Living HEOR, Automating HTA with R

Robert A Smith[1,2,3*], Paul P Schneider[2,3], and Wael Mohammed[2,3]

[1]**Lumanity, Sheffield**
[2]**Dark Peak Analytics, Sheffield**
[3]**School of Health and Related Research, University of Sheffield**

Robert A Smith: rasmith3@sheffield.ac.uk
Paul P Schneider:
Wael Mohammed:
* Corresponding author

Abstract **Background** The process of updating economic models is time-consuming and expensive, and often involves the transfer of sensitive data between parties. Here, we demonstrate how HEOR can be conducted in a way that allows clients to retain full control of their data, while automating reporting as new information becomes available.
Method We developed an automated analysis and reporting pipeline for health economic modelling and made the source code openly available on a GitHub repository. It consists of three parts: An economic model is constructed by the consultant using pseudo data (i.e. random data, which has the same format as the real data). On the client side, an application programming interface (API), generated using the R package plumber, is hosted on a server. An automated workflow is created. This workflow sends the economic model to the client API. The model is then run within the client server. The results are sent back to the consultant, and a (PDF) report is automatically generated using RMarkdown. This API hosts all sensitive data, so that data does not have to be provided to the consultant.
Results & Discussion The method is relatively complex, and requires a strong understanding of R, APIs, RMarkdown and GitHub Actions. However, the end result is a process, which allows the consultant to conduct health economic (or any other) analyses on client data, without having direct access – the client does not need to share their sensitive data. The workflow can be scheduled to run at defined time points (e.g. monthly), or when triggered by an event (e.g. an update to the underlying data or model code). Results are generated automatically and wrapped into a full report. Documents no longer need to be revised manually.
Conclusions This example demonstrates that it is possible, within a HEOR setting, to separate the health economic model from the data, and automate the main steps of the analysis pipeline. We believe this is the first application of this procedure for a HEOR project.

Keywords
HEOR, HTA, APIs, R, Plumber.

Introduction

The process of updating economic models is time-consuming and expensive, and often involves the transfer of sensitive data between parties. This paper aims to demonstrate how updates to models in the Health Economics & Outcomes Research (HEOR) industry can be conducted in a way that allows clients, primarily those in the pharmaceutical industry, to retain full control of their data. We continue on to show how to automate reporting as new information becomes available, without the transfer of data between parties.

A previous publication by Adibi et al. [1] made the case for cloud based platforms to improve the accessibility, transparency and standardization of health economic models, particularly highlighting the benefits of hosting computationally burdensome models on remote servers. However, to our knowledge this is the first publication to demonstrate the feasibility of the use of APIs to avoid the need for companies to share sensitive data, and also the first to provide open source code for the semi-automation of model updates in health economics.

Method

We developed an automated analysis and reporting pipeline for health economic modelling and made the source code openly available on a GitHub repository. It consists of three parts:

- An economic model is constructed by the consultant using pseudo data (i.e. random data, which has the same format as the real data).
- On the company side, an application programming interface (API), generated using the R package plumber, is hosted on a server. An automated workflow is created. This workflow sends the economic model to the company API. The model is then run within the company server. The results are sent back to the consultant, and a (PDF) report is automatically generated using RMarkdown.
- This API hosts all sensitive data, so that data does not have to be provided to the consultant.
- All of these processes can be controlled through an RShiny app, based on the tutorial application in our previous paper [2].

All of the scripts discussed in this paper, as well as the code for the demonstration app can be found contained within an open access GitHub repository.

The model code

This model code has been amended from the DARTH group's open source Cohort state-transition model (the Sick-Sicker Model) which can be found in this GitHub repository and is discussed in Alarid-Escudero et al. [3]. The code includes several functions, but for the purpose of this example we can treat the model as a black box, as a single function called *run_model* which runs the DARTH Sick Sicker model. The *run_model* function takes a single argument, *psa_inputs*, which is a data-frame containing Probabilistic Sensitivity Analysis parameter inputs for the model variables that are allowed to vary.

The data-frame has four columns: * *parameter* - the name of the parameter (e.g. p_HS1) * *distribution* - the distribution of that parameter (e.g. "beta") * *V1* - the first parameter for the distribution in R (for beta this would be shape1, for normal this would be mean) * *V2* - the second parameter for the distribution in R (for beta this would be shape2, for normal this would be sd)

The *run_model* function returns a data-frame with six columns and a thousand rows. Each row is the result of the model run for a random draw from the PSA inputs. The first three columns are costs for each treatment option, and the second three columns are QALY for each treatment option.

The full model code is available on GitHub here, however it is not important to understand this specific example model, therefore we move on to describe the creation of the API.

The API

An application programming interface is a set of rules, in the form of code, that allow different computers to interact with one another in real time. Whereas user-interfaces such as those generated by *shiny* allow humans to interact with data, APIs are designed to enable computers to interact with data.

When a 'client' application wants to access data, it initiates an API call (*request*) via a web-server, to retrieve the data. If this request is deemed valid, the API makes a call to an external program/server, the server sends a response to the API with the data, and the API transfers the data to the 'client' application. In a sense, the API is the broker (or middle-man) between two systems.

There are numerous benefits to APIs: - in aiding speed of collaboration between institutions, ensuring inputs and outputs are standardised so that applications can 'talk' to one another. - in security, eliminating the necessity to share data manually (e.g. via email). All interaction with data can be logged and access can be restricted by passwords and by limiting IP address access. - eliminating computational burden on the client side (since all computation is done on the API owner side.)

There are lots of different implementations of APIs, but the main focus of this paper is on **Partner APIs**, which are created to allow data transfer between two different institutions. This requires a medium level of security, usually through the creation of login keys that are shared with partners.

In the examples below we use JSON to pass information to and from our API.

Plumber The R package plumber allows programmers to create web APIs by decorating R source code with roxygen-style comments. The source code looks like a series of functions, which are not assigned to objects, with roxygen style comments specifying the parameters input to the function, the name of the function, and the output

from the function [4, R2021citation]. These functions are then made available as API endpoints by plumber.
The code below give an example function which echos a message. The function takes one input, a string with the message, and outputs the message contained within a list. If this function was created in R it would return a list containing some text, like this: The message is: 'example_msg'.
The API can be called using any type of request, the below shows and example of the 'GET' request (the default for web-browsers).

```
#* Echo back the input
#* @param msg The message to echo
#* @get /echo
function(msg="") {
  list(msg = paste0("The message is: '", msg,
}
```

The code for the model function uses the same principles, but is much more developed. There are three parameter inputs:

- *path_to_psa_inputs*
- *model_functions*
- *param_updates*

The function sources the model functions from GitHub, obtains the model parameter data from within the API, and then overwrites the rows of the parameter updates that exist in *param_updates*. It then runs the model functions using the updated parameters, post-processes the results, checks that no sensitive data is included in the results, and then returns a data-frame of results. This entire process occurs on the server on which the API is hosted, with inputs and outputs passed to the API over the web in JSON format.

```
##################

library(dampack)
library(readr)
library(assertthat)

#* @apiTitle Client API hosting sensitive dat
#*
#* @apiDescription This API contains sensitiv
#* want to share this data but does want a co
#* economic model using it, and wants that co
#* the model for various inputs
#* (while holding certain inputs fixed and le

#* Run the DARTH model
#* @serializer csv
#* @param path_to_psa_inputs is the path of the csv
#* @param model_functions gives the github repo to source the model code
#* @param param_updates gives the parameter updates to be run
#* @post /runDARTHmodel
function(path_to_psa_inputs = "parameter_distributions.csv",
         model_functions = paste0("https://raw.githubusercontent.com/",
                                  "BresMed/plumberHE
         param_updates = data.frame(
           parameter = c("p_HS1", "p_S1H"),
           distribution = c("beta", "beta"),
           v1 = c(25, 50),
           v2 = c(150, 70)
         )) {


  # source the model functions from the shared GitHu
  source(model_functions)

  # read in the csv containing parameter inputs
  psa_inputs <- as.data.frame(readr::read_csv(path_t

  # for each row of the data-frame containing the va
  for(n in 1:nrow(param_updates)){

  # update parameters from API input
  psa_inputs <- overwrite_parameter_value(
                              existing_df = psa_inputs
                              parameter = param_update
                              distribution = param_upd
                              v1 = param_updates[n,"v1
                              v2 = param_updates[n,"v2
  }

  # run the model using the single run-model functio
  results <- run_model(psa_inputs)

  # check that the model results being returned are
  # here we expect a single dataframe with 6 columns
  assertthat::assert_that(
    all(dim(x = results) == c(1000, 6)),
    class(results) == "data.frame",
    msg = "Dimensions or type of data are incorrect,
  please check the model code is correct or contact
  This has been logged"
  )

  # check that no data matching the sensitive csv da
  # searches through the results data-frame for any
  # if any exist they will flag a TRUE, therefore we
  assertthat::assert_that(all(psa_inputs[, 1] %in%
        as.character(unlist(x = results,
                              recursive = T)) == F))

  return(results)

}
```

Deploying an API In this example we have deployed the API on RStudio Connect. An account is required for this, but once you have one it is possible to deploy the API direct to RStudio connect from the Rstudio IDE. RStudio have a blog on how to publish an API created using Plumber to RStudio connect here. There are numerous other providers of cloud computing services, many at cheaper prices, but no others with such ease of deploy-

ment from RStudio.

Automating the model run

The model run can be automated. We first show how to run the model from an R script, calling the API. We then continue to show how to use GitHub actions to automate the process.

Interact with API from RScript We use the *POST* function from the *httr* package to query the API [5] - as shown in the code chunk below. This function requires an internet connection. We provide values for several arguments:

- *url* - the URL of the RStudio Connect server hosting the API we have created using plumber.
- *path* - the path to the API within the server URL.
- *query* & *body* - objects passed to the API in list format, with names matching the plumber function arguments.
- *config* - allows the user to specify the KEY needed to access the API.

The *content* function attempts to determine the correct format for the output from the API based upon the content type. This function ensures that the results object is a dataframe.
The script then then goes on to save the data and generate a PDF report from the outputs using the RMarkdown package [6], the code for which can be found here. The markdown report uses functions adapted from the *darkpeak* R package.

```
# remove all existing data from the environment
rm(list = ls())

library(ggplot2)
library(jsonlite)
library(httr)

# run the model using the connect server API
results <- httr::content(
  httr::POST(
    # the Server URL can also be kept confide
    url = "https://connect.bresmed.com",
    # path for the API within the server URL
    path = "rhta2022/runDARTHmodel",
    # code is passed to the client API from G
    query = list(model_functions =
                   paste0("https://raw.github
                          "BresMed/plumberHE/
    # set of parameters to be changed ...
    # we are allowed to change these but not
    body = list(
      param_updates = jsonlite::toJSON(
        data.frame(parameter = c("p_HS1","p_S
                   distribution = c("beta","b
                   v1 = c(25, 50),
                   v2 = c(150, 100))
      )
    ),
    # we include a key here to access the API ... li
    config = httr::add_headers(Authorization = paste

  )
)

# write the results as a csv to the outputs folder.
write.csv(x = results,
          file = "outputs/darth_model_results.csv")

source("report/makeCEAC.R")
source("report/makeCEPlane.R")

# render the markdown document from the report folde
# passing the results dataframe to the report.
rmarkdown::render(input = "report/darthreport.Rmd",
                  params = list("df_results" = resul
                  output_dir = "outputs")
```

Use GitHub actions to automate the process Once the API is created and hosted online, it can be called any time. The advantage of this is that any updates to either the model code, or the data used by the model, can be undertaken separately and the model re-run by either party. Calls to the API can also be scheduled at routine intervals. This would enable the health economic evaluation model report to be updated, without human interaction, at regular intervals to reflect the most up-to-date data.
In the example below we show how a GitHub Actions (other providers available) workflow can be used to automate an update to a health economic evaluation [7]. The workflow runs at 0:01AM on the first day of every month. It first clones the GitHub repository on a GitHub actions Windows 2019 server, then installs the necessary dependencies, before running the script described above to generate the model report. It creates a pull request to the repo with this new updated report. If GitHub is not the preferred location of report storage, it is possible to send the report via email or save to cloud storage solutions such as Google Drive or Dropbox.

```
on:
  push:
    branches:
    - main
  schedule:
    - cron: '1 1 1 * *'

name: Run DARTH model on client API
jobs:
  createPullRequest:
    runs-on: windows-2019
    env:
      GITHUB_PAT: ${{ secrets.GITHUB_TOKEN }}
  # Load repo and install R
    steps:
    - uses: actions/checkout@master
    - uses: r-lib/actions/setup-r@master
```

```
    - name: Setup pandoc
      uses: r-lib/actions/setup-pandoc@v2
      with:
        pandoc-version: '2.17.1.1'

    - name: Install TinyTeX
      uses: r-lib/actions/setup-tinytex@v2
      env:
          # install full prebuilt version
          TINYTEX_INSTALLER: TinyTeX

    - name: Install dependencies
      run: |
          install.packages(
          c("reshape2","jsonlite","httr", "readr", "rmarkdown", "markdown")
          )
          install.packages(
          "scales", dependencies = TRUE, repos = 'http://cran.rstudio.com/'
          )
          install.packages(
          "ggplot2", dependencies = TRUE, repos = 'http://cran.rstudio.com/'
          )
      shell: Rscript {0}

    - name: Run the model from API and create report
      env:
         CONNECT_KEY: ${{secrets.PLUMBER_SECRET}}
      run: |
          source("scripts/run_darthAPI.R")
      shell: Rscript {0}

    - name: Create Pull Request
      uses: peter-evans/create-pull-request@v3
      with:
        token: ${{ secrets.GITHUB_TOKEN }}
        commit-message: Automated Model Run from API
        title: 'Living HTA Automated Model Run'
        body: >
          Automated model run
        labels: report, automated pr
```

Discussion

As the collection & storage of large data sets has become more commonplace in health & health care settings, this data is increasingly being used to inform decision making. However, concerns about the security of this data, and the ethical implications about linked data sets, make the owners of this valuable resource particularly reluctant to share data with health economic modeling teams. The ability to host APIs on data-owners' servers, and send the model to the data rather than the data to the model, is one potential solution to this problem. The example described in this paper may be relatively simple, but gives a tech savvy health economist everything they need to set up a modelling framework which does not rely on the sharing of data by a company (or other data-owner).
The framework described have a number of benefits.

- Firstly, no data needs to leave the data-owner's server. This is likely to significantly reduce administrative burden for both the company and the consultant, and reduce the number of data-leaks.

- Separating the model code from the data can significantly improve the transparency of the health economic model. Allowing others to critique methods & hidden structural assumptions, test the code and identify bugs should improve the quality of models in the long run.

- The computational burden of the model is handled on a remote server. The power of these servers is considerably greater than that of a typical laptop, speeding up model run time considerably.

- API calls can be made at any time, and will always reflect the data held by the company. In many cases these datasets are updated regularly, allowing companies, and other stakeholders, to see the results of the decision model based on the most up to date data, without needing human intervention to: send new datasets, re-run analysis, write a report, and provide that report in a suitable format for the company. Automating model updates at set schedules, or when data is updated, may be invaluable where data is updated regularly, as has been the case throughout the COVID-19 pandemic.

However, the framework has a number of limitations:

- Firstly, the method is relatively complex, and requires a strong understanding of health economic modeling in R, API creation and hosting, RMarkdown or other automated reporting packages, and GitHub Actions. While we hope that this paper provides a useful resource to health economists seeking to utilise these methods, the bulk of the industry still operates in MS Excel. Providing tuition to upskill health economists, or creating teams consisting of both health economists and data-scientists & software engineers may mediate this limitation somewhat. The R for HTA consortium has the potential to play a crucial role in upskilling the industry.

- There are still likely to be concerns about data security, even with the authentication procedures built in to the API functionality. Collaboration with experts in this field may mediate this significantly, since there is no fundamental reason why health data is any more sensitive, or vulnerable, than the plethera of other data (including banking data) that relies on APIs every day. It will be important to reassure companies that the use of APIs is likely to reduce, not increase the risk of data breaches, and that every interaction with the data can be logged.

- There is a risk that running the model remotely will result in the perception that the model is a 'black box'. The use of user-interfaces (such as those increasingly being created in shiny) to interrogate the

model, as well as the increased transparency associated with being able to share code on sites such as GitHub, should reassure stakeholders that this framework is more transparent than the existing spreadsheet based solutions.

- Often, when building a model, it is helpful to have the underlying data to be able to investigate the data, often through the generation of descriptive statistics. The process of sharing pysudo-data enables modelers to ensure that the models they create conform to the structure of the data input. However, the modeler still needs to be able to write code that is versatile enough to cope with data with unknown distributions & ranges. Code that will not break when the number of observations changes, or when the range or distribution of a variable changes. This is easily solved, again by improved training and the use of standard packages.

A recent working paper by Adibi et al. [1] has provided a similar call to action, extolling the virtues of the API for decision modeling, and showing how APIs can be used to shift much of the computational burden away from key stakeholders & make models more accessible. This paper goes one step further, providing open source code for the creation and deployment of an API with an accompanying automated health economic evaluation update framework. It also provides clearly described open source code on two new pieces of additional functionality not previously described elsewhere; firstly it demonstrates how companies can host APIs themselves to negate the need to share data with subject experts, and secondly it demonstrates how model updates can be automated with GitHUb actions.

Conclusions

This example framework, with accompanying open source code base, demonstrates that it is possible, within a HEOR setting, to separate the health economic model from the data, and automate the main steps of the analysis pipeline. We believe this is the first application of this procedure for a HEOR project, and is certainly the first example to be made open source for the benefit of the wider community. We hope that this framework will improve the transparency of health economic models, reduce the cost & administrative burden of updating models, and increase the speed at which updates can occur.

Author contributions

R.S. & P.S. developed the original concept, R.S. wrote the source code & the original article. W.M. & P.S. reviewed both the source code and article. All errors are the fault of R.S.

Competing interests

R.S., P.S. & W.M have no competing interests to declare.

Grant information

R.S., P.S. & W.M. are joint funded by the Wellcome Trust Doctoral Training Centre in Public Health Economics and Decision Science [108903] and the University of Sheffield.
The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Software availability

Source code is available on GitHub at **Insert Final GitHub repo Link**.
Archived source code at the time of publication can be found at **Insert Zenodo Link**.
Licence: MIT

Acknowledgements

We would like to thank the participants at R-HTA Oxford, Richard Birnie and Dawn Lee for feedback on the original concept and manuscript. All errors are the fault of the authors.

References

[1] Amin Adibi, Stephanie Harvard, and Mohsen Sadatsafavi. Programmable interface for statistical & simulation models (prism): Towards greater accessibility of clinical and healthcare decision models. *arXiv preprint arXiv:2202.08358*, 2022.

[2] Robert Smith and Paul Schneider. Making health economic models shiny: A tutorial. *Wellcome Open Research*, 5, 2020.

[3] Fernando Alarid-Escudero, Eline Krijkamp, Eva A Enns, Alan Yang, Myriam Hunink, Petros Pechlivanoglou, and Hawre Jalal. Cohort state-transition models in r: A tutorial. *arXiv preprint arXiv:2001.07824*, 2020.

[4] Barret Schloerke and Jeff Allen. *plumber: An API Generator for R*, 2021. URL `https://CRAN.R-project.org/package=plumber`. R package version 1.1.0.

[5] Hadley Wickham. *httr: Tools for Working with URLs and HTTP*, 2020. URL `https://CRAN.R-project.org/package=httr`. R package version 1.4.2.

[6] Yihui Xie, Christophe Dervieux, and Emily Riederer. *R Markdown Cookbook*. Chapman and Hall/CRC, Boca Raton, Florida, 2020. URL `https://bookdown.org/yihui/rmarkdown-cookbook`. ISBN 9780367563837.

[7] Chaminda Chandrasekara and Pushpa Herath. Introduction to github actions. In *Hands-on GitHub Actions*, pages 1–8. Springer, 2021.

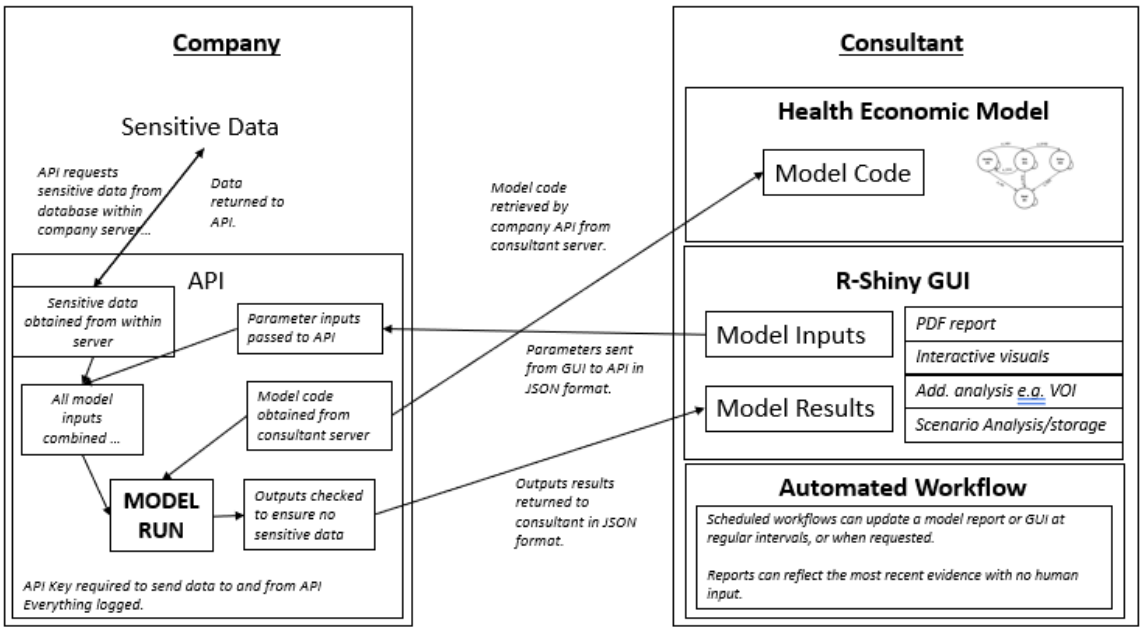

Figure 1. Schematic showing the interaction between the Company API and the Consultant Automated Workflow