Article from
Predictive Analytics & Futurism
December 2017
Issue 16

From R Studio to Real-Time Operations

By Jeff Heaton and Edmond Deuser

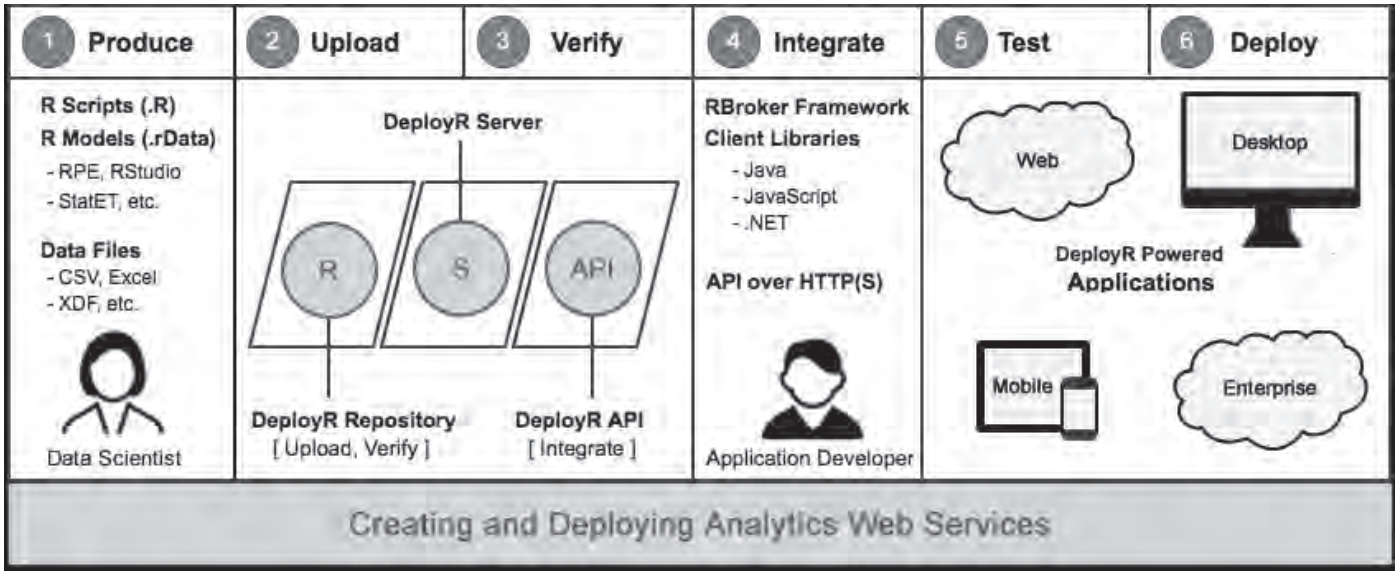
Today more and more data is being created and of ever more importance is the ability to provide real-time access to capabilities on that data and how companies operate those capabilities. This age of data insights will drive how we deliver and communicate these insights. There are several examples of model delivery yet how does the delivery and the self-service capabilities of modeling get operationalized for customers and legacy systems in real time. In this paper, we will explore moving predictive modeling capabilities in R to real time operations. Though this paper specifically targets R, some of these techniques discussed could be applied to other languages. Below describes a basic data science workflow that we will be describing in detail in this paper.

HOW DO WE GO FROM R TO WEB SERVICE?

An application programming interface (API) is an interface to your models provided by a web service. Creating an API involves somewhat different steps than creating a model. The steps of data preprocessing, model fitting, model scoring/prediction, and validation are often intertwined in the R script that the actuary produces. While this sort of organization might work for rapid model development, it will not support the deployment of a real-time model. To deploy the model, the scoring process must be completely separated from the model fitting which is a computationally expensive process, that often uses a large amount of potentially sensitive data. The model should be fit and saved to a binary file, without evaluation or fitting code included. This file will be loaded by the R scoring script that is deployed.

There are two important considerations for the real-time scoring script that will be deployed. First, manual steps will not allow our model to be real time; so, we should remove manual steps. One of our models had a column in the training data that was created by an underwriter, using our underwriting manual. Obviously, this will not work for a real-time system. For this column, we had to devise a lookup table that accomplished a similar result as the underwriter. This resulted in inconsistencies between fitting and deployment with the underwriting column and this is not a recommended practice. Always look for manual steps ahead of time and think of how they will be addressed by a deployed real-time model. The second difference between a real-time model and fitting is that the real-time model will only process one row of data at a time. During model fitting the script might normalize columns by the standard deviation and mean of the entire training set. These kinds of statistical measures cannot be calculated for a single row. Such statistics, such as mean or standard deviation, must be calculated on the training set and then essentially hard-coded into the deployed R script that does the scoring.

Figure 1
https://docs.microsoft.com/en-us/r-server/deployr/deployr-about



WHAT DOES OUR MODEL NEED AS INPUT AND PROVIDE AS OUTPUT?

The recommended format for communication between models, model consumers and service provider is JavaScript Object Notation (JSON). Another common choice is eXtensible Markup Language (XML). As an example, consider a simple web service designed to predict the survival probability of a Titanic passenger (using the very popular Kaggle Titanic Dataset). The JSON model input could appear as:

```
{
 "class": 1,
 "gender": "female",
 "age": 35,
 "siblings": 1,
 "parents": 0,
 "fare": 57.5,
 "embarked": "S"
}
```

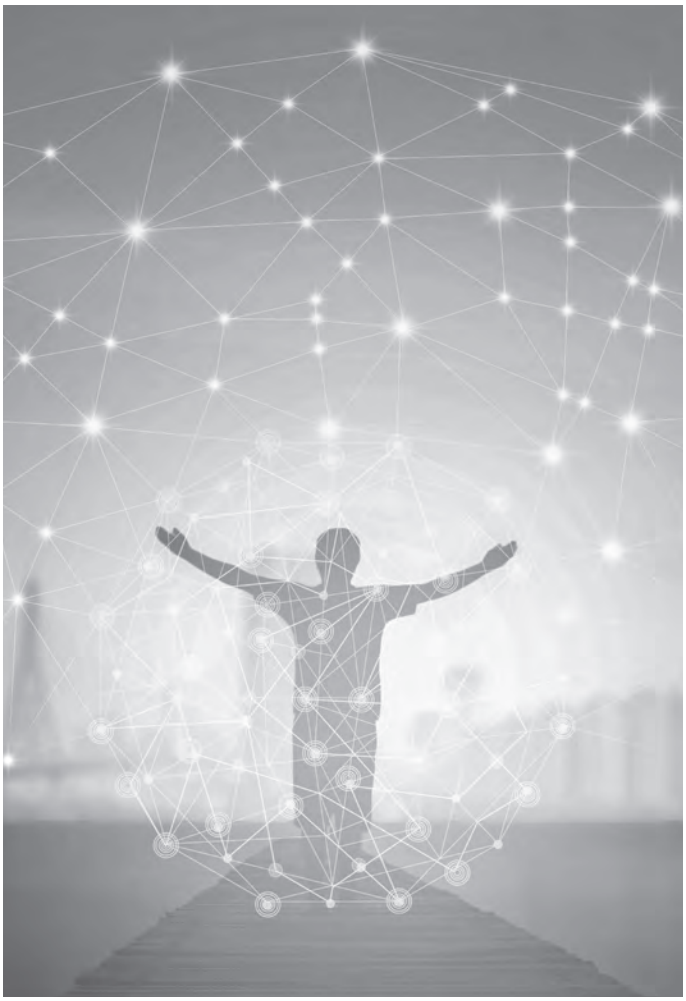
The above data will be transmitted to the deployed R script as JSON. We use the "jsonlite" library to parse this input format into individual variables for the model to use for prediction. The above JSON intentionally contains minimal personally identifiable information (PII). Only pass PII, such as name, address, date of birth, etc., if necessary. If PII is necessary one should include appropriate compliance and legal teams in the process.

The data that is sent by the client (JSON) is often quite different than the actual input to the model. Gender will probably be passed as M, F, U and be transformed into 0 or 1. The age might be transformed into a Z-Score, which will require knowledge of the mean and standard deviation of all ages in the training data. Similarly, the age and gender might together be used to lookup a value in one of the company's mortality tables. There are options for technologies to transform the high-level client data into the low-level model input. After all this is completed, the deployed R script will produce another JSON, such as the following:

```
{
 "date":"2017-08-19 17:18:14",
 "id":"4b495b7a-852c-11e7-9ef7-f7deab256915",
 "decision":"survive",
 "confidence":0.9026,
 "version":"titanic model v1.0 (build 1)"
}
```

ROBUSTNESS OF DEPLOYMENT SCRIPT

Once the model is trained, a simple script should be created that accepts a sample JSON file and produces the correct output.



This script will become the scoring R script that will be ultimately deployed and the robustness of this script is critical. One such area is to know how long the script takes to produce a single prediction. How long the script takes to execute is how long the client must wait for a single prediction. If the script takes more than a few seconds to run, this might be a problem. Another area to consider is how much memory the script needs to execute. We have seen models that will sometimes require the loading of several gigabytes of binary models to make a single prediction. When this is done the loading of this file may take up precious time, before predictions can even be made. If such complex models are truly required they can be preloaded into RAM. However, such a system's complexity is more difficult to implement and it decreases the ability to scale the model to many requests. Ideally, the deployed R script should take less than 10 seconds to execute. The following R code shows a sample scoring script for R that could be deployed:

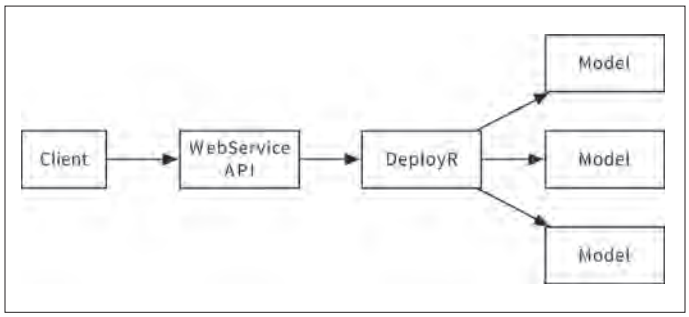
```
library(jsonlite)
library(uuid)

model_version <- 'titanic model v1.0 (build 1)'

json_data <- fromJSON(model_input)

# Extract only what we need from JSON
Age <- as.numeric(json_data['age'])
Sex <- toString(json_data['gender'])
Pclass <-  as.numeric(json_data['class'])
SibSp <- as.numeric(json_data['siblings'])
Parch <- as.numeric(json_data['parents'])
Fare <- as.double(json_data['fare'])
Embarked <- toString(json_data['embarked'])

# Load model and predict
inp <- data.frame(Age,Sex,Pclass,SibSp,Parch,
Fare,Embarked)
load(file=" titanic_glm.rdata")
pred <- predict(model,newdata=inp,type='response')

# Build response JSON's
l <- list()
l[['date']] <- Sys.time()
l[['id']] <- UUIDgenerate()

if (pred>0.5) {
  l[['decision']] <- "survive"
  l[['confidence']] <- as.numeric(pred)
} else {
  l[['decision']] <- "perish"
  l[['confidence']] <- as.numeric(1.0-pred)
}
l[['version']] <- model_version

model_response <- toJSON(l,auto_unbox=TRUE)
```

The above code has three main parts. First, the JSON is parsed from the variable model_input. Next, the model is loaded and the passenger is scored. Finally, the model output is encoded into the JSON response and is stored into the **model_output** variable. This code does not perform any validation. For a real system, validation is important and should generate an appropriate error response.

OPERATIONAL CONSIDERATIONS

Now that an operational model has exposed an API that will allow systems to communicate and integrate with, how does the API that the model has exposed get secured and accessible for others in the world to utilize so we can realize our data insights more broadly? There are several questions a team should ask and/or prove when trying to complete this objective, here are a few that we will cover in this paper to get to the finished version as seen below.

Figure 2
Target State Model



The target state depiction above shows how requests from the client are accepted by the WebService API routed through DeployR, which is an integration technology for deploying R analytics of one or more models that might be made available to clients. There are other technologies the team utilized to realize capabilities such as authorization, authentication, logging, monitoring, etc., yet we will not discuss those in this paper.

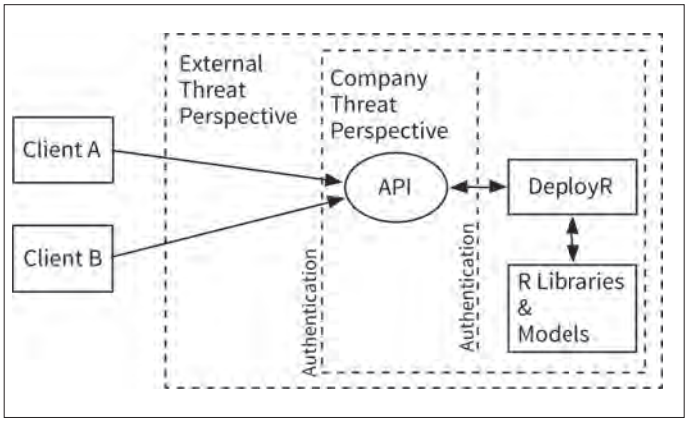
HOW DO WE INTEGRATE WITH DEPLOYR 8.0.5?

How we interact with the DeployR 8.0.5 API and how we efficiently spin up and spin down the DeployR model was a critical decision in achieving agreed upon service level commitments. In a real-time model, the processing should take seconds for the response(s); and starting up DeployR and how the data gets posted to the model might take up precious time that could be used for the calculation. DeployR is a batch oriented system, so how do we take these individual calls and work with them? An analogy of how DeployR works is how an airplane operates, whether it carries one passenger or 100 passengers it takes the same amount of time to complete. What we had to do was determine how we could setup a collection of these projects that would fill an itinerary for the plane then send on to DeployR for execution. As a practice, stateful services is almost always seen as an anti-pattern, yet with this version of DeployR there was no good way to complete the operation in a performant way using a stateless service given that the DeployR would have to spin up to complete the operation for each call. For this reason and the performance requirement, we needed to figure out how to complete these operations in a couple seconds. The method that was completed was a project queue for the requests and responses. In a future article, we will describe more technical details of this process.

HOW DO I SECURE THE INTELLECTUAL PROPERTY OF THE MODEL?

In the age of data insights, the one with the best algorithm wins; so, securing those algorithms or models is of utmost importance. First thing we need to realize is with unlimited time and budget someone could compromise what we are trying to protect. Security is not about whether the feat is impossible to complete, but more of how much time and money is needed to be able to get what you are trying to protect without being detected. That is why with any project, especially one exposed publicly, we should take a step back and understand how a potential attacker would compromise our system and mitigate appropriately to the risk and exposure. A simple technique to use when going through this exercise is "threat modeling," which is "a procedure for optimizing Network/Application/Internet Security by identifying objectives and vulnerabilities, and then defining countermeasures to prevent, or mitigate the effects of, threats to the system."[1]

Figure 3
Threat Model



As seen from Figure 2, we identified three principle attack surfaces we needed to mitigate or prevent the attack. We will only focus on the last one as this is the most jermaine to the article. The scenario we will discuss is when a role that has access to the models decides to put in a threat or divulge that sensitive information.

Access control lists that are reviewed and approved regularly start the security. All code for the library is stored in a source control system that does builds and verifications, so if someone does decide to merge in a vulnerability, the tests will catch the issue before it gets deployed. The other item is how this model is promoted to DeployR8.0.5 which we did a fair amount of research on and decided to compile the libraries with the critical algorithms for use in the model, much like a local CRAN mirror would do for us. What this does is, even if someone has access control they can only see a compiled version of the library and would take some time to understand how to decompile. The final strategy is robust monitoring and logging. The naysayers will tell you that robust monitoring is over engineering, yet when an attack is occurring this monitoring can help the team understand the attack is occurring and allow precious time to find the holes they used in the threat model. The monitoring needs to understand that this attack is occurring and lock the doors to our intellectual property. Which is why monitoring and logging needs to be a discussion the team has as this will give the team all the context available so a game time decision can be made.

This paper explored moving modeling capabilities in R to real time operations. The age of data insights will continue to evolve and the methods at which we analyze the data and base our predictions will change, but having those insights faster and in varied ways will not. Like anything else, decisions are relative to the situation at hand. And while we focused on answering a subset of questions, we would expect the team to understand all requirements as the service is operationalized and there may be many more questions that could and should be considered. ■

ACKNOWLEDGEMENT

As always a project takes a team to complete and we wouldn't have been able to complete this project without the help of RGA Automation and Monitoring, Global Research, Development and Analytics, Actuarial Solutions and Underwriting teams, and last but not least, Larry Anderson from Ocelot Consulting (*www.ocelotconsulting.com*) for the wonderful work on the delivery of this service.

Jeff Heaton, Ph.D., is lead data scientist, Reinsurance Group of America (RGA) in Chesterfield, Mo. He can be reached at *jheaton@rgare.com.*

Edmond Deuser is technical architect and developer, Reinsurance Group of America (RGA) in Chesterfield, Mo. He can be reached at *edeuser@rgare.com*

REFERENCE

Threat modeling 1 – *https://www.owasp.org/index.php/Application_Threat_Modeling*

ONLINE RESOURCE

Swagger Hub API - *https://swaggerhub.com/apis/RGA/ROperational/1.0.0*