Social Investment Agency
Toi Hau Tāngata

Accelerated Data Analysis & Pipeline Toolkit (ADAPT)

User Guide

March 2026

Te Kāwanatanga o Aotearoa
New Zealand Government

Author

Simon Anastasiadis

Acknowledgements

Wian Lusse, Dan Young, Charlotte Rose, and Andrew Webber, at SIA, for their testing and feedback during development.

Data Lab and Output Checking team, at Stats NZ, without whom we would not have access to the IDI or have the toolkit available outside the data lab to share.

Creative Commons Licence

Liability

Citation

Social Investment Agency 2026. Accelerated Data Analysis & Pipeline Toolkit (ADAPT). Wellington, New Zealand.

ISBN 978-1-997329-00-8 (online)

Published in March 2026 by
Social Investment Agency
Wellington, New Zealand

Contents

Accelerating analytic delivery

Data is a valuable resource for creating knowledge and informing decisions. Its usefulness depends not just upon the content of the data that is collected but also the process by which it can be transformed into insight. With increasing data volumes and requests for insights, effective delivery of analytics requires dependable and scalable approaches.

In response to this challenge, staff at the Social Investment Agency (SIA) have developed a set of tools to accelerate the creation of new insights. These tools provide a standardised and automated way to conduct a range of common tasks. Together they have provided an order of magnitude speed-up to our delivery – for example, projects that might once have taken months can now be delivered in weeks, and projects that might once have taken weeks can now be delivered in days.

These tools not only reduce the effort to deliver analytic results, they also produce higher quality insights. Quality requires that insights are consistent, transparent, and robust. By imposing structure on a project, the toolkit simplifies repeatable analyses, clarifies process flow, and reduces errors.

We call this set of tools the Accelerated Data Analysis & Pipeline Toolkit (ADAPT). ADAPT is now available for other researchers and analysts to use. This document introduces the toolkit and serves as a guide for users.

Designed for the IDI, useful beyond

We developed this toolkit in the context of SIA's research with the Integrated Data Infrastructure (IDI).[1] As such, parts of its design have been shaped by the specific characteristics of the secure IDI environment – the need to apply confidentiality rules to summarised output, being the most obvious.

However, as many projects involve fetching data from different sources, creating summary results, and automating processes, ADAPT will have applications beyond the IDI. We have designed the toolkit with flexibility so that each component tool can be used by itself and can be given analysis-specific instructions. This supports analysts to use only the tools they find useful and to adapt them to their own projects.

Four component tools

ADAPT contains four component tools that share a range of common features and functionality. They are bundled together as a toolkit for ease of use (and to avoid the duplication that would arise from distributing and maintaining four separate tools).

[1] The IDI is a large research database containing deidentified microdata about people and households. For more information see: https://www.stats.govt.nz/integrated-data/integrated-data-infrastructure/

The tools within ADAPT are as follows:

- The **Summarise tool** simplifies the creation of summary results. For example, producing counts, totals, and means from unit record data. It allows for such summaries to be subset or filtered by any number or combination of variables.
- The **Confidentialise tool** follows a user specification to apply confidentiality rules to summarised results. This streamlines the process of ensuring output is safe to release from the secure IDI environment.
- The **Assembly tool** fetches data from a range of source tables, drawing it together to produce a rectangular table. By standardising this as a single stage, it makes it easy to trace data back to its origins and to recreate the table if source data changes.
- The **Pipeline tool** automates the running of entire analyses. It contains scheduling options and robust error handling so that code can be run without continuous staff monitoring.

We have listed these tools in their recommended learning order. This is the same order that they are documented below. This guides a new user through the component tools in order of increasing complexity and consequence. However, when designing a new analysis, it is more common for experienced staff who use the entire toolkit to work with the pipeline and assembly tools before the summarise and confidentialise tools.

Analytic delivery framework

Implicit within the design of ADAPT is the framework by which SIA researchers arrange their IDI projects. While not essential for the use of the toolkit, it provides another way to understand the relationships between the component tools and how they combine to support end-to-end delivery.

Figure 1: Analytic framework with tools overlaid

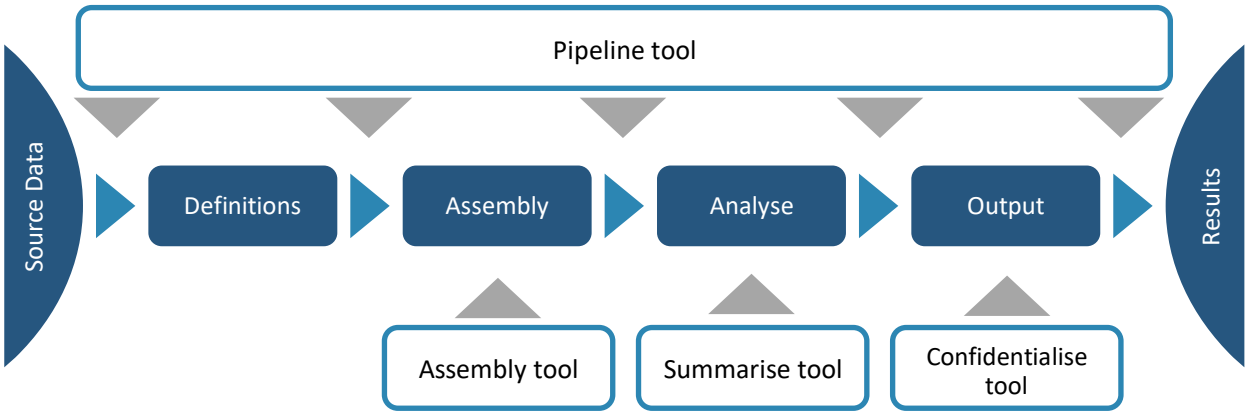


Figure 1 outlines our framework showing the four key stages of a project from source data to results. These stages are:

1. **Definitions** – Identifies and extracts from source data the concepts or measures that are required for analysis.
2. **Assembly** – Combines all the individual pieces of data together in a single rectangular table ready for analysis. This table becomes the single source of truth for subsequent analysis and is referred to as the master table to reflect this.

3. **Analyse** – Conducts whatever analysis the project requires. This often includes data cleaning, model fitting, and summarisation of results.
4. **Output** – Contains any steps required to ensure results are suitable for delivery. Applying data standards, validation, and privacy protections all fit within this stage.

Figure 1 also provides an indication of where each of the component tools map across the stages of a project. The assembly, summarise, and confidentialise tools apply to a single stage, while the pipeline tool applies to the execution of all code within the project.

We do not have a specific tool for the definitions stage of our projects. Instead, we use two resources: a library of IDI concept definitions drawn from previous SIA projects[2] and the Code Modules developed by the IDI community under a Stats NZ-led initiative.[3]

Surpassing our previous tools

At the end of 2020, the SIA (then the Social Wellbeing Agency) published the Dataset Assembly Tool. Summarisation and confidentialisation tools were later developed and bundled together with the assembly tool. This bundle became the core tools for several of SIA's subsequent IDI projects.

ADAPT is the successor to this previous bundle of tools. It combines the best features of our original tools with more recent innovations and developments by SIA staff. The result is a step improvement in tool quality. ADAPT has seen rapid adoption across SIA's analytic teams, replacing the previous bundle as the basis for all SIA's routine analytics.

Control files facilitate documentation and execution

Central to the design of ADAPT is the role of control files for running each tool. A control file is an Excel or CSV file with a specific structure that contains the instructions for a tool to execute. When any tool is run, it first confirms that the instructions in its control file are valid before executing the action described within it.

While identical behaviour can be accomplished via direct programming, part of the strength of the toolkit is the way the control files define a standardised structure for giving these instructions. This makes the control files a form of documentation: capturing just the details that might change between steps and presenting them in a tabular format for ease of editing or review.

The control files also make ADAPT accessible to users regardless of their preferred programming language. While the toolkit is written in R and provided as an R package, no R knowledge is needed to create or modify a control file – this can all be done in Excel (or software with equivalent

[2] https://github.com/nz-social-investment-agency/definitions_library

[3] https://www.stats.govt.nz/integrated-data/code-modules-initiative/

functionality). As a result, users are mostly insulated from R – we provide templates with the toolkit, so users only ever need to provide simple text inputs like the names of their control files.

Tested for reliability

Uncertainty about whether a tool is reliable or error free is a common barrier to adoption. Throughout its development ADAPT has been subject to significant testing. The central role the toolkit now plays in our analytic product pipelines is testament to how confident SIA is of its reliability as a result of this testing.

The main approach used for testing the reliability of ADAPT is automated testing. For each part of the toolkit these tests specify example input and expected output. The tests then confirm that when the tool is applied to the example input it produces the expected output.

In total ADAPT has over 700 automated tests. These tests are run every time the toolkit is updated to ensure everything works as expected. Each time a new bug is found, or new feature is added, the tests are expanded to check the bug no longer occurs or that the new feature works as designed. In total, just over half the code for the R package is automated tests – which appears consistent with professionally developed R packages.

Reliability requires not just that a tool is error free, but also that a user understands what inputs to provide to get the results they want. For this purpose, ADAPT includes at least 15 worked examples and templates as guidance resources. The worked examples include input, user instructions, and the expected results and each is checked by the automated tests to ensure that it is correct.

In addition, each tool in the toolkit has inbuilt validation of user input. This validation checks whether a user's input can be executed by the corresponding tool and provides corrective instructions if the input is invalid. Between the worked examples, templates, and tool validation, users of ADAPT have many ways to be confident of what the toolkit does and how it can be used.

Getting started

The toolkit has been designed to minimise the effort required for new users to get started. We do this by providing installation instructions and a range of learning materials.

Installation is straightforward

Like any piece of software, the toolkit must be installed before it can be used. As ADAPT is provided as an R package, users must have R installed. We recommend a recent version of R with RStudio as the development environment.

To install the toolkit, you need a copy of the package file in a folder that is accessible from R. The latest public version of the package can be downloaded from on our GitHub page[4]: this is the ADAPT_*.tar.gz file within the ADAPT repository.

Once you have a copy of the package, Code block 1 provides the fastest way to install ADAPT. This uses the remotes package to automate installation of all its dependencies. When you execute Code block 1, it will open a file select window and prompt you to navigate to the package (*.tar.gz) file for installation.

Code block 1: install ADAPT and dependencies in R

```
# installation helper
if(!"remotes" %in% installed.packages()){
  install.packages("remotes")
}

# locate package file
package_path = rstudioapi::selectFile(caption = "Select R package",
filter = "Packages (*.gz)")

# install
remotes::install_local(path = package_path)
```

Should you need to uninstall the ADAPT package – perhaps to reinstall an updated version – this can be done using Code block 2. When uninstalling, you may need to approve a pop-up request to restart R.

Code block 2: uninstall ADAPT package in R

```
# uninstall
if("ADAPT" %in% installed.packages()){
  remove.packages("ADAPT")
}
```

[4] https://github.com/nz-social-investment-agency

Built-in learning material

In addition to this user guide, we highlight three other resources to assist users of ADAPT: worked examples, command list, and in-built R help. All three require that the package is already installed, as per the instructions in the previous section.

Code block 3 demonstrates how to access the worked examples and control file templates: Running the provide-example function with no arguments (empty brackets) will cause the toolkit to return a list of available examples. Running the provide example function with the name of an available example and a folder, will cause the toolkit to copy the example to this folder and return a list of the new files that are available.

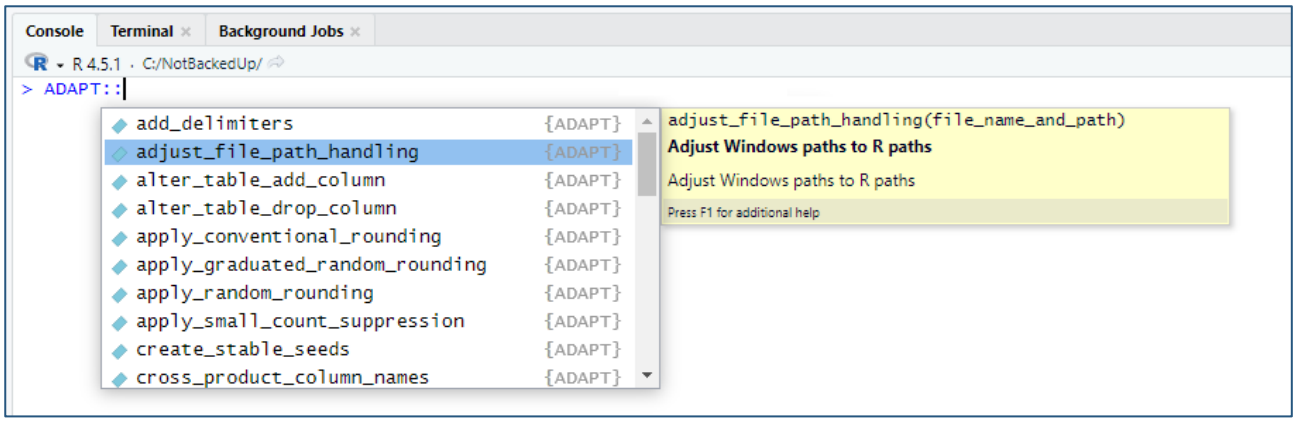
Code block 3: Provide worked examples and control file templates in R

```
ADAPT::provide_example()

# Results:
#
# Available examples:
# assembly_control_file
# assembly_control_file_w_metadata
# ...
# summary_wide_worked_example

ADAPT::provide_example("assembly_control_file_w_metadata",
"C:/Myfolder/metadata example")

# Results:
#
# [1] "C:/Myfolder/metadata example/control_file.xlsx"
```

Note that providing a folder is optional. If no folder is provided, then R writes to its current working directory. R requires forward slashes `/` or double backslashes `\\` when providing folder paths. Single backslashes `\` will cause errors. Also note that this copying overrides existing files, so the safest option is to use an empty or temporary folder for this purpose.

Code block 4 demonstrates how to display a list of commands for ADAPT. Within RStudio, type `ADAPT::` at the console. Upon typing the second colon, RStudio should provide a list of functions available with the package. This list can be scrolled through, and each item can be moused over to provide some additional information.

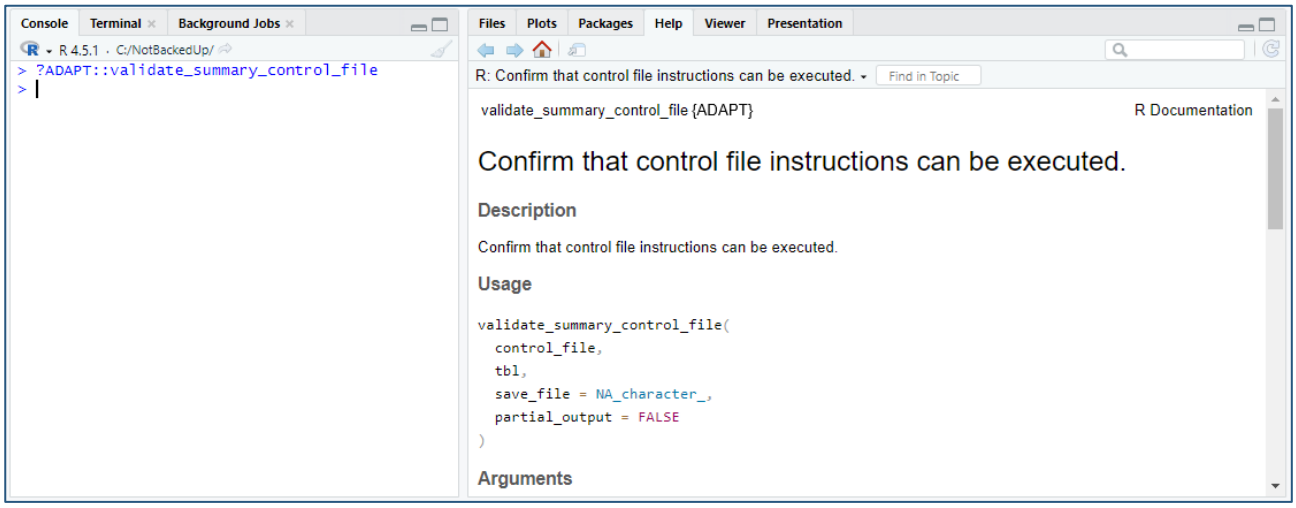
Code block 4: Command list in R



In some cases, the command list may not show until the tab button is pushed. If R is busy executing other commands, then the command list may not appear until these are finished. If the command list disappears, try deleting and retyping the second colon.

Code block 5 demonstrates how to access the in-built R help for ADAPT commands. Every function in the toolkit has help documentation written. This can be accessed by placing a question-mark before the function.

Code block 5: In-built R help



The in-built documentation includes an overview of each function, its inputs, the output it generates, and details of its workings. The alternative to `?` demonstrated in Code block 5, is `??`. Used in the same way, a double-question-mark causes R to search for all help pages containing the provided text. For example, `??control_file` gives a list of eight functions that include the text 'control_file' in their name.

An overview for each tool

The following sections introduce each of the four tools in ADAPT. They cover the core functionality relevant to starting users. For more detailed guidance, including customisation and additional options for use, we recommend the worked examples or in-built R help.

The summarise tool

The summarise tool exists to simplify and standardise the process of producing multiple summaries from a table. For example, we might wish to count the number of people by several different types of geographic area, and by several different types of demographic attributes. The `run_summary` function in the toolkit starts from a data object, often a tidy master table in SQL, and applies the instructions in its control file to produce summarised results.

Control file is one row per summary

For each row in the control file, the summary tool produces a summary of the data table according to the specified instructions. The accepted columns for the control file are:

- **Enabled** – A optional true/false column. If this column is included, then rows set to false will be omitted.
- **File** – The file path and name for where the summary output should be saved. Output can be divided across multiple files. Files and folders will be created if they do not exist, and overwritten if they do exist.
- **Label** – A free text column allowing you to include arbitrary text in your output. A common use case for this is providing a description of the summary.
- **Group** – The name of a column in the data table. When the summary for the row is produced, the output is grouped by all group columns.
- **Count** – The name of a column in the data table. The summary will count the number of non-missing values in the column.
- **Distinct** – The name of a column in the data table. The summary will count the number of distinct values in the provided column.
- **Sum** – The name of a column in the data table. The summary calculates the sum total of the column, during validation the tool will error if this column is not numeric.
- **Stddev** – The name of a column in the data table. The summary calculates the standard deviation of the column.
- **Entity** – Similar to Distinct but also checks for columns with `*__min` and `*__max` suffixes. If these exist it takes a distinct over the union of these columns, otherwise it takes the distinct over the provided column. Designed for counting entities for IDI output rules.
- **Seed** – Similar to Sum but takes remainder by 100 first. Intended to be applied to ID columns to generate a value that can serve as a seed for consistent rounding.
- **Where** – A dynamic formula in `{`curly brackets`}` to filter the input table before summarising.
- **Notes** – A free text column that is ignored and does not affect output. Intended for adding notes to the control file. Any other column names are also ignored but generate a warning.

Except for Enabled and File, you can have any number of columns of each type. When you have multiple columns of the same type, adding a numeric suffix (for example `GROUP01`) is recommended. The tool will automatically add numeric suffixes if any columns have identical names.

Some users might ask why there are options to produce a sum or standard deviation, but no option to produce a mean using the summary tool. This is because confidentiality rules for means require separating the numerator and the denominator. Where means are required, we recommend producing both the sum and count, and dividing one by the other after confidentialisation.

Output saved to disk with progress reporting

The summary tool produces output in two forms. First, it saves summarised results to disk as per the file column. Where multiple summaries are saved to the same file, these summaries will be appended one after the other. All results files produced by the same control file will have the same structure.

Second, it returns as an R data frame the contents of the control file with three new columns giving the start time, end time, and success or failure of each summary. The contents of this data frame are used to update the control file with progress reporting. This makes it straightforward to confirm how the tool performed and to assess its runtime.

Summary tool worked example

Table 1 and Table 2 provide a simplified example of a summary control file and its output. Table 1 shows a control file designed to produce six summaries of people and income: the same three for each of region and district.

Table 1: Example simplified summary control file

GROUP1	GROUP2	COUNT1	COUNT2	SUM	WHERE
region		person_id			
region		person_id	income	income	{ age > 15 }
region	income_band	person_id			{ age > 15 }
district		person_id			
district		person_id	income	income	{ age > 15 }
district	income_band	person_id			{ age > 15 }

Table 2: Example output from simplified summary control file

grplabel1	group1	grplabel2	group2	count1	count2	sum
region	1, 2, …, 9	NA	NA			
region	1, 2, …, 9	NA	NA			
region	1, 2, …, 9	age_band	16-25, 26-35, …			
district	01, 02, …, 40	NA	NA			
district	01, 02, …, 40	NA	NA			
district	01, 02, …, 40	age_band	16-25, 26-35, …			

Table 2 has been compacted for ease of display: the first row of our example would be nine rows in the tool results – one for each of the nine regions – and the last row in our example could be several-hundred rows in the tool results – one for each combination of district and age band. NA is

the notation used to indicate an empty or 'not available' cell. Note that the exact number of rows and the contents of the group1 and group2 columns would depend on the contents of the region, district, and age-band columns in the data table.

Aids to create summary control files

There is no limit on the size of a summarise control file. Our data lab staff have undertaken projects with upwards of 5000 summaries. Because it can be tedious to create large summary control files with many combinations, the toolkit includes two supporting functions to simplify this process:

- The **expand-compact-summary-groups** function allows for wildcards and pipes to be used to write in a single row what would otherwise require many rows in the summary control file. The function takes the compact form and expands it into a control file ready for use in the summary tool. For example:
  - The cell `region | district` would be expanded into two rows, one with `region` and the other with `district`.
  - The cell `income |+` would be expanded into two rows, one with `income` and the other with a blank (no grouping).
  - The cell `ethnicity_*` would be expanded into as many rows as there were columns that start with "ethnicity_" as a prefix (the equivalent option for suffixes also works).
- The **generate-combinations-df** function takes sets of variables and produces all interactions of them. This can be an effective way to produce the `GROUP` columns for a summary control file.

  For example: the two sets {region, district} and {income-band, NA} would produce four rows / combinations: each of region and district by themselves and with income-band.

While most users will have no need of these functions, where a project makes heavy use of the summarise tool, these offer significant potential. For users unfamiliar with either function, we recommend starting with the compact notation approach.

The confidentialise tool

The confidentialise tool exists to simplify the process of applying confidentialisation rules to summarised output. By handling the application of confidentiality rules, the tool allows users to focus on the decision of what rules to apply. The `run_confidential` function in the toolkit starts from a data frame containing summarised results and applies the instructions in its control file to confidentialise these results.

Control file is one row per rule

The control file for the confidentialise tool has one row for each rule or command that needs to be imposed. The first column in the control file always contains the list of commands with the remaining columns matching the column names of the summarised results table. The accepted commands for the first column and their use are:

- **Rename** – Text given in this row is used to rename columns. No checks are applied to ensure the text makes sensible column names: this is the users responsibility.
- **Drop** – Where this value is true, the column will be dropped from the output. The anticipated use case is to remove unneeded group-label columns.
- **Missing_to** – A numeric value to replace missing values in the column with. One use case for this command is to set default values for entity count columns when entity counts are not applicable, allowing for the suppression command to run across all rows in the results even where entity counts apply to only some rows.
- **Seed** – An optional command. If provided allows a column to be specified as the seeds for any random rounding. This ensure consistent rounding wherever the seed is identical. Designed to receive the seed summary type from the summary tool.
- **Round** – The type of rounding to apply. Accepted options are: RR3 and GRR (for random rounding and graduated random rounding), and CONV10, CONV100, and CONV1000 (for conventional rounding to base 10, 100, or 1000).
- **Suppress** – When to suppress a column in the results. Specified as an instruction of the form: `'column-name < value'`. Accepts input column names and renamed column names. Where there are multiple reasons data might be suppressed, add one condition per row.
- **Notes** – Allows for a row of notes that does not affect output.

When making a confidentialise control file, we often begin by copying the column headers from the results file to be confidentialised into the top row of the control file. Individual rules can then be specified under the results column they relate to.

Output adds confidentialised columns to input

The confidentialise tool outputs a table of results created by applying the specified rules to the summarised results table. Because it is often useful to have the original and confidentialised columns side-by-side, the tool avoids applying certain modifications in place. Instead, when

applying rounding or suppression, a second column is created with "conf_" as a prefix with rounding and suppression applied to this column.

By having raw or original columns alongside the confidentialised columns, the tool makes it easy to check that confidentialised results are safe. In practice, we take the output from the tool, delete any columns containing values that have not have confidentiality rules applied, delete any rows that are fully suppressed[5], and proceed with confidence the resulting file is safe for subsequent use.

Confidentialise tool worked example

Table 3 provides an example of a confidentialise control file. When applied to a summarised results table like Table 4, it creates a confidentialised results table like Table 5. All the results in these tables are fictional.

Table 3: Example confidentialise control file

COMMAND	grplabel1	group1	grplabel2	group2	count1	count2	sum
RENAME				age_ band	num_ people	num_w_ income	total_ income
DROP			TRUE				
ROUND					RR3	RR3	
SUPPRESS					count1 < 6		count1 < 20
SUPPRESS						count2 < 6	count2 < 20

Table 4: Example summarised results before applying confidentiality tool

grplabel1	group1	grplabel2	group2	count1	count2	sum
Region	01	NA	NA	802	229	134,000
Region	01	age_band	16-25	23	5	4,000
Region	01	age_band	26-35	44	12	7,400

Table 5: Example summarised results after applying confidentiality tool

grplabel1	group1	age_ band	num_ people	num_w_ income	total_ income	conf_num_ people	conf_num_w_ income	conf_total_ income
Region	01	NA	802	229	134,000	801	231	134,000
Region	01	16-25	23	5	4,000	24	NA	NA
Region	01	26-35	44	12	7,400	45	12	NA

Table 5 shows several key changes from Table 4. First, because the third column of Table 4 only included the text age-band, the drop and rename commands have been used to make this label text the column name. Second, all three numeric columns have been renamed. Fourth, a set of

[5] The prevents the presence or absence of a suppressed row in the data from being used to infer information about the raw results.

confidentialised columns have been added with rounded values and suppressed values replaced by ‘NA’ as their values are not available after confidentialisation.

Component functions available

For some applications, users may prefer to use the implementation of individual confidentiality rules separate from the overarching tool. The four functions that implement these rules are:

- The **apply-random-rounding** function. This defaults to applying random rounding to base 3 (RR3) but can be adjusted for other bases. It includes options for seeds to enforce consistent rounding, and for thresholds that restrict rounding (forcing an unrounded value above/below a threshold to remain above/below the threshold after rounding).
- The **apply-graduated-random-rounding** function. This applies graduated random rounding (GRR) where the base for the rounding is determined by the magnitude of the value being rounded. It includes the same options as the random rounding function.
- The **apply-conventional-rounding** function, which applies conventional rounding.
- The **apply-small-count-suppression** function. This replaces numeric values with missing values (NA) when comparison values are too small. It includes options for how missing comparison values should be treated.

While these separate functions do not handle every situation required by the confidentiality rules, they cover the vast majority of cases. Should new rules be required, these existing functions provide a template for further development.

The assembly tool

The assembly tool exists to accelerate the process of drawing data from a range of different sources together into a single table ready for analysis: the master table. The `run_assembly` function in the toolkit starts from an initialised master table and applies the instructions in its control file to add columns of measures.

The assembly tool has been designed with flexibility, to avoid prescribing how concept definitions must be designed. Recalling our analytic delivery framework, definitions are the way we move from source data to concepts that are useful for analysis. The approach we recommend in the IDI is to develop concept definitions for all observed people and time periods. We then rely on the assembly tool to filter to the people and time periods of interest. This reduces manual effort and makes definitions more reusable between analyses.

Control file is one row per measure for analysis

For each row in the control file, the assembly tool adds a measure column to the master table for use in analysis. The accepted columns for the control file are:

- **Enabled** – An optional but recommended column that allows for rows of the control file to be turned on and off. If this column is included, then rows set to false will not be executed.
- **Description** – An optional but recommended column. While this column has no effect during assembly, we recommend using it to record a user readable description of the intended measure. When used, this column allows for the assembly control file to be used as an initial data dictionary.
- **Population_uid** – The name of the identity/uid column in the master table to use for assembly. This is compared against the `Measure_uid` column from the `Measure_table` table to filter the concept definition to the people of interest.
- **Period_start** – The start of the time period being studied. This can be a column of the master table, a constant, or an SQL code snippet. Together with the `Period_end` column, this is compared against the `Measure_start` and `Measure_end` columns from the `Measure_table` to filter the concept definition to the time period(s) of interest.
- **Period_end** – The end of the time period being studied. This can be a column of the master table, a constant, or an SQL code snippet.
- **Measure_file** – The name of the SQL file that prepares the measure, if applicable. When used, this column allows for tracking the lineage of a measure.
- **Measure_table** – The name of the SQL table or view from which the measure is drawn.
- **Measure_uid** – The name of the identity/uid column in the measure table. Compared against the `Population_uid` column.
- **Measure_start** – The start of the measure event. This can be a column of the measure table, a constant, or an SQL code snippet.
- **Measure_end** – The end of the measure event. This can be a column of the measure table, a constant, or an SQL code snippet.

- **Measure_value** – The value that should be summarised to create a column in the master table, can be a column of the measure table, a constant, or an SQL code snippet.
- **Output_name** – The name of the output column. This should be delimited with `""` and unique within its column. For ease of subsequent use, it is best to use underscores instead of spaces in these names.
- **Output_method** – How the value from `Measure_value` should be summarised to create the column `Output_name` in the master table. The options for this column are listing in their own section below.
- **Output_type** – The SQL data type of the new column.
- **Notes** – A free text column that is ignored and does not affect output. Intended for adding notes to the control file. Any other column names are also ignored but generate a warning

Because many entries in an assembly control file can be of different types, the control file requires cells to be delimited to ensure clarity. SQL objects (like table and column names) should be delimited with `[]`, constants should be delimited with `""`, and dynamic input should be delimited with `{}`.

Initialise the master table with who and when

The assembly tool adds columns of measures to a master table. It does not create the master table. Before the tool is run, the master table must be initialised: created with some minimum contents.

Observe that the `Population_uid`, `Period_start`, and `Period_end` columns in the assembly control file (can) contain the names of columns in the master table. These three columns describe who is part of the study population and when is being studied. Initialising the master table involves creating a table with these details so they can be used by the assembly process.

We provide two examples to illustrate this idea:

- Consider a study of all New Zealand residents in the 2020 calendar year. For this study, initialising the master table could be done by creating a table with three columns: the first column containing the identity/uid of all residents, the second column containing the date for the first day in 2020, and the third column containing the date for the last day in 2020.
- Consider a cohort study, examining the lives of children born in 2010 until their fifth birthday. Such a study might use panel data: one row for each child for each year of life. For this study, initialising the master table could be done by creating a table with four columns: the identity/uid of the child, the year of life (a value from 1 to 5), the start date for that year of life (their birth day), and the end date for that year of life (the day before their next birthday). This table could then be populated by getting a list of all identities/uid for children born in 2010 and adding one row to the master table for each of their first five years of life.

Output methods provide summary of concepts

The `Output_method` column of the assembly control file specifies how the `Measure_value` column should be summarised to create a new column in the master table. Summarisation is

required so that each identity/uid and period receives a single value even if a concept definition has more than one relevant record. For example, a concept definition might contain one record for each hospital admission. As people can have different numbers of admissions, including the values from every admission would complicate data preparation. Instead, assembly allows for summaries like 'number of admissions' or 'whether any admissions were for respiratory conditions' which are more applicable for subsequent analysis.

The accepted values for the `Output_method` column are:

- **EXISTS** – Indicates whether a record in `Measure_value` exists. Gives the value 1 if there is at least one relevant record for the identity/uid and time period, otherwise NULL.
- **COUNT** – Counts the number of records in `Measure_value`. Records with a NULL value are not counted; `"1"` can be used as a placeholder in `Measure_value` to count all records.
- **DISTINCT** – Counts the number of distinct values among the records in `Measure_value`.
- **MIN** – Takes the minimum value of the records in `Measure_value`. If the input is text, then gives the first value after alphabetical sorting.
- **MAX** – Takes the maximum value of the records in `Measure_value`. If the input is text, then gives the last value after alphabetical sorting.
- **MEAN** – Takes the mean of the records in `Measure_value`. It will error if `Measure_value` is not a numeric column. We often avoid this summary option and instead produce separate sums and counts – dividing to get the mean after release from the data lab – as we find this an easier way to meet the confidentiality requirements.
- **SUM** – Takes the sum of the records in `Measure_value`. Considers all records where the `Measure_start` to `Measure_end` date range overlaps with the `Period_start` to `Period_end` date range, regardless of the amount of overlap. Will error if `Measure_value` is not a numeric column.
- **SUM_WITHIN** – Like SUM, but pro-rata's `Measure_value` based on the proportion of overlap between the measure and period date ranges. For example: if `Measure_start` and `Measure_end` define a 12-month date range and `Period_start` and `Period_end` define an overlapping 3-month date range, then only one quarter of `Measure_value` would be used for the sum.
- **DURATION** – Sums the number of days that the `Measure_start` to `Measure_end` date range overlaps with the `Period_start` to `Period_end` date range.
- **ENTITY** – Produces two output columns with the minimum and maximum `Measure_value` respectively. These columns are named with `*__min` and `*__max` suffixes. These output columns are intended to be paired with the Entity summary type in the summarise tool to produce entity counts that satisfy IDI output checking rules. The production of two columns allows the dataset to record up to two entities per person.

All these summary types use the who-when assembly pattern demonstrated below in Figure 2. This pattern means that the summary method is only applied to records that have matching identity/uid values and overlapping date ranges. See the section below for more details.

Modifies the master table with progress reporting

The assembly tool produces output in two forms. First, it makes permanent modifications to the master table in the form of new columns added (or existing columns removed and re-added).

Second, it returns as an R data frame the contents of the control file with three new columns giving the start time, end time, and success or failure of each assembly. Like the summary tool, the contents of this data frame are used to update the control file with progress reporting. This makes it straightforward to confirm how the tool performed and to assess its runtime.

Because the assembly tool makes permanent modifications to the master table, it is not straightforward to undo or reverse its actions. For example, if it is used to re-assemble an existing column, then it will overwrite the existing column according to its current instructions, preventing the recovery of the original column contents.

One of the few ways to reverse the actions of the assembly tool is to drop columns it has created. Code block 6 demonstrates this command in SQL. While this is useful in some instances, we recommend managing your data preparation process so that it is straightforward to drop and recreate the entire master table as this ensures a more robust process.

Code block 6: Dropping a column from the master table in SQL

```
ALTER TABLE [database].[schema].[master_table]
    DROP COLUMN IF EXISTS [column_name]
```

Assembly tool worked example

Table 6 and Table 7 provide a simplified, fictional example of a master table and an assembly control file. The first three columns of Table 6 are created when the table is initialised before assembly. Every master table needs to be initialised prior to assembly with at least some minimum information. Most often this is the list of ID numbers and the date range being studied.

The last three columns of Table 6 would be created by the assembly instructions demonstrated in Table 7. Observe how the names and formats of the columns in Table 6 match the values in the `output_name` and `output_type` columns of Table 7.

Table 6: Example master table after applying assembly tool

snz_uid	year_start	year_end	ED_visits	island_of_birth	income_000_cal_yr
1111	2021-01-01	2021-12-31	1	north	71.005
2222	2021-01-01	2021-12-31	2	south	59.999
3333	2021-01-01	2021-12-31	NULL	north	20.202
4444	2021-01-01	2021-12-31	NULL	south	40.000

Table 7: Example assembly control file, subset of columns

measure_ start	measure_ end	measure_ value	output_ name	output_ method	output_ type
[hospital_date]	[hospital_date]	"1"	"ED_visits"	COUNT	INT
"1900-01-01"	"5000-01-01"	[island_name]	"island_of_birth"	MAX	VARCHAR(5)
[tax_year_start]	[tax_year_end]	{0.001*[gross_income]}	"income_000_cal_yr"	SUM_WITHIN	FLOAT

For Table 7 we show only the most relevant subset of columns. We describe the logic for each row of this example in turn:

- ED visits use the same date `hospital_date` as both the measure start and end date. This is because the concept definition for ED visits treats visits as a point in time / single day event.

  For each person, the ED-visits column in the master table contains a count of the number of events where `hospital_date` falls between the `year_start` and `year_end` dates. Observe that, zero/no events have appeared as missing values.

- Island of birth uses two constant dates in the distant past and the future for the measure start and end dates. As island of birth does not change over time, this is a practical way to ensure the concept definition is available regardless of the time period being studied.

  For each person, the island-of-birth column in the master table contains the maximum value from the `island_name` column. As every person should have at most one island of birth, taking the maximum is a practical way to satisfy the control file structure and to protect against data quality errors giving an individual more than one record.

- Income by calendar year uses `{}` delimiters to include a simple calculation in the control file: multiplying `gross_income` by 0.001 to convert from dollars to thousands-of-dollars.

  Because of the tax-years in the concept definitions do not align perfectly with the calendar-years in the master table, the assembly instruction uses `SUM_WITHIN` to convert between them. For each person, the income-by-calendar-year column in the master table contains the sum of income for tax years that overlap with the calendar year *after* pro rata adjusting for the proportion of tax-year days that are within the calendar year.

Understanding the who-when assembly pattern

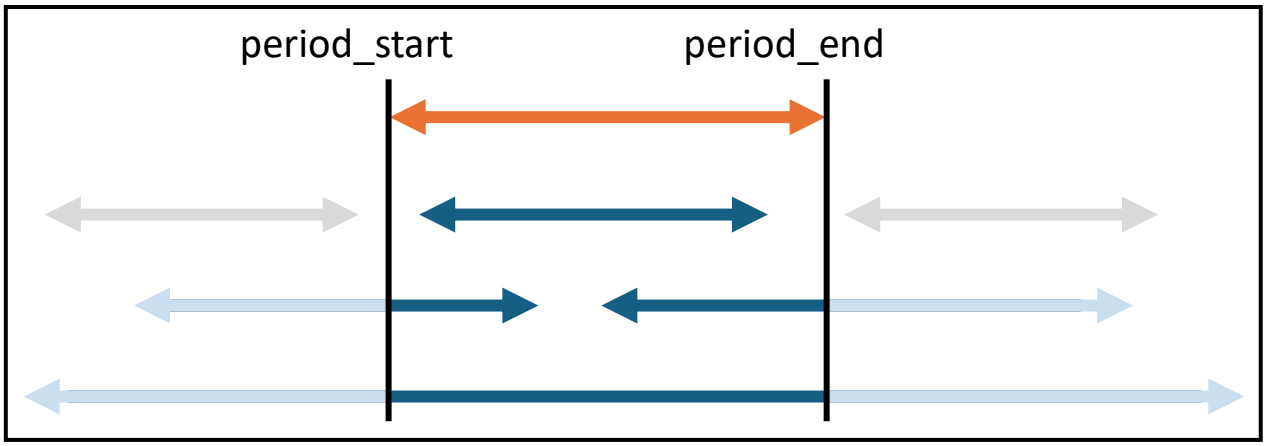
Central to the working of the assembly tool is a who-when pattern: the tool always connects the master table and the concept measure table using both who (the ID number) and when (date range). This means that if the concept definition or measure table contains information on people or time periods that are not present in the master table, this extra information has no effect on the results.

Figure 2 illustrates how the date ranges in the master table interact with the date ranges in the concept definition. The orange interval represents the study period. This is drawn from the master table and appears in the `period_start` and `period_end` columns of the control file. As represented by the vertical black lines, date ranges in the concept definition must overlap with this period to be included in the assembly process.

The grey intervals represent events or spells that do not overlap with the study period and hence are ignored during assembly. The blue intervals represent events or spells that overlap with the study period and hence are included during assembly.

For `output_methods` like `MAX` or `COUNT` there is no distinction between the dark and light blue intervals. But for options like `DURATION` and `SUM_WITHIN` events and spells are scaled to reflect only the dark blue intervals.

Figure 2: Interaction of period dates and measure dates



The process illustrated in Figure 2 is applied separately to each row in the master table. This means that every row can have different `period_start` and `period_end` values, and that the same identity can appear in multiple rows in the master table with independent processing of each row.

The pipeline tool

The pipeline tool exists to automate the process of running standard analyses that are spread across multiple files. The `run_pipeline` function in the toolkit locates and executes each file listed in its control file. It comes with options to delay execution, allowing intensive tasks to be scheduled outside of business hours.

Effective use of pipelines enables the complete automation of established analyses. For work under development, this tool can execute a set of files in order, reducing the extent to which users need to monitor progress.

Control file is one row per script to execute

For each row in the control file, the pipeline tool reads the commands and passes them to SQL or R for execution. The accepted columns for the control file are:

- **Enabled** – An optional true/false column. If this column is included, then rows set to false will not be executed.
- **Order** – An optional column for setting the order of files to be run. Files without an order are run in the order they appear in the control file after all files with an order.
- **Folder** – The folder that contains the file to be executed. Some pre-processing is applied to this column to support paths being copied as text from Windows even if R is running on a different operating system.
- **File** – The file to be executed. Only R and SQL scripts are accepted.
- **Notes** – A free text column that is ignored and does not affect output. Intended for adding notes to the control file. Any other column names are also ignored but generate a warning.

As an alternative to providing a folder and file, the text `STOP IF ANY FAILURES` can be entered in the File column. This is a special command and will cause the tool to stop early if any script so far has errored. This is useful for separating stages of the pipeline where subsequent stages should run only if all of the previous stage(s) complete.

Progress reporting output with failure messages

The pipeline tool can execute any code that a user might run themselves, including other tools in this toolkit. This means it can produce almost any output that a user can generate, with one key distinction: The pipeline does not receive results back from scripts that it runs. This means that output from scripts need to be saved to disk if they are to be used by later steps in the pipeline or reviewed by a user once execution concludes.

The pipeline tool provides a small amount of direct output: the contents of the control file with three new columns giving the start time, end time, and success or failure of each script. Like the summary and assembly tools, the contents of this data frame are used to update the control file with progress reporting. This makes it straightforward to confirm how the tool performed and to assess its runtime.

Pipeline tool worked example

Table 8 provides an example of a pipeline control file. It shows a simple project containing a mix of R and SQL scripts. In this control file, the order and notes columns have been used to distinguish between the stages of the project.

Table 8: Example pipeline control file

Enabled	Order	Folder	File	Notes
TRUE	1	I:\project\folder	config_settings.R	Setup
TRUE	1.1	I:\project\folder	initialise_master_table.sql	Setup
TRUE	2	I:\project\folder\defns	island of birth.sql	Definitions
TRUE	2.1	I:\project\folder\defns	ED visits.sql	Definitions
TRUE	2.2	I:\project\folder\defns	income by tax year.sql	Definitions
TRUE	2.3		STOP IF ANY FAILURES	Definitions
TRUE	3	I:\project\folder	run_tool_assembly.R	Assembly
TRUE	3.1	I:\project\folder	tidy_master_table.sql	Assembly
TRUE	3.2		STOP IF ANY FAILURES	Assembly
TRUE	4	I:\project\folder	fit_models.R	Analysis
TRUE	4.1	I:\project\folder	run_tool_summary.R	Analysis
TRUE	4.2		STOP IF ANY FAILURES	Analysis
TRUE	5	I:\project\folder	run_tool_confidentialise.R	Output

Robust error handling within the pipeline means that if a file errors, the pipeline will move to the next file and continue to execute. The intention of this design is to maximise the progress achieved on each execution of the tool. However, it can result in cascading errors where a failure during the execution of one file leads to errors in all downstream files.

Table 8 demonstrates our standard practice to reduce cascading errors: each stage ends with `STOP IF ANY FAILURES` to prevent execution of future stages. Without these rows execution will continue to run creating guaranteed errors. For example, suppose the island-of-birth definition (row 3) contains a mistake and errors during execution. The pipeline tool will still attempt to run the ED visits and income-by-tax-year definitions (rows 4 and 5). Without `STOP IF ANY FAILURES` on row 6 the assembly stage will attempt to execute and error due to the failure of the island-of-birth definition. Hence the inclusion of `STOP IF ANY FAILURES` on row 6 means the pipeline stops at the point it can no longer make meaningful progress.

Isolated execution environments

The flexibility to run arbitrary R and SQL scripts makes the pipeline tool the most powerful tool in the toolkit. However, safe use of this power requires that (1) that files are self-contained, and (2) files are developed to run end-to-end without user intervention.

The need for files to be self-contained arises because the pipeline tool executes each R and SQL script in an isolated environment. For R scripts, this means each script starts from an empty R session with no packages or variables loaded. For SQL scripts, this means that each script starts from a fresh ODBC connection with no temporary tables or declared variables.

Isolated environments are a required part of the pipeline tool. Without isolated environments, it is possible for code in one script can cause arbitrary failures in other scripts or in the tool itself. This would lead to significant challenges with debugging errors and with obtaining repeatable results.

The need for files to run end-to-end without user intervention arises whenever a user moves from hands-on development to automation. In our experience any adjustments to code are often minor, and it is the different ways of thinking that automation requires that is the more challenging change for users.

Designed for ease of fault fixing

Throughout the development of ADAPT, significant attention has been given to the detection and correction of faults. We divide faults into two types: software faults and use errors. Software faults arise from mistakes made during development. Use errors arise from mismatches between a user's actions and the behaviour of the software.

As discussed near the start of this document, development of the toolkit has included over 700 automated tests. By specifying pairs of example inputs and expected outputs these tests help ensure the toolkit is free of software faults. Please let us know if you encounter a software fault (for example, by emailing info@sia.govt.nz). But ensure you first follow the advice below to avoid reporting your own use errors.

The toolkit has been designed to assist the user to solve 'use' errors. This section covers a range of approaches for users to solve such faults. We cover these approaches not because the toolkit is prone to faults, but because deliberate work was undertaken during the design and development of the toolkit to make this process as straightforward as possible.

Validate control files

The toolkit has been designed around control files as a way to give users flexibility in how they apply the tools. However, this also makes users responsible for creating control files that conform to the expectations of the corresponding tool.

To help users confirm that a control file can be run by a tool, each tool has its own validate function. When any tool is run, the control file is validated before any data processing takes place. If users want to validate a control file without running the tool, then they can make direct use of the corresponding function. As the commands for validation look a little different for the commands to run a tool, Code block 7 provides an example of using the validate command for the summary tool – a key difference here is the use of `load_control_file`.

Code block 7: Validation of summary control file in R

```
# example master table is csv file
table = read.csv('path/to/data/my master table.csv')

# read control file into memory
control_file = ADAPT::load_control_file('folder/path/control_file.csv')

# check if valid
is_valid = ADAPT::validate_summary_control_file(control_file, table)
```

All the validate functions behave in the same way: They run a list of checks and generate a warning for the user for each check that fails. Each warning includes a description of the fault to assist the user identifying and correcting it.

Should the list of warnings be too long to manage, or it become difficult to locate the cause of a fault, we recommend using the Enabled column in the control file. Rows that are not enabled are not validated. This allows for targeted validation of control files where required.

In addition to warning about each individual fault, the validate functions also return True or False for whether the control file is ready to run. If any critical check fails the function will return False (`is_valid = False`), otherwise the function will return True (`is_valid = True`).

To list all the validate functions and see the checks provided by each function, we recommend using the in-built R help as demonstrated in Code block 5. For example: the command `??validate_` in R will cause the help window to display a list of validation functions.

When validating a control file, it is common to iterate back and forth between changing the control file and rerunning the validate function. As this kind of iteration loads the control file from disk each time, it is essential to save the control file before re-validating. Unsaved edits can not be validated and in our experience are the main cause of confusion when validation fails.

Review logs to track progress

When a fault occurs, identifying where it occurred in the process is often the first step to fixing it. The pipeline, assembly, and summary tools all log progress by printing to the R console.[6] Reviewing these logs is an easy way to observe what occurred immediately before and after a fault occurred.

Printing log messages to the R console is only suitable for small applications. For larger applications, saving the log to a file is recommended. The pipeline tool includes a parameter `sink_file` for this purpose. While R includes a standard function `sink` for this purpose, the version within the pipeline tool provides better logging of errors and warnings.

In addition to reviewing the log, we also recommend reviewing each control file after it has been used. As described above, the summary, assembly, and pipeline tools output the start time, end time, and success or failure of each step, and use this information to update their control file. While a small number of faults can prevent this update (for example, a crash of the server or a reset of the user's profile), seeing this information aligned with each row in the control file is an effective way to identify where any fault occurred.

Set a debug folder

When working with database tables, the assembly and summarise tools both generate and execute internal SQL queries. By default, such queries are generated on demand and discarded after use. In some instances, it can be useful to review these queries or to run them manually.

To save these SQL queries, both tools have a `debug_folder` parameter. If a folder is provided, then copies of the internally generated queries are saved into the folder prior to execution. If a fault occurs, it will often be caused by the most recent file saved to the debug folder.

[6] This logging is done with the `run_time_inform_user` function, which provides time-stamped messages. If writing custom scripts for a pipeline, we recommend use of this function for logging.

On a handful of occasions, using the debug folder has been the eventual way we determined the cause of a fault. For example: when using the summary tool to calculate the sum of a column, the logic was sound, the test cases ran without error, but an error message continued to occur. Enabling the debug folder and running the generated SQL script manually revealed that the sum was too large for the data type when run against the full dataset and was causing an overflow error.