BUILDING WEB MASHUPS OF DATA WRANGLING OPERATIONS FOR TRAFFIC DATA

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2016

By
Hapsoro Adi Permana

School of Computer Science

Contents

Abstract **9**

Declaration **10**

Copyright **11**

Acknowledgements **12**

1 Introduction **14**

1.1 Motivation 14
1.2 Research Questions 15
1.3 Aim 15
1.4 Project Scope 16
1.5 Objectives 16
1.6 Dissertation Structure 16

2 Background **18**

2.1 Big Data 18
2.2 Data Wrangling 19
2.3 Web Mashups 21
2.4 Trifacta Wrangler and Data Wrangler 22
2.5 R 23
2.5.1 Data Wrangling Packages for R 24
2.5.2 Exposing R Functions as A Web Service 25
2.6 OpenRefine 26
2.6.1 Inspecting OpenRefine Web Application Programming Interface (API) 26
2.6.2 OpenRefine Flow of Work 27

2.7 Python 29
2.7.1 Pandas and Numpy 29
2.7.2 Python as A Web Service 30
2.8 Taverna Workbench 30
2.8.1 Taverna: A Brief History 30
2.8.2 Using Taverna Workbench 31
2.8.3 Taverna Promotes Data Wrangling Characteristics 32
2.8.4 Taverna Components 33
2.9 Summary 34

3 Conceptual Design **35**
3.1 Executable Use Case 35
3.1.1 Data Wrangling Task 1: DWT_1 36
3.1.2 Data Wrangling Task 2: DWT_2 38
3.1.3 Data Wrangling Task 3: DWT_3 40
3.1.4 Data Wrangling Task 4: DWT_4 40
3.1.5 Assumptions 41
3.2 Wrangling Operations Summary 41
3.3 Architecture 44
3.4 Summary 46

4 Implementation **47**
4.1 Agile Development Methodology 47
4.2 Implementation Plan 48
4.3 Environment 49
4.4 Sprint One 49
4.4.1 Sub-Sprint: Calling R Functions using OpenCPU 50
4.4.2 Sub-Sprint: Encapsulating Chart Functions in R 52
4.4.3 Sub-Sprint: Taverna Interaction with OpenRefine Functions . 53
4.4.4 Sub-Sprint: Traffic Data Wrangling Web Services in Python . 57
4.4.5 Sub-Sprint: Implementation of DWT_1 58
4.5 Sprint Two 60
4.5.1 Sub-Sprint: Implementation of Reusable Interactions 62
4.5.2 Sub-Sprint: Implementation of DWT_2, DWT_3, and DWT_4 . . . 63
4.6 Sprint Three 65
4.6.1 Migrating Interactions into Taverna Components 65

4.6.2 Sub-Sprint: Improving Data Wrangling Tasks Using Taverna Components . . . 66
4.7 Implementation Challenges . . . 67
4.8 Summary . . . 68

5 Testing and Evaluation **69**
5.1 Iterative Testing Approach . . . 69
5.1.1 Unit Testing . . . 70
5.1.2 Integration Testing . . . 72
5.2 Evaluation . . . 75
5.2.1 Client Side Computational Evaluation . . . 76
5.2.2 Network Load . . . 78
5.3 Summary . . . 80

6 Conclusions and Future Works **82**
6.1 Conclusions . . . 82
6.2 Future Works . . . 83

Bibliography **85**

A Data Wrangling Task Formalisations **91**

B Interactions Between Taverna and Data Wrangling Services **95**

C Data Wrangling Task Workflows **98**

D Data Wrangling Task Results **115**

E Improved Data Wrangling Task Workflows **119**

Word Count: 20133

List of Tables

3.1 Wrangling operation requirements for the four wrangling tasks 43

4.1 Environment for system development 50

5.1 Unit Testing Summary 71
5.1 Unit Testing Summary 72
5.2 Integration Testing Summary 73
5.2 Integration Testing Summary 74
5.2 Integration Testing Summary 75
5.3 Network traffic sent and received by the client-side (in Mega Bytes) . 79
5.4 Network traffic sent and received by each server (in Mega Bytes) . . . 80

List of Figures

2.1 Trifacta Wrangler 23
2.2 A screenshot of RStudio GUI for Operating R Scripts 24
2.3 Google Chrome's Developer Tools was used in inspecting HTTP API of OpenRefine 27
2.4 Open Refine flow of work 28
2.5 Taverna Workbench's designer view which consists of: (A) design panel, (B) service panel, and (C) explorer. 32

3.1 Flowchart as the formal representation of DWT_1 39
3.2 Software architecture of the mashup 45

4.1 Workflow to Read Data from a URL to OpenCPU 51
4.2 OpenRefine processes for importing a JSON file implemented as a Taverna interaction 55
4.3 OpenRefine Rename Column List Handling 56
4.4 Activity diagram for the implementation of DWT_1 which represents the interaction between Taverna Workbench and R, OpenRefine, and Python services 61
4.5 typical interaction of Taverna and the R server for column wrangling operations 63
4.6 Taverna components organized into families 67

5.1 Taverna Workbench Performance 77

A.1 Wrangling Task Formalisation for task DWT_2 92
A.2 Wrangling Task Formalisation for task DWT_3 93
A.3 Wrangling Task Formalisation for task DWT_4 94

B.1 Interaction between Taverna and wrangling services for task DWT_2 . . 96

B.2 Interaction between Taverna and wrangling services for task DWT_3 . . 96
B.3 Interaction between Taverna and wrangling services for task DWT_4 . . 97

C.1 Taverna workflow implementation for data wrangling task DWT_1 (1/7) 99
C.2 Taverna workflow implementation for data wrangling task DWT_1 (2/7) 100
C.3 Taverna workflow implementation for data wrangling task DWT_1 (3/7) 101
C.4 Taverna workflow implementation for data wrangling task DWT_1 (4/7) 102
C.5 Taverna workflow implementation for data wrangling task DWT_1 (5/7) 103
C.6 Taverna workflow implementation for data wrangling task DWT_1 (6/7) 103
C.7 Taverna workflow implementation for data wrangling task DWT_1 (7/7) 104
C.8 Taverna workflow implementation for data wrangling task DWT_2 (1/2) 105
C.9 Taverna workflow implementation for data wrangling task DWT_2 (2/2) 106
C.10 Taverna workflow implementation for data wrangling task DWT_3 (1/3) 107
C.11 Taverna workflow implementation for data wrangling task DWT_3 (2/3) 108
C.12 Taverna workflow implementation for data wrangling task DWT_3 (3/3) 109
C.13 Taverna workflow implementation for data wrangling task DWT_4 (1/5) 110
C.14 Taverna workflow implementation for data wrangling task DWT_4 (2/5) 111
C.15 Taverna workflow implementation for data wrangling task DWT_4 (3/5) 112
C.16 Taverna workflow implementation for data wrangling task DWT_4 (4/5) 113
C.17 Taverna workflow implementation for data wrangling task DWT_4 (5/5) 114

D.1 The output of Data Wrangling Task DWT_1 116
D.2 A sample of the output of Data Wrangling Task DWT_2 116
D.3 The output of Data Wrangling Task DWT_3 117
D.4 The output of Data Wrangling Task DWT_4 118

E.1 Taverna workflow implementation for data wrangling task DWT_1 using customized components (1/2) 120
E.2 Taverna workflow implementation for data wrangling task DWT_1 using customized components (2/2) 120
E.3 Taverna workflow implementation for data wrangling task DWT_2 using customized components 121
E.4 Taverna workflow implementation for data wrangling task DWT_3 using customized components 121
E.5 Taverna workflow implementation for data wrangling task DWT_4 using customized components 122

Acronyms

API Application Programming Interface.

CRAN Comprehensive R Archive Network.

CSV Comma-Separated Values.

EUC Executable Use Case.

GREL Google Refine Expression Language.

GUI Graphical User Interface.

HTTP HyperText Transfer Protocol.

JSON JavaScript Object Notation.

PNG Portable Network Graphics.

RDBMS Relational Database Management System.

REST Representational State Transfer.

SAAM Scenario-Based Analysis of Software Architecture.

SOAP Simple Object Access Protocol.

TCP Transmission Control Protocol.

TfGM Transport for Greater Manchester.

UML Unified Modelling Language.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

WSDL Web Service Definition Language.

XML Extensible Markup Language.

Abstract

BUILDING WEB MASHUPS OF DATA WRANGLING OPERATIONS FOR TRAFFIC DATA

Hapsoro Adi Permana

A dissertation submitted to the University of Manchester for the degree of Master of Science, 2016

Data wrangling is essential to prepare data for traffic analysis. Traffic observations, as well as other sensed data, might contain records which are distant from the majority of the distribution. There is also a possibility that missing values are present. To prevent misleading analysis, imputation is crucial. Moreover, the study of traffic involves not only road traffic observation but also other variables which affect traffic, which means data would come from multiple sources and, hence, the format of one dataset varies from another. Unfortunately, there doesn't exist one tool which comprises all functionalities required to wrangle traffic data: preparing traffic data for analysis requires utilisation of more than one wrangling tool.

This research project aimed to explore the possibility of combining data wrangling operations from a selection of wrangling tools. In this research, R and OpenRefine were involved, as well as a set of self-implemented, domain-specific wrangling operations. The latter was implemented in Python. Wrangling operations from each tool were made accessible as Representational State Transfer (REST) web APIs. OpenCPU framework was used to expose R wrangling operations as a web API whilst Bottlepy framework was used for Python. OpenRefine already had their functions readily accessible via HyperText Transfer Protocol (HTTP). Taverna Workbench was utilised as the user interface, from which a data wrangling workflow was synthesised.

The outcome of this research was tested to assure that it has behaved expectedly and furthermore evaluated to assess the design strategy. The results showed that our approach produced an insignificant amount of network load at the client-side. Conversely, huge network load was observed to occur on the server-side. More importantly, using the web mashup concept, data wrangling operations from various tools were successfully integrated.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to take this opportunity to thank the government of Indonesia, especially LPDP (Indonesia Endowment Fund for Education) for granting the scholarship to pursue Master's degree in Manchester, United Kingdom. It was a dream came into realisation.

I would also like to express my gratitude to my supervisor, Dr. Sandra Sampaio, who provided support, guidance, and mentorship throughout the completion of this dissertation.

Thank you to my friends from the University of Manchester School of Computer Science and Indonesia who have been there in good and bad throughout the year. Thank you to my big family and my loved one. Thank you to the Almighty God, to whom I wished wisdom and strength.

Finally, I would like to say the biggest thank you to my parents for the unconditional love and support before, during, and after this study. without them I would not be here.

For Mama, Papa, my late Eyangkung, and Eyangti

Chapter 1

Introduction

This chapter introduces the motivation that drives the proposal of this project and what research questions are tried to be answered through this dissertation. Furthermore, the aim and objectives of this dissertation project are explained. Report structure is presented at the end of this chapter.

1.1 Motivation

The study of traffic is an important domain that could be analysed to understand human behaviour on a massive scale [1]. Such study involves transportation data from various sources and current technology and infrastructure has enabled traffic data to be captured from a range of methods [2] [3]. Guo et al [4] presented in their paper that traffic data would come from the following sources: individual transceiver onboard of each vehicle, a sensor that is situated in a fixed location, and from social media. Data from different sources means they come in different formats and, thus, it is necessary that they are transformed into a uniform, structured format before processing [5]. Moreover, there are also domain specific challenges. Traffic data recorded by TfGM, for example, does not contain complete temporal information.

The study of traffic involves not only transportation data, but also other data which contains variables affecting traffic. Yau [6], for example, conducted a research on factors of traffic accident severity by putting into account safety and environment variables. Moreover, Zhang et al [7] included human and time factors as well as weather condition. It exposes more challenges to traffic analysis.

Moreover, there are challenges related specifically to the domain of traffic analysis. Data integration is yet another challenge in preparing data for traffic analysis. Due to

its spatial and temporal characteristics [8], data integration for traffic analysis requires special techniques.

Additionally, the variety of sources has also been a challenge for preparing data for traffic analysis. Federal Highway Administration of U.S. Department of Transportation stated the challenges of traffic analysis tools which include the aforementioned, followed by challenges in preparing the human resources to be able to use the tools and functionalities offered by data wrangling tools[9]. Typical data manipulation tool such as Microsoft Excel would not suffice the scale. Moreover, tools which are not mainly purposed to perform data wrangling would not support wrangling requirements. Data wrangling in Excel, for example, is not easily reproducible. Consequently, when data wrangling task is passed from one operator to the other, the effort of transferring the knowledge is immense. Other tools are available to help analysts in preparing data, such as OpenRefine and Data Wrangler. However, there is no single tool that comprises all the operations needed to completely wrangle these data. One tool has its advantage over the other. It is a conception that traffic data analyst should master multiple tools to prepare traffic analysis data.

1.2 Research Questions

With such issues mentioned in Section 1.1, the following questions, which are used as the research questions, arise.

- What are the data wrangling operations necessary to produce traffic analysis-ready traffic data?
- Which wrangling operations for corresponding cases are covered by existing data wrangling tools?
- Could there be a solution that combines functionalities provided by these tools?

1.3 Aim

The aim of this research project is to develop and evaluate a web mashup of data wrangling operations from a selection of tools, which would enable the use of functionalities from various tools within one interface. Furthermore, the tool would be able to run on a machine with low specification and produces low network traffic. From this point forward, the aimed software artefact is referred to as the mashup.

1.4 Project Scope

The focus of this research project is the implementation of a web mashup of data wrangling operations from a selection of existing tools. Additionally, there are functionalities which are specific in the domain of traffic and does not exist in any tools. This project would attempt to implement these requirements as part of the mashup. It is not the concern of the project, however, to measure the efficiency of and optimise the algorithm of such functions.

1.5 Objectives

To achieve the aim of the project, several objectives were defined as project milestones. The objectives were as follows.

1. Review literatures which were related to data wrangling, mashups concept, and web services. These would provide a background towards designing the mashup.
2. Review existing data wrangling tools. A comprehensive literature review and experimental study was conducted to create a comparison of required functionalities provided by existing tools.
3. Construct a set of traffic analysis use cases to extract data wrangling requirements for the mashup and propose a software architecture design for the mashups implementation.
4. Implement the mashup using the architectural design as a guidance and data wrangling operations from the use cases as requirements.
5. Test and evaluate the produced artefact to assure that the software solution implemented in this research produces expected results.

1.6 Dissertation Structure

The rest of this dissertation is organised in the following order.

Chapter 2 covers literature reviews and background information related directly to the project. This includes a brief introduction of challenges in processing big data, the concept of data wrangling, and mashups. Furthermore, several existing data wrangling tools are reviewed.

Conceptual design and solution architecture are explained in Chapter 3. In this chapter, user is presented with the use cases of traffic analysis which are used as the requirements for the mashup. Each use case is transformed into a flow chart explaining the required wrangling operations. Furthermore, the design architecture is thoroughly explained.

Chapter 4 covers the details of the implementation phase. This chapter includes the iterative and incremental implementation methodology.

Chapter 5 covers testing and evaluation of the approach taken in tackling the research questions.

Finally, conclusions and recommendations for future works are discussed in Chapter 6.

Chapter 2

Background

This chapter provides a background for this research project. A brief introduction to big data is presented to familiarise the reader with challenges in working with traffic data for analysis. Furthermore, readers are introduced with data wrangling tools and the concept of mashups and web services. It is followed by a description of several existing wrangling tools. From this description, the author selected a set of tools from which the mashups could be developed with. Finally, Taverna Workbench as a tool for designing a workflow of web services is described.

2.1 Big Data

Development of information technology has enabled capture of unprecedented amount of data. As of today, the world is full of computer-enabled gadgets. The invention of the internet and the development of wireless and mobile connectivity have created massive networks of interconnected devices. Data are created from human-computer interactions and machine-to-machine communications. Internet social media are generating double-digit terabytes of data from their users [10], millions of hours are spent on telephone calls per day, and sensors are collecting enormous amount of data. Almost everything that human does generates data.

In the era of information, data is treated as a vital asset for organisations to collect and analyse. Data is a capital that needs to be explored to discover its true potential. For a company to create a product based on data, it has to be in high quality. Due to its characteristics, big data is often seen to be of low quality [5]. According to Mohanty [11], big data has changed the analytics approach from top-down to bottom-up: questions are asked after data is collected. As a consequence from this shift of

perspective, a lot of data has been generated. However, the full potential of big data is still hidden in its three characteristics: *volume, velocity, and variety*.

As more data is generated, the challenge of volume arises. In 2010, data stored are estimated to have reached 13 exabytes [5]. John Gantz and David Reinsel [12] predicted that the number of data captured by a variety of methods would have exceeded 40 Zeta Bytes by 2020. When a huge chunk of data was transferred to the centralized processing unit, it should cope the massive load. Two options are available: store all data when torrential data arrives, or filter the stream and store relevant data. The earlier option would require an organisation to provide large data store. The latter would require sophisticated architecture because big data engines, while effective in storage operations, are not efficient in processing data [13].

Structured data represents only around 20% of data in the world [14]: the bigger portion is inhabited by a mixture of semi-structured and unstructured data. Moreover, various sources of data is another conundrum that arises in the context of big data. A variety of data sources has lead to a challenge to process data from multiple formats. Chatterjee and Segev introduced this challenge in their paper in 1991 [15]. Consequently, big data processing unit must be endowed with appropriate technology that enables flexibility in handling a diverse range of data formats. Furthermore, such technology would stipulate resilience from evolving data formats [16], as data format from outside sources is beyond the control of an organisation.

Traffic analysis, as was described in section 1.1, required data from various domain and sources. It, furthermore, could be inferred that the analysis involved big data.

2.2 Data Wrangling

To prepare data for traffic analysis, data wrangling should be performed. Data wrangling is defined as an iterative and or repetitive data manipulation operations performed to unstructured data which is aimed to produce usable, credible, and useful data for analysis [17][18][1]. Some data wrangling activities are explained as follows.

Transforming data changes the structure or value of data. Data transformation is performed as early as data acquisition where original data of different formats arrive from different sources. Heterogeneity in data formats needs to be uniformed before it could further be processed by machine [5]. Computers only processes data when it is structured, i.e. in tabular format. Therefore, unstructured and semi-structured

[1] The activity was previously known as data munging [19].

data, for example JavaScript Object Notation (JSON) and Extensible Markup Language (XML) files, needed to be transformed into a tabular format before it could be further processed. Common data transformation methods which change data shape are rotate and pivot. Data aggregation is regarded as one type of structure transformation. Data aggregation reduces the number of observations. Some functions that are applicable in data aggregation are sum, average, minimum, and maximum. Data which is aggregated is typically grouped beforehand. The grouping enabled aggregate functions to calculate values by group. Some example of value transformation are as follows: high numerical interval attributes are commonly scaled down to the lower range and to reduce computational requirements; skewed numeric values distribution are often normalised; value of an attribute could be derived into new features or into values of different type; and date components could be derived into each component of day, month, year, etc. Kandel et al [17] illustrated data type transformation by replacing postal code into geographical coordinates. Concatenation of several attributes also falls into this category.

Data integration is a process of combining multiple data. There are two categories of data integration: merge and join. Data merge concatenates two or more datasets by row, whilst data join finds matching observations from two datasets.

Data cleaning is a process which identifies and removes tuple or value anomalies. Not rarely missing values cause poor, misleading analysis [20]. It could be resolved by various imputation techniques. If necessary, i.e. missing value percentage is significantly high, records could be removed. The treatments vary from zero values imputation, interpolated using trend, or simply ignored. Furthermore, Megler et al [21] used supervised machine learning, clustering methods, and outlier detection to identify record anomalies.

Prior to tools specific for data wrangling tasks, wranglers had to manually record every operation they have performed. Microsoft Excel, for example, does not have a feature to record wrangling operations that have been performed. This is problematic especially when wrangling procedures have to be replicated for tasks in the future, which turns data wrangling into a clerical task. Moreover, should a wrangling job be handed from its original author, it is possible that the new personnel would not understand the motives of each task. Therefore, Kandel et al [17] proposed that data wrangling tasks include information of each operation.

2.3 Web Mashups

The emergent of cloud computing was the effect of Software Oriented Architecture and Software as a Service business model, which ultimately creates opportunities for smaller enterprises to run IT-supported businesses [22]. Trivago[2] and Skyscanner[3] are two examples of businesses consuming various web services from different service providers to create a product of their own. This method is called web mashups. Zhang et al [8] offered a formal definition of a web mashup, which is a software that combines APIs into a single integrated user interface.

A product constructed by applying mashups have advantages over application built on a single-owned data source. For a product built on top of outsourced web services, maintenance of each web service is not the responsibility of the product developer. Rather, a single web service is supported by their respective service providers. Moreover, no extra effort should be focused on the operations of these services. The second advantage is that multiple services combined together create a unique product. Trivago and Skyscanner offer price comparisons. The data they are displaying come from numerous APIs. Without the mashups concept, these web applications would need to collect, manage, and maintain their own data, which would then incur cost.

Although there exists web mashup technologies which allow users to create a mashup of their own without having to learn a programming language [23], it is common that its development involves programming. The latter approach is preferred due to the flexibility offered to build the aimed artefact.

Web service is the key element of a web mashup. A web service is a software application which function is made available through web protocols. It enables interoperability of softwares from multiple platforms. Web services are more often called web Application Programming Interfaces (web APIs) by developers. The two popular types of web services are Simple Object Access Protocol (SOAP) and REST. SOAP web services need to have its description available before it could be consumed. SOAP required its functionalities to be defined using Web Service Definition Language (WSDL). A WSDL helps the rediscovery of a SOAP web service. Conversely, a REST API does not require the service definition specified beforehand [24]. The latter selection is particularly helpful in agile software development, because its requirements

[2]trivago.co.uk. Trivago is a search engine for hotels. The web application, however, does not have a feature to reserve a room. Rather, it forwards its users to certain hotel booking sites.

[3]skyscanner.net. SkyScanner is a web application that enables users to compare flight price, duration, and transits. Similar to Trivago, SkyScanner does not offer a reservation system.

reduces development effort. However, consumers of such web API would not understand required input parameters, what data is returned, and what the API does as there is no WSDL describing the services. The more information available about these APIs, the easier it is to develop a mashup [25].

A web API is a prerequisite for a web mashup [24]. Having said that, it is impossible to build a mashup from components which do not offer APIs. Related to this project, there are data wrangling tools which functionalities are not available to access.

2.4 Trifacta Wrangler and Data Wrangler

Trifacta Wrangler is a tool specifically designed for data scientists to prepare their data. This product was originally developed by The Stanford Visualisation Group under the project name Data Wrangler[4] before commercially launched as Trifacta Wrangler. The commercial product is available in free edition with feature limitations. The edition evaluated in this project is the free edition. Interactive data wrangling visualisation approach is offered by this tool. As a web application, Data Wrangler does not require installation in the client side.

Trifacta Wrangler is more advanced compared to its predecessor in terms of features. It enables working with multiple datasets and data aggregation, where Data Wrangler lacks. Data wrangling operations available in Trifacta Wrangler are categorised into: transform, join, aggregate, and union. The functions are accessible from bottom left hand side of the Graphical User Interface (GUI), as shown by Figure 2.1. The set of transformation operations are used to change the structure and values of data. Transformation scripts are inputted using Trifacta Wrangler's designated wrangling language[5]. Join enabled the conjuncture of multiple dataset by determining one or more key columns. Aggregation procedure could be achieved by grouping one or multiple columns.

Unfortunately, it is impossible to interoperate with neither Trifacta Wrangler nor Data Wrangler via web service. Trifacta Wrangler API is not open for developers on its free edition. Additionally, the architecture of Data Wrangler does not allow its API to be accessed. Although its application is accessible through web browser, the logic and wrangling functions of Data Wrangler reside in the client side[6]. Thus, it is not

[4]The beta product is still available online by the time this dissertation is being written.

[5]Full reference to its language is found at `https://www.trifacta.com/support/language/`.

[6]Data Wrangler is implemented using Javascript. Due to its architecture, it does not support wrangling large datasets.

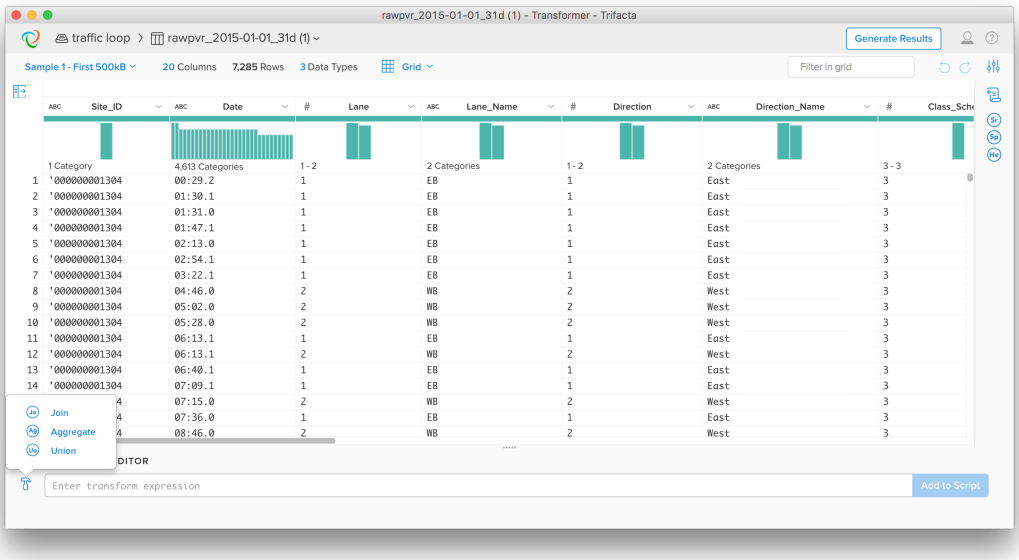


Figure 2.1: Trifacta Wrangler

possible to access its functions through a web API. Therefore, it is concluded that both tools will not be present in the mashup[7].

2.5 R

R is a popular scripting language among statistician due to a variety of statistical functions available to use [26]. R is licensed under GNU General Public License. It is an open source scripting language with a selection of extensions developed by a multitude of creators available online in Comprehensive R Archive Network (CRAN) repository [27]. The packages can be downloaded directly from R console[8].

RStudio is used widely by developers community to work with R scripts. The tool provides users with GUI, as shown in Figure 2.2, and is available in different platforms. It enables visualisation of data that is currently in use. RStudio also enable R packages to be compiled from within.

There are several basic data-types recognised by R. Numbers with decimal places are handled using numeric. Similar to other programming languages, whole numbers are processed as integers. Logical data-type is analogous to boolean values Java. This

[7]There are functions present exclusively in these tools: promote row as header, fill row, shift column, and transpose.

[8]By invoking `install.packages()` command.

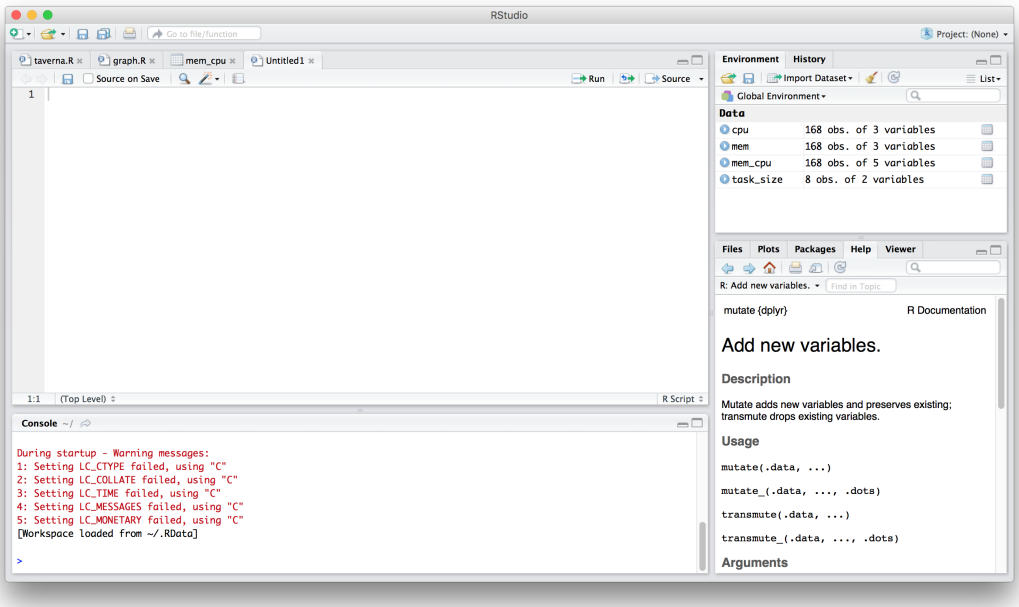


Figure 2.2: A screenshot of RStudio GUI for Operating R Scripts

data-type is used when evaluating conditionals. While in programming language characters and strings are separated in their individual structures, in R both are treated indifferently as a character object. Complex represents mathematical expressions containing imaginary value i.

There are three types of data structures in R: one-dimensional, two-dimensional, and multidimensional. Vectors and list handle one-dimensional array. Two-dimensional data is handled with matrices and data frames. Vectors and matrices handle single data type, for example: a vector of characters, a matrix of integers. Lists and data frames are used when multiple data-types are present. Data frames are particularly useful in handling tabular data; it is analogous to the table structure in Relational Database Management System (RDBMS), where a table consists of different data-types. For data which dimension is greater than two, arrays are used.

2.5.1 Data Wrangling Packages for R

A wide choice of open-source plugin packages are available in R. These packages could be used to help data scientists in solving data problems. Typically, packages which are used for data wrangling are `dplyr` and `tidyr`. Wrangling functions available in R are categorized into data reshape, subset, grouping, aggregation, making new

variables, and data combination [28]. Functions under reshape category are operations that change the layout of a data. Subset operations takes a subset of a dataset. Subsetting could be performed on variables or observations. Data grouping could be performed in computing new variables or in aggregating. Data combination joins multiple datasets. The packages could be used in conjunction with basic R operations.

2.5.2 Exposing R Functions as A Web Service

To achieve the aim of this research, which is described in section 1.3, R functions need to be accessible by other platform over the internet protocol. Therefore, OpenCPU server framework is used. OpenCPU server is a framework that transforms R packages into RESTful web services. To call an R function using OpenCPU, a web URL is pointed to the package name and the required function name[9]. The response of an OpenCPU web API contains several lines of information. In normal circumstances, it returns the following lines of Uniform Resource Identifier (URI) [29].

1. `/ocpu/tmp/<sessionkey>/R/.val`. This URI directs towards sample values of the result dataset. The result data of a wrangling operation is referred to by the data session key.

2. `/ocpu/tmp/<sessionkey>/stdout` shows the output of the R console screen. On successful API call, this will show identical output as the previous URI.

3. `/ocpu/tmp/<sessionkey>/source` shows the function called along with its parameters. Hence, `stdin`.

4. `/ocpu/tmp/<sessionkey>/console`. It shows a combination of `stdin` and `stdout`.

5. `/ocpu/tmp/<sessionkey>/info` shows the OpenCPU server information including packages loaded, R version, and the operating system of the server.

6. `/ocpu/tmp/<sessionkey>/files/DESCRIPTION` includes the information related to the session: version, author, generation date, and full description of the session.

[9]The URL pattern for calling an R function is as follows: `http://<hostname:port>/ocpu/library/<packagename>/R/<function>`. The functions are called using HTTP `POST` method, while result datasets are retrieved using `GET` method.

The result dataset produced by a wrangling operation in R accessed via OpenCPU could be used by an external web service by pointing towards the data Uniform Resource Locator (URL). The data URL is shown by the first response line. R data frames could be exported into several formats using OpenCPU, including JavaScript Object Notation (JSON) and Comma-Separated Values (CSV) [30]. Charts could be exported into widely-accepted Portable Network Graphics (PNG), bitmap, scalable-vector, or portable document format.

Furthermore, a data wrangling task may consist of a number of consecutive operations to be applied to the data. Using OpenCPU, it could be performed by using the output data session key of a wrangling operation.

2.6 OpenRefine

OpenRefine is an open source tool owned by Google, which is developed in Java and its GUI is accessible via a web browser. The tool focuses on column wrangling operations, which are scattered in every column of a dataset. The potential of this tool is hidden under its ability to translate complex wrangling expressed in Google Refine Expression Language (GREL), Jython, or Clojure. These scripting languages are similar to JavaScript, Python, and Lisp respectively, with the first being the signature language of OpenRefine [31].

OpenRefine is a tool which operates in a client-server architecture. The web client calls functions in the server via HTTP API. Due to this architecture, OpenRefine is used in the mashup. The following sections cover the investigation of OpenRefine web API and its respective procedures.

2.6.1 Inspecting OpenRefine Web API

Wrangling operations from OpenRefine were accessible via internet protocol. The APIs were examined using Google Chrome's[10] Network Inspector as shown by Figure 2.3. Headers, response, and preview tabs were important in examining the operations of OpenRefine whilst other tabs were ignored.

The headers tab was used to investigate URL and parameters sent as a request to complete the wrangling operation. There were several sections examined in this tab: general, request headers, query string, and form data. Moreover, request headers were

[10] a web browser developed by Google.

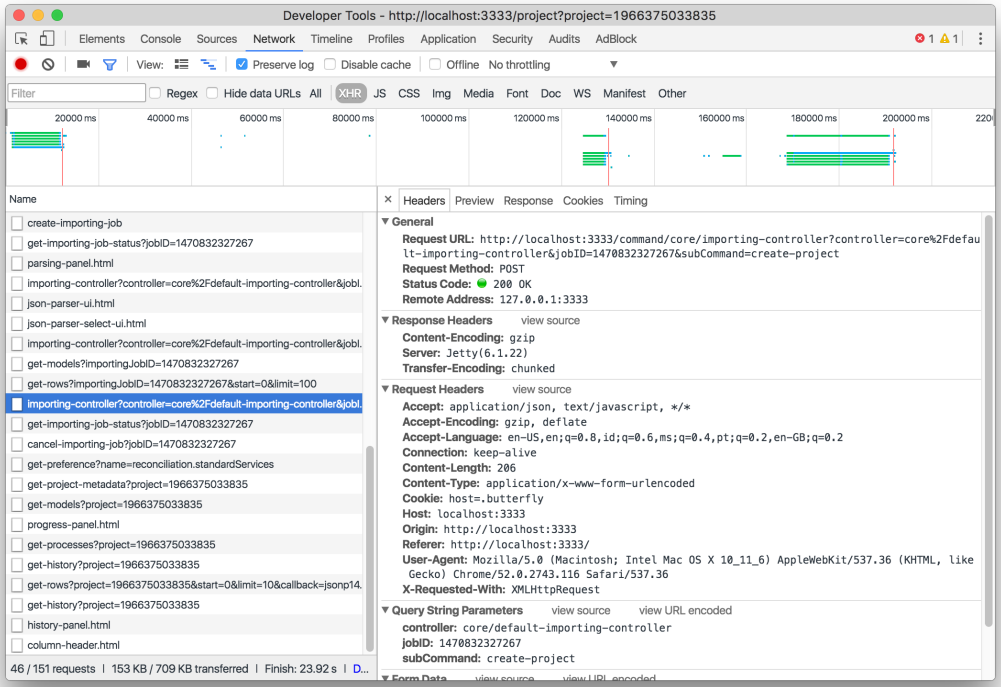


Figure 2.3: Google Chrome's Developer Tools was used in inspecting HTTP API of OpenRefine

important to investigate the encoding format of the request parameters. Parameter contents were inspected in the form data section.

Response tab displayed the content that the server sent to the client. The tab showed responses plainly, unlike preview tab which formatted responses for readability. OpenRefine sent responses in JSON format.

2.6.2 OpenRefine Flow of Work

To understand the workflow of OpenRefine, the author experimented wrangling a JSON formatted, tree-structured file containing weather observations provided by the Met Office using OpenRefine. The aim of the experiment was to transform the semi-structured data into a tabular format.

The first step to wrangling in OpenRefine was to create a new project. To simulate data retrieval from a remote server, data is fetched from Met Office server. Furthermore, a JSON node is selected. Default import settings were used and project was

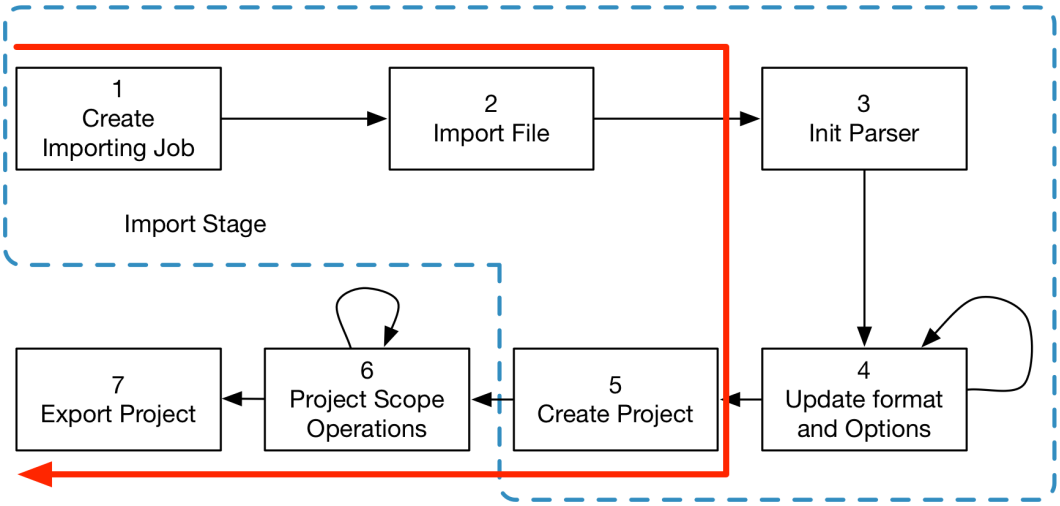


Figure 2.4: Open Refine flow of work

created. Some columns of the imported data needed to be filled down. Consequently, some a new column was created. Afterwards, project was exported into a CSV file.

From the experiment, the OpenRefine flow was inferred, as illustrated by Figure 2.4. In a broad perspective, OpenRefine web API worked in seven stages; stages 1 through 5 were called during file import phase. Moreover, stages 3 and 4 were specific to the web GUI. Therefore, in building the mashup, stages 3 and 4 were bypassed. The required stages are pointed by the red arrow. All methods were called using HTTP `POST` method.

The processes for importing a file from a remote location began by creating a data import job, which produced a job ID that was used throughout the whole import processes, i.e. stages 1 through 5. Secondly, data was imported by calling `load-raw-data` sub-command. This command received parameters encoded in multipart form data. Otherwise, the server would send an error message. Because the import command worked asynchronously, file import job status had to be monitored. Import status of `ready` reflected that data has been downloaded to the OpenRefine server and is qualified for further processing. Other import statuses indicated the data was still being processed by the server or an error had occurred.

OpenRefine included the option for users to select from which JSON node the data was imported. Information which resided outside the given JSON path are truncated. The JSON path format accepted by OpenRefine was similar but not identical to the JSON path notation implemented by Goessner [32]. After defining the record path to

be extracted, we progressed by creating an OpenRefine project. Project creation was invoked by passing the import job ID, imported data format, project name, and other format options, including the record path. Similar to the file import procedure, project creation worked asynchronously. If the project was successfully created, the import job status changed to `created-project`. Moreover, a project ID was also generated.

The project ID was important for further wrangling operations: it indicated to which a wrangling operation was performed. The operations in the experiment involved removing, renaming, filling down, and creating a new column. These operations were done by OpenRefine synchronously: the server responded directly after an operation was executed.

The result data in OpenRefine was available as per request. OpenRefine stored the project in its own format. Therefore, it was necessary to export the project into a text-based file format, for example: a CSV file. Project export was performed by sending an export request to the server. Furthermore, OpenRefine forced the request sender to download the exported project.

2.7 Python

Python, one of the most popular scripting language to date, was introduced in 1989 by Guido van Rossum in the Netherlands. It had not been launched until a year later[11]. However, the launch was only internally to Centrum Wiskunde & Informatica[12] community. Python was released to external communities in 1991. It has undergone several major releases. Python community developers are most familiar with version 2 although version 3.0 has been released since 2008. Similar to other scripting languages, Python is interpreted using JustInTime compiler.

Its existence has been supported by a large community of programmers who actively develop a variety of libraries [33]. Compared to Java, it is more efficient in resources handling when faced with a large number of connections and data [34].

2.7.1 Pandas and Numpy

Python is supported by Pandas library to handle data wrangling operations, which is comparable to dplyr package in R. It is also powered with Numpy library to handle

[11] `http://python-history.blogspot.co.uk/2009/01/brief-timeline-of-python.html`
[12] National Research Institute for Mathematics and Computer Science in the Netherlands

numerical operations on larger datasets, where performance is seen as an important factor. Numpy's performance is comparable to Matlab [35].

2.7.2 Python as A Web Service

Python code is introduced as web service using WSGI[13] framework libraries. There is a selection of Python libraries available to expose functions as web services. Each web server library has its own advantage over the other. The three famous libraries are: Django[14], Flask[15], and Bottlepy[16]. The latter two frameworks are popular for being lightweight and suitable for agile development.

2.8 Taverna Workbench

Taverna is a suite of applications for designing and running workflows [36] which could be downloaded freely under the GNU Lesser General Public License from its project incubator website[17]. The applications were built using Java and are available for Windows, Linux, and Mac. The application suite consists of Apache Taverna Commandline Tool, Workbench, Taverna Server, and their plugins.

2.8.1 Taverna: A Brief History

Taverna was a collaboration project between the University of Manchester, University of Newcastle, and EMBL European Bioinformatics Institute [36]. The idea of the tool is to solve data integration challenges in the domain chemistry. Researchers in the domain had to call multiple web services from various third-party providers. Using the tool, researcher could design the workflow visually. At the end, the tool could be used by a wider audience.

In 2014, Taverna was transferred into Apache's incubating project. The project has since been under Apache Incubator. During its time under Apache's incubation, Taverna has released an update for its command-line tool while other projects are still on development.

[13] Web Server Gateway Interface: web server and web service interface specification for Python.
[14] `https://www.djangoproject.com`
[15] `http://flask.pocoo.org`
[16] `http://bottlepy.org/docs/dev/index.html`
[17] `https://taverna.incubator.apache.org`

2.8.2 Using Taverna Workbench

Taverna Workbench is a desktop tool from the application suite. Because it was developed in Java, the installation of the tool is platform independent. At startup, the application by default will show a designer view. The workflow designer is the main view of Taverna Workbench. The view consists of three panels as shown in Figure 2.5: design, services, and explorer panels. A currently open workflow is shown in the design panel.

A workflow consists of input and output ports, and a set of services. Services and ports are interconnected by data links. A data link is represented by an arrow. The arrowhead points to the next component or port, while the dull end indicates the source of data. A dashed rectangle with a red triangle which points up on the right side indicates the input ports. Similarly, with a green triangle pointing down indicates a workflow's output ports.

Taverna Workbench has a set of built-in services that are organised in the services panel. It exposes local machine services to encode byte array, merge strings, read files, and parse XML. User interaction components are also provided to let users interact with the workflow during a run. Moreover, users could run a custom code using its Beanshell component. The custom code accepted by the component is Beanshell script, which is a lightweight scripting language based on Java. The input and output of this component could be defined flexibly.

Taverna Workbench enables web services discovery and reuse. A web service could be added by pressing import new services button and provide a URL which points to an online WSDL. The web service definition is then saved locally and could be called when required. The tool allows the user to test whether a web service is currently available. This feature does not currently exist for REST services[18].

Before Taverna Workbench runs a workflow, the application would validate the workflow to check should there be an error in parts of a workflow. Taverna would notify the user when there is an error or a warning that prevents the workflow run.

After the workflow is validated, user is prompted with workflow input values. User could manually input the parameters or load values which have been previously saved on their local machine. Moreover, a menu to save the current values is available.

When run workflow button is pressed, user is redirected to the results view. This view contains two panels. A panel situated on the left-hand side of the window shows a list of current and previously run workflows. Users could delete unwanted previous

[18] as in version 2.5.0

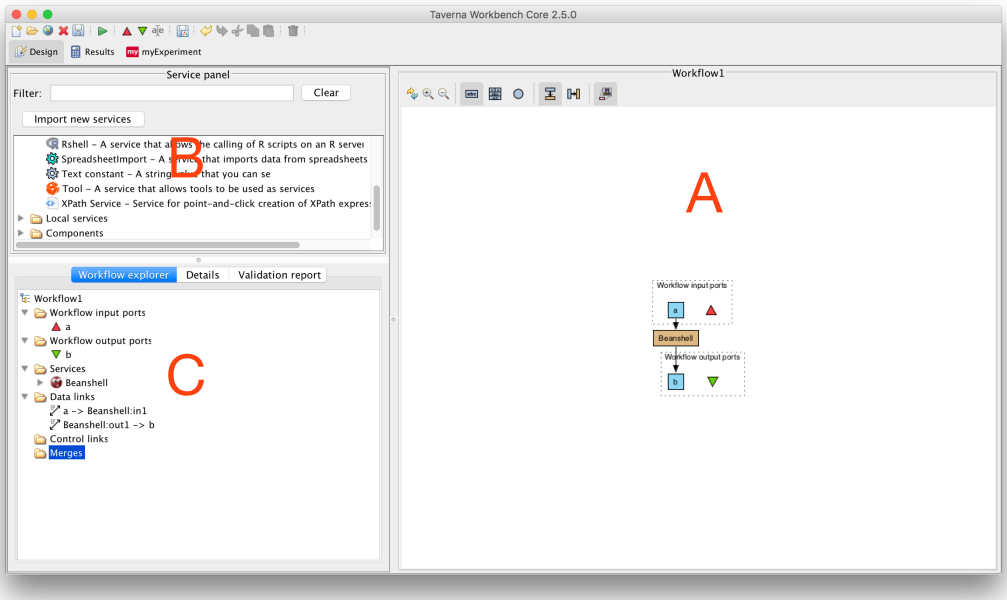


Figure 2.5: Taverna Workbench's designer view which consists of: (A) design panel, (B) service panel, and (C) explorer.

runs. One panel on the right-hand side of the window illustrates the workflow graph and progress report. This panel will show the workflow which is selected on the workflow runs panel. User is given the option to switch between showing the workflow graph or the progress report statistics. When the selected workflow is currently running, the graph will indicate currently called services with thicker borderline. Services that have been successfully called are shaded dark grey while errors are shaded red. Service and workflow input and output values are shown in the panel on the bottom. Value panel changes as user selects a specific service box from the workflow graph panel. Two options to interrupt a workflow run is presented in this view. The user could either pause or stop a workflow. The user could resume a workflow run from where it is paused. If cancel is selected, the whole workflow run is terminated.

2.8.3 Taverna Promotes Data Wrangling Characteristics

Taverna Workbench promotes reusability, auditability and collaboration. By using Taverna Workbench to design, save, and share data wrangling workflows, the three data wrangling requirements proposed by Kandel et al [17] could be fulfilled.

Auditability of data wrangling is realised by annotating a workflow. Workflow

annotation is performed by selecting from workflow explorer context menu that is revealed when right clicking on the root workflow on workflow explorer. Services and data sinks could also be annotated to insert a description of what each of them performs. Adding annotation on the workflow level will help other data wrangler to understand the bigger picture of the data wrangling task, while service-level descriptions will provide a more granular understanding of data wrangling steps.

Reusability is empowered by Taverna's feature in saving a workflow. A previously saved workflow could be reused by other wrangler to re-perform a wrangling task. Moreover, data wrangling workflows could be reused as part of another data wrangling tasks by including them as nested workflows. Hence, offline data wrangling **collaboration**. In addition to saving files locally, data wranglers are able to upload and share their data wrangling workflows on the cloud via `myExperiment.org`. Access to the workflow sharing service is available directly from Taverna Workbench through its third view, namely myExperiment. Users are required to register and log in before access is granted.

2.8.4 Taverna Components

A Taverna component encapsulates a Taverna workflow. By doing so, the complexity of a workflow is hidden from the end-user. A Taverna component is part of a component family; components which are grouped into the same component family share the same component profile. A set components which are grouped into separate families could share the same Taverna component profile.

A component profile is an Extensible Markup Language (XML) document which defines a set of rules that a component should follow. The rules definition consists of data sinks, implementation constraints, annotation, ontology, and workflow annotations. Currently there is a component profile editor prototype from the pre-Apache Taverna team[19].

Taverna Components are shared via the cloud through a designated remote component registry, i.e. `myExperiments.org`. Components which are stored in the cloud are synchronised automatically by Taverna Workbench. By creating components and sharing it through the cloud, or by manual duplication, it has empowered the reusability and collaboration characteristics of data wrangling [17].

[19]Bugs are expected as this is not the release version. Even after a component profile has been successfully created, errors might be present. Therefore, thorough manual checking should be performed.

2.9 Summary

This chapter has presented fundamental background required for the research, including literature review from the perspective of the relation of traffic analysis to big data, data wrangling, web services, and mashups. OpenCPU has enabled R functions to be exposed as a web API; an experiment has been conducted to investigate the HTTP API of OpenRefine; and Python codes could be made into web APIs using a selection of available web server libraries. It has further been discussed that due to API unavailability, Trifacta Wrangler and Data Wrangler were not used in the mashup.

Chapter 3

Conceptual Design

To answer the research challenges described in Chapter 1, a solution was designed by referring to the technologies introduced in Chapter 2. The chapter begins with the introduction of use cases that would be implemented using the mashups concept. It is followed by the summarisation of the wrangling operation requirements of the use cases. The requirements are furthermore mapped to the existing wrangling tools. Finally, an architecture for the solution is proposed.

3.1 Executable Use Case

A use case is essential in requirements gathering as it defines the specification of the software required by the client. In the Unified Modelling Language (UML) approach, the requirements of a software solution are illustrated as a use-case diagram. A UML use-case diagram contains a list of users who can interact with the system, namely actors. The actors are connected to the use-cases those were planned to be able to interact to. However, the author believed the UML approach was too vague. The purpose of UML use-case diagram was to simplify the communication between the client and the developer. On the other hand, using the UML model, use cases are not thoroughly described [37][38]. Accordingly, the author chose to adapt the Executable Use Case (EUC) [39]. It was not a novel approach in requirements engineering: it is an improvement of UML based use cases. An EUC bridges customer understanding and formal software engineering definition to be used in the implementation phase. An EUC contains the following layers.

1. **Prose layer** contains descriptive, human-language explanation of the processes

involved in completing a use case. The prose layer was described by the client and understood by the developer.

2. **Formal layer** is the formal software engineering diagram which helps developers to develop the software solution of the client requirements.

3. **Animation layer**, which is the translation of formal layer into graphics that is understandable by the users.

The prose layer was essential for the use case definition. It was the stage where requirements were thoroughly described in human words. This layer was missing in the UML use-case approach. Furthermore, an EUC also has the advantages of textual use-case modelling proposed by Hoffmann et al [38]. The prose layer was then translated into a more technical context for the formal layer. In this research, flow charts were used to define the formal layer. The client of this research was the supervisor and thus, the animation layer was eliminated.

Four data wrangling tasks were designed as the use cases to be implemented using the mashup. The aim of each task is discussed as well as the hypothesis it attempted to prove. Furthermore, the datasets involved in completing the task are discussed. The details of the tasks are discussed in the following sections. Finally, it is important to note that the focus of this project was the mashup rather than the insights of the data wrangling tasks results.

3.1.1 Data Wrangling Task 1: DWT_1

DWT_1 was a task that integrates heterogeneous data sources. The aim of the task was to prove the following hypothesis. *When weather was bad, i.e. not dry, in one day of the week the hourly average speed of vehicles observed in one site tends to be slower than the average speed in dry days.*

After the aim of the wrangling task was described, it was furthermore formalised into a technical perspective. The required DWT_1 wrangling operations are shown by Figure 3.1 and are explained as follows.

The task integrated three different datasets. The following are the datasets required for this task. Dataset $DS1_1$ contained traffic observations recorded from an inductive loop. Dataset $DS1_2$ contained the traffic observation site references. The two datasets were provided by Transport for Greater Manchester (TfGM). The third dataset was $DS1_3$ which contained weather observation data, provided by the Met Office.

$DS1_1$ contained information of vehicles passing through an observation point for a period of time. It included the following attributes: Site ID, observation time, lane, direction, vehicle class, length, headway and gap between two subsequent vehicles, speed, weight, vehicle ID, and additional information. The site ID was the observation site identification. The observation time, namely Date, consisted of only minutes, seconds, and milliseconds. Headway and gap indicated the temporal distances between two subsequent vehicles. The difference between headway and gap was that headway was calculated from the front bumper of a vehicle $n-1$ until the front bumper of a vehicle n, whilst gap was calculated from the rear bumper of a vehicle $n-1$. Direction indicated where the traffic directed. Lane was self explanatory; it was the part of the roadway in which a vehicle passed. The unit of measurements for vehicle speed was miles per hour (mph). Other attributes were irrelevant for all wrangling tasks.

$DS1_2$ was a site reference data which contained geographical information of the traffic observation sites, i.e. latitude, longitude, and address. It also included compass direction of the sensors measured in degrees from the north, which was not relevant to the task.

$DS1_1$ and $DS1_2$ were CSV files and therefore were in tabular format. This was not the case with $DS1_3$. The weather information data was in the form of tree-structured JSON file. It contained information about attribute units, observation location, date and time, and weather observation related details, such as: temperature, wind direction, weather condition[1]. The observation location were encoded as latitude and longitude. The observation date and time were separated into two attributes: the date was formatted in ISO 8601[40] standard and time was represented as minutes calculated after midnight, i.e. 00:00. The weather condition was encoded in numbers: each number represents a weather condition.

The aim of DWT_1 was to investigate the average hourly speed of the traffic in a day of the week in various weather condition. To perform the task, the following attributes must be present: day of week, hour and weather condition. The day of week and hour was expected to be extracted from the traffic observation date and time attribute. However, the date attribute in $DS1_1$ did not contain a complete information of observation date and time. To enable the extraction of both, date and time enrichment was essential.

The geographical information of the observation site was not available in $DS1_1$.

[1]Full description of $DS1_3$ was accessible at `http://www.metoffice.gov.uk/media/pdf/3/0/DataPoint_API_reference.pdf`

The traffic observation dataset had to be integrated with $DS1_2$ to retrieve the latitude and longitude of the traffic observation site. The integration would be performed by using the observation site identification as the join key. However, the site identification attribute in $DS1_1$ contained an apostrophe character at the beginning of the attribute value whilst it was not the case in $DS1_2$. Thus, before integration was performed, the character was removed. The integrated data was named $DS1_{1,2}$.

$DS1_3$ was formatted as a JSON file, which was semi-structured. Semi-structured data needed to be transformed into a structured data format, i.e. tabular form, before it could be processed [5]. Afterwards, the time attribute, which was represented as minutes after midnight, needed to be reformatted. The time was then concatenated with the date to form a complete date and time attribute. There were 32 possible values that represents the actual weather condition. These values were generalised in to several broader weather conditions.

The traffic and weather observation datasets were integrated by their geographical location and time. The integrated data was named $DS1_{1,2,3}$. It is important to note that the geographical location of longitude and latitude between the traffic observation site $DS1_1$ and the weather observation site were not identical. This was true also for the date and time. Date and time in $DS_{1,2}$ was then rounded to hours.

Because the traffic observation contained data from all days of the week, data filtration was performed to extract data from a select day of the week only. Furthermore, $DS1_{1,2,3}$ was summarised to calculate the average speed per hour and per weather condition before finally it was visualised into a bar chart.

3.1.2 Data Wrangling Task 2: DWT_2

DWT_2 is a task which is aimed to impute missing values using median value for headway and gap. The task proves the following hypothesis. *Hypothesis 2: Based on observation site, vehicle direction, lane, hour, and day of week, the values for headway and gap could be inferred.*

Data wrangling task DWT_2 required the use of traffic observation data from inductive loops provided by TfGM. This is identical to the dataset used in DWT_1 and therefore date and time enrichment was performed before further wrangling operations were executed.

Based on the hypothesis description, median values of headway and gap were calculated based on the spatial and temporal characteristics of the traffic data. Direction, lane, and observation site represents the spatial characteristic. Hour of day and day of

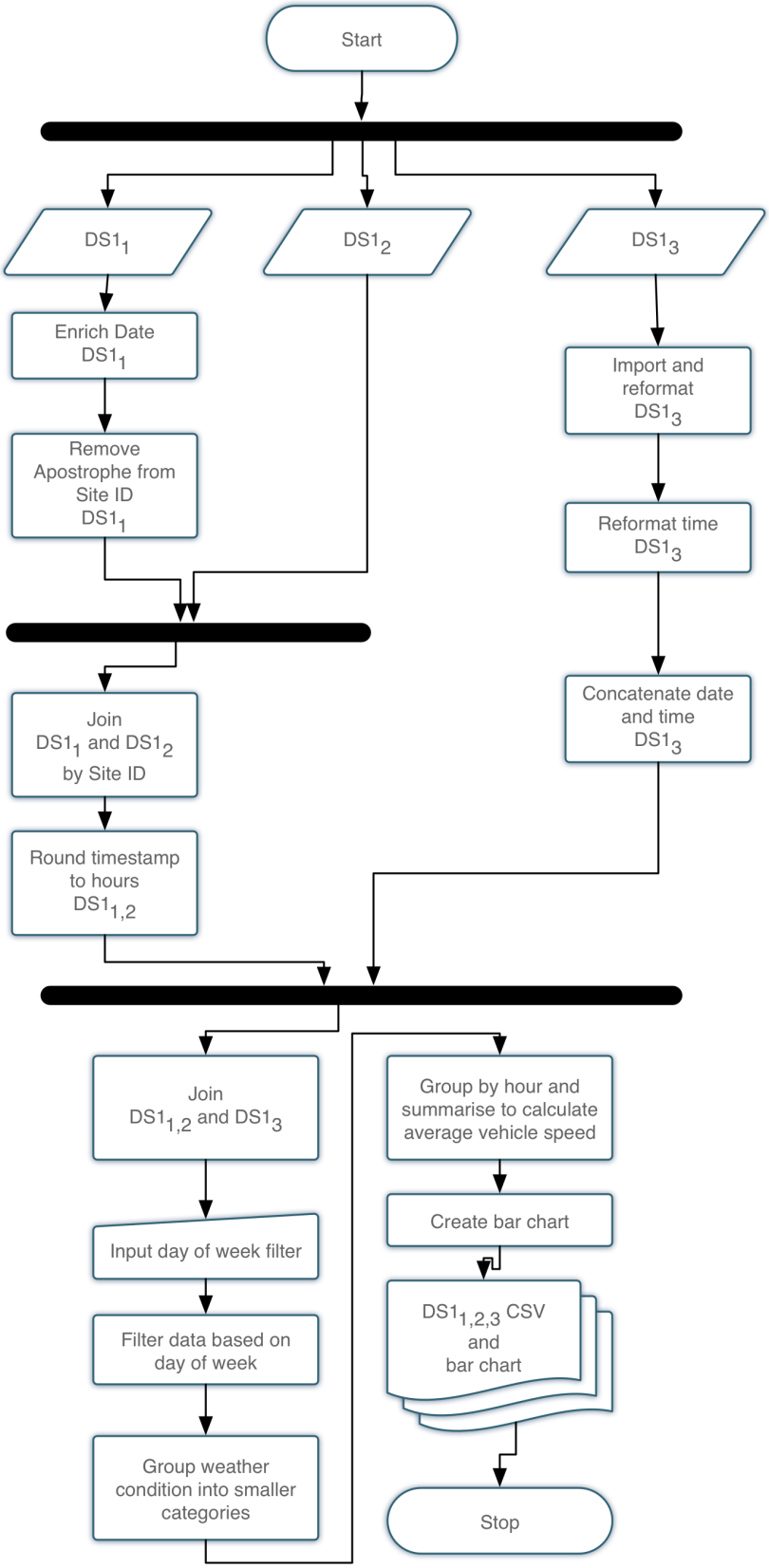


Figure 3.1: Flowchart as the formal representation of DWT_1

week represents its temporal characteristic. Finally, missing values were imputed. The complete formalisation is illustrated by Figure A.1.

3.1.3 Data Wrangling Task 3: DWT_3

The aim of DWT_3 was to analyse traffic volume pattern over the week from one of the busy roads of Manchester. The task tested the following hypothesis. *Hypothesis 3: The volume of traffic varies significantly throughout the day, and from one weekday to the next, but this variation is more obvious on weekdays, when the volume of traffic presents its highest values at particular times of the day, i.e. rush hours.*

To achieve the aim of data wrangling task DWT_3, the required datasets were defined. The first dataset was the traffic observation data. The second dataset was the traffic observation site reference. The second dataset was important to identify the road segment. The two datasets were identical to $DS1_1$ and $DS1_2$ respectively. For this task, the datasets were named $DS3_1$ and $DS3_2$ accordingly. Furthermore, identical date enrichment was performed towards $DS3_1$.

The task formalisation of DWT_2 is presented illustrated by Figure A.2. Similar to DWT_1, before the traffic observation and its site reference were integrated, the apostrophe character from the site identification attribute of $DS3_1$ was removed. The integrated dataset was named $DS3_{1,2}$.

$DS3_{1,2}$ contained traffic observation for both direction of a road segment. For this task, only one traffic direction of a road segment was observed. Thus, the dataset was filtered. Furthermore, the dataset was summarised to calculate the hourly volume. It was finally visualised into a line chart.

3.1.4 Data Wrangling Task 4: DWT_4

The aim of DWT_4 was to remove outliers from a dataset using simple statistics. The task tests the following hypothesis. *Hypothesis 4: Outliers in traffic data could be detected by its speed. given observation site, day of week, traffic direction, lane, and day of week, traffic observation outliers could be removed.*

There was only one dataset required to accomplish DWT_4, which was the traffic observation data from inductive loop identical to $DS1_1$, $DS1_2$, and $DS1_3$. The dataset was named $DS4_1$ for this wrangling task.

Traffic observation data was filtered according to observation site, traffic direction,

lane, and day of week. Day of week was extracted from the enriched timestamp attribute of $DS4_1$. Filtering was performed to enable visualisation using a box plot in the latter step.

Outliers are observations which values are abnormally far from the majority of a data population [citation]. To identify observations which were deemed as outliers in terms of vehicle speed, statistics were calculated to infer lower and upper outer fences. Observations which speed were outside of these boundaries were regarded as outliers and, thus, filtered out. The statistics were calculated by taking into account the traffic data spatial and temporal characteristics. Finally, the cleaned dataset was visualised. The complete formal definition of the task is presented by Figure A.3.

3.1.5 Assumptions

The following assumptions were held true for the traffic observation dataset used in this research.

- Observations were pre-sorted in an ascending order: the oldest observation was located in the first row, while the latest observation was located at last row on the dataset.
- The inductive loop sensor used for recording road traffic worked perfectly. As such, the dataset was complete: there were no vehicles between vehicles observed at obs_n and obs_{n-1}.
- There were vehicles at each hour within the observed period.
- The date attribute was consistent for all observations. It consisted of minutes, seconds, and milliseconds.

The assumption held for the weather observation dataset was that weather observed from a site represented the weather for locations nearby.

3.2 Wrangling Operations Summary

Preliminary experiments were conducted to determine which wrangling operations were required by each wrangling task. The experiments involved data wrangling tools reviewed in Chapter 2: Trifacta Wrangler, Data Wrangler, R, OpenRefine, and Python. However, as described in section 2.4, it is important to note that the API of Trifacta

Wrangler and Data wrangler were not available. Thus, neither tools would be included in the mashup. The complete wrangling operations required by the four wrangling tasks and from which tool the operations were fulfilled are described in Table 3.1.

The Date attribute from traffic observation dataset referred in all data wrangling tasks was incomplete: the attribute needed to be enriched. This function was specific to the dataset and it did not exist in the wrangling tools reviewed. However, the function was critical for the wrangling tasks. Likewise, the integration of traffic observation $DS1_{1,2}$ and weather observation $DS1_3$ datasets was specific for temporal and spatial data, which was also not provided by the reviewed wrangling tools. Integration for the two dataset required the function to match nearby geographical locations and temporal characteristics. Therefore, to satisfy both requirements, an implementation using Python script was proposed.

R offered a selection of data wrangling operations provided by its `dplyr` and `tidyr` packages. Using the operations provided by both packages, the majority of the wrangling requirements from all tasks were satisfied. Removing the leading apostrophe character in the site identification, for example, was performed using its `mutate` function. Furthermore, R also provided operations for data integration. The function was used for DWT_1 and DWT_3. Experimentation for data imputation in task DWT_2 was performed also using R. Its combination of grouping and column creation functions enabled median value imputation for headway and gap. Furthermore, experimentation of DWT_4 using R alone proved that outlier detection and filtration could be performed.

Data visualisation functions were provided by both Python and R. However, as described in Subsection 2.5.2, OpenCPU automatically generated a PNG image file at a data visualisation function call. Due to ease of use provided by the framework, the latter approach was preferable.

The weather observation dataset $DS1_3$ from the Met Office was a JSON formatted file. A JSON file format contained key-value pairs: the key indicated the attribute name; the value could be a plain string, numerics, a JSON sub-tree or an unnamed list. R, Python, Trifacta Wrangler, and Data Wrangler were able to import flat JSON files into a tabular format. The JSON file of $DS1_3$, however, contained a list and thus it was not a flat JSON file. OpenRefine, on the other hand, was able to handle a JSON file of such structure. Therefore, $DS1_3$ was handled using OpenRefine. Moreover, OpenRefine was equipped with the functions to completely wrangle such file format.

Table 3.1: Wrangling operation requirements for the four wrangling tasks

No	Wrangling Operation	Wrangling Tasks				Wrangling Tools		
		DWT_1	DWT_2	DWT_3	DWT_4	OpenRefine	R	Python
1	Import Tabular File Format	+	+	+	+		+	
2	Import JSON Formatted File	+				+		
3	Export to Tabular File	+	+	+	+	+	+	
4	Select columns	+	+	+	+	+	+	
5	Filter	+		+			+	
6	Create Column	+				+	+	
7	Rename Column	+					+	
8	Fill Down	+				+		
9	Enrich Timestamp	+	+	+	+			+
10	Group By	+	+	+	+		+	
11	Summarise	+		+			+	
12	Left Join	+		+			+	
13	Spatial and Temporal Join	+						+
14	Box Plot				+		+	
15	Bar Chart	+					+	
16	Line Chart			+			+	

3.3 Architecture

After data wrangling operations required by the use cases and from which tools they were fulfilled were understood, a software solution architecture was designed.

Software architecture, by definition, is a high-level perspective of a software solution which allows the software to be extended in the future [41]. Boehm et al [42] argued that architectural design would benefit software project in its development phase. The architecture was inspired by mashups architecture proposed by Wohlstadter et al [43]. Similar to their design, the architecture proposed in this research had a client-side layer which managed the orchestration of web services. However, data processing in the design proposed by Wohlstadter et al was performed in the client side. In their case, the size of data was manageable by a web browser and, thus, it could be inferred that the data involved in their research were not voluminous. On the contrary, the author proposed that in this research the data processing was executed on the server side due to the data size and system scalability.

The final architectural design is illustrated by Figure 3.2. The design comprised three layers: User-Interface, Services and Data Sources. Only User-Interface layer is inside the internal environment. Data Store and Services layers are both in the external environment. Data store layer is defined as the remote locations from which datasets were retrieved. Each API in the Services Layer was loosely coupled from one another. As such, one API could be maintained without disrupting services provided by other servers [44].

Intermediary layer as user-interface is not a novel approach to a software solution architecture. It is also known as middleware, which Issarny et al [45] in their article defined as a tool that provide a bridge to connect and coordinate heterogeneous environments. The end-user interacted with the mashup through this layer: it was the layer from which data wrangling workflows were designed using Taverna Workbench. The features of Taverna Workbench enabled designing workflow which orchestrated web services and other locally available services [46]. Using its drag-and-drop GUI, mashups of data wrangling operations could be designed. Moreover, the users executed available workflows from this layer using the Workbench.

R, OpenRefine, and Python services were each assigned to a designated server. OpenCPU served data wrangling operations from R packages; OpenRefine server hosted a list of its own functionalities; and a separate Python server which provided wrangling operations specific for traffic data. These servers construct the services layer. Data wrangling operations were performed in this layer.

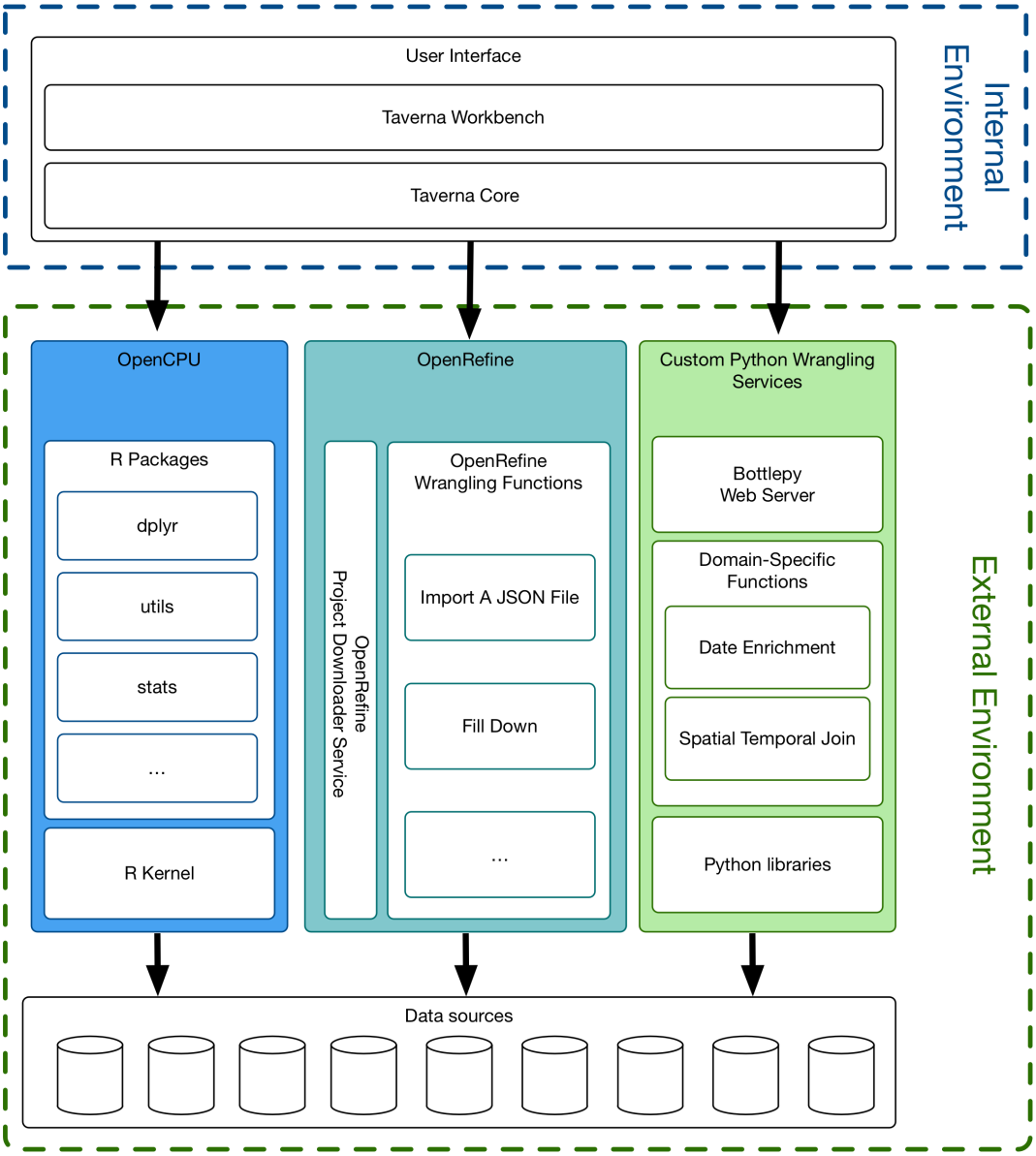


Figure 3.2: Software architecture of the mashup

Accepted data format varied dependant to each service. To enable communication of data towards each service, data formats must be translated into one which the destination wrangling service provider understands. All services commonly understand tabular data format, i.e. CSV files. JSON format files were acceptable by the OpenRefine services.

OpenRefine project had to be exported into a CSV file before the data was readable by other services. It furthermore forced its clients to download the exported project. Hence, the exported data was not stored in OpenRefine server. This exposed a challenge for this architecture: if the data had to be transmitted from the Services Layer to the User-Interface Layer in the middle of a data wrangling task, the aim of the design would not be efficiently achieved[2]. As a solution, an intermediary OpenRefine project downloader service was proposed. The service interacted with the OpenRefine server to export an OpenRefine project into a CSV file and store the file locally.

3.4 Summary

This chapter has covered the design concept for building web mashups of data wrangling operations. Four data wrangling tasks have been proposed as use cases. The tasks required wrangling operations from three tools those were accessible via HTTP: R, OpenRefine, and Python. The architecture for such concept has been presented. It was designed to minimise network traffic towards the end-user. End user interacts with the system via Taverna Workbench, where they are able to design a data wrangling task workflow.

[2]This contradicted the example use of Taverna given by Wolstencroft in his paper [46], where services were executed in the local machine to minimise network load.

Chapter 4

Implementation

In this chapter, a description of the implementation of the mashup is described in detail. The chapter begins with an introduction of agile development methodology and a description of the development plan. It is followed with the explanation of the technological explanation of the implementation environment. Furthermore, the interaction between Taverna Workbench and the external data wrangling components to implement the traffic data wrangling use cases explained in Chapter 3 is thoroughly described. Although tests and evaluations were performed throughout the implementation phases, they are addressed in Chapter 5.

4.1 Agile Development Methodology

Project management history in the domain of software engineering started with the waterfall methodology, which was a linear process of requirements analysis through the testing phase. This model was deemed not suitable for software engineering [47]. Software projects carried out using this model had a low success rate [41]. Following the failure, iteration model was developed before the idea of agile development was eventually coined. The earliest agile development methodology was the extreme programming (XP). It was successfully implemented not only because it produced high quality products but also the cost for changes was lower [41].

Yadav et al [48] in their paper compared between agile and traditional iterative methods. They argued that agile methodology has advantages over conventional iterative development due its: incremental characteristics, customer involvement throughout the lifecycle, project transparency, flexibility to changes, and parallel activities. Moreover, it allows rapid prototyping in each iteration, which is essential to ensure

that the development is in the right direction [49]. It is then tested. The testing yields a feedback. If bugs are found or changes are necessary, they would be processed immediately at the next iteration.

Due to the reasons explained above, agile methodology was used in implementing the mashup. The initial plan for the implementation is elaborated in the next section.

4.2 Implementation Plan

The signature of agile methodology is to iterate and increment throughout its small iterations. The author borrowed the term sprint from the Scrum methodology to represent small iterations [50]. A sprint consists of implementation of a set of requirements which outputs a deliverable product[1] at the end of each sprint. Furthermore, an implementation plan was constructed. The plan is explained as follows.

The implementation of DWT_1 covered the interaction between Taverna[2] with the majority of wrangling operations except for creating line charts and box plots. It was planned these were implemented in Sprint One. Moreover, Sprint One was planned to include wrapping R functions to generate chart graphics. The line chart and box plot functions were used in the implementation of DWT_2 and DWT_4 respectively. Due to the large size of Sprint One, the implementations of DWT_2, DWT_3, and DWT_4 were planned to be performed in Sprint Two.

DWT_1, DWT_2, DWT_3, and DWT_4, as described in Table 3.1, shared common wrangling operations. The Taverna interactions with these operations were implemented in Sprint One. However, they were not readily reusable. As such, Sprint Two was planned to focus on implementing reusable interactions between Taverna and each wrangling operation from R, OpenRefine, and Python. Lastly, DWT_2, DWT_3, and DWT_4 were implemented using the reusable workflows.

A use case implemented using the reusable interactions exhibited complexity due to its large workflow file size. To hide and reduce the complexity, the reusable workflows were then planned to be encapsulated as Taverna Components, as explained in section 2.8.4, in Sprint Three. Furthermore, the implementation of the wrangling tasks were improved by utilising the components.

[1] In Scrum, it is commonly known as product backlog [50].
[2] Taverna Workbench, in this chapter, is referred to as Taverna for short.

4.3 Environment

The environment used in the development of the mashups tool proposed in this research is described in Table 4.1. The files relevant to the data wrangling tasks were stored in the MAMP server running on port 80. The mashup was implemented in the client-side utilising Taverna Workbench Core 2.5.0[3].

R installation was downloaded from its official page[4]. RStudio was also downloaded and used for the implementation to help compiling an R project into an R package. A set of R packages: `tidyr`, `dplyr`, `ggplot2`, and `opencpu` were installed. The latter was an R web server framework which enables R library functions to be exposed as a web service. It required XQuartz to be installed[5].

OpenRefine version 2.5 was used[6]. The version was released in December 2011 but it was claimed by the developers to be the latest stable version. The maximum main memory allowance for OpenRefine was increased to enable data type inference of the JSON file relevant to DWT_1[7].

4.4 Sprint One

Sprint one was aimed to implement all interactions between Taverna Workbench and R, OpenRefine, and Python. The interactions are described in this section. Sprint One was organised into three sub-sprints based on which wrangling tool a function was called from. The first sub-sprint focused on the interaction between Taverna Workbench and R functions, which were exposed as web service using OpenCPU. It was followed with the explanation of the implementation of the interaction between Taverna and OpenRefine. Finally, DWT_1 was implemented in the last sub-sprint.

[3]Random Access Memory (RAM). According to its developer, the software required at least 2 GB of main memory

[4]`https://www.r-project.org`

[5]XQuartz provided the required libraries for OpenCPU to run. The disk image for its installation was downloaded from `https://www.xquartz.org`

[6]Although the official software name for the version of choice is Google Refine, in this research it will be referred to as OpenRefine.

[7]By changing the `VMOptions` property in the configuration plist file with the following value. `-Xms256M -Xmx4096M -Drefine.version=r2407`

Table 4.1: Environment for system development

Aspect	Details
Machine	MacBook Pro (Retina, 13-inch, Early 2015)
Operating System	OS X El Capitan Version 10.11.5 (15F34)
Processor	2.7 GHz Intel Core i5
Main Memory	8 GB 1867 MHz DDR3
XQuartz	XQuartz 2.7.9
Java	Java version "1.8.0_91" Java(TM) SE Runtime Environment (build 1.8.0_91b14)
MAMP	MAMP 3.0
R	R Version 3.2.3 OpenCPU Version 1.6.1
Python	Python 2.7.10
Taverna	Taverna Workbench Core 2.5.0
OpenRefine	Google Refine Version 2.5 (r2407)

4.4.1 Sub-Sprint: Calling R Functions using OpenCPU

The design approach for calling OpenCPU functions has changed throughout the implementation. In was understood that REST web services consumed string parameters, while R functions commonly required a complex R expression to be passed as a parameter. In the earlier experiments, the selected approach was to wrap R functions into a package which consumed string parameters, which was then evaluated as an SE[8] expression. In the later stages, experiments showed that OpenCPU was able to consume complex expressions, i.e. NSE, transmitted via HTTP protocol, as well as lists and vectors. Therefore, the earlier approach was abandoned.

The implementation began with the requirement for importing a CSV file using R. It was understood that R imported a dataset from a remote server using functions from the `utils` package, which included a function for reading a CSV file, namely `read.csv`. The implementation of Taverna Workbench interaction with the function is illustrated by Figure 4.1. To interact with this function, a REST service component from Taverna Workbench was used. It is shown by the dark blue rectangle in the figure. This component was also used to call other wrangling functions. The function

[8]Standard Evaluation, as opposed to Non-Standard Evaluation (NSE). It is common in R that a function accepted complex R expressions as parameters. The complex expression is then evaluated by a certain package, i.e. `lazyeval`, which is the NSE. However, due to interoperability issues, R also provides evaluation of simple expressions, i.e. the SE. These expressions are evaluated as plain strings.

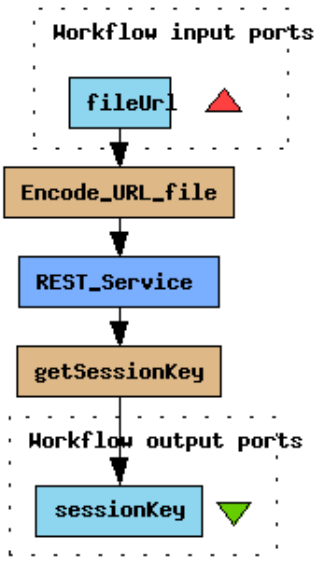


Figure 4.1: Workflow to Read Data from a URL to OpenCPU

`read.csv` consumed a web URL string as the parameter. Using OpenCPU as the R web server, this parameter was sent as the HTTP request body. This approach was common for Taverna interactions with functions provided by R. It is important to note that the interaction was executed via HTTP. In such protocol, there were prohibited characters to be used in the request body, i.e. escape characters. As such, the request body had to be encoded into a format which was accepted by the protocol. The request body encoding accepted by the protocol and by OpenCPU was the `x-www-form-urlencoded`. Unfortunately, the REST service component did not include a feature which automatically encode the response body. Therefore, the request body was encoded using the Beanshell component[9], which was explained in section 2.8.2.

The response of an R service, as was described in section 2.5.2, consisted of several lines. The data session key which pointed to the data which was read by R contained in the first line. To extract the data session key, the Beanshell component was once again used. This data session key was then used as a parameter for the Taverna interactions with data wrangling operations provided by R.

After data session key was ready, the next step was to implement the interaction between Taverna and R data wrangling functions, which were available in the `dplyr` package. The functions were as follows.

[9]the brown-coloured rectangle in Figure 4.1

- `select`. This function was used to select a list of columns and drop the rest. This function consumed a list of selected column names from a dataset `.data` as parameters.

- `filter`. As the name implies, this function enabled filtration of a dataset `.data`. The filter condition is given in an R expression.

- `group_by` was a function which divided observations of a dataset `.data` into several groups. A group is defined using R expression. It is mandatory to give a name to the data group.

- `summarise`. It was a function which was used to aggregate a dataset `.data`. Data aggregation could be performed to a dataset which had been previously grouped. As such, summarisation was performed to each group.

- `mutate`. This function was used to create new columns for a dataset `.data`. The new columns were given their respective names. This function could be used in conjunction with `group_by`. In such cases, the column values were calculated with respect to a group where a record belonged to.

- `left_join` was a function which integrated two datasets `x` and `y`, given key columns to which they were joined on. The function yielded a dataset `z` which contained all observations from `x` coupled with matching observations from `y`.

From the list of functions above, it could be inferred that all functions had a common parameter which referred to a dataset, i.e. `.data`, `x`, or `y`. The dataset parameter referred to the data session key provided from the implementation of reading a CSV file above. Moreover, the data wrangling functions above produced a data session key which pointed to their respective result. Thus, a sequential interaction to data wrangling function in R could be performed. Similar approach was taken to implement the interactions between Taverna Workbench and R wrangling functions: parameters were encoded, REST API was called, and data session key was extracted.

4.4.2 Sub-Sprint: Encapsulating Chart Functions in R

Line charts, bar charts, and box plots contain information on both x and y axes, graphics title, and legends. Due to the requirements of creating line and bar charts with grouping functionality, it was decided that the two charts were implemented using

`ggplot2` package. The charts were drawn procedurally by adding layers of chart components before finally the complete chart was generated. The components were: data ingestion, chart type settings, grouping options, chart colouring, legends options, and chart title. Such method was not possible to be called from OpenCPU, as it only allowed calling functions from R packages. As a solution, the chart generation procedures were encapsulated in an R package.

This, however, was not the case for drawing box plots. In our use cases, the requirement for drawing a box plot did not include variable grouping, and thus, `boxplot` function from `graphics` package was sufficient to satisfy the requirement. It required a single function call to produce a complete box plot; it was not necessary to add chart components which were required in creating line charts and bar charts.

Bar and line chart creation procedures were encapsulated into each a function and compiled into an R package using RStudio. Both functions shared four common parameters: the dataset from which the chart was drawn, variable name to be drawn on the x and y axes, and to which variable the data was grouped. After a successful build, the chart functions were available to access via OpenCPU as a REST web API.

The final aim of Sprint One was to implement data wrangling task DWT_1, which required interaction with the bar chart creation function. The interaction between Taverna Workbench and the bar chart creation function was implemented using a method similar to the interaction between Taverna Workbench and R data wrangling functions. Firstly, parameters were encoded into a format which was accepted by HTTP using the Beanshell component. The REST service component was used to call the chart creation functions, and finally, the data session ID was extracted from the REST service response body.

4.4.3 Sub-Sprint: Taverna Interaction with OpenRefine Functions

Data wrangling task DWT_1 required weather observation dataset $DS1_3$ to be wrangled. $DS1_3$ was a JSON file. The data contained in the JSON file was tree-structured: it consisted of a list of weather observations inside a JSON node. R was able to parse a JSON formatted file should it not contain a list, i.e. typical key-value pairs. Thus the structure of $DS1_3$ was not possible to be parsed using R. On the other hand, OpenRefine was equipped with the feature to parse complex tree-structured JSON files. As such, The interaction between Taverna Workbench and OpenRefine was implemented.

The procedure for importing data from a remote URL using OpenRefine was previously explained in section 2.6.2. The procedures were sequentially: create importing

job, import file, and create project. The interactions between Taverna and OpenRefine were implemented to reflect the procedures, which is illustrated by Figure 4.2. Firstly, an importing job was invoked using the REST service component. OpenRefine responded to this service call by providing an import job ID, formatted in JSON. It was then parsed using the JsonPath component[10].

The procedure continued with sending an import file request to OpenRefine server. This request was sent by passing the import job ID and remote file URLs as parameters. More importantly, OpenRefine server specifically demanded the request encoded as multipart form data or multipart mixed stream. It was impossible to perform the request using REST service component because it did not feature such encoding. Therefore, the request wast implemented using the Beanshell component. The import files request was an asynchronous process: OpenRefine server did not send the response immediately. Thus, the import files status had to be checked repeatedly until the JSON files were readily imported by OpenRefine server. A repeated interaction was implemented using a nested workflow, which is illustrated as a large, light blue rectangle in Figure 4.2. The nested workflow contained several components: a REST service component which was designated to request the import job status is coloured dark blue, and a JsonPath which parsed the job status is indicated by the green rectangle. The nested workflow was configured to loop until the job status was read ready. When OpenRefine server informed that the import job had been successfully carried out, the next step was performed: creating OpenRefine project. Similar to import file request, The project creation request was an asynchronous process. The job status had to be checked repeatedly until the server informed that an OpenRefine project was successfully created. Likewise, it was performed using a nested workflow, which was looped until OpenRefine server returned a project ID.

Furthermore, the scope of operation moved from an OpenRefine import job to an OpenRefine project. Thus, the import job was removed.

The interactions between Taverna and OpenRefine to wrangle data were then implemented. Based on the requirements for DWT_1 described on Table 3.1, data wrangling functions required from OpenRefine were: fill down, drop column, rename column, and create a new column. Fill down was necessary to impute missing values that was due to data format transformation from a tree-structured JSON file into a tabular data format; drop column was required to remove irrelevant information; rename

[10]JsonPath component enabled extraction of a value from a JSON file.

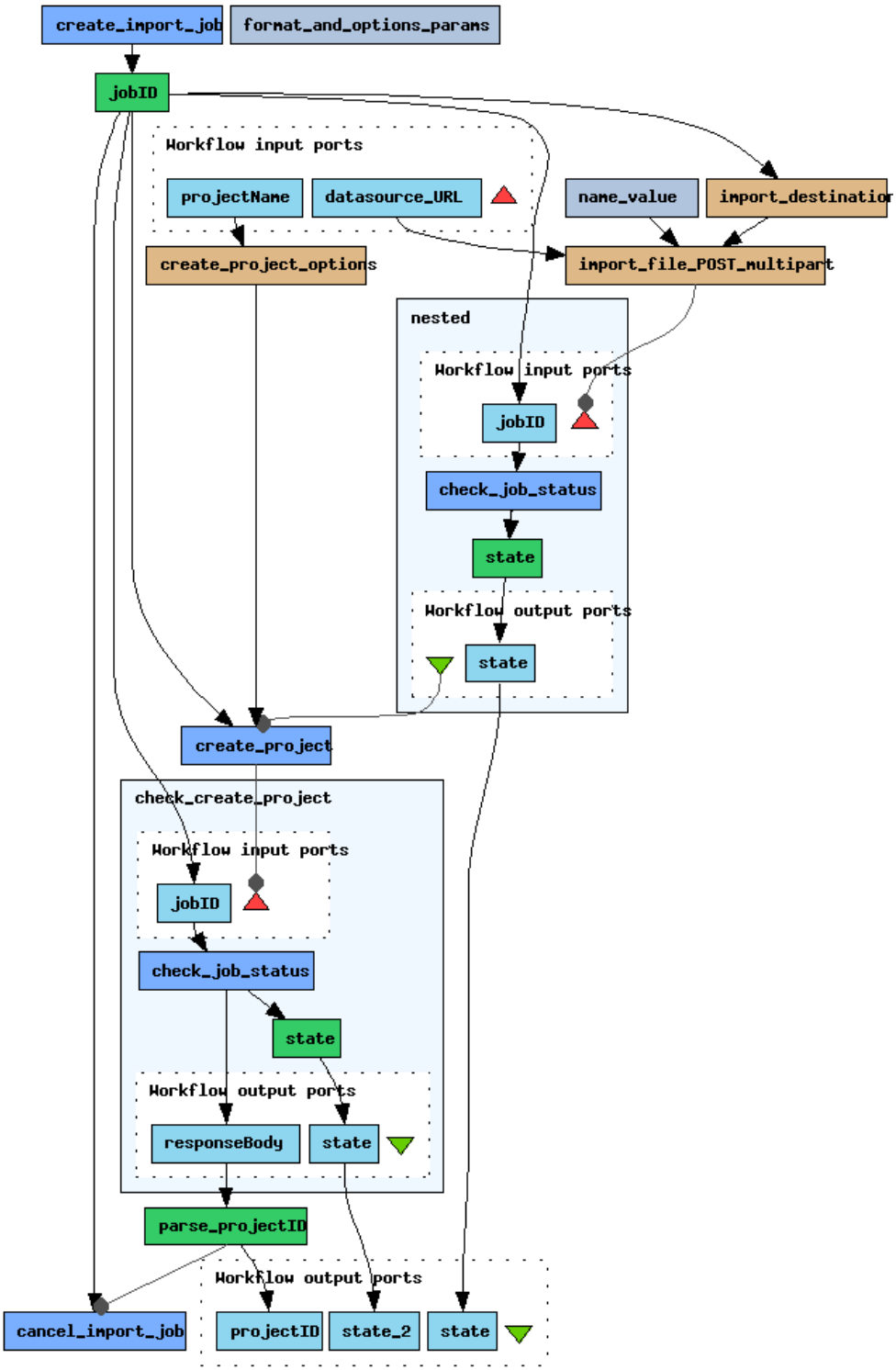


Figure 4.2: OpenRefine processes for importing a JSON file implemented as a Taverna interaction

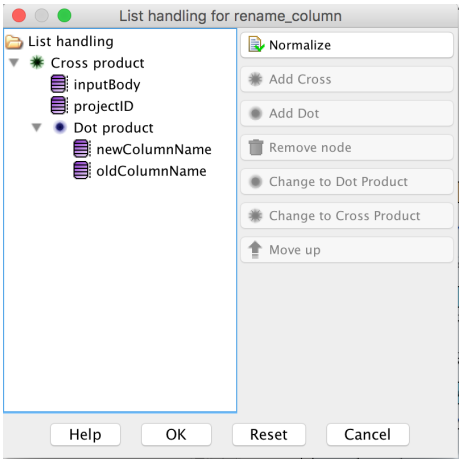


Figure 4.3: OpenRefine Rename Column List Handling

column was crucial for renaming non-meaningful column names which were automatically generated by OpenRefine when the JSON file was imported; and create new column was crucial for wrangling the time format of $DS1_3$[11]. The interactions of Taverna with the earlier three functions were implemented using nested workflows. As such, the user could perform fill down, dropping, and renaming of multiple columns in one interaction. The nested workflows were configured to handle a list of parameters. The interaction implementation for dropping and filling down columns were configured to handle parameters as cross product as the two functions received only one list of column names. On the contrary, renaming multiple columns required two lists of column names: old column names to be renamed and their corresponding proposed names. As such, the two list of column names were handled as a dot product[12]. The configuration for renaming columns in OpenRefine is shown by Figure 4.3.

Finally, the OpenRefine project was exported. An OpenRefine project format was exclusive to OpenRefine. It must be exported to a CSV file before other wrangling services could use the data, and, as described in section 2.6.2, it was not possible to directly retrieve a CSV file from the OpenRefine server. The CSV file was only available if and only if a request to export the project was sent to OpenRefine server. Moreover, OpenRefine forced to download the file. To achieve this, as was described in section 3.3, a downloader web service was built using Python. The purpose of this

[11]The time format for $DS1_3$ was in minutes calculated from midnight, i.e. 00:00.

[12]A cross product configuration matches an element of two list to each other parameters. For example, two lists of n column names configured as cross product yielded n^2 pairs. In contrast, a dot product matched each element from the two lists in one-to-one manner: the first on the old column name list was paired only with the first of the new column name list, and so on, yielding a list of n pairs.

web service was to send export project request to the OpenRefine server. The web service then internally provided a unique filename to the exported project and finally stored the exported OpenRefine project to its own file server. Furthermore, the exported OpenRefine CSV file was retrieved from the Python server.

4.4.4 Sub-Sprint: Traffic Data Wrangling Web Services in Python

$DS1_1$, $DS2_1$, $DS3_1$, and $DS4_1$ all referred to the same traffic dataset provided by TfGM. The file was formatted as a CSV file. More importantly, the observation time consisted only of minute, second, and milliseconds. There was no information of observation year, month, day, and hour. The file name, however, included information of the start and end date of the data. This information was deemed as important to enrich the observation time attribute. A function which enriched the timestamp was implemented by taking account of the information regarding the start date of the data.

The timestamp enrichment function created a new column `Complete Timestamp` which contained complete date and time information for each observation. The algorithm iterated throughout observations in the dataset. As it progressed, a check was performed to compare if the minute of current observation n was less than previous minute of the previous observation $n-1$. If it was, the hour was incremented. Furthermore, the hour was checked whether it has surpassed the day limit, i.e. 24 hours. If so, then the day was incremented and hour restarted to zero. The enriched timestamp was then concatenated to the original traffic data. The function returned the filename of the resulted data.

Furthermore, preliminary experiments had found that geographical location of traffic observation site and the location of the weather observation sites did not exactly match. There was a dispute between geographical locations of the traffic observation site and the weather observation site which represented weather for Manchester. To solve the geospatial entity matching problem, coordinate matching was implemented [51]. The distance function used in the implementation of coordinate matching was Cosine-Haversine formula [52]. The Cosine-Haversine formula calculated the distance between two geospatial[13] locations. Both traffic and weather datasets were spatial and temporal. The coordinate matching had solved the spatial character. The temporal character was solved using the temporal matching. And thus, the space and time join function. The algorithm for space and time join implementation is described

[13] latitude and longitude

in Algorithm 1. The algorithm took datasets x and y as parameters. The algorithm allowed configurable thresholds for the time and space distances: if the nearest weather observation was greater than the accepted threshold, then it was neglected. The function yielded a dataset which merged the two datasets. Similar to the date enrichment function, the result dataset was then given a unique name and stored. The space and time join code implemented in this research used merge sort algorithm[14] to sort the result data. However, the time complexity of the space time join implementation was $\Theta(n^2)$. The implication of the high time complexity was that the space and time join did not perform efficiently for merging large datasets. The investigation and improvement of the space and time join implementation was beyond the scope of this research.

Both functions were implemented and wrapped as a web service in Python.

Finally, the interaction between Taverna and both date enrichment and space and time join functions were implemented. Similar to the previous sub-sprints, the parameters were firstly encoded to HTTP-accepted format using the Beanshell component. The encoded parameters were then mapped as the request body of a REST service component. The output of the two interactions was a URL which pointed to the result datasets.

4.4.5 Sub-Sprint: Implementation of DWT_1

After the interactions between Taverna and the wrangling tools were implemented, the workflow for DWT_1 was developed. The full wrangling requirements for DWT_1 is presented in Table 3.1. The step-by-step interaction between Taverna Workbench and the wrangling tools is illustrated by Figure 4.4.

Data wrangling task DWT_1 required the traffic observation dataset $DS1_1$ provided by TfGM. The dataset contained observed vehicle data passing through an observation site. The data had a date attribute which consisted of minutes, seconds, and milliseconds: it was an incomplete date and time attribute. By using the assumptions stated in Section 3.1.5, Taverna Workbench interacted with the enrich date service from the Python server. The Python service retrieved $DS1_1$ from the web server. This interaction wrangled the date column of $DS1_1$ and produced a new complete date and time attribute.

Secondly, we wanted to integrate the traffic data with the weather observation. The purpose of such integration was to get the weather condition during the time when

[14]Merge sort algorithm has a time complexity of $\Theta(nlog(n))$[53]

Algorithm 1 Space Time Join Pseudocode

```
1:  function SPACETIMEJOIN(x, y, threshold)
2:      x̂ = READFROMURL(x)
3:      ŷ = READFROMURL(y)
4:      d = ARRAY()
5:      x_rows = SIZEOF(x̂)
6:      for it = 0 to x_rows do
7:          δ = CALCULATEDISTANCES(x̂_it, ŷ)
8:          SORTASCENDING(δ)
9:          if δ_0 ≤ threshold then
10:             d_it = δ_0
11:         else
12:             d_it = NA
13:         end if
14:     end for
15:     result = MERGE(x̂, d)
        return result
16: end function
17: function CALCULATEDISTANCES(x, y)
18:     δ_haversine = HAVERSINEDISTANCE(x, y)
19:     δ_temporal = TIMEDISTANCE(x, y)
20:     δ = MERGE(δ_haversine, δ_temporal)
        return δ
21: end function
```

traffic was observed. However, $DS1_1$ did not contain geographical information of the observation site. The geographical information was available in the traffic observation site reference dataset $DS1_2$. Thus, $DS1_1$ was integrated with $DS1_2$. The integrated data was named $DS1_{1,2}$. Before the integration was performed, the two datasets $DS1_1$ and $DS1_2$ were read into the R server. Otherwise the server would not recognise the datasets. The integration was performed using a left join, which returned all rows from $DS1_1$ with the matching rows of $DS1_2$. Site observation ID was used as the join key. There was, however, a dispute between the values of site observation ID attribute in $DS1_1$ and $DS1_2$. In $DS1_1$, it contained a leading apostrophe character whilst the site observation ID attribute of $DS1_2$ did not. The apostrophe character was removed before the integration was executed. These wrangling operations were available in the R server and thus, Taverna interacted with it to send the instructions to perform such operations.

The weather observation dataset $DS1_3$ was a tree-structured JSON file. It was necessary to reformat the semi-structured JSON file into a structured tabular dataset

before any wrangling operations could be performed. To reformat the $DS1_3$ JSON file, Taverna Workbench interacted with the import service from OpenRefine. Irrelevant columns were dropped. The import service generated the column names automatically based on the structure of the JSON file. The long, non-meaningful column names needed to be renamed and thus, the columns rename interaction was performed. The wrangling functions were executed using OpenRefine. Furthermore, the OpenRefine project was exported as a CSV file. This was crucial because the format of OpenRefine project was not readable by other wrangling tools.

After the weather observation dataset $DS1_3$ was exported to a CSV file and $DS1_1$ and $DS1_2$ were integrated into $DS1_{1,2}$, Taverna interacted with the space and time join service to integrate $DS1_{1,2}$ and $DS1_3$. The merged data was named $DS1_{1,2,3}$. The service matched the traffic observation data with the nearest weather observation, both in terms of spatial and temporal characteristics.

Finally, Taverna interacted with the R server and sent instruction to read dataset $DS1_{1,2,3}$ yielded by the space and time join service. Taverna Workbench instructed the R server to summarise $DS1_{1,2,3}$ to calculate the average speed of vehicles passing at each hour and weather condition. Furthermore, The chart interaction was used to generate a bar chart from the summarised $DS1_{1,2,3}$ dataset.

To conclude, the implementation of DWT_1 wrangled and combined traffic observation dataset $DS1_1$, traffic observation site reference $DS1_2$, and weather observation $DS1_3$. The product of the wrangling task was a summarised dataset and a bar chart. In the implementation of DWT_1, Taverna interacted with wrangling services from R, OpenRefine, and Python servers. The Taverna workflow implementation for DWT_1 is presented by Figures C.1 to C.7, while the output of the workflow is shown by Figure D.1 .

4.5 Sprint Two

The interactions between Taverna Workbench and the wrangling operations were implemented in Sprint One. Furthermore, the majority of the interactions were used in the implementation of DWT_1. Due to the size of Sprint One, the wrangling workflow implementation of DWT_2, DWT_3, and DWT_4 were separately developed in Sprint Two. The required wrangling operations for these tasks are described in Table 3.1. From the table, it could be inferred that there were requirements those were commonly used by the four wrangling tasks. Group by, for example, was required by all the four tasks.

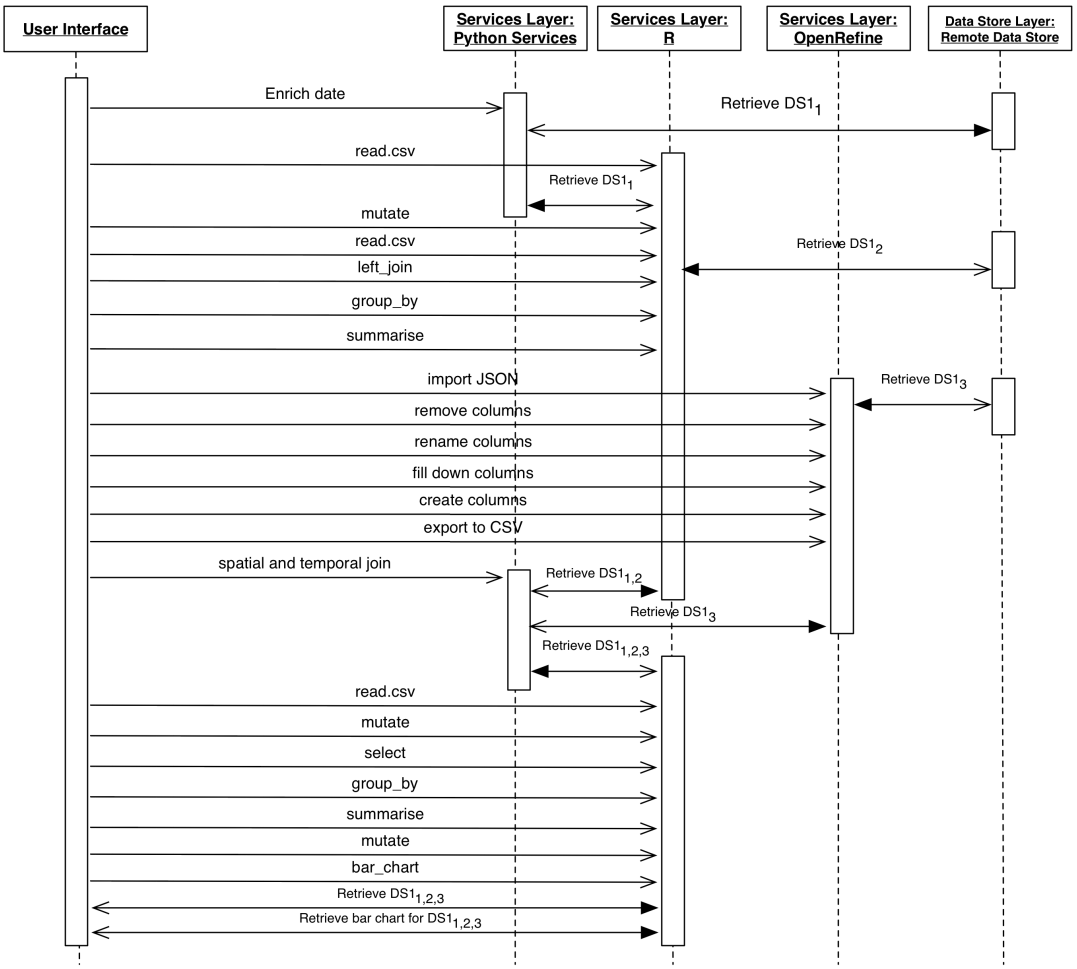


Figure 4.4: Activity diagram for the implementation of DWT_1 which represents the interaction between Taverna Workbench and R, OpenRefine, and Python services

Copying and pasting the interactions from one workflow the other was deemed inefficient and time consuming[15]. Therefore, in Sprint Two, before Taverna workflows for DWT_2, DWT_3, and DWT_4 were implemented, it was decided that the interactions between Taverna and the wrangling services were made reusable by grouping each interaction into a small workflow. These interactions were then used as nested workflows [54] in the implementation of DWT_2, DWT_3, and DWT_4.

In accordance to the plan above, Sprint Two was segregated into two sub-sprints. The first sub-sprint was dedicated to implementing reusable interactions between Taverna and the wrangling services. The product of this sub-sprint was then used to implement DWT_2, DWT_3, and DWT_4. The implementation of the three wrangling tasks were grouped into a sub-sprint due to the relatively small effort.

4.5.1 Sub-Sprint: Implementation of Reusable Interactions

During the implementation of Taverna interactions with both R and OpenRefine column wrangling services, i.e. create column, column rename, fill down, etc, the author observed a similarity. In the interaction of Taverna with R wrangling operation services, each operation required a session key and other relevant parameters. The output of R data wrangling operation interaction was a session key which referred to the result dataset. The typical interaction is illustrated by Figure 4.5. In the same way, the interaction with OpenRefine required a project ID to which wrangling operation was performed. Project ID in OpenRefine was analogous to the data session key in R. In contrast to the interaction with R services, the interaction with an OpenRefine wrangling function did not produce a project ID. However, the author believed it was beneficial to output the project ID for use of the next wrangling operation due to the nature of a workflow, which used the output of a process as an input for the subsequent process. Therefore, a common interaction template was designed to follow such interaction pattern. An interaction of Taverna with column wrangling services provided by R and OpenRefine required a data identifier and other parameters specific to each wrangling operation. The output was an identifier which referred to the result dataset. The date enrichment service was similar to the column wrangling services. Although the input was a URL and resulted in another URL, the pattern was similar.

Different pattern was observed for data import functions. In reading a CSV file, the R server required a remote data URL as an input. The output of such function

[15]Taverna Workbench did not allow selecting a group of components at one time unless they were transformed into a nested workflow.

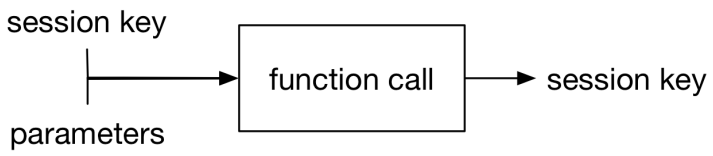


Figure 4.5: typical interaction of Taverna and the R server for column wrangling operations

was data session key, which referred to the imported data. Similar for OpenRefine, the interaction for importing a JSON file also required a data URL as an input parameter. The output of the import procedure was a project ID. The interaction for invoking data import, both in R and OpenRefine, required a data URL as the input parameter and produced a key which referred to the imported files. Moreover, data integration services, i.e. left join and space and time join, integrated two datasets based on key columns and yielded one dataset.

The implementation of Taverna interactions for data visualisation, i.e. box plot, bar chart, and line chart, showed similarities of their own. The three visualisation services required the drawn dataset, variables to be displayed on the x and y axes, and chart title. Bar chart and line chart had both one additional input parameter for variable grouping.

These patterns were used as the basis for the implementation of reusable Taverna interactions. Each interaction was implemented and stored as an individual small Taverna workflow. The sub-sprint produced reusable Taverna interactions, which were grouped into five categories: aggregation, integration, reader, transform, and visualisation. Aggregation group, for example, contained interactions for data aggregations, i.e. group by and summarise.

4.5.2 Sub-Sprint: Implementation of DWT_2, DWT_3, and DWT_4

The interactions between Taverna and the wrangling services were implemented as reusable interaction workflows in the previous sub-sprint. In this sub-sprint, the Taverna workflows implementation of DWT_2, DWT_3, and DWT_4, utilising the reusable interactions, is discussed. The three wrangling tasks employed the date enrichment interaction at the beginning of each workflow to enhance the date and time column of the traffic observation data. Further implementations are discussed below. The interactions between Taverna and the data wrangling services are illustrated by Figures B.1

to B.3 whilst the Taverna workflow implementation of the three tasks are shown by Figures C.8 to C.16. The output of the tasks are presented in Appendix D.

Implementation of DWT_2

The aim of data wrangling task DWT_2, as has been described in Subsection 3.1.2, was to impute missing gap and headway values from the traffic observation data $DS2_1$. According to the preliminary experiments, the interactions required for DWT_2 were shown on Table 3.1.

The missing values were imputed using the median value of the traffic data. The median value was calculated by taking into account the spatial and temporal characteristics of the traffic data. It was performed using subsequent group by and create column services provided by the R server[16]. After empty values had been imputed, the result data was downloaded to the client side. Taverna Workbench instructed the file to be downloaded using the REST service component. Finally, the data was stored into a local CSV file using the Taverna input-output component.

Implementation of DWT_3

The emphasis of DWT_3, as has been described in Subsection 3.1.3, was to observe traffic volume patterns over the week. According to the design, The task required two datasets: the traffic observation dataset $DS3_1$ and the site observation reference dataset $DS3_2$. To calculate the average hourly traffic volume, the Taverna interaction of data grouping and summarisation were used. Lastly, the line chart interaction was used to visualise the traffic volume data.

Implementation of DWT_4

As described in Subsection 3.1.4, Data wrangling task DWT_4 aimed to remove observations from $DS4_1$ which vehicle speed were deemed as outliers. Outliers were defined as observations which vehicle speed value were far from the normal observations. In statistics, an observation was deemed as an outlier if the value of its attribute was below the lower outer fence or greater than the upper outer fence. The majority of wrangling requirements for DWT_4, as was described in Section 3.2, were provided by the R server.

[16]By performing the mutate function on a grouped dataset, R calculated the new column value of each group.

Other than the date enrichment, the required services were provided by the R server.The statistics for identifying the outliers of $DS4_1$ were calculated by utilising the create new column, group by, and summarisation interactions. Using these statistics, outliers from the $DS4_1$ dataset were then filtered out. Finally, a box plot interaction was used to visualise the cleaned dataset.

4.6 Sprint Three

In the previous Sprint Two, reusable Taverna interactions were produced. These interactions were used in implementing data wrangling tasks DWT_2, DWT_3, and DWT_4 as nested workflows. Hence, unnecessary complexity was visible to the end user. The interaction for importing a CSV file in R, for example as shown by Figure 4.1, had three components: the Beanshell component which was used for encoding the parameters of the REST service component, and the Beanshell component which was used to extract data session key. Moreover, the interaction of Taverna and OpenRefine for importing a JSON file exhibited the minutiae of its procedure. The author felt that the complexity of an interaction had to be hidden from the end user because such details were not relevant to the wrangling task at hand. To hide such convolution, the interactions between Taverna and the wrangling services were transformed into Taverna components.

One objective of Sprint Three was to implement Taverna Components from the interactions between Taverna and the wrangling services. The second objective was to improve the wrangling tasks implementation by employing these components to hide the workflow complexity. Sprint Three was divided into two sub-sprints. The objective of the first sub-sprint was to transform the Taverna interactions with the wrangling services into each one component. The second sub-sprint was designated to improve the data wrangling tasks which had been implemented in the previous Sprint One and Sprint Two.

4.6.1 Migrating Interactions into Taverna Components

A Taverna component, as explained in section 2.8.4, was an encapsulation of a workflow. By encapsulating into a component, the complexity of the workflow was hidden from the end user. A Taverna component was a member of a Taverna component family. Taverna components within the same component family shared one component profile.

To create a Taverna component, a profile was a prerequisite. A Taverna component profile was an XML file contained a defined set of rules to which a component must conform. The set of rules ranged from the ports to the component annotations. Component profiles were defined for each Taverna interaction category: column wrangling, data import, data integration, aggregation, and data visualisation. A component profile for the implementation of the interaction contained rules for the input and output ports. The name of the ports were defined to be general to accommodate Taverna interactions with various wrangling operations from R, OpenRefine, and Python. A single input or output port was required to be annotated, i.e. have a description. The component profiles were created using a text editor. Secondly, component families were defined. Component families were created directly from within Taverna Workbench. The component families were registered to a Taverna component registry.

The Taverna interactions were adjusted to the requirements of the component profiles. Input and output ports of Taverna interaction for data filtering using R service, for instance, were renamed. The input port name for the data session key was renamed to `x` whilst the output port, which yielded another data session key, was renamed to `y`. Another example was the Taverna interaction for renaming column using OpenRefine. The interaction required the OpenRefine project ID as the input and output identical project ID. The input and output ports were renamed to `x` and `y` respectively. The input and output port names were not meaningful, but fulfilled the purpose of being general. Hence, input and output ports were annotated: each port was described to present the component user with an understanding of what input a component used and what output it produced. Furthermore, each component was given a description.

The result of this sub-sprint is illustrated by Figure 4.6. The root folder indicates the component registry. A component family is represented by a folder. Each folder contained one or more Taverna interactions which had been encapsulated into components. A component is shown as a U-shaped brick wall icon. A Taverna component was illustrated as a pink-coloured rectangle in the workflow.

4.6.2 Sub-Sprint: Improving Data Wrangling Tasks Using Taverna Components

The implementation of data wrangling tasks were deemed complex. The size of the workflows were large and the complexity of the interactions, which were unnecessary, was visible to the end-user. The Taverna interactions had been encapsulated and

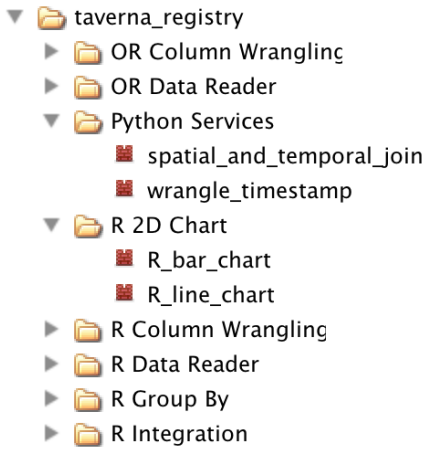


Figure 4.6: Taverna components organized into families

implemented as Taverna components in the previous sub-sprint. A component hid the complexity of the interaction between Taverna and the wrangling service. In this sub-sprint, Taverna interactions which were visible as nested workflows in the implementation of DWT_2, DWT_3, and DWT_4 were replaced by the Taverna components. More importantly, DWT_1 was not implemented using the reusable Taverna interactions. Hence, more effort for replacing the Taverna interactions in DWT_1 was expected.

Consequently, after Taverna interactions were replaced by Taverna components, the file size for all wrangling tasks were reduced. Before, the Taverna workflow file sizes of DWT_1, DWT_2, DWT_3, and DWT_4 were 1.1MB, 186KB, 325KB, and 511KB respectively. After the reusable interactions were replaced by Taverna components, the file sizes decreased to 296KB, 75KB, 110KB, and 114KB respectively. The output of all tasks remained identical.

4.7 Implementation Challenges

There were two challenges faced during the implementation phase. These challenges were summarised as follows.

- During the implementation of data wrangling tasks particularly DWT_1 using Taverna Workbench, the tool crashed on many occasions. It was believed that the size of the workflow caused the problem. Due to this matter, throughout the implementation, the workflow was saved frequently to prevent unrecorded changes.

- Taverna Workbench runs on Java platform and the machine which was used for the implementation phase this research was equipped with a retina display. There was a known bug in the JDK[17] that caused text to be rendered incorrectly[18]. The bug particularly interfered during programming using Beanshell component. As a workaround, Atom[19] was used.

4.8 Summary

This chapter has covered the implementation of mashups of data wrangling operations using Taverna Workbench. Taverna interacted with the wrangling tools via HTTP using REST service components from Taverna. Beanshell components were used to facilitate the inability of Taverna to encode parameters, extract data session key, and sending multipart form data.

Using agile methodology, the development processes were divided into three sprints to reduce risk and to deliver quickly. Sprint One produced the Taverna interactions with the wrangling tools. DWT_1 was implemented in Sprint One. Sprint Two has produced reusable Taverna interactions with wrangling services provided by R, OpenRefine, and Python. The reusable interactions were used to implement DWT_2, DWT_3, and DWT_4, which shared common wrangling interactions. Sprint Three was aimed to reduce and hide the complexity of Taverna interactions with the wrangling services, which was accomplished by encapsulating the interactions into Taverna components. Finally, the wrangling tasks implementation were improved using these components.

The next phase of the research was to test and evaluate the design and implementation, which has been covered in this chapter and in Chapter 3.

[17] Java Development Kit

[18] The bug had already been reported by a developer and could be found at the following URL. `https://bugs.openjdk.java.net/browse/JDK-8014069`

[19] A hackable text editor that was downloadable at `http://atom.io/`

Chapter 5

Testing and Evaluation

In the previous Chapter 4, it has been mentioned that agile methodology was employed to implement Taverna interactions for the data wrangling tasks described in Chapter 4. Hence, software testing was performed at the end of each sprint. Furthermore, evaluation has been conducted at the end of the implementation phase. This chapter covers both testing and evaluation which have been executed and discusses the findings. The chapter begins with the introduction of iterative testing approach. It is followed with a description of unit testing and integration testing carried out in this research. Further in the chapter, the evaluation procedure are described. The evaluation of Taverna Workbench are explained, followed by the description of the network load evaluation.

5.1 Iterative Testing Approach

Software testing is an integral part of a software development. It is vital for determining the quality of a software project [55]. Ellims et al [49] concluded that testing took up to 50% of software development effort. Nevertheless, they were necessary: by performing testing, bugs could be found and fixed before the product reached its audience.

The mashup implementation, as described in Chapter 4, consisted of three iterative and incremental sprints. At the end of each sprint, testing was performed before eventually next sprint is executed. In this section, tests performed during the development of mashups tool are covered. In this research, two types of tests were performed: unit and integration testing. They are discussed in the following sections.

5.1.1 Unit Testing

Ellims et al [49] furthermore defined unit testing as the assessment of every function point of a software. The motivation for unit testing in this research was to assess if the implemented function produce expected results given some input.

The strategy for unit testing was to assess functions implemented in this research, each of which was performed in their corresponding environment. Functions which were already available from R and OpenCPU were out of the scope of the unit testing. From the perspective of this research, these functions were third party services which were not implemented in this research and were assumed that they had been properly tested.

The scope of the implementation covered the following functions: custom wrangling operation, implementation of custom Java program using Beanshell component in Taverna Workbench, and encapsulation of data visualization functions. These activities were implemented using three different programming languages: Python, Java Beanshell script, and R respectively. The unit tests were performed in all sprints, although the majority were performed during Sprint One. It was a consequence to the decision that all Taverna interactions were implemented in Sprint One.

Chapter 4 has discussed the use of Beanshell components in the implementation phase. To summarise, they were used for the following activities. To encode REST parameters before calling remote services; to import file from remote URL using OpenRefine's import file function; to extract session keys OpenCPU responses; and to concatenate result data URLs.

Two services were implemented using Python: the traffic data wrangling service and OpenRefine file downloader. The implementation of traffic data wrangling service consisted of several function points. Each function was tested individually. Furthermore, the two web services were tested on each API function.

Lastly, data visualisation functions implemented in R were tested. They were tested both from the R console using Rstudio and as an API[1]. Bar and line charts implementation were expected to produce multi-coloured charts.

The results of unit testing are described in Table 5.1. It could be inferred that all features were successfully tested and behaved as expected.

[1]Testing functions as an API in OpenCPU was important. Preliminary experiments had shown there were R packages which did not perform as expected when exposed as API using OpenCPU, for example: Amelia package for data imputation.

Table 5.1: Unit Testing Summary

No.	Testing Component	Description	Result
1	**OpenRefine**		
1.1	Create import job	Create an import job using Taverna	Passed
1.2	Concatenate URL for importing job	Concatenate URL and query string for importing JSON file using Taverna	Passed
1.3	Import JSON from remote URL	Create Beanshell script to simulate sending HTTP multipart data	Passed
1.4	Encode create project parameters	Encode parameters for project creation using Beanshell	Passed
1.5	Create project	Create OpenRefine project using Taverna	Passed
1.6	Check job status	Check job status using Taverna REST service	Passed
1.7	Cancel import job	Cancel import job using Taverna REST service	Passed
1.8	Extract importing job state	Extract job state as a result of check job status using JsonPath service	Passed
1.9	Extract project ID	Extract project ID from job status using JsonPath service	Passed
1.10	Remove column	Remove column with encoded parameters using REST service	Passed
2	**OpenRefine Downloader**		
2.1	Random filename generator	Generate random filename of length 36 alphanumeric characters	Passed
2.2	Write to CSV file	Persist a data into CSV	Passed
2.3	Encode parameters in Taverna	Encode parameters for calling the download service in Taverna using Beanshell	Passed
2.4	Download from OpenRefine	Perform export and save of OpenRefine project using Python	Passed
3	**Python**		
3.1	Read from URL	Read a CSV data from remote URL into Pandas data frame	Passed
3.2	Write to CSV file	Write to a CSV file	Passed

Table 5.1: Unit Testing Summary

No.	Testing Component	Description	Result
3.3	Retrieve file	Retrieve a file persisted in the Python server	Passed
3.4	Enrich date	Perform date enrichment of traffic observation data	Passed
3.5	Space and time join	Call space and time join	Passed
3.6	Encode parameters in Taverna	Encode parameters for calling enrich date and space and time join	Passed
4	**R**		
4.1	Encode parameters for R functions	Encode parameters for calling R functions and the chart creation	Passed
4.2	Line chart	Encapsulate line chart creation expressions into a function	Passed
4.3	Bar chart	Encapsulate bar chart creation expressions into a function	Passed
4.4	Extract data session ID	Extract response body of R function result to retrieve data session ID	Passed
4.5	Retrieve image	Retrieve PNG image generated by chart functions	Passed

5.1.2 Integration Testing

Palmer, in his book *Feature-Driven Development*, explained that a feature consist of several units performing different functions to reach one goal [56]. In this research, each wrangling operation was equivalent to a feature[2].

In the perspective of this research, the author proposed two integration testing levels. Furthermore, the scope for each integration testing level was defined. Level One tested one feature, i.e Taverna wrangling interaction, which consisted of parameter encoding, remote procedure call, and response decoding. It comprised reusable interactions implemented in Sprint Two and Taverna Components from Sprint Three.

[2]Palmer [56] gave an example in the form of activity diagram. One function call from the client-side was defined as one feature. Our activity diagrams, shown by Figures 4.4, B.1, B.2, and B.3, were consistent to their definition.

Integration testing Level Two tested data wrangling tasks implemented in Sprint One and Two and finally tested the improved wrangling tasks implemented in Sprint Three. In terms of proportion the integration testing contrasted to unit testing: integration tests were equally performed in all Sprints.

The integration test results are presented in Table 5.2. The test results showed that each Taverna interaction, both as a workflow and as a component, and the wrangling tasks have produced the expected results.

Table 5.2: Integration Testing Summary

No.	Testing Component	Description	Result
1	**Phase One**		
1.1	OpenRefine		
1.1.1	Import JSON file procedure	Procedurally import JSON file	Passed
1.1.2	Call wrangling functions	Called remove columns, rename columns, fill down columns, and create new columns	Passed
1.1.3	Export a project into CSV	Export file procedures	Passed
1.2	R		
1.2.1	Read data	Read a CSV file from remote URL and extract data session ID	Passed
1.2.2	Create column	Create column using function: mutate and retrieve data session ID	Passed
1.2.3	Select column	Select columns using function: select and retrieve data session ID	Passed
1.2.4	Filter data	Filter data using function: filter and retrieve data session ID	Passed
1.2.5	Group data	Create data groups by column values and retrieve data session ID	Passed
1.2.6	Aggregate data	Aggregate a previously grouped data and retrieve its data session ID	Passed
1.2.7	Generate chart	Generate chart and retrieve both data session ID and image file	Passed
1.3	Python		

Table 5.2: Integration Testing Summary

No.	Testing Component	Description	Result
1.3.1	Enrich date	Encode parameters, enrich date, and retrieve data URL	Passed
1.3.2	Time and space join	Encode parameters, call time and space join, and retrieve data URL	Passed
1.4	Data Wrangling Task		
1.4.1	DWT_1	Perform testing on data wrangling task DWT_1	Passed
2	**Phase Two**		
2.1	Create Reusable Workflows		
2.1.1	R: Read from URL	Reading reusable read-from-URL workflow	Passed
2.1.2	Python: enrich date	Create small workflow that calls enrich date function	Passed
2.1.3	R: create columns	Test a reusable create-columns workflow	Passed
2.1.4	R: select columns	Test a reusable select-columns workflow	Passed
2.1.5	R: filter columns	Test a reusable filter-columns workflow	Passed
2.1.6	R: group by and aggregate columns	Test a reusable data grouping and aggregation workflow	Passed
2.2	Data Wrangling Task		
2.2.1	DWT_2	Perform testing on data wrangling task DWT_2 which was built using reusable workflows	Passed
2.2.2	DWT_3	Perform testing on data wrangling task DWT_3 which was built using reusable workflows	Passed
2.2.3	DWT_4	Perform testing on data wrangling task DWT_4 which was built using reusable workflows	Passed
3	**Phase Three**		
3.1	Create components		

Table 5.2: Integration Testing Summary

No.	Testing Component	Description	Result
3.1.1	Create components for R functions	Wrapped small workflows of R functions into Taverna components	Passed
3.1.2	Create components for Python functions	Wrapped small workflows of Python functions into Taverna components	Passed
3.1.3	Create components for OpenRefine functions	Wrapped OpenRefine functions into smaller components and wrap their functions into Taverna components	Passed
3.2	Improvement of Wrangling Tasks		
3.2.1	Improvement of DWT_1	Improved DWT_1 by replacing services with components	Passed
3.2.2	Improvement of DWT_2	Improved DWT_2 by replacing services with components	Passed
3.2.3	Improvement of DWT_3	Improved DWT_3 by replacing services with components	Passed
3.2.4	Improvement of DWT_4	Improved DWT_4 by replacing services with components	Passed

5.2 Evaluation

Software architecture evaluation was performed to assess the quality of the selected design approach [57]. The aim of this evaluation was not to produce a verdict but to show the strengths and threats that an architecture posses [58]. To evaluate the selected design, scenario based evaluation strategy was chosen. It was adapted from Scenario-Based Analysis of Software Architecture (SAAM), which was introduced by Kazman et al in their paper in 1994 [59]. Scenario-based approach was chosen due because the data wrangling tasks that we used were equivalent to evaluation scenarios. In contrast to the early evaluation approach in SAAM, the architecture evaluation for this research was performed after the implementation phase because we did not aim to compare and contrast between architectural choices.

Two aspects of the architecture were evaluated: the first evaluation focused on the

client side whereas the second was conducted to assess the network load. As was described in section 3.3 and implemented in Chapter 4, the architecture enabled the mashup user to design a data wrangling workflow on the clientside and run the task by calling wrangling operations as web services. The evaluations were performed in the same environment as the implementation phase, which is described by Table 4.1.

5.2.1 Client Side Computational Evaluation

Client side computational evaluation was performed to measure the level of robustness of Taverna Workbench. The use of processor and memory were measured.

Default Taverna Workbench configurations were used. Provenance data capture option, which allows intermediate results, was enabled. Taverna Workbench provided an option to store information of a workbench session[3] in the main memory. If this option was selected, data between sessions would not be stored. Otherwise, data would be stored in the secondary memory and will be available between sessions. The default was to store in the main memory.

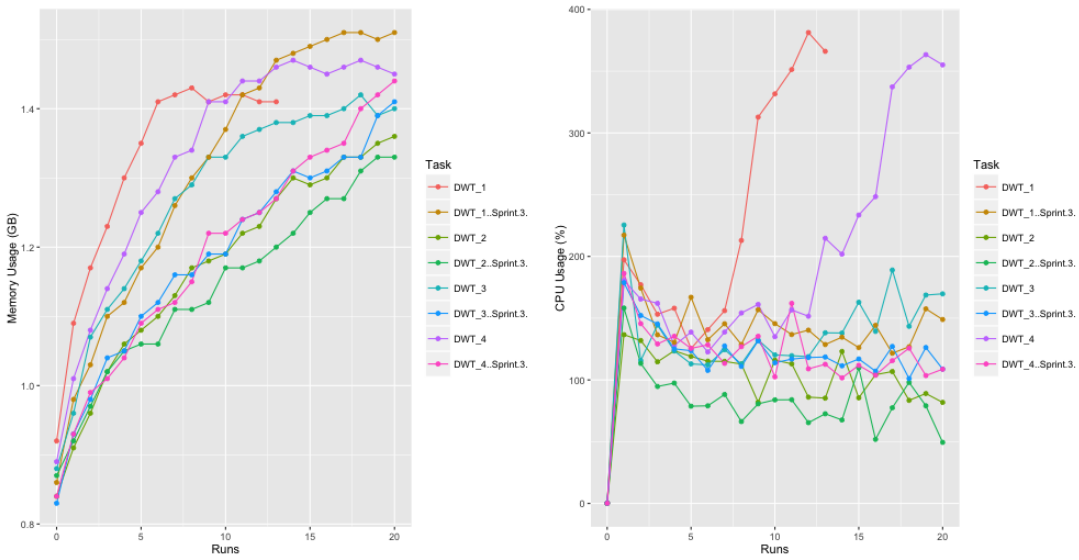
The evaluation was performed using Activity Monitor [4]. Changes in main memory consumption and use of processor were observed and noted. Changes in both measurements occurred several times during each run. At each run, the peak memory usage and processor power were taken into note and lower numbers were ignored. Each wrangling task was performed up to 20 times in a single workbench session. The repetition number was chosen arbitrarily. There were 510 seconds interlude between each iteration. Each repetition for each wrangling task was executed three times. The average measurements of each iteration from the three executions were then calculated for the evaluation.

CPU usage was monitored in percentage unit. The machine that was used during evaluation was equipped with dual core processor, with each processor capable of multi-threading, i.e. running dual processes in a single core. Therefore, when a processor was fully occupied the usage would show 200% usage.

The Taverna workflow file size of DWT_1 was 1.1MB whilst the other three tasks were less than 1MB. The file size of DWT_2, DWT_3, and DWT_4 before implementation Sprint Three were 186KB, 325KB, and 511KB respectively. After implemented using Taverna Components, the workflow file size of all wrangling tasks were reduced to

[3]One workbench session represented one launch of Taverna Workbench. When the application was closed, the session stopped.

[4]The default application in the OS X platform for monitoring machine performances.



(a) Memory Usage (GigaBytes) (b) CPU Usage (%)

Figure 5.1: Taverna Workbench Performance

296KB, 75KB, 110KB, and 114KB for DWT_1, DWT_2, DWT_3, and DWT_4 respectively.

The results of this evaluation is illustrated by Figure 5.1. The results showed that main memory consumed by Taverna Workbench was at the range of 800MB~1.51GB. It was lowest when no workflow had been run, i.e. 0 run. Memory usage climbed steadily as the number of runs increased. Notable feature was observed in the use of processor time, as shown by Figure 5.1b. At the first run, CPU usage steeply increased. Afterwards, it was observed that CPU usage gradually decreased until run 5. More importantly, It showed that as the iteration progressed, the use of processor drastically increased especially when running a large Taverna workflow file, i.e. DWT_1. After 12-13 runs of DWT_1 and 19-20 runs of DWT_4 Taverna workflow implementations before Sprint Three, Taverna Workbench crashed. The application became irresponsive was when CPU usage exceeded 300% before eventually crashed after several further iterations. The crash was caused by Java garbage collector overload. For such cases, further iterations were impossible to be executed.

5.2.2 Network Load

There are two types of network traffic monitoring: online and offline [60]. Online method was commonly used to monitor operational system in real-time. Tools supporting such method are equipped with graphical instruments. Offline network monitoring tools allowed network data collection and store them for later analysis. Hence the name. In this research project, the latter approach was used to evaluate the architecture. Wireshark[5] was used to monitor network packets transport.

There were two network evaluations performed in this research. The first network evaluation was performed to calculate the number of data sent from and to the client, whilst the second was performed to measure the traffic load each server had to withstand. The two network evaluations included packets sent via Transmission Control Protocol (TCP) and HTTP because data packets were transported in both layers.

It has been discussed in the implementation phase in Chapter 4 that the servers were run on a single machine. To monitor network traffic from and to local servers, loopback capture feature from Wireshark was used.

The evaluation was performed by running Taverna workflow implementation of data wrangling tasks DWT_1 through DWT_4, both implemented with and without the Taverna Components[6]. Each workflow was run once. The start and end time of each workflow run were taken into note, which were used to filter the packets. It is important to note that Taverna Workbench rounded start and end time up to the level of seconds. Any events occurring within milliseconds were floored to the nearest second. Hence, the end time filter for Wireshark was incremented by one second.

The number of bytes sent and received by the client side is shown by Table 5.3. The total bytes received from and sent to the servers per task are summarised at the right end of the table. There was no data packets sent to and received from OpenRefine and the downloader service during the run of Taverna workflows for DWT_2, DWT_3, and DWT_4 as the tasks did not involve the use of OpenRefine.

The dataset produced in DWT_2 reached 25MB whilst the output dataset of DWT_4 was 1MB. As a reminder, tasks DWT_2 did not involve data aggregation, although data filtration was performed in DWT_4. Consequently, the traffic from and to the client side for DWT_2 and DWT_4 in total were relatively high compared to the other two tasks.

[5] `https://www.wireshark.org`. Wireshark is a tool for capturing network activities. It allowed network packet analysis. The tool allowed data collection and dumping, which supports the offline network monitoring method.

[6] As implemented in Sprint Three, which was discussed on section 4.6.

Table 5.3: Network traffic sent and received by the client-side (in Mega Bytes)

Data Wrangling Task	Sent To			Received From			Total
	OpenRefine**	R	Python	OpenRefine**	R	Python	
DWT_1	0.01	0.01	0.00	0.04	0.16	0.00	0.23
DWT_1*	0.01	0.01	0.00	0.03	0.16	0.00	0.23
DWT_2	-	0.04	0.00	-	25.11	0.00	25.14
DWT_2*	-	0.04	0.00	-	25.45	0.00	25.49
DWT_3	-	0.01	0.00	-	0.10	0.00	0.11
DWT_3*	-	0.01	0.00	-	0.10	0.00	0.11
DWT_4	-	0.01	0.00	-	1.27	0.00	1.28
DWT_4*	-	0.01	0.00	-	1.27	0.00	1.28

* Taverna Workflows after implementation Sprint Three
** Including the data sent to OpenRefine downloader service

DWT_1 and DWT_3 produced small datasets and PNG images. The output images produced by R were relatively small in size. Images generated by DWT_1, DWT_3, and DWT_4 were 16KB, 83KB , and 24KB respectively.

The number of bytes sent by the client were insignificant compared to the number of bytes received. DWT_1, for example, sent a total 0.02 MB of data to the data wrangling services. This was because data sent to the wrangling services contained only the wrangling instructions whilst the data resided on the server side throughout the task. Data sent to the R server during the run of DWT_2 was comparatively higher than the other tasks. The higher bytes sent was due to the higher number of data received: a large data was divided into smaller data packets when sent through the network; the larger number of bytes received meant the client had to send more acknowledgement responses to the server to indicate that a data packet was successfully received.

The results for the second network evaluation is presented in Table 5.4. Traffic variations between wrangling task implementation before and after Sprint Three were caused by packets failure or duplicates. From the table it could be inferred that the Python server, although consist of only two services it generated the most network traffic; Python server alone generated 57% of the total traffic. Similar, if not identical,

Table 5.4: Network traffic sent and received by each server (in Mega Bytes)

Server	Received Bytes [Sent Bytes]								Total Per Server
	DWT_1	DWT_1*	DWT_2	DWT_2*	DWT_3	DWT_3*	DWT_4	DWT_4*	
OpenRefine	2.10	2.31	N/A	N/A	N/A	N/A	N/A	N/A	4.42
	[0.96]	[0.73]	[N/A]	[N/A]	[N/A]	[N/A]	[N/A]	[N/A]	[1.69]
R	30.22	30.16	29.86	29.86	29.91	29.91	29.84	29.84	239.62
	[0.16]	[0.16]	[25.11]	[25.45]	[0.10]	[0.10]	[1.27]	[1.27]	[53.62]
Python	20.95	16.54	21.34	21.34	21.75	18.17	20.54	16.89	157.52
	[30.21]	[30.15]	[29.83]	[29.83]	[29.83]	[29.83]	[29.83]	[29.83]	[239.34]
Total Per Task	53.28	49.01	51.21	51.21	51.66	48.08	50.38	46.73	401.56
	[31.34]	[31.04]	[54.94]	[55.28]	[29.93]	[29.93]	[31.10]	[31.10]	[294.65]

* Taverna Workflows after implementation Sprint Three

network traffic was observed sent by Python web service during runs of tasks DWT_2-DWT_4 notably because the web service was called once in each task to enrich traffic data date. The second busiest was the R server. Unlike the Python server, which traffic was equally busy receiving and sending data, the R server was mainly high in its incoming traffic. Data received by R server built up to 82% of the total network traffic it generated. It was discussed in Section 3.2 that the majority of data wrangling operations were performed in R. Moreover, wrangling tasks performed in the R server involved data reduce operations, i.e. filter and/or aggregation. Due to the two reasons, data sent by R server was not as high as it has received.

It was also inferred that a JSON file is greater in size relative to CSV. This could be observed by comparing the data received by OpenRefine and the size of data sent. Important to note that no data reduce operation was performed using OpenRefine. A JSON file structure consist of key and value definition. Such structure would yield greater file size compared to a tabular data format, such as a CSV file. Thus, it was expected that OpenRefine server would receive greater traffic than it sends.

To conclude, the architecture evaluations had shown that data movements were intensive at the server side whilst the client-side, i.e. the User Interface Layer, sent an insignificant amount of data.

5.3 Summary

This chapter has discussed the approach taken to measure the results of this research. Unit and integration testing were performed iteratively to assure the produced artefact

behaved as expected. Two types of evaluations were performed to measure the performance of the selected application for the User Interface Layer and to measure the network load. The earlier evaluation showed that Taverna Workbench was optimum when running smaller workflows. From the latter evaluation, it could be inferred that the network traffic load of the mashup was not imposed to the User Interface Layer, i.e the client side.

Chapter 6

Conclusions and Future Works

This chapter summarises the research conducted in this dissertation. The author furthermore attempted to draw a conclusion. Finally, recommendations for future works are presented.

6.1 Conclusions

The aim of this project was to implement and evaluate a web mashup of data wrangling tools for traffic data. To achieve this, objectives were defined at the beginning of the research, which became milestones as the project progressed. Literatures were visited and experiments were carried out to create a foundation for the project. The background study revolved around the concept of data wrangling, web mashups, and the challenges present in processing big data. It was found that there are tools from which data wrangling operations could be mashed. Conversely, there were also constrains which limited the extent of this research. The architecture of Data Wrangler, for example, did not allow a web API to be built. The software solution was designed carefully to meet the aim of this research. It included a set of use cases, from which requirements were extracted, and a high-level architectural design. With regards to the web mashups concept, wrangling tools were wrapped as web API. To access the operations provided by each API, Taverna Workbench was used. Based on this design, the implementation was successfully carried out. It was supported by the software testing results.

Different wrangling tools offered their respective advantages over the other. OpenRefine, for example, had a superior functionality compared to R in parsing a JSON file. Using the concept of web mashups, this research had successfully combined data

wrangling operations from a selection of available tools. This was possible by exposing each wrangling tool as web API. Data wrangling operations were performed on the server side, which helped the operator to reduce workstation and network requirement at the client side. This was proven by the results of the evaluations. More importantly, it showed that network traffic to the client side was insignificantly low, which contrasted the statement of Wolfstencroft [46] that a workflow should be executed on the client side to reduce network traffic. Evidently, although it was not a novel approach, web mashups was proven to be a solution to the issues in preparing traffic data for analysis. Moreover, our selection of the client side application was able to be installed on multiple platforms. Therefore, it is safe to declare that our approach was platform-independent.

There were, however, some domain-specific requirements which could not be resolved using data wrangling operations offered by any existing tools. These requirements were implemented using Python.

Finally, iterative and incremental development was proven to be beneficial to the implementation of the mashup. The first implementation of the wrangling tasks, although effective, was not efficient. The selected client-side application, Taverna Workbench, was not very powerful to handle large data wrangling task workflows. Sprint Three has helped reducing the complexity of the workflows and, thus, diminished processor time required by the client-side.

6.2 Future Works

This research, of course, was subject to some limitations. Time, for instance, was a finite resource that bounded the achievements of this project. For future researches, the author recommended the following.

- Data wrangling tools reviewed in this research were open source and free. Therefore, the API of Trifacta Wrangler was out of scope. Future researches should include the tool because it offered exclusive data wrangling features.

- The types of dataset used in this research were limited. In total, there were three datasets involved in solving the use cases. A future research could incorporate more data source variations.

- Due to the scope of the project, the implementation of space and time join function was not optimised. With a time complexity of $\Theta(n^2)$, it would not perform

efficiently on large datasets. This function could be improved in the future.

- Taverna Workbench version 2.5.0 used in this research was a product of Taverna before it was moved into the incubation program of Apache. It was expected that the incubation would produce a more resource friendly software. Therefore, future researches should use the next release of Taverna Workbench.

- Features offered by Taverna Workbench complied with the auditability, reproducibility, and collaboration aspects of data wrangling. The extent of this research, however, did not cover the evaluation of such aspects. It is recommended that future works would have these aspects assessed.

Bibliography

[1] R. Bolla and F. Davoli. Road traffic estimation from location tracking data in the mobile cellular network. *2000 IEEE Wireless Communications and Networking Conference. Conference Record (Cat. No.00TH8540)*.

[2] Parliamentary Office of Science and UK House of Parliament Technology (POST). *Big and Open Data in Transport*. Research Briefings. 2014.

[3] Rob Kitchin. The real-time city? big data and smart urbanism. *SSRN Electronic Journal SSRN Journal*, 79(1):1–14, February.

[4] Chenjuan Guo, Christian S. Jensen, and Bin Yang. Towards total traffic awareness. *ACM SIGMOD Record SIGMOD Rec.*, 43(3):18–23, Apr 2014.

[5] H. V. Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big data and its technical challenges. *Commun. ACM*, 57(7):86–94, July 2014.

[6] Kelvin K.w. Yau. Risk factors affecting the severity of single vehicle traffic accidents in hong kong. *Accident Analysis & Prevention*, 36(3):333–340, 2004.

[7] Jun Zhang, Joan Lindsay, Kathy Clarke, Glenn Robbins, and Yang Mao. Factors affecting the severity of motor vehicle traffic crashes involving elderly drivers in ontario. *Accident Analysis & Prevention*, 32(1):117–125, 2000.

[8] He-Sheng Zhang, Yi Zhang, Zhi-Heng Li, and Dong-Cheng Hu. Spatial-temporal traffic data analysis based on global data management using mas. *IEEE Transactions on Intelligent Transportation Systems*, 5(4):267–275, Dec 2004.

[9] Turner-Fairbank Highway Research Center. Traffic analysis toolbox, Jul 2004.

[10] IBM, Paul Zikopoulos, and Chris Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edition, 2011.

[11] Hrushikesha Mohanty. Big data: a primer. chapter 1, pages 1–28. Springer, 2015.

[12] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. Dec 2012.

[13] EMC. Data lakes for data science: Integrating analytics tools with shared infrastructure for big data, jul 2014.

[14] IBM. 2015 ibm annual report. online, 2016.

[15] Abhirup Chatterjee and Arie Segev. Data manipulation in heterogeneous databases. *ACM SIGMOD Record SIGMOD Rec.*, 20(4):64—-68, Jan 1991.

[16] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data mining with big data. *IEEE Trans. Knowl. Data Eng. IEEE Transactions on Knowledge and Data Engineering*, 26(1):97–107, 2014.

[17] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. Van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, P. Buono, and et al. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization*, 10(4):271—-288, Feb 2011.

[18] Wes McKinney. *Python for Data Analysis*. O'Reilly Media, 1st edition edition, 2012.

[19] David Cross. *Data Munging with Perl*. Manning Publications, 2001.

[20] Ben Shneiderman and Catherine Plaisant. Sharpening analytic focus to cope with big data volume and variety. *IEEE Comput. Grap. Appl. IEEE Computer Graphics and Applications*, 35(3):10–14, 2015.

[21] V. M. Megler, Kristin Tufte, and David Maier. Improving data quality in intelligent transportation systems. *CoRR*, abs/1602.03100, 2016.

[22] Bryant Duhon. Software as a service. *AIIM E - Doc Magazine*, 21(5):10, Sep 2007. Copyright - Copyright AIIM International Sep/Oct 2007; Last updated - 2011-09-26; SubjectsTermNotLitGenreText - United States–US.

[23] Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock. Building mashups by example. *Proceedings of the 13th international conference on Intelligent user interfaces - IUI '08*, 2008.

[24] Mario Andrés Paredes-Valverde, Giner Alor-Hernández, Alejandro Rodríguez-González, Rafael Valencia-García, and Enrique Jiménez-Domingo. A systematic review of tools, languages, and methodologies for mashup development. *Softw. Pract. Exper.*, 45(3):365–397, March 2015.

[25] Jamie Kurtz. *ASP.NET MVC 4 and the Web API: building a REST service from start to finish*. 1st edition, 2013.

[26] Phil Spector. *Data manipulation With R*. Springer Verlag, 2008.

[27] S. Voulgaropoulou, G. Spanos, and L. Angelis. Analyzing measurements of the r statistical open source software. In *Software Engineering Workshop (SEW), 2012 35th Annual IEEE*, pages 1–10, Oct 2012.

[28] RStudio Inc. Data wrangling cheat sheet.

[29] Jeroen Ooms. The opencpu system: Towards a universal interface for scientific computing through separation of concerns. *eprint arXiv:1406.4806*, 2014.

[30] Opencpu. `https://www.opencpu.org/api.html`. Accessed: 2016-06-22.

[31] Kelli Ham. Openrefine (version 2.5). http://openrefine.org. free, open-source tool for cleaning and transforming data. *J Med Libr Assoc Journal of the Medical Library Association : JMLA*, 101(3):233–234, 2013.

[32] Stefan Goessner. Jsonpath - xpath for json.

[33] Li Jun and Li Ling. Comparative research on python speed optimization strategies. In *Intelligent Computing and Integrated Systems (ICISS), 2010 International Conference on*, pages 57–59, Oct 2010.

[34] I. J. Cheng, P. H. Cheng, S. Y. Tsai, P. H. Chen, Y. J. Lai, F. C. Yang, and C. R. Rau. A comparison of python and java capabilities on somatosensory game for cloud-based rehabilitation system. In *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*, pages 660–661, Oct 2015.

[35] Jules Kouatchou. Comparing python, numpy, matlab, fortran, etc., Nov 2009.

[36] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34, Jan 2006.

[37] Jens Baek Jorgensen. Executable use cases: a supplement to model-driven development? *Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES'07)*, 2007.

[38] Veit Hoffmann, Horst Lichter, Alexander Nyßen, and Andreas Walter. Towards the integration of uml- and textual use case modeling. *JOT The Journal of Object Technology*, 8(3):85–100, 2009.

[39] J. B. Jorgensen and C. Bossen. Executable use cases: requirements for a pervasive health care system. *IEEE Software*, 21(2):34–41, March 2004.

[40] Data elements and interchange formats – information interchange – representation of dates and times, mar 2005.

[41] Alan Cline. *Agile development in the real world*. Apress, 2016.

[42] Barry Boehm, Jo Ann Lane, and Richard Turner. Architected agile solutions for software-reliant systems, 2010.

[43] E. Wohlstadter, P. Li, and B. Cannon. Web service mashup middleware with partitioning of xml pipelines. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 91–98, July 2009.

[44] Manfred Sneps-Sneppe and Dmitry Namiot. On physical web models. *CoRR*, abs/1602.00841, 2016.

[45] Valerie Issarny, Mauro Caporuscio, and Nikolaos Georgantas. A perspective on the future of middleware-based software engineering. *Future of Software Engineering (FOSE '07)*, 2007.

[46] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, and et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1), Feb 2013.

[47] L. Williams and A. Cockburn. Agile software development: it's about feedback and change. *Computer*, 36(6):39–43, June 2003.

[48] M. Yadav, N. Goyal, and J. Yadav. Agile methodology over iterative approach of software development -a review. In *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, pages 542–547, March 2015.

[49] Michael Ellims, James Bridges, and Darrel C. Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31, 2006.

[50] George Ellis. *Project Management in Product Development*. Elsevier, 2016.

[51] Vivek Sehgal, Lise Getoor, and Peter D Viechnicki. Entity resolution in geospatial data integration. *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems - GIS '06*, 2006.

[52] C. C. Robusto. The cosine-haversine formula. *The American Mathematical Monthly*, 64(1):38–40, 1957.

[53] C. Bron. Merge sort algorithm [m1]. *Commun. ACM*, 15(5):357–358, May 1972.

[54] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[55] H. K. Brar and P. J. Kaur. Differentiating integration testing and unit testing. In *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, pages 796–798, March 2015.

[56] Steve R Palmer and Mac Felsing. *A practical guide to feature-driven development*. Pearson Education, 2001.

[57] A. Patidar and U. Suman. A survey on software architecture evaluation methods. In *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, pages 967–972, March 2015.

[58] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating software architectures: methods and case studies*. Addison-Wesley, 2011.

[59] Rick Kazman, Len Bass, Mike Webb, and Gregory Abowd. Saam: A method for analyzing the properties of software architectures. In *Proceedings of the 16th*

International Conference on Software Engineering, ICSE '94, pages 81–90, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[60] C. Williamson. Internet traffic measurement. *IEEE Internet Computing*, 5(6):70–74, Nov 2001.

Appendix A

Data Wrangling Task Formalisations

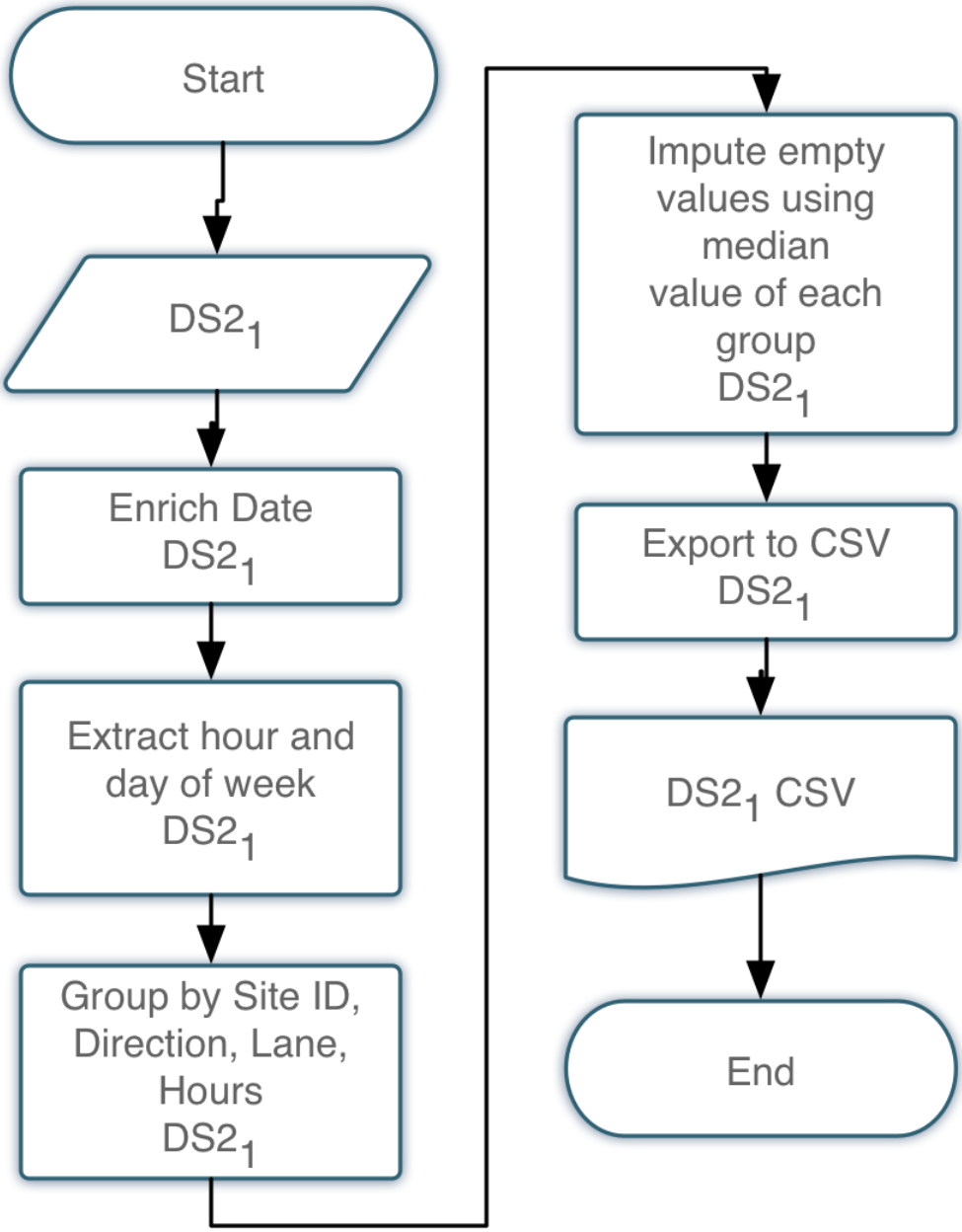


Figure A.1: Wrangling Task Formalisation for task DWT_2

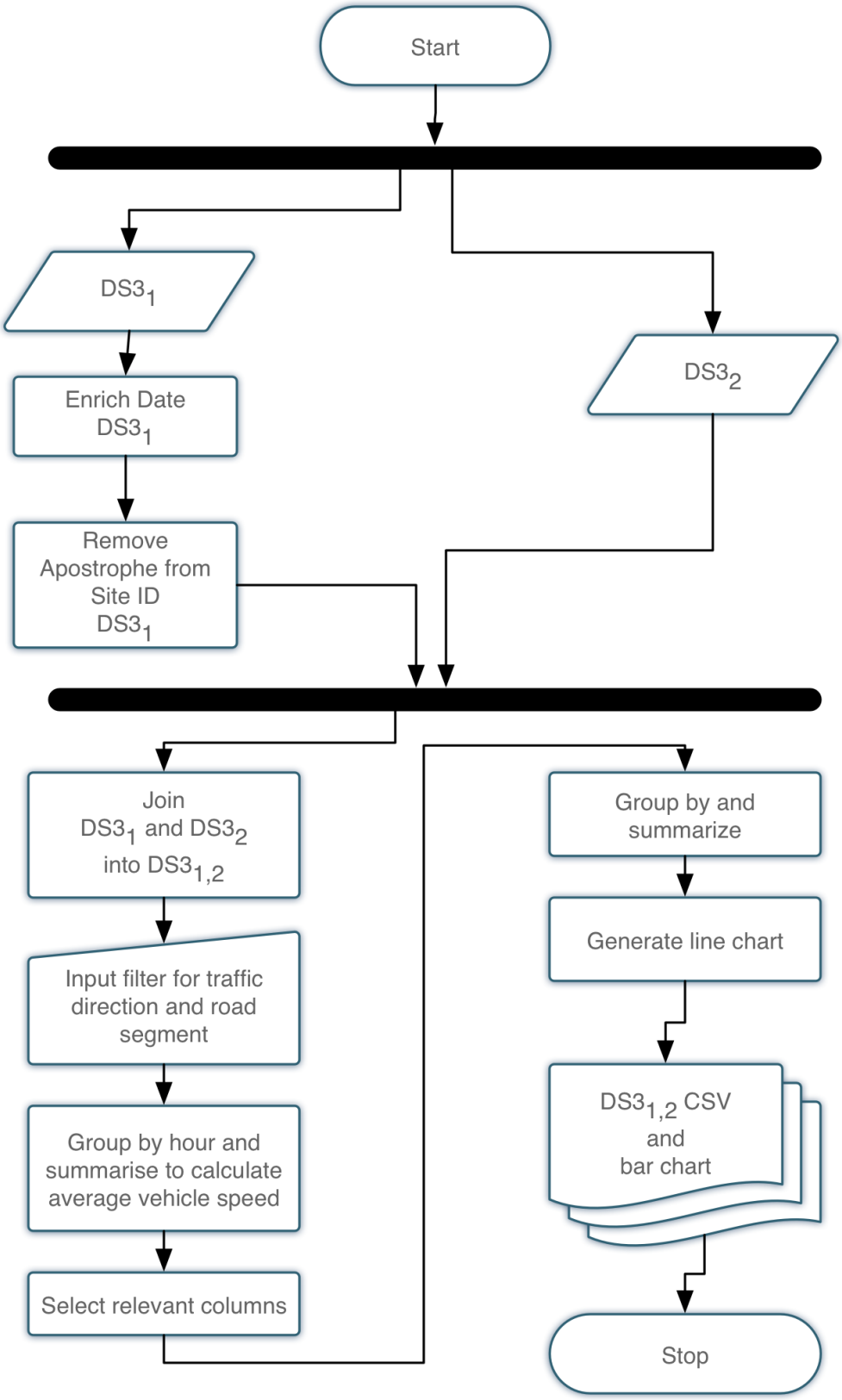


Figure A.2: Wrangling Task Formalisation for task DWT_3

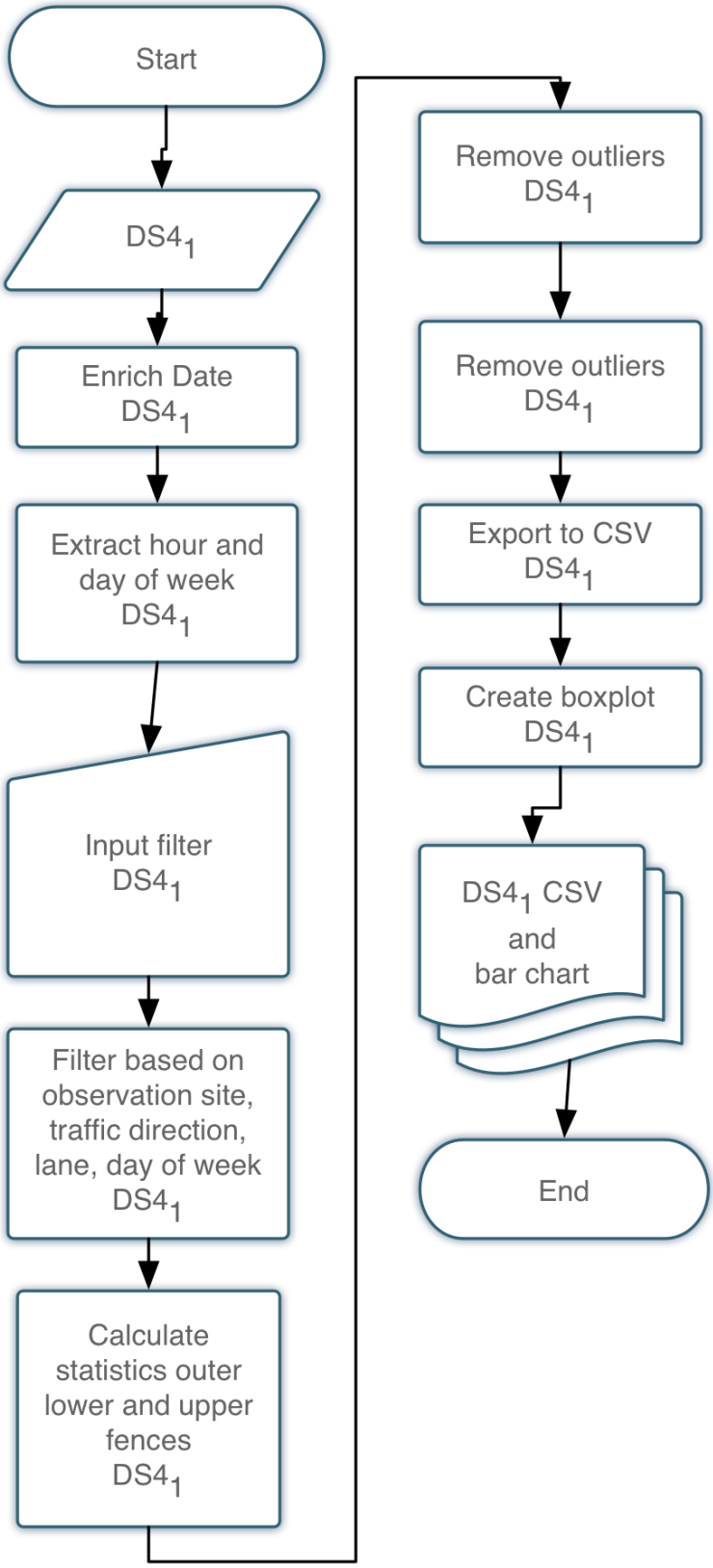


Figure A.3: Wrangling Task Formalisation for task DWT_4

Appendix B

Interactions Between Taverna and Data Wrangling Services

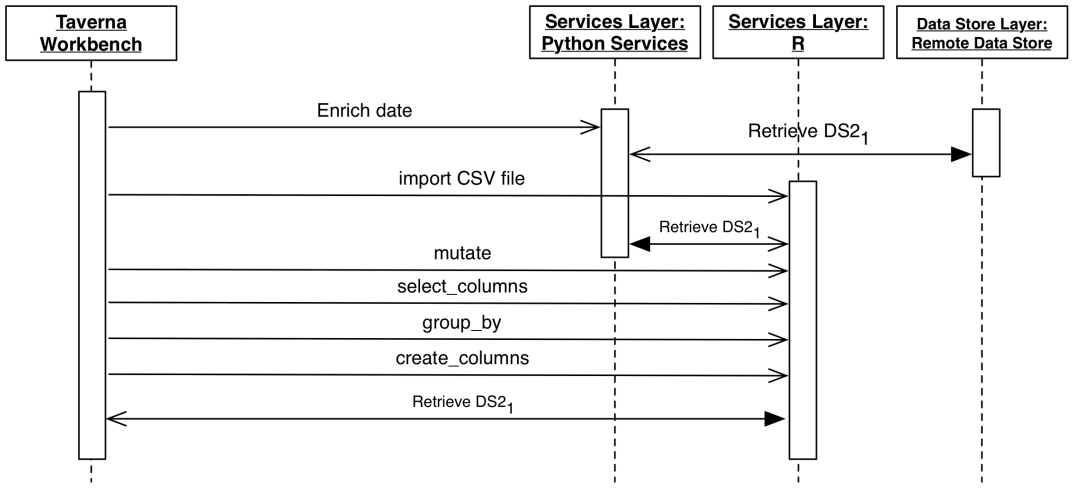


Figure B.1: Interaction between Taverna and wrangling services for task DWT_2

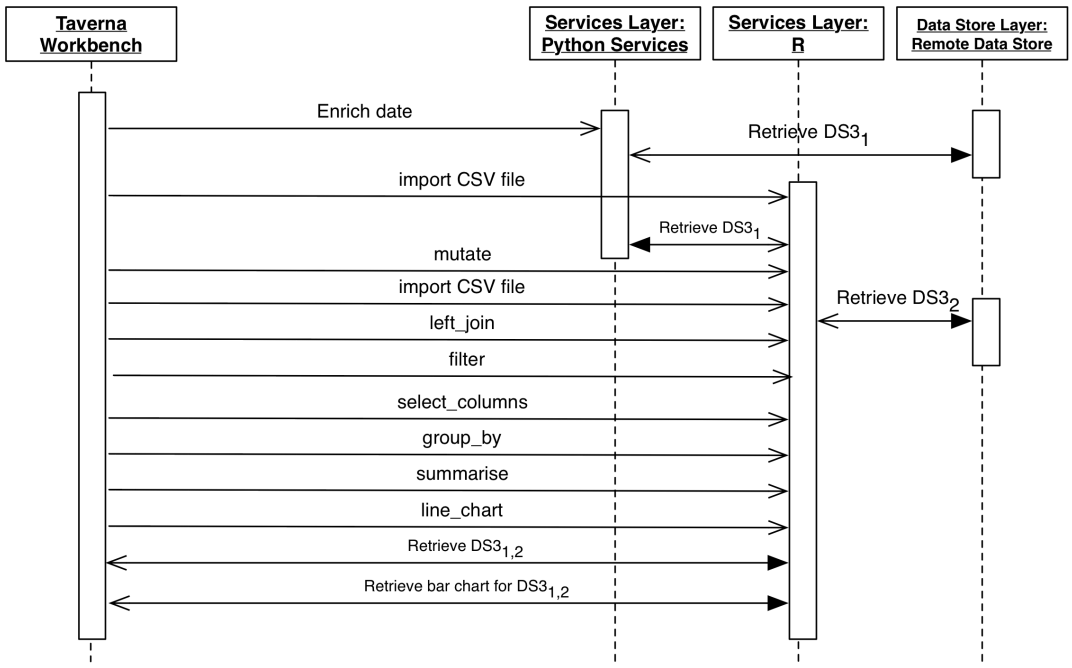


Figure B.2: Interaction between Taverna and wrangling services for task DWT_3

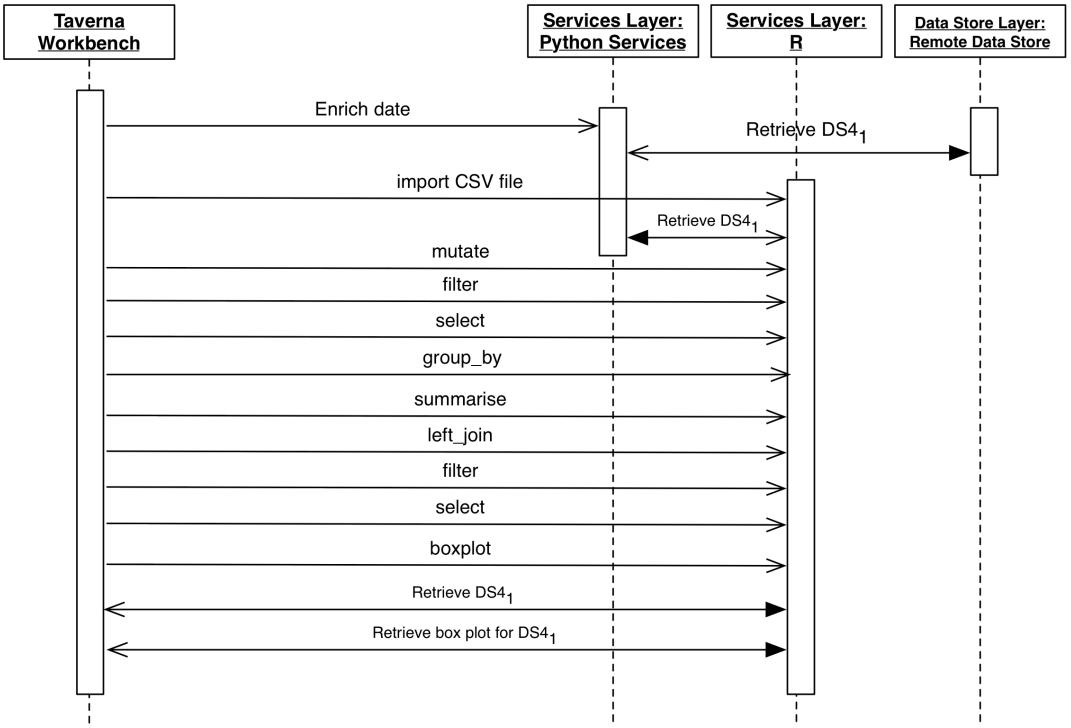


Figure B.3: Interaction between Taverna and wrangling services for task DWT_4

Appendix C

Data Wrangling Task Workflows

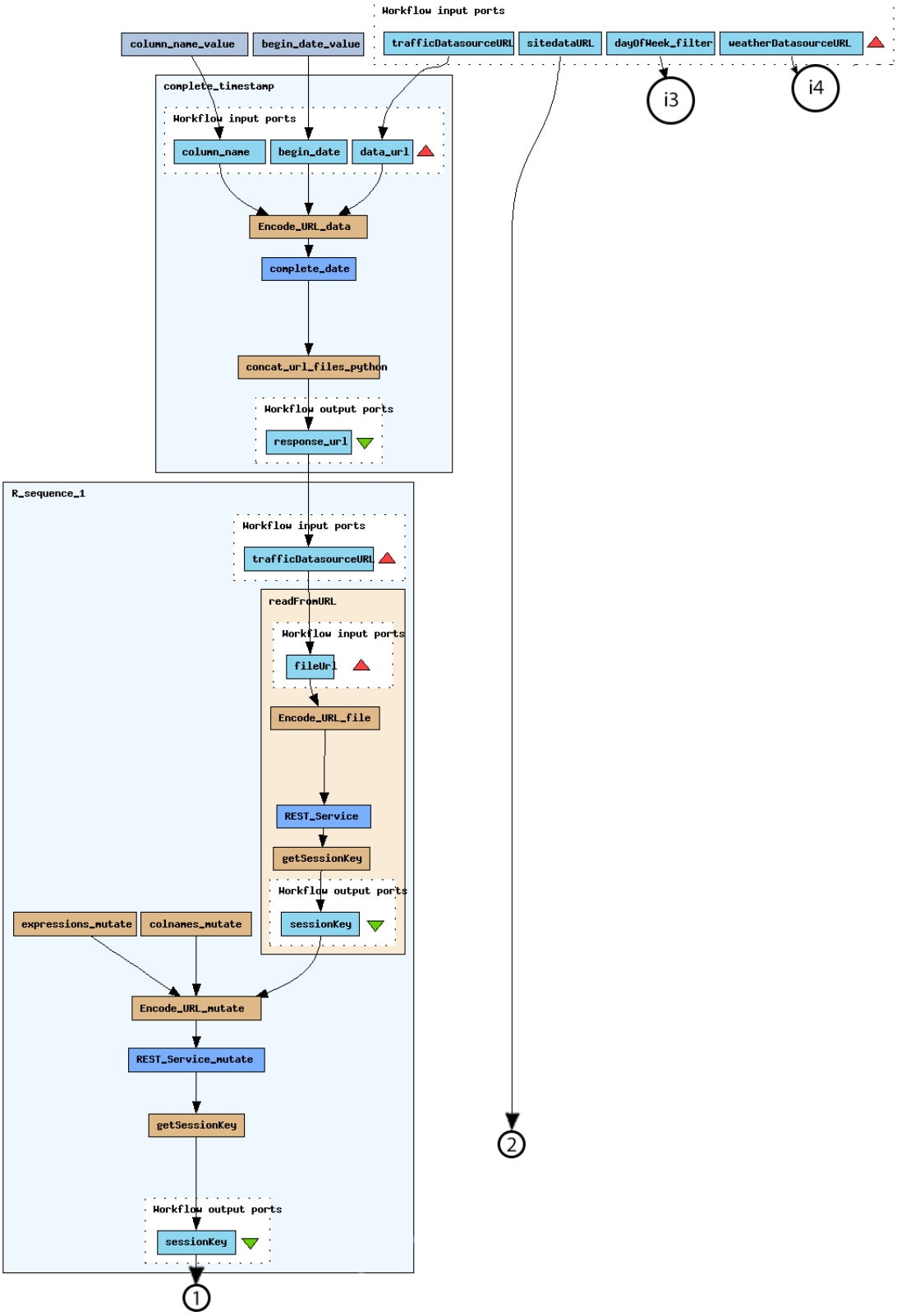


Figure C.1: Taverna workflow implementation for data wrangling task DWT_1 (1/7)

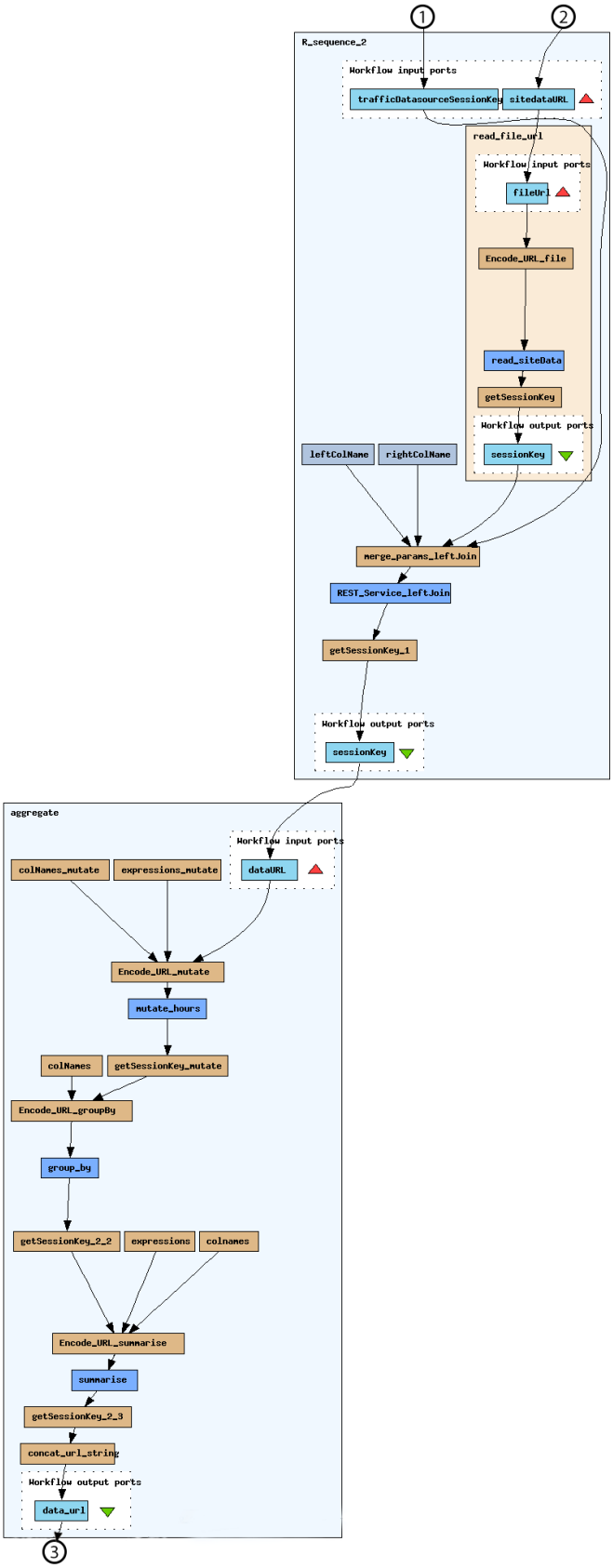


Figure C.2: Taverna workflow implementation for data wrangling task DWT_1 (2/7)

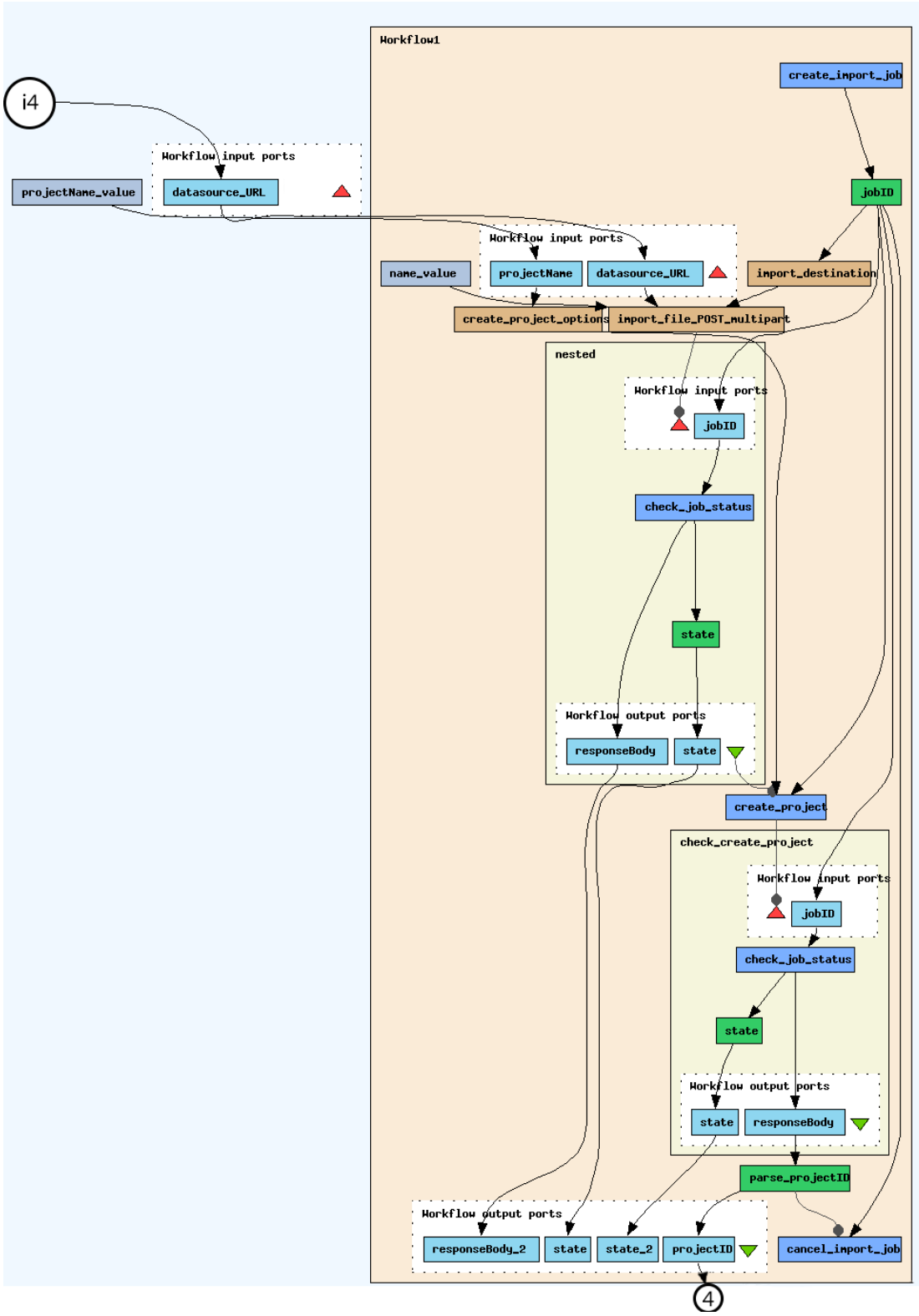


Figure C.3: Taverna workflow implementation for data wrangling task DWT_1 (3/7)

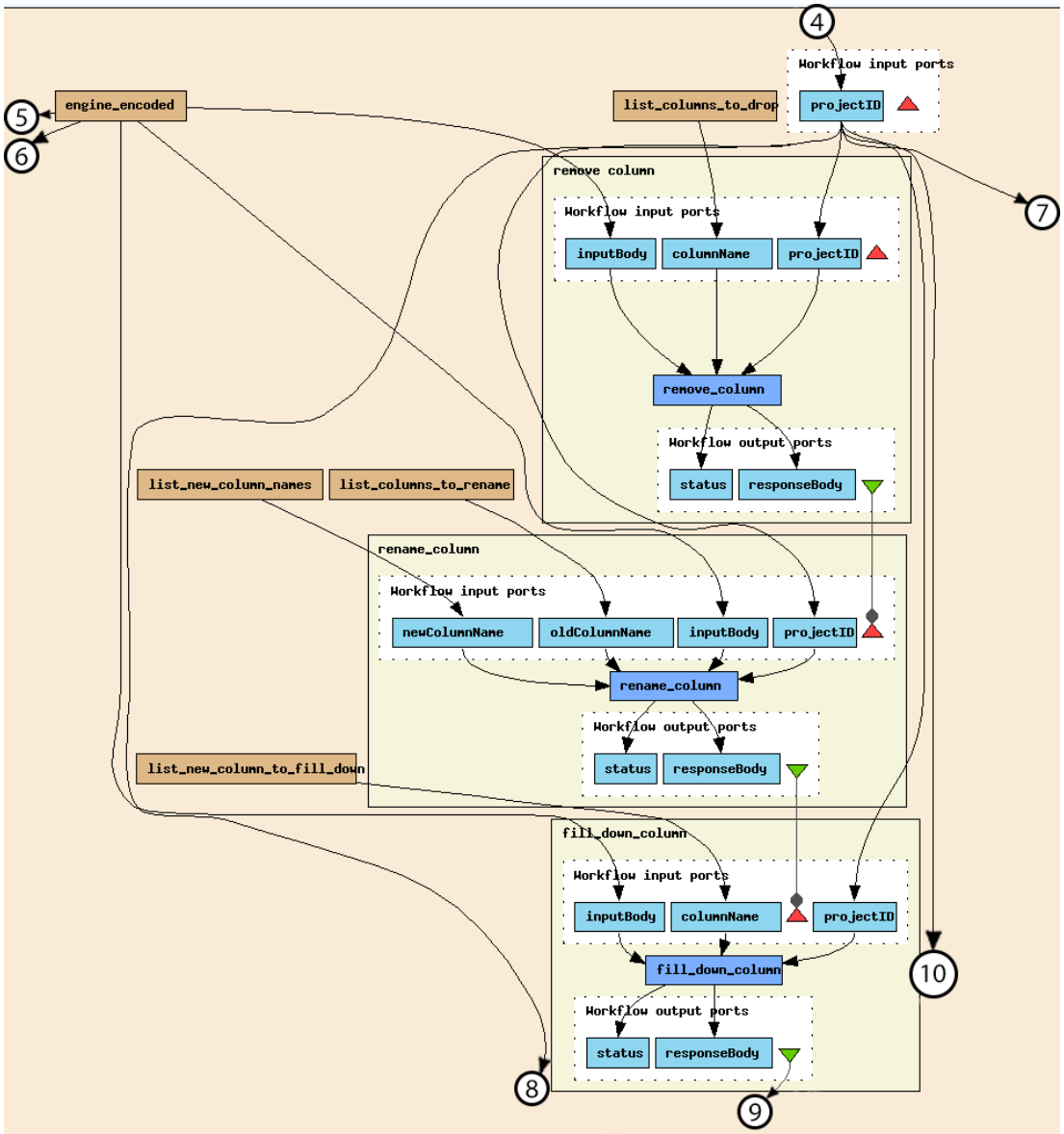


Figure C.4: Taverna workflow implementation for data wrangling task DWT_1 (4/7)

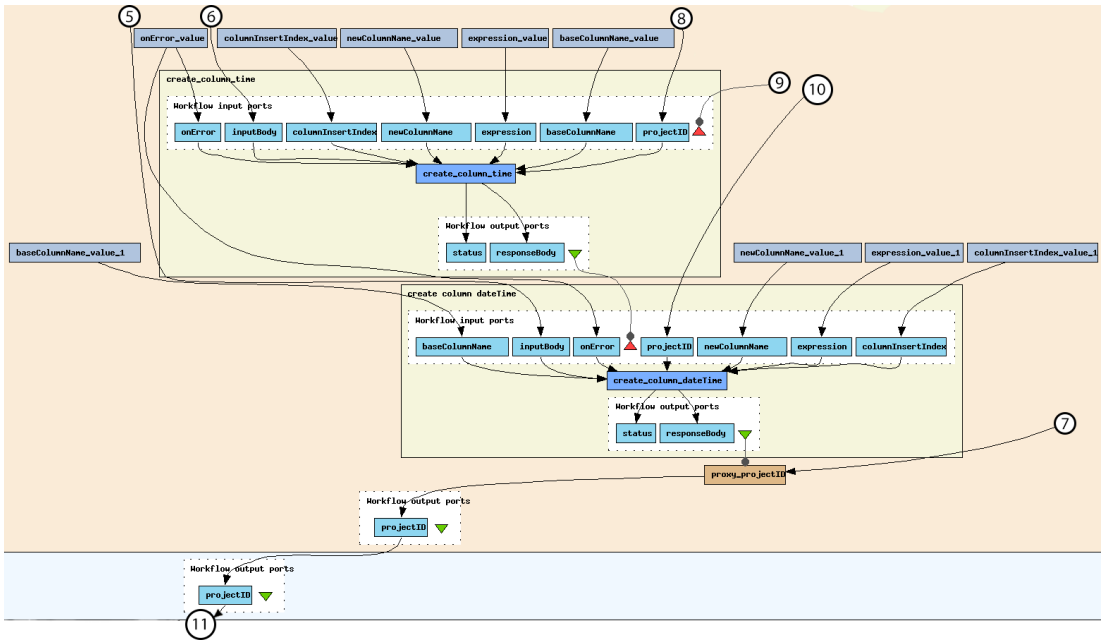


Figure C.5: Taverna workflow implementation for data wrangling task DWT_1 (5/7)

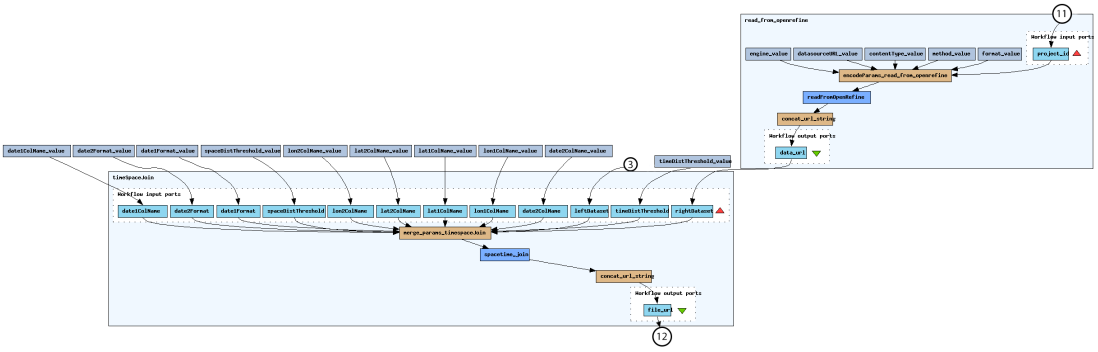


Figure C.6: Taverna workflow implementation for data wrangling task DWT_1 (6/7)

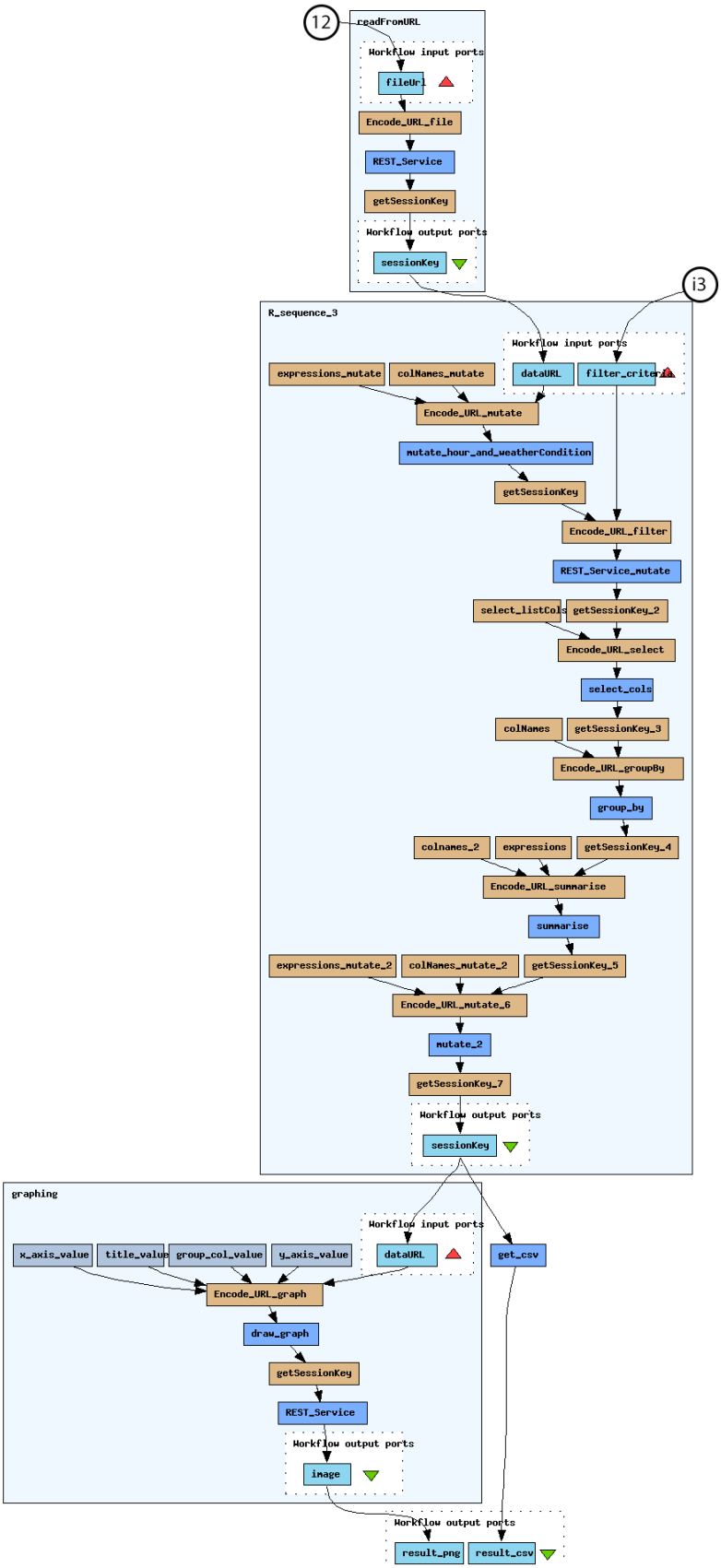


Figure C.7: Taverna workflow implementation for data wrangling task DWT_1 (7/7)

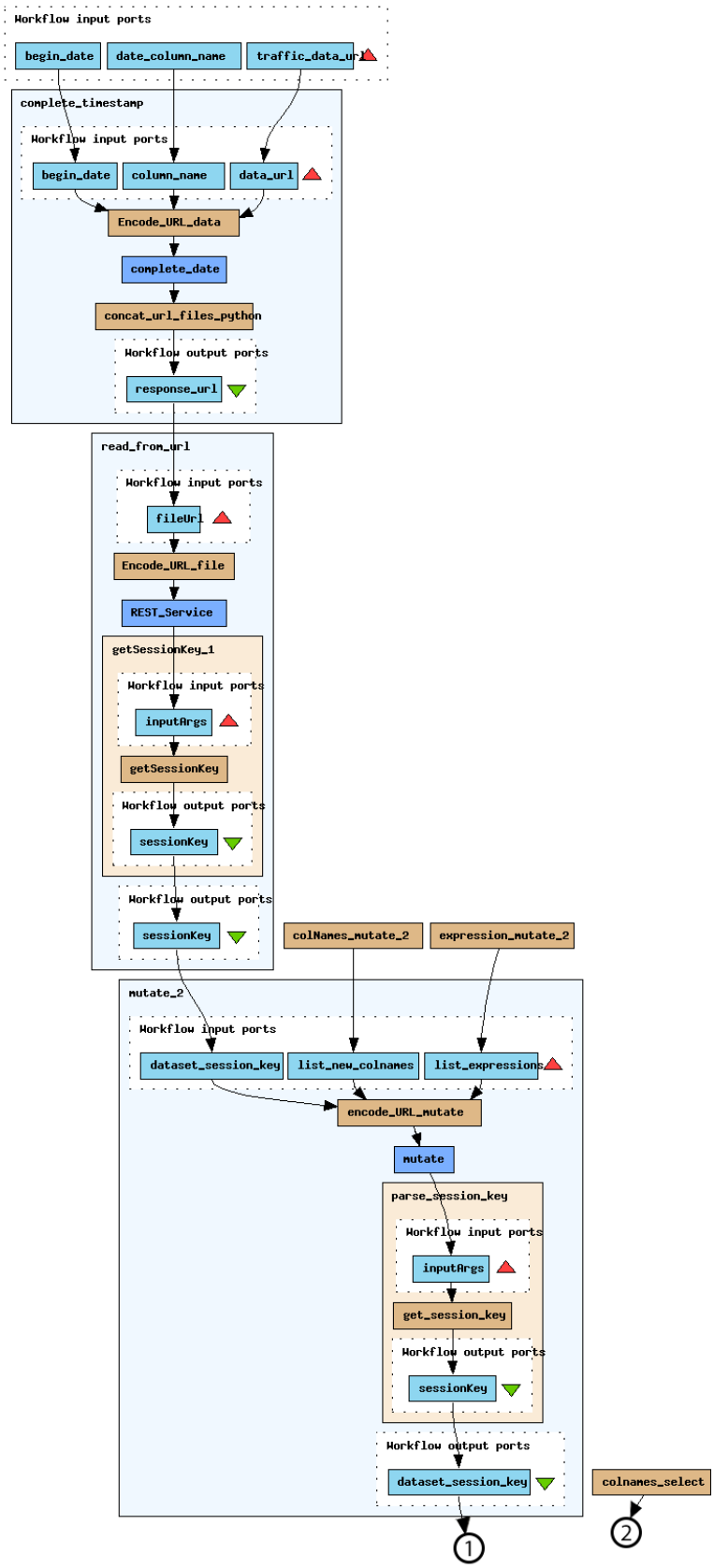


Figure C.8: Taverna workflow implementation for data wrangling task DWT_2 (1/2)

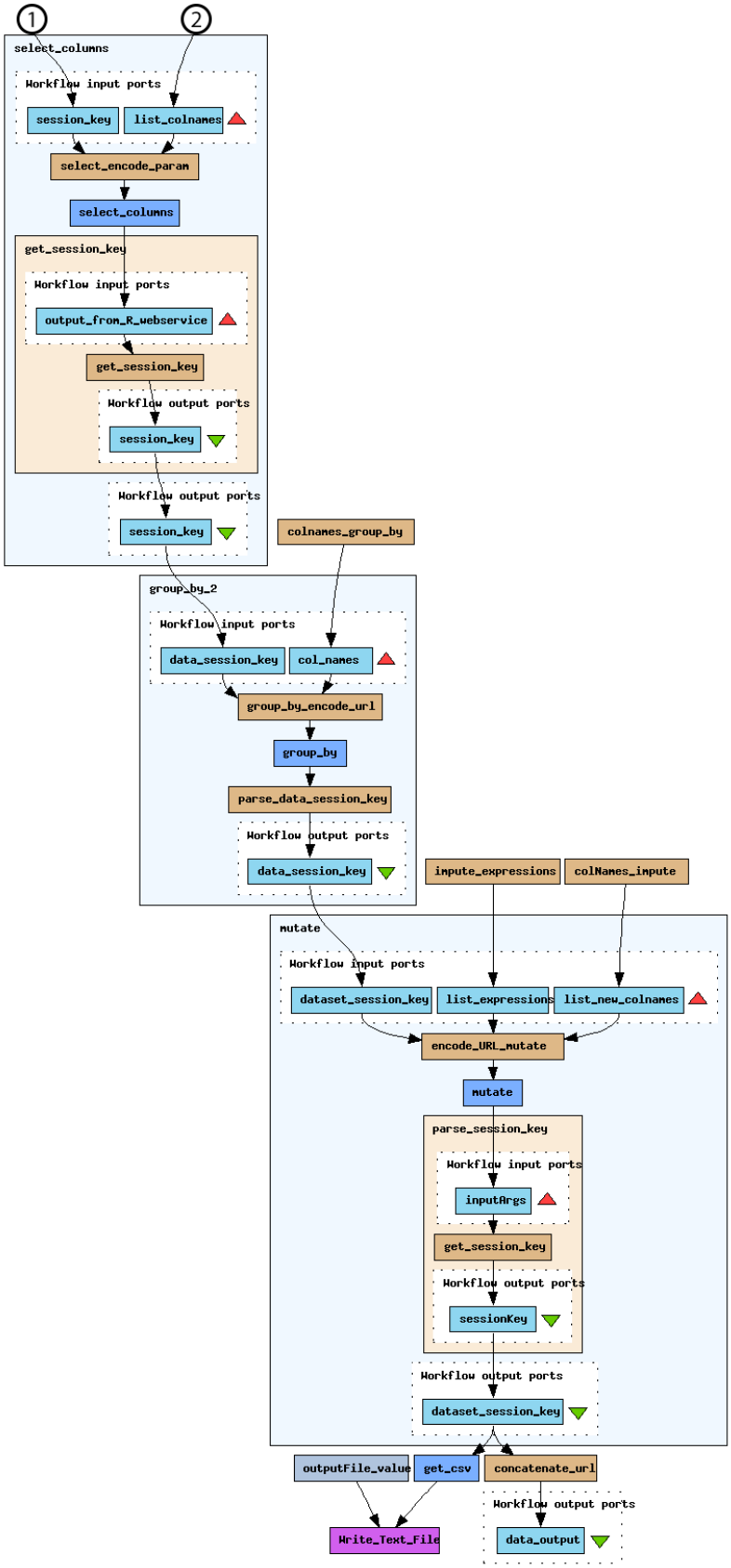


Figure C.9: Taverna workflow implementation for data wrangling task DWT_2 (2/2)

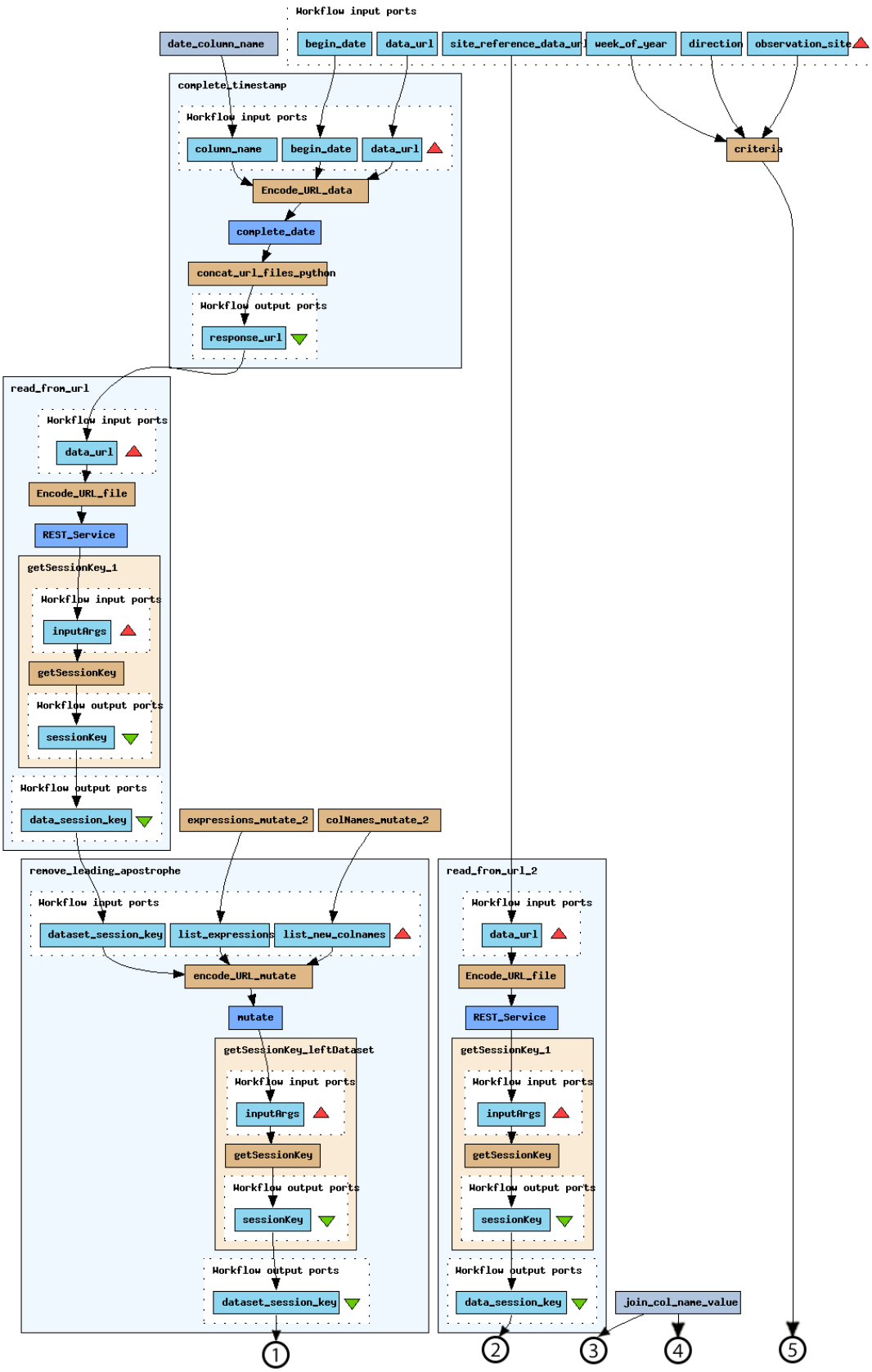


Figure C.10: Taverna workflow implementation for data wrangling task DWT_3 (1/3)

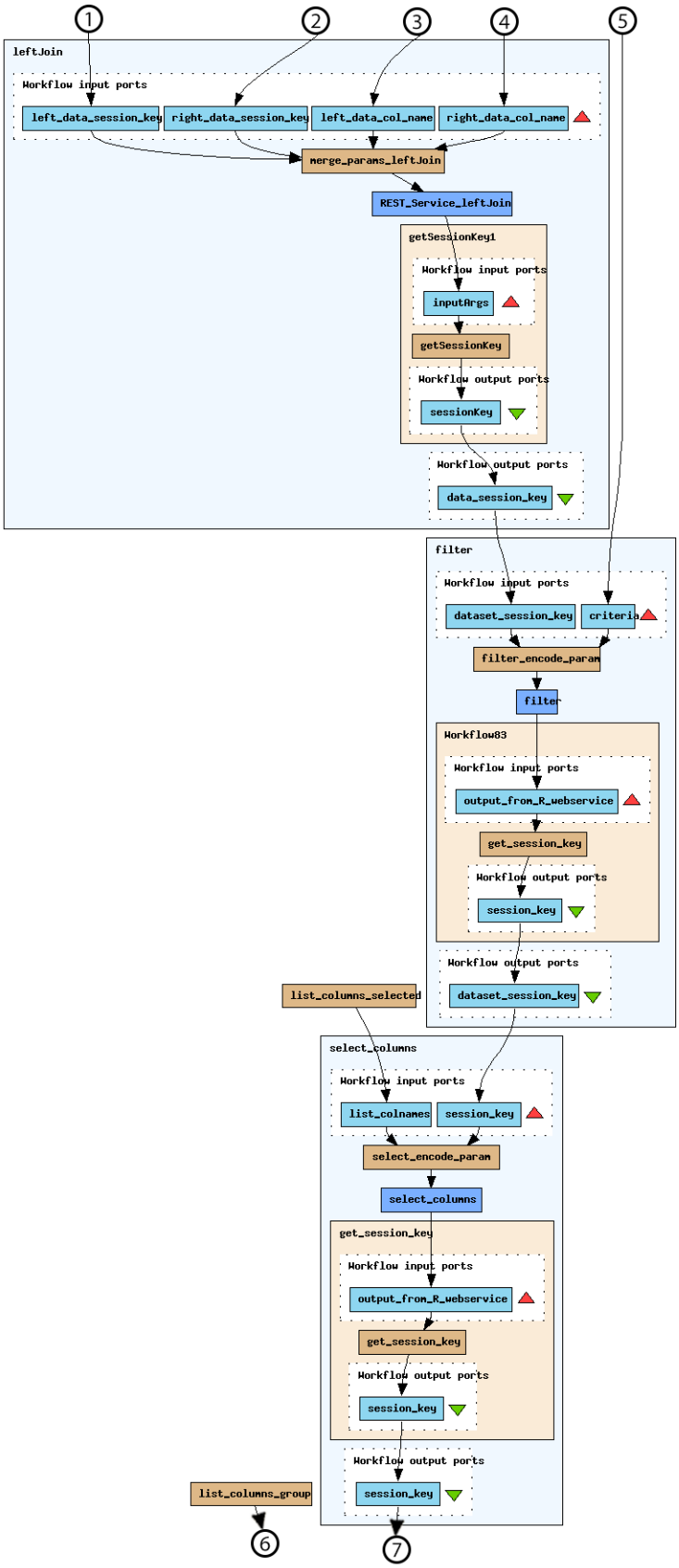


Figure C.11: Taverna workflow implementation for data wrangling task DWT_3 (2/3)

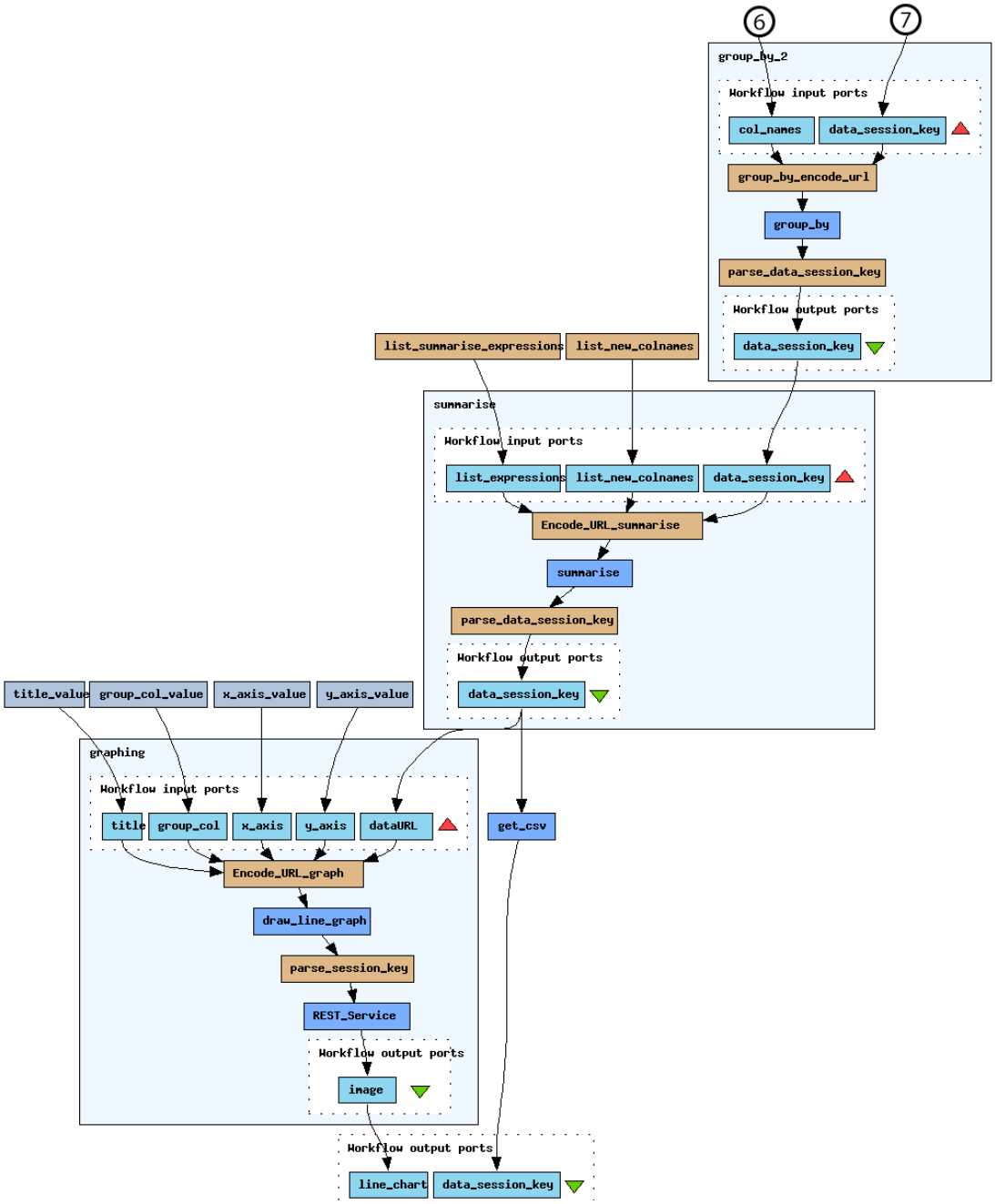


Figure C.12: Taverna workflow implementation for data wrangling task DWT_3 (3/3)

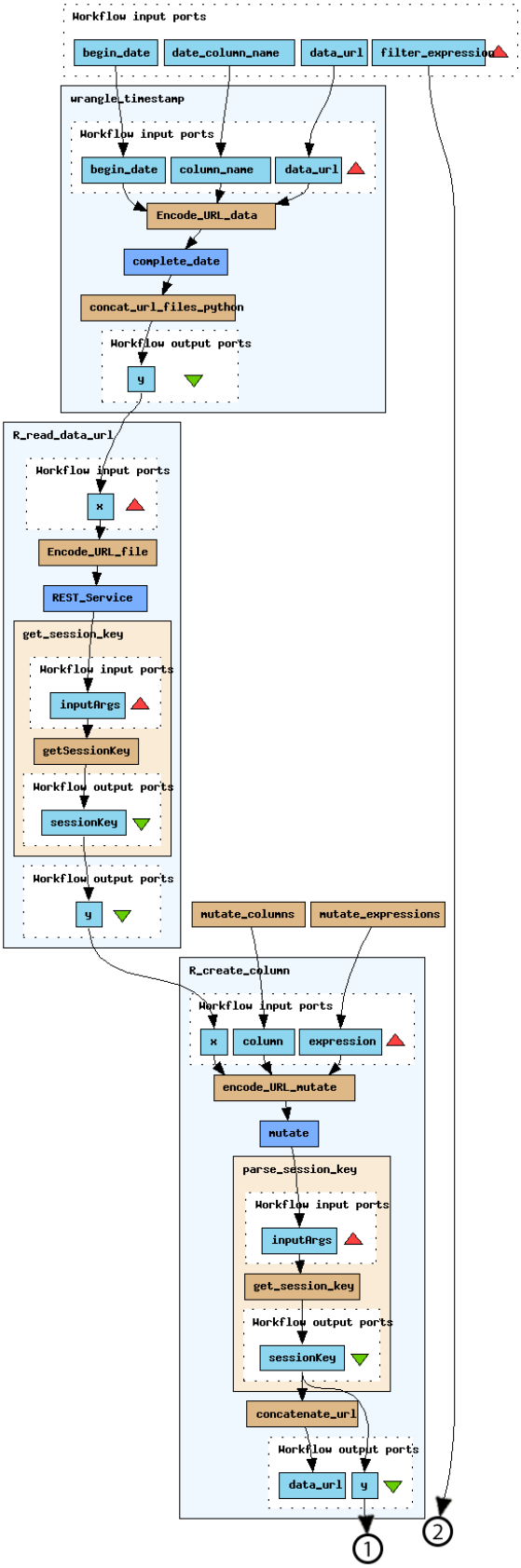


Figure C.13: Taverna workflow implementation for data wrangling task DWT_4 (1/5)

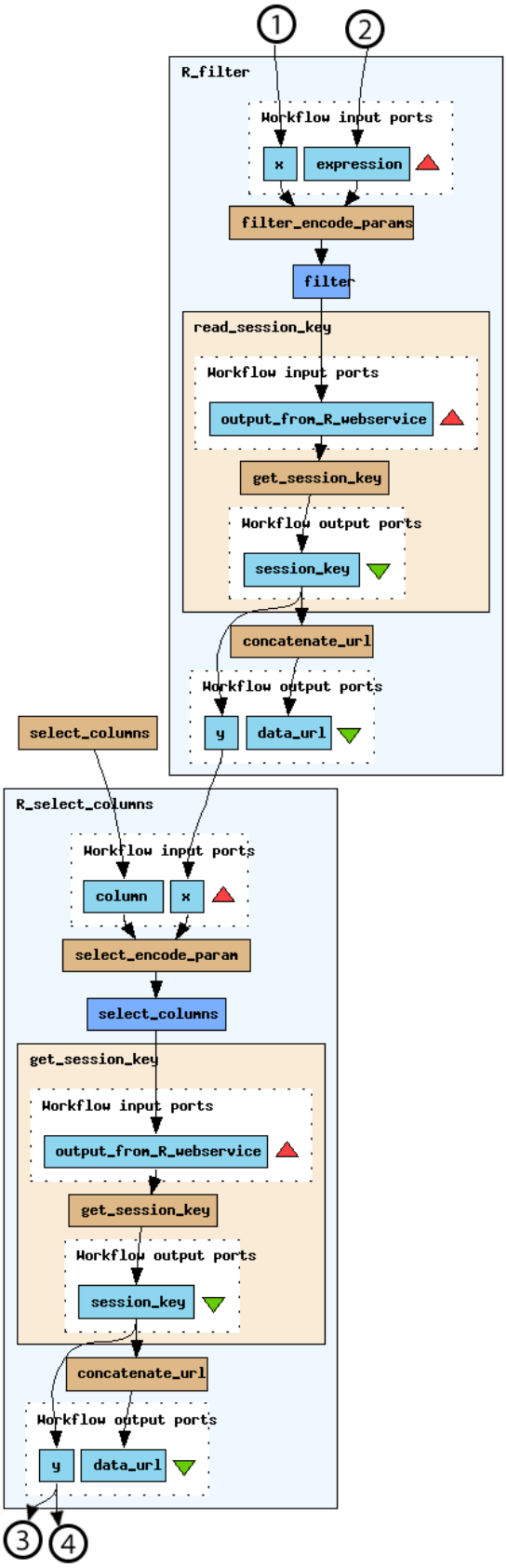


Figure C.14: Taverna workflow implementation for data wrangling task DWT_4 (2/5)

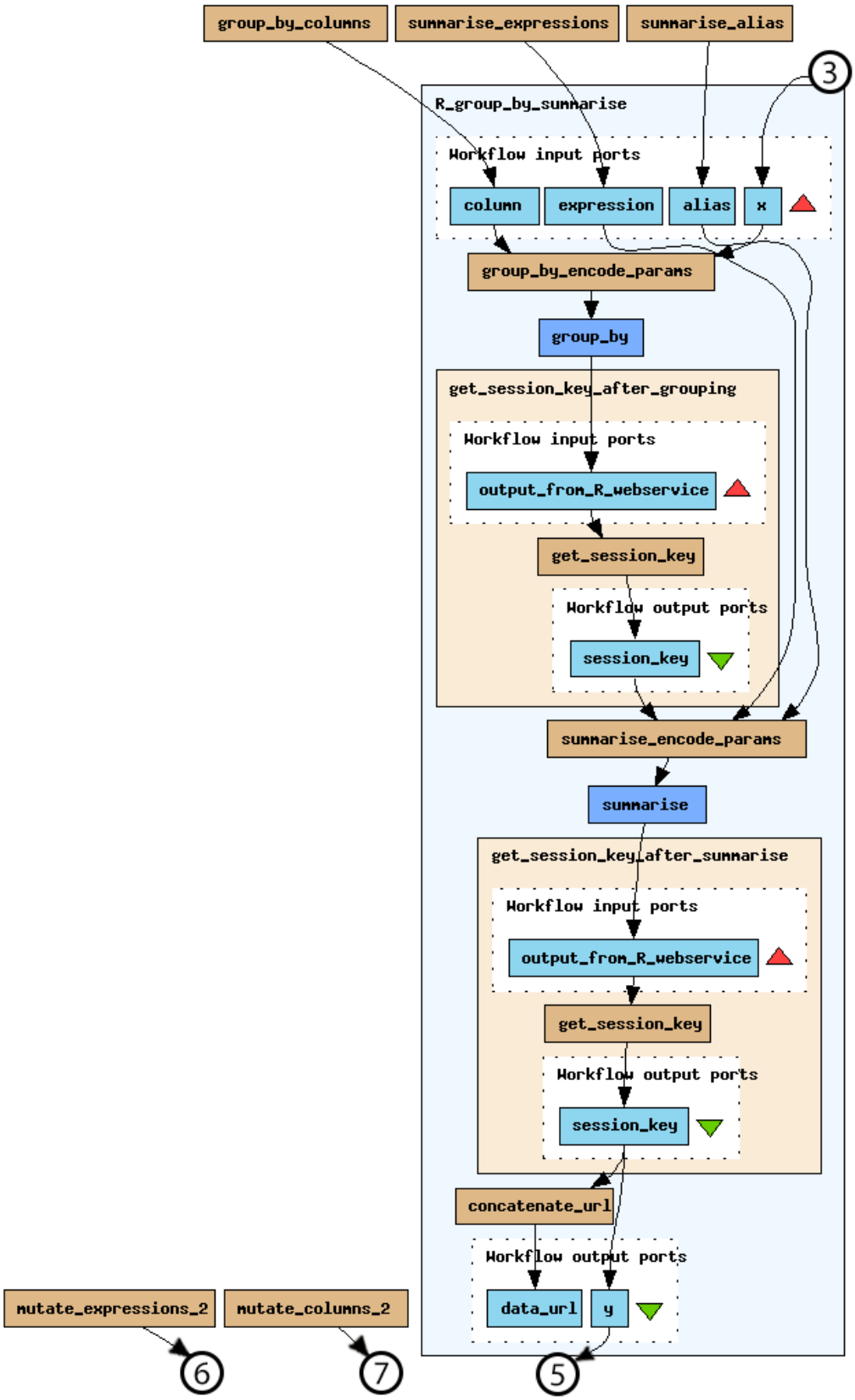


Figure C.15: Taverna workflow implementation for data wrangling task DWT_4 (3/5)

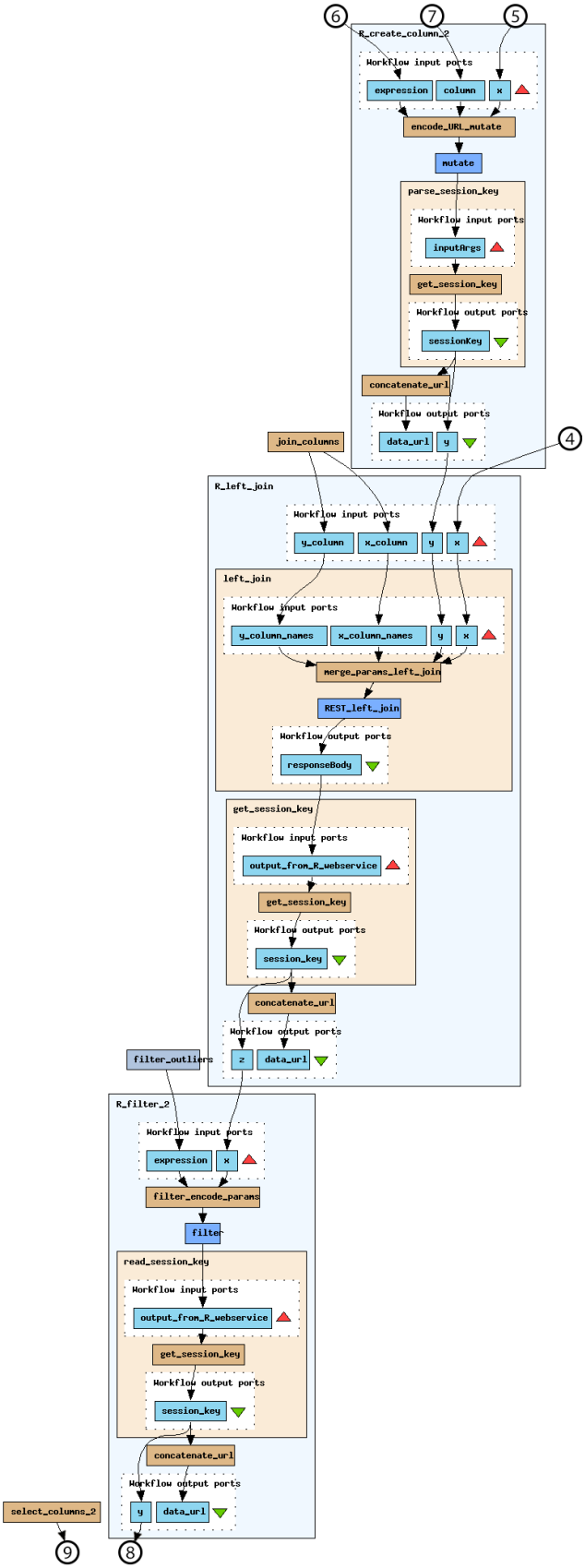


Figure C.16: Taverna workflow implementation for data wrangling task DWT_4 (4/5)

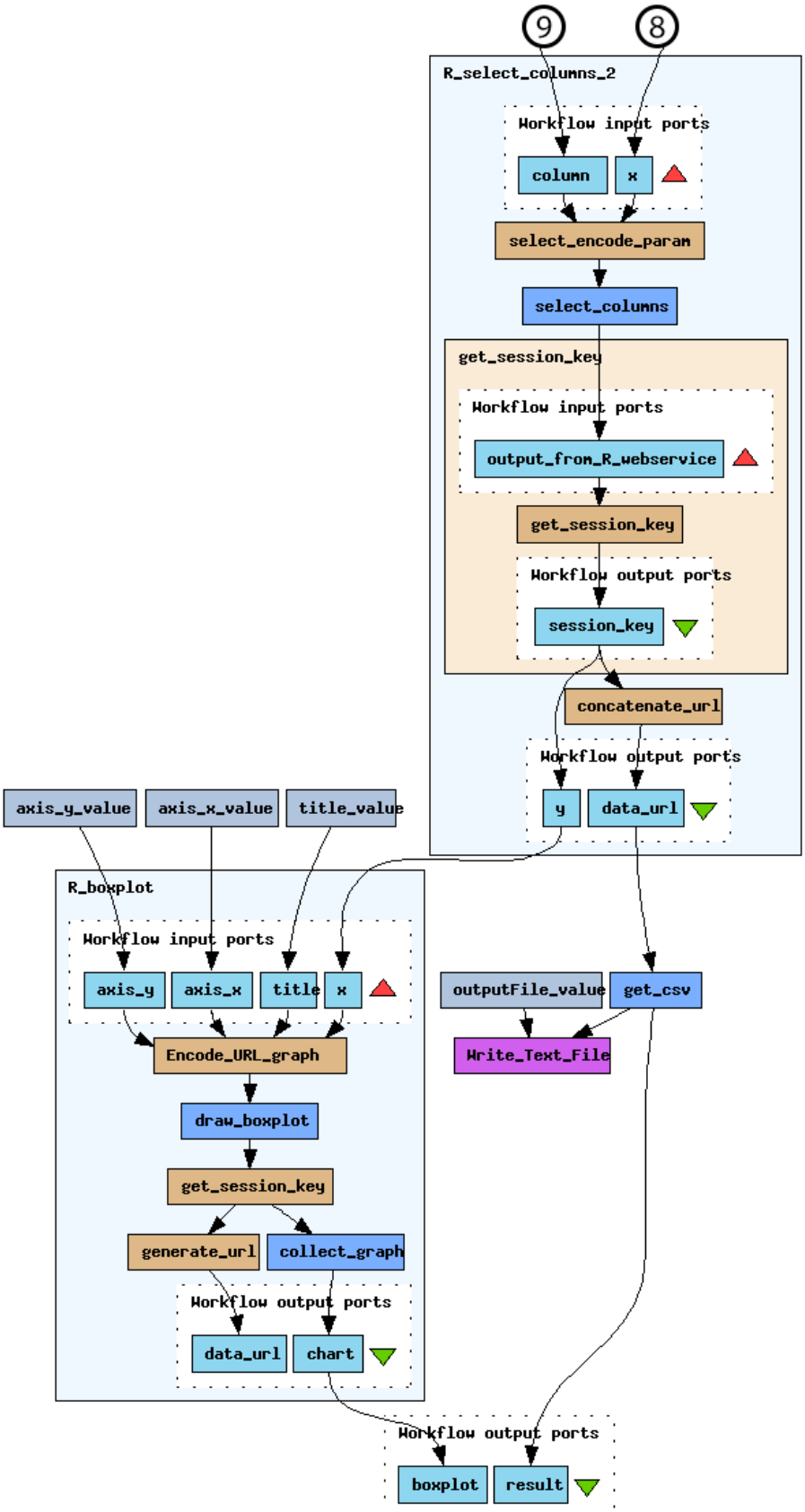


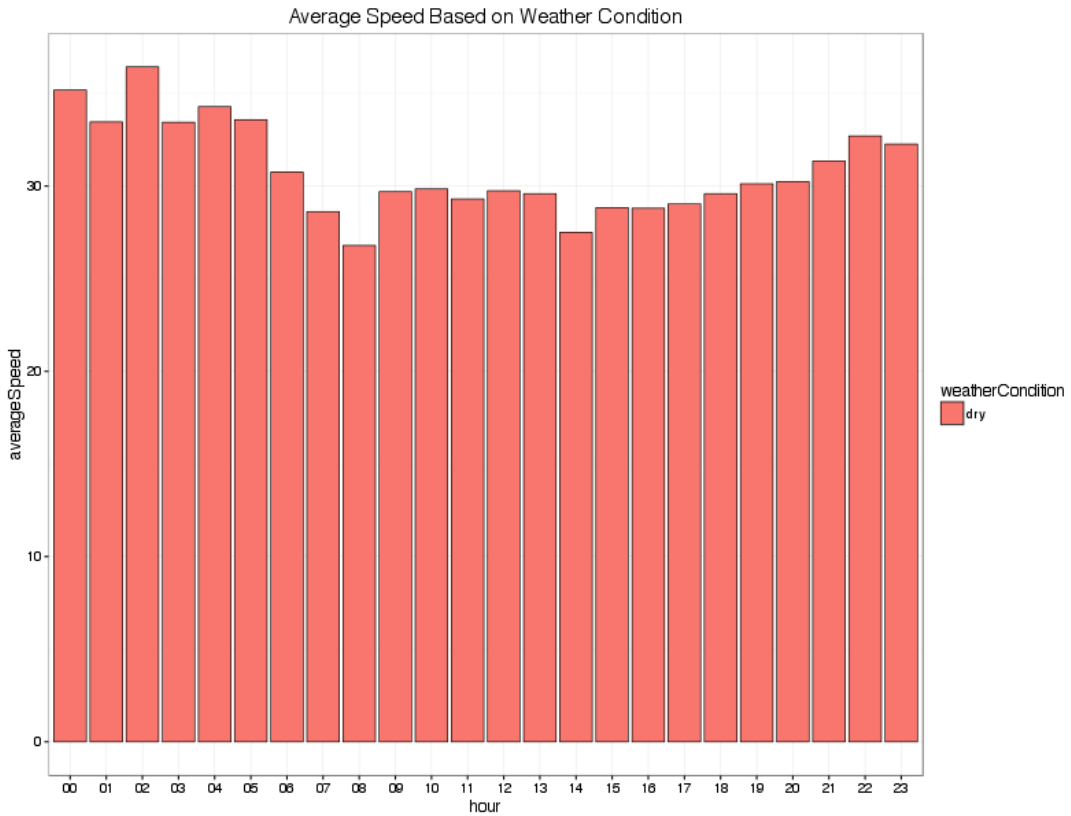
Figure C.17: Taverna workflow implementation for data wrangling task DWT_4 (5/5)

Appendix D

Data Wrangling Task Results

```
"Site.ID","hour","weatherCondition","sumSpeed","countSpeed","averageSpeed"
1304,"00","dry",2498.5,71,35.1901408450704
1304,"01","dry",1505.8,45,33.4622222222222
1304,"02","dry",1129.5,31,36.4354838709677
1304,"03","dry",2073.5,62,33.4435483870968
1304,"04","dry",2811.6,82,34.2878048780488
1304,"05","dry",12794.7,381,33.5818897637795
1304,"06","dry",42808.9,1392,30.7535201149425
1304,"07","dry",59509.2,2079,28.6239538239538
1304,"08","dry",64573.7,2410,26.7940663900415
```

(a) A sample of the result data of DWT_1



(b) A bar chart as an output of DWT_1

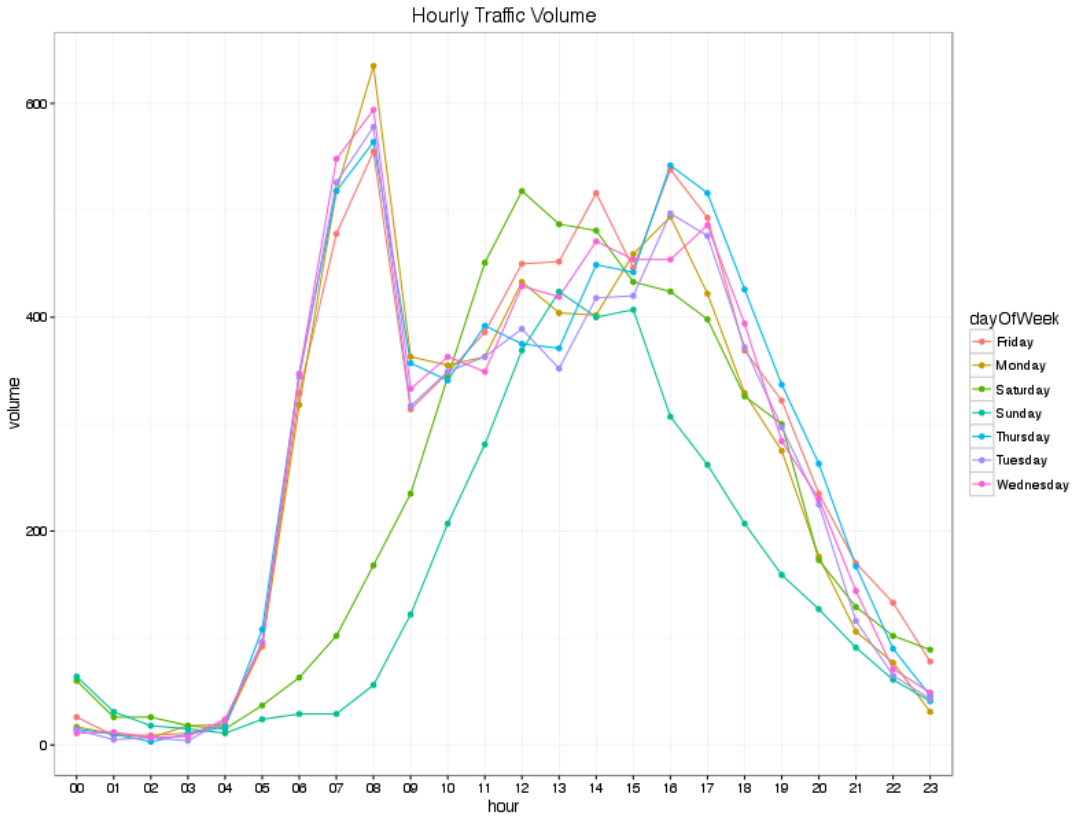
Figure D.1: The output of Data Wrangling Task DWT_1

```
"Site.ID","Complete.Timestamp","Direction.Name","Lane","hour","dayofweek","Headway..s.","Gap..s.","Speed..mph."
"'000000001304","2015-01-01 0:00:29.2","East",1,"00","Thursday",36.5,36.25,33.6
"'000000001304","2015-01-01 0:01:30.1","East",1,"00","Thursday",60.8,60.5,30.4
"'000000001304","2015-01-01 0:01:31.0","East",1,"00","Thursday",1.5,1.6,32.9
"'000000001304","2015-01-01 0:01:47.1","East",1,"00","Thursday",15.3,15,32.3
"'000000001304","2015-01-01 0:02:13.0","East",1,"00","Thursday",26.6,26.3,39.1
"'000000001304","2015-01-01 0:02:54.1","East",1,"00","Thursday",40.5,40.2,32.9
"'000000001304","2015-01-01 0:03:22.1","East",1,"00","Thursday",36.5,36.25,33.6
"'000000001304","2015-01-01 0:04:46.0","West",2,"00","Thursday",34.3,34,36.7
"'000000001304","2015-01-01 0:05:02.0","West",2,"00","Thursday",16,15.7,40.4
```

Figure D.2: A sample of the output of Data Wrangling Task DWT_2

```
"Site.ID","dayOfWeek","hour","volume"
"000000001304","Friday","00",26
"000000001304","Friday","01",9
"000000001304","Friday","02",9
"000000001304","Friday","03",11
"000000001304","Friday","04",19
"000000001304","Friday","05",96
"000000001304","Friday","06",329
"000000001304","Friday","07",478
"000000001304","Friday","08",555
```

(a) A sample of the result data of DWT_3

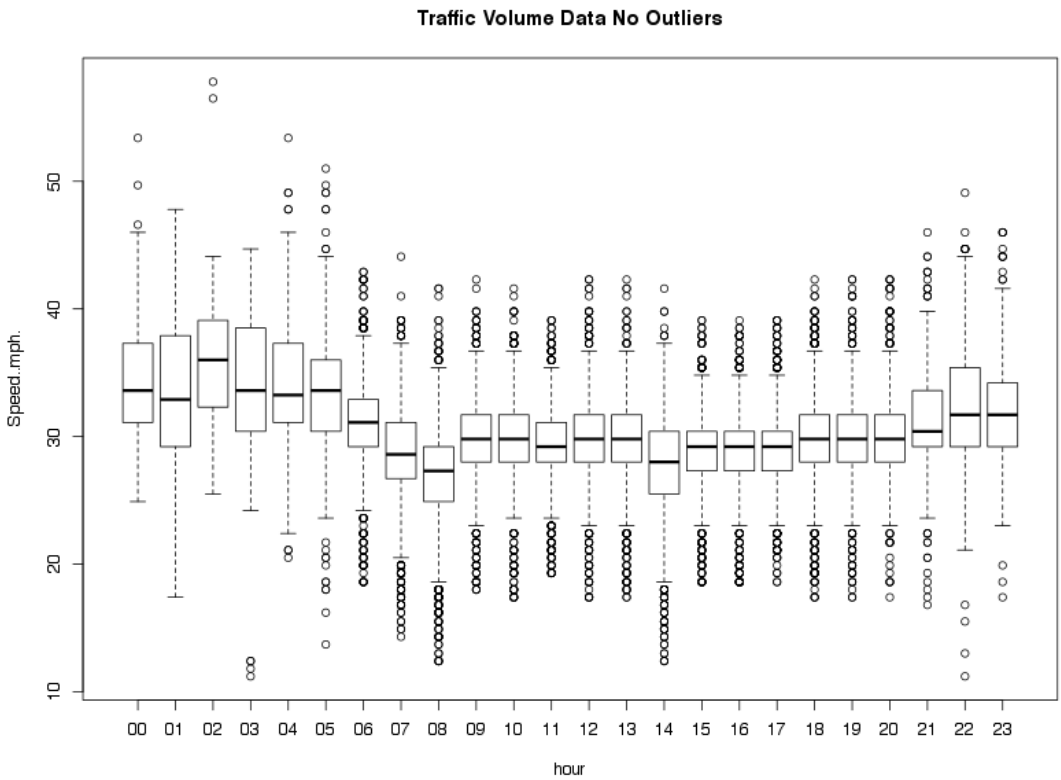


(b) A line chart as an output of DWT_3

Figure D.3: The output of Data Wrangling Task DWT_3

```
"Complete.Timestamp","hour","dayofweek","Direction.Name","Speed..mph."
"2015-01-05 0:01:56.2","00","Monday","East",29.8
"2015-01-05 0:02:03.0","00","Monday","East",36.7
"2015-01-05 0:04:20.1","00","Monday","East",35.4
"2015-01-05 0:07:30.0","00","Monday","East",37.9
"2015-01-05 0:09:14.0","00","Monday","East",36.7
"2015-01-05 0:11:10.0","00","Monday","East",32.9
"2015-01-05 0:11:43.1","00","Monday","East",32.9
"2015-01-05 0:13:35.1","00","Monday","East",31.7
"2015-01-05 0:29:08.2","00","Monday","East",29.2
```

(a) A sample of the result data of DWT_4



(b) A box plot as an output of DWT_4

Figure D.4: The output of Data Wrangling Task DWT_4

Appendix E

Improved Data Wrangling Task Workflows

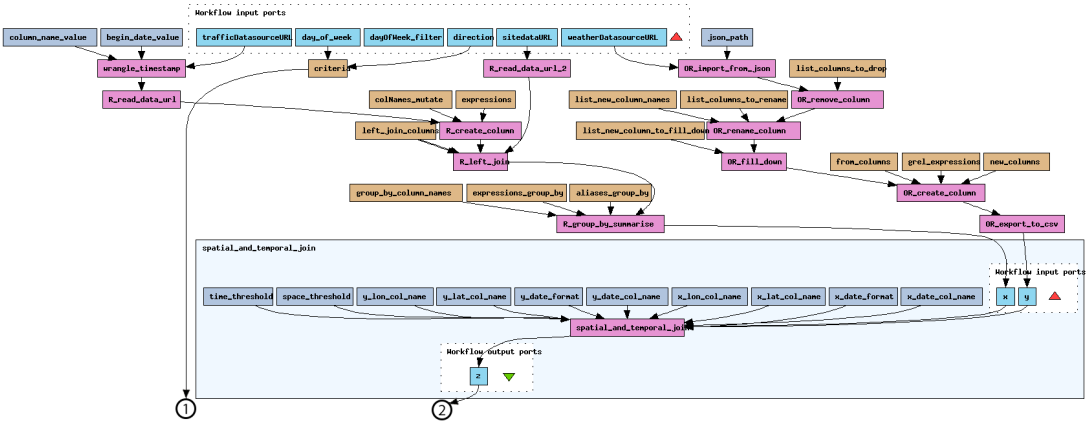


Figure E.1: Taverna workflow implementation for data wrangling task DWT_1 using customized components (1/2)

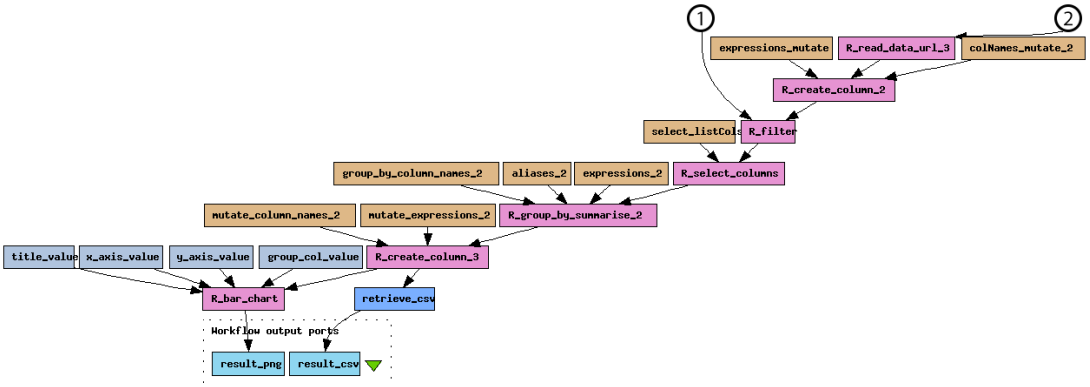


Figure E.2: Taverna workflow implementation for data wrangling task DWT_1 using customized components (2/2)

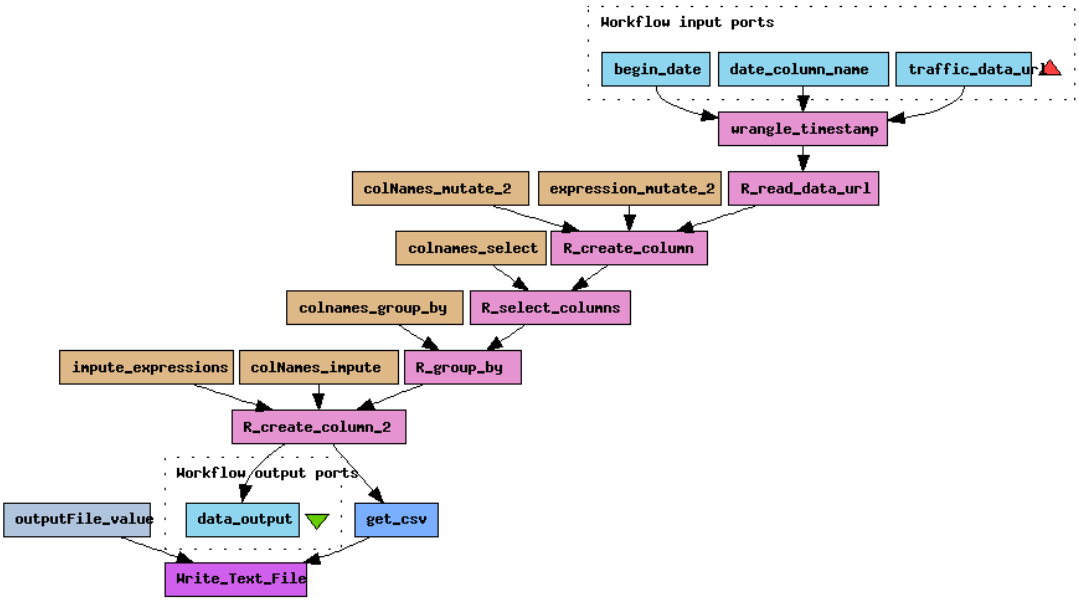


Figure E.3: Taverna workflow implementation for data wrangling task DWT_2 using customized components

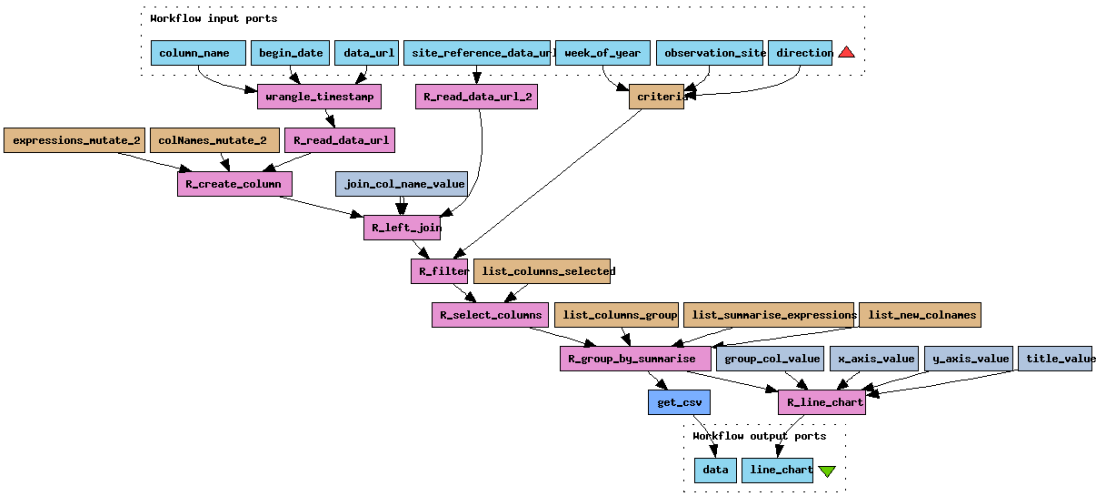


Figure E.4: Taverna workflow implementation for data wrangling task DWT_3 using customized components

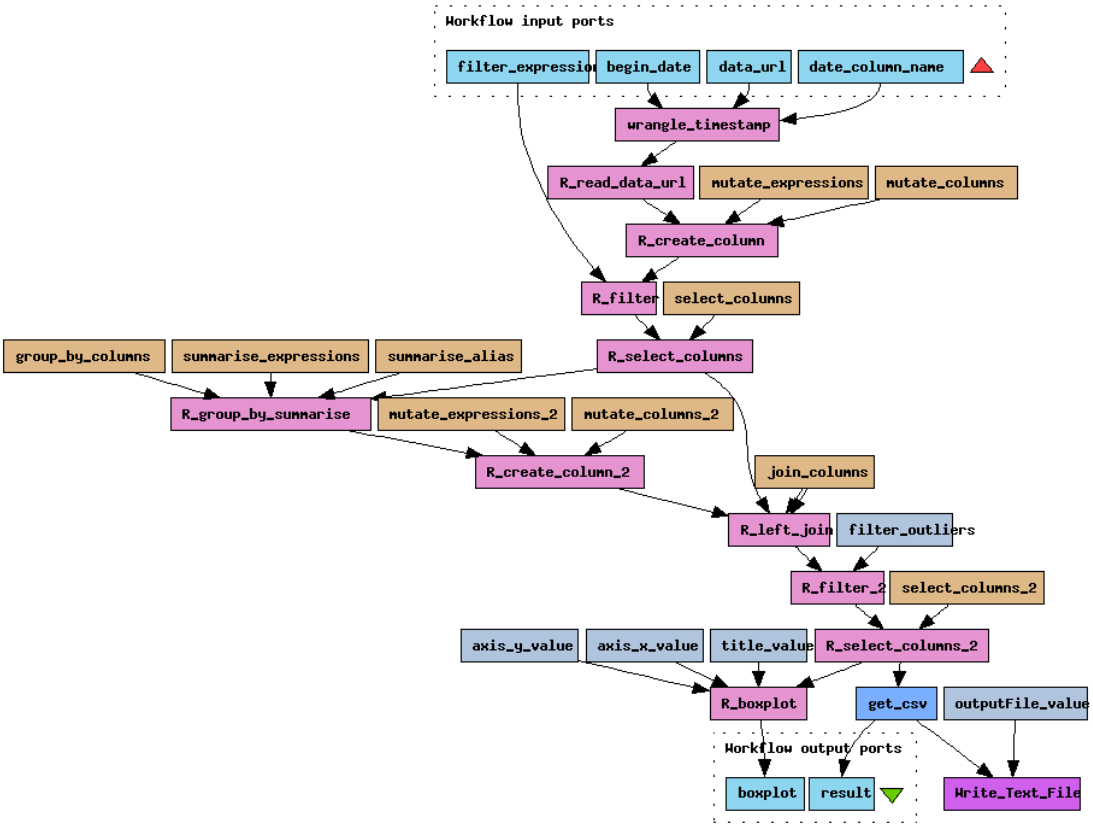


Figure E.5: Taverna workflow implementation for data wrangling task DWT_4 using customized components