Package 'LLMRpanel'

September 12, 2026

Type Package

Title Benchmarked Silicon Samples for Survey and Experiment Design

Version 0.6.1

Description Administers survey and experimental instruments to panels of language-model personas, with respondent-level randomization, benchmark comparison against human data, and conjoint estimation from recorded respondent-level profile assignments. Samples of language-model personas follow Argyle et al. (2023) <doi:10.1017/pan.2023.2>; the case for benchmarking them against human data is set out in Bisbee et al. (2024) <doi:10.1017/pan.2024.5>; the conjoint estimand is the average marginal component effect of Hainmueller et al. (2014) <doi:10.1093/pan/mpt024>.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.2)

Imports LLMR (>= 0.8.9), tibble, rlang, cli, stats, utils

Suggests testthat (>= 3.0.0), ggplot2, knitr, rmarkdown, shiny, bslib, DT, LLMR.shiny (>= 0.1.2)

Config/testthat/edition 3

VignetteBuilder knitr

URL https://github.com/asanaei/LLMRpanel, https://asanaei.github.io/LLMRpanel/

BugReports https://github.com/asanaei/LLMRpanel/issues

NeedsCompilation no

Author Ali Sanaei [aut, cre, cph]

Maintainer Ali Sanaei <sanaei@uchicago.edu>

Repository CRAN

Date/Publication 2026-09-12 13:20:08 UTC

Contents

`as_persona_frame` *Attach the persona contract to a data frame*

Description

Attaches persona metadata to a decoded data frame. When a `persona_frame` is passed to `panel_from_data()` without a template, demographic fields and stated answers are rendered separately. A question map supplies the wording used for answer fields. Plain data frames use the flat key-value rendering.

Usage

```
as_persona_frame(data, questions = NULL, demographics = NULL, answers = NULL)
```

Arguments

`data`	A decoded data frame, one respondent per row. Values should already be human-readable labels (decode a labelled survey file with, for example, `haven::as_factor()` first).
`questions`	Optional named character vector mapping column names to the human question wording, e.g. `c(pid = "Party identification", ab = "Abortion position")`. Columns absent from this map keep their column name. Without it the column names stand in for the questions, which is formatting, not a faithful translation.

demographics	Optional character vector of columns to treat as demographic background (the rest become stated answers). Defaults to the common demographic names found in data.
answers	Optional character vector restricting which columns may appear as stated answers. Defaults to every column that is not a demographic, an id-named, or a weight-named column, so analysis-only columns do not leak into the prompt.

Value

data with the persona contract attached and class persona_frame.

See Also

panel_from_data(), panel_from_personas(), LLMR::llm_persona_split().

Examples

```
df <- data.frame(
  age = c("35-44", "65+"),
  pid = c("Strong Democrat", "Strong Republican"),
  ab  = c("Always legal", "Never legal"))
pf <- as_persona_frame(
  df,
  questions = c(pid = "Party identification", ab = "Abortion position"),
  demographics = "age")
```

---

conjoint_amce *AMCEs from a conjoint administration*

Description

Average marginal component effects from a conjoint_instrument() administration: one OLS regression of profile choice on treatment-coded dummies for all attributes simultaneously, with CR1 cluster-robust standard errors clustered by persona and 95% intervals on the t distribution with G - 1 degrees of freedom (G personas). Under uniform, independent profile randomization this is the standard AMCE estimator. The regression uses the respondent-level profiles recorded during administration, not the profiles in the initial design table.

Usage

```
conjoint_amce(responses)
```

Arguments

responses	A panel_administer() result whose instrument came from conjoint_instrument().

Value

A `conjoint_amce` tibble: `attribute`, `level`, `estimate`, `std_error`, `ci_lo`, `ci_hi`. Baseline levels (the first level present, in the design's order) appear with estimate 0 and `std_error = NA`, so the table feeds the familiar conjoint plot directly. The ordinary columns `n_profiles`, `n_respondents`, `n_dropped_na`, and `n_execution_failures` record the profile rows used, the respondents administered, missing task responses dropped, and failed executions.

References

Hainmueller, Jens, Daniel J. Hopkins, and Teppei Yamamoto (2014). "Causal Inference in Conjoint Analysis: Understanding Multidimensional Choices via Stated Preference Experiments." *Political Analysis* 22(1), 1-30.

Examples

```
set.seed(110)
panel <- panel_from_margins(list(group = c(A = .5, B = .5)), n = 6)
design <- conjoint_design(
  list(color = c("blue", "red"), cost = c("low", "high")),
  n_tasks = 6)
instrument <- conjoint_instrument(design)
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b")

# A synthetic respondent who always takes the red profile: the AMCE for red
# is recovered, and cost is flat.
prefers_red <- function(experiments, ...) {
  shown <- vapply(experiments$messages, `[[`, "", "user")
  first <- sub("Profile 2.*", "", shown)
  experiments$response_text <- ifelse(grepl("red", first),
                                      "Profile 1", "Profile 2")
  experiments
}
r <- panel_administer(panel, instrument, cfg, .runner = prefers_red)
conjoint_amce(r)


if (nzchar(Sys.getenv("GROQ_API_KEY"))) {
  conjoint_amce(panel_administer(panel, instrument, cfg))
}
```

`conjoint_design` *Conjoint tasks*

Description

Random profile pairs (or k-tuples) over the supplied attributes, the design for a forced-choice conjoint. Profiles are sampled uniformly and independently per attribute; set a seed beforehand for a reproducible design (the function never sets one). At administration, fresh profiles are drawn independently for every respondent from the same attribute levels.

Usage

```
conjoint_design(attributes, n_tasks = 5L, profiles_per_task = 2L)
```

Arguments

attributes	Named list of level vectors.
n_tasks	Tasks per respondent.
profiles_per_task	Profiles shown per task (default 2).

Value

A conjoint_design list with fields profiles, a tibble containing task, profile, and one column per attribute, and attributes, the named list of attribute levels. Render it into forced-choice items with conjoint_instrument() and estimate with conjoint_amce() after administration. Profiles within a task are distinct when the attribute space permits them. When it does not, duplicates remain and a warning is issued.

Examples

```
set.seed(110)
conjoint_design(
  list(price = c("$10", "$20"), speed = c("slow", "fast")),
  n_tasks = 4)
```

---

conjoint_instrument *Build a conjoint instrument*

Description

Converts a conjoint_design() into one forced-choice item per task: each respondent receives a fresh independent profile draw at administration and is asked to pick one by label.

Usage

```
conjoint_instrument(design, question = "Which profile do you prefer?")
```

Arguments

design	A conjoint_design() object.
question	Question text shown above each task's profiles.

Details

Only option order is randomized; item order stays fixed so the task ids remain interpretable. Attribute order inside each profile description follows the design's column order. The profiles recorded in the design are not reused across respondents.

Value

A panel_instrument whose items are task-level choice items (ids task_1, task_2, ...; options "Profile 1", "Profile 2", ...). Each item carries the attribute levels used for its respondent-level draws, and the instrument's $conjoint field carries the design metadata for conjoint_amce().

Examples

```
set.seed(110)
panel <- panel_from_margins(list(group = c(A = .5, B = .5)), n = 4)
design <- conjoint_design(
  list(economy = c("weak", "strong"), taxes = c("lower", "higher")),
  n_tasks = 3)
instrument <- conjoint_instrument(design, "Which candidate do you prefer?")
instrument
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b")

first_shown <- function(experiments, ...) {
  experiments$response_text <- "Profile 1"
  experiments
}
panel_administer(panel, instrument, cfg, .runner = first_shown)


if (nzchar(Sys.getenv("GROQ_API_KEY"))) {
  panel_administer(panel, instrument, cfg)
}
```

---

panel_administer *Administer an instrument to a panel*

Description

Creates one request for each combination of persona and item. The persona goes in the system message, the item and its options in the user message. A closed-item reply is matched against the options offered, and anything unmatched becomes NA. Open items come back verbatim.

Usage

```
panel_administer(
  panel,
  instrument,
  config,
  max_calls = 5000L,
  confirm = FALSE,
  price_table = NULL,
  tokens_per_call = NULL,
  .runner = NULL,
  ...
)
```

Arguments

panel	A panel_from_margins(), panel_from_data(), or panel_from_personas() result.
instrument	A panel_instrument().
config	An LLMR::llm_config() for a generative model.
max_calls	Integer. If the run would make more than this many calls (personas times items), it stops unless confirm = TRUE. Default 5000.
confirm	Logical. Set TRUE to proceed past max_calls.
price_table, tokens_per_call	Optional. When both are supplied, the preflight reports a cost figure computed from your own price_table (the LLMR::llm_usage() format: columns model, input, output, prices per million tokens) and your tokens_per_call assumption. One number is a per-call total, priced as a range from all-input to all-output; two, c(input, output), price exactly. The package itself ships no prices and estimates no token counts.
.runner	Optional runner for offline or deterministic testing: a function(experiments, ...) that receives a data frame with config and messages list-columns and returns those rows with request_id and response_text columns. Each submitted request_id must appear once; returned rows may be in any order. Defaults to a live LLM call via LLMR::call_llm_par().
...	Passed to the runner (e.g. tries, progress).

Value

A panel_responses object with fields data, panel, instrument, benchmark, and usage. data is a tibble with persona_id, item_id, type, item_position (the item's fixed 1-based position in the instrument; each request is independent, so no questionnaire order is ever shown to the model), option_order (what this respondent saw, |-separated), response (matched option or NA; verbatim text for open items), and score (1-based scale position for Likert items). score uses the item's canonical scale rather than its displayed order. response_text, response_id, success, error_message, finish_reason, model, and provider retain execution provenance as ordinary columns. Conjoint administrations also include a profiles list-column. benchmark is NULL until panel_benchmark() is called. usage retains execution diagnostics and any token counts or per-call duration; it is NULL when the runner returned none of those usage fields.

Examples

```
set.seed(110)   # the panel draw is local; the model call is not
panel <- panel_from_margins(list(party = c(left = .5, right = .5)), n = 6)
instrument <- panel_instrument(
  item_likert("wk4", "A four-day work week would benefit society."),
  randomize = character(0))
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b")

# The `.runner` seam answers without a provider, for tests or for a
# deterministic or external respondent:
deterministic <- function(experiments, ...) {
```

```
  experiments$response_text <- "agree"
  experiments
}
panel_administer(panel, instrument, cfg, .runner = deterministic)


if (nzchar(Sys.getenv("GROQ_API_KEY"))) {
  resp <- panel_administer(panel, instrument, cfg)
  resp
}
```

`panel_batch_fetch` *Fetch and parse a completed panel batch job*

Description

Retrieves the batch results and parses them into a `panel_responses`, identical in shape to a synchronous `panel_administer()` run. Responses are joined to the grid by request id, so the order the provider returns them in does not matter.

Usage

```
panel_batch_fetch(job)
```

Arguments

`job`	A `panel_batch_submit()` handle (or a `state_path` to one).

Value

A `panel_responses` object.

See Also

`panel_batch_submit()`, `panel_batch_status()`.

Examples

```
# Fetching reads the job saved at submission, once the provider reports it
# complete; the result has the same shape as a synchronous run.
state <- file.path(tempdir(), "panel_job.rds")

if (file.exists(state)) {
  responses <- panel_batch_fetch(state)
}
```

panel_batch_status *Check the status of a panel batch job*

Description

Check the status of a panel batch job

Usage

```
panel_batch_status(job)
```

Arguments

job	A panel_batch_submit() handle (or a state_path to one).

Value

The LLMR batch status (a one-row tibble).

See Also

panel_batch_submit(), panel_batch_fetch().

Examples

```
# A job saved by panel_batch_submit(state_path = ) is read back by path,
# so its progress can be checked from a later session.
state <- file.path(tempdir(), "panel_job.rds")

if (file.exists(state)) {
  panel_batch_status(state)
}
```

panel_batch_submit *Administer a panel asynchronously through the batch API*

Description

Submits one request per persona and item to a provider's batch API and returns a job handle. Use panel_batch_status() to inspect the job and panel_batch_fetch() to retrieve completed results. Provider services determine prices and completion times.

Usage

```
panel_batch_submit(
  panel,
  instrument,
  config,
  state_path = NULL,
  max_calls = 5000L,
  confirm = FALSE
)
```

Arguments

panel	A panel_from_margins() / panel_from_data() / panel_from_personas() result.
instrument	A panel_instrument().
config	An LLMR::llm_config() for a generative model on a provider with a supported batch API (OpenAI, Groq, Anthropic, Gemini).
state_path	Optional path; when given the job is also saved there as RDS so it can be fetched from another session.
max_calls	Integer. If the run would make more than this many calls, it stops unless confirm = TRUE. Default 5000.
confirm	Logical. Set TRUE to proceed past max_calls.

Details

All personas are administered under one config (one model). The handle carries the survey prompts and rendered persona text. When state_path is supplied, the API key must be referenced through an environment variable so its value is not written to the saved state.

Value

A panel_batch_job handle.

See Also

panel_batch_fetch(), panel_batch_status(), panel_administer().

Examples

```
panel <- panel_from_margins(list(party = c(left = .5, right = .5)), n = 200)
instrument <- panel_instrument(item_likert("wk4", "A four-day work week helps."))
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b",
                        api_key = LLMR::llm_api_key_env("GROQ_API_KEY"))
state <- file.path(tempdir(), "panel_job.rds")


if (nzchar(Sys.getenv("GROQ_API_KEY"))) {
  job <- panel_batch_submit(panel, instrument, cfg, state_path = state)
```

```
  panel_batch_status(job)
}
```

---

panel_benchmark *Compare silicon responses with a human benchmark*

Description

Compares closed-item response shares with human benchmark shares supplied by the user. The result contains deviations for covered item-response pairs, the number of closed items covered, and nonresponse rates by item. The function does not alter responses or adjust response shares. Without a benchmark, response shares describe the configured model under the supplied personas, not a human population.

Usage

```
panel_benchmark(responses, benchmark, benchmark_name = "benchmark")
```

Arguments

responses	A panel_administer() result.
benchmark	A data frame with columns item_id, response, and share (human marginal proportions). Shares within an item should sum to 1; a deviation beyond rounding draws a warning.
benchmark_name	How the source should be cited in reports (e.g. "ANES 2024 pilot").

Value

responses with its benchmark field set: $table (per covered item and response: share_silicon, share_human, deviation), $nonresponse (nonresponse and execution failure rates per item), $items_covered / $items_total, $mean_abs_dev, $max_dev.

Examples

```
set.seed(110)
panel <- panel_from_margins(list(party = c(left = .5, right = .5)), n = 12,
                            persona_template = "A voter who leans {party}.")
instrument <- panel_instrument(item_choice("plan", "Which plan do you prefer?",
                                           c("A", "B")))
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b")

by_party <- function(experiments, ...) {
  experiments$response_text <- ifelse(
    grepl("leans left", vapply(experiments$messages, `[[`, "", "system")),
    "A", "B")
  experiments
}
```

```
r <- panel_administer(panel, instrument, cfg, .runner = by_party)
bench <- data.frame(item_id = "plan", response = c("A", "B"),
                    share = c(.5, .5))
panel_benchmark(r, bench, "toy human study")


if (nzchar(Sys.getenv("GROQ_API_KEY"))) {
  live <- panel_administer(panel, instrument, cfg)
  panel_benchmark(live, bench, "toy human study")
}
```

`panel_bias_audit` *Summarize execution failures, parse failures, and first-option sensitivity*

Description

Counts execution and parse failures by item. For closed items administered with randomized option order, it also applies a chi-squared test to the chosen response and the option shown first. The test does not use the full option permutation.

Usage

```
panel_bias_audit(responses)
```

Arguments

`responses`	A `panel_administer()` result.

Value

A tibble: `item_id`, `n`, `parse_failures`, `execution_failures`, `order_effect_p` (the first-option chi-squared p-value; NA when order was not randomized or cells are too sparse).

Examples

```
panel <- panel_from_margins(list(group = c(A = 1)), n = 4)
instrument <- panel_instrument(
  item_choice("pick", "Choose one.", c("A", "B")),
  randomize = character(0))
config <- LLMR::llm_config("groq", "example-model")
runner <- function(experiments, ...) {
  experiments$response_text <- "A"
  experiments$success <- TRUE
  experiments
}
responses <- panel_administer(panel, instrument, config, .runner = runner)
panel_bias_audit(responses)
```

panel_from_data *Draw a persona panel from microdata rows*

Description

Samples rows from a data frame with replacement, which preserves the joint distribution of the selected attributes, and renders each sampled row as a persona. This is the joint-distribution counterpart of `panel_from_margins()`, which samples attributes independently. The margins the report cites are computed from the source data, one `prop.table(table())` per selected column.

Usage

```
panel_from_data(
  data,
  n,
  persona_template = NULL,
  columns = NULL,
  weights = NULL
)
```

Arguments

`data`	A data frame, one row per source case.
`n`	Panel size.
`persona_template`	Text with `{attribute}` placeholders rendered per persona. `NULL` builds a plain "attribute: value" persona.
`columns`	Attribute columns to keep. Defaults to every column except the `weights` column when one is given.
`weights`	Optional name of a single column of nonnegative sampling weights (rows are drawn with probability proportional to it).

Details

For a reproducible panel, set a seed before calling (the function never sets one itself).

Value

A `silicon_panel`: a tibble with `persona_id`, the selected attribute columns, and `persona`.

Examples

```
set.seed(110)
src <- data.frame(
  education = c("college", "college", "no college", "no college"),
  income    = c("high", "high", "low", "low"),
  weight    = c(2, 2, 1, 1))
```

```
panel_from_data(src, n = 10, columns = c("education", "income"),
                weights = "weight",
                persona_template = "A {education} respondent earning {income}.")
```

`panel_from_margins` *Draw a persona panel from population margins*

Description

Samples `n` personas with attributes drawn independently from the supplied margins, and renders each persona's text from a template. Attributes are sampled independently, so the result does not preserve their joint distribution. Use `panel_from_data()` to sample complete microdata rows.

Usage

```
panel_from_margins(margins, n, persona_template = NULL)
```

Arguments

`margins`	A named list; each element a named probability vector, e.g. `list(age = c("18-34" = .3, "35-64" = .45, "65+" = .25))`. Probabilities are renormalized if they do not sum to 1.
`n`	Panel size.
`persona_template`	Text with `{attribute}` placeholders rendered per persona. `NULL` builds a plain "attribute: value" persona.

Details

For a reproducible panel, set a seed before calling (the function never sets one itself).

Value

A `silicon_panel`: a tibble with `persona_id`, one column per attribute, and `persona` (the rendered text).

Examples

```
set.seed(110)
panel <- panel_from_margins(
  list(cohort = c(young = .3, middle = .45, older = .25),
       party  = c(left = .45, right = .45, independent = .10)),
  n = 50,
  persona_template = "A {cohort} voter who leans {party}."
)
panel
```

`panel_from_personas` *Draw a panel from a persona data frame*

Description

Turns rows of a persona data frame (one respondent per row, demographics plus survey or attitude answers) into a `silicon_panel` whose personas can be administered survey items. It is built for frames following the LLMR persona contract, such as `LLMR::anes_2024_personas`: the demographics and the answers are read with `LLMR::llm_persona_split()` (so answers are keyed by their question wording when the frame carries a dictionary), and each persona is rendered as a person to answer in character.

Usage

```
panel_from_personas(data, n = NULL, rows = NULL, weights = NULL)
```

Arguments

`data`	A persona data frame, such as `LLMR::anes_2024_personas`. Those 100 rows are diversity-selected United States respondents and carry no survey weights, so they stand for no population.
`n`	Optional panel size. With `NULL`, every selected row is used; with a number, rows are sampled (without replacement when `n` does not exceed the pool, otherwise with replacement).
`rows`	Optional row selector: an integer or logical vector, or a predicate `function(df)` returning a logical vector. Applied before sampling.
`weights`	Optional survey weights for the draw: a column name in `data`, or a numeric vector aligned to the selected rows. Used only when `n` is given. `NULL` (default) draws uniformly.

Details

Unlike `panel_from_margins()` and `panel_from_data()`, this constructor keeps each selected respondent's answers together. The `margins` attribute contains the demographic distribution of the selected rows.

For a reproducible draw, set a seed before calling (the function never sets one itself).

Value

A `silicon_panel`: a tibble with `persona_id`, the demographic columns, and `persona`.

See Also

LLMR::anes_2024_personas, `panel_administer()`.

Examples

```
if (requireNamespace("LLMR", quietly = TRUE)) {
  set.seed(110)
  panel <- panel_from_personas(LLMR::anes_2024_personas, n = 8)
}
```

`panel_generics` *Shared generic methods*

Description

LLMRpanel provides `LLMR::diagnostics()`, `LLMR::report()`, and `plot()` methods for `panel_responses` objects. It provides `tibble::as_tibble()` methods for `panel_responses` and `silicon_panel` objects.

`panel_instrument` *Assemble an instrument*

Description

Assemble an instrument

Usage

```
panel_instrument(items, randomize = "option_order")
```

Arguments

`items`	A list of panel_items (`item_likert()`, `item_choice()`, `item_open()`), or a single item, which is wrapped in a list. Ids must be unique.
`randomize`	Which orders to randomize per respondent. The only implemented value is `"option_order"` (the default): the options of a choice item are permuted per response, and a Likert scale is shown reversed for a random half of responses (an ordered scale has two readable orders, not `k!`). `"item_order"` is refused: every persona-item pair is an independent request, so the model never sees a questionnaire order and shuffling one would fabricate an exposure that was not administered. `character(0)` randomizes nothing. What each respondent saw is recorded in the responses.

Value

An object of class `panel_instrument`.

Examples

```
panel_instrument(list(
  item_likert("trust", "How much do you trust the city council?"),
  item_open("reason", "What is the main reason for your answer?")))
```

`panel_items` *Survey items*

Description

Three item types cover most quantitative instruments: a Likert item (an agree-disagree battery row), a forced choice, and an open item (free text, returned verbatim). Likert responses also get a numeric `score` (position on the scale as given, 1-based).

Usage

```
item_likert(
  id,
  text,
  scale = c("strongly disagree", "disagree", "neutral", "agree", "strongly agree")
)

item_choice(id, text, options)

item_open(id, text)
```

Arguments

`id`	Item identifier (unique within an instrument).
`text`	The question text.
`scale`	For `item_likert()`: response options from low to high.
`options`	For `item_choice()`: the choice options.

Value

An object of class `panel_item`.

Examples

```
item_likert("wk4", "A four-day work week would benefit society.")
item_choice("vote", "Which proposal do you prefer?", c("A", "B"))
item_open("why", "In one sentence, why?")
```

panel_usage *Usage diagnostics for an administered panel*

Description

Summarizes token and outcome diagnostics recorded by `panel_administer()` or `panel_batch_fetch()`. The diagnostics are stored in the `usage` field of a `panel_responses` object and summarized by `LLMR::llm_usage()`. Model and provider remain in the returned frame. A supplied `price_table` adds a cost column. The package contains no price table. When the runner records per-call `duration`, its sum is returned as `duration_s`.

Usage

```
panel_usage(responses, price_table = NULL)
```

Arguments

`responses`	A `panel_administer()` result.
`price_table`	Optional price table passed to `LLMR::llm_usage()`.

Value

A one-row usage tibble, or a typed empty tibble when the runner returned no recorded usage fields.

See Also

`panel_administer()`, `LLMR::llm_usage()`.

Examples

```
panel <- panel_from_margins(list(group = c(A = 1)), n = 2)
instrument <- panel_instrument(
  item_choice("pick", "Choose one.", c("A", "B")),
  randomize = character(0))
config <- LLMR::llm_config("groq", "example-model")
runner <- function(experiments, ...) {
  experiments$response_text <- "A"
  experiments$sent_tokens <- 4L
  experiments$rec_tokens <- 1L
  experiments$total_tokens <- 5L
  experiments$success <- TRUE
  experiments
}
responses <- panel_administer(panel, instrument, config, .runner = runner)
panel_usage(responses)
```

`plot.panel_responses` *Plot a benchmark comparison*

Description

Plots the comparison recorded by `panel_benchmark()`. Each covered response level has one point for the panel share and one for the benchmark share, joined by a segment. Response levels follow the instrument's option order, and items appear in separate panels. The method requires a benchmark record.

Usage

```
## S3 method for class 'panel_responses'
plot(x, ...)
```

Arguments

`x`	A `panel_administer()` result that `panel_benchmark()` has been run on (the comparison is stored in `x$benchmark`).
`...`	Ignored; reserved for generic dispatch.

Value

A ggplot object.

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  set.seed(110)
  panel <- panel_from_margins(list(party = c(left = .5, right = .5)),
                              n = 12,
                              persona_template = "A voter who leans {party}.")
  instrument <- panel_instrument(
    item_choice("plan", "Which plan do you prefer?", c("Plan A", "Plan B")))
  cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b")
  by_party <- function(experiments, ...) {
    experiments$response_text <- ifelse(
      grepl("leans left", vapply(experiments$messages, `[[`, "", "system")),
      "Plan A", "Plan B")
    experiments
  }
  r <- panel_administer(panel, instrument, cfg, .runner = by_party)
  bench <- data.frame(item_id = "plan",
                      response = c("Plan A", "Plan B"),
                      share = c(.55, .45))
  plot(panel_benchmark(r, bench, "city survey 2025"))
}
```

`run_panel_studio` *Launch the LLMRpanel Shiny GUI*

Description

Starts a Shiny application that builds a persona panel, administers a choice item or conjoint instrument, and presents the package's diagnostics and design analyses. Choice-item response shares can be compared with an optional benchmark. The application can download the responses and report in a zip file, with the benchmark table when one is available.

Usage

```
run_panel_studio(...)
```

Arguments

`...`	Passed to `shiny::runApp()` (e.g. `port`, `launch.browser`).

Details

The GUI is optional. It needs the suggested packages `shiny`, `bslib`, `DT`, and `LLMR.shiny`; install them first. Keys are read from environment variables only, never pasted into the app; a deterministic demo mode runs offline.

Value

Invisibly, the value of `shiny::runApp()`; called for the side effect of starting the app.

Examples

```
if (interactive() &&
    requireNamespace("shiny", quietly = TRUE) &&
    requireNamespace("LLMR.shiny", quietly = TRUE)) {
  run_panel_studio()
}
```

Index