FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

# Assignment of master's thesis

| | |
|---|---|
| **Title:** | Automated data analysis pipelines |
| **Student:** | Bc. Michael Vrána |
| **Supervisor:** | doc. Ing. Filip Křikava, Ph.D. |
| **Study program:** | Informatics |
| **Branch / specialization:** | System Programming |
| **Department:** | Department of Theoretical Computer Science |
| **Validity:** | until the end of summer semester 2023/2024 |

## Instructions

Data analysis combines computation and statistics, forming a pipeline from data acquisition and cleaning through a sequence of data transformation tasks to final reporting and visualization.
This work is about developing a make-like language and runtime for defining and executing such pipelines in an automated way.
Design an embedded domain-specific language for defining such pipelines, i.e., a graph of tasks with their dependencies.
Define the model of these tasks (input, output, body) and how to specify parallel execution (patterns for dynamic branching such as map, cross or slice).
Implement a runtime for the language, which allows one to orchestrate and supervise the execution of such pipelines, including the ability to run tasks in parallel and restart failed tasks.
Document the language and test the runtime.

---

*Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 16 January 2023 in Prague.*

Master's thesis

# AUTOMATED DATA ANALYSIS PIPELINES

**Bc. Michael Vrána**

Faculty of Information Technology
Department of Theoretical Computer Science
Supervisor: doc. Ing. Filip Křikava, Ph.D.
May 4, 2023

Czech Technical University in Prague
Faculty of Information Technology


Citation of this thesis: Vrána Michael. *Automated data analysis pipelines.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

# Contents

# List of Figures



# List of Tables



# List of code listings

*First and foremost, I would like to thank my supervisor, doc. Ing. Filip Křikava, Ph.D., for the help and advice he provided me throughout the thesis. I would also like to thank my girlfriend Veronika, my family, and my friends for supporting me through my studies.*

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity

In Prague on May 4, 2023 .....

# Abstract

Data analysis pipelines describe data analysis as a sequence of interdependent steps. These pipelines enable reproducibility and effective execution of the analysis. This thesis describes the design and implementation of an R package called Pipelinr, a domain-specific language and a runtime for data analysis pipelines. The designed DSL allows the user to describe the pipeline as a set of interdependent stages. Furthermore, it allows the user to use various composable dynamic branching patterns to break down a stage into a set of tasks, which can be executed in parallel using GNU Parallel. The runtime also provides the user with metadata about the pipeline's execution, which can also be used as input to the pipeline itself.



**Keywords** R, domain specific language, data analysis, pipeline



# Abstrakt

Pipeliny pro analýzu dat popisují datovou analýzu, jako sekvenci na sobě závisejících kroků. Tyto pipeliny jsou reproducibilní a lze je vyhodnocovat efektivně. Tato práce se zabývá návrhem a implementací balíčku v jazyce R nazvaným Pipelinr, což je doménově specifický jazyk a runtime pro pipeliny pro analýzu dat. Navržené DSL umožňuje uživateli popsat pipeline jako množinu na sobě navazujících stagí. Dále umožňuje uživateli využívat různé komponovatelné vzory pro dynamické větvení, které umožňují rozpad stage na množinu úloh, jež lze parallelně spouštět pomocí GNU Parallel. Runtime nadále poskytuje metadata ohledně běhu pipeline, které lze využít jako vstup do pipeline samotné.



**Klíčová slova** R, doménově specifický jazyk, datová analýza, pipeline

# Acronyms

| | |
|---|---|
| DSL | Domain-specific language |
| URL | Uniform resource locator |
| OOP | Object oriented programming |
| API | Application programming interface |
| CPU | Central processing unit |
| AST | Abstract syntax tree |
| SSH | Secure shell |
| HPC | High-performance compute |
| CRAN | Comprehensive R Archive Network |
| HTTP | Hyper Text Transfer Protocol |
| REST | Representational State Transfer |
| JSON | JavaScript Object Notation |
| SQL | Structured Query Language |
| IDE | Integrated development environment |
| REPL | Read evaluate print loop |
| GUI | Graphical user interface |

# Introduction

Data is an essential component in the modern world. The amount of data created and captured increases rapidly [1] every year. This large amount of data can be used, for example, by businesses to make better data-driven decisions or by scientists to understand nature's phenomena. At the same time, data plays a crucial role in research. However, in recent times it has also highlighted issues in the scientific community.

The recent replication crisis in science has negatively impacted the credibility of the research. It has been shown that in fields like psychology or some branches of science, a large portion of studies could not have been reproduced [2]. Although many complex factors are playing their role in this crisis, researchers can facilitate a diverse selection of tools to make their research open and reproducible [3]. Datasets and the data analysis code can be open-sourced and published using platforms like GitHub or GitLab. The data analysis can be implemented by many different tools that can aid reproducibility at many different scales [4].

Data analysis could be thought of as a sequence of individual steps that form a pipeline. Typically the data needs to be acquired, cleaned, transformed, and evaluated. Nowadays, there are many tools that let users define and execute a data processing pipeline using different technologies [5]. Simple pipelines can be constructed using GNU Make [6], even though it is typically used as a build system. On larger scales, in the field of high-performance computing tools like DAGMan [7] for HTCondor [8] enables the user to define workflows that are then executed on a cluster, grid, or in a cloud.

A lot of data science is done using the R language [9]. Some packages in the R language ecosystem took inspiration from the GNU Make approach and expanded on it. Packages like `drake` [10] or `targets` [11] allow the user to specify the pipeline using a domain-specific language. They also offer many different execution backends, which enable the user to parallelize the pipeline execution. They even have support for pipeline execution using HPC grid engines.

This thesis describes the design and implementation of an R package called Pipelinr, which allows the user to define and execute a pipeline using a DSL. One of its goals is to enable the user to execute the pipeline in a parallel and distributed manner. It also aims to give the user the ability to introspect metadata about the pipeline's execution, which could be processed by the pipeline itself.

# Chapter 1

# Background and related work

## 1.1 R programming language

R is a programming language that is used primarily in statistical computing. It could be thought of as a DSL for statisticians and data scientists.

The R language could be thought of as a continuation of the S language, which was language for statisticians developed in 1976 at Bell Laboratories [12]. It shares a very similar syntax with S, even up to the point where some S programs could be interpreted as R programs [9]. The first release of R was in 1993 [13].

It is a high-level dynamically typed language, where every value is an object, including functions. Every primitive value in R is a vector. The benefit of this approach is that operations, which in more traditional programming languages would only operate on primitive values, work on whole vectors. This includes, for example, arithmetic or logical operators. This approach also allows R to vectorize these operations to execute them more efficiently [14]. An example of R vector operations is shown in code listing 1.

Another differentiating factor from other popular languages is the non-standard evaluation. R has functions like many other programming languages, and they are themselves an object. Arguments of functions in R are lazy evaluated, meaning that they are only evaluated once they are needed. This is in contrast to the majority of popular programming languages, which are eagerly evaluated. Lazy evaluation can be a powerful feature. However, it bears performance penalties because every function argument needs to be a promise object. An example demonstrating the benefit of non-standard evaluation can be seen in code listing 2.

```r
# Sequence operator ":" creates a vector of numbers from 1 to 5 inclusive.
numbers <- 1:5

# This operation doubles the values of elements in numbers.
doubled_numbers <- numbers * 2

# Logical operators on vector return a logical vector with
# results of a given operation on each element of the vector.
# Result of the follwing operation will be vec(FALSE, FALSE, TRUE, TRUE, TRUE).
is_number_greater_than_five <- doubled_numbers > 5
```

**Code listing 1** R code example demonstrating vector operations.

```r
# This represents a function making an expensive computation.
expensive_computation <- function() { ... }

# create_logger is a higher order function that returns a logging function
create_logger <- function(enabled) function(data) {
    # If the enabled flag is not true, the logger function will return early
    if (!enabled) return()

    print(data)
}

log <- create_logger(enabled = FALSE)

# Since logger was disabled on the previous line, R will not evaluate
# the "data" argument, thus skipping the expensive computation.
log(expensive_computation())
```

**Code listing 2** R code example demonstrating the benefit of non-standard evaluation.

While R is more focused on the functional paradigm, it also offers multiple OOP systems. The systems packaged with base R are named S3 and S4.

```r
# Cat is a function returning an object with the class "cat".
cat <- function(name) {
    structure(list(name = name), class = "cat")
}

# This defines a generic function
make_sound <- function(animal) UseMethod("make_sound", animal)

# S3 dynamic dispatch works by searching for a concrete implementation of a
# generic function with the name in the pattern of "<generic_name>.<class>".
# This is a concrete implementation of the make_sound() generic function
# for the class "cat".
make_sound.cat <- function(animal) {
    paste("Cat", animal$name, "says: meow") |> str()
}

felix <- cat("Felix")

# This will print "Cat Felix says: meow"
make_sound(felix)
```

**Code listing 3** Demonstration of the S3 object system in R.

In S3, objects can be assigned a `class` attribute. This attribute is a character vector, where each element is a class name. S3 methods are generic functions that perform a dynamic dispatch based on the `class` attribute. Inheritance is achieved by defining multiple class names in the `class` attribute character vector. The classes specified at the begging of this vector inherit from classes behind them [15]. An example of a code using the S3 object system can be seen in code

listing 3.

The S4 object system is similar to S3 but is stricter. It lets the user define a class using the `setClass()` function. Compared to S3, classes have explicit slots. Similarly to S3, a character vector is used to identify the class. Since classes are explicitly defined, they need to be instantiated using the `new()` function, as opposed to S3, where a class is assigned to an object. It's possible to specify a prototype defining default slot values. S4 defines the `@` operator for accessing the slot values. Similarly to S3, generics and concrete methods can be defined using the `setGeneric()` and `setMethod()` functions respectively.

R also has a metaprogramming functionality [15]. It provides ways for the programmer to retrieve function argument expressions' ASTs as language objects without the expressions being evaluated. Furthermore, these language objects can be inspected and modified. This facilitates a metaprogramming ability within R, which is non-standard compared to other modern programming languages.

Surrounding the language, there is a large ecosystem of packages within the CRAN repository [9]. Packages from the `tidyverse` [16] package collection use this to their advantage to essentially build DSLs. The package `dplyr` [17] specifies its own DSL for data manipulation, simplifying the process of working with data frames. It is commonly used with the `ggplot2` [18] plotting library, which also specifies its own DSL. Furthermore, the `magrittr` [19] package introduces a pipe operator, which is supported by the packages mentioned above to further expand on their DSL. Combined, they provide the user with powerful DSL that would be hard to implement in any other programming language. The code listing showcases how the `tidyverse` packages are used together. Another thing to note is that R packages' user interface is, in a lot of cases, meant to be operated from R's REPL.

Many R packages are actually not entirely written in R. This is possible because R packages provide a convenient way to interface with code from other lower-level languages, for example using the `Rcpp` [21] package and library for C++. C++ gives the programmer more control over the program execution than R does, enabling the programmer to write more efficient code than he could in R while at the same time being harder to work with.

## 1.2 Data analysis pipelines

Data analysis consists of multiple steps. Data usually needs to be collected, cleaned, transformed, and evaluated. These series of steps could be thought of as a data analysis pipeline. Each step in the pipeline depends on outputs from previous steps.

This has many benefits. Breaking down the analysis into single steps adheres to the single responsibility principle, making it easier to reason about and maintain. The tools that facilitate the creation and execution of such pipelines can leverage such features as parallel distribution or reproducibility. Furthermore, the pipeline structure allows these tools to automate many tasks [22]. They can identify which parts of the pipeline need to be evaluated without the need to evaluate the whole pipeline. Such a feature can aid the exploratory analysis by reducing the time needed to execute the given pipeline [22]. They can also automate the storage of results, which can then be published with the code of a given pipeline. In the case of compute-intensive pipelines, this can reduce the need to have access to expensive compute clusters while also preserving the ability to inspect and review the code of the analysis.

## 1.3 Existing solutions

Simple pipelines can be linear in nature. In fact, pipelines can be implemented as bash scripts. This can suffice for simple and small workflows, but even in those cases, advanced tools can aid the pipeline's development. Bash does not offer the functionality of other tools, but the users can implement them themselves. However, if they choose this approach, in the end, they

```
dataframe_solar %>%
  select(System.Size, Customer.Segment) %>%
  na.omit() %>%
  filter(System.Size > 1) %>% # miscoded data
  filter(Customer.Segment != "OTHER") %>%
  na.omit() %>%
  ggplot(aes(System.Size, fill = Customer.Segment, color = Customer.Segment)) +
  geom_density(alpha = 0.2) +
  scale_x_log10() +
  facet_wrap(~Customer.Segment)
```

**Code listing 4** Demonstration of R code using the `tidyverse` [16] package collection. The resulting plot can be seen in figure 1.1. Taken from [20].

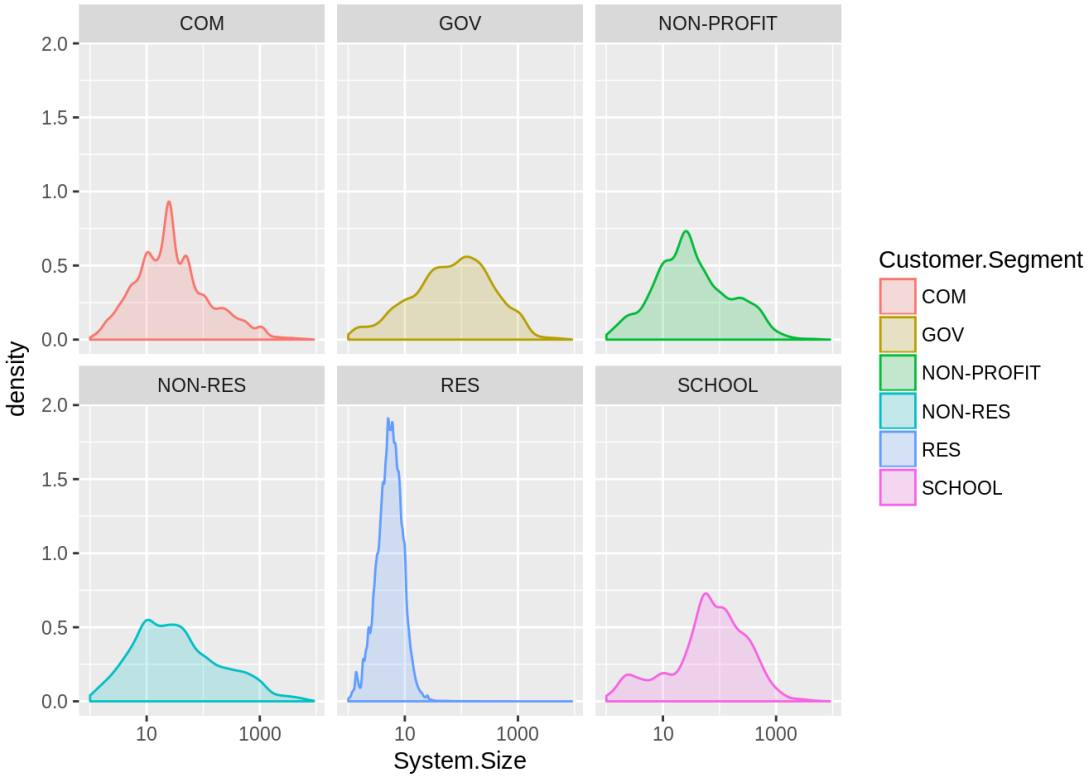


**Figure 1.1** Example of a plot produced by the code listing 4 using the `ggplot` [18] package. Taken from [20].

```
first_dataset.csv:
        curl https://get-dataset.org > first_dataset.csv

second_dataset.json:
        scp user@hostname:data.json ./second_dataset.json

second_dataset.csv:
        ./json-to-csv second_dataset.json

first_dataset_clean.csv: first_dataset.csv
        ./cleanup_data.sh first_dataset.csv

aggregated_data.csv: first_dataset_clean.csv second_dataset.csv
        Rscript aggregate.R first_dataset_clean.csv second_dataset.csv

graphs: aggregated_data.csv
        Rscript plot_graphs.R aggregated_data.csv
```

**Code listing 5** Example of a hypothetical pipeline made using GNU Make

will probably implement a poor man's version of the pipeline, which could have been better implemented by using a tool suited for the job.

Such tools can take many shapes or forms. For lower-scale analyses, R packages like `drake`, `targets` or even build systems like GNU Make can be used to implement such pipelines.

On larger scales, they can be implemented using HPC tools like DAGMan for HTCondor, although these tools are intended for a different audience than researchers and are usually coupled with a particular grid engine.

### 1.3.1 GNU Make

GNU Make [6] is a tool primarily used as a build system. Make uses a file called `Makefile` that specifies rules for each so-called target. These rules tell the Make how to build each target. It is also necessary to specify dependencies for a given target. Dependencies are also files, which themselves can be other targets. This way, the targets, and their dependencies form a dependency graph. Each time Make is run, it gathers information about which targets to rebuild based on the modification timestamp of each file in the dependency graph.

Make can be used to model a data analysis pipeline [23]. Each step of the pipeline could be modeled as a target. Since Make is language agnostic, it does not constrain users to specific technologies. The code for each target could be written in different tools, such as bash scripts of R programs. An example of such a pipeline is shown in code listing 5.

Using Make can be a good choice for simple pipelines, but it can be a hindrance for more complex ones. Results between stages need to be stored on the filesystem, which could affect the execution speed of the pipeline. Make also does not offer any functionality for distributed execution, leaving this problem to the pipeline developer. While Make offers freedom for the pipeline developers to choose their own technologies, it can create issues when building more complex pipelines. The dependency graph can be complex, making it hard to develop and maintain. This is because each step of the pipeline does not have clearly defined dependencies. For this reason, it can be challenging for the pipeline developer to navigate through the pipeline's code.

```yaml
sources:
  - code.R

targets:
  all:
    depends: plot.pdf

  data.csv:
    command: download_data(target_name)

  processed:
    command: process_data("data.csv")

  plot.pdf:
    command: myplot(processed)
    plot: true

  report.md:
    depends: processed
    knitr: true
```

**Code listing 6** Example of a Remakefile from `remake` documentation [24]

### 1.3.2 Remake

Make is primarily tailored for building software, even though it can be used for data analysis workflows. The `remake` R package takes the concept of Make and applies it to R [24]. It uses the same notion of rules, targets, and dependencies as Make does. Instead of writing a Makefile, the user defines the pipeline in a YAML file called Remakefile. Each rule has associated R commands, which define how to build the given target, similar to Make. Dependencies are defined by a target name rather than by filenames, as is the case in Make. It does not support any advanced parallel or distributed scheduling. An example of a Remakefile defining a `remake` pipeline can be seen in code listing 6.

The `remake` package is not longer maintained since 2017.

### 1.3.3 Drake

The `drake` R package is a data analysis workflow management tool [10]. It takes inspiration from Make and `remake`, and builds an R DSL on top of their concepts. At the core of its model sits a plan. A plan describes targets, and R expressions describe how to evaluate the target. Each target is an output of its command. This means that each target is represented by an R value instead of a file, as is the case in Make. Dependencies are defined simply by specifying a symbol of another target in the target's expression. An example of a `drake` plan can be seen in code listing 7.

`drake` makes a distinction between two types of branching. First is static branching. It knows how many targets will be built before executing the plan. This gives the option to users to compactly write more complex plans. To achieve this, `drake` defines four transformations: `map()`, `cross()`, `split()`, and `combine()`.

The `map()` operation takes a data frame or a variable number of named vector parameters

```
plan <- drake_plan(
  raw_data = readxl::read_excel(file_in("raw_data.xlsx")),

  data = raw_data %>%
    mutate(Ozone = replace_na(Ozone, mean(Ozone, na.rm = TRUE))),

  hist = create_plot(data),

  fit = lm(Ozone ~ Wind + Temp, data),

  report = rmarkdown::render(
    knitr_in("report.Rmd"),
    output_file = file_out("report.html"),
    quiet = TRUE
  )
)
```

**Code listing 7** Example of a `drake` plan from it's documentation [25].

and creates a new target for each of their values. When given named vector parameters, they are zipped together, and it creates a new target for each of their zipped values.

The `cross()` operation works like `map()`, except it creates a new target for every combination of the values of its arguments.

The `split()` operation takes a slice size and a data frame or a vector as its arguments as `map()` does. It then creates slices of a given size and maps them to a new target. This can be useful when the user does not want to create too many targets with small inputs.

The `combine()` is used for aggregation. The user needs to specify a symbol representing a column to group by. It then takes a data frame and performs the "group by" operation.

The second type of branching is dynamic branching. This type of branching should be used when the number of targets cannot be determined without executing the plan. Another situation where it could be helpful is when static branching produces a large number of targets. Dynamic branching is achieved by using one of the three available dynamic transformations: `map()`, `cross()` and `group()`. `map()` and `cross()` behave like their static counterparts and `group()` behaves like static `combine()`.

Every time a plan is executed, `drake` can determine a subset of targets that have been changed and, thus, which targets need to be executed. It achieves this by saving information about target definitions to the file system.

Another feature of `drake` is its integration with HPC schedulers through the `clustermq` [26] package. Because of this, targets can be executed in parallel on a compute cluster. This allows it to achieve a much higher degree of scalability than Make.

### 1.3.4 Targets

The `targets` package is a successor to `drake` and offers similar functionality [11]. Its model consists of targets, but unlike `drake`, they do not need to be assembled inside a plan object. It uses a specialized file for pipeline definition called `_targets.R`, similar to Makefile or Remakefile. Example of the `_targets.R` file is shown in code listing 8.

It supports multiple types of targets, such as files, reports, or plots, using a separate package called `tarchetypes` [28]. Archetypes are specialized targets, which can simplify the definition of common types of targets, such as `tar_read()` archetype, which reads from a file. Other

```
library(targets)
source("R/functions.R")

tar_option_set(packages = c("readr", "dplyr", "ggplot2"))

list(
  tar_target(file, "data.csv", format = "file"),
  tar_target(data, get_data(file)),
  tar_target(model, fit_model(data)),
  tar_target(plot, plot_model(model, data))
)
```

**Code listing 8** Example of a `_targets.R` file from the `targets` package documentation [27].

archetypes can produce an output of a particular type. For example, the `tar_render()` renders its input into an `Rmd` file. Some dynamic branching patterns are implemented as archetypes. Even an alternative pipeline definition syntax similar to `drake_plan()`, which does not need the `_targets.R` file, is implemented as an archetype.

Static branching is supported by the `tar_map()` archetype, which is a counterpart to the `drake` package's static transformation `map()`. Other static branching operations are supported by their respective archetypes.

`targets` expands on the dynamic branching aspect of `drake` by introducing new operations while also making the operations composable. It calls the dynamic branching operations patterns. There are patterns such as `map()`, `cross()`, and `slice()` which behave the same as their `drake` counterparts. Furthermore, it introduces `head()`, `tail()`, and `sample()` patterns.

It has a similar support for parallel execution and HPC schedulers as `drake`.

## 1.4 Summary

Data analysis can be broken down into a sequence of steps, which can depend on each other's outputs. Many tools can aid this approach by automating certain tasks and giving the user to execute the pipeline efficiently. These tools can be simple scripts or even build tools, like GNU Make, to give an example. In R, a programming language aimed at data science, a few packages expand on the philosophy of GNU Make pipelines. These tools allow the user to define and execute a pipeline using a DSL. They can efficiently evaluate the pipeline and even execute it distributively using HPC grid engines.

# Chapter 2
# Design

This chapter will describe a design of an R package called Pipelinr. Its design has two goals. First is a DSL, which is used for pipeline definition and execution. Second, a runtime focuses on providing the ability to execute a pipeline in a parallel and distributed manner. Pipelinr as a whole also aims to address shortcomings of `drake` and `targets`, such as the handling of failed tasks in a distributed environment.

To summarize, the tool should fulfill the following requirements:

- *Idempotence* – The pipeline execution should be an idempotent operation. Every time the pipeline is executed, it should reach some final state.
- *Feedback* – The runtime should give the user some feedback about the execution. Some pipelines could take a long time to execute, so the tool should report the execution progress to the user.
- *Metadata* – It should be possible to gather some metadata about the pipeline's execution. This should aid in debugging and troubleshooting of a pipeline.
- *Parallelization* – The runtime should be able to execute the pipeline in parallel and distributively.

Finally, there is one non-functional requirement to make the implementation as compact as possible regarding code size and the number of used dependencies.

## 2.1 Running example

Throughout this chapter, we will use a running example. This example describes a simple data analysis pipeline. The naive implementation in the form of an R script file can be seen in code listing 9. By being a simple Rscript file, it lacks any features that one would from a data analysis pipeline expect. The user cannot select a part of the pipeline to rerun, nor are the results cached. The pipeline is not separated into clearly distinct steps and does not provide any feedback about the failures of each step. The pipeline also is not parallelized and would not scale well when it would work with larger datasets. Since it is a small pipeline, these problems are not significant. Given the dataset's small size and small pipeline complexity, this simple implementation works well.

The pipeline consists of multiple steps. First is the data acquisition step. In this step, the pipeline fetches a list of Nigerian and Turkish cities' names from the Countries Now public REST API [29]. Then it uses the same API to fetch each city's population. In the next step, the pipeline

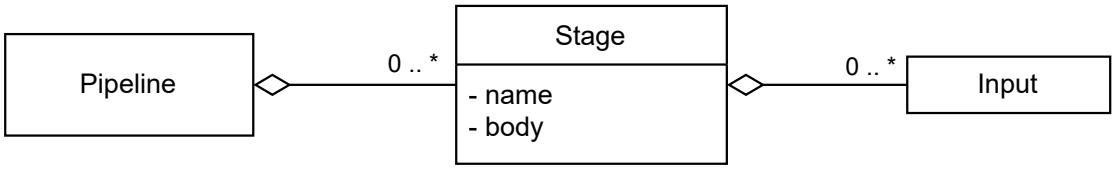


**Figure 2.1** Diagram describing the pipeline's model

computes the total city population for both countries. The final step creates a chart comparing both countries' total city populations. The described pipeline is an example and should be viewed as such, the described analysis is by no means accurate nor uses proper data.

To simplify the pipeline's implementation, multiple R packages have been used. The `httr` [30] and `jsonlite` [31] packages have been used to send an HTTP request with JSON body payload. The data frame manipulation is facilitated by the `dplyr` [17] package. The plot is created using the `ggplot2` [18] packages and a few functions are also used from the package `purrr` [32].

## 2.2 Model

At the core of Pipelinr's model is a *pipeline*. A pipeline is composed of multiple *stages*. A stage describes a simple computation. This is analogous to a target in `drake` or `targets`.

Stages are described using two components: a body function used to execute the stage and inputs. Stage essentially could be considered a function, taking some inputs and producing some results. Every stage also has a unique name in a given pipeline. Stage results can be passed to other stages in the pipeline as their inputs. This allows the user to define a dependency graph of the pipeline's stages. This graph cannot contain cycles. It needs to form a directed acyclic graph.

A diagram describing the composition of pipelines and stages is shown in figure 2.1.

## 2.3 Pipeline construction in DSL

Pipelinr's DSL provides two main functions to build pipelines. The stage is constructed using the `stage()` function, and the pipeline is constructed using the `make_pipeline()` function. Stages are passed to the pipeline constructor as named arguments, where the name also defines the stage's name. This ensures that every stage name is unique. Based on the running example, we could create the first two stages, which retrieve Nigerian and Turkish cities, by creating a body function that calls the `get_cities()` function and pass it to the `stage()` function:

```
make_pipeline(
    nigerian_cities = stage(function() get_cities("nigeria")),

    turkish_cities = stage(function() get_cities("turkey"))
)
```

The code snippet above defines the stages `nigerian_cities` and `turkish_cities`. These stages do not take any inputs. To construct a stage that takes the results of these stages as its input, we could create another stage whose body function arguments will match the stages' names:

```
library(httr)
library(jsonlite)
library(dplyr)
library(purrr)
library(ggplot2)

cities_by_country_url <-
    "https://countriesnow.space/api/v0.1/countries/cities"

city_population_url <-
    "https://countriesnow.space/api/v0.1/countries/population/cities"

get_cities <- function(country) {
    body <- list(country = country) %>% toJSON(., auto_unbox = TRUE)

    POST(
        cities_by_country_url,
        add_headers("Content-Type" = "application/json"),
        body = body
    ) %>%
        content() %>%
        pluck("data") %>%
        unlist() %>%
        data.frame(name = ., country = country)
}

get_city_population <- function(city) {
    body <- list(city = city) %>% toJSON(., auto_unbox = TRUE)

    population_counts <- POST(
        city_population_url,
        add_headers("Content-Type" = "application/json"),
        body = body
    ) %>%
        content() %>%
        pluck("data", "populationCounts", 1, "value") %>%
        as.numeric()
}

nigerian_cities <- get_cities("nigeria")
turkish_cities <- get_cities("turkey")

cities <- bind_rows(nigerian_cities, turkish_cities)

cities$population <- map(cities$name, get_city_population) %>% unlist()

total_city_population <- cities %>%
    group_by(country) %>%
    summarize(total_population = sum(population))

ggplot(total_city_population, aes(x = country, y = total_population)) +
    geom_col() +
    xlab("Country") +
    ylab("City population") +
    ggtitle("Total city population by country")

ggsave("total_city_population.png")
```

**Code listing 9** Naive implementation of a data analysis pipeline in the form of an R script file.

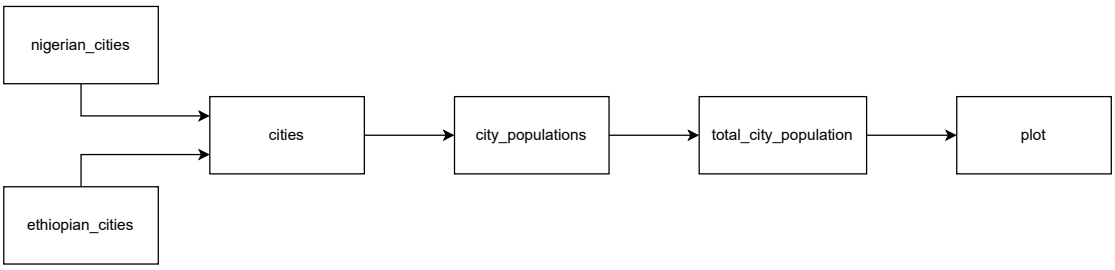


**Figure 2.2** Stage dependency graph of a pipeline from the running example

```
make_pipeline(
    nigerian_cities = stage(function() get_cities("nigeria")),

    turkish_cities = stage(function() get_cities("turkey")),

    cities = stage(function(nigerian_cities, turkish_cities) {
        bind_rows(nigerian_cities, turkish_cities)
    })
)
```

If we would like to model the rest of our running example as a Pipelinr pipeline, the stage dependency graph could look like it is described in figure 2.2. Code listing 10 shows the whole pipeline from our running example described as a Pipelinr pipeline.

The examples above defined stage dependencies by reusing a stage name as another stage's body argument name. The stage dependencies can also be defined by the `stage_inputs()` function. The `stage_inputs()` function defines a mapping from a stage body argument to a stage name using the named parameters syntax, similar to how stages are given names in a pipeline. The following snippet is an equivalent stage definition of the `cities` stage from the running example, but it uses the `stage_inputs()` function:

```
make_pipeline(
    ...
    cities = stage(
        inputs = stage_inputs(
            nigerian = nigerian_cities,
            turkish = turkish_cities
        ),
        body = function(nigerian, turkish) {
            bind_rows(nigerian, turkish)
        }
    ),
    ...
)
```

Defining dependencies using `stage_inputs()` can be combined with those defined using the body function's argument names. A conflict could arise if the body function argument name matches the name of an input defined in the `inputs` argument. If such conflict arises, the one defined by the `inputs` argument takes precedence.

The omitting of the `inputs` argument of the `stage()` function could be thought of as a shorthand definition of dependencies. It should make the definition of simpler stages require fewer lines of code and thus be more readable. The drawback is that it does not allow the user to define

```
make_pipeline(
    nigerian_cities = stage(function() get_cities("nigeria")),

    turkish_cities = stage(function() get_cities("turkey")),

    cities = stage(function(nigerian_cities, turkish_cities) {
        bind_rows(nigerian_cities, turkish_cities)
    }),

    city_populations = stage(function(cities) {
        cities$population <- map(cities$name, get_city_population) %>% unlist()
        cities
    }),

    plot_city_population_by_country = stage(function(city_populations) {
        total_city_population <- city_populations %>%
            group_by(country) %>%
            summarize(total_population = sum(population))

        ggplot(total_city_population, aes(x = country, y = total_population)) +
            geom_col() +
            xlab("Country") +
            ylab("City population") +
            ggtitle("Total city population by country")

        ggsave("total_city_population.png")
    })
)
```

**Code listing 10** Pipeline from the running example written using Pipelinr

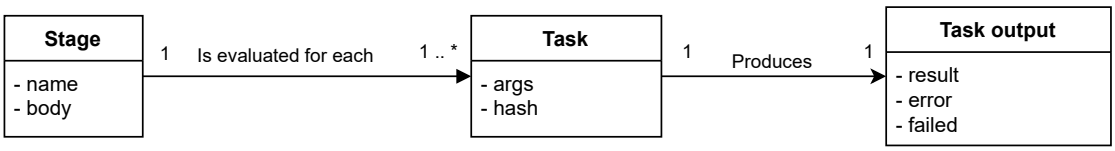


**Figure 2.3** Diagram describing the model of tasks and task outputs.

more complex dependencies between stages. This is because the values of the `stage_inputs()` do not just need to be stage identifiers. They can be DSL expressions that define more complex dependencies. Using these expressions, the user can utilize dynamic branching.

## 2.4 Dynamic branching

*Dynamic branching* is a technique that allows running a stage body function multiple times for a dynamic number of arguments. One of the main benefits of dynamic branching is that it allows runtime to parallelize the execution of a given stage.

Pipelinr's model has a notion of tasks. A task is essentially a list containing a named list of body function's arguments. When a stage is run, the body function is called for every task's arguments. Pipelinr assumes that stage body functions are pure functions, meaning that they return the same value for the same arguments. This means that the stage would produce the same output for each task. One of the benefits of this assumption is that the tasks can be identified by their hashes.

When a task is evaluated, it produces outputs. These outputs contain the result, which is the value return value of the body function. However, not every invocation of the body function successfully returns a value because it can throw an error. Pipelinr also saves these errors and a flag indicating whether the task evaluation has failed or not. This is important because when the stage is used as an input for another stage, a task is created for every non-failed task's result of the stage. A diagram describing the model of tasks is shown in figure 2.3.

Pipelinr defines a set of DSL functions that are available inside the `stage_inputs()` expressions. These functions serve two purposes. The first purpose is to allow the user to use dynamic branching. The second purpose is to allow the user to preprocess the stage inputs. These DSL functions are named using verbs in the past perfect tense, meaning they use the "-ed" suffix. This naming scheme, however, is not suitable for all functions. The available DSL functions are:

- `mapped()` – Used to produce tasks from atomic vectors, lists or data frames.
- `remapped()` – Remaps input values using a provided function.
- `filtered()` – Filters out inputs.
- `chained()` – Concatenates inputs to form a single input.
- `zipped()` – Combines values yielded by the input iterators into tuples.
- `crossed()` – Produces all possible combinations of inputs.
- `take()` – Filters out first $n$ inputs.
- `metadata()` – Retrieves metadata about another stage's execution. It's discussed in a later section.

Pipelinr assumes that the `stage_inputs()` expressions do not contain any assignments. They are intended to be function calls composed by function composition or pipe operator. Their behavior is undefined if any of these expressions contain assignment or super-assignment.

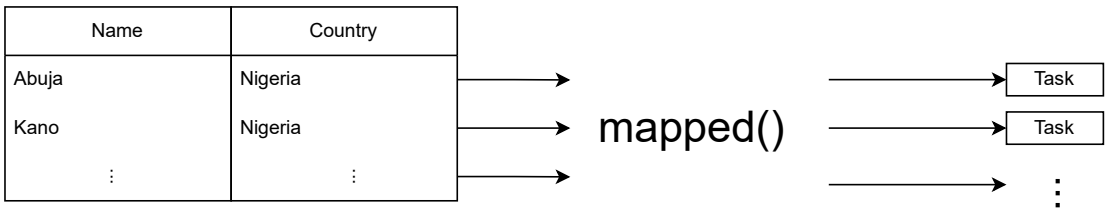


**Figure 2.4** Diagram demonstrating how rows of a data frame are mapped to a task using a `mapped()` function

The simplest dynamic branching pattern is `mapped()`. We will explain it using our running example. In the example, the `city_populations` stage makes a synchronous HTTP request for every city in our dataset. Because of this, the stage could take a lot of time to execute. Ideally, we would like to execute these requests in parallel. We will use the `mapped()` function to create a task for each row in the `cities` stage's results. This means the `city_populations` stage body will be called for each city. An example of this modified pipeline can be seen in example 11. Figure 2.4 demonstrates how the rows of the result of the `cities` stage are mapped to individual tasks.

Notice how the `city_populations` stage results need to be collected using the `collect_df()` function. When a stage produces a certain number of results, they are then passed to the next stage in separate tasks. Assuming that all tasks successfully evaluate, then the number of tasks is preserved throughout the stages until the input values are collected to a single object using the `collect()` or `collect_df()` function.

The results of expressions defined by the `stage_inputs` functions are iterators, on which the DSL functions like `mapped()` operate. Even when the result of the expression is not an iterator object, it is wrapped in an iterator that yields the result of the expression. An *iterator* is a design pattern that is used to abstract away the underlying collection when iterating over it [33]. Pipelinr comes with a small iterator library that implements basic operations on iterators such as fold, map, filter, and more.

The `mapped()` function takes as an argument an iterator or a list to make it more versatile. If a list was passed as an argument, it's wrapped in an iterator, and the function behaves as if the iterator was passed to it. It expects that the iterator will yield only lists or data frames. It then converts those lists to iterators that yield the list's values. If it encounters a data frame instead of a list, it's treated as a list of rows. After that, those iterators are concatenated to form a single iterator. It is similar to a flat operation on an iterator of lists. The following snippet demonstrates the behavior of `mapped()`:

```
numbers <- 1:3
strings <- c("a", "b")

# Yields 1, 2, 3.
mapped(numbers)

# Iter yields 1:3, c("a", "b").
iter <- concat_iter(make_iter(numbers), make_iter(strings))

# Yields 1, 2, 3, "a", "b".
mapped(iter)
```

```
make_pipeline(
    ...
    city_populations = stage(
        inputs = stage_inputs(
            city = mapped(cities)
        ),
        function(city) {
            city$population <- get_city_population(city$name)
            city
        }
    ),

    total_city_population = stage(
        inputs = stage_inputs(
            cities_with_populations = collect_df(city_populations)
        ),
        function(cities_with_populations) {
            cities_with_populations %>%
                group_by(country) %>%
                summarize(total_population = sum(population))
        }
    ),
    ...
)
```

**Code listing 11** Pipeline from the running example making use of dynamic branching

Since a stage can produce multiple results, the results are also represented by an iterator of their results. Because stage inputs evaluate to iterators and stage outputs are also iterators, a stage could be considered a function taking a variable number of iterator arguments and returning an iterator to its results. The input iterators are zipped together to form a single iterator returning a list of named arguments for the body function. This list of body function's arguments is called a task in the Pipelinr's model.

Pipelinr supports more dynamic branching patterns such as cross using the `crossed()` function, which creates an iterator yielding all combinations of values from both argument iterators.

The `filtered()` function allows users to filter out iterators elements based on a predicate. It takes two arguments. The first is an iterator, and the second is a predicate function. This function needs to take a single argument, which is a value of the iterator. It also needs to return a boolean value, which is a logical vector of length one in R's terms. In the running example, we could construct a new stage, which could take as input only Nigerian cities with their respective population. Using the `filtered()` function, this hypothetical stage could look like the following snippet:

```
make_pipeline(
    ...
    filtered_stage = stage(
        inputs = stage_inputs(
            nigerian_cities_with_population = filtered(
                city_populations,
                function(city_with_population) {
                    city_with_population$country == "nigeria"
                }
            )
        ),
        body = function(nigerian_cities_with_population) { ... }
    ),
    ...
)
```

## 2.5 Pipeline execution

Once the pipeline is defined, the user can execute the pipeline. Pipelinr's API is designed to be operated from the R REPL. The function that facilitates the pipeline execution is `make()`. The `make()` function executes a pipeline by evaluating stages in their topological order.

By default, `make()` expects the pipeline to be defined in a R source file named `pipeline.R` in the current working directory. This is similar to a `_targets.R` file from `targets`. The last expression in this file must evaluate to a pipeline object. Every time `make()` is invoked, it evaluates the pipeline definition file, meaning that all changes made to the pipeline definition should be reflected in the current `make()` invocation. Pipelinr also offers an option to pass the pipeline object directly to the `make()` function in the `pipeline` argument. Using the `pipeline` argument has the drawback of requiring the user to rebuild the pipeline himself with every change.

Pipelinr persists all tasks on the filesystem. When a pipeline is run, `make()` searches for unevaluated tasks for the currently executed stage on the filesystem and evaluates them. When a task is evaluated, its outputs are also stored on the filesystem. Thanks to this, by default, only the unevaluated tasks will be evaluated when the user reruns the pipeline. This makes a `make()` an idempotent operation that puts the pipeline in the same state every time it is run. Another

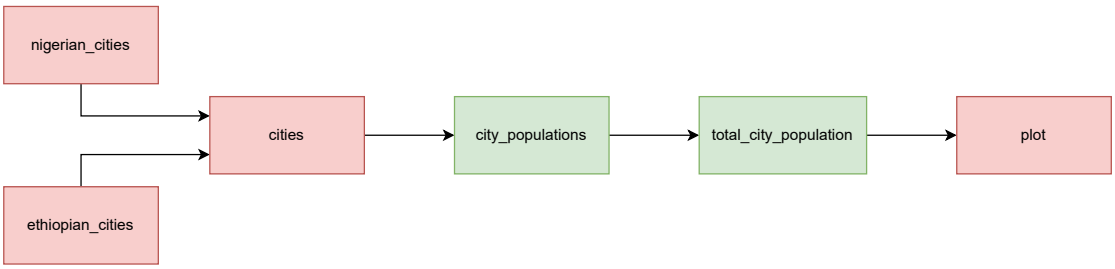


**Figure 2.5** Stage dependency graph of a pipeline from the running example visualizing which stages are evaluated by using the `only` stage filter with the following expression: `make(c(city_populations, total_city_population))`
Stages colored in red are not evaluated, while stages colored in green are evaluated.

benefit is that if a pipeline execution is somehow interrupted or, for some reason, does not run into completion, it will continue from the exact point it left off the next time it is run.

The tasks and their outputs are preserved even when a stage's implementation changes to prevent unintentional data loss. This allows the user to make changes to the pipeline without the fear of losing the previous tasks and their outputs. If the user wishes to clean the old saved tasks with their outputs, he can set the `clean` flag to `TRUE`, which forces `make()` to delete the stage's persisted tasks and outputs before stage evaluation. This way the stage's tasks are always up to date the same way as if the pipeline was run from a completely unevaluated state.

The `make()` function takes an optional first argument named `only` that filters out stages by their names. If the user, for example, would like to run only the `city_populations` stage from our running example, he can call the `make()` while passing a symbol with the same name as the stage like so:

```
make(city_populations)
```

This would have the effect of skipping the evaluation of all stages except the `city_populations` stage. Beware that in the case where previous stages of currently filtered stages are not evaluated, the filtered stages may be missing their inputs. Because of this, their evaluation will not create any outputs. If a user would like to run more stages than a single stage, he can pass an expression that returns a list of stages. An example of this expression that filters out multiple stages could be written as a function call to the `c` function:

```
make(c(city_populations, total_city_population))
```

This would have the effect of filtering out the `city_population` stage together with the `total_city_population` stage. The `only` argument expression is being evaluated in an environment where each stage name is available as a variable with its name. Pipelinr expects the result of this expression to be a character vector. This allows the user to select multiple stages using the `c()` function, which combines all of the character vectors. Another option is to use string literals directly:

```
make(c("city_populations", "total_city_population"))
```

Because the snippet above evaluates to a character vector, it is a valid stage filter expression. However, compared to symbols, the string literal is more cumbersome to type.

Similarly to the `only` filter, `make()` accepts another optional argument named `from`. The `from` filter filters out the specified stages using the expression the same way `only` does, but it also includes successive stages of the filtered-out stages. Figure 2.6 demonstrates the `from` filter on the running example.

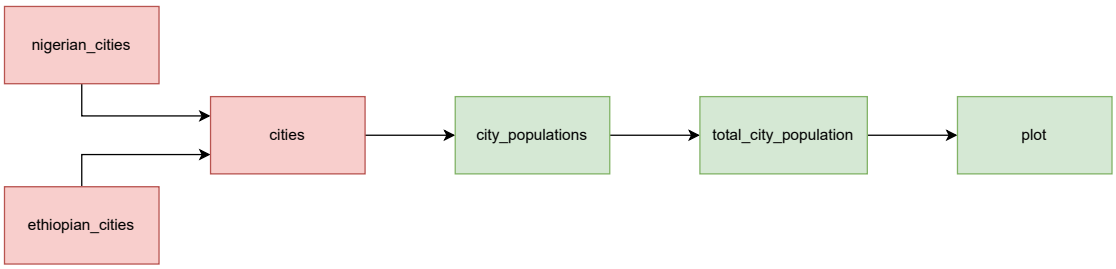


**Figure 2.6** Stage dependency graph of a pipeline from the running example visualizing which stages are evaluated by using the `from` stage filter with the following expression: `make(from = city_populations)`
Stages colored in red are not evaluated, while stages colored in green are evaluated.

During pipeline development, the user might wish to reevaluate some tasks. An example scenario would be a bug in a stage's implementation that affects only a handful of tasks. The user might want to rerun tasks that do not satisfy the default condition of being unevaluated and instead define his own conditions under which tasks for evaluation should be selected. Pipelinr allows the user to provide a task filtering expression in the `make()` function named `filter`. The `filter` argument is an expression that must evaluate to a boolean value, which is a logical vector of length one in R's terms. The expression is being evaluated for every task in a data-masked environment, where tasks and their metadata elements are accessible by identifiers of their respective names. If the expression evaluates to `TRUE`, the task is selected for evaluation, otherwise it is skipped. Let us imagine that in our running example, we have discovered a bug in the implementation of the `city_population` stage. We have fixed a bug that was causing a handful of tasks to fail with an error, thus producing no results. In this case, we would like to rerun only the affected tasks. This could be accomplished by using a combination of the `from` filter to make the pipeline run only from the affected stage and using the `filter` task filter to select only failed or unevaluated tasks. A call to the `make()` function with the aforementioned filters could look like this:

```
make(from = city_populations, filter = failed || !exists(result))
```

After a pipeline is executed, stage results can be retrieved using the `read()` and `read_df()` functions. The `read()` function returns all results in a list, while `read_df()` returns the results as a data frame. The data frame can be constructed by concatenating data frames or named lists, where each named list will be a row in the returned data rame.

## 2.6 Executors

Pipelinr offers the user an option to select so-called task executors. The task executor is responsible for executing the stage's tasks. In the current implementation, Pipelinr supports two executors: the default R executor, which executes the tasks in the same R process, and a GNU Parallel executor, which, as its name suggests, executes tasks using GNU Parallel [34].

The R executor is simple. It is the default executor and is primarily intended to be used with smaller workloads. It may be useful during the development of the pipeline, where the pipeline could be tested with just a fraction of the input dataset. It also does not have the overhead costs of a GNU Parallel executor, which needs to start a new process for each task or even establish SSH connections.

As mentioned, the GNU Parallel executor is used to parallelize the execution of the stage's tasks. Unlike the R executor, the GNU Parallel executor needs to be instantiated using the

```
make_pipeline(
    ...,
     city_populations = stage(
         inputs = stage_inputs(
             city = mapped(cities)
         ),
         body = function(city) {
             city$population <- get_city_population(city$name)
             city
         },
         executor = gnu_parallel_executor
     ),

     city_populations_meta = stage(
         inputs = stage_inputs(
             meta = metadata(city_populations)
         ),
         body = function(meta) {
             # this function will be called with metadata
             # of each task from city_populations stage
             ...
         }
     ),
    ...
)
```

**Code listing 12** Pipeline from code listing 11 with an added stage `city_populations_meta` that demonstrates how metadata could be consumed.

`make_gnu_parallel_executor` function. This function takes a single optional argument `ssh_login_file`, which is a string path to the SSH login file. If this option is specified, the GNU Parallel will execute tasks over SSH on the specified hosts. It will essentially pass the argument to the `--sshloginfile` option of GNU Parallel. If the argument is missing or if it is an empty string, the tasks will be executed as child processes on the same machine as the master process.

The user can specify a default executor for a given pipeline in the `executor` argument of the `make()` function. Pipelinr also allows the user to specify an executor of a single stage by using the `executor` argument of a `stage()` function. The value of this argument takes precedence over the pipeline's executor.

## 2.7 Metadata

When a task is evaluated, it also produces metadata about the task's execution along with the task's result. This metadata is accessible to any stage using the `metadata()` DSL function inside input expressions. It accepts a stage identifier as a single argument and returns an iterator to the stages metadata. Code listing 12 demonstrates how metadata could be consumed by a pipeline's stage.

Another option is retrieving the metadata usingthea `metadata()` function that is used outside of the `stage_inputs()` expressions. This function only shares the name with the function that is available in the input expressions, the only difference being that the function accessible in the input expressions returns an iterator, while the other function returns a collected list of each

task's metadata. Pipelinr also implements a function `metadata_df()` that returns the metadata in a data frame. Both of these functions accept a stage's symbol as a parameter. A metadata object is a named list with the following elements:

- `hash` – Task's hash. This is supposed to be the task's identifier.
- `args` – A named list containing arguments that were passed to the body function.
- `result` – The value that was returned from the `body` function. This will be missing in the case of an unevaluated task or in the case of a failed task evaluation.
- `failed` – A boolean value signaling whether the task evaluation has failed or not.
- `error` – An error message captured during the evaluation of the task.
- `stdout` – Stdout output captured during the execution of the task.
- `stderr` – Stderr output captured during the execution of the task.
- `started_at` – Timestamp of the task evaluation start time.
- `duration` – Time it took for the task to be evaluated.
- `exit_code` – A worker process exit code. This will only be available if the task was evaluated using a GNU Parallel executor.

  If the task was not evaluated, all elements except `hash` and `args` will be missing.

# Chapter 3

# Implementation

Pipelinr is implemented as an R package, according to the conventions and guidelines described in the R packages book [35].

## 3.1 Pipeline construction

In this section, we will describe what happens during the pipeline construction. Pipeline is constructed by passing stage objects to the `make_pipeline()` function as named arguments, which produces the following syntax:

```
make_pipeline(
    stage_identifier_1 = <stage_object>,
    stage_identifier_2 = <stage_object>,
    ...
    stage_identifier_n = <stage_object>
)
```

Implementation of this DSL syntax is simple. The `make_pipeline()` simply takes a variable number of arguments using the ellipsis syntax.

The `make_pipeline()` function has two primary responsibilities. The first responsibility is to assemble stage objects inside a named list, where each stage is accessible by its name. The second responsibility is to search for dependencies between stages.

Stage objects are composed of two elements: a body function and an inputs object. Input objects are constructed using the `stage_inputs()` function, which captures the expressions of its named arguments. These expressions are captured as a list of quosures using the `enquos()` function from the `rlang` [36] package.

To explain quosures, we need to first dive deeper into R metaprogramming. As mentioned in the first chapter, R uses a non-standard evaluation. This means that argument values inside a function are internally represented by a promise [15]. This promise contains an expression and an environment in which the expression is meant to be evaluated. By accessing the argument's value inside a function, the promise gets evaluated, and the result is stored inside the promise. The next time the promise gets used, the expression inside the promise does not get reevaluated, instead the stored result is used. The following example discusses the quirks of non-standard evaluation:

```
foo <- function(a, b) b + a

x <- 1
y <- 2

foo(x, { x <- 2; y })
```

The call to `foo()` in this example will return 4. This is because the promise for argument `b` gets evaluated before the promise for argument `a`. If we would change the order of the operands in the addition expression to `a + b`, the return value would be 3. The drawback this has is that it could make R code less predictable than code written in eagerly evaluated languages.

R functions have the capability to capture the expressions that have been passed to them as arguments. In base R this can be done using the `substitute()` function. The `substitute()` function extracts the expression of the promise without evaluating the promise. A quosure is just a pair containing an expression and an environment in which it should be evaluated.

There is no direct need for Pipelinr to use quosures, the expressions could be captured using `substitute()`, and their environments could be retrieved using `parent.frame()`. Quosures are used because it simplifies the implementation, and the author did not want to reinvent the wheel.

During pipeline construction, the expressions in the list of quosures that were captured by the inputs object are searched for other stage identifiers. The AST of these expressions is recursively walked until a global identifier with the same name as a stage is found.

R language objects can be of the following types:

- *Literal value* – Examples: `1`, `0.5`, `TRUE`, `"Hello world"`
- *Symbol* – represents a variable. Examples: `x`, `variable`, `print`
- *Pairlist* – as it's name suggests, this is a list of pairs. You can encounter pairlists in the definition of functions arguments. This is because function arguments can have default values, thus they need to represented by pairs.
- *Call expression* – these expressions include everything else that wasn't covered by the types above. This includes, as one would expect, function calls, but it also includes operators and even blocks. In R, operators are functions that just happen to use a special syntax. Blocks and semicolons are also operators, thus they are also functions.
  Examples: `foo(1, 2)`, `1 + 2`, `function(x) x + 1`

The code that searches for dependencies in the `stage_inputs()` expression first gathers all used symbols in a given expression. Then it filters out only those symbols that match other stages' names. The function that searches for the symbols is named `find_symbols()`. Its implementation is relatively straight forward, it simply needs to handle all types of the language objects.

If the currently walked AST node is a symbol, we check if it is a stage name, and if it is, we return the string value of the symbol wrapped in a list. In the case of a literal value, an empty list is returned. Finally, in the case of a pairlist or a call, we can convert them to a list and recurse on each element of it's elements. After that, we concatenate the results and return them as a single list. The implementation of the `find_symbols()` function is shown in code listing 13.

It should be noted that this used symbol analysis is an approximation. If we would analyze an expression that would also make use of metaprogramming, for example, by using some of the functions from the `dplyr` package, we could possibly find symbols that refer to variables in a data-masked environment. Example of such a pipeline where this analysis would break down can be easily constructed and could look like this:

```
# Recursively walks language objects to retrieve used symbols
find_symbols <- function(expr) {
    # In the case of a literal return an empty list
    if (rlang::is_syntactic_literal(expr)) {
        return(list())
    }

    # In the case of symbol, return it's string representation
    if (is.symbol(expr)) {
        return(list(as_string(expr)))
    }

    # In other cases, the AST node is either list or a pairlist.
    # In both those cases we recurse on their child nodes.
    purrr::map(as.list(expr), find_symbols) %>% purrr::flatten()
}
```

**Code listing 13** Recursive function for searching all used symbols in a language object

```
data <- data.frame(a = 1:10)

make_pipeline(
    a = stage(function() {...}),
    b = stage(
        inputs = stage_inputs(x = data %>% dplyr::filter(a > 5)),
        body = function(x) {...}
    )
)
```

In the snippet above, stage `b` clearly does not depend on stage `a`. However, because our analysis of used symbols returns all symbols inside a given expression, it would find a dependency between stages `a` and `b`. Such issues cannot be easily remedied.

After the stage dependencies are found, Pipelinr sorts stages by their topological ordering. This is an invariant of the pipeline objects, which simplifies the implementation of code that uses them.

## 3.2 Iterators

An *iterator* is a design pattern used to abstract away a collection when iterating over it [33]. Pipelinr's iterator implementation has been largely inspired by the ECMAScript iterators [37]. Their structure is similar. Pipelinr's iterators are represented by a list containing three named elements:

- `value` – value yielded by the iterator.
- `done` – logical vector of length one indicating whether the iterator has finished or not.
- `next_iter` – function returning the next iterator.

The main difference from the ECMAScript iterators is that Pipelinr's iterators are immutable. The `next_iterator()` function returns a new iterator with the next value. When an iterator reaches its end, the next iterator it returns is an empty iterator.

```
# Create an iterator of 10 random numbers between 0 and 1.
iter <- runif(10) %>% vec_to_iter()

# Find the greatest value yielded by the iter
fold_iter(iter, function(acc, curr) {
    if (curr > acc) curr else acc
})
```

**Code listing 14** Example usage of the `fold_iter()` function

```
collect <- function(iter) fold_iter(iter, list(), function(acc, curr) {
    c(acc, list(curr))
})
```

**Code listing 15** Implementation of the `collect()` function built upon the fold operation.

An empty iterator is created by the `make_empty_iter()` function. The empty iterator's `value` is always `NULL`, and the `done` flag is always `TRUE`. The next iterator it returns is always also an empty iterator.

Iterators can be constructed using the `make_iter()` function. It takes two arguments, first is a value and the second is an optional `next_iter()` function that defaults to `make_empty_iter()`. If the second argument is not specified, it creates an iterator yielding a single value.

Pipelinr further defines more functions that work with iterators. The `fold_iter()` functions takes three arguments, an iterator, an initial accumulator value, and a folding function. This folding function takes two arguments, an accumulator and the current value. It then calls the folding function with each iterator's value and with the result of the previous invocation in the first argument. Code listing 14 demonstrates the usage of `fold_iter()`.

During Pipelinr's development, the `fold_iter()` function was written in a more functional manner. It consumed the iterator recursively, which in a more standard language, would benefit from the recursive tail call optimization. However, the R runtime does not do this optimization, and because of that, this function did not work with large iterators and had to be rewritten using a loop.

On top of `fold_iter()`, we can build more functions. One of those functions is a `collect()` function that collects the iterators into a list. The implementation of this function is simple, it invokes `fold_iter()`, with an initial accumulator value of an empty list, and then it appends each element to the list, as is described in code listing 15.

Similarly, the `collect_df()` function is used to collect an iterator yielding named lists or data frames into a single data frame. It is implemented as a wrapper over the `collect()` function, which converts the collected list into a data frame using the `bind_rows()` function from the `dplyr` [17] package.

The `for_each_iter()` function is primarily intended to be used for making side effects based on iterators values. In addition to an iterator, it takes a function, which is called for every value yielded by the iterator.

A useful function when working with iterators could be `map_iter()`. This function works on iterators similarly to how, for example, `lapply()` or `map()` from the `purrr` [32] package work on lists and atomic vectors. It returns a new iterator whose values are mapped through the mapping function. The function is implemented to map the values lazily only when the iterator is iterated through the `next_iter()` function call. It takes as its input two arguments, an iterator, and a mapping function, that takes in a single argument, which is the current value of the iterator. The implementation is simple and straightforward. It returns a new iterator with the value remapped

```
iter <- as_iter(1:3)

# Yields 1, 4, 27
map_iter(iter, function(x) x ^ x)
```

**Code listing 16** Example usage of the `map_iter()` function.

```
filter_iter <- function(iter, predicate) {
    # Find first element which satisfies the predicate.
    while (!iter$done && !predicate(iter$value)) iter <- iter$next_iter()

    # If the iterator was consumed, return an empty iterator.
    if (iter$done) {
        return(make_empty_iter())
    }

    # Return a new iterator with the found value and recursively continue
    # when next_iter() is called.
    make_iter(
        value = iter$value,
        next_iter = function() filter_iter(iter$next_iter(), predicate)
    )
}
```

**Code listing 17** Implementation of the `filter_iter()` function.

using the mapping function. The `next_iter()` function calls the `map_iter()` function again to map the values recursively throughout the returned iterators. Code listing 16 demonstrates how it can be used.

Pipelinr implements a `filter_iter()` function that allows the users to filter out only certain iterators based on their value. One could see it as the `keep()` function from the `purrr` package that works on iterators. It works similarly to `map_iter()` in the sense that it filters values lazily without the need to collect the whole iterator first. Its implementation is shown in code listing 17.

Iterators can be concatenated together with the `concat_iter()` function. This function takes a variable number of iterators and concatenates them together to form a single iterator yielding values from the argument iterators. An example demonstrating its usage can be seen in code listing 18.

The `zip_iter()` function zips iterators together to form a single iterator returning a list containing their values. It takes a variable number of iterator arguments. The names of the

```
iter1 <- as_iter(1:3)
iter2 <- c("a", "b") |> as_iter()

# Yields 1, 2, 3, "a", "b".
concat_iter(iter1, iter2)
```

**Code listing 18** Example usage of the `concat_iter()` function.

```
iter1 <- as_iter(1:3)
iter2 <- c("a", "b") %>% as_iter()
iter3 <- c(TRUE, FALSE) %>% as_iter()

zip_iter(iter1, string = iter2, iter3)
# The resulting iterator yields the following values:
# list(1, string = "a", TRUE)
# list(2, string = "b", FALSE)
# list(3, string = NULL, NULL)
```

**Code listing 19** Example usage of the `zip_iter()` function.

```
iter1 <- as_iter(1:3)
iter2 <- c("a", "b") %>% as_iter()

cross_iter(iter1, string = iter2)
# Yields the following values:
# list(1, string = "a")
# list(2, string = "a")
# list(3, string = "a")
# list(1, string = "b")
# list(2, string = "b")
# list(3, string = "b")
```

**Code listing 20** Example usage of the `cross_iter()` function.

arguments are propagated to the resulting lists. The returned iterator has the same length as the longest iterator in the arguments. If an argument iterator has a shorter length than others, `NULL` will be assigned to the resulting lists instead of its values once the iterator is done. A demonstrative example of its usage is shown in code listing 19.

The `cross_iter()` function creates all possible combinations of iterator values. It also takes a variable number of iterator arguments as `zip_iter()` and propagates the arguments to the resulting lists. If any of the argument iterators is done, then it returns an empty iterator.

The iterator functions are available in `stage_inputs()` expressions under different names as dynamic branching functions. Table 3.1 describes how the functions are renamed.

**Table 3.1** Mapping of iterator functions to dynamic branching functions in the `stage_inputs()` expressions.

| Iterator function | Dynamic branching function |
|---|---|
| `map_iter` | `remapped` |
| `filter_iter` | `filtered` |
| `concat_iter` | `chained` |
| `zip_iter` | `zipped` |
| `cross_iter` | `crossed` |
| `head_iter` | `take` |

## 3.3 Pipeline execution

Pipeline execution is facilitated by the `make()` function. It needs to load a pipeline if it wasn't provided. After that, it needs to decide which stages to execute based on the provided stage filters. Then it creates a pipeline directory with stage directories. Finally, it executes the filtered out stages.

*Pipeline loading.* When invoked, it first needs to load the pipeline object if it wasn't provided as an argument. The pipeline is, by default, loaded from the `pipeline.R` file. This file needs to be parsed and evaluated to retrieve the pipeline object. Base R provides the `parse()` function that takes a path as an argument and returns the AST of the file. After that, the AST is evaluated using the `eval()` function together in an empty environment to minimize the pollution of the global environment of the R session the `make()` was called in. The result of the `eval()` call is expected to be a pipeline object. This is because Pipelinr has a constraint that the last expression in the `pipeline.R` must evaluate to a pipeline object. The user is able to change the path to the pipeline file by setting the option `pipelinr_pipeline_file`.

*Stage filters.* Once the pipeline object is obtained, `make()` must decide which stages should be evaluated. To decide which stages to evaluate, it first needs to evaluate the `from` and `only` expressions. The expressions of these arguments are captured as quosures using the `enquo()` function from the `rlang` [36] package. The captured expressions are evaluated in a data-masked environment, where the stage names are available as identifiers with their respective names. The default values of the `filter` and `only` arguments are expressions that evaluate to character vector of all stage names. The `enquo()` function is able to capture the expressions of default function arguments, which simplifies the implementation.

The `from` filter also needs to collect the child stages. Because the pipeline is checked for the existence of cycles during its construction, the algorithm for collecting the child stages does not need to worry about them. The implementation recursively traverses the stage dependency graph using the depth-first search algorithm. This recursive approach could be a problem in the case of very large dependency graphs, where the depth of the dependency graph could be over a thousand stages. However, the implementation assumes the pipelines constructed using it will have a smaller depth. If this were a problem, the child stage search could be modified to use a loop instead of recursion.

*Stage directories.* Before Pipelinr starts executing stages, it also creates the directory structure for every stage. The directory path is configurable using the `pipelinr_dir` option. The root Pipelinr directory contains directories bearing the names of their corresponding stages. These stage directories contain saved tasks and their outputs. Their filenames contain the task's hash to make them identifiable just from the filename. The `qs` [38] package is used for R object serialization.

*Stage execution.* After the stages have been filtered and their directories have been created, Pipelinr starts executing the stages. The stage filtering process preserves the order of the stages, so at this point, there is no need for topological sorting.

To execute a stage, Pipelinr first needs to evaluate the stage's inputs. The stage input expression quosures are evaluated in a data-masked environment, from which they can access the dynamic branching DSL functions. This environment also contains iterators of stage results the current stage depends on. They are accessible by identifiers in the form of their stage names. If the result of the stage input expression is not an iterator, the resulting value is wrapped in an iterator using the `make_iter()` function. Pipelinr does not use any of the R's object systems, so to verify if the value is an iterator, it needs to be verified by the object's structure.

The input iterators are combined using the `zip_iter()` function. The names from the input quosure list are preserved up to this point. Because the names of the input quosures are the arguments of the body function, the result of the `zip_iter()` call essentially returns the arguments of the body function. If the stage did not depend on any other stage, the body arguments iterator would be empty. For this reason, Pipelinr checks if the stage has any dependencies, and

if it does not have any, then the empty argument iterator is swapped for an iterator yielding a single empty list. If this would not have been done, the body function wouldn't have been invoked.

The argument iterator is then converted to a task iterator. Tasks are named lists containing two elements, arguments, and a hash. The hash is computed from the arguments using the `hash()` function from the `rlang` [36] package. The `hash()` function can compute the hash of any R object. Because of this, it can be used to compute a hash of a task. After the hash is computed, the tasks are persisted in the stage directory.

At this point, the tasks are ready to be executed by the executor. The executor is invoked with the task iterator, and the same process continues for the next stages.

## 3.4 R executor

In the implementation, an executor is represented by a function. It takes two arguments, the first is a task iterator, and the second is a stage object. The executor's main responsibility is to evaluate the tasks and save their outputs to the stage directory. They are also responsible for displaying the stage's execution progress. The progress bar rendering is facilitated by the `progress` [39] package. The executors also need to collect the metadata about task's evaluation.

The R executor is an executor that evaluates all tasks in the same R process as the `make()` was called from. Its implementation is simple. It evaluates the task by calling the `do.call()` with the stage body function and the task's arguments.

The `stdout` and `stderr` output is captured using the `capture.output()` base R function. The `capture.output()` takes in an expression as its first argument. It evaluates the expression and redirects the output of the given type to a connection. Connection in R is described as a "generalized file" [40]. Opened files, input and output streams, sockets, pipes, and more are represented in R as connections.

Pipelinr uses the `textConnection()` to capture the output to a character vector. The `textConnection()` can be used to create a connection that reads input from a character vector or writes output also to a character vector. To collect the output to a character vector, it must first be instantiated and assigned to a variable. Once this is done, we can call the `textConnection()` to create a connection to a vector. The first argument must be the variable's name to which our output vector has been assigned. The `open` argument needs to be set to `"w"` to specify that the connection is an output connection. Furthermore, if the vector has been declared in an environment other than the global environment, we also need to set the `local` argument to `TRUE`. Once this is done, the returned connection will write to the specified character vector.

The `capture.output` differentiates between two types of output. This distinction is essentially between the output written to `stdout` or `stderr`. To define which type of output it should capture, the `type` parameter should be set to either `output` in the case of an output written to `stdout` or `message` in the case of an output written to `stderr`. Code listing 21 shows how the output capture is implemented.

Each element in the output character vectors corresponds to an output line. To get the output in a single string, it needs to be merged together with line breaks as separators.

The R executor also measures the execution time of the task. This is implemented by simply taking two timestamps, one before and one after the stage body function call. Then it calculates the duration using the `lubridate` [41] package.

## 3.5 GNU Parallel

Pipelinr does not directly handle the orchestration of the worker process, which need to be used to parallelize the execution of R code. Instead, it offloads this responsibility to GNU Parallel [34]. As its name suggests, the GNU Parallel executor uses GNU Parallel to parallelize the

```
# Initialize local character vectors for both output streams.
stdout <- character()
stderr <- character()

# Create output text connections.
out_con <- textConnection("stdout", "w", local = TRUE)
err_con <- textConnection("stderr", "w", local = TRUE)

# The capture.output() function needs to be called for each output stream.
capture.output(
    capture.output(
        <expression>,
        file = out_con,
        type = "output"
    ),
    file = err_con,
    type = "message"
)

# Connections need to be manually closed.
close(out_con)
close(err_con)

# At this point, the stdout and stderr character vectors contain outputted
# lines as their elements.
```

**Code listing 21** Implementation of output capture using the `textConnection()` and `output.capture()` functions.

execution of a stage. GNU Parallel is a program for running shell commands in parallel. Since GNU Parallel is a complex tool with a lot of features, we will describe only those features that are being used by Pipelinr's implementation.

To run a command in parallel, the user should pass the command to be parallelized as an argument to the `parallel` command. After that, he needs to specify input sources with which the program will be parallelized. The syntax uses the `:::` delimiter to specify an input source as a sequence of arguments. As an example, the user can run the following `echo` command in parallel using the following command:

```
$ parallel echo Hello ::: 1 2 3 4 5
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
```

The command above defines a single input source of numbers from 1 to 5. GNU Parallel then builds five commands in the format of `echo Hello <number>` and executes them in parallel. The input source can also be taken from `stdin`, where each line corresponds to one input.

In addition to executing shell commands in parallel, it also allows the user to execute them distributively over SSH. To achieve this, the user can use the `-S` and specify the SSH host. We could modify the previous example to run on a hypothetical SSH host `example.com`:

```
$ parallel -S user@example.com echo Hello ::: 1 2 3 4 5
```

Instead of using the `-S` option, the user can also specify the hosts in the so-called SSH login file. In this file, each line specifies an SSH host and the host's number of available CPUs. To use the SSH login file, the user must specify the file's path using the `--sshloginfile` option. The following snippet demonstrates the structure of the SSH login file:

```
4/user@example.com
2/another_user@hostname.org
```

The snippet above defines two SSH hosts, the first having four available CPUs and the second having two.

Some shell commands need to read inputs from files, or their output is in the form of a file. For this reason, GNU Parallel allows the user to transfer both input and output files to the SSH hosts. Using the `--transfer` option, the user can specify which files should be transferred to the host. The following snippet demonstrates the usage of the `--transfer` option to transfer files `foo.txt` and `bar.txt` to the SSH host.

```
$ parallel -S user@example.com --transfer cat ::: foo.txt bar.txt
```

To retrieve files the remotely executed command could have produced, the user can use the `--return` option. With this option, the user can specify, using replacement strings, which output files should be transferred back based on the filenames of the input files. Suppose we would want to compress some files using the `zip` command. If we would like to run this command in parallel on a remote host using SSH, we could invoke the `parallel` command with the following arguments:

```
Seq  Host      Starttime       JobRuntime  Exitval  Command
1    worker-4  1682254963.849  0.217       0        echo Hello 1
2    worker-2  1682254963.850  0.217       0        echo Hello 2
3    worker-3  1682254963.851  0.217       0        echo Hello 3
4    worker-1  1682254963.852  0.216       0        echo Hello 4
5    worker-4  1682254964.080  0.205       0        echo Hello 5
```

**Code listing 22** An example of a simplified GNU Parallel job log file.

```
$ parallel
    -S 2/user@example.com
    --transfer
    --return {.}.zip
    zip {.}.zip ::: foo.txt bar
```

The snippet above creates `zip` archives of the input files. The files are transferred to the remote host, on which the `zip` command is invoked to create a zip archive of the file with the same filename but with the file extension changed to `.zip`. This is accomplished using the {.} replacement string, which strips the filename from the input source of its file extension. GNU Parallel implements many other replacement strings, but they will not be discussed in this thesis.

Some shell commands could depend on some local files, which are not available on the host. An example of this could be just an invocation of a shell script. In such a case, the script wouldn't be available on the remote host. For this use case, GNU Parallel provides the `--basefile` option. It is followed by a path argument to the file that should be transferred to all remote hosts before the command is executed. Suppose we have a script named `script.sh` in the current working directory, and we would like to execute it distributively on our remote hosts. The following snippet demonstrates how `parallel` could be invoked to transfer the script to the remote hosts:

```
$ parallel
    -S 2/user@example.com
    --basefile ./script.sh
    ./script.sh ::: foo bar
```

The files transferred to the SSH hosts are only deleted if the `--cleanup` option is set. It deletes all of the files on the SSH host that were transferred to or from the SSH host. This includes even files transferred using the `--basefile` option.

Because the `--transfer`, `--return`, and `--cleanup` options are frequently used together, a shorter option `--trc` can be used instead of the aforementioned options. It needs to be followed by the argument of the `--return` option.

Using the `--joblog` option, the user can tell GNU Parallel to save the outputs and metadata about command execution into a file. This could be useful when executing a large number of commands to get an overview of their execution. An example of a simplified job log file can be seen in code listing 22.

The file-transferring capability of GNU Parallel is implemented using `rsync` [42]. For this reason, it needs to be installed on the SSH host. In addition to `rsync`, PERL [43] must also be installed.

## 3.6 GNU Parallel executor

Because the R runtime is are single-threaded, parallelism can only be achieved by running multiple R processes in parallel. By using GNU Parallel, Pipelinr can use it to spawn task evaluation

worker processes.

To evaluate tasks, the executor invokes the `exec_task_and_collect_metadata.sh` shell script. This script invokes two other R scripts. The first is the `exec_task.R` script, which deserializes and executes the provided task. The outputs from the task execution are saved in a separate file, the same way they are saved when using the R executor. The other script that is executed is `collect_metadata.R`. Purpose of this script is to collect metadata, which could not be collected in the R process spawn by the `exec_task.R` script. In the current implementation, this script only appends process exit code to the task outputs.

These scripts are located in the `inst` directory in the project root directory. This directory is used to distribute files other than source files together with the package onto the user's machine. These files can then be located using the `system.file()` base R function. This function takes a filename and a package name as arguments and then searches the package for the given file. The returned values is a path to the found file.

When the execution is done over SSH, the script files and a serialized body function are transferred at the beginning of stage execution using the GNU Parallel `--basefile` option. The files transferred using GNU Parallel are always transferred to the same path as on the remote host. For example, when transferring a local file located at `/usr/lib64/R/library/foo`, GNU Parallel will try to transfer it to the same path on the remote machine. This creates a problem when the files are transferred from directories where the user has read only permissions. To remediate this, the path can be changed to a relative by placing an additional current directory dot symbol inside the path. For example, if we would want to transfer a file located at `/usr/lib64/R/library/foo` to the home directory of the user we log in through SSH, we could add a `./` before the filename: `/usr/lib64/R/library/./foo`. The transferred file will then be located at `~/foo`. The executor assumes the home directory on the SSH host has read and write permissions, so it transfers all files to the home directory using the aforementioned path modification technique.

When the execution is not done over SSH, the script files are located in the package install directory. This is in contrast to when the execution is done over SSH because, in that case, the script files are located in the home directory. Due to this reason, the `exec_task_and_collect_metadata` script takes as its first two arguments the path to the `exec_task.R` and `collect_metadata.R` scripts, otherwise, it would not be able to locate them.

The GNU Parallel process is executed using the `processx` package [44]. The `processx` package is used primarily because it allows execution of the child process in a non-blocking way, as opposed to the base R `system2()` base R function, which always waits for the child process to finish. The non-blocking execution is crucial, because otherwise, the R process would not be able to report on the stage execution progress.

During the execution of the child process, the executor polls each second whether the child process has finished or not. If it has not yet finished, it reads the job log file to retrieve the number of completed tasks. The number of completed tasks is essentially the number of lines in the job log file minus the header line.

The arguments of the child process in the case of a local execution are relatively simple. We only need to specify the `--joblog` option and the command, which invokes the `exec_task_and_collect_metadata.sh` script. An example of the arguments passed to the child process can be seen in code listing 23.

In the case of a remote execution over SSH, the arguments get more complex. To have the task execution scripts available on the SSH hosts, we need to specify the `--basefile` options of the scripts and stage body. After that, we need to specify the `--trc` option with the replacement `{.}_out.qs`, specifying the file name pattern of task output files. The scripts are executed using their relative paths as opposed to absolute paths in the local execution.

The task filenames are passed as an input source through `stdin`. If they were passed as arguments, the maximum argument count could be exceeded. Thus this approach should work with a larger number of tasks.

```
$ parallel
    --joblog joblog
    /home/user/projects/pipelinr/inst/exec_task_and_collect_metadata.sh
    /home/user/projects/pipelinr/inst/exec_task.R
    /home/user/projects/pipelinr/inst/collect_metadata.R
```

**Code listing 23** Example of the `parallel` arguments provided by the GNU Parallel executor in the case of a local execution.

```
$ parallel
    --joblog joblog
    --sshloginfile /home/user/projects/pipelinr/ssh_worker/nodefile
    --basefile /home/user/projects/pipelinr/inst/./exec_task_and_collect_metadata.sh
    --basefile /home/user/projects/pipelinr/inst/./exec_task.R
    --basefile /home/user/projects/pipelinr/inst/./collect_metadata.R
    --basefile body.qs
    --trc {.}_out.qs
    ./exec_task_and_collect_metadata.sh
    ./exec_task.R
    ./collect_metadata.R
```

**Code listing 24** Example of the `parallel` arguments provided by the GNU Parallel executor in the case of a remote execution over SSH.

## 3.7 Stage body serialization

The GNU Parallel executor needs to serialize the stage body so it can be then loaded in another R worker process. A problem arises when the body function uses globals or functions and variables from non-base packages.

To explain this problem, we first need to discuss how R functions work, and before that, we need to explain environments. Environments describe the mapping from a variable name to its value. When a variable's value is retrieved, the runtime searches for it in the environment. Environments can have a parent environment. They essentially form a linked list. If a variable has not been found in the current environment, the lookup then continues recursively through the parent environments. In the case a variable has not been found in the environment, R throws an error. Every R session has a global environment in which global variables are declared. The global environment extends loaded package environments and other namespaces. R environments are objects themselves and can be accessed and manipulated during runtime [15].

R functions have three components: arguments, body, and an environment, sometimes called closure in other languages. When a function is called, its environment gets extended with a new environment in which the arguments are assigned. This new environment then hosts the execution of the function's body [15].

The function's environment is the environment, in which the function was defined. Consider the following functions:

```
x <- 1

foo <- function() {
    x <- 2
    function() x
}

bar <- foo()
```

The function `foo()` is defined in the global environment, which becomes its environment. When `foo()` was called, a new environment extending the global environment was created. In this environment, we define a new function which will then be immediately returned. In the global environment, we assign this new function to the variable `bar`. The `bar()` function environment will be the environment created when `foo()` was called. For this reason, if we would call `bar()`, it would return 2. This is an example of lexical scoping.

A problem arises when serializing R functions that use globals. If we would serialize the function using the `qs` [38] package or even using the base R `saveRDS()` function, it would only serialize the function's environment chain only up to the last environment before the global environment. This means that it would not include globals or functions and variables from other packages. This behavior makes sense because serializing the global environment could take up a large amount of memory.

This is primarily a problem when serializing a stage body function. To solve this, the GNU Parallel executor analyzes the function for used globals and packages, which are then serialized with the function body and loaded during the task's execution.

To analyze used globals, the `codetools` [45] package is used. This package provides the `findGlobals()` function for analyzing used globals within a function body. The found globals are checked to see if they are defined in the global environment or if they are a function or a variable from a package. If the value is a global and at the same time is a function, we also need to analyze it for used globals and packages recursively.

The analysis the `findGlobals()` implements is an approximation. It suffers many drawbacks thanks to R's dynamic nature, as discussed in the used symbols analysis in the pipeline construction section. It also does not analyze the function for used packages. Consider the following snippet:

```
foo <- function() ns::bar()

# returns "::"
codetools::findGlobals(foo)
```

In the snippet above, the function `foo()` accesses the function `bar()` in the `ns` namespace. Because the `findGlobals()` function analyzes only the usages of global variables, it only returns the `::` namespace access operator as the only used global, which is the correct result from the perspective of the analysis. Because of this reason, an analysis of used package namespaces was implemented.

Pipelinr assumes that all namespace accesses made using the `::` and `:::` operators access package namespaces, not some user-defined namespaces. The used package namespace analysis simply traverses the AST to find the used packages. In the R's AST, the namespace access operators are represented by a `call` AST node, as are all operators.

Both analyses are implemented in the `find_used_globals_and_packages()` function. This function returns a pair containing a named list of used globals with their values and a character vector of names of used packages.

The GNU Parallel executor serializes the stage body functions into a list with three elements: the body function, the global environment, and a character vector of names of used packages.

The packages are loaded before the task evaluation. If a given package is not installed, an error will be thrown. The user must make sure that all dependencies are installed on the SSH hosts.

## 3.8 Test environment

To test the implementation, Pipelinr emulates a full production environment, including the SSH hosts. A Docker [46] container for the SSH hosts has been developed, to achieve this. It's implemented using a Dockerfile, which extends the Alpine Linux [47] image. The Dockerfile installs the necessary dependencies for R, GNU Parallel, and an OpenSSH server [48]. It also installs the necessary R packages for the GNU Parallel executor scripts, which are `qs` and `lubridate`. The Dockerfile takes in a single build argument `PUB_KEY`, which is supposed to be a path to a public key, which is then added to the authorized keys of the `worker` user.

Starting a single SSH worker to emulate the production environment is not enough. Ideally, we should run multiple workers in parallel. These workers need to be specified in an SSH login file. GNU Parallel also would not work out of the box with default SSH settings, so we need to generate an OpenSSH configuration file. These problems are handled by the `run_test_workers.sh`, which builds and starts up multiple container replicas using Docker Compose [49] and then generates the corresponding OpenSSH configuration files with the SSH login file.

The script begins by generating an RSA key pair used for the SSH authorization. This key pair is generated only if it does not yet exist.

To build the SSH login file, the script needs to know the local addresses of the started-up containers. Docker allows the inspection of a network where the containers are running using the `docker network inspect` command. Docker Compose runs the replicas in a bridge network, which is given a name in the format `<dir>_default`, where `<dir>` is the directory of the `docker-compose.yml` file. In our case, the name of the network is `ssh_worker_default`. By running the `docker network inspect ssh_worker_default` command, we get detailed information about the network in a JSON format. The `Containers` object field contains information about the running containers. The script parses this information using the `jq` [50] program and extracts the names of the containers together with their local addresses.

The script then continues with the creation of the OpenSSH configuration file. This file is not necessarily needed and could be substituted by command line options. The main benefit of it is that it makes debugging and troubleshooting of the SSH workers easier. Another benefit is that the developer does not need to pollute his own SSH configuration file with SSH worker identities, which also could have different addresses each time they are started. The generated file primarily specifies aliases for every worker. It also turns off the host key check, which creates issues when connecting to newly built containers because the OpenSSH client requires the user to confirm a prompt manually. We can afford to turn off the host key check since, in the case of local development, security is not an issue. The path to the known hosts file is set to `/dev/null`, because storing the known hosts' information could pose issues even when the strict host checking is turned off. The host key information could be saved, for example, in the case of debugging, during which the developer could manually connect to the SSH worker without using the generated OpenSSH configuration file.

After the OpenSSH config is generated, the SSH login file is generated. The structure of the file is constructed essentially by prepending the number of available CPUs before a hostname. In the current implementation, each worker is defined as having only a single CPU available.

As a test runner, the `testthat` [51] package is used. As is common in the R ecosystem, it has an API that is intended to be used from the R REPL. To make it easier for the developer to drop into the R REPL and run the tests, a `.Rprofile` file has been prepared. This file ensures that the `devtools` [52], which then is used to load the development version of Pipelinr. The `run_test_workers.sh` script is also run to start the SSH workers. Furthermore, the `GNU_PARALLEL_SSH` environment variable is set up to make GNU Parallel load the generated

OpenSSH configuration. Thanks to this, the developer can simply call the `test()` function to run all tests.

# Chapter 4

# Assessment

This chapter assesses if our design and implementation follow the requirements we have defined and then, we will demonstrate the viability of Pipelinr on a real pipeline.

## 4.1 Requirements

To reiterate, our requirements were that the pipeline execution should be idempotent, the runtime should provide some feedback about execution, it should be possible to access metadata about the execution, and the runtime should be able to execute the pipeline in parallel and distributively.

- *Idempotence* – Every time a pipeline is evaluated, it should reach the same final state. In the current implementation, this is dependent on the stage and task filters. If a `make()` is invoked with the same stage and task filter, it is idempotent. However, when `make()` has been invoked with different stage filters, the execution is not idempotent.

  Consider a pipeline with stages `a` and `b`, where `b` depends on `a`. If we invoke `make(b)`, the pipeline will not produce any tasks for stage `b` because stage `a` has no results. If we then invoke `make()`, the pipeline will create outputs for stage `a` and, from them, tasks and outputs for stage `b`. Because of this, the pipeline reaches a different state.

  To summarize, the idempotence requirement is fulfilled only when `make()` is invoked with the same stage filters.

- *Feedback* – When a pipeline is being executed, it renders a progress bar with the stage execution progress. It also calculates the remaining time estimate for the given stage. We can safely say this requirement has been fulfilled.

- *Metadata* – Pipelinr has a notion of metadata directly in its model. It has been designed with this requirement in mind. Metadata about the stage's execution can be accessed by the pipeline itself using stage input expressions or using the `metadata()` and `metadata_df()` functions. The metadata contains information about a given task's arguments, result, execution time and strings printed to output streams.

- *Parallelization* – Users can use the GNU Parallel executor, which can execute tasks in a single stage in parallel. It also has the ability to execute those tasks distributively over SSH.

Furthermore, we had defined a single non-functional requirement, to make the implementation as compact as possible. As of the time of writing, the current implementation consists of 1536 lines of R code, including comments and blank lines. It has ten dependencies, of which seven are part of the `tidyverse` [16] package collection.

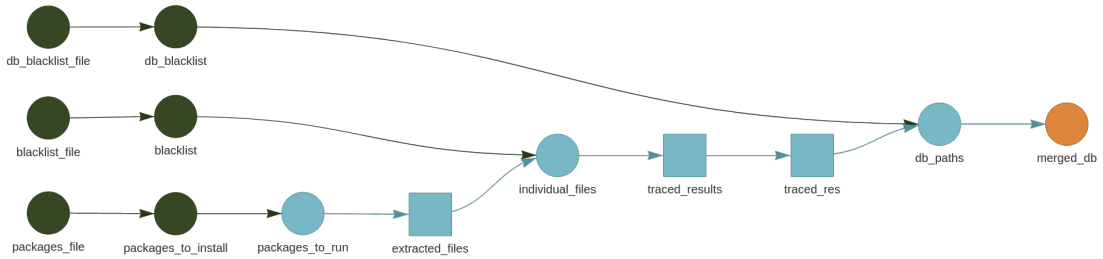


**Figure 4.1** Visualization of the Signatr tracing `targets` pipeline visualized by the `tar_visnetwork()` function.

## 4.2 Pipeline rewrite

To assess Pipelinr's viability on a practical example, we chose a pipeline implemented using the `targets` package and rewrote it. The pipeline in question is a pipeline from the `signatr` paper's [53] artifact [54]. This tool uses dynamic tracing and fuzzing to infer function signatures from code that has been extracted from various R packages. It contains two pipelines, one for dynamic tracing and one for fuzzing.

In this section, we will first discuss how to rewrite simple targets, that do not use dynamic branching. Then we will discuss how dynamic branching targets can be rewritten. After that, we will investigate which further improvements can be made to the pipeline.

The tracing pipeline was implemented using `targets`. It retrieves a predefined list of packages and installs them. Then it extracts file paths from those packages, which are then passed to the tracer. The tracing results are then post-processed and merged into a database. The visualized `targets` pipeline can be seen in figure 4.1.

Rewriting simple targets, that do not use dynamic branching to Pipelinr is fairly simple. The targets are simply converted to stages, and the target's expression is wrapped in a function. Consider the target `packages_to_run`:

```
list(
    ...,
    tar_target(
        packages_to_run,
        install_cran_packages(packages_to_install, lib_path, NULL),
        deployment = "main",
        cue = tar_cue(mode = "always")
    ),
    ...
)
```

Such a target can be rewritten to a stage like this:

```
make_pipeline(
    ...,
    packages_to_run = stage(function(packages_to_install) {
        install_cran_packages(packages_to_install, lib_path, NULL)
    }),
    ...
)
```

With this approach, it is possible to rewrite all stages that do not use dynamic branching nor directly depend on a target, which uses dynamic branching. This is because `targets` and Pipelinr behave differently when dynamic branching is used. In Pipelinr's terms, we could say that every target is automatically collected into a single input. The most straightforward strategy for rewriting these targets is, therefore, to use `collect()` or `collect_df()` to collect the results of a stage, which corresponds to a target, that uses dynamic branching.

The such case arises in the `extracted_files` and `individual_files` stages, where the `individual_files` stage needs to collect results from the previous stage using the `collect()` function. The collected results form a list of character vectors, but the `individual_files` body function expects a single character vector. For this reason, the collected results need to flattened into a single character vector using the `unlist()` function, as can be seen in the snippet below:

```
make_pipeline(
    ...
    extracted_files = stage(
        inputs = stage_inputs(
            package = mapped(packages_to_run)
        ),
        body = function(package) {
            extract_code_from_package(package, lib_path, extracted_output)
        }
    ),

    individual_files = stage(
        inputs = stage_inputs(
            extracted_files = collect(extracted_files) %>% unlist()
        ),
        body = function(extracted_files) {
            remove_blacklisted(extracted_files, blacklist)
        }
    ),
    ...
)
```

Another such case arises in the `traced_results` and `trace_res` stages. In this case, the results of `traced_results` stage only need to be collected using the `collect_df()` function.

The stages mentioned above, which use dynamic branching, can benefit from parallelization using the GNU Parallel executor. The paths it reads from global variables were normalized to make the `extracted_files` stage work. This is needed because when the GNU Parallel executor is used, the code is executed inside the stage directory to not pollute the current working directory. This breaks relative paths, so they need to be normalized. After this change, the stage executed correctly without any errors.

The `traced_results` stage would benefit the most parallel execution. However, all tasks

in the stage ended with a segmentation fault when the GNU Parallel executor was used. This probably due to a problem when invoking the modified R dyntrace runtime.

If we rewrite the pipeline with the described approach, we will discover that some stages are unnecessary. These stages come from targets with the file format. The primary reason for their existence in the `targets` pipeline was to make `targets` aware of those files. Thanks to this, `targets` can determine which targets to rebuild. Pipelinr does not use the same approach. It evaluates stage input expressions that produce tasks and then evaluates tasks that pass the task filter. There is no notion of automatic task rerun, even when stage implementation changes, to not lose task outputs unintentionally, which can be the case when debugging. For this reason, these stages can be removed.

Upon closer inspection, we would further discover that the file format targets were followed by targets, which essentially read unique lines from those files. The stages that stem from those targets can be replaced by global variables, which contain the unique lines of their given file. This is the case for stages that come from `packages_to_install`, `blacklist`, and `db_blacklist` targets.

The original `targets` implementation consisted of 398 lines of code, of which 136 were in the `_targets.R` file and 262 were in the `R/functions.R` file. After the Pipelinr rewrite, the `pipeline.R` file, counterpart to the `_targets.R`, had 64 lines. Some code was moved to the `R/functions.R`, increasing the number of lines to 276. Overall the rewritten pipeline consists of 340 lines of code, which is 58 lines of code less than the original pipeline.

There are some parts of pipeline, which could be further refactored to make in more in line with Pipelinr's idioms. The responsibility `individual_files` and `db_paths` stages is to remove blacklisted paths from their inputs. They could be rewritten to invocations of the `filtered()` function in stage inputs, which could filter out the blacklisted values without the need for a whole stage. This would require refactoring of the `remove_blacklisted()` function, which handles removal of blacklisted paths from a character vector.

The code for the implemented package is available in a forked GitHub repository of the `signatr` artifact on the `pipelinr` git branch, which is available on the following URL: https://github.com/MichaelVrana/sle22-signatr-artifact.

```
make_pipeline(
    packages_to_run = stage(function() {
        install_cran_packages(packages_to_install, lib_path, NULL)
    }),

    extracted_files = stage(
        inputs = stage_inputs(
            package = mapped(packages_to_run)
        ),
        body = function(package) {
            extract_code_from_package(package, lib_path, extracted_output)
        },
        executor = gnu_parallel_executor
    ),

    individual_files = stage(
        inputs = stage_inputs(
            extracted_files = collect(extracted_files) %>% unlist()
        ),
        body = function(extracted_files) {
            remove_blacklisted(extracted_files, blacklist)
        }
    ),

    traced_results = stage(
        inputs = stage_inputs(
            file = mapped(individual_files)
        ),
        body = function(file) {
            trace_file(file, lib_path, output_path)
        }
    ),

    traced_res = stage(function(traced_results) {
        fix_traced_res(traced_results)
    }),

    db_paths = stage(
        inputs = stage_inputs(
            traced_res = collect_df(traced_res)
        ),
        body = function(traced_res) {
            remove_blacklisted(
                traced_res$db_path,
                db_blacklist,
                only_real_paths = TRUE
            )
        }
    ),

    merged_db = stage(function(db_paths) merge_db(db_paths, sxpdb_output))
)
```

**Code listing 25** Pipelinr pipeline definition of the rewritten `signatr` tracing pipeline.

# Conclusion

The goal of this thesis was to develop a DSL together with a runtime for defining and executing data analysis pipelines. Such DSL was implemented in the form of the Pipelinr R package.

The DSL allows the user to define a pipeline of multiple interdependent stages, forming a dependency graph. The actual computation is expressed as an ordinary R function. Pipelinr enables the user to use dynamic branching patterns for defining complex dependencies between stages. The implementation of dynamic branching is built on top of a small iterator library, which makes the dynamic branching patterns composable and flexible, allowing the user to preprocess stage inputs using ordinary R expressions.

Dynamic branching allows the runtime to parallelize the execution of a given stage. Pipelinr implements two executors for executing the pipeline's stages. One of those is a GNU Parallel executor, which can execute a given stage in parallel or even distributively over SSH. To enable this, Pipelinr uses static analysis to search for the stage's dependencies in the form of global variables and packages and recreates the same environment in different R processes.

Pipelinr was designed to provide metadata about the pipeline's execution. Metadata are a core part of Pipelinr's model and are paired with each task that in a given stage. Based on the metadata, the user can troubleshoot the pipeline and even specify which parts of the pipeline should be reevaluated.

To assess Pipelinr's viability, a pipeline implemented using the `targets` R package has been rewritten using Pipelinr.

The Pipelinr package is available for installation from a public GitLab repository: https://gitlab.fit.cvut.cz/vranami8/r-dsl.

## Future work

Ideas on improving Pipelinr would be based primarily on the usage of its user base. However, it has not been used by any user other than the author, so we list some ideas we think it can be improved.

A new executor could use HPC grid engines using the `clustermq` package for distributed execution.

The static analysis of globals and used packages, which is used for stage body function serialization could be extended by the analysis of used S3 and S4 methods since those are currently not being analyzed.

Pipelinr currently stores all tasks and task outputs on the file system. The storage layer could be extended to provide more options. For example, a SQL or NoSQL database could be used.

To provide better feedback about the stages execution, a GUI application or extensions to IDEs or other text editors could be built.

Pipelinr could also apply for submission to the CRAN repository.

# Bibliography


1. TAYLOR, Petroc. *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025*. 2022. Available also from: https://www.statista.com/statistics/871513/worldwide-data-created/.
2. DAVEY, Reginald. *What is the Replication Crisis?* 2022. Available also from: https://www.news-medical.net/life-sciences/What-is-the-Replication-Crisis.aspx.
3. VITEK, Jan; KALIBERA, Tomas. R3: Repeatability, reproducibility and rigor. *ACM SIGPLAN Notices*. 2012, vol. 47, pp. 30–36. Available from DOI: 10.1145/2442776.2442781.
4. ALSTON, Jesse M.; RICK, Jessica A. A Beginner's Guide to Conducting Reproducible Research. *The Bulletin of the Ecological Society of America*. 2021, vol. 102, no. 2, e01801. Available from DOI: https://doi.org/10.1002/bes2.1801.
5. DI TOMMASO, Paolo. *Awesome Pipeline*. GitHub, 2023. Available also from: https://github.com/pditommaso/awesome-pipeline.
6. FREE SOFTWARE FOUNDATION, INC. *GNU Make*. 2023. Available also from: https://www.gnu.org/software/make/.
7. CENTER FOR HIGH THROUGHPUT COMPUTING, COMPUTER SCIENCES DEPARTMENT, UNIVERSITY OF WISCONSIN-MADISON. *DAGMan Workflows*. 2022. Available also from: https://htcondor.readthedocs.io/en/latest/users-manual/dagman-workflows.html.
8. UNIVERSITY OF WISCONSIN-MADISON. *HTCondor*. 2023. Available also from: https://htcondor.org/index.html.
9. THE R FOUNDATION. *What is R?* 2023. Available also from: https://www.r-project.org/about.html.
10. LANDAU, William Michael. The drake R package: a pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software*. 2018, vol. 3, no. 21. Available also from: https://doi.org/10.21105/joss.00550.
11. LANDAU, William Michael. The targets R package: a dynamic Make-like function-oriented pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software*. 2021, vol. 6, no. 57, p. 2959. Available also from: https://doi.org/10.21105/joss.02959.
12. BECKER, Richard A. A Brief History of S. [N.d.]. Available also from: https://web.archive.org/web/20150723044213/http://www2.research.att.com/areas/stat/doc/94.11.ps.
13. IHAKA, Ross. R : Past and Future History. 1998. Available also from: https://www.stat.auckland.ac.nz/~ihaka/downloads/Interface98.pdf.

14. MOUNT, John. *What does it mean to write "vectorized" code in R?* 2019. Available also from: https://win-vector.com/2019/01/03/what-does-it-mean-to-write-vectorized-code-in-r/.
15. WICKHAM, Hadley. *Advanced R*. 2019. ISBN 978-0815384571. Available also from: https://adv-r.hadley.nz/.
16. HADLEY WICKHAM, et al. Welcome to the tidyverse. *Journal of Open Source Software*. 2019, vol. 4, no. 43, p. 1686. Available from DOI: 10.21105/joss.01686.
17. WICKHAM, Hadley; FRANÇOIS, Romain; HENRY, Lionel; MÜLLER, Kirill; VAUGHAN, Davis. *dplyr: A Grammar of Data Manipulation*. 2023. Available also from: https://dplyr.tidyverse.org.
18. WICKHAM, Hadley. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN 978-3-319-24277-4. Available also from: https://ggplot2.tidyverse.org.
19. BACHE, Stefan Milton; WICKHAM, Hadley. *magrittr: A Forward-Pipe Operator for R*. 2022. Available also from: https://magrittr.tidyverse.org.
20. MORONEY, Matthew. *Exploring Data in R with dplyr and ggplot*. 2017. Available also from: https://rstudio-pubs-static.s3.amazonaws.com/259438_4890109724d345c699899c2f506e5646.html.
21. EDDELBUETTEL, Dirk; FRANÇOIS, Romain. Rcpp: Seamless R and C++ Integration. *Journal of Statistical Software*. 2011, vol. 40, no. 8, pp. 1–18. Available from DOI: 10.18637/jss.v040.i08.
22. STOUDT, Sara; VÁSQUEZ, Váleri N.; MARTINEZ, Ciera C. Principles for data analysis workflows. *PLoS Computational Biology*. 2021, no. 17. Available also from: https://doi.org/10.1371/journal.pcbi.1008770.
23. BROMAN, Karl. *Minimal Make*. 2023. Available also from: https://kbroman.org/minimal_make/.
24. FITZJOHN, Rich. *remake*. 2023. Available also from: https://www.rdocumentation.org/packages/remake/versions/0.3.0.
25. ELI LILLY AND COMPANY. *The drake R Package User Manual*. 2023. Available also from: https://books.ropensci.org/drake/index.html.
26. SCHUBERT, Michael. clustermq enables efficient parallelisation of genomic analyses. *Bioinformatics*. 2019. Available from DOI: 10.1093/bioinformatics/btz284.
27. ELI LILLY AND COMPANY. *The targets R package user manual*. 2023. Available also from: https://books.ropensci.org/targets/.
28. LANDAU, William Michael. *tarchetypes: Archetypes for Targets*. 2021. Available also from: https://docs.ropensci.org/tarchetypes/.
29. ONUOHA, Martins. *Countries Now*. 2023. Available also from: https://countriesnow.space/.
30. WICKHAM, Hadley. *httr: Tools for Working with URLs and HTTP*. 2023. Available also from: https://httr.r-lib.org/.
31. OOMS, Jeroen. The jsonlite Package: A Practical and Consistent Mapping Between JSON Data and R Objects. *arXiv:1403.2805 [stat.CO]*. 2014. Available also from: https://arxiv.org/abs/1403.2805.
32. WICKHAM, Hadley; HENRY, Lionel. *purrr: Functional Programming Tools*. 2023. Available also from: https://purrr.tidyverse.org/.

33. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. Behavioral Patterns. In: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995, p. 257. ISBN 0-201-63361-2.
34. TANGE, Ole. GNU Parallel - The Command-Line Power Tool. *The USENIX Magazine*. 2011, vol. 36, pp. 42–47. Available also from: `https://www.researchgate.net/publication/291448776_GNU_Parallel_-_The_Command-Line_Power_Tool`.
35. WICKHAM, Hadley; BRYAN, Jennifer. *R Packages*. O'Reilly Media, Inc., 2023. Available also from: `https://r-pkgs.org`.
36. HENRY, Lionel; WICKHAM, Hadley. *rlang: Functions for Base Types and Core R and 'Tidyverse' Features*. 2022. Available also from: `https://rlang.r-lib.org`.
37. MOZILLA CORPORATION. *Iterators and generators*. 2023. Available also from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators#iterators`.
38. CHING, Travers. *Using qs*. 2023. Available also from: `https://github.com/traversc/qs`.
39. CSÁRDI, Gábor; FITZJOHN, Rich. *Progress*. 2023. Available also from: `https://github.com/r-lib/progress`.
40. *connections: Functions to Manipulate Connections (Files, URLs, ...)* 2023. Available also from: `https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/connections`.
41. GROLEMUND, Garrett; WICKHAM, Hadley. Dates and Times Made Easy with lubridate. *Journal of Statistical Software*. 2011, vol. 40, no. 3, pp. 1–25. Available also from: `https://www.jstatsoft.org/v40/i03/`.
42. TRIDGELL, Andrew; MACKERRAS, Paul; DAVISON, Wayne. *rsync*. 2023. Available also from: `https://rsync.samba.org/`.
43. WALL, Larry. The PERL Programming Language. 2011. Available also from: `https://www.researchgate.net/publication/265628667_The_PERL_Programming_Language`.
44. CSÁRDI, Gábor; CHANG, Winston. *processx: Execute and Control System Processes*. 2023. Available also from: `https://processx.r-lib.org`.
45. TIERNEY, Luke. *codetools: Code Analysis Tools for R*. 2023. Available also from: `https://cran.r-project.org/web/packages/codetools/index.html`.
46. DOCKER INC. *Docker*. 2023. Available also from: `https://www.docker.com/`.
47. ALPINE LINUX DEVELOPMENT TEAM. *Alpine Linux*. 2023. Available also from: `https://www.alpinelinux.org/`.
48. OPENBSD FOUNDATION. *OpenSSH*. 2023. Available also from: `https://www.openssh.com/`.
49. DOCKER INC. *Docker Compose*. 2023. Available also from: `https://docs.docker.com/compose/`.
50. DOLAN, Stephen. *jq*. 2023. Available also from: `https://stedolan.github.io/jq/`.
51. WICKHAM, Hadley. testthat: Get Started with Testing. *The R Journal*. 2011, vol. 3, pp. 5–10. Available also from: `https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf`.
52. WICKHAM, Hadley; HESTER, Jim; CHANG, Winston; BRYAN, Jennifer. *devtools: Tools to Make Developing R Packages Easier*. 2022. Available also from: `https://devtools.r-lib.org/`.

53. TURCOTTE, Alexi; DONAT-BOUILLUD, Pierre; KŘIKAVA, Filip; VITEK, Jan. Signatr: A Data-Driven Fuzzing Tool for R. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. Auckland, New Zealand: Association for Computing Machinery, 2022, pp. 216–221. SLE 2022. ISBN 9781450399197. Available from DOI: 10.1145/3567512.3567530.

54. DONAT-BOUILLUD, Pierre; KŘIKAVA, Filip. *Signatr Artifact*. 2022. Available also from: https://github.com/PRL-PRG/sle22-signatr-artifact.