Paper AS04

The Art of Work Flowing Your Clinical Submission Scripts

Brian Varney, Experis, Kalamazoo, USA
Kevin Putschko, Experis, Ann Arbor, USA

ABSTRACT

As more subject data flows into a clinical database, series of programs in development will likely need to be run repeatedly to keep an eye on table, listing, and figure results. For many years, pharmaceutical companies have been using SAS® functionality of either driver programs and/or SAS Enterprise Guide® project flows. In R, there are a few options that parallel what has traditionally been performed in SAS. This paper intends to explore how to make use of R and RStudio to assist with the management of driver programs, as well as alternative project workflow methodologies. We will compare the benefits of using a typical driver script with the "make" style of programming using the 'targets' package. In addition, we will explore the best practices when developing a workflow that is intended only for yourself, and a workflow that is intended to be shared across a wide group of users.

INTRODUCTION

This paper is for developers in R that would like to leverage methods for constructing clinical project workflows to run a series of programs in serial and sometimes parallel processes. We will begin by examining typical driver programs using SAS Software and then show how to do similar processes in RStudio and R. Some prior knowledge of SAS, RStudio, and R would be helpful to understand the concepts in this paper.

While we may mention SAS Enterprise Guide in this paper, our examples will be Base SAS program code.

USING THE SAS %INCLUDE()

This is an excerpt from an SDTM data set build driver program typically used in SAS. Individual build programs are included after setting global SAS Macro variables and program options.

```
%let study = ABC;

options <options>;

%include(build_sdtm_dm.sas);
%include(build_sdtm_ae.sas);
%include(build_sdtm_vs.sas);
```

USING THE R SOURCE() FUNCTION

The R source() function in R is similar to using the %include() functionality in SAS Software. Using a series of R source() functions will allow us to run a series of R programs in a serial manner. An excerpt from an SDTM data frame build driver program used in R follows.

```
study = ABC

source(build_sdtm_dm.R)
source(build_sdtm_ae.R)
source(build_sdtm_vs.R)
```

The remainder of this paper demonstrates how to use the targets package to allow us to set up a series of R processes than can run in serial and parallel while allowing us to define dependencies. Defining these dependencies allows us to only run the processes that need to be run based on what has been updated earlier in the pipeline.

USING A 'TARGETS' PIPELINE

The *targets* package is a set of functions that assists in the creation of an asynchronous pipeline that runs operations only when the source code or the incoming data has been updated and skips operations when they don't need to be

run. This package follows the *make* approach of structuring a pipeline, and is the successor to the *drake* package in R.

THE 'TARGETS' PHILOSOPHY

The *targets* package, and *make* style pipelines in general, uses a series of targets and expressions to define input and output throughout the pipeline. Each target is a single incremental step in the overall pipeline structure. The idea behind these target steps is to make them big enough to save a considerable amount of time when they can be skipped, but small enough to contain only the necessary operations for that specific step.

USING THE 'TARGETS' PACAKGE

There are functions that assist with the creation of the targets pipeline, and functions that assist with the management of the pipeline. A straightforward way to setup and maintain a targets pipeline is to use two separate scripts: one to define the pipeline configuration, and one to execute and investigate the pipeline's results.

In the pipeline configuration script, you will need to load the targets package, along with other packages that are useful in defining the structure of the pipeline. It is entirely possible to only load the targets package in this step, even though you are using many other packages and functions inside the pipeline itself. Next, use the *tar_script()* function to create the steps / targets in the actual pipeline. This function will create the two files *_targets.R* and *_targets.yaml*, which contain the definitions and instructions of the pipeline. It is not recommended to modify these two files directly.

```
library(targets)
tar_script(code = {…})
```

Inside the *code* argument of the *tar_script()*, we load the packages and functions that will be required *inside* the pipeline execution, not including the targets package itself. We can use the *source* function to ensure our user-defined functions are available inside the pipeline, and we must use *tar_option_set(packages = c())* to define the packages we want to use inside the pipeline operations.

```
library(targets)
tar_script(code = {
       source("user_defined/functions.R")
       tar_option_set(packages = c("…", "…"))
})
```

Then, we can finally begin defining the target steps of the pipeline using *tar_targets(),* and wrapping them in a *list()*. The *name* argument contains the unquoted name of the object you are creating in the *command* argument. This is like using the assignment operator in typical R syntax, `` `name <- command` ``, where the *command* argument can be any arbitrary R code with references to names defined in other *tar_target()* definitions.

```
library(targets)
tar_script(code = {
       source("user_defined/functions.R")
       tar_option_set(packages = c("…", "…"))

       list(  tar_target(name = …, command = …),
              tar_target(name = …, command = …),
              tar_target(name = …, command = …))
})
```

Once we run this script, and the corresponding *_targets.R* file is created, we can begin using the pipeline maintenance functions. As suggested earlier, it is best to use these functions in a separate script from the *tar_script()* definition. To check the pipeline configuration for any issues, we can use *tar_validate()*. To see the pipeline in a table format, we can use *tar_manifest()*; to see the pipeline in a visual format, we use *tar_glimpse()*. To run the pipeline, we use *tar_make();* and to view a visual status of each target step, we can use either *tar_progress_summary()* or *tar_visnetwork().* It is worth noting that the *tar_visnetwork()* graph will display the target steps that are either up to date and can be skipped, or out of date and should be run. Finally, to see an overview of each step, its runtime, errors, messages, and more, we can use *tar_meta();* while *tar_load()* will actually load the results of the target step into memory so that we can inspect it interactively.

USE CASE: CREATING A PIPELINE

Now that we have provided a brief overview of the targets package and workflow, we will create a sample target pipeline for demonstration purposes.

In this pipeline, we're going read in a raw data frame, prepare the data for an analysis, create summary tables and figures, and finally build a linear model. Notice that we are loading a separate script with our function definitions using the *source()* function, and then we are using those functions throughout the target step definitions.

```
# Load Functions to Help Define the Pipeline
library(targets)  # To use tar_script, tar_option_set, and tar_target
library(dplyr)    # To use %>% in command definitions if desired


## Define the Pipeline ----
targets::tar_script(      # Create '_targets.R' file in project/working directory
  ask = FALSE,            # Set 'ask' to TRUE to confirm overwrite of existing file
  code = {                # Use {...} to wrap many operations in the pipeline


    # -- Load Packages & Personal Functions --
    # These are the packages and functions that will be required in the pipeline
    tar_option_set(packages = c("dplyr", "ggplot2", "stringr", "readr",
                                "janitor", "broom", "tibble"))

    source("code/pipeline_fns.R")


    # -- Create a Pipeline Definition with `list()` --
    # Use a list() to hold the`tar_target()` pipeline steps
    # 'Name' gives the target an informative name so we know what the process is doing
    # 'Command' is the R code we want to run at this step
    # 'Format' means that the pipeline will monitor this file for changes
    list(

      # Specifying format = "file" can monitor any type of file for changes
      tar_target(
        name    = monitor_adam_adsl,
        command = "data/ADaM/adsl.rds",
        format  = "file"),

      tar_target(
        name    = monitor_sdtm_ae,
        command = "data/SDTM/ae.rds",
        format  = "file"),

      # Load the Data
      tar_target(
        name    = read_adam_adsl,
        command = readRDS(monitor_adam_adsl)
      ),

      tar_target(
        name    = read_sdtm_ae,
        command = monitor_sdtm_ae %>% readRDS()
      ),

      # The 'command' argument accepts any amount of R code, here we begin using
      # our personal functions
```

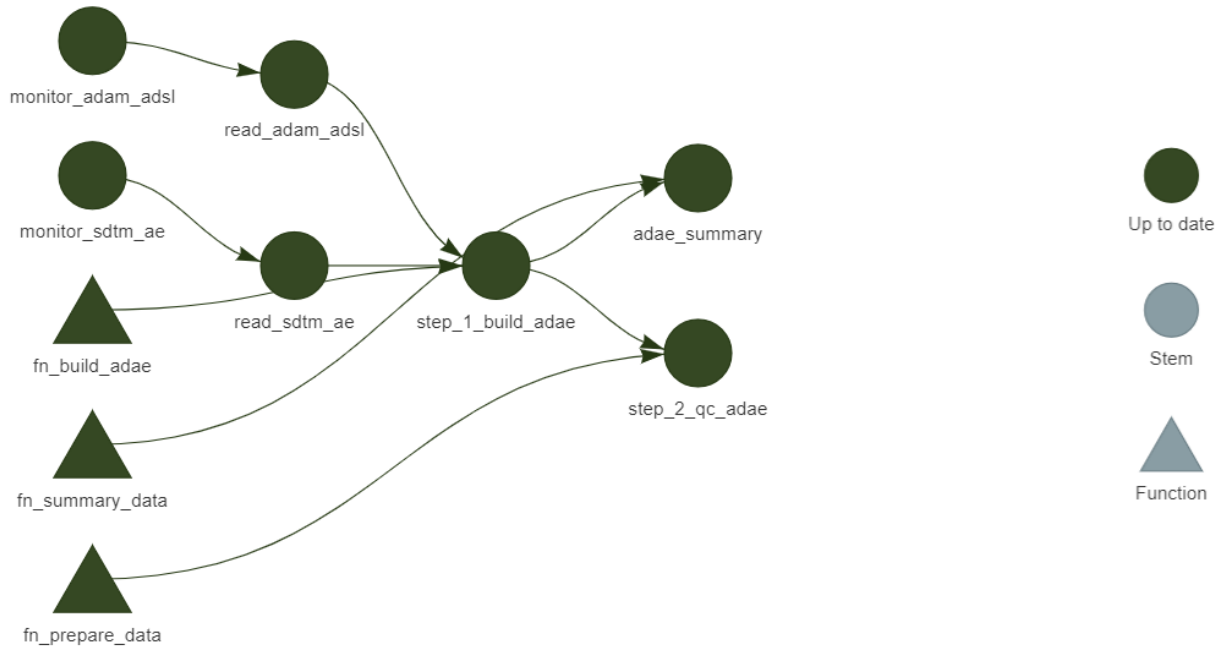
```
    tar_target(
      name    = step_1_build_adae,
      command = fn_build_adae(read_adam_adsl, read_sdtm_ae)),


    # But using custom functions will make it easier to debug the pipeline
    tar_target(
      name    = step_2_qc_adae,
      command = fn_prepare_data(step_1_build_adae)),


    # No need to keep track of sequence order, 'targets' will track it for us
    # Notice the following steps are not in sequential order:
    tar_target(name = adae_summary,  fn_summary_data(step_1_build_adae))
  )
})
```

USE CASE: MAINTAINING A PIPELINE

After creating this pipeline, we can run it using *tar_make()*, and we can investigate this pipeline visually using *tar_visnetwork()*. We provide a snapshot of this pipeline below. You can see the defined target steps displayed as circles and functions displayed as triangles; further, blue symbols are outdated and will be run the next time we use *tar_make()*, and dark-green symbols are up to date and will not be run.



When we want to see a summary of the most recent pipeline run, we can use *tar_meta()*. This displays a table that shows various metadata for each target step, include any errors or warnings that occur.

After identifying a warning or error that we want to inspect, we can include the name of the target like so, *tar_make(step_3_preparedata, callr_function = NULL)*. This will ensure that the specified step runs, with its prior dependencies, in the interactive R environment rather than running as a background job, allowing us to step through the operation in a typical fashion.

Regarding parallelization of targets pipelines, we won't get into the details here, but simply including the line of code in the *tar_script()* definition will get you started with a simple parallelization plan. This uses the 'future' package inside the targets pipeline to parallelize the operations when it can.

```
future::plan(future::multisession)
```

CONCLUSION

The *targets* package offers a simplified method of managing complex data pipelines. Although there is a learning curve in building your operations around functions, and defining appropriate target steps, the time is well invested in allowing you to better maintain up-to-date results without having to run the entire pipeline.

.

CONTACT INFORMATION

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

Author Name: Brian Varney
Company: Experis
Email: brian.varney@experis.com

Author Name: Kevin Putschko
Company: Experis
Email: kevin.putschko@experis.com