Package: survey160r (via r-universe)

October 27, 2026

Title R Client for Survey160 Data

Version 0.78.1

Description Access Survey160 campaign data from R. Reads campaign exports from Google Cloud Storage or local files and triggers fresh exports via the Survey160 API. Computes per-campaign recipient-latency reports, screens phone samples against the disposition dataset and the opt-out list, and rolls disposition and campaign metrics up to funnel counts and rates. Results are in-memory data frames; persistence and fleet orchestration live in consumer projects.

License MIT + file LICENSE

URL https://github.com/survey160/survey160r

BugReports https://github.com/survey160/survey160r/issues

Encoding UTF-8

Depends R (>= 4.1)

Imports googleCloudStorageR (>= 0.7.0), googleAuthR, gargle, httr, jsonlite, lubridate, digest, dplyr, rlang, lifecycle, data.table, nanoparquet (>= 0.4.0)

Suggests askpass, DBI, dbplyr (>= 2.4.0), duckdb (>= 1.0.0), testthat (>= 3.1.9), mockery, withr, lintr, covr, knitr, rmarkdown

VignetteBuilder knitr

Config/testthat/edition 3

RoxygenNote 7.3.3

Config/pak/sysreqs cmake make libuv1-dev libssl-dev

Repository https://survey160.r-universe.dev

Date/Publication 2026-10-27 16:36:55 UTC

RemoteUrl https://github.com/survey160/survey160r

RemoteRef HEAD

RemoteSha 36fc10d6af6a5194fbfd6ac22205aa0d2efc3d63

Contents

survey160r-package . . . 3
campaign_metrics_pull . . . 6
campaign_metrics_records . . . 7
campaign_metrics_summary . . . 8
disposition_funnel . . . 9
disposition_input_columns . . . 11
disposition_paths . . . 13
disposition_pull . . . 15
disposition_records . . . 16
disposition_run . . . 18
disposition_screen . . . 20
disposition_steps . . . 23
disposition_summary . . . 26
funnel_rates . . . 28
latency_build_config . . . 29
latency_config_hash . . . 30
latency_discover_questions . . . 31
latency_funnel . . . 32
latency_input_columns . . . 33
latency_report . . . 34
latency_run . . . 35
latency_validate_config . . . 37
opt_out_pull . . . 38
opt_out_screen . . . 39
parse_campaign_timestamps . . . 40
question_funnel . . . 41
question_timestamps . . . 42
s160_api_auth . . . 43
s160_api_campaign_get . . . 44
s160_api_campaign_results . . . 46
s160_config . . . 47
s160_csv_header . . . 48
s160_datasets . . . 48
s160_gcs_campaign_results_files . . . 49
s160_gcs_campaign_results_list . . . 50
s160_gcs_campaign_results_read . . . 51
s160_gcs_campaign_results_status . . . 52
s160_gcs_init . . . 53
s160_read_csv . . . 55
utils_fix_double_utf8 . . . 56

Index **58**

survey160r-package *survey160r: R Client for Survey160 Data*

Description

Access Survey160 campaign data from R. The `s160_*` functions reach Survey160's own systems – Google Cloud Storage, the Survey160 API, and raw campaign-export CSVs. The rest work on what those return or on datasets survey160r produces: **latency** reports, **disposition** and **opt-out** screening of phone samples, and **campaign metrics**. Every function, grouped by task, is listed below. The articles (`vignette("latency")`, `vignette("disposition")`, `vignette("opt-out")`, `vignette("campaign-metrics")`) walk through each task.

First-time setup

Functions that touch Survey160 data need a one-time sign-in:

- `s160_gcs_init` – authenticate to Google Cloud Storage (a browser sign-in on first run, then cached; `adc = TRUE` for a non-interactive run). Needed to read campaign exports and to download the disposition dataset, the opt-out list, and the campaign-metrics projection (a cache hit needs no sign-in).
- `s160_api_auth` – authenticate to the Survey160 API. Needed only to trigger a fresh export with `s160_api_campaign_results`.

Campaign exports

Read a campaign's raw results:

- `s160_gcs_campaign_results_read` – read a campaign export from Cloud Storage.
- `s160_read_csv` / `s160_csv_header` – read a campaign export, or just its column names, from a local file.
- `s160_gcs_campaign_results_list` / `s160_gcs_campaign_results_files` / `s160_gcs_campaign_results_statu` – list campaigns, list a campaign's files, and check when its export last updated.
- `s160_api_campaign_results` – trigger a fresh export and read it; `s160_api_campaign_get` reads a campaign's attributes.
- `s160_config` / `s160_datasets` – the environments and datasets the package can reach.

Latency analysis

Turn a campaign export into a recipient-latency report (`vignette("latency")`):

- `latency_run` – build the config from the export and run the report; `latency_report` runs it on a config you supply.
- `latency_build_config` / `latency_validate_config` / `latency_discover_questions` / `latency_config_hash` – build, check, and fingerprint a config.
- `latency_input_columns` – the columns to read, to project a large export.
- `latency_funnel` – reduce a report to funnel counts.
- `question_funnel` / `question_timestamps` / `parse_campaign_timestamps` – per-question views straight from the export.

Disposition screening

Screen a phone sample on prior contact, and measure the funnel across recipients (`vignette("disposition")`):

- `disposition_pull` – download the disposition dataset (cached locally).
- `disposition_screen` – append the screening columns to your sample, 1:1 with its rows.
- `disposition_summary` / `disposition_records` – query the dataset: one row per phone, or the stored rows, one per phone and campaign.
- `disposition_funnel` – funnel counts and rates by any dimension.
- `disposition_steps` / `disposition_paths` – each number's campaigns in time order: per-campaign history (what happened after k non-responses) or one path per phone with sequence patterns (no reply, then engaged, then completed).
- `disposition_run` / `disposition_input_columns` – build the per-respondent rows from one campaign export (the producer side of the dataset).

Opt-out screening

Flag opted-out numbers in a phone sample (`vignette("opt-out")`):

- `opt_out_pull` – download the opt-out list (cached locally).
- `opt_out_screen` – append `opted_out` and `opt_out_date` to your sample, 1:1 with its rows.

Campaign metrics

Work with the consolidated per-campaign metrics projection – funnel counts and Project Tracker dimensions per campaign, date, and hour (`vignette("campaign-metrics")`):

- `campaign_metrics_pull` – download the projection (cached locally).
- `campaign_metrics_records` – read it into analysis-ready cells, one row per cell.
- `campaign_metrics_summary` – funnel counts and rates by any dimension.
- `funnel_rates` – the funnel-rate definitions shared with `disposition_funnel`.

Utilities

- `utils_fix_double_utf8` – detect or repair double-encoded text (mojibake) in an export.

Local cache

`disposition_pull`, `opt_out_pull` and `campaign_metrics_pull` download once into `tools::R_user_dir("survey160r", "cache")` and return the cached path on later calls, saying how old the copy is ("Using cached disposition projection (3 days old): ..."). A cache hit needs no sign-in and never checks Cloud Storage, so pass `refresh = TRUE` for today's data, or `dest =` to keep a copy elsewhere. Remove the directory to start clean.

Errors and messages

Every error the package raises says which function failed, what it received, and what to do:

```
s160_api_campaign_results: `timeout` must be a positive number, not -1.
s160_gcs_campaign_results_read: file not found: gs://.../744_raw_data_download.csv.
 List the campaign's files with s160_gcs_campaign_results_files(744, env = "prod"), ...
```

Progress (downloads, export polling, cache hits) goes to `message()`, so `suppressMessages()` silences it and a log captures it.

Errors are classed, so a script can catch exactly what it expects with `tryCatch()` instead of matching message text:

- `s160_error` – every error raised by survey160r itself (never one from R or a dependency).
- `s160_not_initialized` – `s160_gcs_init` / `s160_api_auth` has not run in this session.
- `s160_not_found` – a campaign, export file, local file or dataset does not exist.
- `s160_http_error` – the Survey160 API answered a request with an error; `s160_auth_error` – the sign-in itself was refused. Both carry the HTTP status code in the condition's `status` field.
- `s160_config_error` – a latency config (or the export it is checked against) fails `latency_validate_config`.

```
tryCatch(
  s160_gcs_campaign_results_read(744),
 s160_not_initialized = function(e) { s160_gcs_init(); s160_gcs_campaign_results_read(744) },
  s160_not_found = function(e) NULL
)
```

Persisting results, walking the whole campaign fleet, and scheduling live in consumer projects; this package returns in-memory data frames, except `disposition_steps` and `disposition_paths`, which return lazy **dbplyr** tables that DuckDB computes on `collect()`.

Author(s)

Maintainer: Lennon Shimokawa <lshimokawa@survey160.com>

See Also

Useful links:

- https://github.com/survey160/survey160r
- Report bugs at https://github.com/survey160/survey160r/issues

`campaign_metrics_pull` *Download the campaign-metrics projection from Cloud Storage*

Description

Pulls `campaign_all.parquet` – the fleet's consolidated per-campaign metrics and recipient-latency view – from the environment's analytics bucket to a local file and returns the path, ready to hand to `campaign_metrics_summary` (funnel counts and rates by any dimension) or `campaign_metrics_records` (the analysis-ready cells). Downloaded once and reused from the local cache on later calls (pass `refresh = TRUE` to force a fresh download). Parallels `disposition_pull` and reaches GCS the same way: authenticate first with `s160_gcs_init()` so the session's GCS credentials are set. A download without an initialized session errors with "GCS not initialized. Run s160_gcs_init() first." (a cache hit is served without needing auth).

Usage

```
campaign_metrics_pull(
  env = .ENV_CHOICES,
  dest = NULL,
  bucket = NULL,
  refresh = FALSE,
  progress = interactive()
)
```

Arguments

`env`	Environment: `"prod"` (default) or `"dev"`. There is no staging analytics tier; passing `env = "staging"` errors clearly.
`dest`	Where to save. `NULL` (default) caches under `tools::R_user_dir("survey160r", "cache")`. A directory saves the default filename (`<bucket>.campaign_all.parquet`) inside it; any other single string is treated as the exact output path (its parent is created).
`bucket`	**Deprecated.** Select data with `env =` instead; a supplied bucket is honored with a warning for back-compat.
`refresh`	When `FALSE` (default), reuse an existing local copy; `TRUE` always re-downloads (the projection is republished on every producer run, so refresh to pick up a newer one).
`progress`	Show a download progress bar. Defaults to `interactive()`: a live bar in an interactive session, silent in batch or scheduled runs.

Details

The projection is one row per `(campaign_id, date, hour_local, segment, threshold)` and carries the funnel counts, the recipient-latency distribution, and the `tracker_*` project dimensions. It is aggregate metrics, not phone-keyed PII, so it lives in the `s160_analytics_<env>` bucket rather than the disposition bucket, and there is no screening helper. For counts and rates, use

campaign_metrics_summary: it reads through campaign_metrics_records (which drops the rows the file repeats across its latency segment and threshold rows) and sums one grain – the day totals, not the hour cells as well – so a send is counted once. A raw Parquet reader, or a sum over the records' hour and day cells together, multi-counts every send.

Value

The local path to the downloaded Parquet (a single string), ready for campaign_metrics_summary / campaign_metrics_records.

See Also

campaign_metrics_summary, campaign_metrics_records, disposition_pull, opt_out_pull, s160_gcs_init

Examples

```
## Not run:
s160_gcs_init()   # one-time browser OAuth
path <- campaign_metrics_pull()          # prod
campaign_metrics_summary(path, by = "registration_id")
cells <- campaign_metrics_records(path)  # the cells, for your own slicing

## End(Not run)
```

---

campaign_metrics_records

Read the campaign-metrics projection into analysis-ready cells

---

Description

Reads `campaign_all.parquet` (from campaign_metrics_pull) into a tidy frame: the `tracker_*` project dimensions renamed to short names (`registration_id`, `brand`, `client`, `fielding_location`, `mode`), the funnel counts (`n_sent`, `n_engaged`, `n_opted_in`, `n_completed`), and the cell keys (`campaign_id`, `date`, `hour_local`).

Usage

```
campaign_metrics_records(x, dedup = TRUE)
```

Arguments

x	A path to a campaign-metrics Parquet (from campaign_metrics_pull), or an in-memory data frame already read (carrying either the `tracker_*` names or the short names). The four `n_*` count columns are required.
dedup	Collapse the latency/threshold replication to one row per cell (default `TRUE`). `FALSE` returns the projected columns without collapsing (replicated rows remain).

Details

The projection is stored at (campaign_id, date, hour_local, segment,threshold) grain, and the funnel counts are **replicated** across the latency threshold (and segment) rows – so summing the raw file double-counts. With dedup = TRUE (default) the frame is collapsed to **one row per cell** (distinct rows over the returned columns, dropping the replicating grain), which is the correct base for campaign_metrics_summary. This is the step every consumer must get right; do not sum the raw pull.

The cells still carry **two grains** per (campaign_id, date): hour cells (hour_local 0-23) and a day-total cell (hour_local = NA) with the same sends. Sum one grain – the day totals, or the hour cells – never both; campaign_metrics_summary does this for you.

Value

A data frame of cells: campaign_id, date, hour_local, the present project dimensions, and the four n_* counts.

See Also

campaign_metrics_pull, campaign_metrics_summary, funnel_rates

Examples

```
## Not run:
s160_gcs_init()
cells <- campaign_metrics_records(campaign_metrics_pull())

## End(Not run)
```

---

campaign_metrics_summary

Summarize campaign metrics to funnel counts and rates by dimension

---

Description

Rolls the campaign-metrics cells up to summed funnel counts and the standard funnel rates, grouped by any project dimension(s). Reads and de-duplicates through campaign_metrics_records first, so it is safe to pass either a projection **path** or a frame – the latency/threshold replication is always collapsed before summing. Each (campaign_id,date) is counted at one grain: its day-total cell (hour_local = NA) when present, else its hour cells, so no send is counted twice.

Usage

```
campaign_metrics_summary(x, by = "campaign_id", rates = TRUE, percent = FALSE)
```

Arguments

x	A path to a campaign-metrics Parquet, or a data frame (raw or a campaign_metrics_records frame).
by	Character vector of grouping columns, from the cell dimensions (campaign_id (default), registration_id, brand, client, fielding_location, mode).
rates	Append the funnel rates via funnel_rates (default TRUE).
percent	Passed to funnel_rates: proportions (default) or 0-100.

Value

A data frame, one row per by group: the by columns, campaigns (distinct campaign count), the summed n_sent, n_engaged, n_opted_in, n_completed, and (when rates) engagement_rate, opted_in_rate, opted_in_engaged_rate, completion_rate.

See Also

campaign_metrics_records, funnel_rates

Examples

```
## Not run:
s160_gcs_init()
path <- campaign_metrics_pull()
campaign_metrics_summary(path, by = "registration_id", percent = TRUE)
campaign_metrics_summary(path, by = c("brand", "mode"))

## End(Not run)
```

---

disposition_funnel *Summarize the disposition dataset to funnel counts and rates by dimension*

Description

The disposition dataset is one row per (phone, campaign_id), contacted-only, with 0/1 funnel flags per recipient. disposition_funnel() groups those rows by one or more dimensions (by, e.g. carrier) and – when grain = "day" – by disposition_date (the recipient's latest activity date), returning the funnel COUNTS per group (campaigns, n_sent, n_engaged, n_opted_in, n_completed, n_ineligible, n_refused) and, when rates = TRUE, the standard funnel rates via funnel_rates.

Usage

```
disposition_funnel(
  x,
  by = "carrier",
  grain = c("day", "all"),
  rates = TRUE,
  percent = FALSE
)
```

Arguments

x	Either a path to a disposition Parquet file (the read projection, e.g. from disposition_pull) or an in-memory disposition data frame (from disposition_records). With **duckdb** installed a path is grouped inside the Parquet scan, so only the result reaches R (the production projection in under a second and well under 1 GB); otherwise it is read with **nanoparquet**, projected to the columns this rollup needs, and grouped in R (tens of GB for the production projection). Pre-filter (by campaign / date) with disposition_records first when you do not want the whole projection.
by	Character vector of grouping column name(s). Default "carrier"; any stored dimension works (survey_mode, tracker_client, tracker_topic, ...). Every name must be present in x.
grain	One of "day" (default) or "all". "day" adds disposition_date to the grouping (one row per (by..., date)); "all" collapses the date axis (one row per by...).
rates	Append the funnel rates via funnel_rates (default TRUE): engagement_rate, opted_in_rate, opted_in_engaged_rate, completion_rate.
percent	Passed to funnel_rates: proportions (default) or 0-100.

Details

It is the disposition analogue of campaign_metrics_summary (same by / rates / percent interface, a campaigns count, and the shared funnel_rates definition), over the disposition projection rather than the latency one, plus the disposition-only terminal split (n_ineligible / n_refused) and a grain switch for the disposition_date axis (the latency sibling instead takes the date as a by column). Because the disposition grain is already one row per recipient, "how many per carrier" is a straight group-and-sum – there is no denormalised fan-out to collapse (contrast latency_funnel). The disposition dataset is contacted-only (sent == 1), so n_sent is the group's row count; campaigns is its distinct campaign count.

A carrier the uploaded list did not supply is NA, kept as its own group (the "unknown carrier" bucket); this function does not fold carrier aliases (e.g. Metro PCS into T-Mobile) – do that in the caller. Off-channel (t2w_external) completes are unknowable, so completed is NA on those rows; a group whose completes are *all* NA returns n_completed = NA (and a NA completion rate), not a false 0.

Value

A data frame, one row per group: the by column(s), then `disposition_date` (only when `grain = "day"`), `campaigns`, the counts `n_sent`, `n_engaged`, `n_opted_in`, `n_completed`, `n_ineligible`, `n_refused`, and (when `rates`) the four `funnel_rates` columns. Sorted by the grouping columns.

See Also

`campaign_metrics_summary` (the latency-side analogue), `funnel_rates`, `disposition_records`, `disposition_summary` (the per-phone rollup)

Examples

```
records <- data.frame(
  phone = c("5551112222", "5553334444", "5555556666", "5557778888"),
  campaign_id = c(101L, 101L, 102L, 102L),
  sent = c(1L, 1L, 1L, 1L),
  engaged = c(1L, 1L, 0L, 1L),
  opted_in = c(1L, 0L, 0L, 1L),
  completed = c(1L, 0L, 0L, 0L),
  ineligible = c(0L, 0L, 0L, 0L),
  refused = c(0L, 1L, 0L, 0L),
  carrier = c("AT&T", "AT&T", "Verizon", NA),
  disposition_date = as.Date(c("2026-01-10", "2026-01-10",
                               "2026-01-11", "2026-01-11")),
  stringsAsFactors = FALSE
)
disposition_funnel(records)                # by carrier, per disposition_date
disposition_funnel(records, grain = "all", percent = TRUE)
```

---

disposition_input_columns

List the export columns disposition_run() reads

Description

Returns the (dot-form) column names `disposition_run()` touches, so a caller can project a wide export down to just those columns and get output identical to a full read. This is the disposition analogue of `latency_input_columns()` (latency), with two deliberate differences: disposition is decoupled from the question flow (no `config` argument), it reads `phone` (the row key), and it does NOT read `campaignid` – the `campaign_id` is stamped from the `disposition_run()` argument, not the data.

Usage

```
disposition_input_columns(available = NULL, population = NULL)
```

Arguments

available	Optional character vector of the actual (dot-form) column names present in the file (e.g. from s160_csv_header()). When supplied, the close-message Text columns and every survey-body scriptDate are retained. Effectively required: omitting it yields a lossy projection that can undercount opted_in on a multi-question campaign (see Details).
population	Optional population-filter expression defining opted_in. NULL (default) is routing-based opt-in – the recipient reached a continuation step (any non-opener, non-terminal scriptDate), which is language- and phrasing-agnostic and reads no answer text. A completion also sets opted_in (disposition_run() OR-s in completed), so for a text-to-web campaign whose link sits in the intro a web completion can be the only opt-in signal. A caller may instead pass an explicit filter (e.g. id.intro.finalText == "Yes"); its columns are added so a custom population's inputs are not projected away.

Details

Some columns are data-dependent, so available (e.g. the result of s160_csv_header()) is effectively required for a faithful projection:

1. the close-message Text columns detect_survey_mode() greps to tell t2w_external from sms – omit them and a t2w_external campaign is misclassified as sms;
2. every survey-body scriptDate, which the routing-based opted_in keys on. Without available the set cannot enumerate the body questions (it degrades to intro + the fallback close only), so a projected read of a multi-question campaign would DROP the body timestamps and silently undercount opted_in (a recipient who reached a body question but not the close). Pass available (or read the full file) whenever a campaign has body questions.

Value

A character vector of unique dot-form column names, including phone and (best-effort) the optional carrier misc column – matched case-insensitively when available is supplied, else requested as lowercase "carrier". Pass it as columns = to s160_read_csv() / s160_gcs_campaign_results_read().

See Also

latency_input_columns, the latency analogue. It leads with its *required* config; here available leads because it is the argument you almost always pass (see the example), and there is no required argument – a deliberate difference, not an inconsistency.

Examples

```
## Not run:
header <- s160_csv_header(path)
data <- s160_read_csv(path, columns = disposition_input_columns(header))
disposition <- disposition_run(1234, data)$consolidated

## End(Not run)
```

`disposition_paths` *Read the disposition dataset as one path per phone, and match sequence patterns*

Description

Returns one row per phone with its campaigns, in time order, as a letter path (`"NNEC"`: two non-responses, engaged, completed), the run-length shape (`"N+E+C+"`), and – with `pattern` – whether and where the path matches a sequence such as `"non_response > engaged|opted_in > completed|web_complete"` (no reply, then later engaged or opted in, then later completed). Count shapes, filter matches, or join the result back to a sample to flag numbers by their history. For per-campaign history (what happened *after* a given step) see `disposition_steps`, which shares the ordering, labels, letters and resources documented there.

Usage

```
disposition_paths(
  dataset,
  pattern = NULL,
  phones = NULL,
  sample_frac = NULL,
  date_from = NULL,
  date_to = NULL,
  terminal = c("raw", "engaged"),
  undeliverable_codes = .DISPOSITION_UNDELIVERABLE_CODES
)
```

Arguments

`dataset`	Path to the disposition Parquet file, e.g. from `disposition_pull`.
`pattern`	Optional sequence pattern (see Pattern syntax). Adds `matched`, `match_start`, `match_end` (step positions in the path, `NA` without a match), `match_start_date` and `match_end_date`. Every phone is returned; `filter(matched)` keeps the matches.
`phones`	Optional character vector of phone numbers; keeps only these phones (all of their campaigns). Matched digit-normalized, as the other readers do. `character(0)` returns no rows.
`sample_frac`	Optional number in (0, 1]: keep a deterministic sample of that fraction of phones (by an md5 of the phone, so the same phones on every machine and DuckDB version). `0.1` makes full-projection exploration ~10x cheaper; the 10% sample matches the full data to within 0.02 points on prod.
`date_from, date_to`	Optional `Date`/date-string bounds on `disposition_date` (inclusive): only campaigns in range form the path.

`terminal` How to label an unsplit `terminated` row (terminated but neither refused nor ineligible): `"raw"` (default) keeps `terminated`, matching `disposition_summary`; `"engaged"` labels it by the step it reached (`opted_in` or `engaged`). On prod most of these rows come from a producer change in August 2026 rather than respondent behaviour.

`undeliverable_codes` Delivery-error codes that make a no-reply row `undeliverable` rather than `non_response`. Defaults to the hard carrier rejections (4700, 4701, 4702, 4720, 4750, 4753); `NULL` labels every no-reply row `non_response`, as the summary does.

Details

Pattern syntax. Steps are label names or their letters – `non_response` N, `undeliverable` X, `engaged` E, `opted_in` O, `terminated` T, `ineligible` I, `refused` R, `completed` C, `web_complete` W – or `*` for any label.

- `a > b`: `b` comes later than `a`; other campaigns may sit between.
- `a >> b`: `b` is the very next campaign.
- `a|b`: either label; `a+`, `a{3}`, `a{3,}`, `a{2,4}`: repeats in a row.
- `^` / `$` at the start / end: the phone's first / most recent campaign.

`"non_response > completed"` and `"N > C"` are the same pattern. `match_start` / `match_end` locate the leftmost match: a repeat takes its whole run (`"N+"` on `"XNNT"` is steps 2-3) and a `>` gap stops at the earliest next step (`"N > C"` on `"NCNC"` is steps 1-2).

Date window. Unlike `disposition_steps`, `date_from` / `date_to` choose which campaigns form the path (the phone's 2025+ path, say), so they apply before the path is built.

Value

A lazy **dbplyr** table, one row per phone: `phone`, `n_campaigns`, `path`, `shape`, `first_date`, `last_date`, `n_unknown_completion` (campaigns whose completion is never reported – text-to-web external – so a `completed` step there cannot match), and with `pattern` the match columns.

See Also

`disposition_steps` (per-campaign history), `disposition_screen` (screen a sample on counts)

Examples

```
## Not run:
library(dplyr)
dataset <- disposition_pull()

# The commonest path shapes
disposition_paths(dataset) |> count(shape, sort = TRUE) |> head(20) |> collect()

# No reply, then engaged or opted in, then completed
disposition_paths(dataset, "N > E|O > C|W") |> filter(matched) |> count() |> collect()
```

```
# Flag a sample's numbers whose last three contacts got no reply
flags <- disposition_paths(dataset, "non_response|undeliverable{3,}$",
                           phones = my_sample$phone) |>
  select(phone, path, matched) |>
  collect()

## End(Not run)
```

`disposition_pull` *Download the disposition dataset from Cloud Storage*

Description

Pulls the consolidated disposition projection (`disposition_all.parquet`) from the environment's disposition bucket to a local file and returns the path – ready to hand to `disposition_summary` / `disposition_screen`. Downloaded once and reused from the local cache on later calls (pass `refresh = TRUE` to force a fresh download). This is the one `disposition_*` function that reaches GCS: authenticate first with `s160_gcs_init()` so the session's GCS credentials are set. A download without an initialized session errors with "GCS not initialized. Run s160_gcs_init() first." (a cache hit is served without needing auth).

Usage

```
disposition_pull(
  env = .ENV_CHOICES,
  dest = NULL,
  bucket = NULL,
  refresh = FALSE,
  progress = interactive()
)
```

Arguments

`env`	Environment: `"prod"` (default) or `"dev"`. There is no staging disposition tier; passing `env = "staging"` errors clearly.
`dest`	Where to save. `NULL` (default) caches under `tools::R_user_dir("survey160r", "cache")`. A directory saves the default filename (`<bucket>.parquet`) inside it; any other single string is treated as the exact output path (its parent is created).
`bucket`	**Deprecated.** Select data with `env =` instead; a supplied bucket is honored with a warning for back-compat.
`refresh`	When `FALSE` (default), reuse an existing local copy; `TRUE` always re-downloads (the projection is rebuilt each pipeline pass, so refresh to pick up a newer one).
`progress`	Show a download progress bar. Defaults to `interactive()`: a live bar in an interactive session, silent in batch or scheduled runs. The projection is a few hundred MB, so an interactive pull otherwise looks stalled while it transfers.

Value

The local path to the downloaded Parquet (a single string), ready for disposition_screen, disposition_summary, disposition_records, disposition_funnel, disposition_steps and disposition_paths. The first pull transfers a few hundred MB (a minute or two on an office connection); a later call with refresh = FALSE and the same dest reuses that copy and returns at once.

See Also

disposition_summary, disposition_screen, opt_out_pull, s160_gcs_init

Examples

```
## Not run:
s160_gcs_init()   # one-time browser OAuth
dataset <- disposition_pull()                    # download (cached)
disposition_screen(my_sample, dataset)

## End(Not run)
```

---

disposition_records *Read the disposition records, one row per phone and campaign*

Description

Reads the disposition Parquet projection and returns its rows **as stored** – one row per (phone, campaign_id), carrying the full disposition schema: phone, campaign_id, sent, engaged, opted_in, completed, web_complete, refused, ineligible, terminated, error, carrier, survey_mode, tracker_loi, tracker_topic, tracker_mode, tracker_registration_id, tracker_client, tracker_project, tracker_brand, tracker_state, tracker_vendor, tracker_fielding_location, tracker_pricing_structure, tracker_voter_file_source, tracker_n_questions, disposition_date. This is the level directly beneath disposition_summary: where summary rolls every phone up to a single screening row, records hands back the raw per-campaign rows – for inspection, export, or a custom rollup.

Usage

```
disposition_records(
  dataset,
  phones = NULL,
  campaign_ids = NULL,
  date_from = NULL,
  date_to = NULL,
  page = NULL,
  page_size = NULL
)
```

Arguments

dataset	Path to a disposition Parquet file (the read projection), e.g. from disposition_pull.
phones	Optional character vector of phone numbers to keep. Matched digit-normalized (a leading US 1 is dropped so 11-digit numbers match 10-digit ones). NULL (default) returns every row.
campaign_ids	Optional vector; keep only rows for these campaigns.
date_from, date_to	Optional Date/date-string bounds on disposition_date. A row with an NA disposition date is dropped by any bound. Supplying a bound when the projection has no disposition_date column at all is an error.
page, page_size	Optional 1-based pagination over the (phone, campaign_id)-ordered rows.

Details

Only the canonical columns *present in the file* are returned, in the order above – a legacy or minimal projection that lacks a column (e.g. error, tracker_loi, tracker_topic, or disposition_date) omits it, rather than filling an all-NA column. A projection written straight from disposition_run carries the funnel flags (including the refused / ineligible terminal split as of 0.51.0; a pre-0.51.0 projection omits those two) plus survey_mode (the DETECTED mode), error (the carrier delivery-error code), carrier (the recipient's mobile carrier, from the export's optional misc column), and disposition_date (max(scriptDate)); the tracker_* dimensions (loi, topic, mode – the DECLARED Tracker mode – registration_id, client, project, brand, state, vendor, fielding_location, pricing_structure, voter_file_source, n_questions) are added by the tracker enrichment, so only the enriched projection carries all 27. disposition_date is NA for a row with no send; error is NA when the export carries no usable code (a clean send, or an export lacking the column). phone is digit-normalized for matching, and a stored row whose phone is blank or unparseable is dropped. With **duckdb** installed, phones, campaign_ids, date_from and date_to all filter inside the Parquet scan, so only the matching rows are read (one campaign of the production projection in seconds and under 2 GB; see disposition_screen, Memory); otherwise the whole projection is read into memory and filtered in R (tens of GB).

Two differences from the per-phone rollup follow from the raw grain: a screened phone that was never contacted has **no** row here (there is no stored record to return, unlike the never_contacted row summary synthesises), and there is no statuses argument – that selects a per-phone latest_disposition, which exists only after the rollup.

Value

A data frame, one row per (phone, campaign_id), with the canonical disposition columns present in the file (see Details), ordered by phone then campaign_id.

See Also

disposition_summary (the per-phone rollup), disposition_screen, disposition_pull

Examples

```
## Not run:
dataset <- disposition_pull()
```

```
disposition_records(dataset, phones = my_sample$phone)

## End(Not run)
```

`disposition_run` *Build the per-respondent disposition frame for one campaign*

Description

Turns an in-memory campaign results CSV (one row per respondent) into a list carrying the per-respondent disposition frame in `consolidated` (one row per contacted phone, with 0/1 funnel flags `sent`, `engaged`, `opted_in`, `completed`, `web_complete`, `refused`, `ineligible`, `terminated` (the union of the two), the campaign's `survey_mode`, the raw carrier delivery-error code `error`, the recipient's mobile `carrier` (from the export's optional `misc` column, `NA` when the export omits it), and the `disposition_date` (the last-send day, `max(scriptDate)`)) plus source provenance in `meta`. Pure function, no I/O – pair with `s160_gcs_campaign_results_read(hash = TRUE)` for the GCS source. Persisting the frame (any enrichment, provenance, and Parquet output) is handled by consumer projects.

Usage

```
disposition_run(
  campaign_id,
  data,
  population = NULL,
  contacted_only = TRUE,
  field_timezone = "America/New_York"
)
```

Arguments

`campaign_id`	Campaign id (numeric or character). Stamped on every row as an integer.
`data`	In-memory campaign results CSV as a data frame (one row per respondent). Must contain a phone column.
`population`	Optional population-filter expression defining `opted_in`. `NULL` (default) is routing-based: a recipient opted in when the opener routed them FORWARD – i.e. they reached a continuation step (any non-opener, non-terminal `scriptDate`: a survey-body question or the close family). This is language- and phrasing-agnostic (it reads no answer text), so a Spanish or non-"Yes" opt-in still counts, where the legacy `id.intro.finalText == "Yes"` match silently read 0. Pass an explicit filter to decide `opted_in` by a specific answer instead. A completion is also treated as an opt-in (`opted_in` is OR-ed with `completed`), so `opted_in >= completed` in every mode – this matters for `t2w` campaigns whose web link sits in the intro, where a web completion is the only opt-in evidence. The latency view resolves `n_opted_in` from the same routing signal and the same fold, so the two views agree.

`contacted_only` A single logical. When `TRUE` (default), return only contacted records (rows where `sent == 1`). When `FALSE`, return one row per input respondent.

`field_timezone` IANA timezone (a name in `OlsonNames()`; default `"America/New_York"`) the `disposition_date` is bucketed to; an unknown zone is rejected rather than silently mis-bucketed. Each `id.<step>.scriptDate` send timestamp is stored naive-UTC; the row-wise max is converted to this zone before its calendar date is taken – matching the latency view's send-date bucketing and the live DB producer's `lastsms::date`.

Details

By default (`contacted_only = TRUE`) the frame holds only records that were actually contacted – rows where an intro was dispatched (`sent == 1`). A never-attempted record has no disposition to report, so it is excluded; non-responses (contacted but no reply) are kept. Pass `contacted_only = FALSE` to emit one row per input respondent instead.

Grain: one row per `(phone, campaign_id)`. Phone is unique within a campaign export. Fully-identical duplicate rows (a common export artifact – a re-export appended to itself, a batch duplicated upstream) are collapsed first, since an exact duplicate carries no information the first copy does not; the function then *stops* if a duplicate phone remains – i.e. a phone whose rows *differ* – rather than silently merging a genuine conflict. The dedup and survey-mode classification always run on the full data, so the `contacted_only` filter never changes `survey_mode` or masks a conflict.

The `completed` flag is survey-mode dependent: for a `t2w` campaign it is the `web_complete` callback; for `sms` it is reaching the close – any close-family `scriptDate` (`id.close.scriptDate` / `id.close_sp.scriptDate` / ...), so a bilingual campaign's Spanish completers count; for `t2w_external` it is not computable and is `NA` for every row. `survey_mode` is classified per campaign from the data.

Value

A list mirroring `latency_run()`'s shape: `consolidated` (a data frame, one row per (contacted) respondent, with columns `phone` (character), `campaign_id` (integer), the 0/1 integer flags `sent`, `engaged`, `opted_in`, `completed`, `web_complete`, `refused`, `ineligible`, `terminated` (`refused | ineligible`) – `completed` is `NA` under `t2w_external` – `survey_mode` (character), `error` (character; the raw carrier delivery-error code, `NA` when the export carries no usable error code), `carrier` (character; the recipient's mobile carrier from the export's optional `misc` column, surrounding whitespace trimmed, `NA` when the export omits it or the value is blank), and `disposition_date` (a `Date`: the row-wise max `id.<step>.scriptDate` bucketed to `field_timezone` – the last outbound send – or `NA` when no send time survives); under the default `sent` is `1` for every row) and `meta` (the source `source_csv_hash` / `source_csv_path`, or `NA`). A zero-row input, or a campaign where nobody was contacted, yields a zero-row `consolidated` frame.

Examples

```
## Not run:
data <- s160_gcs_campaign_results_read(1234, hash = TRUE)
res  <- disposition_run(1234, data)
res$consolidated  # the disposition frame; res$meta carries source provenance

## End(Not run)
```

`disposition_screen` *Screen a sample against the disposition dataset*

Description

Takes a sample data frame (a phone column plus whatever else – strata, quota cells, ...) and returns it **unchanged with the disposition summary columns appended**, 1:1 with the input rows and preserving the original phone formatting. The caller then filters and writes it (dropping the numbers already completed or refused). Mirrors the "append columns to my uploaded list" screening workflow.

Usage

```
disposition_screen(
  sample,
  dataset,
  phone_col = "phone",
  campaign_ids = NULL,
  date_from = NULL,
  date_to = NULL,
  columns = "summary",
  as_of = Sys.Date(),
  undeliverable_codes = .DISPOSITION_UNDELIVERABLE_CODES
)
```

Arguments

`sample`	A data frame with a phone-number column.
`dataset`	Path to a disposition Parquet file (the read projection).
`phone_col`	Name of the phone column in `sample` (default `"phone"`).
`campaign_ids, date_from, date_to`	Optional scoping of the disposition rows considered (see `disposition_summary`). No `statuses` or pagination here – every sample row is returned.
`columns`	Which blocks to append: `"summary"` (default; the `disposition_summary` columns), `"history"` (the next-contact history block, see Details), or both.
`as_of`	The date the `days_since_*` columns count to (default today); a `Date` or date string. Only read with `"history"`.
`undeliverable_codes`	Delivery-error codes that make a no-reply campaign `undeliverable` in the history block, as in `disposition_steps`; `NULL` treats every no-reply row as a plain non-response. Only read with `"history"`.

Details

History. `columns = "history"` (or `c("summary", "history")`) appends instead of, or as well as, the counts what each number's contact history says about its *next* send: the `prior_completes`, `prior_replies`, `prior_unanswered` counts, the `history_segment` (`first_contact` for a number with no campaign in scope, else `completer_2_plus`, `completer_1`, `replied`, `unanswered_3_plus` or `unanswered_1_2`), and the recency columns `steps_since_reply` / `steps_since_complete` / `steps_since_undeliverable` (campaigns since the last one; `0` when it was the latest campaign) and `days_since_reply` / `days_since_complete` / `days_since_undeliverable` (`as_of` minus that campaign's `disposition_date`), `NA` until the event has happened. They are the columns `disposition_steps(columns = "history")` carries per step, with the same names and definitions, computed here as of the campaign that would come next, so a rule measured on the steps table applies to a sample unchanged. On the 2026 production rows the segments complete at about 17% (`completer_2_plus`), 9% (`completer_1`), 1.1% (`replied`), 0.55% (`first_contact`), 0.33% (`unanswered_1_2`) and 0.17% (`unanswered_3_plus`) of sends, and a number rejected by the carrier (`undeliverable`) starts replying again only after about two years; the usual rule drops `unanswered_3_plus` and a recent undeliverable and sends the completers first (see Examples). A campaign's letter follows `disposition_steps`: a no-reply row whose `error` is one of `undeliverable_codes` is undeliverable, every other label but `non_response` is a reply, `completed` / `web_complete` a complete; a text-to-web external send (`completed` `NA`) is a reply whose completion is unknown. The `campaign_ids` / `date_from` / `date_to` scope applies to the history too (a number whose campaigns all fall outside it is at its `first_contact`).

Value

`sample` with the `disposition_summary` columns appended (see there for their meaning and order): `n_campaigns`, `campaigns`, `n_engaged`, `n_opted_in`, `n_completed`, `n_web_complete`, `n_terminated`, `n_refused`, `n_ineligible`, `n_error`, `latest_disposition`, `latest_campaign_id`, `best_disposition`, `best_campaign_id`, `first_disposition_date`, `last_disposition_date`. A valid phone that is absent from the rows selected by `campaign_ids`, `date_from`, and `date_to` (the whole dataset when those are unset) gets a `never_contacted` row (`n_campaigns = 0`, the `n_*` counts `0`, the dates `NA`, `latest_disposition = "never_contacted"`, `campaigns = NA`); only a phone that digit-normalizes to nothing (blank/unparseable) gets an all-`NA` block. With `"history"` in `columns` the block `prior_completes`, `prior_replies`, `prior_unanswered`, `history_segment`, `steps_since_reply`, `steps_since_complete`, `steps_since_undeliverable`, `days_since_reply`, `days_since_complete`, `days_since_undeliverable` follows (a never-contacted phone: counts `0`, `first_contact`, recency `NA`); with `columns = "history"` alone it is the only block.

Memory

With the **duckdb** package installed (Suggests; used automatically), the sample's phone match runs inside the Parquet scan, so only the sample's rows are read into R: screening the full production projection (over a hundred million rows) takes seconds and well under 1 GB of RAM. Without it, reading a large projection (over 100 MB) on the default `"auto"` engine would load the whole file into R (tens of GB), so survey160r instead offers **duckdb**: an interactive session prompts to install it, a non-interactive one stops with the same guidance. `options(survey160r.disposition_engine = "nanoparquet")` takes the whole-file **nanoparquet** read instead (no prompt); `"duckdb"` errors if the package is not installed, and the default `"auto"` uses duckdb when installed. The DuckDB-written production projection always takes the duckdb path; a few unusual files (a path with glob

characters, a file written by arrow or nanoparquet, a column type outside the plain set) are read with nanoparquet. The result is identical either way.

See Also

disposition_summary, disposition_records, opt_out_screen

Examples

```
# Against a tiny local projection (no sign-in), so this runs. In practice the
# dataset is the path disposition_pull() returns.
dataset <- tempfile(fileext = ".parquet")
nanoparquet::write_parquet(data.frame(
  phone = c("5551234567", "5551234567", "5559876543"),
  campaign_id = c(101L, 102L, 101L),
  engaged = c(1L, 1L, 0L), opted_in = c(1L, 0L, 0L), completed = c(1L, 0L, 0L),
  web_complete = c(0L, 0L, 0L), refused = c(0L, 1L, 0L), ineligible = c(0L, 0L, 0L),
  terminated = c(0L, 1L, 0L),
  disposition_date = as.Date(c("2026-01-10", "2026-01-20", "2026-01-15"))
), dataset)
my_sample <- data.frame(phone = c("5551234567", "5559876543", "5550000000"),
                        stratum = c("A", "B", "A"))
cleaned <- disposition_screen(my_sample, dataset)
cleaned[, c("phone", "stratum", "n_campaigns", "n_completed", "n_refused",
            "latest_disposition")]
# keep numbers not already finished and not a hard stop
subset(cleaned, !((n_completed > 0 | n_web_complete > 0) %in% TRUE |
                    (n_terminated > 0 | n_refused > 0 | n_ineligible > 0) %in% TRUE))
# the history block: what each number's next send would carry
history <- disposition_screen(my_sample, dataset, columns = "history",
                              as_of = "2026-02-01")
history[, c("phone", "history_segment", "prior_completes",
            "steps_since_complete", "days_since_complete")]
## Not run:
dataset <- disposition_pull()
cleaned <- disposition_screen(my_sample, dataset, phone_col = "phone")
# drop finished (completed/web) and every hard stop (refused / ineligible /
# terminated); blank-phone rows come back all-NA and are kept
subset(cleaned, !((n_completed > 0 | n_web_complete > 0) %in% TRUE |
  (n_terminated > 0 | n_refused > 0 | n_ineligible > 0) %in% TRUE))

# the history rule: skip numbers that never answered three or more campaigns
# or were rejected by the carrier in the last two years, then send the
# completers first
cleaned <- disposition_screen(my_sample, dataset, columns = c("summary", "history"))
keep <- subset(cleaned, !(history_segment == "unanswered_3_plus") %in% TRUE &
  !(days_since_undeliverable < 730) %in% TRUE)
keep[order(keep$history_segment != "completer_2_plus",
           keep$history_segment != "completer_1"), ]

## End(Not run)
```

`disposition_steps` *Read the disposition dataset as a sequence: one row per campaign, with history*

Description

Orders each phone's campaigns in time and returns one row per (`phone`, `campaign_id`) carrying the phone's *history up to that campaign*: its position (`step` of `n_campaigns`), a single disposition `label`, the previous and next labels and the gaps between them, running counts, the length of the current run (e.g. the third non-response in a row), and the history itself as a letter path. Filter, group and summarise it with **dplyr** verbs to answer questions that depend on what came before – "after three non-responses in a row, how often does the next campaign get a reply?", "once a number has completed twice, how often does it complete again?". For one row per phone and sequence patterns (non-response, then engaged, then completed) see `disposition_paths`.

Usage

```
disposition_steps(
  dataset,
  phones = NULL,
  sample_frac = NULL,
  campaign_ids = NULL,
  date_from = NULL,
  date_to = NULL,
  columns = c("neighbors", "so_far", "run_length", "path_so_far"),
  pattern = NULL,
  terminal = c("raw", "engaged"),
  undeliverable_codes = .DISPOSITION_UNDELIVERABLE_CODES
)
```

Arguments

`dataset`	Path to the disposition Parquet file, e.g. from `disposition_pull`.
`phones`	Optional character vector of phone numbers; keeps only these phones (all of their campaigns). Matched digit-normalized, as the other readers do. `character(0)` returns no rows.
`sample_frac`	Optional number in (0, 1]: keep a deterministic sample of that fraction of phones (by an md5 of the phone, so the same phones on every machine and DuckDB version). `0.1` makes full-projection exploration ~10x cheaper; the 10% sample matches the full data to within 0.02 points on prod.
`campaign_ids`	Optional vector; keep rows of these campaigns (history still covers all campaigns).
`date_from, date_to`	Optional `Date`/date-string bounds on `disposition_date` (inclusive); keep rows in range (history still covers all campaigns).

`columns` Which history column groups to compute: any of `"neighbors"` (`prev_label`, `next_label`, `next_date`, `gap_prev_days`, `gap_next_days`), `"so_far"` (`completes_so_far`, `non_responses_so_far`, `replies_so_far`, counting this step), `"run_length"` (how many steps in a row, ending here, have this label), `"path_so_far"` (the history as letters, ending with this step) and `"history"` (`prior_*` counts, the `history_segment`, and the `steps_since_*` / `days_since_*` recency columns; see below). The default is the first four; `"history"` is opt-in. Each costs a few seconds on the full projection.

`pattern` Optional sequence pattern (see `disposition_paths` for the syntax); adds `history_matched`: whether the history up to and including this step matches. Use a `$` anchor for "history ends with", e.g. `"non_response|undeliverable{3,}$"`.

`terminal` How to label an unsplit `terminated` row (terminated but neither refused nor ineligible): `"raw"` (default) keeps `terminated`, matching `disposition_summary`; `"engaged"` labels it by the step it reached (`opted_in` or `engaged`). On prod most of these rows come from a producer change in August 2026 rather than respondent behaviour.

`undeliverable_codes` Delivery-error codes that make a no-reply row `undeliverable` rather than `non_response`. Defaults to the hard carrier rejections (4700, 4701, 4702, 4720, 4750, 4753); `NULL` labels every no-reply row `non_response`, as the summary does.

Details

A lazy table. The result is a **dbplyr** table that DuckDB computes when you `collect()` (or print) it, not a data frame: the history columns are window functions over every campaign a phone has, which on the production projection (~165M rows) only fit inside DuckDB. Aggregate before you `collect()` – the full table is about 16 GB in R – or scope it with `phones` / `sample_frac`. Rows come back unordered; `arrange()` when order matters. All tables share one session-wide DuckDB connection (see Resources).

Order and labels. A phone's campaigns are ordered by `disposition_date` (its last send in that campaign), then `campaign_id` compared as a number; a missing date or id counts as the oldest, as in `disposition_summary`, so a phone's last step is its `latest_disposition`. Campaigns that overlapped in time can come out in either order (0.5% of consecutive pairs on prod, all under 30 days apart). Each row gets the furthest step it reached, by the precedence of `disposition_summary`'s `latest_disposition`: `web_complete`, `completed`, `refused`, `ineligible`, `terminated`, `opted_in`, `engaged`, then `undeliverable` (no reply and a delivery-error code in `undeliverable_codes`) and `non_response`. Path letters: N non_response, X undeliverable, E engaged, O opted_in, T terminated, I ineligible, R refused, C completed, W web_complete.

History segments. With `columns = "history"` each row also carries what the phone did *before* this campaign – `prior_completes`, `prior_replies` (any label but non_response / undeliverable) and `prior_unanswered` (the rest, so `step - 1 - prior_replies`) – and `history_segment`, the first of: `first_contact` (step 1), `completer_2_plus`, `completer_1`, `replied` (replied before, never completed), `unanswered_3_plus`, `unanswered_1_2`. On the 2026 production rows the segments complete at about 17%, 9%, 1.1%, 0.55% (first contact), 0.17% and 0.33% of sends, so `unanswered_3_plus` and a previous `undeliverable` are the usual suppression rule and the completers the usual priority; group by it to reproduce those rates on a newer projection.

Recency. The same group carries how long ago the phone last did something, counted both in campaigns and in days: `steps_since_reply`, `steps_since_complete`, `steps_since_undeliverable` (how many campaigns since the last reply / complete / undeliverable before this one; `0` means the previous campaign was one) and `days_since_reply`, `days_since_complete`, `days_since_undeliverable` (this step's `disposition_date` minus that campaign's). All `NA` until the event has happened. On the 2026 production rows campaigns, not days, are what predicts the next outcome: a number completes 20% of the time when its previous campaign was a complete, 13% / 10% / 8% / 5% after one / two / three / four unanswered campaigns in between and 3% after five, whatever the calendar gap; after an undeliverable the next reply rate climbs from 0.3% within a week to 3.4% past two years, so `days_since_undeliverable` is the one day count worth a rule.

Filters. `phones` and `sample_frac` keep or drop whole phones *before* the history is computed. `campaign_ids`, `date_from` and `date_to` drop rows *after* it, so a kept row still carries its full history (a 2026 row counts the phone's 2025 non-responses).

Resources. The connection uses a 4 GB DuckDB `memory_limit` and spills to disk past it (a full-projection query can spill ~20 GB, freed when it finishes); set `options(survey160r.duckdb_memory = "8GB")` or `options(survey160r.duckdb_temp_dir = "/path")` to change either (a new spill directory applies only before the session's first spill; each session spills into its own subdirectory, removed when the session ends). Measured on the production projection (~165M rows): a full-scan aggregate in ~16 s at ~5.5 GB of process memory; a phone-scoped table (up to millions of phones) in 1-5 s. Each distinct `phones` vector stays registered on the connection for the session, so later operations on the table keep working.

Value

A lazy **dbplyr** table, one row per `(phone, campaign_id)`: `phone`, `campaign_id`, `disposition_date`, `step`, `n_campaigns`, `label`, `completion_known` (`FALSE` where `completed` is `NA`, i.e. text-to-web external, where a complete is never reported), the requested history columns, `history_matched` (with `pattern`), then every other stored column (funnel flags, `survey_mode`, `carrier`, `error`, `tracker_*`, ...).

See Also

`disposition_paths` (one row per phone), `disposition_summary` (per-phone counts), `disposition_records` (the stored rows)

Examples

```
## Not run:
library(dplyr)
dataset <- disposition_pull()
steps <- disposition_steps(dataset, sample_frac = 0.1)

# Completion of sends in 2026 by what the number did before (history still spans 2021+)
disposition_steps(dataset, date_from = "2026-01-01", columns = "history") |>
  filter(completion_known) |>
  group_by(history_segment) |>
  summarise(sends = n(), completion = mean(as.integer(
    label %in% c("completed", "web_complete")))) |>
  arrange(desc(completion)) |>
  collect()
```

```
# Completion by how many campaigns ago the number last completed
disposition_steps(dataset, date_from = "2026-01-01", columns = "history") |>
  filter(completion_known, !is.na(steps_since_complete)) |>
  group_by(since = pmin(steps_since_complete, 5L)) |>
  summarise(sends = n(), completion = mean(as.integer(
    label %in% c("completed", "web_complete")))) |>
  arrange(since) |>
  collect()

# Reply rate on the next campaign after k non-responses in a row
steps |>
  filter(label == "non_response", !is.na(next_label)) |>
  group_by(run = pmin(run_length, 5L)) |>
  summarise(n = n(), reply_next = mean(as.integer(
    !next_label %in% c("non_response", "undeliverable")), na.rm = TRUE)) |>
  arrange(run) |>
  collect()

## End(Not run)
```

`disposition_summary` *Summarize the disposition dataset to one row per phone*

Description

Rolls the disposition data up to **one row per phone** – each number's cross-campaign status counts, date span, and latest/best disposition. Pass either the projection **path** (read it, then summarize) or an **in-memory frame** already read with `disposition_records` (summarize it directly, no I/O), which lets you read once and summarize several phone lists. For cleaning a sample file in place, use `disposition_screen`; for the raw rows behind the rollup (one per `(phone, campaign_id)`), use `disposition_records`.

Usage

```
disposition_summary(
  x,
  phones = NULL,
  campaign_ids = NULL,
  statuses = NULL,
  date_from = NULL,
  date_to = NULL,
  page = NULL,
  page_size = NULL
)
```

Arguments

x	Either a path to a disposition Parquet file (the consolidated read projection, e.g. from disposition_pull) or an in-memory disposition data frame (from disposition_records). A path is read projected to the summary columns (only the phones rows when given and **duckdb** is installed; see disposition_screen, Memory); a frame must carry phone, campaign_id, engaged, opted_in, completed, web_complete, and terminated. refused and ineligible (the terminal split) are optional – absent (a pre-0.51.0 projection) they default to 0, so terminals summarize as plain terminated; present, they split it (see statuses / Value). disposition_date is optional – it orders each phone's latest campaign and backs the date_from/date_to filters; an un-enriched disposition_records frame that omits it still summarizes (disposition dates treated as unknown), but a date bound then errors.
phones	Optional character vector of phone numbers to screen. When supplied, **every** input number is returned – never-contacted ones with n_campaigns = 0 and latest_disposition = "never_contacted". NULL (default) summarizes every phone present. Matched digit-normalized (a leading US 1 is dropped so 11-digit numbers match 10-digit ones).
campaign_ids	Optional vector; restrict the underlying rows to these campaigns before summarizing.
statuses	Optional subset of the derived disposition categories (never_contacted, non_response, engaged, opted_in, terminated, ineligible, refused, completed, web_complete); keep only phones whose latest_disposition is one of them. ineligible (screened out) and refused (declined) are the split of terminated; terminated now denotes only the unsplit residual (a terminal the routing name-match could not classify), so screen on all three to catch every hard stop.
date_from, date_to	Optional Date/date-string bounds on disposition_date. A row whose disposition_date is NA is dropped by any bound (an all-NA column drops every row, with a warning). A projection with **no** disposition_date column is a different case: setting a bound then **errors** (see x); it is not a silent drop.
page, page_size	Optional 1-based pagination over the per-phone result.

Value

A data frame, one row per phone, columns in this order: phone; n_campaigns and campaigns (how many campaigns, and the comma-separated id list); the cumulative status counts n_engaged, n_opted_in, n_completed, n_web_complete, n_terminated (campaigns the producer flagged as a hard stop), then n_refused and n_ineligible (the name-classified terminals; each 0 on a pre-0.51.0 projection; they need not sum to n_terminated, since a downstream projection can set terminated independently of the split – less when a terminal was unsplit, more when the split flagged a terminal the status had not) – each 0 when the phone never reached that status and > 0 the number of the phone's campaigns that did (they overlap: a completed campaign is also engaged) – plus n_error (how many campaigns carried a carrier delivery-error code); latest_disposition + latest_campaign_id (the category of the phone's most-recent campaign and that campaign's id – now one of refused / ineligible / terminated for a hard stop); best_disposition + best_campaign_id (the furthest-reached category across all the phone's campaigns – ranked by

the same funnel precedence, so `completed` / `web_complete` rank highest and the terminal band (`terminated` < `ineligible` < `refused`) above `opted_in` – and the campaign that reached it); and `first_disposition_date` / `last_disposition_date` (earliest and latest `disposition_date` across the phone's campaigns, `NA` when none is dated). A never-contacted phone has `n_campaigns = 0` and `campaigns = NA`; a contacted phone whose campaign ids are all `NA` has `campaigns = ""`. Campaign ids are returned as character.

The counts assume the dataset's grain – one row per `(phone, campaign_id)` – and count rows: if two stored rows normalize to the same phone in the same campaign (e.g. `"5551234567"` and `"1 555 123 4567"`), each counts, so an `n_*` count can exceed `n_campaigns`. The "max campaign id" tie-break expects numeric ids (as the projection stores them); non-numeric ids tie-break by row order.

See Also

`disposition_screen`, `disposition_records`, `disposition_pull`

Examples

```
# On an in-memory frame (no I/O), so this runs:
records <- data.frame(
  phone = c("5551234567", "5551234567", "5559876543"),
  campaign_id = c(101L, 102L, 101L),
  engaged = c(1L, 1L, 0L),
  opted_in = c(1L, 0L, 0L),
  completed = c(1L, 0L, 0L),
  web_complete = c(0L, 0L, 0L),
  terminated = c(0L, 1L, 0L),
  disposition_date = as.Date(c("2026-01-10", "2026-01-20", "2026-01-15")),
  stringsAsFactors = FALSE
)
disposition_summary(records, phones = c("5551234567", "5550000000"))
## Not run:
# Or straight from the projection on disk:
dataset <- disposition_pull()
disposition_summary(dataset, phones = my_sample$phone)

## End(Not run)
```

---

funnel_rates *Add the standard funnel rates to a counts frame*

Description

The single definition of the Survey160 funnel rates, so every consumer computes them the same way. Given a frame carrying the summed funnel counts, appends four columns: `engagement_rate` (`engaged` / `sent`, a first reply), `opted_in_rate` (`opted_in` / `sent`, opt-ins as a share of everyone sent – the funnel-anchored rate), `opted_in_engaged_rate` (`opted_in` / `engaged`, opt-in conversion among the engaged), and `completion_rate` (`completed` / `sent`). The two opt-in rates answer

different questions – end-to-end yield per send vs conversion once a recipient replies – so both are emitted. A rate with a zero denominator is NA (not NaN/Inf). Used by campaign_metrics_summary and disposition_funnel, and usable on any counts frame.

Usage

```
funnel_rates(
  x,
  sent = "n_sent",
  engaged = "n_engaged",
  opted_in = "n_opted_in",
  completed = "n_completed",
  percent = FALSE
)
```

Arguments

x	A data frame with the four count columns named by the arguments below (defaults match campaign_metrics_records).
sent, engaged, opted_in, completed	Column names of the funnel counts.
percent	When FALSE (default) rates are proportions in [0, 1]; TRUE scales them to 0-100.

Value

x with engagement_rate, opted_in_rate, opted_in_engaged_rate, and completion_rate appended (same type as x).

See Also

campaign_metrics_summary

Examples

```
df <- data.frame(n_sent = c(1000, 500), n_engaged = c(80, 0),
                 n_opted_in = c(20, 0), n_completed = c(5, 0))
funnel_rates(df, percent = TRUE)
```

---

latency_build_config *Build a latency config from a campaign export*

Description

Pure function. Derives flow.questions from the CSV column names via latency_discover_questions() and assembles the rest of the config from the named arguments. No I/O, no API call, no auth precondition.

Usage

```
latency_build_config(
  campaign_id,
  data,
  field_timezone = "UTC",
  project_id = NULL,
  date_filter = NULL,
  respondent_id_column = NULL
)
```

Arguments

campaign_id	Campaign id (numeric or character).
data	A data frame of CSV results (or a character vector of column names) used to discover the question flow.
field_timezone	Tz used to bucket the Parquet date and hour_local columns. Default "UTC".
project_id	Optional Survey160 project id; defaults to the campaign id as a placeholder.
date_filter	Optional character/Date vector restricting which survey dates are processed (interpreted in field_timezone).
respondent_id_column	Optional column name used to dedupe rows by respondent. Default NULL (no dedupe).

Value

A config list ready to pass to latency_report(), which calls latency_validate_config() before consuming it.

Examples

```
config <- latency_build_config(
  campaign_id = 1234,
  data = c("id.intro.scriptDate", "id.intro.batchDate",
           "id.q1.scriptDate", "id.q1.batchDate", "id.close.scriptDate"),
  field_timezone = "America/New_York"
)
config$flow$questions
```

---

latency_config_hash *Hash a latency config*

Description

Hashes a canonical form of the config so the same logical config always produces the same hash even across different YAML serializations.

Usage

```
latency_config_hash(config)
```

Arguments

config	The config list.

Value

A hex sha256 string.

Examples

```
config <- latency_build_config(
  1234,
  c("id.intro.scriptDate", "id.intro.batchDate", "id.q1.scriptDate")
)
latency_config_hash(config)
```

---

latency_discover_questions

Discover the question flow from export column names

Description

Scans the column names of an in-memory CSV data frame (as returned by `read.csv` / `s160_gcs_campaign_results_read`) for `id.<q>.scriptDate` columns and returns the question ids in their original column order. Terminal flow states (`refusal`, `ineligible`) are dropped so the result is usable directly as `config$flow$questions`.

Usage

```
latency_discover_questions(data)
```

Arguments

data	A data frame or character vector of column names.

Details

Survey160 v2 CSV headers are emitted as `id[<q>]scriptDate` on disk; `read.csv` converts the brackets to dots. Both forms are accepted so callers can pass either a data frame or a character vector of raw header tokens.

Value

A character vector of question ids in flow order.

Examples

```
latency_discover_questions(
  c("campaignid", "id.intro.scriptDate", "id.intro.batchDate",
    "id.q1.scriptDate", "id.q1.batchDate", "id.close.scriptDate")
)
```

`latency_funnel` *Reduce a latency report to funnel counts*

Description

`latency_run()` / `latency_report()` return a `consolidated` table whose funnel anchor columns (`n_sent`, `n_engaged`, `n_opted_in`, `n_completed`) are denormalised: one value per (`campaign_id`, `date`, `hour_local`) bucket is repeated across every (`segment`, `threshold_min`) row of that bucket, and the frame holds two grains at once (per-hour rows plus a day-rollup row with `hour_local = NA`). Summing an anchor column over the raw frame multi-counts. This accessor collapses the `segment` x `threshold` fan-out and selects a single grain, returning one row per bucket with the funnel anchors carried through unchanged.

Usage

```
latency_funnel(consolidated, grain = c("day", "hour"))
```

Arguments

`consolidated`	The `consolidated` element of a `latency_run()` / `latency_report()` result (a data frame; a `data.table` or tibble is accepted and coerced).
`grain`	One of `"day"` (default) or `"hour"`. `"day"` reads the day-rollup rows (`hour_local = NA`) and returns one row per (`campaign_id`, `date`) – use it for daily funnel counts; there is no need to sum the hourly result. `"hour"` reads the per-hour rows and returns one row per (`campaign_id`, `date`, `hour_local`) for a time-of-day view. The two grains are never mixed (that would double-count). Because every funnel anchor is send-anchored (a recipient with no parseable send is excluded from all four counts, so none land in an unknown-hour bucket), summing an anchor across the `"hour"` result equals the matching `"day"` value.

Details

Pass the whole thing across campaigns: the reduction keys on `campaign_id`, so a fleet-wide `consolidated` (many campaigns) reduces correctly. When a `consolidated` carries more than one fleet pass (differing `run_at_utc`) per campaign, de-duplicate to the pass you want before calling; this function does not pick a pass for you.

Value

A data frame with `campaign_id`, `date`, `hour_local` (only when `grain = "hour"`), and the funnel anchors `n_sent`, `n_engaged`, `n_opted_in`, `n_completed`, sorted by campaign then date (then hour). For a `"t2w_external"` campaign `n_completed` is `NA` (completion is off-platform), inherited from the source frame.

Examples

```
data <- data.frame(
  campaignid = c(1L, 1L),
  id.intro.finalText = c("Yes", "Yes"),
  id.intro.scriptDate = c("2026-01-26 21:00:00Z", "2026-01-26 22:05:00Z"),
  id.intro.batchDate  = c("2026-01-26 21:00:30Z", "2026-01-26 22:05:20Z"),
  id.q1.scriptDate    = c("2026-01-26 21:01:00Z", "2026-01-26 22:06:00Z"),
  id.q1.batchDate     = c("2026-01-26 21:01:20Z", "2026-01-26 22:06:15Z"),
  id.close.scriptDate = c("2026-01-26 21:02:00Z", "2026-01-26 22:07:00Z"),
  check.names = FALSE, stringsAsFactors = FALSE
)
config <- latency_build_config(1L, data, field_timezone = "America/New_York")
result <- latency_report(data, config, run_at = as.POSIXct("2026-01-01", tz = "UTC"))
latency_funnel(result$consolidated)              # one row per (campaign, date)
latency_funnel(result$consolidated, grain = "hour")  # one row per hour
```

---

latency_input_columns *List the export columns the latency report reads*

Description

Returns the (dot-form) column names `latency_report()` touches for a given `config`: the per-question `scriptDate`/`batchDate` set, the population-filter columns (extracted from `config$filters$population`), the campaign-id and optional respondent-id columns, plus the fixed non-flow support columns (`web_complete`, `id.ineligible.scriptDate`, `id.refusal.scriptDate`). The opener `finalText` is not fixed here: it is one of the population-filter columns above, so it is retained per campaign only when the population references it.

Usage

```
latency_input_columns(config, available = NULL)
```

Arguments

`config`	A config list from `latency_build_config()` (or one with the same shape). Build it from a header-only peek (`s160_csv_header()`) to avoid reading the file twice.
`available`	Optional character vector of the actual (dot-form) column names present in the file (e.g. from `s160_csv_header()`). When supplied, columns matching the report's data-dependent patterns (the close-message Text columns) are retained. Strongly recommended.

Details

Some columns the report reads have data-dependent names – the close-message Text columns (`id.close*.scriptText`/`batchText`) that `detect_survey_mode()` greps to tell `t2w_external` from `sms`. Pass `available` (e.g. the result of `s160_csv_header()`) so these are matched against the

real header and retained; omitting it risks projecting them away and misclassifying a `t2w_external` campaign as `sms`.

Pass the result as `columns =` to `s160_read_csv()` / `s160_gcs_campaign_results_read()` to parse only the columns the algorithm needs – the projection yields output identical to a full read.

Value

A character vector of unique dot-form column names.

Examples

```
## Not run:
header <- s160_csv_header(path)
config <- latency_build_config(1980, header, field_timezone = "America/New_York")
data   <- s160_read_csv(path, columns = latency_input_columns(config, header))

## End(Not run)
```

---

latency_report *Compute a latency report for one campaign*

Description

Pure function of `(data, config, run_at)`: no I/O, no mutable globals. With a supplied `run_at` the output is fully deterministic; left `NULL` it stamps `run_at_utc` from `Sys.time()`. Implements the algorithm in `campaign_scripts.md` section 2.

Usage

```
latency_report(data, config, run_at = NULL, compact = FALSE)
```

Arguments

`data`	A data frame with one row per respondent and the per-question timestamp columns named `id.<q>.scriptDate` / `id.<q>.batchDate` for each question in `config$flow$questions`, plus the population column `id.intro.finalText` and the campaign id column.
`config`	Config list from `latency_build_config()` (or a hand-built list with the same shape).
`run_at`	Optional `POSIXct` timestamp to stamp on every row's `run_at_utc` column. `NULL` (default) uses `Sys.time()`. Fleet runners pass a single timestamp here so every campaign in one fleet pass shares the same `run_at_utc`, making "last fleet run" queries trivial.
`compact`	When `FALSE` (default) the full per-(respondent, segment) long frame is materialised and returned as `latency_frame`. When `TRUE` the report is computed segment-by-segment WITHOUT materialising the long frame: only rows with a real (non-NA) `segment_date_local` are kept, and the dropped NA-date rows (respondents who never reached a question) are summarised as per-segment

counts and rebuilt exactly into the (date=NA, hour_local=NA) day-rollup bucket. consolidated and diagnostics are byte-identical to the default path; only peak memory differs (bounded by real engagement, not by (n_questions - 1) x N), and latency_frame then contains only the kept real-date rows. Use it for very wide, high-volume campaigns whose long frame is mostly NA drop-off.

Value

A list with consolidated (one row per (campaign_id, date, hour_local, segment, threshold_min)), latency_frame (one row per respondent x segment; only the real-date rows when compact = TRUE), diagnostics (counts and breakdowns per spec section 3.3), and meta (algorithm_version, config_hash, run_at_utc).

Examples

```
data <- data.frame(
  campaignid = c(1L, 1L),
  id.intro.finalText = c("Yes", "Yes"),
  id.intro.scriptDate = c("2026-01-26 21:00:00Z", "2026-01-26 21:05:00Z"),
  id.intro.batchDate  = c("2026-01-26 21:00:30Z", "2026-01-26 21:05:20Z"),
  id.q1.scriptDate    = c("2026-01-26 21:01:00Z", "2026-01-26 21:06:00Z"),
  id.q1.batchDate     = c("2026-01-26 21:01:20Z", "2026-01-26 21:06:15Z"),
  id.close.scriptDate = c("2026-01-26 21:02:00Z", "2026-01-26 21:07:00Z"),
  check.names = FALSE, stringsAsFactors = FALSE
)
config <- latency_build_config(1L, data, field_timezone = "America/New_York")
result <- latency_report(data, config, run_at = as.POSIXct("2026-01-01", tz = "UTC"))
head(result$consolidated)
```

---

latency_run *Build the config and run the latency report for one campaign*

Description

User-facing one-campaign runner. Given an in-memory campaign CSV (already read by the caller), (optionally) builds the report config from the CSV header and runs the latency algorithm. No I/O; pair with s160_gcs_campaign_results_read(hash = TRUE) for the GCS source path or read.csv() / readr::read_csv() / anything else for off-GCS sources.

Usage

```
latency_run(
  campaign_id,
  data,
  config = NULL,
  run_at = NULL,
  run_by = NULL,
  compact = FALSE,
  ...
)
```

Arguments

campaign_id	Campaign id (numeric or character).
data	In-memory campaign CSV as a data frame (one row per respondent, columns named id.<q>.scriptDate / id.<q>.batchDate per question plus the population-filter column id.intro.finalText and the campaign id column).
config	Optional pre-built config. When NULL (default), the config is auto-built from data's header; pass any latency_build_config() overrides (field_timezone, project_id, date_filter, respondent_id_column) through .... Mutually exclusive with .... Note field_timezone defaults to "UTC": pass the zone the campaign fielded in (e.g. "America/New_York") or the hour_local buckets are shifted by the offset.
run_at	Optional POSIXct timestamp stamped on every row's run_at_utc column. NULL (default) uses Sys.time(). Fleet runners pass one stamp here so every campaign in a pass shares the same run_at_utc.
run_by	Optional string stamped on every row's run_by provenance column. NULL (default) leaves the column as NA_character_; typically filled at write time by the step that persists the result.
compact	Forwarded to latency_report(). FALSE (default) materialises the full long frame; TRUE computes the identical consolidated / diagnostics without materialising it (bounded peak memory), returning only the real-date rows as latency_frame. Use for very wide, high-volume campaigns. See latency_report.
...	Forwarded to latency_build_config() when config is NULL. Must be empty when config is supplied (passing both errors).

Details

Two call shapes:

- Convenience – omit config and pass any latency_build_config() overrides through .... latency_run() derives the config from the CSV header.
- Custom – pre-build the config with latency_build_config() (mutating as needed) and pass it via config. ... must be empty in that case (passing both errors).

Provenance: if data carries source_csv_hash or source_csv_path attributes (set by s160_gcs_campaign_results_read(= TRUE) for GCS reads), latency_report() surfaces them on result$meta. Callers pulling CSVs from other sources can attach the attributes themselves before calling, e.g.

```
attr(df, "source_csv_path") <- "dropbox:campaign_1234.csv"
attr(df, "source_csv_hash") <- paste0(
  "sha256:", digest::digest(file = local_path, algo = "sha256"))
```

Value

The list returned by latency_report(): consolidated, latency_frame, diagnostics, meta (with source_csv_hash and source_csv_path from data's attributes, or NA when absent).

Examples

```
## Not run:
# GCS source -- pair with s160_gcs_campaign_results_read(hash = TRUE).
s160_gcs_init()
data   <- s160_gcs_campaign_results_read(1980, hash = TRUE)
result <- latency_run(1980, data, field_timezone = "America/New_York")
result$meta$source_csv_hash

# Off-GCS source -- bring your own CSV.
data   <- read.csv("~/Dropbox/campaign_1980.csv", stringsAsFactors = FALSE)
result <- latency_run(1980, data)

# Custom config (mutate before running).
config <- latency_build_config(1980, data, field_timezone = "America/New_York")
config$flow$questions <- c("intro", "q1_custom")
latency_run(1980, data, config = config)

## End(Not run)
```

---

latency_validate_config

Validate a latency config against a campaign export

Description

Implements the fail-fast checks from spec section 2.4. Aborts with a named error on the first failing rule.

Usage

```
latency_validate_config(config, data)
```

Arguments

config	The config list (typically from latency_build_config).
data	The data frame the report will run against.

Value

Invisible TRUE on success; otherwise stops with an error.

Examples

```
data <- data.frame(
  campaignid = 1L,
  id.intro.finalText = "Yes",
  id.intro.scriptDate = "2026-01-26 21:00:00Z",
  id.intro.batchDate  = "2026-01-26 21:00:30Z",
  id.q1.scriptDate    = "2026-01-26 21:01:00Z",
```

```
  id.q1.batchDate     = "2026-01-26 21:01:20Z",
  id.close.scriptDate = "2026-01-26 21:02:00Z",
  check.names = FALSE, stringsAsFactors = FALSE
)
config <- latency_build_config(1L, data, field_timezone = "America/New_York")
latency_validate_config(config, data)
```

`opt_out_pull` *Download the opt-out list from Cloud Storage*

Description

Pulls the opt-out Parquet (`global_opt_out.parquet`) from the environment's disposition bucket to a local file and returns the path – ready to hand to `opt_out_screen`. Downloaded once and reused from the local cache on later calls (pass `refresh = TRUE` to force a fresh download). Parallels `disposition_pull` and reaches GCS the same way: authenticate first with `s160_gcs_init()` so the session's GCS credentials are set. A download without an initialized session errors with "GCS not initialized. Run s160_gcs_init() first." (a cache hit is served without needing auth).

Usage

```
opt_out_pull(
  env = .ENV_CHOICES,
  dest = NULL,
  bucket = NULL,
  refresh = FALSE,
  progress = interactive()
)
```

Arguments

`env`	Environment: `"prod"` (default) or `"dev"`. There is no staging opt-out tier; passing `env = "staging"` errors clearly.
`dest`	Where to save. `NULL` (default) caches under `tools::R_user_dir("survey160r", "cache")`. A directory saves the default filename (`<bucket>.global_opt_out.parquet`) inside it; any other single string is treated as the exact output path (its parent is created).
`bucket`	**Deprecated.** Select data with `env =` instead; a supplied bucket is honored with a warning for back-compat.
`refresh`	When `FALSE` (default), reuse an existing local copy; `TRUE` always re-downloads (the list is republished on every sync, so refresh to pick up a newer one).
`progress`	Show a download progress bar. Defaults to `interactive()`: a live bar in an interactive session, silent in batch or scheduled runs.

Details

The opt-out list shares the environment's `s160_disposition_<env>` bucket with the disposition projection – it is the same phone-keyed PII class – so the two `*_pull()` helpers fetch different objects from one bucket, into distinct cache files.

Value

The local path to the downloaded Parquet (a single string), ready for `opt_out_screen`.

See Also

`opt_out_screen`, `disposition_pull`, `s160_gcs_init`

Examples

```
## Not run:
s160_gcs_init()   # one-time browser OAuth
cleaned <- opt_out_screen(my_sample, opt_out_pull())

## End(Not run)
```

---

opt_out_screen *Screen a sample against the opt-out list*

Description

Takes a sample data frame (a phone column plus whatever else – strata, quota cells, ...) and returns it **unchanged with two opt-out columns appended**, 1:1 with the input rows and preserving the original phone formatting. This function **never drops rows**: the caller decides whether to suppress the opted-out numbers, on its own rules. Parallels `disposition_screen` – chain both to screen a sample on prior disposition and current opt-out status in one pass, then filter once.

Usage

```
opt_out_screen(sample, dataset, phone_col = "phone")
```

Arguments

`sample`	A data frame with a phone-number column.
`dataset`	Path to an opt-out Parquet file (a `phone` column and an optional `date_added`).
`phone_col`	Name of the phone column in `sample` (default `"phone"`).

Value

`sample` with `opted_out` (logical) and `opt_out_date` appended, 1:1 with the input rows and in input order. A valid phone present in the list gets `opted_out = TRUE` and its `date_added` value as `opt_out_date`; a valid phone absent from the list gets `opted_out = FALSE` and `opt_out_date = NA`; a phone that digit-normalizes to nothing (blank/unparseable) gets `opted_out = NA` – unknown, not a claim that it is contactable – and `opt_out_date = NA`. `opt_out_date` carries the list's `date_added` value uninterpreted (`NA` throughout when the list omits that column).

See Also

`disposition_screen`, `opt_out_pull`

Examples

```
# Against a tiny local list (no sign-in), so this runs. In practice the
# dataset is the path opt_out_pull() returns.
optout <- tempfile(fileext = ".parquet")
nanoparquet::write_parquet(
  data.frame(phone = "5559876543", date_added = as.Date("2026-02-01")),
  optout
)
my_sample <- data.frame(phone = c("5551234567", "5559876543", ""),
                        stratum = c("A", "B", "A"))
cleaned <- opt_out_screen(my_sample, optout)
cleaned                                  # columns appended; the blank phone is NA
subset(cleaned, !(opted_out %in% TRUE))  # drop opted-out, keep the NA row to fix
## Not run:
cleaned <- opt_out_screen(my_sample, opt_out_pull())
# chain with disposition, then drop on the caller's own rules:
cleaned <- disposition_screen(cleaned, disposition_pull())
subset(cleaned, !(opted_out %in% TRUE |
                    (n_completed > 0 | n_web_complete > 0) %in% TRUE))

## End(Not run)
```

---

parse_campaign_timestamps

Parse campaign-export timestamps

Description

Parses the timestamp strings found in a Survey160 campaign export (e.g. `id.intro.scriptDate`) to `POSIXct` in UTC. The export encodes timestamps as `"2026-01-26 17:30:16.853688Z"` (UTC, microseconds, trailing `Z`); this strips the trailing `Z` and parses the date-time. It is a pure decoder of that specific format, not a general timestamp parser.

Usage

```
parse_campaign_timestamps(x)
```

Arguments

x	A character vector of export timestamp strings. A POSIXct input is returned in UTC with its instant preserved (not re-parsed).

Details

Blank and NA inputs return NA; non-blank inputs that do not match the export format also return NA (no error, no warning), so a parse-failure mask is !is.na(raw) & nzchar(raw) & is.na(result).

Used internally by the latency pipeline (per-column parsing with diagnostics) and the config validators; exported so report authors can decode an export column directly.

Value

A POSIXct vector in UTC, the same length as x.

Examples

```
parse_campaign_timestamps(c("2026-01-26 17:30:16.853688Z", "", NA))
```

---

question_funnel *Count the recipients reaching each question of a campaign export*

Description

For each question in questions (in the given order), counts the records that reached it – those with a parseable field timestamp for the question (i.e. were shown it). Returns one row per question with the reached count and its share of the funnel head. This is the per-question companion to latency_funnel (which returns the send / opt-in / complete anchors): use it for the "records reaching each stage" view that latency_funnel does not provide.

Usage

```
question_funnel(data, questions, field = "scriptDate")
```

Arguments

data	A campaign export data frame.
questions	Ordered character vector of question stems forming the funnel, e.g. c("adrecall", "favorability", "transparent","problemsolution", "close"). The first entry is the funnel head.
field	Timestamp field marking "reached" (default "scriptDate", the send of that question).

Details

The first entry of `questions` is the funnel head, and `pct_reached` is each stage's percentage of that head (so the head is 100). For a well-formed flow reached is non-increasing down the questions; counts are computed independently per question, so they reflect the data as-is and are not forced to be monotone.

Value

A data frame with one row per question, in the order given: `question`, `question_index` (1-based flow order), `n_reached`, and `pct_reached` (share of the funnel head, 0-100; `NA` when the head count is zero).

See Also

`latency_funnel` for the send/opt-in/complete anchors.

Examples

```
data <- data.frame(
  id.adrecall.scriptDate = c("2026-01-26 21:01:00Z", "2026-01-26 21:05:00Z"),
  id.close.scriptDate    = c("2026-01-26 21:02:00Z", NA),
  check.names = FALSE
)
question_funnel(data, c("adrecall", "close"))
```

---

question_timestamps *Read one question's timestamps from a campaign export*

Description

Resolves the `id.<question>.<field>` column of a Survey160 campaign export and parses it to `POSIXct` (UTC) via `parse_campaign_timestamps`. A robust replacement for hand-built `data[[paste0("id.", q, ".", field)]]` access in report/analysis code.

Usage

```
question_timestamps(data, question, field = "scriptDate")
```

Arguments

`data`	A campaign export data frame (e.g. from `s160_gcs_campaign_results_read` or `s160_read_csv`).
`question`	Question stem (character scalar), e.g. `"intro"`, `"close"`, `"adrecall"`.
`field`	Timestamp field (character scalar): `"scriptDate"` (the outbound send, the default) or `"batchDate"` (the inbound reply).

Details

A column that is already `POSIXct` (e.g. a pre-parsed export) is returned as-is in UTC, without a lossy re-parse (parsing a non-UTC `POSIXct` through `as.character()` would shift the instant).

Value

A `POSIXct` vector in UTC, one element per row of `data`.

See Also

`question_funnel` for the per-question reached-count funnel built on this.

Examples

```
data <- data.frame(
  id.close.scriptDate = c("2026-01-26 21:02:00Z", NA),
  check.names = FALSE
)
question_timestamps(data, "close")
```

---

s160_api_auth *Authenticate to the Survey160 API*

Description

One entry point, addressed by environment *name* so the base URL, the campaign-results GCS bucket, and the API key are resolved together and cannot be mismatched. It authenticates and returns a *connection* – an opaque handle holding the JWT, credentials, base URL, environment name, and paired bucket. How you use the return value gives single- or multi-environment behaviour from the same call:

Usage

```
s160_api_auth(env = .ENV_CHOICES)
```

Arguments

`env`	Environment name: `"prod"` (default), `"staging"`, or `"dev"`. Each reads its API key from `S160_<ENV>_API_KEY` (prod also accepts the legacy `S160_API_KEY`). Pass it by name (`env = "staging"`); a positional value is deprecated.

Details

- **Single environment**: ignore the return value. The call also refreshes the package's default connection, so subsequent `s160_api_campaign_results` / `s160_api_campaign_get` calls with no conn use it. `s160_api_auth(); df <- s160_api_campaign_results(744)`.

- **Both environments at once**: capture each connection and pass it as `conn =`. `prod <- s160_api_auth(env = "prod"); stg <- s160_api_auth(env = "staging")`, then `s160_api_campaign_results(744, conn = stg)`. Each connection is independent, so prod and staging can be held live in the same session – e.g. to compare a campaign across both.

Credentials come from `~/.Renviron` (never typed into code): the user ID from `S160_API_USERID`, and the API key from a per-environment variable – `S160_STAGING_API_KEY` for staging, and `S160_PROD_API_KEY` (or, as a fallback, the legacy `S160_API_KEY`) for prod. Missing values prompt once on an interactive run and are saved.

Value

A connection object (an environment) to pass as `conn`, returned invisibly. As a side effect, the package's default connection is updated to this one so conn-less calls use the most recent authentication.

Before you start

You need an API user ID and a per-environment API key from your team lead (most Survey Managers never need the API: the last export of every campaign is already on Cloud Storage, see `s160_gcs_campaign_results_read`; the API is for an export fresher than that). The first `s160_api_auth(env)` prompts for whatever `~/.Renviron` lacks and saves it, so you type each value once. To rotate a key, edit the `S160_<ENV>_API_KEY` line in `~/.Renviron` and restart R. A refused sign-in raises `s160_auth_error` naming the variables and URL it used.

The in-session JWT refresh (tokens expire after 10 minutes; re-auth at 8) reuses the credentials stored on the connection, so a staging connection held alongside prod keeps refreshing against staging rather than the default.

Examples

```
## Not run:
# Single environment -- ignore the return, use the default connection:
s160_api_auth()                     # defaults to prod
df <- s160_api_campaign_results(744)

# Both environments at once -- capture each, pass conn =:
s160_gcs_init()  # one GCS auth covers all buckets
prod <- s160_api_auth(env = "prod")
stg  <- s160_api_auth(env = "staging")
df_prod <- s160_api_campaign_results(744, conn = prod)
df_stg  <- s160_api_campaign_results(744, conn = stg)

## End(Not run)
```

`s160_api_campaign_get` *Read one campaign's attributes from the Survey160 API*

Description

Wraps the Survey160 API endpoint `GET /campaigns/<campaign_id>`, which returns every column on the `campaigns` table for one campaign. Useful for confirming attributes after a state-changing call (for example, reading `archive_scheduled_date` after scheduling an archive) without dropping to direct database access.

Usage

```
s160_api_campaign_get(campaign_id, conn = NULL)
```

Arguments

`campaign_id`	Campaign ID (numeric or character).
`conn`	Connection to use. Defaults to the package's default connection (the most recent `s160_api_auth`). Pass a connection returned by `s160_api_auth` to target a specific environment.

Details

Enriched, API-only fields returned by the endpoint (`listlength`, `list`, `login`, `exports`, `has_texting_started`, `sandbox_configuration`, `aggregator`, `has_assigned_registration`) are dropped; the result mirrors the `campaigns` table only. JSON-valued columns (`script`, `prompt`, `quotas`, ...) come back as length-1 list-columns holding the parsed structure.

The endpoint runs several server-side subqueries on each call; this is a per-campaign read, not appropriate for tight loops over hundreds of IDs. A batch variant would need a backend extension and is out of scope.

Value

A single-row data frame. Scalar columns are scalar; ISO-8601 timestamp columns are coerced to `POSIXct` in UTC; JSON columns are list-columns of length 1.

Examples

```
## Not run:
s160_api_auth()
info <- s160_api_campaign_get(2107)
info$active
info$script[[1]]  # parsed JSON

## End(Not run)
```

`s160_api_campaign_results` *Trigger a fresh campaign export and read it*

Description

Triggers a fresh campaign results export, polls GCS until the file is updated, and returns the results as a data frame. Requires both API auth (`s160_api_auth`) and GCS auth (`s160_gcs_init`).

Usage

```
s160_api_campaign_results(
  campaign_id,
  filter_open = FALSE,
  timeout = 300,
  poll_interval = 5,
  destdir = NULL,
  conn = NULL,
  ...
)
```

Arguments

`campaign_id`	Campaign ID (numeric or character).
`filter_open`	Logical. Exclude open/uncontacted conversations? Default `FALSE`.
`timeout`	Timeout in seconds for export completion. Default 300.
`poll_interval`	Maximum polling interval in seconds. Default 5. Polling uses exponential back-off starting at the smaller of 2s and this value, capped at this value.
`destdir`	Directory to save the downloaded CSV. `NULL` (default) uses a temporary file.
`conn`	Connection to use. Defaults to the package's default connection (the most recent `s160_api_auth`). Pass a connection returned by `s160_api_auth` to target a specific environment; the export trigger, the completion poll, and the CSV read all use that connection's environment and paired GCS bucket.
`...`	Additional arguments passed to `read.csv()`.

Value

A data frame with one row per survey response.

Examples

```
## Not run:
s160_gcs_init()
s160_api_auth()
df <- s160_api_campaign_results(1980)
df <- s160_api_campaign_results(1980, filter_open = TRUE, timeout = 600)
```

```
# Target a non-prod environment: authenticate against it, pass the
# connection. This is the usual way an analyst switches environment -- the
# connection carries the env, and results are read from its bucket.
stg <- s160_api_auth(env = "staging")
df  <- s160_api_campaign_results(1980, conn = stg)

# Compare the same campaign across two environments concurrently:
prod <- s160_api_auth(env = "prod")
stg  <- s160_api_auth(env = "staging")
df_prod <- s160_api_campaign_results(744, conn = prod)
df_stg  <- s160_api_campaign_results(744, conn = stg)

## End(Not run)
```

`s160_config` *Read the Survey160 environment configuration*

Description

Returns the configuration survey160r uses to reach Survey160's backends: the environments, each one's API base URL, and the datasets (with their GCS locations) it carries. This is the single source of truth behind `s160_api_auth()` and the GCS readers. You rarely need it directly, but it is handy for inspecting or scripting against what is available.

Usage

```
s160_config(refresh = FALSE)
```

Arguments

`refresh`	If `TRUE`, drop the cached config and reload it.

Details

The config is bundled with the package; `refresh = TRUE` reloads it (and, in a later release, re-fetches an updated copy from Survey160).

Value

A nested list with `schema_version` and `environments`; each environment carries an optional `api_url` and a `datasets` map of `dataset -> list(bucket, object)`.

See Also

`s160_datasets()` for a tidy `(dataset, env)` table.

Examples

```
config <- s160_config()
names(config$environments)
```

`s160_csv_header` *Read the column names of a campaign export*

Description

Peeks the first line of a CSV and returns its column names in the same `make.names()`-munged (dot-form) form the readers produce, without parsing the body. Pair with `latency_build_config()` + `latency_input_columns()` to derive a column-projection set for a large file before reading it (see the projection example under `latency_input_columns`).

Usage

```
s160_csv_header(path, encoding = "UTF-8")
```

Arguments

`path`	Path to the CSV.
`encoding`	File encoding for the header peek (`"UTF-8"` default), kept consistent with the body read so a UTF-8/BOM file munges to the same names regardless of reader.

Value

Character vector of dot-form column names.

Examples

```
path <- tempfile(fileext = ".csv")
writeLines(c("campaignid,id.intro.scriptDate", "1,2026-01-26 21:00:00Z"), path)
s160_csv_header(path)
unlink(path)
```

`s160_datasets` *List the Survey160 datasets and their environments*

Description

Lists the datasets survey160r can read and the environments each is available in – the values you pass as `env =` to the campaign readers, the `disposition_pull()` / `opt_out_pull()` / `campaign_metrics_pull()` pulls, and `s160_api_auth()`. Use it to discover the valid `(dataset, env)` combinations instead of guessing and hitting a "no <env> tier" error.

Usage

```
s160_datasets()
```

Details

The physical storage location is intentionally not exposed: select data by `env` and survey160r resolves the bucket internally, so a bucket reorg never changes what you call.

Value

A data frame with one row per available `(dataset, env)` tier:

`dataset` Logical dataset name: `"campaign_results"` (the campaign readers and `s160_api_*`), `"disposition"` (`disposition_pull()`), `"opt_out"` (`opt_out_pull()`), or `"campaign_metrics"` (`campaign_metrics_pull()`).

`env` An environment the dataset is available in: `"prod"`, `"staging"`, or `"dev"`.

See Also

`s160_config()` for the full nested configuration.

Examples

```
s160_datasets()
```

---

s160_gcs_campaign_results_files

List the files in a campaign's Cloud Storage folder

Description

Returns the file names inside a campaign's folder in the results bucket. Returns `character(0)` with a message if the campaign has no files.

Usage

```
s160_gcs_campaign_results_files(campaign_id, env = .ENV_CHOICES, bucket = NULL)
```

Arguments

`campaign_id`	Campaign ID (numeric or character). Must be a single value.
`env`	Environment: `"prod"` (default), `"staging"`, or `"dev"`.
`bucket`	**Deprecated.** Select data with `env =` instead; a supplied bucket is honored with a warning for back-compat.

Value

Character vector of file names (without the campaign_id prefix).

Examples

```
## Not run:
s160_gcs_init()
s160_gcs_campaign_results_files(1980)

## End(Not run)
```

`s160_gcs_campaign_results_list` *List the campaign ids in the campaign-results bucket*

Description

Returns a sorted character vector of campaign IDs (top-level folder names) in the results bucket. Objects at the bucket root (not inside a folder) are excluded.

Usage

```
s160_gcs_campaign_results_list(env = .ENV_CHOICES, bucket = NULL)
```

Arguments

`env`	Environment: `"prod"` (default), `"staging"`, or `"dev"`.
`bucket`	**Deprecated.** Select data with `env =` instead; a supplied bucket is honored with a warning for back-compat.

Value

Character vector of campaign IDs, sorted.

Examples

```
## Not run:
s160_gcs_init()
s160_gcs_campaign_results_list()

## End(Not run)
```

s160_gcs_campaign_results_read *Read a campaign export from Cloud Storage*

Description

Downloads the CSV from GCS and reads it into R. By default, the file is downloaded to a temporary location and cleaned up automatically. Set `destdir` to keep a local copy.

Usage

```
s160_gcs_campaign_results_read(
  campaign_id,
  filename = NULL,
  destdir = NULL,
  env = .ENV_CHOICES,
  bucket = NULL,
  columns = NULL,
  columns_fn = NULL,
  hash = FALSE,
  timestamps = c("character", "POSIXct"),
  ...
)
```

Arguments

`campaign_id`	Campaign ID (numeric or character). Must be a single value.
`filename`	File name in the campaign folder. Defaults to `<campaign_id>_raw_data_download.csv` (the standard export filename). Must not contain path separators.
`destdir`	Directory to save the downloaded file. When `NULL` (default), a temporary file is used and cleaned up automatically. Use `"."` for the current directory.
`env`	Environment: `"prod"` (default), `"staging"`, or `"dev"`.
`bucket`	**Deprecated.** Select data with `env =` instead; a supplied bucket is honored with a warning for back-compat.
`columns`	Optional character vector of (dot-form) column names to keep, e.g. from `latency_input_columns()`. When set, only those columns are parsed (via `data.table::fread`'s column projection), cutting read time and memory on wide exports. `NULL` (default) reads every column.
`columns_fn`	Optional resolver `function(header) -> columns` for the common case where the projection set depends on the file's own header (e.g. `function(h) latency_input_columns(latenc` `h),available = h)`). Used only when `columns` is `NULL`: the header is peeked from the downloaded file and passed to `columns_fn`, whose result becomes the projection. A resolver (or header-peek) error emits a warning and falls back to a full read. A non-function value is rejected. This is the GCS counterpart to the downstream fleet reader's local projection, so both paths keep only the columns the transform needs. `NULL` (default) applies no header-derived projection.

hash	When TRUE, stamp provenance on the returned frame – the sha256 of the downloaded CSV bytes as source_csv_hash and the canonical gs:// source as source_csv_path, which latency_run() / latency_report() then surface on result$meta. FALSE (default) skips the extra hashing read and returns a plain frame. (The local-file sibling is s160_read_csv, whose hash stamps the same provenance.)
timestamps	How the export's id.<q>.scriptDate / id.<q>.batchDate columns are returned: "character" (default) keeps the export's strings; "POSIXct" decodes them to UTC POSIXct at read time via parse_campaign_timestamps. A parsed column takes 8 bytes per cell against ~60 for the strings, so "POSIXct" roughly halves a wide export's footprint in memory; every transform (latency_run(), disposition_run(), question_funnel(), ...) accepts either form and produces the same result (an unparseable non-blank string is NA in the parsed column and is still counted in latency_report()'s parse_failures_per_column).
...	Additional arguments forwarded to the CSV reader (data.table::fread, or utils::read.csv when data.table is unavailable), e.g. na.strings, nrows, sep.

Details

GCS path: gs://<bucket>/<campaign_id>/<filename>

Value

A data frame with one row per survey response. With hash = TRUE it also carries source_csv_hash and source_csv_path attributes.

Examples

```
## Not run:
s160_gcs_init()
df <- s160_gcs_campaign_results_read(1980)
df <- s160_gcs_campaign_results_read(1980, destdir = ".")
df <- s160_gcs_campaign_results_read(1980, destdir = "~/data")
df <- s160_gcs_campaign_results_read(1980, hash = TRUE)   # + provenance attrs

## End(Not run)
```

---

s160_gcs_campaign_results_status

Check a campaign export's update time and size

Description

Returns GCS file metadata for the campaign's export file without triggering a new export. Requires GCS auth (s160_gcs_init).

Usage

```
s160_gcs_campaign_results_status(
  campaign_id,
  env = .ENV_CHOICES,
  bucket = NULL
)
```

Arguments

campaign_id	Campaign ID (numeric or character).
env	Environment: "prod" (default), "staging", or "dev".
bucket	**Deprecated.** Select data with env = instead; a supplied bucket is honored with a warning for back-compat.

Value

Named list with name, updated, and size, or NULL if no export file exists.

Examples

```
## Not run:
s160_gcs_init()
s160_gcs_campaign_results_status(1980)

## End(Not run)
```

---

s160_gcs_init *Authenticate to Google Cloud Storage*

Description

Authenticates the session to GCS. By default this uses the Survey160 Desktop OAuth client (browser sign-in); pass adc = TRUE for non-interactive Application Default Credentials instead (see below).

Usage

```
s160_gcs_init(bucket = NULL, adc = FALSE)
```

Arguments

bucket	**Deprecated.** Formerly set a session-global default bucket. Superseded by per-reader env =: select an environment with env = on the reader (readers default to prod). If supplied here it is still accepted for back-compatibility (with a warning) but should not be used in new code.
adc	Logical (default FALSE). When TRUE, authenticate non-interactively with Application Default Credentials rather than the Survey160 browser OAuth client.

Details

On first run, prompts for the client secret (get it from your team lead) and saves it to `~/.Renviron`. Subsequent runs read it automatically. Also opens a browser for Google sign-in on first use; the OAuth token is cached in a platform-dependent directory (run `gargle::gargle_oauth_sitrep()` to locate it).

Authentication is account-level: one call covers every bucket the account can read. Reader functions resolve their own bucket (defaulting to prod), so you no longer pass a bucket here.

Value

Invisible `NULL`.

Before you start

Two things come from outside R, both one-time:

- **The OAuth client secret** – ask your team lead. You paste it at the first prompt and it is saved to `~/.Renviron` as `S160_GCS_CLIENT_SECRET`.
- **A bucket grant** – Storage Object Viewer on the bucket you read (`campaign_results` for exports, `s160_disposition_prod` for the disposition dataset and opt-out list, `s160_analytics_prod` for campaign metrics), via the `gcp-campaign-readers` group. Without it the sign-in succeeds but the first read fails with a 403; ask a sysadmin to add you.

If the browser does not open (a server, a container, or an R session with no default browser), set one first with `options(browser = "xdg-open")` (or the path to a browser), or sign in headlessly with `adc = TRUE` after `gcloud auth application-default login`. To start over, delete the `S160_GCS_CLIENT_SECRET` line from `~/.Renviron`, remove the token cache `gargle::gargle_oauth_sitrep()` lists, and restart R.

For headless runs (reports, CI, scheduled jobs) pass `adc = TRUE` to authenticate with Application Default Credentials instead of the browser flow: no client secret and no browser, using the credentials from `gcloud auth application-default login` or a service-account environment. It requests the `cloud-platform` scope and errors clearly if no ADC are available.

Examples

```
## Not run:
s160_gcs_init()             # interactive browser OAuth
s160_gcs_init(adc = TRUE)   # headless, via Application Default Credentials

## End(Not run)
```

s160_read_csv *Read a campaign export from a local file*

Description

Local-source sibling of `s160_gcs_campaign_results_read()` (with `hash = TRUE`). Reads the CSV via `data.table::fread` (falling back to `utils::read.csv`) and stamps `source_csv_hash` and `source_csv_path` attributes on the returned data frame so downstream `latency_report()` / `latency_run()` surface them on `result$meta`. Use for backfills (archived campaign CSVs stored on disk, Dropbox, S3 mounts, etc.).

Usage

```
s160_read_csv(
  path,
  columns = NULL,
  hash = TRUE,
  timestamps = c("character", "POSIXct"),
  ...
)
```

Arguments

`path`	Path to the CSV. Recorded verbatim on `attr(., "source_csv_path")`.
`columns`	Optional character vector of (dot-form) column names to keep (e.g. from `latency_input_columns()`). When set, only those columns are parsed, cutting read time and memory on wide exports. `NULL` (default) reads every column.
`hash`	When `TRUE` (default), compute the sha256 of the file for `source_csv_hash`. Set `FALSE` to skip the hashing pass (a full second read of the file) on large backfills where provenance hashing is not needed; `source_csv_hash` is then `NA`.
`timestamps`	`"character"` (default) keeps the export's `id.<q>.scriptDate` / `id.<q>.batchDate` strings; `"POSIXct"` decodes them to UTC `POSIXct` at read time, which roughly halves a wide export's footprint in memory. See the same argument of s160_gcs_campaign_results_rea
`...`	Forwarded to the CSV reader (`data.table::fread`, or `utils::read.csv` when data.table is unavailable), e.g. `na.strings`, `sep`. `stringsAsFactors` defaults to `FALSE`.

Value

A data frame with `source_csv_hash` and `source_csv_path` attributes set.

Examples

```
## Not run:
data <- s160_read_csv("~/Dropbox/archive/campaign_500.csv")
attr(data, "source_csv_hash")
```

```
latency_run(500, data, field_timezone = "America/New_York")

## End(Not run)
```

`utils_fix_double_utf8` *Detect or repair double-encoded UTF-8 in export text*

Description

Reverses a double UTF-8 encoding in which a value's original UTF-8 bytes were mis-read as Latin-1 (ISO-8859-1) and re-encoded to UTF-8 – so an en dash arrives as three stray Latin-1-supplement characters, a trademark sign as three, an accented letter as two, and so on. This is the corruption a raw Survey160 campaign export can carry in its appended sample columns (treatment labels, vendor-appended employer/title text).

Usage

```
utils_fix_double_utf8(x, apply = FALSE, quiet = FALSE)
```

Arguments

`x`	A character vector, or a data frame (its character columns are inspected; other columns are returned unchanged).
`apply`	If `FALSE` (the default), do a dry run: report what would be repaired and return `x` unchanged. Set `TRUE` to return the repaired data.
`quiet`	If `FALSE` (the default), emit a one-line `message()` summary. Set `TRUE` to silence it (e.g. when calling in a loop).

Details

The function verifies by default and repairs on request. With `apply = FALSE` (the default) it is a **dry run**: it reports how many values it would repair (and, for a data frame, which columns) and returns `x` unchanged. With `apply = TRUE` it returns the repaired data. Run it once to see what it would do, then again with `apply = TRUE`; or, since the repair is a no-op on clean input, call it with `apply = TRUE` directly.

The reversal is applied only to Latin-1-supplement code points (U+0080..U+00FF) – the only thing a Latin-1 mis-decode can produce – and only where the bytes re-read as valid UTF-8, so text that was never doubled (a genuine en dash, an emoji, an accented name that is correctly single-encoded) is left byte-for-byte unchanged, even where such a character sits directly beside a doubled one. A clean input is therefore returned unchanged, and the operation is idempotent. Only character columns of a data frame are inspected; factor and other columns are left unchanged (read the export with `stringsAsFactors = FALSE`, the R 4.x default, so sample columns are character). Unless `quiet = TRUE`, a one-line summary is emitted with `message()`, so neither a dry run nor a repair is silent.

Mojibake repair is heuristic. The guard above makes a false repair unlikely, but a value that legitimately contains a Latin-1 run whose bytes happen to be valid UTF-8 would be "repaired" to the character(s) it decodes to. The durable fix is to stop the double encoding at its source (the upload that produced the export); this function repairs exports produced before that fix.

Value

With `apply = TRUE`, `x` with double-encoded runs reversed (a character vector of the same length, or the same data frame with its character columns repaired). With `apply = FALSE`, `x` unchanged. `NA` and non-character columns always pass through untouched. The value is returned invisibly, so a bare console call (e.g. a dry-run check) prints only the one-line summary, not the data; assign or pipe it to use it.

See Also

`s160_read_csv` and `s160_gcs_campaign_results_read`, which read the raw export this repairs.

Examples

```
# The escaped bytes below are the mojibake form of a single en dash.
# Dry run (default): reports what it would repair, returns the input unchanged.
utils_fix_double_utf8("Treatment Group \u00e2\u0080\u0093 Control")

# Apply the repair:
utils_fix_double_utf8("Treatment Group \u00e2\u0080\u0093 Control", apply = TRUE)

# A whole export frame -- character columns are repaired, others untouched:
df <- data.frame(
  EMPLOYER = "Acme Corp\u00e2\u0084\u00a2",
  complete = 1L,
  stringsAsFactors = FALSE
)
utils_fix_double_utf8(df, apply = TRUE, quiet = TRUE)
```

Index

campaign_metrics_pull, *4*, 6, *7, 8*
campaign_metrics_records, *4*, *6*, 7, *7–9*, *29*
campaign_metrics_summary, *4*, *6, 7*, 8, *8*, *10, 11*, *29*

disposition_funnel, *4*, 9, *16*, *29*
disposition_input_columns, *4*, 11
disposition_paths, *4, 5*, 13, *16*, *23–25*
disposition_pull, *4*, *6, 7*, *10*, *13*, 15, *17*, *23*, *27, 28*, *38, 39*
disposition_records, *4*, *10, 11*, 16, *16*, *22*, *25–28*
disposition_run, *4*, *17*, 18
disposition_screen, *4*, *14–17*, 20, *26–28*, *39, 40*
disposition_steps, *4, 5*, *13, 14*, *16*, *20, 21*, 23
disposition_summary, *4*, *11*, *14–17*, *20–22*, *24, 25*, 26

funnel_rates, *4*, *8–11*, 28

latency_build_config, *3*, 29
latency_config_hash, *3*, 30
latency_discover_questions, *3*, 31
latency_funnel, *3*, *10*, 32, *41, 42*
latency_input_columns, *3*, *12*, 33, *48*
latency_report, *3*, 34, *36*
latency_run, *3*, 35
latency_validate_config, *3*, *5*, 37

opt_out_pull, *4*, *7*, *16*, 38, *40*
opt_out_screen, *4*, *22*, *38*, 39, *39*

parse_campaign_timestamps, *3*, 40, *42*, *52*

question_funnel, *3*, 41, *43*
question_timestamps, *3*, 42

s160_api_auth, *3*, *5*, 43, *45, 46*
s160_api_campaign_get, *3*, *43*, 44
s160_api_campaign_results, *3*, *43*, 46
s160_config, *3*, 47, *49*
s160_csv_header, *3*, 48
s160_datasets, *3*, *47*, 48
s160_gcs_campaign_results_files, *3*, 49
s160_gcs_campaign_results_list, *3*, 50
s160_gcs_campaign_results_read, *3*, *42*, *44*, 51, *55*, *57*
s160_gcs_campaign_results_status, *3*, 52
s160_gcs_init, *3*, *5–7*, *15, 16*, *38, 39*, 53
s160_read_csv, *3*, *42*, *52*, 55, *57*
survey160r *(*survey160r-package*)*, 3
survey160r-package, 3

utils_fix_double_utf8, *4*, 56