Data Science Worklows in R

An introduction to deploying production quality R code

Dean Marchiori

2024-01-01

Table of contents

Preface

This book is a resource for data analysts and data scientists looking to improve the way the way they write R code. While R remains a popular choice for statistical modelling and data analysis, its rapid development has enabled users to progress their work right through to being deployed into Production. However the type of work done when conducting experiments and developing models is very different to packaging up this work so it can reliably drive decisions in an organisation. This book will provide readers with an overview of contemporary frameworks for how data analysis is done in practice. It will cover how R projects are usually structures and how this can evolve based on project complexity. It will examine what is meant by experimental vs production analysis code and which principles need to be adopted. Finally it will show current tools and frameworks for taking experimental R code and strengthening it to align with best practice for reliable production grade software. Readers can step through a case study and download code to follow along.

Who is this for?

This book is intended as an introductory guide for R users who have experience writing code and fitting models, but want to improve their practices for translating these models into robust code that is reliable and used to make real-world decisions.

This was initially developed as course materials for the tutorial "Turning R code into production quality software" at the 2024 *Workshop Organised by the Monash Business Analytics Team (WOMBAT)*.

Terminology

Throughout this book the terms 'data science' and 'data analysis' should be considered interchangable and represent the application of advanced data analysis and statistical techniques to data to achieve an outcome. The word 'analyst' will be adopted as the primary role for someone completing these tasks.

Contact Me

If you would like to get in touch head over to deanmarchiori.com

Contributing

Contributions to this work are welcomed via Issues on the Github page.

Please note that this project uses a Contributor Code of Conduct. By contributing to this book, you agree to abide by its terms.

Licence

1 Setup

To get started, ensure you have a recent verison or R and RStudio installed.

1.1 Software

1.1.1 Install R

To install R head to https://cran.rstudio.com/.

1.1.2 Install RStudio

Next, install RStudio Desktop IDE at https://posit.co/download/rstudio-desktop/

1.1.3 Packages

To install the required packages, run:

```
install.packages("tidyverse")
install.packages("mgcv")
install.packages("pins")
install.packages("vetiver")
install.packages("plumber")
install.packages("here")
```

You should be able to now run the following commands:

```
library(tidyverse)
library(mgcv)
library(pins)
library(vetiver)
library(plumber)
library(here)
```

1.2 Code

1.2.1 Download from Github

Materials for the case study can be cloned by using the following command from the `usethis` package:

Alternatively visit: https://github.com/deanmarchiori/beachwatch

```
usethis::create_from_github("deanmarchiori/beachwatch", fork = FALSE)
```

2 Data Science Frameworks

There are many approaches to tackling a data science project. Following a framework is important to ensuring successful delivery and management of a project. The key benefits of using a dedicated data science workflow is not only ensuring the work gets done right, but it can also provide a useful means for engaging stakeholders, providing project management updates and ensuring everyone is focused on the right areas.

2.1 Popular Frameworks

2.1.1 CRISP-DM

CRISP-DM is a traditional framework for approaching data mining/data science/analytics projects. Developed in the 1990's it has stood the test of time and remains to popular choice today for several reasons.

Firstly, it has a strong focus on understanding the business problem and guiding the exploration of the data with subject matter experts.

Secondly, the framework provides for strict evaluation of solutions with business experts before deployment effort.

Finally, the ethos of iterative, continual improvement is built it, which aligns well with modern agile philiophies.

The key components of CRISP-DM are:

- Business Understanding
- Data Understanding
- Data Preparation
- Modelling
- Evaluation
- Deployment

A drawback of this framework is the way in which deployment is handled in modern use-cases. The need to manage the provide, hosting and monitoring of cloud or server resources has resulted in extensions to CRISP-DM.

Figure 2.1: Kenneth Jensen, CC BY-SA 3.0, via Wikimedia Commons

2.1.2 Inner Loop vs Outer Loop

If we ignore the Deployment step in the CRISP-DM framework we have a nice workflow for completing 'experimental' development and modelling work.

At some point the analyst will want to deploy their work. A useful abstraction that separates the analytical work and the engineering tasks associated with deployment is through the inner loop vs outer loop concept.

The inner loop is the above mentioned CRISP-DM framework right up until deployment.

The outer loop involves the provision of computing infrastructure for model inference, model registration and versioning, deployment and endpoint provisioning, and finally monitoring and evaluation of the deployed model.

While this framework is commonly applied to deploying predictice models, adaptations can be made in the case of a dashboard, web-app or dynamic report output.

- **Outer Loop**
- Infrastructure Deployment
  - **Inner Loop**
    - Business Understanding

    - Data Understanding
    - Data Preparation
    - Modelling
    - Evaluation
- Model Registration and Deployment
- Monitoring

2.2 Development vs Production

A key distinction introduced above is the separation of *development* practices from *production* practices. The typical project will primarily involve development practices outlined in the 'Inner Loop'. When we refer to production work, it typically refers to work to support the deployment of development work in a context where it used to facilitate decision making. We will explore this concept further and analyse the key elements of 'production' code.

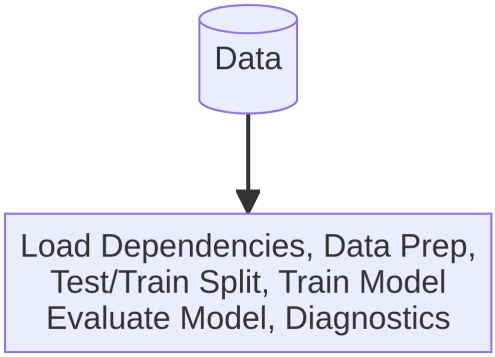
> **Note**
>
> The use of the terms *development* and *production* here represent the intent of the workflow and not specifically a computing environment. In many organisations there are dedicated computing and infrastructure environemnts (often called 'development', 'dev', 'test', 'sandbox', 'prod', uat' etc.) to support the physical separation of these workflow paradigms. This will be explored later.

3 R Code Workflows

In the previous chapter we introduced two frameworks to think about the steps required for a successful data science project. Next we will explore commonly used R code workflows that are used to orchestrate the end-to-end running of analysis or modelling projects.

3.0.1 R Scripts

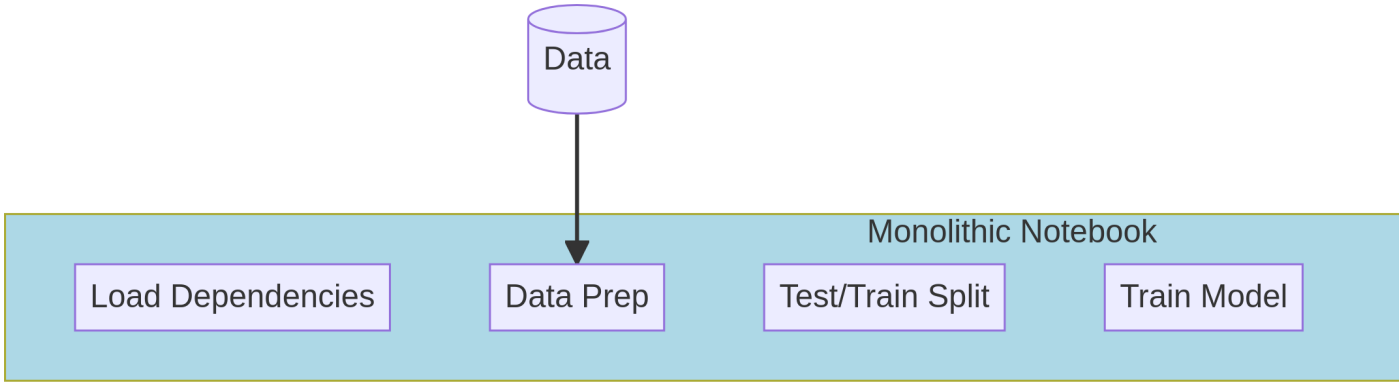
The most basic workflow is to colocate all code into a single R script. This is a common starting place for beginners or when completing small, basic tasks in R. An obvious limitation is the inability to separate out logical components for readability, testing, debugging and control flow.



3.0.2 Monolithic Markdown

Commonly adopted tools to promote more literate programming are notebooks such as RMarkdown or more recently quarto. These tools allow users to write plain english commentary in markdown or a visual editor and splice in 'code-chunks'. Typically this notebook style document is then sequentially rendered in order and knitted into some for of output like HTML, PDF or Word etc.

This is a great way to make code more readble and self-contained while managing complexity. Obvious drawbacks exist around the inflexible execution order, control flow and caching. These can also get very long!



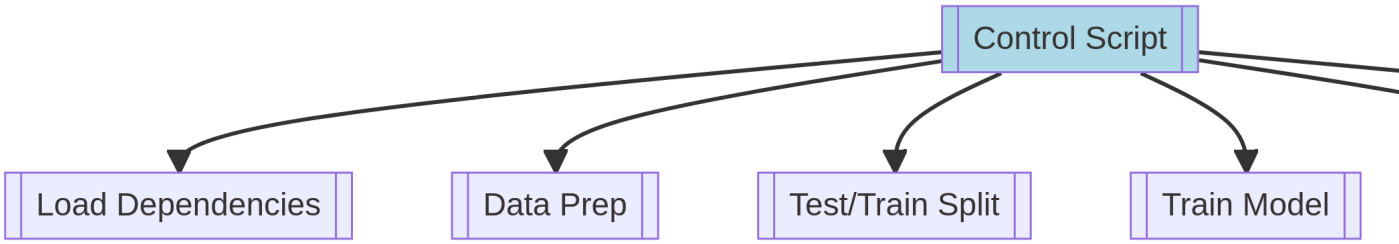
3.0.3 Control Scripts

Analysts who prefer a more scripted workflow will often attempt to break down the complexity of their project into smaller chunks, often placing parts of the analysis into their own R script.

The next question is, how do we orchestrate the running of all these R scripts? This is usually solved with a 'control' or 'run' script, which `source()`'s the relevant scripts in the right order.

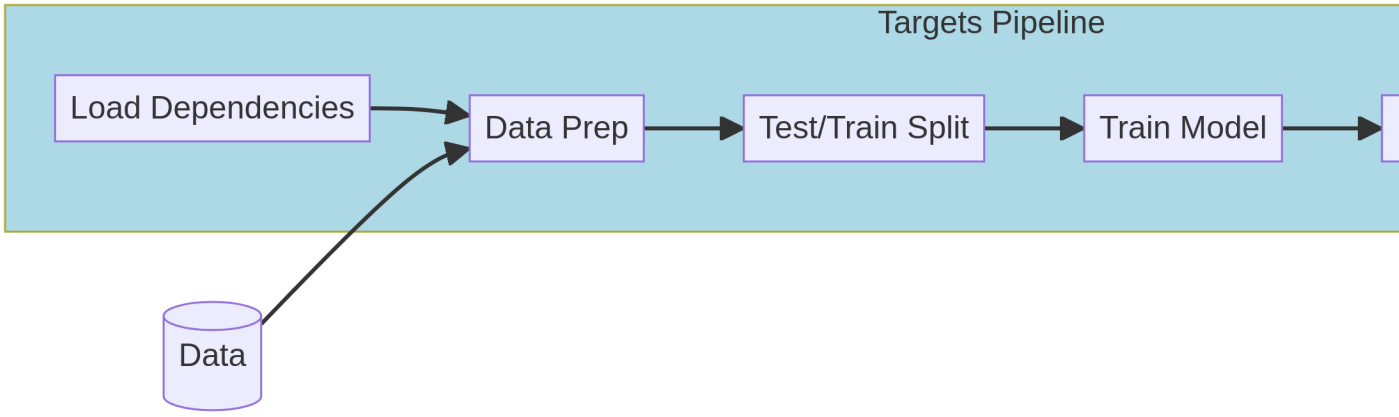
This is a step in the right direction, but requires lots of overhead in managing state and data flows between scripts, often by manually 'caching' results. The scripts are often not self-contained and this can quickly be a recipe for disaster for more complex projects.

[projecttemplate]



3.0.4 {targets}

`{targets}` is an R package that allows users to adopt a make-like pipeline philosophy for their R code. This has the advantage of more sophisticated handling of computationally-intensive workflows and provides a more opinionated structure to follow. With this are the drawbacks or forcing your collaborators to adopt the same framework and dealing with the initial learning curve.

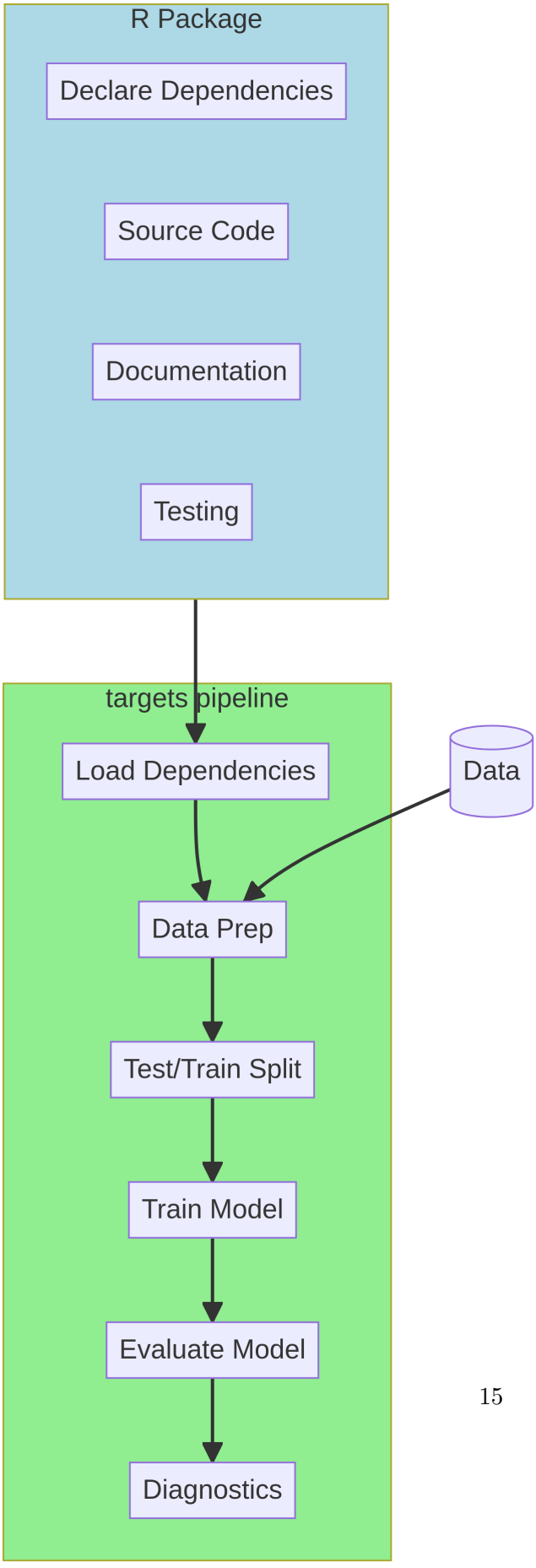


3.0.5 R Package

An R Package is the canonical way to organise and 'package' R code for use and sharing. It provides easy means to share, install, document, test and run code.

Given it is *the* adopted standard for packaging R code, many users have adapted the structure to run analysis projects and research compendia [MARWICK]. This is possible, however the structure is most commonly applied when building tools or algorithms rather than analysis workflows.

[MARWICK]


R Package
Declare Dependencies
Source Code
Documentation
Testing
targets pipeline
Load Dependencies
Data
Data Prep
Test/Train Split
Train Model
Evaluate Model
Diagnostics

3.1 Choosing the right workflow

So which workflow should you use?

Unfortunately this is not a straightforward decision. For quick experimental code you are unlikely to create a new R package. For a complex production deployed model, you really dont want all your code in one giant R script.

Picking the correct workflow needs to align the project goals and scope. Often this choice can evolve throughout the project.

3.1.1 An evolution

A concept or idea might be tested in a single R script, like how you would use the back of a napkin for an idea. Next you might break this down into chunks and add some prose, heading and plots so you can share and have other understand it. Next you might refactor the messy code into functions to better control the flow and the improve development practices. These functions can be documented and unit tested once you know you want to rely on them. To orchestrate the running and dependency structure to avoid re-running slow and complex code you may use the `{targets}` package. Finally to re-use, share and improve on the functions you might spin these out into their own R package!

3.1.2 Repro-retro

I talked a little about how you might want to weight and prioritise the elements of reproducibility in an rtudio::global talk in 2019. Feel free to conduct your own reproducibility-retrospective (repro-retro).

4 Development

When data science projects start, often the outcome is unknown. The work is inherently exploratory and experimental. In many cases this results in a more informal way of working. Frameworks and workflow choices to support this work also need to be designed differently.

4.1 Definition

development is a way of working, but it is also a technical term for a hardware and software environment. In many data science organisations, development is usually a logical (and physical) separation of tools, hardware and software that allow analysts to perform work that is not to be directly relied upon for real-world decision making. This type of environment is prone to running experimental workloads, tests and day-to-day development of projects and ideas by a data science team.

A key defining feature of development code is the colocation of 'what' the code does and 'how' the code is run. That is, the functional elements of the code and the orchestration of those elements are not separated.

4.2 Roles

We can assess the various roles that take place in the development workflow cross sectionally, but looking at the full life cycle from raw data through to a production deployed system.

In most organisations, not all roles will be present. In some organisations, there may even be more specialized roles. For example an ML Ops engineer may handle the deployment of machine learning workloads, while a data engineer May focus entirely on data operations. In addition a data scientist may or may not be present entirely.

4.2.1 Customer

A customer (end-user) is a key role. Customers should be involved throughout the entire process but in particular are active in projects at the raw data and production stage to contextualize, understand and provide understanding of the raw data in addition to being the end user of production systems.

4.2.2 Data Engineer

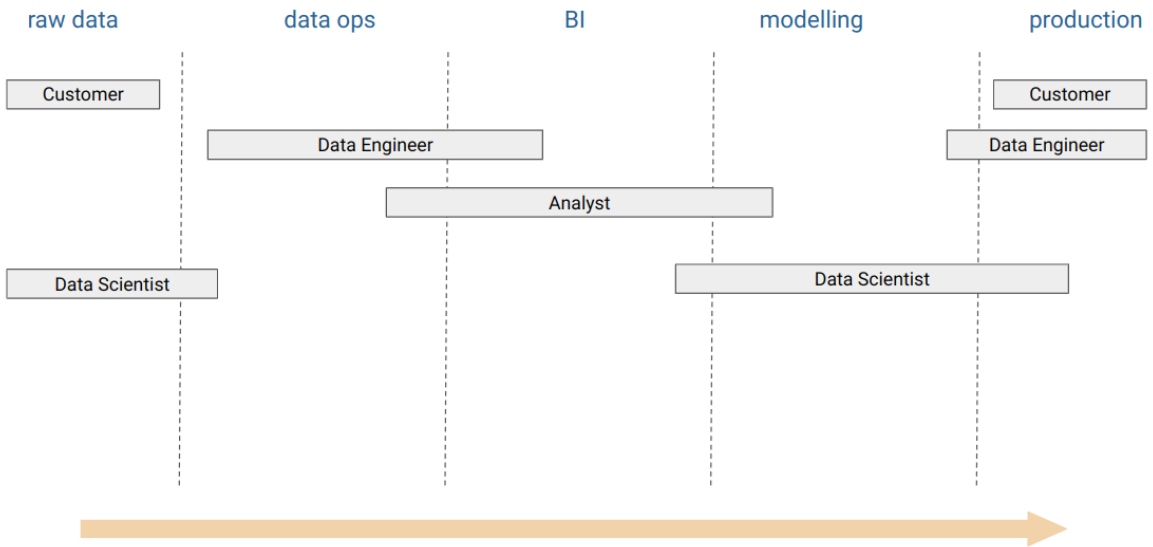
In many organisations an enrichment process from raw data into some kind of data warehouse environment is conducted by data engineers. A data engineer may also be responsible for provisioning infrastructure and assisting with production deployments.

4.2.3 Data Analyst

Once data is landed in a place where it is outside of front-end systems, analysts can typically use this data to perform analysis, Business intelligence (BI), report generation and dashboard building.

4.2.4 Data Scientist

The role of the data scientist can take many shapes. Typically, it is viewed as being of most value in the stage after basic data analysis when more advanced modelling statistical analysis and AI/ML applications are required. In reality, it is advisable for data scientists to be involved in the raw data stage with the customer. This aligns with earlier data analysis frameworks mentioned such as CRISM-DM where an iterative and collaborative approach is required from across all stages of the process.



4.3 Tools

Technical tools to support development workloads usually exist on the analysts local workstation. Most users will have an Integrated Development Environment (IDE) such as RStudio, VSCode or Positron. Some analysts prefer to have a Graphical User Interface (GUI) tool for analysis however this is not recommended when the intention is for the work to be relied upon for reasons we will explore later.

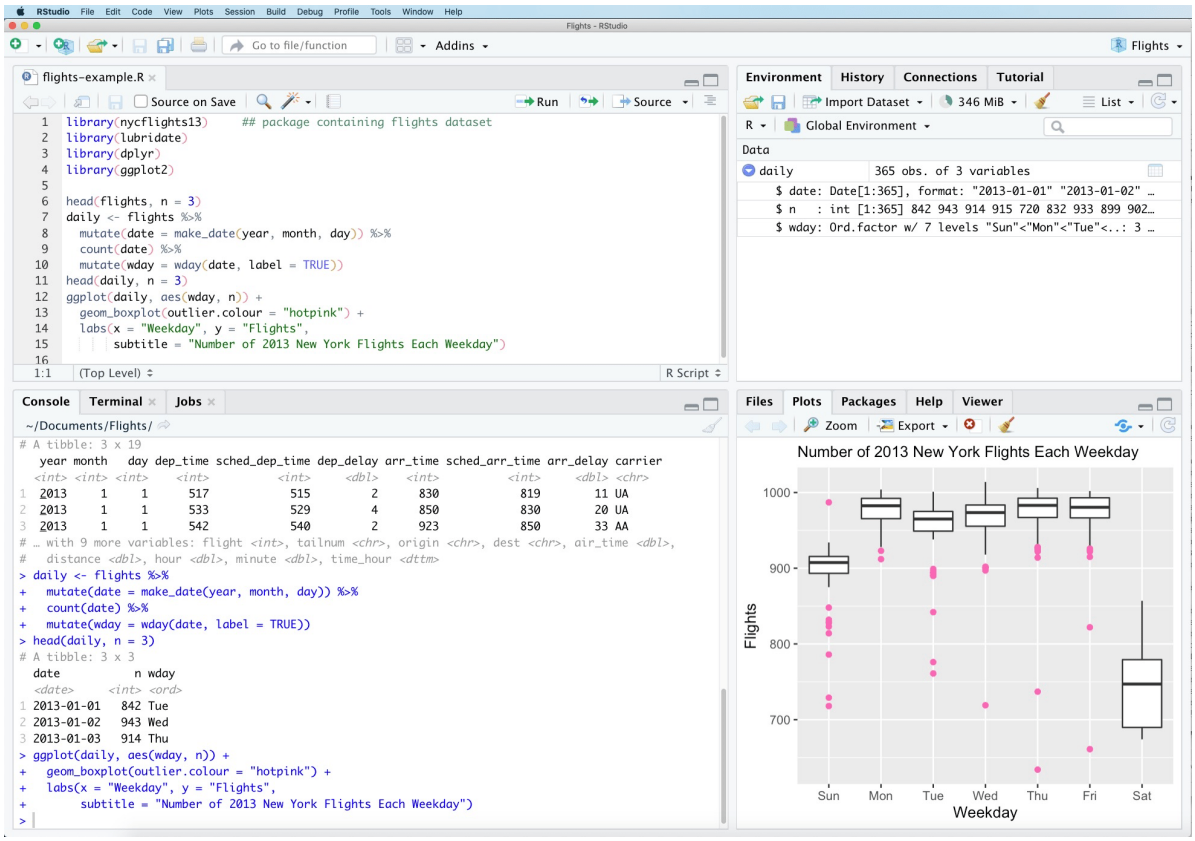


Figure 4.1: By cdhowe - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=101293607

5 Production

5.1 Definition

Production is a conceptual place where your work is being used by the intended users to make real-world decisions.

- Predictive Model integrating with front line business system via API
- Dashboard available to support user's making decisions
- A monthly report that informs management meetings
- An insight from a statistical model that has changed policy

Note how not all of the above involve a technology implementation like an API.

> Example
>
> Jenny trained a statistical model to help predict safety related incidents at worksites for her employer. One of the key factors that influenced and increase in safety incidents was significant rainfall in the previous 24 hours. The safety team has now adjusted their Safe Work checklists to include a check item for the amount of recent rainfall at that site. If its >50mm then a mandatory inspection is performed.
> This change is hardcoded in a paper form, but is still a data-science driven model in production.

5.2 Principles

For a data science project to be successully relied upon in production it should follow some key characteristics.

- Available to end users directly

- Running on stable infrastructure
- Used to make real-world decisions
- Can be replicated
- Can be reproduced safely
- Is documented sufficiently
- Can be maintained by others

In the NYR10 Conference Hadley Wickham summarises his views as:

- Not just once
- Not just my computer
- Not just me

5.3 Patterns for R Code

In terms of R specific outputs, what does a data science output look like from an R perspective.

- An R script that is run on a server on a schedule
- A `{shiny}` app hosted using on a server
- A `{plumber}` API service serving model predictions
- A hosted `{quarto}` dashboard
- An `{rmarkdown}` report
- An R package on CRAN or Github

Regardless of the solution, the fundamental concept is to get it off the developers laptop and have it be reliably available to an end user.

We will build this image up further in the next chapter when we examine more closely the elements of production deployed R code.

6 Elements of Production Deployed R Code

R has a tricky historical reputation as a programming language. With its origins in academia it is often argued that it is not fit for production deployment.
The recent development of tools to support the deployment of R workloads are changing this. Below we will explore some key technical elements that should be included into any stable deployment of a data science worklow adapted from Kreuzberger, Kühl, and Hirschl (2023).

- Orchestration
- Automation
- Reproducibility
- Version Control
- Metadata and Documentation
- Testing & Monitoring

6.1 Orchestration

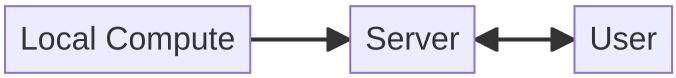
Orchestration refers to how your code is structured and run. We explored various ways of structuring our code for data science workflows in chapter two. There are two key facets to code orchestration. The first is separating the functional components of your project, and the orchestration of those functional components to do some task.

As R is a functional programming language the canonical way to structure our code is to use functions. In a practical sense, functions are a convenient way to package, document, test and control the execution of code in a project.

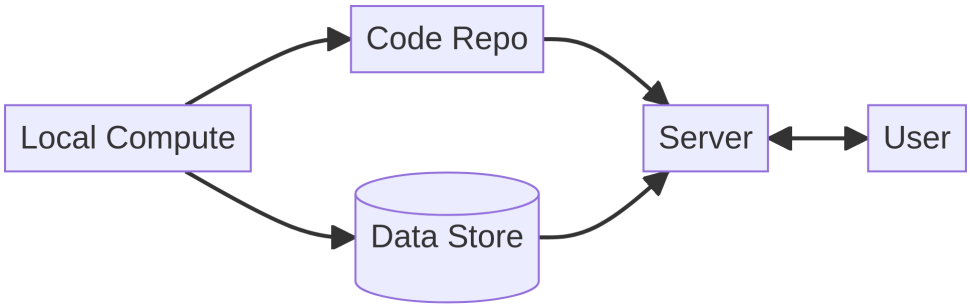
Once the functional elements of your project are stored, documented and tested appropriately they need to be run in the correct order. This can be achieved using a number of frameworks (see X) such as using notebooks, an R Package, targets and more.

6.2 Automation

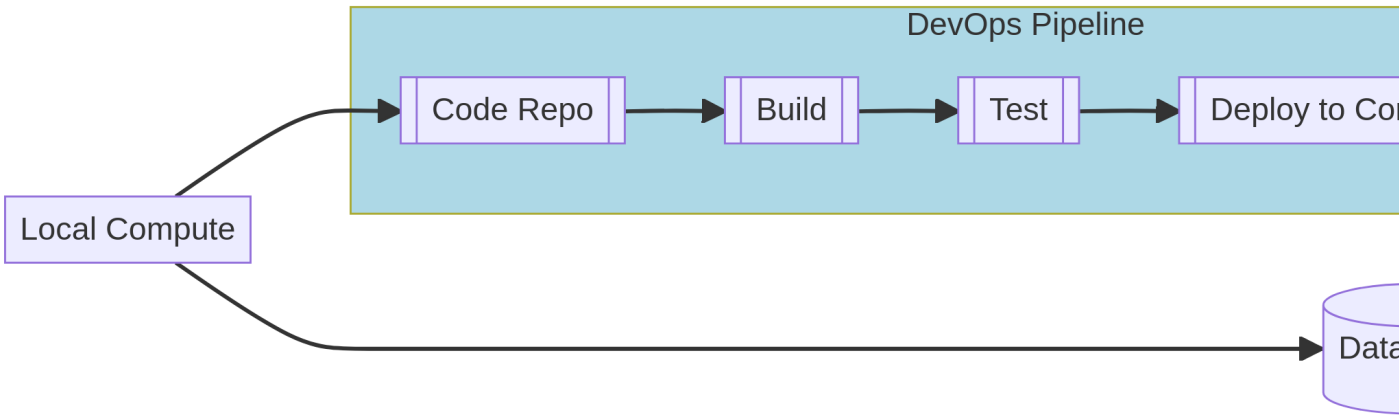
Automation refers to how your project is 'built'. In other words how it is pushed and pulled from your local development environment into a remote environment where other users can access the results. Again, many options exist and there are varying levels of automation. Manual approaches such as click button deployment from your IDE or manually copying files across to a remote server are one option.



A better approach would be to stage your code using a remote version control repository. This will act as a store or source code which can be more conveniently 'pulled' or synchronised with the server where your analysis is hosted for users. Often, accompanying model artefacts or data will also to stored somewhere remotely for the server to access.



More contemporary approaches involve Continuous Integration and Continuous Deployment (CI/CD) practices. This is a DevOps style workflow that will automatically build, test and deploy code that has been pushed to a remote version control repository. A thorough exploration of CI/CD Solutions is beyond the scope of this book and is now a clearly defined sub specialty known as MLOps.



6.3 Reproducibility

6.3.1 Code dependencies

We talk about reproducibility, there are many elements of a data science workflow that need to be considered Code dependencies. The first step to a reproducible pipeline is ensuring that all users have. The same code that is being used to run the project. The recommended approach here is to ensure that all code is checked in to a remote version control repository, using a version control system such as git.

Is that why other collaborators can clone the code base from the remote git repository? And, Branch. Or Fork from the code repository in order to make their own changes, it can then be integrated back. Using proper Version Control principles. Two users running the same code may not have the same R packages installed on this system.

6.3.2 Packages dependencies

A key element of reproducibility is ensuring that all users. Can resolve. Our package dependencies this includes having the correct packages. Installing those packages from the correct locations. And ensuring the version of those packages is equivalent. A convenient solution for these problems is the RN package. The RN package does this?

6.3.3 System dependencies

System dependencies. System dependencies describe software, that is installed on. The computational environment that the project is being run on. These may include. External libraries. Such as Example. Tools like deposit, public package manager can provide some analysis of

the system dependencies. That are required to support our packages on various operating systems.

Another solution explored in the next section. Is using Technologies, such as Docker. Operating system dependencies. Users, even when running the same code with the same, Our packages may find differences in how The code performs based on the operating system they're using, for example, Windows versus Linux versus Mac OS.

6.3.4 OS dependencies

It is common in a production setting to deploy code. To an external server using. A Linux operating system. A way to control operating system and system dependencies. Is. Use Technologies such as Docker, Docker does this. A basic Docker file is shown below. Finally. The dependencies.

6.3.5 Hardware dependencies

Hardware dependencies are a little trickier. These are physical Hardware infrastructure constraints on how a project is run. Regardless of the operating system and software that is running on it. For example, this is commonly seen with the use of CPU versus GPU Technologies and different types of processor chips. A thorough exploration of Hardware dependencies is outside. The scope of this guide.

6.4 Version Control

Version Control. Version Control is a critical aspect. To ensure. Reproducibility instability in production deployed data, science Solutions. Version control includes. Not only Version Control for code, but also Version Control for data and models. Version control for code is commonly achieved using the get Tool.

Git is a version control system that allows users to Do this. If the data science project results in a Statistical and machine learning model being fit. It is the model itself that will be the deployed artefact in order for inference to be performed. Therefore it is critical that this model B also versioned. Again, there are many solutions for this. However, a recent development in, the r ecosystem is the vetivert package. A bit of a package, does this? The exploration of Version Control with data is beyond the scope of this book. However, It is recommended. That data is also versioned and stored appropriately in a secured data store, such as a data Lake or data warehouse.

6.5 Metadata and Documentation

Metadata and documentation. It is important. When deploying artefacts into a production setting, that there is appropriate metadata. Metadata refers to. Tags versions dates and other relevant information to describe what work is being deployed. We'll explore this further. In a modelling context, using the bit of a package. In terms of metadata and documentation for the functional aspects of the code, We can rely on the Internal documentation used in our packages.

Based on the oxygen package. This provides a useful template for documenting, our functions using oxygen tags. That are automatically generated into help documents. Another form of documentation in an R package is a readme. The package readme is an important artefact to tell users, what? Software does how it is to be run in any other important information such as the licence or prerequisites or dependencies.

Also how to contribute and get help? Low form documentation.

6.6 Testing

Testing. Finally. Test unit testing is important. Element of software engineering. In terms of our code. Testing, can be performed using the test that package. The test that package does this. And here is an example of it. Conveniently the test. That package is integrated into the build process for our packages.

Using the L Studio IDE. Testing can also be performed on a directory. Using this function.

7 Practical Considerations

7.1 Working with IT teams

- Install
- Admin
- Maintainence
- Debugging

7.2 Open Source vs Commerical Software

7.3 Network Security

7.4 Authentication

7.5 Changes

- Model drift
- New features

8 Case Study

8.1 About

We would like to build a predictive model to explain and predict water temperature at Sydney beaches. The codebase will have two branches.

1. `main`: A demonstration of a typical monolithic notebook style analysis

2. `production-ready`: A demonstration of a refactoring of the above to include elements of production-ready code.

To clone the codebase from Github:

```
usethis::create_from_github("deanmarchiori/beachwatch", fork = FALSE)
```

8.2 Development Code

Ensure you are on the `main` branch.

Let's take a look at the structure of the repository.

We have out raw data in the `data` directory and a Quarto notebook called `model_notebook.qmd` which contains the end-to-end workflow. This notebook also exports our final fitted model as an `.rds` object in the `deploy` directory.

```
.
  beachwatch.Rproj
  data
      Water quality-1727670437021.csv
  deploy
      beachwatch_model.rds
  model_notebook.qmd
```

Interactive Session

Explore the contents of `model_notebook.qmd` locally or at https://github.com/deanmarchiori/beachwatch/blob/main/model_notebook.qmd

8.3 Production Code

Next we will make a number of changes to the above structure to implement the elements of production quality R code.

Switch to the `production-ready` branch.

Let's take a look at the structure of the repository. We will go through this step by step.

```
.
  beachwatch.Rproj
  data
      sydney_water_temp.rda
  data-raw
      sydney_water_temp_raw.R
  DESCRIPTION
  Dockerfile
  inst
      analysis
          model_notebook.qmd
      deploy
          sydney-beach-gam
              20241007T015537Z-bb3f3
                  data.txt
                  sydney-beach-gam.rds
              20241007T021501Z-bb3f3
                  data.txt
                  sydney-beach-gam.rds
      extdata
          Waterquality1727670437021.csv
  LICENSE
  LICENSE.md
  man
      fit_water_temp_model.Rd
      sydney_water_temp.Rd
  NAMESPACE
```

```
plumber.R
R
    data.R
    fit_water_temp_model.R
tests
    testthat
        test-gam-function.R
    testthat.R
vetiver_renv.lock
```

8.3.1 R Package

The most radical change was the adoption of an R Package framework. This is not strictly necessary, however it will help us adopt common features around how we document and test our code.

To create an R package you can use the RStudio IDE (File > New Project) or use the `{usethis}` command below:

```
usethis::create_package(path = here::here())
```

```
> usethis::create_package(path = here::here())
  Setting active project to "/home/deanmarchiori/workspace/beachwatch".
  Creating R/.
  Writing DESCRIPTION.
Package: beachwatch
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R (parsed):
    * First Last <first.last@example.com> [aut, cre] (YOUR-ORCID-ID)
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to
    pick a license
Encoding: UTF-8
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.0.0
  Writing NAMESPACE.
  Writing beachwatch.Rproj.
  Adding "^beachwatch\\.Rproj$" to .Rbuildignore.
  Adding "^\\.Rproj\\.user$" to .Rbuildignore.
  Opening /home/deanmarchiori/workspace/beachwatch/ in new RStudio session.
  Setting active project to "<no active project>".
```

We can see this sets up the package skeleton for us.

We can install the R package once we are in the Project by running

```
devtools::install(pkg = ".")
```

8.3.2 Metadata

Next we should update and review the various metadata that are created.

- The DESCRIPTION file is a useful way of including details on what your does and who the maintainers and developers are.
- LICENCE provides a way to convey the licence you are releasing the software under
- NAMESPACE is a special file that is automatically created for us.

8.3.3 Data

In the development code we had an arbitrary directory with the data stashed in there as a csv file. We make the handling of data more formal here.

1. The raw data is placed in `inst/extdata`. The `inst` directory is a place for arbitrary files in an R package. We will use this for our raw data and our analysis and deployment artifacts.
2. A script to read in and clean the raw data is available in `data-raw`.
3. The above script ends with the code `usethis::use_data` which formally exports the clean data for use by your R package.

The below helper functions can assist here:

```
usethis::use_data_raw("sydney_water_temp_raw")
```

```
> usethis::use_data_raw("sydney_water_temp_raw")
  Setting active project to "/home/deanmarchiori/workspace/beachwatch".
  Creating data-raw/.
  Adding "^data-raw$" to .Rbuildignore.
  Writing data-raw/sydney_water_temp_raw.R.
  Modify data-raw/sydney_water_temp_raw.R.
  Finish writing the data preparation script in data-raw/sydney_water_temp_raw.R.
  Use `usethis::use_data()` to add prepared data to package.
```

```
usethis::use_data(sydney_water_temp, overwrite = TRUE)
```

```
> usethis::use_data(sydney_water_temp, overwrite = TRUE)
  Setting active project to "/home/deanmarchiori/workspace/beachwatch".
  Adding R to Depends field in DESCRIPTION.
  Setting LazyData to "true" in DESCRIPTION.
  Saving "sydney_water_temp" to "data/sydney_water_temp.rda".
  Document your data (see <https://r-pkgs.org/data.html>).
```

Next we need to document our data set using the template seen in `R/data.R`.

```
#' NSW Beachwatch Water Quality Data
#'
#' Water temperature and quality data collected under the NSW Government Beachwatch program.
#'
#' @format
#' A data frame with 8947 rows and 7 columns:
#' \describe{
#'   \item{temp}{Water Temperature in C}
#'   \item{beach}{Water measurement site}
#'   \item{date}{Measurement date}
#'   \item{time}{Measurement time}
#'   \item{month}{Numeric label for month of measurement}
#'   \item{hour}{Numeric label for hour of measurement}
#'   \item{month_lab}{Factor label for month of measurement}
#'   ...
#' }
#' @source <https://beachwatch.nsw.gov.au/waterMonitoring/waterQualityData>
#' @source licensed under the Creative Commons Attribution 4.0 International (CC BY 4.0)
"sydney_water_temp"
```

Once we build and install our package we can run the below code to load the clean, model ready data into our R session.

```
data(sydney_water_temp)
```

```
Warning in data(sydney_water_temp): data set 'sydney_water_temp' not found
```

If we want to understand more about our data, we can read the documentation by running `??sydney_water_temp`.

8.3.4 R Functions

We can now refactor any R code in out notebook to be formal R Functions.

Each function can occupy its own R Script in the `R/` directory of the project. Properly documented with Roxygen tags, these will automatically generate help pages when the package is documented and built.

In any function, we should explicitly namespace the functions within using the `pkg::foo()` standard. To capture these dependencies we can run:

```
usethis::use_package(package = "pkg")
```

8.3.5 Tests

Unit tests can be written using the `{testthat}` package.

```
usethis::use_testthat()
```

```
> usethis::use_testthat()
  Setting active project to "/home/deanmarchiori/workspace/beachwatch".
  Adding testthat to Suggests field in DESCRIPTION.
  Adding "3" to Config/testthat/edition.
  Creating tests/testthat/.
  Writing tests/testthat.R.
  Call usethis::use_test() to initialize a basic test file and open it for editing.
```

8.3.6 Analysis

> Interactive Session
>
> Explore the contents of `model_notebook.qmd` locally or at https://github.com/deanmarchiori/beachwatch/blob/production-ready/inst/analysis/model_notebook.qmd

We can see the Quarto notebook containing our workflow still exists. We have moved it to `inst/analysis` for convenience. This is still the primary document that analysts will use to develop and iterate on their models. However, unlike before when all the functional and orchestration code was co-mingled, we now have the case where the key elements of our workflow have been decomposed into discrete functions.

> **Note**
>
> It's important here to note that this isn't the only workflow choice possible. Users may prefer to implement a {targets} workflow for instance rather than persist with a notebook. This is fine, and in fact modularising the code as we have done is a common first step in this type of refactoring.

8.3.7 Deployment artefacts

At the end of the analysis we need some way to get our model off our laptop. The framework we are using in this example are the tools from the `{vetiver}` package.

The first step is the save our fitted model somewhere. Rather than just save a serialised object in a folder, it would be ideal to capture some other details:

- Model Name
- Model Type
- Metadata such as
  - Model Version
  - Metric (e.g. AIC)
- Persisted versions

The use of the `{pins}` and the `{vetiver}` package can help us.

First we configure a 'board' to 'pin' our model to. In this case we will just register a local directory as a board (called `deploy_board`), but you can configure this to save to multiple locations such as Posit Connect, Azure Storage, AWS S3, Dropbox, Github etc.

```
deploy_board <- pins::board_folder(path = here("inst/deploy"), versioned = TRUE)
```

Next we save our model (called `mod_gam`) as a 'vetiver model' object.

We give it a nice name, some metadata and we can also include a sample of new data for testing purposes, which will be handy later.

```
v <- vetiver_model(
  model = mod_gam,
  model_name = "sydney-beach-gam",
  metadata = list(aic = mod_gam$aic),
  save_prototype = data.frame(
    month = 12,
```

```
    hour = 6,
    beach = factor("Bondi Beach", levels = levels(sydney_water_temp$beach))
  )
)
```

This can be written to our 'pins' board

```
vetiver_pin_write(board = deploy_board, vetiver_model = v)
```

We can see this is saved a versioned object in `inst/deploy`

```
inst
    analysis
        model_notebook.qmd
    deploy
        sydney-beach-gam
            20241007T015537Z-bb3f3
                data.txt
                sydney-beach-gam.rds
            20241007T021501Z-bb3f3
                data.txt
                sydney-beach-gam.rds
```

```
file: sydney-beach-gam.rds
file_size: 416173
pin_hash: bb3f3af062cceea5
type: rds
title: 'sydney-beach-gam: a pinned list'
description: A generalized additive model (gaussian family, identity link)
tags: ~
urls: ~
created: 20241007T015537Z
api_version: 1
user:
  aic: 29114.71767
  required_pkgs: mgcv
  renv_lock: ~
```

We can also use various functions to read our `pins` boards

```
pins::pin_list(board = deploy_board)
vetiver::vetiver_pin_read(board = deploy_board, name = "sydney-beach-gam")
```

8.3.8 Deploy

Now we have saved and versioned our model appropriately, we need a way to deploy it as an API endpoint that users can supply new data to and get a model prediction.

While there are other pathways (particularly for users of Posit Connect) we will focus on using Docker.

`veriver` also has helper functions to create the deployment artifacts we need.

Firstly, to test our deployment locally we can use

```
pr() %>%
  vetiver_api(v) |>
  pr_run(port = 8080)
```

This takes our 'vetiver model' object and uses the `plumber` package to rig up an API endpoint and documentation. It also helpfully creates other endpoints for health-checks and metadata.

To deploy as a Docker container we need to define a docker file and identify the system and package dependencies we need. You can do this manually, or we can rely on the helful function:

```
vetiver_prepare_docker(board = deploy_board,
                       name = "sydney-beach-gam",
                       docker_args = list(port = 8080),
                       path = here::here())
```

This will create three files:

1. Dockerfile
2. renv lockfile

3. plumber.R

> **Tip with Title**
>
> If you have saved your model to a local folder, instead of a remote pin you will need to edit the Dockerfile to copy across this folder. It is generally recommended to remotely save your model object. Use e.g. `COPY inst/deploy /opt/ml/inst/deploy`

To build you docker image:

```
docker build -t beach .
```

and to run it:

```
docker run -p 8080:8080 beach
```

The endpoint should now be live at `http://127.0.0.1:8080`

9 Resources

Link to github analysis flow

10 Acknowledgements

References

Kreuzberger, Dominik, Niklas Kühl, and Sebastian Hirschl. 2023. "Machine Learning Operations (MLOps): Overview, Definition, and Architecture." *IEEE Access* 11: 31866–79. https://doi.org/10.1109/ACCESS.2023.3262138.