MODELS

As mentioned above, both Model Studio and SAS Model Manager provide comparison reports. These reports contain comparisons of the major model properties, input and output variables, fit statistics based on the training data set, and plot data, which can include plots for Lift, KS, Gini, and so on. The output is quite extensive, but key elements can be pulled out programmatically for comparison.

Because the tabular output is in JSON format, comparison is best viewed through the web application, but it is possible to retrieve the report content with the following API call.

```
POST: /modelManagement/reports
Headers:
   "Accept": "application/vnd.sas.models.report.comparison.model+json",
   "Authorization": "Bearer <Token>"
Body:
{"name":"compareModels",
 "description":"",
 "modelUris":[
    "/modelRepository/models/{model_1_id}",
    "/modelRepository/models/{model_2_id}"
 ]
}
```

TESTING MODELS

SAS has created a RESTful API for testing all SAS score code. The API has two parts, the Score Definition service and the Score Execution service. A score definition is a description of a repeatable scoring task for a score code file. The Score Definition service creates and stores the definition so that it can be used at a later time. The Score Execution service executes scoring code against a CAS data set and writes the execution output to a CAS data set. The Score execution output includes a SAS log and a reference to the output data set. Prior to scoring, it is necessary to make sure that the input data source is available to all CAS sessions and that the user has Write access to the output library. The Score services provide a rich environment for testing score code execution

The scoring test definition specifies items such as the input and output data sources and column mappings and allows a test to be run multiple times. Typically, the scoring definition is created in the SAS Model Manager web application.

Once the test definition is created, it is executed to test the score code of the model. In this example, execution requires the ID of a Score Definition that has been previously created. The Score Execution service retrieves the model scoring file, typically *score.sas*, to generate wrapper code specific to the code type and the parameters of the score definition. The service then submits the wrapped score code to the appropriate scoring environment: Base SAS or CAS.

Execution is asynchronous, so to see results, the execution job is repeatedly checked until it reaches a completed success or failure state. If execution succeeds, then the output data set is created. Output data sets are stored on the CAS server in the output library specified in the score definition and are named with a combination of the test name, model name, model version, and current date and time (for example, Test_1_MushroomAnalysisModel_v1_0_2018_01_22_17_13_51_output). The result data set is made global in the CAS library, so that all authorized users can access it, but it is not saved to a file.

```
# The following variables were defined in the above code: protocol, server,
myToken, projectId, modelId

import time

headers={
  'Content-type': 'application/json',
  'Accept': 'application/json',
  'Authorization': 'Bearer ' + myToken
}
scoreExecutionRequest = {'scoreDefinitionId': scoreDefinitionId,
                         'name':'ScoringTest'}
url = protocol + '://' + server + '/scoreExecution/executions'
scoreExecutionResult = requests.post(url,
data=json.dumps(scoreExecutionRequest), headers = headers)
scoreExecutionId = scoreExecutionResult.json()['id']
scoreState = 'running'
pollingUrl = url + '/' + scoreExecutionId
while scoreState == 'running':
    time.sleep(5)
    scoreExecutionPoll = requests.get(pollingUrl, headers = headers)
    scoreState = scoreExecutionPoll.json()['state']
executionLogUri = scoreExecutionPoll.json()['logFileUri']
```

In the above example, “scoreState” variable is populated with the final execution state: completed or failed, and “executionLogUri” variable is populated with the file service URI of the execution log.

DETERMINING THE BEST MODEL

After the models are analyzed, one model is selected as the champion. Setting the model as the champion signifies that it is ready for publishing to the production system.

In order to set the champion, the project output variable must first be set. Since the project was created by Model Studio with a binary target, we can use the EM_EVENTPROBABILITY variable as the output variable. The EM_EVENTPROBABILITY variable is a standard variable added to binary models by Model Studio. The output variable must be set before setting the champion model since setting it afterward clears the champion flag from the project. The code below sets a champion model but assumes that variable mapping has already been handled.

```
headers={
  'Accept': 'application/vnd.sas.models.project+json',
  'Authorization': 'Bearer ' + myToken
}
url = protocol + '://' + server + '/modelRepository/projects/' + projectId
+ '/champion'
urlParams = 'modelId=' + championModelId + '&keepChallenger=false'
setChampionResult = requests.post(url + '?' + urlParams, headers = headers)
setChampionResult.json()
```

When we discuss workflow later, you will see how to request that a model be approved by one or more people prior to being set as the champion.

PUBLISHING MODELS

After the model is set as the champion, it is published to a production system. The Model Publish API supports publishing of many types of models to publish destinations. The supported publish destinations are CAS, Teradata, Hadoop, and SAS® Micro Analytic Service. Publish destinations and permissions are configured by a SAS administrator.

The Model Publish API is a general-purpose API for publishing model code that can come from many sources. For publishing models from the SAS Model Manager repository, the Model Management API provides a publish endpoint that is better suited for life cycle automation.

```
POST: /modelManagement/publish?force=false
Headers:
   "Content-Type": " application/vnd.sas.models.publishing.request+json",
   "Authorization": "Bearer <Token>"
Body:
{
    "name":"Published model MushroomAnalysisModel1",
    "notes":"Publish models",
    "modelContents":[{
        "modelName":"MushroomAnalysisModel",
        "sourceUri":"/modelRepository/models/{modelId}",
        "publishLevel":"model"
    }],
    "destinationName":"{publishDestination}"
}
```

MONITORING MODELS

Model monitoring tracks the performance of a model over a period of time. In a monitoring process reports are produced that allow analysts to see if the published model is still performing as expected.

SAS Model Manager is quite flexible in how performance is monitored. It includes model performance monitoring that supports both champion and challenger models. This paper presents one approach to preparing the input tables and performing the periodic execution and analysis of the performance job.

PREPARE THE MONITORING DATA SET

A monitoring data set is created in a CAS library. At a minimum, the data set contains columns for all of the project input variables and columns containing the actual outcomes. Additional variables can be included, these won't be used in the performance calculations, but can be helpful for monitoring distribution drift or for persisting the observation id. For this example, the table contains variables for both the actual outcome but does not include the predicted outcome. The predicted outcome is discussed below.

For automated monitoring, SAS Model Manager uses a naming convention for the performance data sets. The format is: <data_prefix>_<sequence>_<time_period_label>. For example: MushroomPerfData_1_Jan2018, indicates that the data prefix is "MushroomPerfData", that this is the performance data set for the initial time period, and that on reports the data points are labeled "Jan2018". So, considering that performance checks are run monthly, after the April performance data is prepared, the following data sets are available:

MushroomPerfData_1_Jan2018
MushroomPerfData_2_Feb2018
MushroomPerfData_3_Mar2018
MushroomPerfData_4_Apr2018

Run an ETL process to merge production data into the monitoring data set. After the ETL process is complete, the predicted output variables might be empty or missing. If so, then it is necessary to generate the predicted output variables prior to getting the performance results.

DEFINE A PERFORMANCE TASK THAT GENERATES THE PREDICTED OUTPUT VARIABLES

While in some processes the predicted output is already part of the prepared monitoring data set, in our scenario it is assumed that the predictions are not available. In this case, it is necessary to generate the predicted output and merge that data into the monitoring data set.

To get the predicted output it is possible to run the model score code using the monitoring data set as input. As discussed above in the Testing Models section, the Score Execution API is used to execute the model score code and generates an output data set that contains the prediction variables.

However, it is also possible to generate the scoring output during the execution of the performance monitoring task. For automation, this reduces the number of data preparation steps and streamlines the processing. To generate the scoring output during execution, set the *scoreExecutionRequired* parameter of the performance task to true.

When creating the predictions, the best model to use is the champion model for the time period when the data was gathered. This information is available from the project history:

```
GET: /modelRepository/projects/{projectId}/history
Headers:
   "Authorization": "Bearer <Token>"
```

The example below assumes that the champion model has not been updated since the data was gathered. Therefore, in the performance task request, the value of the “modelId” parameter is set to the ID of the current champion model. Since the “scoreExecutionRequired” parameter is set to True, the predicted values will be calculated using the current champion model.

```
casLibrary = '/casManagement/servers/cas-shared-default/caslibs/public'
resultLibrary = '/casManagement/servers/cas-shared-default/caslibs/mmLib'
inputVarList = []
outputVarList = []
for variable in projectInputVars:
    inputVarList.append(variable['name'])

for variable in projectOutputVars:
    outputVarList.append(variable['name'])

performanceTask = {
    'modelId': championModelId,
    'inputVariables': inputVarList,
    'outputVariables': outputVarList,
    'maxBins': 10,
    'resultLibraryUri': resultLibrary,
    'dataLibraryUri': casLibrary,
    'dataPrefix': 'mushroom_scored_',
    'scoreExecutionRequired': True,
    'performanceResultSaved': True,
    'forcingRunAllData': True,
    'loadingPerfResult': True
}
headers={
  'Content-type': 'application/vnd.sas.models.performance.task+json',
  'Accept': 'application/vnd.sas.models.performance.task+json',
  'Authorization': 'Bearer ' + myToken
}
url = protocol + '://' + server + '/modelManagement/performanceTasks'
performanceTaskResult = requests.post(url,
data=json.dumps(performanceTask), headers=headers)
```

EXECUTE THE PERFORMANCE TASK

After a task is defined, it is executed using the performance task ID. The ID is retrieved from the “performanceTaskResult” variable. Here is an example that retrieves the performance task ID and executes the task to update the performance results:

```
import time
performanceTask = performanceTaskResult.json()
performanceTaskId = performanceTask['id']
headers={
  'Authorization': 'Bearer ' + myToken
}
url = protocol + '://' + server + '/modelManagement/performanceTasks/' +
performanceTaskId
performanceExecutionResult = requests.post(url, headers = headers)
performanceExecutionResult.json()
performanceJob = performanceExecutionResult.json()
performanceJobId = performanceJob['id']
performanceJobState = performanceJob['state']
pollingUrl = url + '/performanceJobs/' + performanceJobId
while performanceJobState == 'running':
    time.sleep(5)
    performanceExecutionPoll = requests.get(pollingUrl, headers = headers)
    performanceJobState = performanceExecutionPoll.json()['state']
```

When the “performanceResultsSaved” parameter of the performance task is set to True, the performance results are written to a collection of data sets in the CAS library specified in the “resultLibraryUri” parameter. The result data sets contain information specific to model lift, characteristic shifts, output stability, and so on. The type of the model controls the tables that are produced. The data sets are stored in the CAS library with the following naming convention *_<project Id><result type>*. For example, the fit statistics for my project are stored in a data set named “_FFC50731-FAA4-4632-9DA7-9DF1FE384C11MM_FITSTAT”.

REVIEW THE PERFORMANCE RESULTS

To visualize the results, SAS provides SAS Visual Analytics to create graphs and reports. However, SAS also provides the SWAT Python library for communicating with CAS and using this library, Python code can be used to generate graphs and reports based on the performance results. The following example generates a single response lift chart using SWAT and mathplotlib inside of a Jupyter notebook:

```
import requests as req
import json
from swat import *
%matplotlib inline  # For use only in Jupyter notebooks.
import matplotlib.pyplot as plt

cashost = '<cas server ip>'
casport = <cas server port>
s = CAS(cashost,casport,'<username>','<password>')

liftDf =s.CASTable('FFC50731-FAA4-4632-9DA7-9DF1FE384C11.MM_LIFT')
liftDf.fetch()
```

```
line1, = plt.plot(liftDf['Percent'].values[0:99],
liftDf['Lift'].values[0:99], '-', linewidth=2)
plt.ylabel('Lift')
plt.xlabel('Percent')
plt.show()
```

RETRAINING MODELS

Over time, due to changing conditions in the data, the performance of a model degrades. When analyzing the new species of mushrooms, it is noticed that a significant number of the Amazonian mushrooms that appear edible based on previous classifications are actually poisonous. The percentage of errors has exceeded our threshold, so analysts decide to retrain the model using new data.

Since the “MushroomAnalysis” project originated in Model Studio, retraining is performed using Model Studio APIs. SAS Model Manager maintains a retraining link to the Model Studio pipeline project and uses that link in the retrain request. The retraining link is stored in the project parameter “retrainUri”. If the project has been removed from Model Studio, then retraining cannot be performed.

The first step to retraining the data is to produce an updated training data set. The variables of this data set must contain the variables in the training data set and the data set must be stored in a CAS library. Retraining is performed by calling the Data Mining API to request a retrain of the Model Studio project. The call below performs the retrain.

```
POST:
/dataMining/projects/{modelStudioProjectId}/retrainJobs?dataUri=/dataTables
/dataSources/cas~fs~cas-shared-
default~fs~Public/tables/MUSHROOM_DATA_JAN2018&action=batch
Headers:
   "Accept": " application/json",
   "Authorization": "Bearer <Token>"
```

After retraining is completed, the new models are registered in the SAS Model Manager project. The life cycle then returns to the **Test and Compare** step.

A WORKFLOW BASED APPROACH TO LIFE CYCEL AUTOMATION

SAS provides a workflow creation tool, SAS Workflow Manager, that allows creation of BPMN standard workflow definitions. SAS also provides a run-time service that is used to execute the workflow processes. SAS Workflow executes the workflow process as an independent service: workflow-as-a-service. Running workflow as an independent service reduces the coupling between the application and the workflow engine and makes it possible to provide centralized management of all workflow processes instead of managing on an application-by-application basis.

The workflow tools ship as a core piece of the SAS Viya architecture and are integrated with Model Manager. This paper does not provide a detailed description of the SAS Workflow tools, but shows how the Workflow service can be used to provide a more robust model life cycle automation.

This paper discusses a small subset of the BPMN elements, primarily user tasks, and service tasks. A user task is an activity that requires user interaction, and a service task performs an automated function such as making a REST API call and processing the results. For more information about the BPMN standard, see the SAS® Workflow Manager 2.1: User’s Guide, which provides an overview of all elements supported in the SAS workflow tools, and the Object Management Group (OMG) provides the full specification.

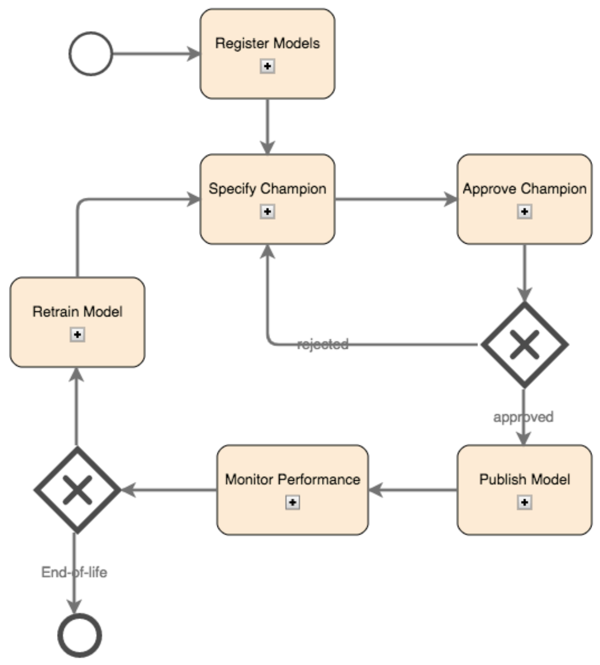


Figure 2: Workflow for a Complete Model Management Life Cycle

Figure 2 shows a BPMN model that can be used for life cycle automation. The workflow definition contains a collection of high-level sub-processes that model the steps of the life cycle stages being performed. Each step in the definition represents several other steps. Let's take a deeper look at two of the areas: model evaluation and approval, and publish, monitor, and retrain.

ENHANCED USER INTERACTIONS

To highlight how workflow helps to improve the interaction among users in various modeling roles, the **Approve Champion** sub-process is added to the overall life cycle. Figure 3 provides detail of the approval flow. In this flow, a user in the model analyst role reviews the champion model and submits it for approval. A user in the data manager role reviews the approval request and chooses to approve or reject the model. If the model is approved, then the workflow moves to the **Publish Model** step, otherwise the workflow returns to the **Specify Champion** step. The timer on the **Approve champion model** task is set to fire in 5 days if the task has not been completed. If the timer fires, then an email notification is sent to the data manager responsible for the approval request.

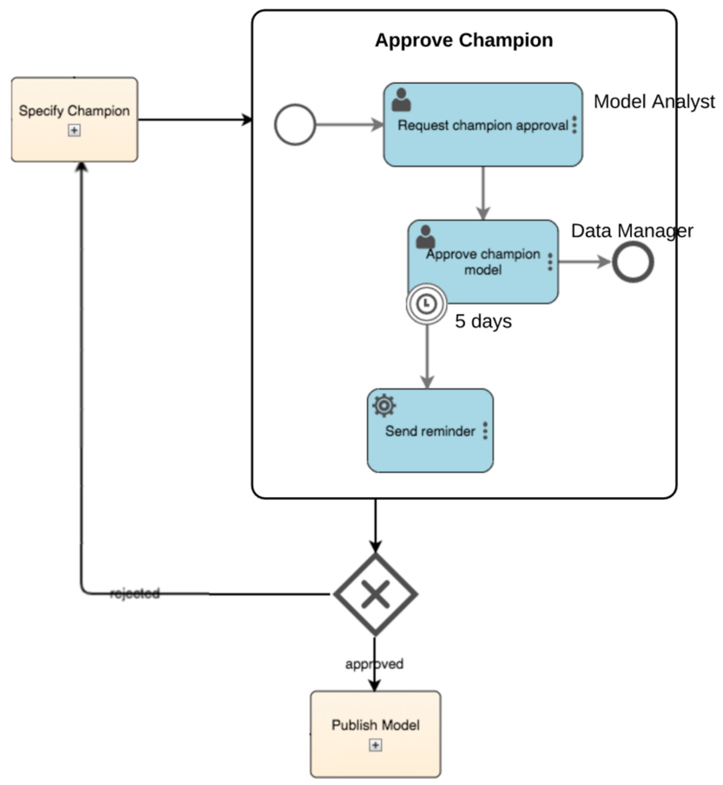
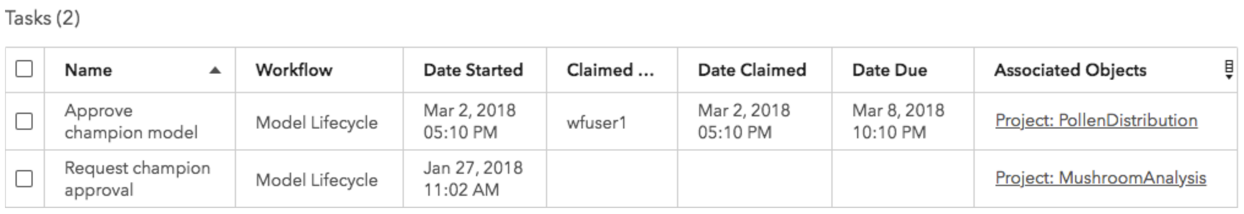


Figure 3: Approval Sub-Process Detail

The **Approve Champion** sub-process shows how to integrate user interactions into the model management life cycle. This list is also available through the Model Management API.

```
GET: /modelManagement/workflowTasks
Headers:
   "Accept": " application/vnd.sas.collection+json",
   "Authorization": "Bearer <Token>"
```

This information is also available in the SAS Model Manager application. Figure 4 shows the SAS Model Manager Tasks category, which displays all model management workflow tasks that are owned by or can be performed by the authenticated user. From this list, users choose to work on tasks or open the model project to review information associated with the task.



Tasks (2)

☐	Name	Workflow	Date Started	Claimed ...	Date Claimed	Date Due	Associated Objects
☐	Approve champion model	Model Lifecycle	Mar 2, 2018 05:10 PM	wfuser1	Mar 2, 2018 05:10 PM	Mar 8, 2018 10:10 PM	Project: PollenDistribution
☐	Request champion approval	Model Lifecycle	Jan 27, 2018 11:02 AM				Project: MushroomAnalysis

Figure 4: Workflow Tasks View

In the above example, the user, “wfuser1”, is in both the model analyst and data manager roles. Therefore, the user “wfuser1” can work on two tasks; currently the user “wfuser1” has claimed the **Approve champion model** task for the “PollenDistribution” project. The user “wfuser1” is also one of the users that can choose to work on the **Request champion approval** task for the “MushroomAnalysis” project.

The workflow definition can request that a user provide feedback when they complete a task. The feedback request is modeled as a prompt on the user task. When the user completes the task, they submit answers to the prompts. In the SAS Model Manager web application, the prompts are displayed as a data-entry form. When the user opens the **Approve champion model** task, they are presented the form seen in Figure 5. Using this input, the user indicates if the champion model is acceptable for deployment to the production system.

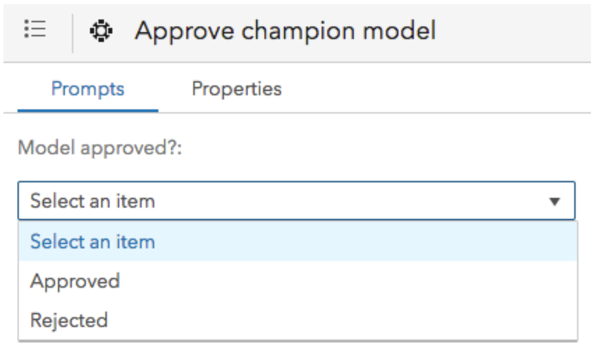


Figure 5: Approve Champion Model Prompt

A set of data objects is defined in the workflow definition. Data objects are similar to variables in a computer program. They hold information that is required to successfully execute a process. Among others in this process, we have a data object for “ApprovalState”. When the user completes the **Approve champion model** task, the value for the “Model approved?” prompt is set. This value is stored in the “ApprovalState” data object. The **Approve Champion** step leads into a decision gateway. The gateway examines the “ApprovalState” data object and if it is set to “Rejected” the workflow routes back to **Specify Champion**. Otherwise, the workflow continues to the **Publish Model** step.

ORCHESTRATE SERVICE TASKS

Orchestrated execution of service tasks is a primary use of workflow for life cycle automation. Orchestration is the coordinated interaction of multiple services or applications. The above code examples have shown usage of the model repository, model management, model publish, score, and data mining services. Using the SAS Workflow tools, we make the same service calls and join those service calls with user tasks for a fully automated experience.

Figure 6 shows a combined service and user task orchestration. Each of the service tasks is configured to make a REST call to a SAS Viya service and the user tasks prompt users to complete necessary work that can’t be automated. Configuring a service task to make a REST call requires the same knowledge of the call that is required to call from Python: HTTP method, headers, and body, but also requires expectations of the result headers and body.

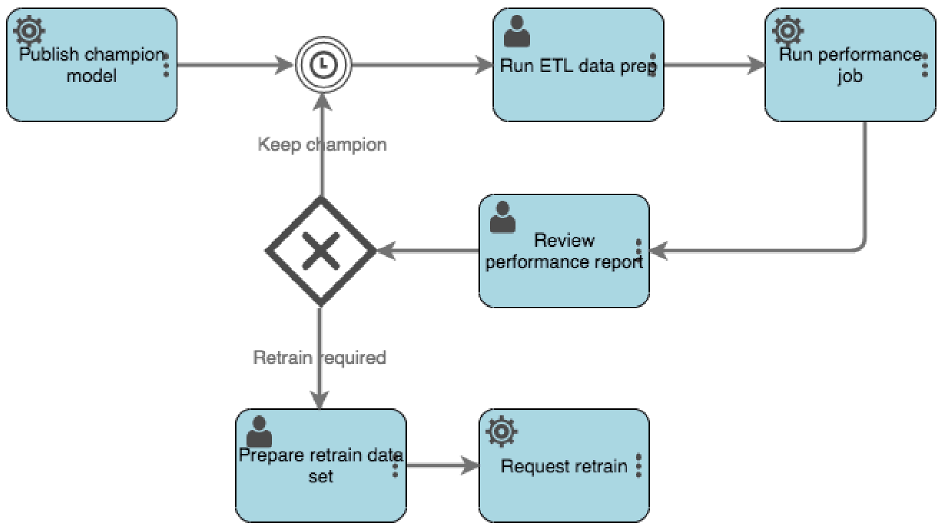


Figure 6: Publish, Monitor, and Retrain Orchestration

One key difference between calling the RESTful APIs from Python or other language and calling from the Workflow service is authorization. The Workflow service handles all authorization when SAS Viya RESTful API calls are executed. To maintain proper security, limits are placed by API providers on which API calls the workflow service is permitted to make. Additional information about security is available in the SAS Viya Administration documentation.

This example logically follows the “Approve champion model” prompt example that is shown in Figure 5. The workflow definition defines the “championModelId” and “publishDestination” data objects. The **Publish champion model** task uses those data objects to construct the REST service call for the /modelManagement/publish endpoint. In the definition of the service task the following values are entered:

```
URL: /modelManagement/publish
Method: POST
Headers: Accept = application/json
         Content-type = application/vnd.sas.models.publishing.request+json
Result code: 201
Input type: JSON
Body: {
         "name":"${publishedModelName}",
         "modelContents": [{
            "modelName": "${modelName}",
            "sourceUri": "/modelRepository/models/${championId}"
         }],
         "destinationName": "${publishDestination}"
      }
```

In the body specification, the values entered with “*${name}*” are data object references. In this example, data objects are used to specify the name and id of the champion model, the destination where the model is published, and the name to assign the model in the publish destination. These values are set on the user tasks prior to this task when users fill in prompts, similar to how the approval state is set in the example above.

The **Run ETL data prep** task, provides a notification to the data analysts that updated performance data is needed. Once one of the analysts has completed the data preparation, the **Run performance job** task is called, and the performance data is updated for the project. At this point another analyst reviews the performance data to make a decision if it is time to retrain the model or keep the current champion. If retraining is required, then it is necessary for the IT department provide prepared retraining data. In this workflow, the user task, **Prepare retrain data set**, provides an indication to the IT department that data preparation is required. After the task is completed, the service task requests the retraining from Model Studio and the flow will return to the **Specify Champion** sub-process (see Figure 2) and the life cycle begins another iteration.

CONCLUSION

This paper has provided an overview of the SAS Viya REST APIs and workflow tools to show how they help to facilitate the model management life cycle. By providing public APIs and workflow integration, SAS is providing a flexible framework to support automating any model life cycle.

REFERENCES

Chu, R., Duling, D. and Thompson, W. 2007. "Best Practices for Managing Predictive Models in a Production Environment", Proceedings of the 2007 Midwest SAS Users Group Conference. Available: http://www.mwsug.org/proceedings/2007/saspres/MWSUG-2007-SAS02.pdf

SAS Institute Inc. 2017. "Understanding SAS® REST APIs." Available: https://developer.sas.com/guides/rest/concepts (accessed January 5, 2018).

SAS Institute Inc. 2018. "Getting Started with SAS® Viya® REST APIs". Available: https://developer.sas.com/guides/rest.html (accessed January 5, 2018).

Justin Richer. 2018. "User Authentication with OAuth 2.0." Available: https://oauth.net/articles/authentication/ (accessed January 5, 2018).

The OpenID Foundation. 2018. "Welcome to OpenID Connect." Available: http://openid.net/connect (accessed January 5, 2018).

SAS Institute Inc. 2017. "SAS® Model Manager 15.1: User's Guide." Available: http://support.sas.com/documentation/onlinedoc/modelmgr/ (accessed January 5, 2018).

SAS Institute Inc. 2018. "SAS® Workflow Manager 2.1: User's Guide." Available: http://support.sas.com/documentation/prod-p/wfmgr/index.html (accessed March 5, 2018).

Object Management Group. 2011. "Business Process Model and Notation (BPMN)." Available: http://www.omg.org/spec/BPMN/2.0/PDF (accessed January 28, 2018)

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Glenn Clingroth
SAS Institute Inc.
Glenn.Clingroth@sas.com

Robert Chu
SAS Institute Inc.
Robert.Chu@sas.com

Wenjie Bao
SAS Institute Inc.
Wenjie.Bao@sas.com