

Roadmap: TruCol

A decentralised collaboration protocol for test-driven development

September 5, 2022

1 Introduction

This document presents the roadmap for the TruCol company as well as the estimated costs of the accompanying human labour and other posts. The objective of this document is to provide some basic insight into the order of magnitude of the costs of developing a sustainable variant of the TruCol company such that it is able to generate returns for its potential investors.

?? specifies the assumptions, ?? describes how the costs are estimated, section 3 presents the projected basic planning and briefly details what each activity in the Gantt chart represents. Section 6 provides some nuance on the accuracy of the estimates, and section 7 summarises the body of information in this document.

2 Cost Model

This section describes the expected costs to bring the TruCol company to an operational break-even. These costs are described in the form of a cost model, that is described in section 2.1 and specified in section 2.2. The assumptions are given in section 2.3.

2.1 Description

The total costs are computed based on the following 2 factors:

- The cumulative amount of human labour hours that are planned to be executed, multiplied with their respective hourly wage costs.
- Bounty subsidisation to attract new users.
- Buffer costs to generate wide-spread adoption of the TruCol protocol.

2.2 Specification

The costs untill TruCol is expected to be operationally break-even (OBE) are estimated using eq. (1):

$$costs_{OBE} = labour + bounty\ subsidisation + buffer \quad (1)$$

The 3 right hand terms are specified in section 2.2.1 to section 2.2.3.

2.2.1 Labour Cost Specification

The labour costs costs are mathematically described in Equation (2) and computed automatically in the Gantt chart creation using Python. They are specified as the sum of the labour costs of all workers i . The labour costs of a worker i is defined as the amount of hours worked by worker i , multiplied with the labour costs of the worker i :

$$labour = \sum_{day=1}^{day=OBE} \sum_{worker=1}^{worker=n(day)} hrs(worker, day) \cdot wage(worker) \quad (2)$$

With:

- $labour[€]$: The total expected labour cost to reach operational break-even (OBE).
- $worker[-]$: The index representing a TruCol employee.
- $n(day)[workers]$: The number of workers at TruCol on day day .
- $OBE[day]$: The index of the day on which TruCol reaches operational break-even (OBE).

- $hrs(worker, day)[hours]$: The number of hours that TruCol employee *worker* has worked on day *day*.
- $wage(worker)[€]$: The hourly labour costs of a specific TruCol employee *worker*. For simplicity, the hourly labour costs of a TruCol employee are taken as the average hourly cost of that worker, over a time of 1 year.

2.2.2 Bounty Subsidisation Cost Specification

The bounty subsidisation costs up to OBE, are estimated as:

$$bounty\ subsidisation = €100.000 \quad (3)$$

This cost is estimated with the aim of subsidising 2 to 20 companies. This range has 1 order of magnitude as range to accomodate different strategies. Either one can focus on 1 or 2 leading companies and motivate them to try the TruCol protocol. After these companies have used the protocol, work can be performed to ensure the rest of the market follows, based on the competitive advantages experiences by these 2 leading companies. Otherwise, multiple smaller companies may be motivated to use the TruCol protocol to generate a larger degree of interaction and engagement with the protocol.

2.2.3 Buffer Cost Specification

The buffer costs up to OBE, are estimated as:

$$buffer = €100.000 \quad (4)$$

This is to overcome known unknowns and possibly unknown unknown occurrences.

2.3 Assumptions

2.3.1 Decentralisation Developer Wages

The hourly wage of the developers working on decentralised technology is based on a mixture of ± 3 junior developers working at €100.000,- per year, and 2 senior developers working at €200.000,- per year. This yields an average developer cost of

$$\frac{3 \cdot 50 + 2 \cdot 100}{5} = \frac{350}{5} = €70, - \quad (5)$$

The datapoints used to come to this estimate are the promoted starting wages for Junior Developers/Engineers at Optiver in Amsterdam, Think-cell in Berlin, and a third Zurich company, which all ranged between 80 to 120k at the time of inspection (Around March 2021). No proper datapoint is used to estimate the salary of the senior developers. Previous experience in co-working with senior developers led to an estimate that their hourly contributions are at least twice as valuable as that of a junior developer. Another indicator for the doubling in wage between junior and senior dev may be the hear-say high demand in solidity/decentralisation developers.

The 70,- hourly wage is interpreted as €76, - per hour to be on the conservative side of estimates.

2.3.2 Website/Platform Wages

The website+API+GUI development is estimated at €41 per hour. This estimate is based on a reduced hourly wage of the junior decentralised technology developers (from €50, - to €40, -). Some of the development costs for these activities may be performed at a lower hourly cost price. However, the platform development work also contains UX design. And excellent UX design is quite costly, hence the average hourly wage for this estimate is kept at €41, -.

2.3.3 Business wages

The hourly wages for the business development side of our company is estimated at €36, - per hour. This estimate is based on a reduced hourly wage of the junior platform developers (from €40, - to €35, -).

2.3.4 Activity durations

The estimates for the durations of the activities for both decentralised technology development as well as ecosystem development are extrapolations of our experience in developing in these disciplines. The business development activities durations are based roughly on estimating what those activities entail and how long it would take to complete them.

3 Results

Figure 1 contains the Gantt chart that is generated to plan the development of the TruCol company. One can observe that several of the development-activities can be performed in parallel, these are accordingly stacked vertically. Dependencies of outputs of activities imply a "stairway" pattern in the Gantt chart.

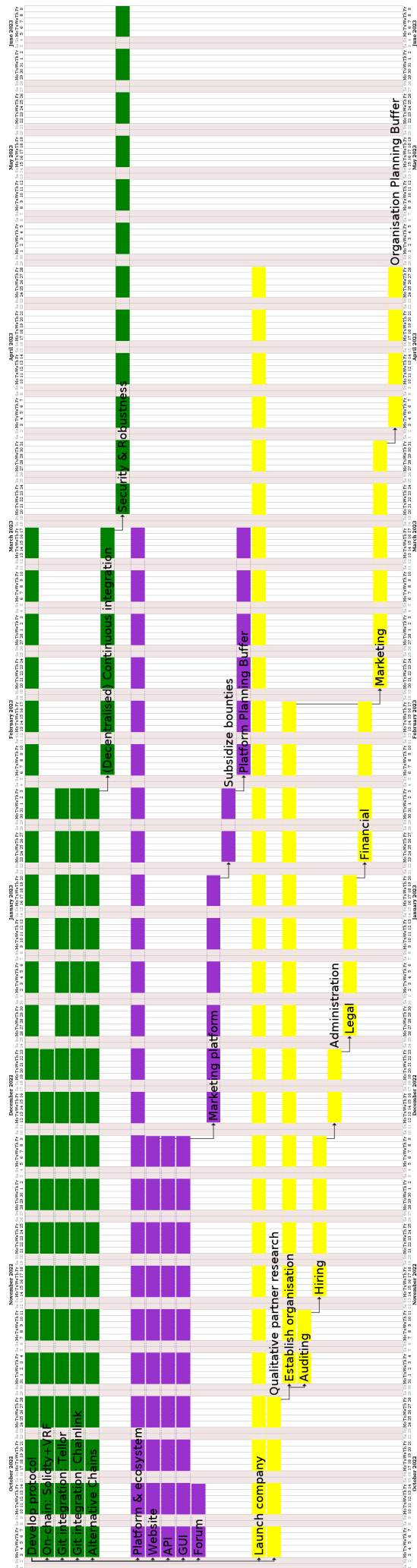


Figure 1: Gantt chart that is generated to plan the development of the TruCol company. (Source code in appendices, and on github.com/trucol/roadmap).

The description of the activities can be given as:

3.1 Decentralised Technology Development

- **Develop protocol** - The programming work and documentation work that is required to render the TruCol protocol to a mature and robust state.
- **On-chain: Solidity+VRF** - Finalisation of the Solidity to Solidity implementation of the TruCol protocol implementation that leverages Chainlinks Verifiable Random function (VRF).
- **Git integration: Tellor** - Providing a lower-cost option to the users whilst allowing the user to apply the TruCol protocol in practically any programming language using Tellor oracles that query Git repository content and (build) status.
- **Git integration: Chainlink** - Same as the Tellor option, except using Chainlinks oracles.
- **Alternative Chains** - Implementing the TruCol protocol in alternative chains to facilitate easy use for the users whilst possibly lowering costs and/or modulating the desired levels of decentralisation
- **(Decentralised) Continuous Integration** - Realizing a mature implementation in which the oracles can verify the build status (whether the tests in the smart contract actually passed or not) in a robust fashion. Ideally implementing support for decentralised CIs.
- **Security & Robustness** - A security audit of the TruCol protocol implementations.

3.2 Platform Development

- **Platform & Ecosystem** - Development of the online platform that provides a convenient place for users to use-, discuss- and learn about the TruCol protocol and its various implementations.
- **Website** - Completion of the company website.
- **API** - Application programming interface that allows users to submit contracts using the command line interface (CLI).
- **GUI** - Graphical user interface, that makes it easy and intuitive for new users to start using the TruCol protocol for their applications.
- **Forum** - Environment a-la stack-overflow that cultivates a knowledge base around the use of the TruCol protocol.
- **Marketing platform** - Development of the approach to realise wide-spread adoption of the TruCol ecosystem.
- **Subsidize bounties** - Subsidisation of bounties to attract new users to the platform.
- **Platform Planning Buffer** - A buffer accounting for unknown unknowns/unexpected delays.

3.3 Business Development

- **Launch company** - The administrative and non-technical aspects of growing the TruCol company.
- **Qualitative partner research** - An analysis to identify relevant partners in the growth of our company.
- **Establish Organisation** - organisational aspects of growing the company, with "Auditing, Hiring, Administration, Legal & Financial tasks as its respective subset".
- **Marketing** - Development of the approach to realise wide-spread adoption of the TruCol protocol whilst realising a steady stream of new customers.
- **Organisation Planning Buffer** - A buffer accounting for unknown unknowns/unexpected delays.

4 Cost estimates

After multiplying the amount of labour hours with their respective hourly labour costs, a total estimated labour costs of €470.400, – is generated. This is composed of:

- Decentralised Tech Development: €252.000, –
- Platform Development: €112.000, –
- Business Development €106.400, –

An additional €100.000, – are included for bounty subsidisation and as a buffer, yielding a total expected cost to create a healthy company of roughly $€470.000 + €100.000 = €570.000, –$ which is rounded to approximately .6 Meuro.

5 Sensitivity Analysis

6 Discussion

The hourly wage estimates for the senior decentralised technology development, platform development and business estimates could be refined by using databases of competitive salaries for the respective tasks/work. Furthermore, a refinement of the duration estimates is proposed at the moment the tasks are started, as we expect more relevant information will be available at those times. In addition, external resources should be addressed to check whether any (critical) components are omitted in this Gantt planning.

7 Conclusion

This document presents the planning to develop the healthy TruCol company within the timeframe of roughly 9 months, and documents the assumptions and methods used to generate this planning. The main tasks in this planning are composed of decentralised technology development, TruCol platform development and business development. A total labour cost of roughly €470k is estimated, and another €100k is estimated for bounty subsidisation and as a buffer.

At the end of this planning, the TruCol company is expected to sustainably operate, generating ROI. Our aim is to gradually increase the cultivation of the diversification potential of the TruCol protocol at this point, whilst starting to develop a strategy to develop our own in-house automation/AI-engine based on the dataset that we continuously grow.

References

A Appendix src/__main__.py

```
1  """Entry point for this project, runs the project code and exports
   ↪ data if
2  export commands are given to the cli command that invokes this script
   ↪ ."""
3
4
5  from src.Cost_model_params import Params
6  from src.arg_parser import parse_cli_args
7  from src.export_data.export_data import export_data
8
9  # Project code imports.
10 from src.Gantt import Gantt
11
12 # Export data import.
13
14
15 # Parse command line interface arguments to determine what this
   ↪ script does.
16 args = parse_cli_args()
17
18 # Load Params
19 params=Params()
20
21 # create gantt chart
22 gantt = Gantt("src/export_data/Diagrams/Dynamic_diagrams/gantt.uml",
   ↪ params)
23
24 # Run data export code if any argument is given.
25 if not all(
26     arg is None for arg in [args.l, args.dd, args.sd, args.c2l, args.
   ↪ ec2l]
27 ):
28     export_data(args,params)
```

B Appendix src/arg_parser.py

```
1  """This is the main code of this project nr, and it manages running
   ↪ the code
2  and outputting the results to LaTeX."""
3  import argparse
4
5
6  def parse_cli_args():
7      """Parses the command line arguments to determine what this code
   ↪ should
8      do."""
9      # Instantiate the parser
10     parser = argparse.ArgumentParser(description="Optional app
   ↪ description")
11
12     # Include argument parsing for data exporting code.
13     # Compile LaTeX
14     parser.add_argument(
15         "--l",
16         action="store_true",
17         help="Boolean indicating if code compiles LaTeX",
18     )
19
20     # Generate, compile and export Dynamic PlantUML diagrams to LaTeX
   ↪ .
21     parser.add_argument(
22         "--dd",
23         action="store_true",
24         help=(
25             "A boolean indicating if code generated diagrams are
   ↪ compiled"
26             + " and exported."
27         ),
28     )
29     # Generate, compile and export Static PlantUML diagrams to LaTeX.
30     parser.add_argument(
31         "--sd",
32         action="store_true",
33         help=(
34             "A boolean indicating if static diagrams are compiled and
   ↪ "
35             + " exported."
36         ),
37     )
38
39     # Export the project code to LaTeX.
40     parser.add_argument(
41         "--c2l",
42         action="store_true",
43         help="A boolean indicating if project code is exported to
   ↪ LaTeX.",
44     )
45
46     # Export the exporting code, and the project code to LaTeX.
47     parser.add_argument(
48         "--ec2l",
49         action="store_true",
50         help=(
51             "A boolean indicating if code that exports code is
   ↪ exported"
52             + " to LaTeX."
```



```
53         ),
54     )
55     # Load the arguments that are given.
56     args = parser.parse_args()
57     return args
```

C Appendix src/export_data/Hardcoded_data.py

```
1  """Specify hardcoded output data."""
2
3
4  # pylint: disable=R0902
5  # pylint: disable=R0903
6  class Hardcoded_data:
7      """Contains a list of parameters that are used in this project
8          ↪ that
9          combines Python code with a LaTeX document."""
10
11     def __init__(self):
12
13         # Specify code configuration details
14         # TODO: include as optional arguments.
15         self.await_compilation = True
16         self.verbose = True
17         self.gantt_extension = ".uml"
18         self.diagram_extension = ".png"
19
20         # Filenames.
21         self.main_latex_filename = "main.tex"
22         self.export_data_dirname = "export_data"
23         self.diagram_dir = "Diagrams"
24         self.plantuml_java_filename = "plantuml.jar"
25
26         # Appendix manager filenames
27         self.export_code_appendices_filename = "
28             ↪ export_code_appendices.tex"
29         self.export_code_appendices_filename_from_root = (
30             "export_code_appendices_from_root.tex"
31         )
32         self.project_code_appendices_filename = "
33             ↪ project_code_appendices.tex"
34         self.project_code_appendices_filename_from_root = (
35             "project_code_appendices_from_root.tex"
36         )
37         self.automatic_appendices_manager_filenames = [
38             self.export_code_appendices_filename,
39             self.export_code_appendices_filename_from_root,
40             self.project_code_appendices_filename,
41             self.project_code_appendices_filename_from_root,
42         ]
43
44         self.manual_appendices_filename = "manual_appendices.tex"
45         self.manual_appendices_filename_from_root = (
46             "manual_appendices_from_root.tex"
47         )
48         self.manual_appendices_manager_filenames = [
49             self.manual_appendices_filename,
50             self.manual_appendices_filename_from_root,
51         ]
52         self.appendix_dir_from_root = "latex/Appendices/"
53
54         # Folder names.
55         self.dynamic_diagram_dir = "Dynamic_diagrams"
56         self.static_diagram_dir = "Static_diagrams"
57
58         # Specify paths relative to root.
59         self.path_to_export_data_from_root = f"src/{self.
60             ↪ export_data_dirname}"
```

```
57     self.jar_path_relative_from_root = (  
58         f"{self.path_to_export_data_from_root}"  
59         + f"/{self.plant_uml_java_filename}"  
60     )  
61     self.diagram_output_dir_relative_to_root = (  
62         f"latex/Images/{self.diagram_dir}"  
63     )  
64  
65     # Path related variables  
66     self.append_export_code_to_latex = True  
67     self.path_to_dynamic_ganttts = (  
68         f"{self.path_to_export_data_from_root}/"  
69         + f"{self.diagram_dir}/{self.dynamic_diagram_dir}"  
70     )  
71     self.path_to_static_ganttts = (  
72         f"{self.path_to_export_data_from_root}/"  
73         + f"{self.diagram_dir}/{self.static_diagram_dir}"  
74     )
```

D Appendix src/export_data/Plot_to_tex.py

```
1  """File used to export plots to a latex directory.
2  Call this from another file, for project 11, question 3b:
3  from Plot_to_tex import Plot_to_tex as plt_tex
4  multiple_y_series = np.zeros((nrOfDataSeries,nrOfDataPoints), dtype=
    ↪ int);
5  # actually fill with data
6  lineLabels = [] # add a label for each dataseries
7  plt_tex.plotMultipleLines(plt_tex,single_x_series,multiple_y_series,"
    ↪ x-axis
8  label [units]","y-axis label [units]",lineLabels,"3b",4,11)
9  4b=filename
10 4 = position of legend, e.g. top right.
11 For a single line, use:
12 plt_tex.plotSingleLine(plt_tex,range(0, len(dataseries)),dataseries,"
    ↪ x-axis
13 label [units]","y-axis label [units]",lineLabel,"3b",4,11)"""
14
15 import os
16
17 import matplotlib.pyplot as plt
18 import numpy as np
19 from matplotlib import lines
20
21 # You can also plot a table directly into latex, see
    ↪ example_create_a_table(..)
22 # Then put it in latex with for example:
23 # \begin{table}[H]
24 #     \centering
25 #     \caption{Results some computation.}\label{tab:some_computation}
26 #     \begin{tabular}{|c|c|} % remember to update this
27 # % to show all columns of table
28 #         \hline
29 #         \input{latex/project3/tables/q2.txt}
30 #     \end{tabular}
31 # \end{table}
32
33
34 class Plot_to_tex:
35     """Object used to output plots to latex directory of project."""
36
37     def __init__(self):
38         self.script_dir = self.get_script_dir()
39
40     # plot graph (legendPosition = integer 1 to 4)
41     def plotSingleLine(
42         self,
43         x_path,
44         y_series,
45         x_axis_label,
46         y_axis_label,
47         label,
48         filename,
49         legendPosition,
50         project_name,
51     ):
52         """
53
54         :param x_path: param y_series:
55         :param x_axis_label: param y_axis_label:
56         :param label: param filename:
```

```

57 :param legendPosition: param project_name:
58 :param y_series: param y_axis_label:
59 :param filename: param project_name:
60 :param y_axis_label: param project_name:
61 :param project_name:
62
63 """
64 # pylint: disable=R0913
65 # TODO: reduce 9/5 arguments to at most 5/5 arguments.
66 fig = plt.figure()
67 ax = fig.add_subplot(111)
68 ax.plot(x_path, y_series, c="b", ls="--", label=label,
69         ↪ fillstyle="none")
70 plt.legend(loc=legendPosition)
71 plt.xlabel(x_axis_label)
72 plt.ylabel(y_axis_label)
73 plt.savefig(
74     os.path.dirname(__file__)
75     + f"/../../../latex/{project_name}"
76     + "/Images/"
77     + filename
78     + ".png"
79 )
80
81 # plt.show();
82
83 # plot graphs
84 def plotMultipleLines(
85     self,
86     x,
87     y_series,
88     x_label,
89     y_label,
90     label,
91     filename,
92     legendPosition,
93     project_name,
94 ):
95     """
96
97     :param x: param y_series:
98     :param x_label: param y_label:
99     :param label: param filename:
100     :param legendPosition: param project_name:
101     :param y_series: param y_label:
102     :param filename: param project_name:
103     :param y_label: param project_name:
104     :param project_name:
105
106     """
107     # pylint: disable=R0913
108     # TODO: reduce 9/5 arguments to at most 5/5 arguments.
109     fig = plt.figure()
110     ax = fig.add_subplot(111)
111
112     # generate colours
113     cmap = self.get_cmap(len(y_series[:, 0]))
114
115     # generate line types
116     lineTypes = self.generateLineTypes(y_series)
117
118     for i in range(0, len(y_series)):

```

```

118         # overwrite linetypes to single type
119         lineTypes[i] = "_"
120         ax.plot(
121             x,
122             y_series[i, :],
123             ls=lineTypes[i],
124             label=label[i],
125             fillstyle="none",
126             c=cmap(i),
127         )
128         # color
129
130     # configure plot layout
131     plt.legend(loc=legendPosition)
132     plt.xlabel(x_label)
133     plt.ylabel(y_label)
134     plt.savefig(
135         os.path.dirname(__file__)
136         + f"/../../../latex/{project_name}"
137         + "/Images/"
138         + filename
139         + ".png"
140     )
141
142     # Generate random line colours
143     # Source: https://stackoverflow.com/questions/14720331/
144     # how-to-generate-random-colors-in-matplotlib
145     def get_cmap(self, n, name="hsv"):
146         """Returns a function that maps each index in 0, 1, ..., n-1
147             ↪ to a
148             distinct RGB color; the keyword argument name must be a
149             ↪ standard mpl
150             colormap name.
151
152             :param n: param name: (Default value = "hsv")
153             :param name: Default value = "hsv")
154             """
155         return plt.cm.get_cmap(name, n)
156
157     def generateLineTypes(self, y_series):
158         """
159
160         :param y_series:
161
162         """
163         # generate varying linetypes
164         typeOfLines = list(lines.lineStyles.keys())
165
166         while len(y_series) > len(typeOfLines):
167             typeOfLines.append("-.")
168
169         # remove void lines
170         for i in range(0, len(y_series)):
171             if typeOfLines[i] == "None":
172                 typeOfLines[i] = "_"
173             if typeOfLines[i] == ":":
174                 typeOfLines[i] = ":"
175             if typeOfLines[i] == " ":
176                 typeOfLines[i] = "___"
177         return typeOfLines

```

```

177 # Create a table with: table_matrix = np.zeros((4,4),dtype=object
178     ↪ ) and pass
179 # it to this object
180 def put_table_in_tex(self, table_matrix, filename, project_name):
181     """
182     :param table_matrix: param filename:
183     :param project_name: param filename:
184     :param filename:
185     """
186     cols = np.shape(table_matrix)[1]
187     some_format = "%s"
188     for _ in range(1, cols):
189         some_format = some_format + " & %s"
190     some_format = some_format + ""
191     # TODO: Change to something else to save as txt.
192     np.savetxt(
193         os.path.dirname(__file__)
194         + f"/../../../latex/{project_name}"
195         + "/tables/"
196         + filename
197         + ".txt",
198         table_matrix,
199         delimiter=" & ",
200         fmt=format,
201         newline=" \\\\ \\hline \n",
202     )
203
204
205 # replace this with your own table creation and then pass it to
206 # put_table_in_tex(..)
207 def example_create_a_table(self):
208     """Example on how to create a latex table from Python."""
209     project_name = "1"
210     table_name = "example_table_name"
211     rows = 2
212     columns = 4
213     table_matrix = np.zeros((rows, columns), dtype=object)
214     table_matrix[:, :] = "" # replace the standard zeros with
215         ↪ empty cell
216     print(table_matrix)
217     for column in range(0, columns):
218         for row in range(0, rows):
219             table_matrix[row, column] = row + column
220     table_matrix[1, 0] = "example"
221     table_matrix[0, 1] = "grid sizes"
222
223     self.put_table_in_tex(table_matrix, table_name, project_name)
224
225 def get_script_dir(self):
226     """returns the directory of this script regardless of from
227         ↪ which level
228     the code is executed."""
229     return os.path.dirname(__file__)
230
231
232 def export_plot(self, some_plt, filename):
233     """
234     :param plt:
235     :param filename:
236     """

```

```

236         self.create_target_dir_if_not_exists("latex/Images/", "graphs
237         ↪ ")
238         some_plt.savefig(
239             "latex/Images/" + "graphs/" + filename + ".png", dpi=200
240         )
241     def create_target_dir_if_not_exists(self, path, new_dir_name):
242         """
243
244         :param path:
245         :param new_dir_name:
246
247         """
248         if os.path.exists(path):
249             if not os.path.exists(f"{path}/{new_dir_name}"):
250                 os.makedirs(f"{path}/{new_dir_name}")
251         else:
252             raise Exception(f"Error, path={path} did not exist.")
253
254 if __name__ == "__main__":
255     main = Plot_to_tex()
256     main.example_create_a_table()
257

```

E Appendix src/export_data/create_dynamic_diagrams.py

```
1  """Generates .uml PlantUML files based on Python code.
2
3  Typically used for Gantt charts.
4  """
5  from src.export_data.plantuml_compile import (
6      compile_diagrams_in_dir_relative_to_root,
7  )
8  from src.export_data.plantuml_generate import
9      ↪ generate_all_dynamic_diagrams
10 from src.export_data.plantuml_to_tex import export_diagrams_to_latex
11
12 def create_dynamic_diagrams(args, hd):
13     """Generates the dynamic diagrams."""
14     # Generate PlantUML diagrams dynamically (using code).
15     if args.dd:
16         generate_all_dynamic_diagrams(
17             f"{hd.path_to_export_data_from_root}/Diagrams/
18             ↪ Dynamic diagrams"
19         )
20
21     # Compile dynamically generated PlantUML diagrams to images.
22     compile_diagrams_in_dir_relative_to_root(
23         hd.await_compilation,
24         hd.gantt_extension,
25         hd.jar_path_relative_from_root,
26         hd.path_to_dynamic_gantts,
27         hd.verbose,
28     )
29
30     # Export dynamic PlantUML text files to LaTeX.
31     export_diagrams_to_latex(
32         hd.path_to_dynamic_gantts,
33         hd.gantt_extension,
34         hd.diagram_output_dir_relative_to_root,
35     )
36
37     # Export dynamic PlantUML diagram images to LaTeX.
38     export_diagrams_to_latex(
39         hd.path_to_dynamic_gantts,
40         hd.diagram_extension,
41         hd.diagram_output_dir_relative_to_root,
```

F Appendix src/export_data/create_static_diagrams.py

```
1  """This file creates the diagrams that are written directly in plain
2      ↪ .uml
3      files."""
4  from src.export_data.plantuml_compile import (
5      compile_diagrams_in_dir_relative_to_root,
6  )
7  from src.export_data.plantuml_to_tex import export_diagrams_to_latex
8
9  def create_static_diagrams(args, hd):
10     """Generates .png images of the static diagram .uml files.
11
12     :param args:
13     :param hd:
14     """
15     # PlantUML
16     if args.sd:
17         # Compile statically generated PlantUML diagrams to images.
18         compile_diagrams_in_dir_relative_to_root(
19             hd.await_compilation,
20             hd.gantt_extension,
21             hd.jar_path_relative_from_root,
22             hd.path_to_static_gantts,
23             hd.verbose,
24         )
25
26         # Export static PlantUML text files to LaTeX.
27         export_diagrams_to_latex(
28             hd.path_to_static_gantts,
29             hd.gantt_extension,
30             hd.diagram_output_dir_relative_to_root,
31         )
32
33         # Export static PlantUML diagram images to LaTeX.
34         export_diagrams_to_latex(
35             hd.path_to_static_gantts,
36             hd.diagram_extension,
37             hd.diagram_output_dir_relative_to_root,
38         )
```

G Appendix src/export_data/export_data.py

```
1  """File used to export data to latex."""
2  from src.export_data.helper_dir_file_edit import overwrite_file
3  from src.export_data.create_dynamic_diagrams import
4      ↪ create_dynamic_diagrams
5  from src.export_data.create_static_diagrams import
6      ↪ create_static_diagrams
7  from src.export_data.Hardcoded_data import Hardcoded_data
8  from src.export_data.latex_compile import compile_latex
9  from src.export_data.latex_export_code import export_code_to_latex
10
11 def export_data(args, params):
12     """Parses the Python arguments that specify what should be
13     ↪ compiled.
14
15     :param args:
16     """
17
18     hd = Hardcoded_data()
19
20     # Export model parameters to .tex file with LaTeX variables.
21     param_lines=[]
22     for key,value in params.wages.items():
23         param_lines.append('\newcommand'+chr(92)+str(key.replace("_"
24             ↪ ,""))+ "{"+str(value)+"}")
25     print(f'param_lines={param_lines}')
26     # Export parameters to file.
27     overwrite_file("latex/Tables/params.tex", param_lines)
28     exit()
29
30     # Generating PlantUML diagrams
31     create_dynamic_diagrams(args, hd)
32     create_static_diagrams(args, hd)
33
34     # Plotting graphs using Python code, and export them to latex.
35     # Generate plots.
36     # Export plots to LaTeX.
37
38     # Export code to LaTeX.
39     if args.c2l:
40         # TODO: verify whether the latex/{project_name}/Appendices
41         ↪ folder
42         # exists before exporting.
43         # TODO: verify whether the latex/{project_name}/Images folder
44         ↪ exists
45         # before exporting.
46         export_code_to_latex(hd, False)
47     elif args.ec2l:
48         # TODO: verify whether the latex/{project_name}/Appendices
49         ↪ folder
50         # exists before exporting.
51         # TODO: verify whether the latex/{project_name}/Images folder
52         ↪ exists
53         # before exporting.
54         export_code_to_latex(hd, True)
55
56     # Compile the accompanying LaTeX report.
57     if args.l:
58         compile_latex(True, True)
59         print("")
```

```
print("\n\nDone exporting data.")
```

H Appendix src/export_data/helper_bash_commands.py

```
1  """Code used to execute Bash commands from Python."""
2  import subprocess # nosec
3
4  from src.export_data.plantuml_compile import
   ↪ get_output_of_bash_command
5
6
7  def run_a_bash_command(await_compilation, bash_command, verbose):
8      """Runs a bash commands from Python.
9
10     :param await_compilation:
11     :param bash_command:
12     :param verbose:
13     """
14     if await_compilation:
15         if verbose:
16             subprocess.call(
17                 bash_command,
18                 shell=True, # nosec
19                 stdout=subprocess.PIPE,
20                 stderr=subprocess.STDOUT,
21             )
22         else:
23             # pylint: disable=E1129
24             # pylint: disable=R1732
25             subprocess.call(
26                 bash_command,
27                 shell=True, # nosec
28                 stderr=subprocess.DEVNULL,
29                 stdout=subprocess.DEVNULL,
30             )
31     else:
32         if verbose:
33             # pylint: disable=R1732
34             subprocess.Popen(
35                 bash_command,
36                 shell=True, # nosec
37                 stdout=subprocess.PIPE,
38                 stderr=subprocess.STDOUT,
39             )
40         else:
41             with subprocess.Popen(
42                 bash_command,
43                 shell=True, # nosec
44                 stderr=subprocess.DEVNULL,
45                 stdout=subprocess.DEVNULL,
46             ) as process:
47                 get_output_of_bash_command(process)
```

I Appendix src/export_data/helper_dir_file_edit.py

```
1  """A helper file that is used for directory and file editing."""
2  import glob
3  import os
4  import shutil
5
6
7  def file_contains(filepath, substring):
8      """Returns True if a file exists. None otherwise.
9
10     :param filepath:
11     :param substring:
12     """
13     with open(filepath, encoding="utf-8") as f:
14         return substring in f.read()
15
16
17  def get_dir_filelist_based_on_extension(dir_relative_to_root,
18     ↪ extension):
19     """
20     :param dir_relative_to_root: A relative directory as
21     seen from the root dir of this project.
22     :param extension: File extension that is used/searched in this
23     ↪ function.
24
25     """
26     selected_filenames = []
27     # TODO: assert directory exists
28     for filename in os.listdir(dir_relative_to_root):
29         if filename.endswith(extension):
30             selected_filenames.append(filename)
31     return selected_filenames
32
33  def create_dir_relative_to_root_if_not_exists(dir_relative_to_root):
34     """
35
36     :param dir_relative_to_root: A relative directory as
37     seen from the root dir of this project.
38
39     """
40     if not os.path.exists(dir_relative_to_root):
41         os.makedirs(dir_relative_to_root)
42
43
44  def dir_relative_to_root_exists(dir_relative_to_root):
45     """
46
47     :param dir_relative_to_root: A relative directory as
48     seen from the root dir of this project.
49
50     """
51     if not os.path.exists(dir_relative_to_root):
52         return False
53     if os.path.exists(dir_relative_to_root):
54         return True
55     raise Exception(
56         f"Directory relative to root: {dir_relative_to_root}"
57         + " did not exist, nor did it exist."
58     )
```

```

59
60
61 def get_all_files_in_dir_and_child_dirs(extension, path,
    ↳ excluded_files=None):
62     """Returns a list of the relative paths to all files within the
        ↳ some path
63     that match the given file extension. Also includes files in child
64     directories.
65
66     :param extension: File extension that is used/searched in this
        ↳ function.
67     The file extension of the file that is sought in the appendix
        ↳ line. Either
68     ".py" or ".pdf".
69     :param path: Absolute filepath in which files are being sought.
70     :param excluded_files: Default value = None) Files that will not
        ↳ be
71     included even if they are found.
72     """
73     filepaths = []
74     for r, _, f in os.walk(path):
75         for file in f:
76             if file.endswith(extension):
77                 if (excluded_files is None) or (
78                     excluded_files is not None)
79                     and (file not in excluded_files)
80                 ):
81                     filepaths.append(r + "/" + file)
82     return filepaths
83
84
85 def get_filepaths_in_dir(extension, path, excluded_files=None):
86     """Returns a list of the relative paths to all files within the
        ↳ some path
87     that match the given file extension. Does not include files in
88     child_directories.
89
90     :param extension: File extension that is used/searched in this
        ↳ function.
91     The file extension of the file that is sought in the appendix
        ↳ line.
92     Either ".py" or ".pdf".
93     :param path: Absolute filepath in which files are being sought.
94     :param excluded_files: Default value = None) Files that will not
        ↳ be
95     included even if they are found.
96     """
97     filepaths = []
98     current_path = os.getcwd()
99     os.chdir(path)
100    for file in glob.glob(f"*.{extension}"):
101        print(file)
102        if (excluded_files is None) or (
103            excluded_files is not None) and (file not in
            ↳ excluded_files)
104        ):
105            # Append normalised filepath e.g. collapses b/src/../d to
            ↳ b/d.
106            filepaths.append(os.path.normpath(f"{path}/{file}"))
107    os.chdir(current_path)
108    return filepaths
109

```

```

110 def sort_filepaths_by_filename(filepaths):
111     """
112
113     :param filepaths:
114
115     """
116     # filepaths.sort(key = lambda x: x.split()[1])
117     filepaths.sort(key=lambda x: x[x.rfind("/") + 1 :]) # noqa: E203
118     for filepath in filepaths:
119         print(f"{filepath}")
120     return filepaths
121
122
123 def get_filename_from_dir(path):
124     """Returns a filename from an absolute path to a file.
125
126     :param path: path to a file of which the name is queried.
127     """
128     return path[path.rfind("/") + 1 :] # noqa: E203
129
130
131 def delete_file_if_exists(filepath):
132     """
133
134     :param filepath:
135
136     """
137     try:
138         os.remove(filepath)
139     except OSError:
140         pass
141
142
143 def convert_filepath_to_filepath_from_root(filepath,
144     ↪ normalised_root_path):
145     """
146
147     :param filepath:
148     :param normalised_root_path:
149
150     """
151     normalised_filepath = os.path.normpath(filepath)
152     filepath_relative_from_root = normalised_filepath[
153         len(normalised_root_path) : # noqa: E203
154     ]
155     return filepath_relative_from_root
156
157
158 def overwrite_file(filepath, lines):
159     """
160
161     :param filepath:
162     :param lines:
163
164     """
165     with open(filepath, "w", encoding="utf-8") as the_file:
166         for line in lines:
167             the_file.write(f"{line}\n")
168
169 def append_lines_to_file(filepath, lines):
170     """

```



```

171
172 :param filepath:
173 :param lines:
174
175 """
176 with open(filepath, "a", encoding="utf-8") as the_file:
177     for line in lines:
178         the_file.write(f"{line}\n")
179
180
181 def append_line_to_file(filepath, line):
182     """
183
184     :param filepath:
185     :param line:
186
187     """
188     with open(filepath, "a", encoding="utf-8") as the_file:
189         the_file.write(f"{line}\n")
190         the_file.close()
191
192
193 def remove_all_auto_generated_appendices(hd):
194     """
195
196     :param hd:
197
198     """
199
200     # TODO: move identifier into hardcoded.
201     all_appendix_files = get_all_files_in_dir_and_child_dirs(
202         ".tex", hd.appendix_dir_from_root, excluded_files=None
203     )
204     for file in all_appendix_files:
205         if "Auto_generated" in file:
206             delete_file_if_exists(file)
207
208
209 def delete_dir_relative_to_root_if_not_exists(dir_relative_to_root):
210     """
211
212     :param dir_relative_to_root: A relative directory as
213     seen from the root dir of this project.
214
215     """
216     if os.path.exists(dir_relative_to_root):
217         # Remove directory and its content.
218         shutil.rmtree(dir_relative_to_root)

```

J Appendix src/export_data/helper_tex_editing.py

```
1  """Helper that modifies LaTeX file to allow automatically including
   ↪ Pyathon
2  code as appendices."""
3  import os
4
5  from src.export_data.helper_dir_file_edit import (
6      append_line_to_file,
7      append_lines_to_file,
8      convert_filepath_to_filepath_from_root,
9      delete_file_if_exists,
10     get_filename_from_dir,
11 )
12
13
14 def code_filepath_to_tex_appendix_filename(
15     filename, _, is_project_code, is_export_code
16 ):
17     """
18
19     :param filename:
20     :param from_root:
21     :param is_project_code:
22     :param is_export_code:
23
24     """
25
26     # TODO: Include assert to verify filename ends at .py.
27     # TODO: Include assert to verify filename doesn't end at .py
   ↪ anymore.
28     filename_without_extension = os.path.splitext(filename)[0]
29
30     # Create appendix filename identifier segment
31     verify_input_code_type(is_export_code, is_project_code)
32     if is_project_code:
33         identifier = "Auto-generated-project-code-appendix-"
34     elif is_export_code:
35         identifier = "Auto-generated-export-code-appendix-"
36
37     appendix_filename = f"{identifier}{filename_without_extension}"
38     return appendix_filename
39
40
41 def verify_input_code_type(is_export_code, is_project_code):
42     """
43
44     :param is_export_code:
45     :param is_project_code:
46
47     """
48     # Create appendix filename identifier segment
49     if is_project_code and is_export_code:
50         raise Exception(
51             "Error, a file can't be both project code, and export
   ↪ code at"
52             + " same time."
53         )
54     if not is_project_code and not is_export_code:
55         raise Exception(
56             "Error, don't know what to do with files that are neither
   ↪ project"
```

```

57         + " code, nor export code."
58     )
59
60
61 def tex_appendix_filename_to_inclusion_command(appendix_filename,
62     ↪ from_root):
63     """
64     :param appendix_filename:
65     :param from_root:
66
67     """
68     # Create full appendix filename.
69     if from_root:
70         # Generate latex inclusion command for latex compilation from
71         ↪ root dir.
72         appendix_inclusion_command = (
73             f"\\input{{latex/Appendices/{appendix_filename}.tex}} \\
74             ↪ newpage"
75         )
76         # \input{latex/Appendices/Auto_generated_py_App8.tex} \
77         ↪ newpage
78     else:
79         # \input{Appendices/Auto_generated_py_App8.tex} \newpage
80         appendix_inclusion_command = (
81             f"\\input{{Appendices/{appendix_filename}.tex}} \\newpage
82             ↪ "
83         )
84     return appendix_inclusion_command
85
86
87 def create_appendix_filecontent(
88     latex_object_name, filename, filepath_from_root, from_root
89 ):
90     """
91     :param latex_object_name:
92     :param filename:
93     :param filepath_from_root:
94     :param from_root:
95
96     """
97     # Latex titles should escape underscores.
98     filepath_from_root_without_underscores = filepath_from_root.
99     ↪ replace(
100         "-", r"\-"
101     )
102     lines = []
103     lines.append(
104         rf"\\{latex_object_name}{{Appendix "
105         + rf"\\{filepath_from_root_without_underscores}}}"
106         + rf"\\label{{app:{{filename}}}"
107     )
108     if from_root:
109         lines.append(rf"\pythonexternal{{latex/..{filepath_from_root
110         ↪ }}}}")
111     else:
112         lines.append(rf"\pythonexternal{{latex/..{filepath_from_root
113         ↪ }}}}")
114     return lines

```

```

111 def create_appendix_manager_files(hd):
112     """
113
114     :param hd:
115
116     """
117     # Verify target directory exists.
118     if not os.path.exists(hd.appendix_dir_from_root):
119         raise Exception(
120             "Error, the Appendices directory was not found at:"
121             + f"{hd.appendix_dir_from_root}"
122         )
123
124     # Delete appendix manager files.
125     list(
126         map(
127             lambda x: delete_file_if_exists(f"{hd.
128                 ↪ appendix_dir_from_root}{x}"),
129             hd.automatic_appendices_manager_filenames,
130         )
131     )
132
133     # Create new appendix_manager_files
134     list(
135         map(
136             # pylint: disable=R1732
137             lambda x: open(
138                 f"{hd.appendix_dir_from_root}{x}", "a", encoding="utf
139                 ↪ _8"
140             ),
141             hd.automatic_appendices_manager_filenames,
142         )
143     )
144
145     # Ensure manual appendix_manager_files are created.
146     list(
147         map(
148             # pylint: disable=R1732
149             lambda x: open(
150                 f"{hd.appendix_dir_from_root}{x}", "a", encoding="utf
151                 ↪ _8"
152             ),
153             hd.manual_appendices_manager_filenames,
154         )
155     )
156
157     # pylint: disable=R0913
158     def create_appendix_file(
159         hd,
160         filename,
161         filepath_from_root,
162         latex_object_name,
163         is_export_code,
164         is_project_code,
165     ):
166         """
167
168         :param hd:
169         :param filename:
170         :param filepath_from_root:
171         :param latex_object_name:

```

```

170 :param is_export_code:
171 :param is_project_code:
172
173 """
174 verify_input_code_type(is_export_code, is_project_code)
175 filename_without_extension = os.path.splitext(filename)[0]
176 if is_project_code:
177     # Create the appendix for the case the latex is compiled from
178     ↪ root.
179     appendix_filepath = (
180         f"{hd.appendix_dir_from_root}/Auto_generated_proj"
181         + f"ect_code_appendix_{filename_without_extension}.tex"
182     )
183
184     # Append latex_filepath to appendix manager.
185     # append_lines_to_file(
186     #     f"{hd.appendix_dir_from_root}{hd.
187     ↪ project_code_appendices_filename}",
188     #     [tex_appendix_filepath_to_inclusion_command(
189     ↪ appendix_filepath)],
190     # )
191
192     # Get Appendix .tex content.
193     appendix_lines_from_root = create_appendix_filecontent(
194         latex_object_name, filename, filepath_from_root, True
195     )
196
197     # Write appendix to .tex file.
198     append_lines_to_file(appendix_filepath,
199     ↪ appendix_lines_from_root)
200 elif is_export_code:
201     # Create the appendix for the case the latex is compiled from
202     ↪ root.
203     appendix_filepath = (
204         f"{hd.appendix_dir_from_root}/Auto_generated_export"
205         + f"_code_appendix_{filename_without_extension}.tex"
206     )
207
208     # Append latex_filepath to appendix manager.
209     # append_lines_to_file(
210     #     f"{hd.appendix_dir_from_root}{hd.
211     ↪ export_code_appendices_filename}",
212     #     [tex_appendix_filepath_to_inclusion_command(
213     ↪ appendix_filepath)],
214     # )
215
216     # Get Appendix .tex content.
217     appendix_lines_from_root = create_appendix_filecontent(
218         latex_object_name, filename, filepath_from_root, True
219     )
220
221     # Write appendix to .tex file.
222     append_lines_to_file(appendix_filepath,
223     ↪ appendix_lines_from_root)
224 # TODO: verify files exist
225
226 def export_python_project_code(
227     hd, normalised_root_dir, python_project_code_filepaths
228 ):
229     """

```

```

224 :param hd:
225 :param normalised_root_dir:
226 :param python_project_code_filepaths:
227
228 """
229 is_project_code = True
230 is_export_code = False
231 from_root = False
232 for filepath in python_project_code_filepaths:
233     create_appendices(
234         hd,
235         filepath,
236         normalised_root_dir,
237         from_root,
238         is_export_code,
239         is_project_code,
240     )
241     create_appendices(
242         hd,
243         filepath,
244         normalised_root_dir,
245         True,
246         is_export_code,
247         is_project_code,
248     )
249
250
251 def export_python_export_code(
252     hd, normalised_root_dir, python_export_code_filepaths
253 ):
254     """
255
256     :param hd:
257     :param normalised_root_dir:
258     :param python_export_code_filepaths:
259
260     """
261     is_project_code = False
262     is_export_code = True
263     from_root = False
264     for filepath in python_export_code_filepaths:
265         create_appendices(
266             hd,
267             filepath,
268             normalised_root_dir,
269             from_root,
270             is_export_code,
271             is_project_code,
272         )
273         create_appendices(
274             hd,
275             filepath,
276             normalised_root_dir,
277             True,
278             is_export_code,
279             is_project_code,
280         )
281
282
283 # pylint: disable=R0913
284 def create_appendices(
285     hd,

```

```

286     filepath,
287     normalised_root_dir,
288     from_root,
289     is_export_code,
290     is_project_code,
291 ):
292     """
293
294     :param hd:
295     :param filepath:
296     :param normalised_root_dir:
297     :param from_root:
298     :param is_export_code:
299     :param is_project_code:
300
301     """
302     # Get the filepath of a python file from the root dir of this
303     ↪ project.
304     filepath_from_root = convert_filepath_to_filepath_from_root(
305         filepath, normalised_root_dir
306     )
307     print(f"from_root={from_root}, filepath_from_root={
308         ↪ filepath_from_root}")
309
310     # Get the filename of a python filepath
311     filename = get_filename_from_dir(filepath)
312
313     # Get the filename for a latex appendix from a python filename.
314     appendix_filename = code_filepath_to_tex_appendix_filename(
315         filename, from_root, is_project_code, is_export_code
316     )
317
318     # Command to include the appendix in the appendices manager.
319     appendix_inclusion_command =
320     ↪ tex_appendix_filename_to_inclusion_command(
321         appendix_filename, from_root
322     )
323     # if from_root:
324     #     print(f'tex_appendix_filename_to_inclusion_command=
325     # + {appendix_inclusion_command}')
326     #     exit()
327
328     append_appendix_to_appendix_managers(
329         appendix_inclusion_command,
330         from_root,
331         hd,
332         is_export_code,
333         is_project_code,
334     )
335
336     # Create the appendix .tex file.
337     # TODO: move "section" to hardcoded.
338     if from_root: # Appendix only contains files readable from root.
339         create_appendix_file(
340             hd,
341             filename,
342             filepath_from_root,
343             "section",
344             is_export_code,
345             is_project_code,
346         )

```

```

345
346 def append_appendix_to_appendix_managers(
347     appendix_inclusion_command, from_root, hd, is_export_code,
348     ↪ is_project_code
349 ):
350     """
351     :param appendix_inclusion_command:
352     :param from_root:
353     :param hd:
354     :param is_export_code:
355     :param is_project_code:
356
357     """
358     # Append the appendix .tex file to the appendix manager.
359     if is_project_code:
360         if from_root:
361             # print('from_root={from_root}Append to:
362             # +f"{hd.project_code_appendices_filename_from_root}')
363             append_line_to_file(
364                 f"{hd.appendix_dir_from_root}"
365                 + f"{hd.project_code_appendices_filename_from_root}",
366                 appendix_inclusion_command,
367             )
368         else:
369             # print(f'from_root={from_root}Append to:"
370             # +f"{hd.project_code_appendices_filename}')
371             append_line_to_file(
372                 f"{hd.appendix_dir_from_root}"
373                 + f"{hd.project_code_appendices_filename}",
374                 appendix_inclusion_command,
375             )
376
377     if is_export_code:
378         if from_root:
379             append_line_to_file(
380                 f"{hd.appendix_dir_from_root}"
381                 + f"{hd.export_code_appendices_filename_from_root}",
382                 appendix_inclusion_command,
383             )
384         else:
385             append_line_to_file(
386                 f"{hd.appendix_dir_from_root}"
387                 + f"{hd.export_code_appendices_filename}",
388                 appendix_inclusion_command,
389             )

```

K Appendix src/export_data/helper_tex_reading.py

```
1  """Helper script to parse Latex files, to support including LaTeX
2  appendices."""
3  from src.export_data.helper_dir_file_edit import file_contains
4
5
6  def verify_latex_supports_auto_generated_appendices(
7      ↪ path_to_main_latex_file):
8      """Ensures the Latex file supports including automatically
9      ↪ appending code
10     as appendices.
11
12     :param path_to_main_latex_file:
13     """
14     # TODO: change verification to complete tex block(s) for
15     ↪ appendices.
16     # TODO: Also verify related boolean and if statement creations.
17     determining_overleaf_home_line = (
18         r"\def\overleafhome{/tmp}% change as appropriate"
19     )
20     begin_apendices_line = "\\begin{appendices}"
21     print(f"determining_overleaf_home_line={
22         ↪ determining_overleaf_home_line}")
23     print(f"begin_apendices_line={begin_apendices_line}")
24
25     if not file_contains(
26         path_to_main_latex_file, determining_overleaf_home_line
27     ):
28         raise Exception(
29             f"Error, {path_to_main_latex_file} does not contain:\n\n"
30             + f"{determining_overleaf_home_line}\n\n so this Python
31             ↪ code "
32             + "cannot export the code as latex appendices."
33         )
34
35     if not file_contains(
36         path_to_main_latex_file, determining_overleaf_home_line
37     ):
38         raise Exception(
39             f"Error, {path_to_main_latex_file} does not contain:\n\n"
40             + f"{begin_apendices_line}\n\n so this Python code cannot
41             ↪ export"
42             + "the code as latex appendices."
43         )
44
45 )
```

L Appendix src/export_data/latex_compile.py

```
1  """Compiles the latex report using the compile script."""
2
3
4  from src.export_data.helper_bash_commands import run_a_bash_command
5
6
7  def compile_latex(await_compilation, verbose):
8      """Compiles the LaTeX report of this project using its compile
9          ↪ script.
10
11      :param await_compilation: Make python wait until the PlantUML
12          ↪ compilation
13      is completed.
14      :param project_name: The name of the project that is being
15          ↪ executed/ran.
16      :param verbose: True, ensures compilation output is printed to
17          ↪ terminal,
18      False means compilation is silent.
19
20      Returns:
21          Nothing.
22
23      Raises:
24          Nothing.
25      """
26
27      # Ensure compile script is runnable.
28      bash_make_compile_script_runnable_command = (
29          "chmod +x latex/compile_script.sh"
30      )
31      run_a_bash_command(
32          await_compilation, bash_make_compile_script_runnable_command,
33          ↪ verbose
34      )
35
36      # Run latex compilation script to compile latex project.
37      bash_compilation_command = "latex/compile_script.sh"
38      run_a_bash_command(await_compilation, bash_compilation_command,
39          ↪ verbose)
```

M Appendix src/export_data/latex_export_code.py

```
1  """Exports code to latex appendices."""
2  import os
3
4  from src.export_data.helper_dir_file_edit import (
5      get_all_files_in_dir_and_child_dirs,
6      get_filepaths_in_dir,
7      remove_all_auto_generated_appendices,
8      sort_filepaths_by_filename,
9  )
10 from src.export_data.helper_tex_editing import (
11     create_appendix_manager_files,
12     export_python_export_code,
13     export_python_project_code,
14 )
15 from src.export_data.helper_tex_reading import (
16     verify_latex_supports_auto_generated_appendices,
17 )
18
19
20 def export_code_to_latex(hd, include_export_code):
21     """This function exports the python files and compiled pdfs of
22         ↪ jupyter
23     notebooks into the latex of the same project number. First it
24         ↪ scans which
25     appendices (without code, without notebooks) are already manually
26         ↪ included
27     in the main latex code. Next, all appendices that contain the
28         ↪ python code
29     are either found or created in the following order: First, the
30         ↪ __main__.py
31     file is included, followed by the main.py file, followed by all
32         ↪ python code
33     files in alphabetic order. After this, all the pdfs of the
34         ↪ compiled
35     notebooks are added in alphabetic order of filename. This order
36         ↪ of
37     appendices is overwritten in the main tex file.
38
39     :param main_latex_filename: Name of the main latex document of
40         ↪ this project
41     number.
42     :param project_name: The name of the project that is being
43         ↪ executed/ran.
44     The number indicating which project this code pertains to.
45     """
46     script_dir = get_script_dir()
47     latex_dir = script_dir + "../..../latex/"
48     path_to_main_latex_file = f"{latex_dir}{hd.main_latex_filename}"
49     root_dir = script_dir + "../..../"
50     normalised_root_dir = os.path.normpath(root_dir)
51     src_dir = script_dir + "../.."
52
53     # Verify the latex file supports auto-generated python appendices
54     ↪ .
55     verify_latex_supports_auto_generated_appendices(
56         ↪ path_to_main_latex_file)
57
58     # Get paths to files containing project python code.
59     python_project_code_filepaths = get_filepaths_in_dir(
60         "py", src_dir, ["__init__.py"]
61     )
```

```

49 )
50
51 get_compiled_notebook_paths(script_dir)
52 print(f"python_project_code_filepaths={
53     ↪ python_project_code_filepaths}")
54
55 # Get paths to the files containing the latex export code
56 if include_export_code:
57     python_export_code_filepaths = get_filepaths_in_dir(
58         "py", script_dir, ["__init__.py"]
59     )
60
61 remove_all_auto_generated_appendices(hd)
62
63 # Create appendix file # ensure they are also deleted at the
64 ↪ start of every
65 # run.
66 create_appendix_manager_files(hd)
67
68 # TODO: Sort main files.
69 export_python_project_code(
70     hd,
71     normalised_root_dir,
72     sort_filepaths_by_filename(python_project_code_filepaths),
73 )
74 if include_export_code:
75     export_python_export_code(
76         hd,
77         normalised_root_dir,
78         sort_filepaths_by_filename(python_export_code_filepaths),
79     )
80
81 def get_compiled_notebook_paths(script_dir):
82     """Returns the list of jupyter notebook filepaths that were
83     ↪ compiled
84     successfully and that are included in the same dias this script (
85     ↪ the src
86     directory).
87
88     :param script_dir: absolute path of this file.
89     """
90     notebook_filepaths = get_all_files_in_dir_and_child_dirs(
91         ".ipynb", script_dir
92     )
93     compiled_notebook_filepaths = []
94
95     # check if the jupyter notebooks were compiled
96     for notebook_filepath in notebook_filepaths:
97         # swap file extension
98         notebook_filepath = notebook_filepath.replace(".ipynb", ".pdf
99         ↪ ")
100
101         # check if file exists
102         if os.path.isfile(notebook_filepath):
103             compiled_notebook_filepaths.append(notebook_filepath)
104     return compiled_notebook_filepaths
105
106 def get_script_dir():

```

```
105     """returns the directory of this script regardless of from which
      ↪ level the
106     code is executed."""
107     return os.path.dirname(__file__)
```

N Appendix src/export_data/plantuml_compile.py

```
1  """This script automatically compiles the text files representing a
   ↪ PlantUML.
2
3  # diagram into an actual figure.
4
5  # To compile locally manually:
6  # pip install plantuml
7  # export PLANTUML_LIMIT_SIZE=8192
8  # java -jar plantuml.jar -verbose sequenceDiagram.txt
9  """
10
11 import os
12 import subprocess # nsec
13 from os.path import abspath
14
15 from src.export_data.helper_dir_file_edit import (
16     get_dir_filelist_based_on_extension,
17 )
18 from src.export_data.plantuml_get_package import got_java_file
19
20
21 def compile_diagrams_in_dir_relative_to_root(
22     await_compilation,
23     extension,
24     jar_path_relative_from_root,
25     input_dir_relative_to_root,
26     verbose,
27 ):
28     """Loops through the files in a directory and exports them to the
   ↪ latex.
29
30     /Images directory.
31
32     Args:
33     :param await_compilation: Make python wait until the PlantUML
   ↪ compilation
34     is completed. param extension: The filetype of the text file that
   ↪ is
35     converted to image.
36     :param jar_path_relative_from_root: The path as seen from root
   ↪ towards the
37     PlantUML .jar file that compiles .uml files to .png files.
38     :param verbose: True, ensures compilation output is printed to
   ↪ terminal,
39     False means compilation is silent.
40     :param extension: The file extension that is used/searched in
   ↪ this
41     function.
42     :param input_dir_relative_to_root: The directory as seen from
   ↪ root
43     containing files that are modified in this function.
44
45     Returns:
46         Nothing
47
48     Raises:
49         Nothing
50     """
51     # Verify the PlantUML .jar file is gotten.
52     got_java_file(jar_path_relative_from_root)
```

```

53 diagram_text_filenames = get_dir_filelist_based_on_extension(
54     input_dir_relative_to_root, extension
55 )
56
57
58 for diagram_text_filename in diagram_text_filenames:
59     diagram_text_filepath_relative_from_root = (
60         f"{input_dir_relative_to_root}/{diagram_text_filename}"
61     )
62
63     execute_diagram_compilation_command(
64         await_compilation,
65         jar_path_relative_from_root,
66         diagram_text_filepath_relative_from_root,
67         verbose,
68     )
69
70
71 def execute_diagram_compilation_command(
72     await_compilation,
73     jar_path_relative_from_root,
74     relative_filepath_from_root,
75     verbose,
76 ):
77     """Compiles a .uml/text file containing a PlantUML diagram to a .
78         ↪ png image
79         using the PlantUML .jar file.
80
81     Args:
82     :param await_compilation: Make python wait until the PlantUML
83         ↪ compilation
84         is completed. param extension: The filetype of the text file that
85         ↪ is
86         converted to image.
87     :param jar_path_relative_from_root: The path as seen from root
88         ↪ towards the
89         PlantUML .jar file that compiles .uml files to .png files.
90     :param verbose: True, ensures compilation output is printed to
91         ↪ terminal,
92         False means compilation is silent.
93     :param input_dir_relative_to_root: The directory as seen from
94         ↪ root
95         containing files that are modified in this function.
96     Returns:
97         Nothing
98
99     Raises:
100         Nothing
101     """
102     # Verify the files required for compilation exist, and convert
103     ↪ the paths
104     # into absolute filepaths.
105     (
106         abs_diagram_filepath,
107         abs_jar_path,
108     ) = assert_diagram_compilation_requirements(
109         jar_path_relative_from_root,
110         relative_filepath_from_root,
111     )
112
113     # Generate command to compile the PlantUML diagram locally.
114     print(

```

```

108         f"abs_jar_path={abs_jar_path},"
109         + f" abs_diagram_filepath={abs_diagram_filepath}\\n\\n"
110     )
111     bash_diagram_compilation_command = (
112         f"java -jar {abs_jar_path} -verbose {abs_diagram_filepath}"
113     )
114     print(
115         f"bash_diagram_compilation_command={
116             ↪ bash_diagram_compilation_command}"
117     )
118     # Generate global variable specifying max image width in pixels,
119     ↪ in the
120     # shell that compiles.
121     os.environ["PLANTUML_LIMIT_SIZE"] = "16192"
122
123     # Perform PlantUML compilation locally.
124     if await_compilation:
125         if verbose:
126             subprocess.call(
127                 bash_diagram_compilation_command, shell=True # nsec
128                 ↪ B602
129             )
130         else:
131             subprocess.call(
132                 bash_diagram_compilation_command,
133                 shell=True, # nsec
134                 stderr=subprocess.DEVNULL,
135                 stdout=subprocess.DEVNULL,
136             ) # nsec
137     else:
138         if verbose:
139             with subprocess.Popen(
140                 bash_diagram_compilation_command,
141                 shell=True, # nsec
142                 stdout=subprocess.PIPE,
143                 stderr=subprocess.STDOUT,
144             ) as process:
145                 get_output_of_bash_command(process)
146         else:
147             with subprocess.Popen(
148                 bash_diagram_compilation_command,
149                 shell=True, # nsec
150                 stderr=subprocess.DEVNULL,
151                 stdout=subprocess.DEVNULL,
152             ) as process:
153                 get_output_of_bash_command(process)
154
155     def get_output_of_bash_command(process, verbose=True):
156         """Returns the output of a bash command."""
157         stdout, stderr = process.communicate()
158         result = stdout.decode("utf-8")
159         if verbose:
160             print(f"result={result}")
161             print(f"stderr={stderr}")
162         return result
163
164     def assert_diagram_compilation_requirements(
165         jar_path_relative_from_root,
166         relative_filepath_from_root,
167     ):

```



```

167 """Asserts that the PlantUML .jar file used for compilation
    ↳ exists, and
168 that the diagram file with the .uml content for the diagram
    ↳ exists. Throws
169 an error if either of two is missing.
170
171 :param relative_filepath_from_root: Relative filepath as seen
    ↳ from root of
172 file that is used in this function.
173 :param output_dir_from_root: Relative directory as seen from root
    ↳ , to which
174 files are outputted.
175 :param jar_path_relative_from_root: The path as seen from root
    ↳ towards the
176 PlantUML .jar file that compiles .uml files to .png files.
177
178 Returns:
179     Nothing
180
181 Raises:
182     Exception if PlantUML .jar file used to compile the .uml to .
    ↳ png files
183     is missing.
184     Exception if the file with the .uml content is missing.
185 """
186 abs_diagram_filepath = abspath(relative_filepath_from_root)
187 abs_jar_path = abspath(jar_path_relative_from_root)
188 if os.path.isfile(abs_diagram_filepath):
189     if os.path.isfile(abs_jar_path):
190         return abs_diagram_filepath, abs_jar_path
191     raise Exception(
192         f"The input diagram file:{abs_diagram_filepath} doesn't
    ↳ exist."
193     )
194 raise Exception(f"The input jar file:{abs_jar_path} does not
    ↳ exist.")

```

O Appendix src/export_data/plantuml_generate.py

```
1  """This script generates PlantUML diagrams and outputs them as .uml
   ↪ files."""
2
3  import os
4  from os.path import abspath
5
6  from src.export_data.helper_dir_file_edit import (
7      create_dir_relative_to_root_if_not_exists,
8      dir_relative_to_root_exists,
9  )
10
11
12  def generate_all_dynamic_diagrams(output_dir_relative_to_root):
13      """Manages the generation of all the diagrams created in this
   ↪ file.
14
15      Args:
16      :param output_dir_relative_to_root: Relative path as seen from
   ↪ the root dir
17      of this project, to which modified files are outputted.
18
19      Returns:
20          Nothing
21
22      Raises:
23          """
24      # Create a example Gantt output file.
25      filename_one, lines_one = create_trivial_gantt("trivial_gantt.uml
   ↪ ")
26      output_diagram_text_file(
27          filename_one, lines_one, output_dir_relative_to_root
28      )
29
30      # Create another example Gantt output file.
31      filename_two, lines_two = create_trivial_gantt("
   ↪ another_trivial_gantt.uml")
32      output_diagram_text_file(
33          filename_two,
34          lines_two,
35          output_dir_relative_to_root,
36      )
37
38
39  def output_diagram_text_file(filename, lines,
   ↪ output_dir_relative_to_root):
40      """Gets the filename and lines of an PlantUML diagram, and writes
   ↪ these to
41      a file at the relative output path.
42
43      Args:
44      :param filename: The filename of the PlantUML Gantt file that is
   ↪ being
45      created.
46      :param lines: The lines of the Gantt chart PlantUML code that is
   ↪ being
47      written to file.
48      :param output_dir_relative_to_root: Relative path as seen from
   ↪ the root
49      dir of this project, to which modified files are outputted.
50
```

```

51 Returns:
52     Nothing
53
54 Raises:
55     Exception if input file does not exist.
56     """
57 abs_filepath = abspath(f"{output_dir_relative_to_root}/{filename}"
58     ↪ ")
59
60 # Ensure output directory is created.
61 create_dir_relative_to_root_if_not_exists(
62     ↪ output_dir_relative_to_root)
63 if not dir_relative_to_root_exists(output_dir_relative_to_root):
64     raise Exception(
65         "Error, the output directory relative to root:"
66         + f"{output_dir_relative_to_root} does not exist."
67     )
68
69 # Delete output file if it already exists.
70 if os.path.exists(abs_filepath):
71     os.remove(abs_filepath)
72
73 # Write lines to file.
74 with open(abs_filepath, "w", encoding="utf-8") as f:
75     for line in lines:
76         f.write(line)
77     f.close()
78
79 # Assert output file exists.
80 if not os.path.isfile(abs_filepath):
81     raise Exception(f"The input file:{abs_filepath} does not
82     ↪ exist.")
83
84 def create_trivial_gantt(filename):
85     """Creates a trivial Gantt chart.
86
87     Args:
88         :param filename: The filename of the PlantUML diagram file that
89             ↪ is being
90             created.
91
92     Returns:
93         The filename of the PlantUML diagram, and the lines of the
94         ↪ uml content
95         of the diagram
96
97     Raises:
98         Nothing
99     """
100     lines = []
101     lines.append("@startuml\n")
102     lines.append("[Prototype design] lasts 15 days\n")
103     lines.append("[Test prototype] lasts 10 days\n")
104     lines.append("\n")
105     lines.append("Project starts 2020-07-01\n")
106     lines.append("[Prototype design] starts 2020-07-01\n")
107     lines.append("[Test prototype] starts 2020-07-16\n")
108     lines.append("@enduml\n")
109     return filename, lines

```

```

108 def create_another_trivial_gantt(filename):
109     """Creates a trivial Gantt chart.
110
111     :param filename: The filename of the PlantUML Gantt file that is
112         ↪ being
113         created.
114
115     Returns:
116         The filename of the PlantUML diagram, and the lines of the
117         ↪ uml content
118         of the diagram
119
120     Raises:
121         Nothing
122     """
123     lines = []
124     lines.append("@startuml\n")
125     lines.append("[EXAMPLE SENTENCE] lasts 15 days\n")
126     lines.append("[Test prototype] lasts 10 days\n")
127     lines.append("\n")
128     lines.append("Project starts 2022-07-01\n")
129     lines.append("[Prototype design] starts 2022-07-01\n")
130     lines.append("[Test prototype] starts 2022-07-16\n")
131     lines.append("@enduml\n")
132
133     return filename, lines

```

P Appendix src/export_data/plantuml_get_package.py

```
1  """Downloads the PlantUML package if it does not yet exist."""
2  import os
3  import subprocess  # nsec
4
5  import requests
6
7
8  def check_if_java_file_exists(relative_filepath):
9      """Safe check to see if file exists or not.
10
11      Args:
12      :param relative_filepath: Path as seen from root towards a file.
13
14      Returns:
15          True if a file exists.
16          False if a file does not exists.
17
18      Raises:
19          Nothing
20      """
21
22      return os.path.isfile(relative_filepath)
23
24
25  def got_java_file(relative_filepath):
26      """Asserts if PlantUML .jar file exists. Tries to download is one
27      ↪ time if
28      it does not exist at the start of the function.
29
30      Args:
31      :param relative_filepath: Path as seen from root towards a file.
32
33      Returns:
34          True if a file exists.
35
36      Raises:
37          Exception if the PlantUML .jar file does not exist after
38          ↪ downloading
39          it.
40      """
41      # Check if the jar file exists, curl it if not.
42      if not check_if_java_file_exists(relative_filepath):
43          # The java file is not found, curl it
44          request_file(
45              "https://sourceforge.net/projects/"
46              + "plantuml/files/plantuml.jar/download",
47              relative_filepath,
48          )
49      # Check if the jar file exists after curling it. Raise Exception
50      ↪ if it is
51      # not found after curling.
52      if not check_if_java_file_exists(relative_filepath):
53          raise Exception(f"File:{relative_filepath} is not accessible"
54          ↪ )
55      print("Got the PlangUML Java file.")
56      return True
57
58
59  def request_file(url, output_filepath):
60      """Downloads a file or file content.
```

```

57
58 Args:
59 :param url: Url towards a file that will be downloaded.
60 :param relative_filepath: The path as seen from the root of this
    ↪ directory,
61 in which files are outputted.
62
63 Returns:
64     Nothing
65
66 Raises:
67     Nothing
68     """
69
70 # Request the file in the url
71 response = requests.get(url, timeout=20) # seconds
72 with open(output_filepath, "wb", encoding="utf-8") as f:
73     f.write(response.content)
74
75
76 def run_bash_command(bashCommand):
77     """Unused method. TODO: verify it is unused and delete it.
78
79     :param bashCommand: A string containing a bash command that
80     can be executed.
81     """
82     # Verbose call.
83     # subprocess.Popen(bashCommand, shell=True)
84     # Silent call.
85     # subprocess.Popen(bashCommand, shell=True, stderr=subprocess.
    ↪ DEVNULL,
86     # stdout=subprocess.DEVNULL)
87
88     # Await completion:
89     # Verbose call.
90     subprocess.call(bashCommand, shell=True) # nsec
91     # Silent call.
92     # subprocess.call(bashCommand, shell=True, stderr=subprocess.
    ↪ DEVNULL,
93     # stdout=subprocess.DEVNULL)

```

Q Appendix src/export_data/plantuml_to_tex.py

```
1  """Exports the generated PlantUML diagrams to the latex Images
   ↳ directory."""
2  import os.path
3  import shutil
4
5  from src.export_data.helper_dir_file_edit import (
6      create_dir_relative_to_root_if_not_exists,
7      get_dir_filelist_based_on_extension,
8  )
9
10
11 def export_diagrams_to_latex(
12     input_dir_relative_to_root, extension,
13     ↳ output_dir_relative_to_root
14 ):
15     """Loops through the files in a directory and exports them to the
16     ↳ latex.
17
18     /Images directory.
19
20     :param dir: The directory in which the Gantt charts are being
21     ↳ searched.
22     :param extension: The file extension that is used/searched in
23     ↳ this
24     function. The filetypes that are being exported.
25     :param input_dir_relative_to_root: Relative path as seen from the
26     ↳ root dir
27     of this project, containing files that modified in this function.
28     :param output_dir_relative_to_root: Relative path as seen from
29     ↳ the root dir
30     of this project, to which modified files are outputted.
31     """
32     diagram_filenames = get_dir_filelist_based_on_extension(
33         input_dir_relative_to_root, extension
34     )
35     if len(diagram_filenames) > 0:
36         # Ensure output directory is created.
37         create_dir_relative_to_root_if_not_exists(
38             ↳ output_dir_relative_to_root)
39
40     for diagram_filename in diagram_filenames:
41         diagram_filepath_relative_from_root = (
42             f"{input_dir_relative_to_root}/{diagram_filename}"
43         )
44
45         export_gantt_to_latex(
46             diagram_filepath_relative_from_root,
47             ↳ output_dir_relative_to_root
48         )
49
50 def export_gantt_to_latex(
51     relative_filepath_from_root, output_dir_relative_to_root
52 ):
53     """Takes an input filepath and an output directory as input and
54     ↳ copies the
55     file towards the output directory.
56
57     :param relative_filepath_from_root: param
58     ↳ output_dir_relative_to_root:
```

```

50 :param output_dir_relative_to_root: Relative path as seen from
51     ↳ the root dir
52 of this project, to which modified files are outputted.
53
54 Returns:
55     Nothing.
56
57 Raises:
58     Exception if the output directory does not exist.
59     Exception if the input file is not found.
60 """
61 if os.path.isfile(relative_filepath_from_root):
62     if os.path.isdir(output_dir_relative_to_root):
63         shutil.copy(
64             relative_filepath_from_root,
65             ↳ output_dir_relative_to_root
66         )
67     else:
68         raise Exception(
69             f"The output directory:{output_dir_relative_to_root}
70             ↳ does"
71             + " not exist."
72         )
73 else:
74     raise Exception(
75         f"The input file:{relative_filepath_from_root} does not
76         ↳ exist."
77     )

```
