DEUTSCHE
BUNDESBANK
EUROSYSTEM

Data Orchestration Blueprint Based on YAML {dobby}

Research data pipelines in R

Technical Report 2021-03, Deutsche Bundesbank, Research Data and Service Centre

Matthias Gomolka
Jannick Blaschke
Constantin Brîncoveanu
Christian Hirsch
Ece Yalcin

Research Data and
Service Centre

Abstract

Research data centres manage access to confidential micro data for research purposes. One essential task is enhancing and documenting data retrieved from Bundesbank data producers. We introduce the R package {dobby} aimed at standardising related workflows including the cleaning, quality testing, and anonymization of micro datasets. A prerequisite for using this package is a YAML configuration file describing all dataset structure and variable properties. This technical report gives a high-level description of the conceptual and technical set-up of {dobby} and highlights the RDSC's main lessons-learned.[1)]

Keywords: dobby, YAML, data pipeline, drake, validate, Research Data Centre, RDC, Data Production

Citation: Gomolka, M. et al (2021). Data Orchestration Blueprint Based on YAML {dobby}: Research data pipelines in R, Technical Report 2021-03. Deutsche Bundesbank, Research Data and Service Centre.

1 The authors gratefully acknowledge the contributions from our colleagues at the RDSC. {dobby} would not have been possible without their preliminary work in the last years. We thank Jens Orben for helpful comments.

Contents

1 Introduction

As one of the largest producers of statistical data in Germany, the Deutsche Bundesbank collects high-quality monetary, financial and other statistical micro data. The Bundesbank's Research Data and Service Centre (RDSC) manages the access to such confidential micro datasets for analysts and researchers[2]. RDSC staff members advise researchers on data selection, data access, data content and appropriate analytical approaches for their respective research projects[3].

Each RDSC staff member is specialised on a number of datasets from the RDSC's portfolio and responsible for answering content-related questions and updating their datasets on a regular basis. Usually, such a production process starts with compiling all exports that the data producing business unit within the Bundesbank sends to the RDSC. The data producer has already quality-checked these data and aligned with the RDSC on a basic data structure, such as e.g. the included variables.

The RDSC then applies additional modifications to the data in order to further improve their usability for research purposes and to ensure consistency with other datasets in the RDSC's portfolio. After that, the RDSC runs further data quality tests, this time to check whether the modifications went as planned and to ensure the data's consistency over time. Finally, the RDSC saves the data in different formats and updates the corresponding data report, a documentation for each dataset that is made available to all analysts and researchers.

As each dataset has individual demands, e.g., when it comes to size, data structure, or selecting appropriate data quality checks, RDSC staff members usually write a customised production code for each dataset. As an attempt to unify these production codes while maintaining the flexibility to account for dataset-specific properties, the RDSC developed the R package {dobby}. {dobby} automatically provides the users with a data production process template, which they need to customise, for example, using own metadata on their dataset. Within the {dobby} framework, there is an opportunity to harmonise both the provision of data and the structure of the datasets, and ultimately to simplify research with these datasets.

This technical report is structured as follows. Section 2 provides a high-level description of the conceptual and technical set-up of the {dobby} framework. Section 3 presents main lessons that we[4] have learned while implementing the {dobby} framework. Finally, Section 4 concludes.

2 Security aspects are not in the focus of this paper. However, dobby is secured by the IT environment of the Deutsche Bundesbank.
3 For more information on the Bundesbank's RDSC see Schönberg (2019)
4 In this paper, „we" and „RDSC" are used synonymously.

2 Introducing {dobby}, the RDSC's approach for a standardized data production process

One integral part of the RDSC's work is the production of standardised research datasets characterised by refined data quality checks and a comprehensive documentation. For the most part, these standardised datasets fall in the category of micro data, i.e. very granular data on the level of the individual entities such as banks, financial companies or households. The micro data in the RDSC today is characterised by their increasing volume and collection frequency as well as their heterogeneity and complexity (Witt & Blaschke, 2019). Judging from recent developments, both is likely to increase in the future by tapping into, for example, unstructured data. It is important to keep these features in mind when designing a standardised data production process and assessing appropriate technical solutions.

In this technical report, we present the R package {dobby}, which is the RDSC's approach for a performant and standardised data production pipeline. {dobby}, which stands for „data orchestration blueprint based on YAML", has been developed especially for the RDSC and is adjusted to its needs. It builds on top of the two well-tested and production-ready R packages {drake} (Landau, 2018)[5)] and {validate} (van der Loo & de Jonge, 2021).

As the datasets, in general have some common issues such as noise, inconsistency, and redundancies, they are not directly applicable for any analysis or project (Garcıa, Ramırez-Gallego, Luengo, Benıtez, & Herrera, 2016). In order to provide high quality of data sets and help to increase the success rate of the future projects, the data processing and preparation are crucial (Dharmarajan & Vijayasanthi, 2015). Unfortunately, the data processing and preparation have been known to be very costly in time and computational resources (Zhang, Zhang, & Yang, 2003). With {dobby}, the RDSC aimed at establishing a production process that brings the benefits of standardisation while still being flexible enough to accommodate for dataset-specific requirements, such as different data quality checks. {dobby} follows {drake}, a data pipeline toolkit, which is available in R. In {drake}, the production code is built in a modular, function-oriented way rather than in a single encompassing script. The core of {drake} and {dobby} is the so-called {drake} plan, which serves as an entry point orchestrating all other parts of the program code. The remainder of the chapter describes the key steps of the data production process using {dobby} as depicted in Figure 1.

When a {dobby} user wants to on-board a new dataset to {dobby}, he or she needs to start a new "production project" for this particular dataset. Users can easily achieve this via a single command in R. {dobby} automatically sets up the necessary directories, templates and files. This includes the {drake} plan and templates for the three components that users can customise in order to adjust {dobby} to their dataset's demands:

i. The YAML file (see Step 2 in Figure 1),
ii. the code lists (see Step 3 in Figure 1) and
iii. the data cleaning function (see Step 4 in Figure 1).

In the next three steps, we describe how the user can adapt them according to his or her own requirements.

5 We are well aware of {targets} (Landau, 2021), the successor of {drake}. But when we started working on {dobby}, {targets} was not yet released. It is likely that further updates of {dobby} will build upon {targets} instead of {drake}.

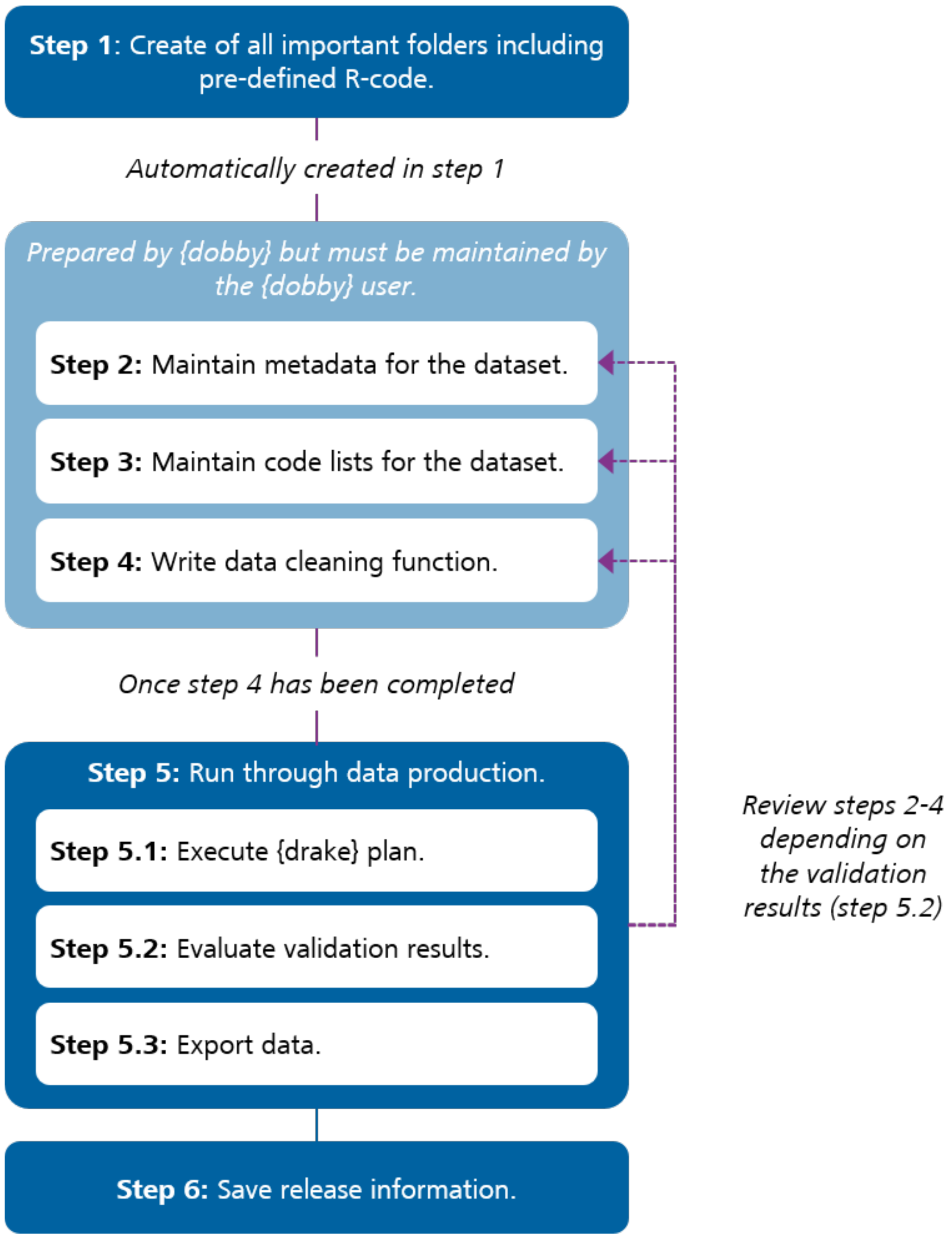
Step 1: Create of all important folders including pre-defined R-code.
Automatically created in step 1
Prepared by {dobby} but must be maintained by the {dobby} user.
Step 2: Maintain metadata for the dataset.
Step 3: Maintain code lists for the dataset.
Step 4: Write data cleaning function.
Once step 4 has been completed
Step 5: Run through data production.
Step 5.1: Execute {drake} plan.
Step 5.2: Evaluate validation results.
Step 5.3: Export data.
Step 6: Save release information.
Review steps 2-4 depending on the validation results (step 5.2)

Figure 1: Simplified flow chart for dobby

The YAML configuration file describes the entire data structure and variable properties and serves as a machine-readable data dictionary. This metadata information is required for both the production and documentation. For example, the YAML includes the location of the data in the folder structure as well as a list and description of all variables in the dataset that can facilitate the generation of the data report. Here, users can already add variable-specific restrictions that need to be satisfied and will be tested during the data production. The code chunk below shows the main structure of a YAML file while the annex contains a full example for a fictional dataset.

```yaml
main:
  short-name: # Short name of dataset
  long-name: # Full name of dataset
  doi: # Digital object identifier (DOI)
  source: # Data provider
  responsible:
  - # Responsible data specialist

orig:
  path: # Path where original data is stored
  regex: # Regular expression identifying files if ‘path’
         # is a directory

columns:
  XXX: # Name of variable
    label: # Variable label
    notes: # Longer description of the variable
    is-key: # ‘yes’, if variable is an ID, otherwise ‘no’
    constraints:
        type: # Type of the variable, such as ‘character’
              # or ‘integer’
        (…) # Other optional constraints per variable that
            # will be tested
```

The RDSC knows that some variables should only contain a pre-defined set of unique values. For example a dummy variable should at most contain values for ‘Yes’, ‘No’ and ‘Unknown’ and also a variable with a sector classification can only have as many unique values as there are sectors. Therefore, {dobby} allows the inclusion of code lists, which contain all possible values of a given variable. During the production process, {dobby} matches these code lists with the actual codes that occur in the data.

The third component that {dobby} users have to add for each of their {dobby} production projects is the cleaning function. As the data cleaning very much depends on the individual dataset, {dobby} only provides a template and a number of utility functions that users may use to customise for the demands of their dataset. It is also possible to integrate existing cleaning code or even scripts in other programming languages such as Stata or Python.

Once these three components have been prepared, the user can run the {drake} plan, which will execute the different parts of the production plan. After the data import and cleaning, the {drake} plan will carry out the data validation using the constraints defined in the YAML file and the code lists. The user will automatically receive an evaluation of the results of the individual checks and thus, directly understand, whether the data cleaning was sufficient. In case of errors or new codes the {dobby} user can use the automatically created reports to, e.g. further discuss those issues with colleagues.

Finally, when the user is satisfied with the result of the previous step, {dobby} exports the cleaned and checked data in different file formats to a pre-defined folder. In addition, the data report, which the RDSC publishes for each new version of a dataset, will automatically be updated. A data report consists of a descriptive part, outlining for example the legal mandate for data collection or available modules, and a part on variables and code lists. As data reports are written in RMarkdown documents, the second part can easily be updated based on the information in the YAML file and additional information derived from the data itself.

Even though this sounds like a linear process, it is usually very iterative. To speed things up, most steps in the {drake} plan run in parallel without any additional configuration from the user's side. On top of that, {drake} always caches the current state of a production project so that it only runs steps that are actually outdated. In order to determine the status of each step, it tracks changes in input files and all parts of the {drake} plan. If {drake} marks all steps as up to date, users can be sure that all steps of the data production concluded as expected.

All code, metadata and code lists are stored under version control. This helps users to maintain a clean codebase and allows tagging all files used to produce a certain release of a dataset. This makes the data production reproducible and more transparent.

3 Lessons learned

In the previous chapter, we described a concrete implementation of a standardised data production pipeline. However, the design of {dobby} is very much tailored to the needs of the RDSC. Nevertheless, we would like to share what the RDSC has learned during the conception and development of {dobby}.

We believe that there are six building blocks, which are the most important success factors for the development of an efficient yet user-friendly data production software in an RDC environment. Readers should view these building blocks as important design decisions they need to take during the development and that may only be changed with large effort once the software is in use.

Lesson 1: Design your software with your largest and most complex data in mind

In our use case of financial data, efficiency gains from producing data increase with size and complexity. For one, because larger and more complex data simply take more computing time to process. Additionally, the task is more challenging for humans. The latter stems from the fact that programs need to be more sophisticated to handle larger and more complex data adequately.

For us this ultimately meant implementing {dobby} using the pipeline tool {drake}. While this design choice offers high utility for large and complex data it may seem a bit over the top for smaller and less complex data in the RDSC. This is true from a pure programming point of view. However, bear in mind that implementing a standardised approach offers additional benefits beyond programming. This brings us to the second lesson that we have learned.

Lesson 2: Enable knowledge sharing in your community and standardise as much as possible

Even though different datasets have diverging demands during the data production process, some common standards can easily be implemented. For example, in {dobby} we have a consistent programming language[6)], a consistent folder structure, and a dedicated step, where the data quality testing takes place. In addition, we encourage a consistent data structure[7)].

Implementing a unified solution for all of your datasets enables sharing of knowledge in two distinct ways. First, it helps sharing of reusable program code and best practice workflows among experienced and less experienced users. This is especially important for on boarding users who are less skilled in writing code.

Second, this is also about how relevant information is available to interested users. Consider for example our choice to store all data quality checks as restrictions in a YAML file. This design choice enables interested users to quickly access restrictions on other data and evaluate whether they want to use these for their own data.

6 {dobby} is written in R (R Core Team, 2020).
7 For the usage of {dobby} we recommend that the data follows the tidy structure (Wickham, 2014).

Lesson 3: You will need to check the quality of your data and talk to someone about it

An important component of the data production workflow of the RDSC is conducting data quality checks confirming that any modifications to improving the usability of data for research purposes did not affect the data content inadvertently. While implementing the checks is important, you will also want to present the outcome of these checks in such a way as to allow for seamless communication with e.g. data producers. For us this means that all outcomes of quality checks are automatically stored in an Excel sheet or CSV files.

Lesson 4: Let the most experienced programmer do the programming part and let the least experienced user document it

The RDSC's staff members come from a variety of different backgrounds and consequently are equipped with different skillsets. During the development phase of {dobby}, we tried to benefit from this circumstance. The development team included technically skilled developers, experienced data specialists as well as new rather inexperienced potential {dobby} users. The latter were mainly responsible for writing the {dobby} user guide so that we could ensure that it is as simple and comprehensible as possible. To further reduce the entry costs especially for less technically experienced users, the RDSC recommends providing examples for YAML files, code lists and data cleaning code from already on-boarded datasets.

Lesson 5: Code-reuse improves code quality and reduces bugs

As many data specialists at the RDSC can easily use {dobby}, bugs are spotted earlier and can be fixed for all users. In addition, when code is written for more than a single occasion, the time invested in writing unit tests pays off quickly.

Lesson 6: Introduce less experienced users to best practices

Since {dobby} takes care of setting up a production project, best practices such as using version control or suitable folder structures can be enforced without overstraining less experienced users.

4 Conclusions

The goal behind the development of {dobby} was to set up a standardised production process for research datasets that builds up on existing knowledge of individual data specialist at the RDSC and combines it to a standardised yet flexible production pipeline. The main goal of the pipeline should be the production of research datasets.

Even though the RDSC typically provides standardised datasets to researchers, they are nevertheless tailored to the typical research use cases that the RDSC has distilled from past research projects. In the context of an increasing volume of provided datasets, it is particularly important to design such a production pipeline in the most time-efficient and performant way possible.

Therefore, the RDSC highly recommends using a programming language, which is well suited for data-oriented tasks but also flexible enough to build a modular data pipeline such as R or Python. Furthermore, we suggests leveraging on already existing data pipeline toolkits such as {drake} or {targets} facilitating parallel workflows and running only outdated steps in the data pipeline.

A comprehensive use of {dobby} leads to major efficiency gains through automation of data production in the RDSC and improved knowledge transfer between RDSC staff. When responsibilities shift, the training period will reduce significantly, as all datasets use the same production pipeline template and even the customised parts, i.e. the YAML, code lists and the quality checks, follow a common structure and taxonomy.

References

Dharmarajan, R., & Vijayasanthi, R. (2015). An overview on data preprocessing methods in data mining. *International Journal for Scientific Research and Development*, *3*.

Garcıa, S., Ramırez-Gallego, S., Luengo, J., Benıtez, J. M., & Herrera, F. (2016). Big data preprocessing: Methods and prospects. *Big Data Analytics*, *1*(1), 1–22. BioMed Central.

Landau, W. M. (2018). *The drake R package: A pipeline toolkit for reproducibility and high-performance computing*. *Journal of Open Source Software* (Vol. 3). Retrieved from https://doi.org/10.21105/joss.00550

Landau, W. M. (2021). *The targets R package: A dynamic make-like function-oriented pipeline toolkit for reproducibility and high-performance computing*. *Journal of Open Source Software* (Vol. 6, p. 2959). Retrieved from https://doi.org/10.21105/joss.02959

R Core Team. (2020). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Schönberg, T. (2019). *Data access to micro data of the deutsche bundesbank*. Technical Report 2019-02, Deutsche Bundesbank, Research Data; Service Centre.

van der Loo, M. P. J., & de Jonge, E. (2021). *Data validation infrastructure for R*. *Journal of Statistical Software* (Vol. 97, pp. 1–31).

Wickham, H. (2014). Tidy data. *Journal of Statistical Software, Articles*, *59*(10), 1–23. Retrieved from https://www.jstatsoft.org/v059/i10

Witt, E., & Blaschke, J. (2019). ECB data for analysis and decision-making: Data governance and technology. *IFC Bulletins chapters*, *49*. Bank for International Settlements.

Zhang, S., Zhang, C., & Yang, Q. (2003). Data preparation for data mining. *Applied artificial intelligence*, *17*(5-6), 375–381. Taylor & Francis.

A Example of a YAML file

This annex provides a concrete example for a fully fledged YAML file using a fictitious and very simple example dataset “Security Holdings of Banks Statistic (SHOBS)”. Imagine, the SHOBS to look as follows:

BANKID	DATE	ISIN	AMOUNT	DOMESTIC	COUNTRY
2002	2020-02	QT1384866837	40102	1	NA
2002	2020-02	VQ0994617539	495	1	NA
1004	2020-02	KT2096065603	1677	0	LU
2345	2020-02	QT1384866837	10424	1	NA
3456	2020-02	ET39009F7089	7579	0	AQ
3456	2020-02	VQ0994617539	NA	0	AQ

Each row of this fictitious dataset corresponds to a security (identifiable by ISIN) held by a bank (identifiable by BANKID). The dataset contains only 6 observations as of February 2020. Missing observations are indicated by NA.

For this dataset, a YAML file could look as follows:

```
main:
  short-name: SHOBS
  long-name: Security Holdings of Banks Statistic
  doi: XYZ.shobs.2002-2002.01.01
  source: RDSC
  responsible:
  - John Doe

orig:
  path: C:\Username\data\shobs
  pattern: "*shobs*"

columns:
  BANKID:
    label: Random bank identifier
    notes: Nothing special about it.
    is-key: yes
    constraints:
      type: character
      complete: yes
      unique: no
      constant-in:
      - DOMESTIC
      - COUNTRY
      greater: 1000
      in-set:
```

```
    - 1004
    - 2002
    - 2345
    - 3456
DATE:
  label: Looks like reporting date
  notes: Just an integer
  is-key: yes
  constraints:
    type: integer
    complete: yes
    unique: yes
ISIN:
  label: International security identification number
  notes: Just a string
  is-key: yes
  constraints:
    type: character
    complete: yes
AMOUNT:
  label: Looks like holding amount
  notes: Just another integer
  is-key: no
  constraints:
    type: integer
    complete: no
    greater: 0
DOMESTIC:
  label: Indicator variable
  notes: Just an indicator variable
  is-key: no
  constraints:
    type: character
    complete: yes
    in-set: [0, 1]
```