Package ‘pairwiseLLM’

December 22, 2025

Title Pairwise Comparison Tools for Large Language Model-Based Writing Evaluation

Version 1.1.0

Description Provides a unified framework for generating, submitting, and analyzing pairwise comparisons of writing quality using large language models (LLMs). The package supports live and/or batch evaluation workflows across multiple providers ('OpenAI', 'Anthropic', 'Google Gemini', 'Together AI', and locally-hosted 'Ollama' models), includes bias-tested prompt templates and a flexible template registry, and offers tools for constructing forward and reversed comparison sets to analyze consistency and positional bias. Results can be modeled using Bradley–Terry (1952) <doi:10.2307/2334029> or Elo rating methods to derive writing quality scores. For information on the method of pairwise comparisons, see Thurstone (1927) <doi:10.1037/h0070288> and Heldsinger & Humphry (2010) <doi:10.1007/BF03216919>. For information on Elo ratings, see Clark et al. (2018) <doi:10.1371/journal.pone.0190393>.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Imports curl, dplyr, httr2, jsonlite, rlang, stats, tibble, tidyselect, tools, utils

Suggests BradleyTerry2, EloChoice, knitr, mockery, purrr, readr, rmarkdown, sirt, stringr, testthat (>= 3.0.0), tidyr, withr

Config/testthat/edition 3

URL `https://github.com/shmercer/pairwiseLLM`, `https://shmercer.github.io/pairwiseLLM/`

BugReports `https://github.com/shmercer/pairwiseLLM/issues`

Depends R (>= 4.1)

VignetteBuilder knitr

NeedsCompilation no

Author Sterett H. Mercer [aut, cre] (ORCID:
<https://orcid.org/0000-0002-7940-4221>)

Maintainer Sterett H. Mercer <sterett.mercer@ubc.ca>

Repository CRAN

Date/Publication 2025-12-22 19:10:02 UTC

Contents

`alternate_pair_order` *Deterministically alternate sample order in pairs*

Description

This helper takes a table of paired writing samples (with columns `ID1`, `text1`, `ID2`, and `text2`) and reverses the sample order for every second row (rows 2, 4, 6, ...). This provides a perfectly balanced reversal pattern without the randomness of `randomize_pair_order()`.

Usage

```
alternate_pair_order(pairs)
```

Arguments

`pairs`	A tibble or data frame with columns `ID1`, `text1`, `ID2`, and `text2`.

Details

This is useful when you want a fixed 50/50 mix of original and reversed pairs for bias control, benchmarking, or debugging, without relying on the random number generator or seeds.

Value

A tibble identical to `pairs` except that rows 2, 4, 6, ... have `ID1`/`text1` and `ID2`/`text2` swapped.

Examples

```
data("example_writing_samples")
pairs <- make_pairs(example_writing_samples)

pairs_alt <- alternate_pair_order(pairs)

head(pairs[, c("ID1", "ID2")])
head(pairs_alt[, c("ID1", "ID2")])
```

---

anthropic_compare_pair_live

Live Anthropic (Claude) comparison for a single pair of samples

---

Description

This function sends a single pairwise comparison prompt to the Anthropic Messages API (Claude models) and parses the result into a small tibble.

Usage

```
anthropic_compare_pair_live(
  ID1,
  text1,
  ID2,
  text2,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  tag_prefix = "<BETTER_SAMPLE>",
  tag_suffix = "</BETTER_SAMPLE>",
  api_key = NULL,
  anthropic_version = "2023-06-01",
  reasoning = c("none", "enabled"),
  include_raw = FALSE,
  include_thoughts = NULL,
  ...
)
```

Arguments

`ID1`	Character ID for the first sample.
`text1`	Character string containing the first sample's text.
`ID2`	Character ID for the second sample.
`text2`	Character string containing the second sample's text.
`model`	Anthropic Claude model name (for example `"claude-sonnet-4-5"`, `"claude-haiku-4-5"`, or `"claude-opus-4-5"`).
`trait_name`	Short label for the trait (for example "Overall Quality").
`trait_description`	Full-text definition of the trait.
`prompt_template`	Prompt template string, typically from `set_prompt_template`. The template should embed the full instructions, rubric text, and `<BETTER_SAMPLE>` tagging convention.
`tag_prefix`	Prefix for the better-sample tag. Defaults to `"<BETTER_SAMPLE>"`.
`tag_suffix`	Suffix for the better-sample tag. Defaults to `"</BETTER_SAMPLE>"`.
`api_key`	Optional Anthropic API key. Defaults to `Sys.getenv("ANTHROPIC_API_KEY")`.
`anthropic_version`	Anthropic API version string passed as the `anthropic-version` HTTP header. Defaults to `"2023-06-01"`.
`reasoning`	Character scalar indicating whether to allow more extensive internal "thinking" before the visible answer. Two values are recognised: • `"none"` – standard prompting (recommended default). • `"enabled"` – uses Anthropic's extended thinking mode by sending a `thinking` block with a token budget; this also changes the default `max_tokens` and constrains `temperature`.
`include_raw`	Logical; if `TRUE`, adds a list-column `raw_response` containing the parsed JSON body returned by Anthropic (or `NULL` on parse failure). This is useful for debugging parsing problems.
`include_thoughts`	Logical or `NULL`. When `TRUE` and `reasoning = "none"`, this function upgrades to extended thinking mode by setting `reasoning = "enabled"` before constructing the request, which in turn implies `temperature = 1` and adds a `thinking` block. When `FALSE` and `reasoning = "enabled"`, a warning is issued but extended thinking is still used. When `NULL` (the default), `reasoning` is used as-is.
`...`	Additional Anthropic parameters such as `max_tokens`, `temperature`, `top_p` or a custom `thinking_budget_tokens`, which will be passed through to the Messages API. When `reasoning = "none"` the defaults are: • `temperature = 0` (deterministic behaviour) unless you supply `temperature` explicitly. • `max_tokens = 768` unless you supply `max_tokens`.

When `reasoning = "enabled"` (extended thinking), the Anthropic API imposes additional constraints:

- `temperature` **must** be 1. If you supply a different value, this function will throw an error.
- `thinking_budget_tokens` must satisfy `thinking_budget_tokens >= 1024` and `thinking_budget_tokens < max_tokens`. If you supply a value that violates these constraints, this function will throw an error.
- By default, `max_tokens = 2048` and `thinking_budget_tokens = 1024`.

Details

It mirrors the behaviour and output schema of `openai_compare_pair_live`, but targets Anthropic's `/v1/messages` endpoint. The prompt template, `<BETTER_SAMPLE>` tag convention, and downstream parsing / BT modelling can remain unchanged.

The function is designed to work with Claude models such as Sonnet, Haiku, and Opus in the "4.5" family. You can pass any valid Anthropic model string, for example:

- `"claude-sonnet-4-5"`
- `"claude-haiku-4-5"`
- `"claude-opus-4-5"`

The API typically responds with a dated model string such as `"claude-sonnet-4-5-20250929"` in the `model` field.

Recommended defaults for pairwise writing comparisons

For stable, reproducible comparisons we recommend:

- `reasoning = "none"` with `temperature = 0` and `max_tokens = 768` for standard pairwise scoring.
- `reasoning = "enabled"` when you explicitly want extended thinking; in this mode Anthropic requires `temperature = 1`. The default in this function is `max_tokens = 2048` and `thinking_budget_tokens = 1024`, which satisfies the documented constraints `thinking_budget_tokens >= 1024` and `thinking_budget_tokens < max_tokens`.

When `reasoning = "enabled"`, this function also sends a `thinking` block to the Anthropic API:

```
"thinking": {
  "type": "enabled",
  "budget_tokens": <thinking_budget_tokens>
}
```

Setting `include_thoughts = TRUE` when `reasoning = "none"` is a convenient way to opt into Anthropic's extended thinking mode without changing the `reasoning` argument explicitly. In that case, `reasoning` is upgraded to `"enabled"`, the default `temperature` becomes 1, and a `thinking` block is included in the request. When `reasoning = "none"` and `include_thoughts` is `FALSE` or `NULL`, the default temperature remains 0 unless you explicitly override it.

Value

A tibble with one row and columns:

custom_id ID string of the form `"LIVE_<ID1>_vs_<ID2>"`.

ID1, ID2 The sample IDs you supplied.

model Model name reported by the API.

object_type Anthropic object type (for example `"message"`).

status_code HTTP-style status code (200 if successful).

error_message Error message if something goes wrong; otherwise NA.

thoughts Summarised thinking / reasoning text when `reasoning = "enabled"` and the API returns thinking blocks; otherwise `NA`.

content Concatenated text from the assistant output (excluding thinking blocks).

better_sample "SAMPLE_1", "SAMPLE_2", or NA.

better_id ID1 if SAMPLE_1 is chosen, ID2 if SAMPLE_2 is chosen, otherwise NA.

prompt_tokens Prompt / input token count (if reported).

completion_tokens Completion / output token count (if reported).

total_tokens Total token count (reported by the API or computed as input + output tokens when not provided).

raw_response (Optional) list-column containing the parsed JSON body.

Examples

```
## Not run:
# Requires ANTHROPIC_API_KEY and network access.
library(pairwiseLLM)

data("example_writing_samples", package = "pairwiseLLM")
samples <- example_writing_samples[1:2, ]

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Short, deterministic comparison with no explicit thinking block
res_claude <- anthropic_compare_pair_live(
  ID1               = samples$ID[1],
  text1             = samples$text[1],
  ID2               = samples$ID[2],
  text2             = samples$text[2],
  model             = "claude-sonnet-4-5",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  reasoning         = "none",
  include_raw       = FALSE
)
```

```
res_claude$better_id

# Allow more internal thinking and a longer explanation
res_claude_reason <- anthropic_compare_pair_live(
  ID1               = samples$ID[1],
  text1             = samples$text[1],
  ID2               = samples$ID[2],
  text2             = samples$text[2],
  model             = "claude-sonnet-4-5",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  reasoning         = "enabled",
  include_raw       = TRUE,
  include_thoughts  = TRUE
)

res_claude_reason$total_tokens
substr(res_claude_reason$content, 1, 200)

## End(Not run)
```

---

anthropic_create_batch

Create an Anthropic Message Batch

Description

This is a thin wrapper around Anthropic's `/v1/messages/batches` endpoint. It accepts a list of request objects (each with `custom_id` and `params`) and returns the resulting Message Batch object.

Usage

```
anthropic_create_batch(
  requests,
  api_key = Sys.getenv("ANTHROPIC_API_KEY"),
  anthropic_version = "2023-06-01"
)
```

Arguments

`requests`	List of request objects, each of the form `list(custom_id = <chr>, params = <list>)`. You can obtain this list from the output of `build_anthropic_batch_requests` via `split` / `Map`, or use `run_anthropic_batch_pipeline`.
`api_key`	Optional Anthropic API key. Defaults to `Sys.getenv("ANTHROPIC_API_KEY")`.
`anthropic_version`	Anthropic API version string passed as the `anthropic-version` HTTP header. Defaults to `"2023-06-01"`.

Details

Typically you will not call this directly; instead, use `run_anthropic_batch_pipeline` which builds requests from a tibble of pairs, creates the batch, polls for completion, and downloads the results.

Value

A list representing the Message Batch object returned by Anthropic. Important fields include `id`, `processing_status`, `request_counts`, and (after completion) `results_url`.

Examples

```
## Not run:
# Requires ANTHROPIC_API_KEY and network access.
library(pairwiseLLM)

data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 2, seed = 123) |>
  randomize_pair_order(seed = 456)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

req_tbl <- build_anthropic_batch_requests(
  pairs             = pairs,
  model             = "claude-sonnet-4-5",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl
)

requests <- lapply(seq_len(nrow(req_tbl)), function(i) {
  list(
    custom_id = req_tbl$custom_id[i],
    params    = req_tbl$params[[i]]
  )
})

batch <- anthropic_create_batch(requests = requests)
batch$id
batch$processing_status

## End(Not run)
```

anthropic_download_batch_results

Download Anthropic Message Batch results (.jsonl)

Description

Once a Message Batch has finished processing (status `"ended"`), Anthropic exposes a `results_url` field pointing to a `.jsonl` file containing one JSON object per request result.

Usage

```
anthropic_download_batch_results(
  batch_id,
  output_path,
  api_key = Sys.getenv("ANTHROPIC_API_KEY"),
  anthropic_version = "2023-06-01"
)
```

Arguments

`batch_id`	Character scalar giving the batch ID.
`output_path`	File path where the `.jsonl` results should be written.
`api_key`	Optional Anthropic API key. Defaults to `Sys.getenv("ANTHROPIC_API_KEY")`.
`anthropic_version`	Anthropic API version string passed as the `anthropic-version` HTTP header. Defaults to `"2023-06-01"`.

Details

This helper downloads that file and writes it to disk. It is the Anthropic counterpart to `openai_download_batch_output()`.

Value

Invisibly, the `output_path`.

Examples

```
## Not run:
# Requires ANTHROPIC_API_KEY and network access.
final <- anthropic_poll_batch_until_complete(batch$id)
jsonl_path <- tempfile(fileext = ".jsonl")
anthropic_download_batch_results(final$id, jsonl_path)

## End(Not run)
```

anthropic_get_batch *Retrieve an Anthropic Message Batch by ID*

Description

This retrieves the latest state of a Message Batch using its `id`. It corresponds to a `GET` request on `/v1/messages/batches/<MESSAGE_BATCH_ID>`.

Usage

```
anthropic_get_batch(
  batch_id,
  api_key = Sys.getenv("ANTHROPIC_API_KEY"),
  anthropic_version = "2023-06-01"
)
```

Arguments

`batch_id`	Character scalar giving the batch ID (for example `"msgbatch_01HkcTjaV5uDC8jWR4ZsDV8d"`).
`api_key`	Optional Anthropic API key. Defaults to `Sys.getenv("ANTHROPIC_API_KEY")`.
`anthropic_version`	Anthropic API version string passed as the `anthropic-version` HTTP header. Defaults to `"2023-06-01"`.

Value

A list representing the Message Batch object, including fields such as `id`, `processing_status`, `request_counts`, and (after completion) `results_url`.

Examples

```
## Not run:
# Requires ANTHROPIC_API_KEY and network access.
# After creating a batch:
batch <- anthropic_create_batch(requests = my_requests)
batch_id <- batch$id

latest <- anthropic_get_batch(batch_id)
latest$processing_status

## End(Not run)
```

anthropic_poll_batch_until_complete
Poll an Anthropic Message Batch until completion

Description

This helper repeatedly calls `anthropic_get_batch` until the batch's `processing_status` becomes `"ended"` or a time limit is reached. It is analogous to `openai_poll_batch_until_complete()` but for Anthropic's Message Batches API.

Usage

```
anthropic_poll_batch_until_complete(
  batch_id,
  interval_seconds = 60,
  timeout_seconds = 86400,
  api_key = Sys.getenv("ANTHROPIC_API_KEY"),
  anthropic_version = "2023-06-01",
  verbose = TRUE
)
```

Arguments

`batch_id`	Character scalar giving the batch ID.
`interval_seconds`	Polling interval in seconds. Defaults to 60.
`timeout_seconds`	Maximum total waiting time in seconds. Defaults to 24 hours (`86400` seconds).
`api_key`	Optional Anthropic API key. Defaults to `Sys.getenv("ANTHROPIC_API_KEY")`.
`anthropic_version`	Anthropic API version string passed as the `anthropic-version` HTTP header. Defaults to `"2023-06-01"`.
`verbose`	Logical; if `TRUE`, prints progress messages.

Value

The final Message Batch object as returned by `anthropic_get_batch` once `processing_status == "ended"` or the last object retrieved before timing out.

Examples

```
## Not run:
# Requires ANTHROPIC_API_KEY and network access.
batch <- anthropic_create_batch(requests = my_requests)
final <- anthropic_poll_batch_until_complete(batch$id, interval_seconds = 30)
final$processing_status
```

```
## End(Not run)
```

---

build_anthropic_batch_requests

Build Anthropic Message Batch requests from a tibble of pairs

---

Description

This helper converts a tibble of writing pairs into a list of Anthropic *Message Batch* requests. Each request has a unique `custom_id` of the form `"ANTH_<ID1>_vs_<ID2>"` and a `params` object compatible with the `/v1/messages` API.

Usage

```
build_anthropic_batch_requests(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  reasoning = c("none", "enabled"),
  custom_id_prefix = "ANTH",
  ...
)
```

Arguments

`pairs`	Tibble or data frame with at least columns `ID1`, `text1`, `ID2`, `text2`. Typically created by `make_pairs`, `sample_pairs`, and `randomize_pair_order`.
`model`	Anthropic Claude model name, for example `"claude-sonnet-4-5"`, `"claude-haiku-4-5"`, or `"claude-opus-4-5"`.
`trait_name`	Short label for the trait (for example "Overall Quality").
`trait_description`	Full-text description of the trait or rubric.
`prompt_template`	Prompt template string, typically from `set_prompt_template`. The template should embed your full instructions, rubric text, and `<BETTER_SAMPLE>` tagging convention.
`reasoning`	Character scalar indicating whether to allow extended thinking; one of `"none"` or `"enabled"`. See details above.
`custom_id_prefix`	Prefix for the `custom_id` field. Defaults to `"ANTH"` so that IDs take the form `"ANTH_<ID1>_vs_<ID2>"`.
`...`	Additional Anthropic parameters such as `max_tokens`, `temperature`, `top_p`, or `thinking_budget_tokens`, which will be passed through to the Messages API.

Details

The function mirrors the behaviour of `build_openai_batch_requests` but targets Anthropic's `/v1/messages/batches` endpoint. It applies the same recommended defaults and reasoning constraints as `anthropic_compare_pair_live`:

- `reasoning = "none"`:
  - Default `temperature = 0` (deterministic behaviour), unless you explicitly supply a different `temperature` via `...`.
  - Default `max_tokens = 768`, unless overridden via `max_tokens` in `...`.
- `reasoning = "enabled"` (extended thinking):
  - `temperature` **must** be 1. If you supply a different value in `...`, this function throws an error.
  - Defaults to `max_tokens = 2048` and `thinking_budget_tokens = 1024`, with the constraint `1024 <= thinking_budget_tokens < max_tokens`. Violations of this constraint produce an error.

As a result, when you build batches without extended thinking (`reasoning = "none"`), the effective default temperature is 0. When you opt into extended thinking (`reasoning = "enabled"`), Anthropic's requirement of `temperature = 1` is enforced for all batch requests.

Value

A tibble with one row per pair and two main columns:

custom_id Character ID of the form `"<PREFIX>_<ID1>_vs_<ID2>"`.

params List-column containing the Anthropic Messages API `params` object for each request, ready to be used in the `requests` array of `/v1/messages/batches`.

Examples

```
## Not run:
# Requires ANTHROPIC_API_KEY and network access.
library(pairwiseLLM)

data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 3, seed = 123) |>
  randomize_pair_order(seed = 456)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Standard batch requests without extended thinking
reqs_none <- build_anthropic_batch_requests(
  pairs             = pairs,
  model             = "claude-sonnet-4-5",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  reasoning         = "none"
)

reqs_none

# Batch requests with extended thinking
reqs_reason <- build_anthropic_batch_requests(
  pairs             = pairs,
  model             = "claude-sonnet-4-5",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  reasoning         = "enabled"
)

reqs_reason

## End(Not run)
```

`build_bt_data` *Build Bradley-Terry comparison data from pairwise results*

Description

This function converts pairwise comparison results into the three-column format commonly used for Bradley-Terry models: the first two columns contain object labels and the third column contains the comparison result (1 for a win of the first object, 0 for a win of the second).

Usage

```
build_bt_data(results)
```

Arguments

`results`	A data frame or tibble with columns `ID1`, `ID2`, and `better_id`.

Details

It assumes that the input contains columns `ID1`, `ID2`, and `better_id`, where `better_id` is the ID of the better sample. Rows where `better_id` does not match either `ID1` or `ID2` (including `NA`) are excluded.

Value

A tibble with three columns:

- `object1`: ID from `ID1`
- `object2`: ID from `ID2`
- `result`: numeric value, 1 if `better_id` == `ID1`, 0 if `better_id` == `ID2`

Rows with invalid or missing `better_id` are dropped.

Examples

```
results <- tibble::tibble(
  ID1       = c("S1", "S1", "S2"),
  ID2       = c("S2", "S3", "S3"),
  better_id = c("S1", "S3", "S2")
)

bt_data <- build_bt_data(results)
bt_data

# Using the example writing pairs
data("example_writing_pairs")
bt_ex <- build_bt_data(example_writing_pairs)
head(bt_ex)
```

`build_elo_data` *Build EloChoice comparison data from pairwise results*

Description

This function converts pairwise comparison results into the two-column format used by the **EloChoice** package: one column for the winner and one for the loser of each trial.

Usage

```
build_elo_data(results)
```

Arguments

`results`	A data frame or tibble with columns `ID1`, `ID2`, and `better_id`.

Details

It assumes that the input contains columns `ID1`, `ID2`, and `better_id`, where `better_id` is the ID of the better sample. Rows where `better_id` does not match either `ID1` or `ID2` (including `NA`) are excluded.

Value

A tibble with two columns:

- `winner`: ID of the winning sample
- `loser`: ID of the losing sample

Rows with invalid or missing `better_id` are dropped.

Examples

```
results <- tibble::tibble(
  ID1       = c("S1", "S1", "S2", "S3"),
  ID2       = c("S2", "S3", "S3", "S4"),
  better_id = c("S1", "S3", "S2", "S4")
)

elo_data <- build_elo_data(results)
elo_data
```

`build_gemini_batch_requests` *Build Gemini batch requests from a tibble of pairs*

Description

This helper converts a tibble of writing pairs into a set of Gemini GenerateContent requests suitable for use with the Batch API (`models/*:batchGenerateContent`).

Usage

```
build_gemini_batch_requests(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  thinking_level = c("low", "medium", "high"),
  custom_id_prefix = "GEM",
  temperature = NULL,
  top_p = NULL,
  top_k = NULL,
  max_output_tokens = NULL,
  include_thoughts = FALSE,
  ...
)
```

Arguments

`pairs`	Tibble or data frame with at least columns `ID1`, `text1`, `ID2`, `text2`. Typically created by `make_pairs`, `sample_pairs`, and `randomize_pair_order`.
`model`	Gemini model name, for example `"gemini-3-pro-preview"`. This parameter is not embedded in each request object (the model is provided via the path), but is included here for symmetry with other backends and potential validation.
`trait_name`	Short label for the trait (for example "Overall Quality").
`trait_description`	Full-text description of the trait or rubric.
`prompt_template`	Prompt template string, typically from `set_prompt_template`. The template should embed your full instructions, rubric text, and `<BETTER_SAMPLE>` tagging convention.
`thinking_level`	One of `"low"`, `"medium"`, or `"high"`. This is mapped to Gemini's `thinkingConfig.thinkingLevel`, where `"low"` maps to "Low" and both `"medium"` and `"high"` map to "High". "Medium" currently behaves like "High".
`custom_id_prefix`	Prefix for the `custom_id` field. Defaults to `"GEM"` so that IDs take the form `"GEM_<ID1>_vs_<ID2>"`.
`temperature`	Optional numeric temperature. If `NULL`, it is omitted and Gemini uses its own default.
`top_p`	Optional nucleus sampling parameter. If `NULL`, omitted.
`top_k`	Optional top-k sampling parameter. If `NULL`, omitted.
`max_output_tokens`	Optional integer. If `NULL`, omitted.
`include_thoughts`	Logical; if `TRUE`, sets `thinkingConfig.includeThoughts = TRUE` so that Gemini returns visible chain-of-thought. For most pairwise scoring use cases this should remain `FALSE`.
`...`	Reserved for future extensions. Any `thinking_budget` entries are ignored (Gemini 3 Pro does not support thinking budgets).

Details

Each pair receives a unique `custom_id` of the form `"GEM_<ID1>_vs_<ID2>"` and a corresponding request object containing the prompt and generation configuration.

Value

A tibble with one row per pair and two main columns:

custom_id Character ID of the form `"<PREFIX>_<ID1>_vs_<ID2>"`.

request List-column containing the Gemini GenerateContent request object for each pair.

Examples

```
data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 3, seed = 123) |>
  randomize_pair_order(seed = 456)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

reqs <- build_gemini_batch_requests(
  pairs             = pairs,
  model             = "gemini-3-pro-preview",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  thinking_level    = "low",
  include_thoughts  = TRUE
)

reqs
```

---

build_openai_batch_requests

Build OpenAI batch JSONL lines for paired comparisons

Description

This helper constructs one JSON object per pair of writing samples, suitable for use with the OpenAI batch API. It supports both `/v1/chat/completions` and `/v1/responses` endpoints.

Usage

```
build_openai_batch_requests(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  endpoint = c("chat.completions", "responses"),
  temperature = NULL,
  top_p = NULL,
  logprobs = NULL,
  reasoning = NULL,
  include_thoughts = FALSE,
  request_id_prefix = "EXP"
)
```

Arguments

`pairs`	A data frame or tibble with columns `ID1`, `text1`, `ID2`, and `text2`.
`model`	Character scalar giving the OpenAI model name. Supports standard names (e.g. `"gpt-4.1"`) and date-stamped versions (e.g. `"gpt-5.2-2025-12-11"`).
`trait_name`	Short label for the trait (e.g., "Overall Quality").
`trait_description`	Full-text definition of the trait.
`prompt_template`	Character template containing the placeholders `{TRAIT_NAME}`, `{TRAIT_DESCRIPTION}`, `{SAMPLE_1}`, and `{SAMPLE_2}`. Defaults to `set_prompt_template()`.
`endpoint`	Which OpenAI endpoint to target. One of `"chat.completions"` (default) or `"responses"`.
`temperature`	Optional temperature parameter. Defaults to `0` for standard models (deterministic). Must be `NULL` for reasoning models (enabled).
`top_p`	Optional top_p parameter.
`logprobs`	Optional logprobs parameter.
`reasoning`	Optional reasoning effort for `gpt-5.1/5.2` when using the `/v1/responses` endpoint. Typically `"none"`, `"low"`, `"medium"`, or `"high"`.
`include_thoughts`	Logical; if TRUE and using `responses` endpoint with reasoning, requests a summary. Defaults `reasoning` to `"low"` for gpt-5.1/5.2 if not specified.
`request_id_prefix`	String prefix for `custom_id`; the full ID takes the form `"<prefix>_<ID1>_vs_<ID2>"`.

Value

A tibble with one row per pair and columns:

- `custom_id`: ID string used by the batch API.
- `method`: HTTP method (`"POST"`).
- `url`: Endpoint path (`"/v1/chat/completions"` or `"/v1/responses"`).
- `body`: List column containing the request body.

Examples

```
## Not run:
# Requires OPENAI_API_KEY and network access.
library(pairwiseLLM)

data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 3, seed = 123) |>
  randomize_pair_order(seed = 456)
```

```
td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# 1. Basic chat.completions batch with no thoughts
batch_tbl_chat <- build_openai_batch_requests(
  pairs             = pairs,
  model             = "gpt-4.1",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  endpoint          = "chat.completions",
  temperature       = 0
)

# 2. GPT-5.2-2025-12-11 Responses Batch with Reasoning
batch_resp <- build_openai_batch_requests(
  pairs = pairs,
  model = "gpt-5.2-2025-12-11",
  trait_name = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  endpoint = "responses",
  include_thoughts = TRUE, # implies reasoning="low" if not set
  reasoning = "medium"
)
batch_tbl_chat
batch_tbl_resp

## End(Not run)
```

`build_prompt` *Build a concrete LLM prompt from a template*

Description

This function takes a prompt template (typically from `set_prompt_template`), a trait name and description, and two writing samples, and fills in the required placeholders.

Usage

```
build_prompt(template, trait_name, trait_desc, text1, text2)
```

Arguments

`template`	Character string containing the prompt template.
`trait_name`	Character scalar giving a short label for the trait (e.g., "Overall Quality").
`trait_desc`	Character scalar giving the full definition of the trait.
`text1`	Character scalar containing the text for SAMPLE_1.
`text2`	Character scalar containing the text for SAMPLE_2.

Details

The template must contain the placeholders: {TRAIT_NAME}, {TRAIT_DESCRIPTION}, {SAMPLE_1}, and {SAMPLE_2}.

Value

A single character string containing the completed prompt.

Examples

```
tmpl <- set_prompt_template()
td <- trait_description("overall_quality")
prompt <- build_prompt(
  template   = tmpl,
  trait_name = td$name,
  trait_desc = td$description,
  text1      = "This is sample 1.",
  text2      = "This is sample 2."
)
cat(substr(prompt, 1, 200), "...\n")
```

---

check_llm_api_keys *Check configured API keys for LLM backends*

Description

This function inspects the current R session for configured API keys used by pairwiseLLM. It checks for known environment variables such as OPENAI_API_KEY, ANTHROPIC_API_KEY, and GEMINI_API_KEY, and returns a small tibble summarising which keys are available.

Usage

```
check_llm_api_keys(verbose = TRUE)
```

Arguments

verbose	Logical; if TRUE (default), prints a human-readable summary to the console describing which keys are set and how to configure missing ones.

Details

It does **not** print or return the key values themselves - only whether each key is present. This makes it safe to run in logs, scripts, and shared environments.

Value

A tibble (data frame) with one row per backend and columns:

backend Short backend identifier, e.g. `"openai"`, `"anthropic"`, `"gemini"`, `"together"`.

service Human-readable service name, e.g. `"OpenAI"`, `"Anthropic"`, `"Google Gemini"`, `"Together.ai"`.

env_var Name of the environment variable that is checked.

has_key Logical flag indicating whether the key is set and non-empty.

Examples

```
## Not run:
# In an interactive session, quickly check which keys are configured:
check_llm_api_keys()

# In non-interactive scripts, you can disable messages and just use the
# result:
status <- check_llm_api_keys(verbose = FALSE)
status

## End(Not run)
```

`check_positional_bias` *Check positional bias and bootstrap consistency reliability*

Description

This function diagnoses positional bias in LLM-based paired comparison data and provides a bootstrapped confidence interval for the overall consistency of forward vs. reverse comparisons.

Usage

```
check_positional_bias(
  consistency,
  n_boot = 1000,
  conf_level = 0.95,
  seed = NULL
)
```

Arguments

`consistency` Either:

- A list returned by `compute_reverse_consistency()` that contains a `$details` tibble; or
- A tibble/data frame with columns `key`, `ID1_main`, `ID2_main`, `better_id_main`, `ID1_rev`, `ID2_rev`, `better_id_rev`, and `is_consistent`.

`n_boot`	Integer, number of bootstrap resamples for estimating the distribution of the overall consistency proportion. Default is 1000.
`conf_level`	Confidence level for the bootstrap interval. Default is 0.95.
`seed`	Optional integer seed for reproducible bootstrapping. If `NULL` (default), the current RNG state is used.

Details

It is designed to work with the output of `compute_reverse_consistency`, but will also accept a tibble that looks like its `$details` component.

Value

A list with two elements:

summary A tibble with:

- `n_pairs`: number of unordered pairs
- `prop_consistent`: observed proportion of consistent pairs
- `boot_mean`: mean of bootstrap consistency proportions
- `boot_lwr`, `boot_upr`: bootstrap confidence interval
- `p_sample1_main`: p-value from a binomial test for the null hypothesis that SAMPLE_1 wins 50\ main (forward) comparisons
- `p_sample1_rev`: analogous p-value for the reverse comparisons
- `p_sample1_overall`: p-value from a binomial test for the null that position 1 wins 50\ *all* (forward + reverse) comparisons
- `total_pos1_wins`: total number of wins by position 1 across forward + reverse comparisons
- `total_comparisons`: total number of valid forward + reverse comparisons included in the overall test
- `n_inconsistent`: number of pairs with inconsistent forward vs. reverse outcomes
- `n_inconsistent_pos1_bias`: among inconsistent pairs, how many times the winner is in position 1 in both directions
- `n_inconsistent_pos2_bias`: analogous for position 2

details The input `details` tibble augmented with:

- `winner_pos_main`: `"pos1"` or `"pos2"` (or `NA`) indicating which position won in the main direction
- `winner_pos_rev`: analogous for the reversed direction
- `is_pos1_bias`: logical; `TRUE` if the pair is inconsistent and position 1 wins in both directions
- `is_pos2_bias`: analogous for position 2

Examples

```
# Simple synthetic example
main <- tibble::tibble(
  ID1      = c("S1", "S1", "S2"),
  ID2       = c("S2", "S3", "S3"),
  better_id = c("S1", "S3", "S2")
)

rev <- tibble::tibble(
  ID1       = c("S2", "S3", "S3"),
  ID2       = c("S1", "S1", "S2"),
  better_id = c("S1", "S3", "S2")
)

rc <- compute_reverse_consistency(main, rev)
rc$summary

bias <- check_positional_bias(rc)
bias$summary
```

`compute_reverse_consistency`
Compute consistency between forward and reverse pair comparisons

---

Description

Given two data frames of pairwise comparison results (one for the "forward" ordering of pairs, one for the "reverse" ordering), this function identifies pairs that were evaluated in both orders and computes the proportion of consistent judgments.

Usage

```
compute_reverse_consistency(main_results, reverse_results)
```

Arguments

`main_results`	A data frame or tibble containing pairwise comparison results for the "forward" ordering of pairs, with columns `ID1`, `ID2`, and `better_id`.
`reverse_results`	A data frame or tibble containing results for the corresponding "reverse" ordering, with the same column requirements.

Details

Consistency is defined at the level of IDs: a pair is consistent if the same ID is selected as better in both data frames. This assumes that each result data frame contains at least the columns `ID1`, `ID2`, and `better_id`, where `better_id` is the ID of the better sample (not "SAMPLE_1"/"SAMPLE_2").

Value

A list with two elements:

- `summary`: a tibble with one row and columns `n_pairs`, `n_consistent`, and `prop_consistent`.
- `details`: a tibble with one row per overlapping pair, including columns key, `ID1_main`, `ID2_main`, `ID1_rev`, `ID2_rev`, `better_id_main`, `better_id_rev`, and `is_consistent`.

Pairs for which `better_id` is `NA` in either data frame are excluded from the consistency calculation.

Examples

```
# Simple synthetic example
main <- tibble::tibble(
  ID1       = c("S1", "S1", "S2"),
  ID2       = c("S2", "S3", "S3"),
  better_id = c("S1", "S3", "S2")
)

rev <- tibble::tibble(
  ID1       = c("S2", "S3", "S3"),
  ID2       = c("S1", "S1", "S2"),
  better_id = c("S1", "S3", "S2")
)

rc <- compute_reverse_consistency(main, rev)
rc$summary

# Using the example writing pairs to reverse the first 10 pairs
data("example_writing_pairs")
main2 <- example_writing_pairs[1:10, ]
rev2 <- main2
rev2$ID1 <- main2$ID2
rev2$ID2 <- main2$ID1
rc2 <- compute_reverse_consistency(main2, rev2)
rc2$summary
```

`ensure_only_ollama_model_loaded`
Ensure only one Ollama model is loaded in memory

Description

`ensure_only_ollama_model_loaded()` is a small convenience helper for managing memory when working with large local models via Ollama. It inspects the current set of active models using the `ollama ps` command and attempts to unload any models that are not the one you specify.

Usage

```
ensure_only_ollama_model_loaded(model, verbose = TRUE)
```

Arguments

`model`	Character scalar giving the Ollama model name that should remain loaded (for example `"mistral-small3.2:24b"`, `"qwen3:32b"`, `"gemma3:27b"`). All other models currently reported by `ollama ps` will be candidates for unloading.
`verbose`	Logical; if `TRUE` (the default), the function prints informational messages about the models detected and any unload operations performed. If `FALSE`, the function runs quietly.

Details

This can be useful when running multiple large models (for example `"mistral-small3.2:24b"`, `"qwen3:32b"`, `"gemma3:27b"`) on a single machine, where keeping all of them loaded simultaneously may exhaust GPU or system memory.

The function is intentionally conservative:

- If the `ollama` command is not available on the system *or* `ollama ps` returns an error or empty output, no action is taken and a message is printed when `verbose = TRUE`.
- If no active models are reported, no action is taken.
- Only models with names different from `model` are passed to `ollama stop <name>`.

This helper is not called automatically by the package; it is intended to be used programmatically in development scripts and ad hoc workflows before running comparisons with `ollama_compare_pair_live()` or `submit_ollama_pairs_live()`.

This function relies on the `ollama` command-line interface being available on the system `PATH`. If the command cannot be executed or returns a non-zero status code, the function will issue a message (when `verbose = TRUE`) and return without making any changes.

The exact output format of `ollama ps` is treated as an implementation detail: this helper assumes that the first non-empty line is a header and that subsequent non-empty lines begin with the model name as the first whitespace-separated field. If the format changes in a future version of Ollama, parsing may fail and the function will simply fall back to doing nothing.

Because `ollama stop` affects the global Ollama server state for the current machine, you should only use this helper in environments where you are comfortable unloading models that might be in use by other processes.

Value

Invisibly returns a character vector containing the names of models that were requested to be unloaded (i.e., those passed to `ollama stop`). If no models were unloaded, an empty character vector is returned.

See Also

- `ollama_compare_pair_live()` for single-pair Ollama comparisons.
- `submit_ollama_pairs_live()` for row-wise Ollama comparisons across many pairs.

Examples

```
## Not run:
# Keep only mistral-small3.2:24b loaded in Ollama, unloading any
# other active models
ensure_only_ollama_model_loaded("mistral-small3.2:24b")

## End(Not run)
```

`example_openai_batch_output` *Example OpenAI Batch output (JSONL lines)*

Description

A small character vector containing three example lines from an OpenAI Batch API output file in JSONL format. Each element is a single JSON object representing the result for one batch request.

Usage

```
data("example_openai_batch_output")
```

Format

A character vector of length 3, where each element is a single JSON line (JSONL).

Details

The structure follows the current Batch API output schema, with fields such as `id`, `custom_id`, and a nested `response` object containing `status_code`, `request_id`, and a body that resembles a regular chat completion response. One line illustrates a successful comparison where `<BETTER_SAMPLE>SAMPLE_1</BETTER_SAMPLE` is returned, one illustrates a case where SAMPLE_2 is preferred, and one illustrates an error case with a non-200 status.

This dataset is designed for use in examples and tests of batch output parsing functions. Typical usage is to write the lines to a temporary file and then read/parse them as a JSONL batch file.

Examples

```
data("example_openai_batch_output")

# Inspect the first line
cat(example_openai_batch_output[1], "\n")

# Write to a temporary .jsonl file for parsing
tmp <- tempfile(fileext = ".jsonl")
writeLines(example_openai_batch_output, con = tmp)
tmp
```

`example_writing_pairs` *Example dataset of paired comparisons for writing samples*

Description

A complete set of unordered paired comparison outcomes for the 20 samples in `example_writing_samples`. For each pair of IDs, the `better_id` field indicates which sample is assumed to be better, based on the `quality_score` in `example_writing_samples`.

Usage

```
data("example_writing_pairs")
```

Format

A tibble with 190 rows and 3 variables:

ID1 Character ID of the first sample in the pair.

ID2 Character ID of the second sample in the pair.

better_id Character ID of the sample judged better in this pair (either `ID1` or `ID2`).

Details

This dataset is useful for demonstrating functions that process paired comparisons (e.g., building Bradley-Terry data and fitting `btm` models) without requiring any calls to an LLM.

Examples

```
data("example_writing_pairs")
head(example_writing_pairs)
```

`example_writing_samples` *Example dataset of writing samples*

Description

A small set of 20 writing samples on the topic "Why is writing assessment difficult?", intended for use in examples and tests involving pairing and LLM-based comparisons. The samples vary in quality, approximately from very weak to very strong, and a simple numeric quality score is included to support simulated comparison outcomes.

Usage

```
data("example_writing_samples")
```

Format

A tibble with 20 rows and 3 variables:

ID Character ID for each sample (e.g., `"S01"`).

text Character string with the writing sample.

quality_score Integer from 1 to 10 indicating the intended relative quality of the sample (higher = better).

Examples

```
data("example_writing_samples")
example_writing_samples
```

`fit_bt_model` *Fit a Bradley–Terry model with sirt and fallback to BradleyTerry2*

Description

This function fits a Bradley–Terry paired-comparison model to data prepared by `build_bt_data`. It supports two modeling engines:

- **sirt**: `btm` — the preferred engine, which produces ability estimates, standard errors, and MLE reliability.
- **BradleyTerry2**: `BTm` — used as a fallback if **sirt** is unavailable or fails; computes ability estimates and standard errors, but not reliability.

Usage

```
fit_bt_model(
  bt_data,
  engine = c("auto", "sirt", "BradleyTerry2"),
  verbose = TRUE,
  ...
)
```

Arguments

`bt_data`	A data frame or tibble with exactly three columns: two character ID columns and one numeric `result` column equal to 0 or 1. Usually produced by `build_bt_data`.
`engine`	Character string specifying the modeling engine. One of: `"auto"` (default), `"sirt"`, or `"BradleyTerry2"`.
`verbose`	Logical. If `TRUE` (default), show engine output (iterations, warnings). If `FALSE`, suppress noisy output to keep examples and reports clean.
`...`	Additional arguments passed through to `sirt::btm()` or `BradleyTerry2::BTm()`.

Details

When `engine = "auto"` (the default), the function attempts **sirt** first and automatically falls back to **BradleyTerry2** only if necessary. In all cases, the output format is standardized, so downstream code can rely on consistent fields.

The input `bt_data` must contain exactly three columns:

1. object1: character ID for the first item in the pair
2. object2: character ID for the second item
3. result: numeric indicator (1 = object1 wins, 0 = object2 wins)

Ability estimates (`theta`) represent latent "writing quality" parameters on a log-odds scale. Standard errors are included for both modeling engines. MLE reliability is only available from **sirt**.

Value

A list with the following elements:

engine The engine actually used ("sirt" or "BradleyTerry2").

fit The fitted model object.

theta A tibble with columns:

- `ID`: object identifier
- `theta`: estimated ability parameter
- `se`: standard error of `theta`

reliability MLE reliability (sirt engine only). `NA` for **BradleyTerry2** models.

Examples

```
# Example using built-in comparison data
data("example_writing_pairs")
bt <- build_bt_data(example_writing_pairs)

fit1 <- fit_bt_model(bt, engine = "sirt")
fit2 <- fit_bt_model(bt, engine = "BradleyTerry2")
```

`fit_elo_model` *Fit an EloChoice model to pairwise comparison data*

Description

This function fits an Elo-based paired-comparison model using the **EloChoice** package. It is intended to complement `fit_bt_model` by providing an alternative scoring framework based on Elo ratings rather than Bradley–Terry models.

Usage

```
fit_elo_model(elo_data, runs = 5, verbose = FALSE, ...)
```

Arguments

`elo_data`	A data frame or tibble containing `winner` and `loser` columns. Typically produced using `build_elo_data`.
`runs`	Integer number of randomizations to use in `EloChoice::elochoice`. Default is 5.
`verbose`	Logical. If `TRUE` (default), show any messages/warnings emitted by the underlying fitting functions. If `FALSE`, suppress noisy output to keep examples and reports clean.
`...`	Additional arguments passed to `EloChoice::elochoice()`.

Details

The input `elo_data` must contain two columns:

1. `winner`: ID of the winning sample in each pairwise trial
2. `loser`: ID of the losing sample in each trial

These can be created from standard pairwise comparison output using `build_elo_data`.

Internally, this function calls:

- `elochoice` — to estimate Elo ratings using repeated randomization of trial order;
- `reliability` — to compute unweighted and weighted reliability indices as described in Clark et al. (2018).

If the **EloChoice** package is not installed, a helpful error message is shown telling the user how to install it.

The returned object mirrors the structure of `fit_bt_model` for consistency across scoring engines:

- `engine` — always `"EloChoice"`.
- `fit` — the raw `"elochoice"` object returned by `EloChoice::elochoice()`.
- `elo` — a tibble with columns:
  - `ID`: sample identifier
  - `elo`: estimated Elo rating

  (Unlike Bradley–Terry models, EloChoice does not provide standard errors for these ratings, so none are returned.)
- `reliability` — the mean unweighted reliability index (mean proportion of “upsets” across randomizations).
- `reliability_weighted` — the mean weighted reliability index (weighted version of the upset measure).

Value

A named list with components:

engine Character scalar identifying the scoring engine (`"EloChoice"`).

fit The `"elochoice"` model object.

elo A tibble with columns `ID` and `elo`.

reliability Numeric scalar: mean unweighted reliability index.

reliability_weighted Numeric scalar: mean weighted reliability index.

References

Clark AP, Howard KL, Woods AT, Penton-Voak IS, Neumann C (2018). "Why rate when you could compare? Using the 'EloChoice' package to assess pairwise comparisons of perceived physical strength." *PLOS ONE*, 13(1), e0190393. doi:10.1371/journal.pone.0190393.

Examples

```
data("example_writing_pairs", package = "pairwiseLLM")

elo_data <- build_elo_data(example_writing_pairs)

fit <- fit_elo_model(elo_data, runs = 5, verbose = FALSE)
fit$elo
fit$reliability
fit$reliability_weighted
```

`gemini_compare_pair_live` *Live Google Gemini comparison for a single pair of samples*

Description

This function sends a single pairwise comparison prompt to the Google Gemini Generative Language API (Gemini 3 Pro) and parses the result into a one-row tibble that mirrors the structure used for OpenAI / Anthropic live calls.

Usage

```
gemini_compare_pair_live(
  ID1,
  text1,
  ID2,
  text2,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  api_key = NULL,
  thinking_level = c("low", "medium", "high"),
  temperature = NULL,
  top_p = NULL,
  top_k = NULL,
  max_output_tokens = NULL,
  api_version = "v1beta",
  include_raw = FALSE,
  include_thoughts = FALSE,
  ...
)
```

Arguments

`ID1`	Character ID for the first sample.
`text1`	Character containing the first sample text.
`ID2`	Character ID for the second sample.
`text2`	Character containing the second sample text.
`model`	Gemini model identifier (for example `"gemini-3-pro-preview"`). The value is interpolated into the path `"/{api_version}/models/<model>:generateContent"`.
`trait_name`	Short label for the trait (e.g. `"Overall Quality"`).
`trait_description`	Full-text trait / rubric description.
`prompt_template`	Prompt template string, typically from `set_prompt_template()`. The template should embed `<BETTER_SAMPLE>` tags.
`api_key`	Optional Gemini API key (defaults to `Sys.getenv("GEMINI_API_KEY")`).
`thinking_level`	One of `"low"`, `"medium"`, or `"high"`. This controls the maximum depth of internal reasoning for Gemini 3 Pro. For pairwise scoring, `"low"` is used by default to reduce latency and cost. Currently, the Gemini REST API only supports `"Low"` and `"High"` values; `"medium"` is mapped internally to `"High"` with a warning.
`temperature`	Optional numeric temperature. If `NULL` (default), the parameter is omitted and Gemini uses its own default (currently 1.0).
`top_p`	Optional nucleus sampling parameter. If `NULL`, omitted.
`top_k`	Optional top-k sampling parameter. If `NULL`, omitted.
`max_output_tokens`	Optional maximum output token count. If `NULL`, omitted.
`api_version`	API version to use, default `"v1beta"`. For plain text pairwise comparisons v1beta is recommended.
`include_raw`	Logical; if `TRUE`, the returned tibble includes a `raw_response` list-column with the parsed JSON body.
`include_thoughts`	Logical; if `TRUE`, requests explicit reasoning output from Gemini via `generationConfig$thinkingConfi` and stores the first text part as `thoughts`, with subsequent parts collapsed into `content`. If `FALSE` (default), all text parts are collapsed into `content` and `thoughts` is `NA`.
`...`	Reserved for future extensions. Any `thinking_budget` entry in `...` is ignored (and a warning is emitted) because Gemini 3 does not allow `thinking_budget` and `thinking_level` to be used together.

Details

It expects the prompt template to instruct the model to choose exactly one of SAMPLE_1 or SAMPLE_2 and wrap the decision in <BETTER_SAMPLE> tags, for example:

<BETTER_SAMPLE>SAMPLE_1</BETTER_SAMPLE>

or

<BETTER_SAMPLE>SAMPLE_2</BETTER_SAMPLE>

If `include_thoughts = TRUE`, the function additionally requests Gemini's explicit chain-of-thought style reasoning (\"thoughts\") via the `thinkingConfig` block and stores it in a separate `thoughts` column, while still using the final answer content to detect the `<BETTER_SAMPLE>` tag.

Value

A tibble with one row and columns:

- `custom_id` - `"LIVE_<ID1>_vs_<ID2>"`.
- `ID1`, `ID2` - provided sample IDs.
- `model` - model name returned by the API (or the requested model).
- `object_type` - `"generateContent"` on success, otherwise `NA`.
- `status_code` - HTTP status code (200 on success).
- `error_message` - error message for failures, otherwise `NA`.
- `thoughts` - explicit chain-of-thought style reasoning text if `include_thoughts = TRUE` and the model returns it; otherwise `NA`.
- `content` - concatenated text of the assistant's final answer (used to locate the `<BETTER_SAMPLE>` tag).
- `better_sample` - `"SAMPLE_1"`, `"SAMPLE_2"`, or `NA`.
- `better_id` - `ID1` if `SAMPLE_1` is chosen, `ID2` if `SAMPLE_2`, or `NA`.
- `prompt_tokens`, `completion_tokens`, `total_tokens` - usage counts if reported by the API, otherwise `NA_real_`.

Examples

```
# Requires:
# - GEMINI_API_KEY set in your environment
# - Internet access
# - Billable Gemini API usage
## Not run:
td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

res <- gemini_compare_pair_live(
  ID1              = "S01",
  text1            = "Text 1",
  ID2              = "S02",
  text2            = "Text 2",
  model            = "gemini-3-pro-preview",
  trait_name       = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  thinking_level    = "low",
  include_thoughts  = FALSE,
  include_raw       = FALSE
)

res
res$better_id

## End(Not run)
```

gemini_create_batch — *Create a Gemini Batch job from request objects*

Description

This is a thin wrapper around the REST endpoint `/v1beta/models/<MODEL>:batchGenerateContent`. It accepts a list of GenerateContent request objects and returns the created Batch job.

Usage

```
gemini_create_batch(
  requests,
  model,
  api_key = Sys.getenv("GEMINI_API_KEY"),
  api_version = "v1beta",
  display_name = NULL
)
```

Arguments

`requests`	List of GenerateContent request objects, each of the form `list(contents = ..., generationConfig = ...)`. You can obtain this list from the output of `build_gemini_batch_requests` via `batch$request`.
`model`	Gemini model name, for example `"gemini-3-pro-preview"`.
`api_key`	Optional Gemini API key. Defaults to `Sys.getenv("GEMINI_API_KEY")`.
`api_version`	API version string for the path; defaults to `"v1beta"`.
`display_name`	Optional display name for the batch.

Details

Typically you will not call this directly; instead, use `run_gemini_batch_pipeline` which builds requests from a tibble of pairs, creates the batch, polls for completion, and parses the results.

Value

A list representing the Batch job object returned by Gemini. Important fields include `name`, `metadata$state`, and (after completion) `response$inlinedResponses` or `response$responsesFile`.

Examples

```
# --- Offline preparation: build GenerateContent requests ---

data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 2, seed = 123)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

batch_tbl <- build_gemini_batch_requests(
  pairs             = pairs,
  model             = "gemini-3-pro-preview",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  thinking_level    = "low"
)

# Extract the list of request objects
requests <- batch_tbl$request

# Inspect a single GenerateContent request (purely local)
requests[[1]]

# --- Online step: create the Gemini Batch job ---
# Requires network access and a valid Gemini API key.
## Not run:
batch <- gemini_create_batch(
  requests = requests,
  model    = "gemini-3-pro-preview"
)

batch$name
batch$metadata$state

## End(Not run)
```

`gemini_download_batch_results`

Download Gemini Batch results to a JSONL file

Description

For inline batch requests, Gemini returns results under `response$inlinedResponses$inlinedResponses`. In the v1beta REST API this often comes back as a data frame with one row per request and a `"response"` column, where each `"response"` is itself a data frame of `GenerateContentResponse` objects.

Usage

```
gemini_download_batch_results(
  batch,
  requests_tbl,
  output_path,
  api_key = Sys.getenv("GEMINI_API_KEY"),
  api_version = "v1beta"
)
```

Arguments

`batch`	Either a parsed batch object (as returned by `gemini_get_batch()`) or a character batch name such as `"batches/123..."`.
`requests_tbl`	Tibble/data frame with a `custom_id` column in the same order as the submitted requests.
`output_path`	Path to the JSONL file to create.
`api_key`	Optional Gemini API key (used only when `batch` is a name).
`api_version`	API version (default `"v1beta"`).

Details

This helper writes those results to a local `.jsonl` file where each line is a JSON object of the form:

```
{"custom_id": "<GEM_ID1_vs_ID2>",
 "result": {
   "type": "succeeded",
   "response": { ... GenerateContentResponse ... }
 }}
```

or, when an error occurred:

```
{"custom_id": "<GEM_ID1_vs_ID2>",
 "result": {
   "type": "errored",
   "error": { ... }
 }}
```

Value

Invisibly returns `output_path`.

Examples

```
# This example requires a Gemini API key and network access.
# It assumes you have already created and run a Gemini batch job.
## Not run:
# Name of an existing Gemini batch
batch_name <- "batches/123456"

# Requests table used to create the batch (must include custom_id)
requests_tbl <- tibble::tibble(
  custom_id = c("GEM_S01_vs_S02", "GEM_S03_vs_S04")
)

# Download inline batch results to a local JSONL file
out_file <- tempfile(fileext = ".jsonl")

gemini_download_batch_results(
  batch        = batch_name,
  requests_tbl = requests_tbl,
  output_path  = out_file
)

# Inspect the downloaded JSONL
readLines(out_file, warn = FALSE)

## End(Not run)
```

gemini_get_batch *Retrieve a Gemini Batch job by name*

Description

This retrieves the latest state of a Batch job using its name as returned by gemini_create_batch.

Usage

```
gemini_get_batch(
  batch_name,
  api_key = Sys.getenv("GEMINI_API_KEY"),
  api_version = "v1beta"
)
```

Arguments

batch_name	Character scalar giving the batch name.
api_key	Optional Gemini API key. Defaults to Sys.getenv("GEMINI_API_KEY").
api_version	API version string for the path; defaults to "v1beta".

Details

It corresponds to a GET request on `/v1beta/<BATCH_NAME>`, where `BATCH_NAME` is a string such as `"batches/123456"`.

Value

A list representing the Batch job object.

Examples

```
# Offline: basic batch name validation / object you would pass
batch_name <- "batches/123456"

# Online: retrieve the batch state from Gemini (requires API key + network)
## Not run:
batch <- gemini_get_batch(batch_name = batch_name)
batch$name
batch$metadata$state

## End(Not run)
```

gemini_poll_batch_until_complete

Poll a Gemini Batch job until completion

Description

This helper repeatedly calls `gemini_get_batch` until the batch's `metadata$state` enters a terminal state or a time limit is reached. For the REST API, states have the form "BATCH_STATE_*".

Usage

```
gemini_poll_batch_until_complete(
  batch_name,
  interval_seconds = 60,
  timeout_seconds = 86400,
  api_key = Sys.getenv("GEMINI_API_KEY"),
  api_version = "v1beta",
  verbose = TRUE
)
```

Arguments

`batch_name`	Character scalar giving the batch name.
`interval_seconds`	Polling interval in seconds. Defaults to 60.

`timeout_seconds`	Maximum total waiting time in seconds. Defaults to 24 hours (86400 seconds).
`api_key`	Optional Gemini API key. Defaults to `Sys.getenv("GEMINI_API_KEY")`.
`api_version`	API version string for the path; defaults to `"v1beta"`.
`verbose`	Logical; if `TRUE`, prints progress messages.

Value

The final Batch job object as returned by `gemini_get_batch`.

Examples

```
# Offline: polling parameters and batch name are plain R objects
batch_name <- "batches/123456"

# Online: poll until the batch reaches a terminal state (requires network)
## Not run:
final_batch <- gemini_poll_batch_until_complete(
  batch_name       = batch_name,
  interval_seconds = 10,
  timeout_seconds  = 600,
  verbose          = TRUE
)
final_batch$metadata$state

## End(Not run)
```

`get_prompt_template` *Retrieve a named prompt template*

Description

This function retrieves a prompt template from either:

- the user registry (see `register_prompt_template`), or
- a built-in template stored under `inst/templates`.

Usage

```
get_prompt_template(name = "default")
```

Arguments

`name`	Character scalar giving the template name.

Details

The function first checks user-registered templates, then looks for a built-in text file `inst/templates/<name>.txt`. The special name `"default"` falls back to `set_prompt_template()` when no user-registered or built-in template is found.

Value

A single character string containing the prompt template.

See Also

`register_prompt_template`, `list_prompt_templates`, `remove_prompt_template`

Examples

```
# Get the built-in default template
tmpl_default <- get_prompt_template("default")

# List available template names
list_prompt_templates()
```

`list_prompt_templates` *List available prompt templates*

Description

This function lists template names that are available either as built-in text files under `inst/templates` or as user-registered templates in the current R session.

Usage

```
list_prompt_templates(include_builtin = TRUE, include_registered = TRUE)
```

Arguments

`include_builtin`	Logical; include built-in template names (the default is `TRUE`).
`include_registered`	Logical; include user-registered names (the default is `TRUE`).

Details

Built-in templates are identified by files named `<name>.txt` within `inst/templates`. For example, a file `inst/templates/minimal.txt` will be listed as `"minimal"`.

Value

A sorted character vector of unique template names.

Examples

```
list_prompt_templates()
```

`llm_compare_pair` *Backend-agnostic live comparison for a single pair of samples*

Description

`llm_compare_pair()` is a thin wrapper around backend-specific comparison functions. It currently supports the `"openai"`, `"anthropic"`, `"gemini"`, `"together"`, and `"ollama"` backends and forwards the call to the appropriate live comparison helper:

- `"openai"` → `openai_compare_pair_live()`
- `"anthropic"` → `anthropic_compare_pair_live()`
- `"gemini"` → `gemini_compare_pair_live()`
- `"together"` → `together_compare_pair_live()`
- `"ollama"` → `ollama_compare_pair_live()`

Usage

```
llm_compare_pair(
  ID1,
  text1,
  ID2,
  text2,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  backend = c("openai", "anthropic", "gemini", "together", "ollama"),
  endpoint = c("chat.completions", "responses"),
  api_key = NULL,
  include_raw = FALSE,
  ...
)
```

Arguments

`ID1`	Character ID for the first sample.
`text1`	Character string containing the first sample's text.
`ID2`	Character ID for the second sample.
`text2`	Character string containing the second sample's text.

`model`	Model identifier for the chosen backend. For `"openai"` this should be an OpenAI model name (for example `"gpt-4.1"`, `"gpt-5.1"`). For `"anthropic"` and `"gemini"`, use the corresponding provider model names (for example `"claude-4-5-sonnet"` or `"gemini-3-pro-preview"`). For "together", use Together.ai model identifiers such as `"deepseek-ai/DeepSeek-R1"` or `"deepseek-ai/DeepSeek-V3"`. For `"ollama"`, use a local model name known to the Ollama server (for example `"mistral-small3.2:24b"`, `"qwen3:32b"`, `"gemma3:27b"`).
`trait_name`	Short label for the trait (for example `"Overall Quality"`).
`trait_description`	Full-text definition of the trait.
`prompt_template`	Prompt template string, typically from `set_prompt_template()`.
`backend`	Character scalar indicating which LLM provider to use. One of `"openai"`, `"anthropic"`, `"gemini"`, `"together"`, or `"ollama"`.
`endpoint`	Character scalar specifying which endpoint family to use for backends that support multiple live APIs. For the `"openai"` backend this must be one of `"chat.completions"` or `"responses"`, matching `openai_compare_pair_live()`. For `"anthropic"`, `"gemini"`, and `"ollama"`, this argument is currently ignored.
`api_key`	Optional API key for the selected backend. If `NULL`, the backend-specific helper will use its own default environment variable (for example `OPENAI_API_KEY`, `ANTHROPIC_API_KEY`, `GEMINI_API_KEY`, `TOGETHER_API_KEY`). For `"ollama"`, this argument is ignored (no API key is required for local inference).
`include_raw`	Logical; if `TRUE`, the returned tibble includes a `raw_response` list-column with the parsed JSON body (or `NULL` on parse failure). Support for this may vary across backends.
`...`	Additional backend-specific parameters. For `"openai"` these are passed on to `openai_compare_pair_live()` and typically include arguments such as `temperature`, `top_p`, `logprobs`, `reasoning`, and `include_thoughts`. For `"anthropic"` and `"gemini"` they are forwarded to the corresponding live helper and may include parameters such as `reasoning`, `include_thoughts`, `max_output_tokens`, or provider-specific options. For `"ollama"`, arguments are forwarded to `ollama_compare_pair_live()` and may include `host`, `think`, `num_ctx`, and other Ollama-specific controls.

Details

All backends are expected to return a tibble with a compatible structure, including:

- `custom_id`, `ID1`, `ID2`
- `model`, `object_type`, `status_code`, `error_message`
- `thoughts` (reasoning / thinking text when available)
- `content` (visible assistant output)
- `better_sample`, `better_id`
- `prompt_tokens`, `completion_tokens`, `total_tokens`

For the `"openai"` backend, the `endpoint` argument controls whether the Chat Completions API (`"chat.completions"`) or the Responses API (`"responses"`) is used. For the `"anthropic"`, `"gemini"`, and `"ollama"` backends, `endpoint` is currently ignored and the default live API for that provider is used.

Value

A tibble with one row and the same columns as the underlying backend-specific live helper (for example `openai_compare_pair_live()` for "openai"). All backends are intended to return a compatible structure including `thoughts`, `content`, and token counts.

See Also

- `openai_compare_pair_live()`, `anthropic_compare_pair_live()`, `gemini_compare_pair_live()`, `together_compare_pair_live()`, and `ollama_compare_pair_live()` for backend-specific implementations.
- `submit_llm_pairs()` for row-wise comparisons over a tibble of pairs.
- `build_bt_data()` and `fit_bt_model()` for Bradley–Terry modelling of comparison results.

Examples

```
## Not run:
# Requires an API key for the chosen cloud backend. For OpenAI, set
# OPENAI_API_KEY in your environment. Running these examples will incur
# API usage costs.
#
# For local Ollama use, an Ollama server must be running and the models
# must be pulled in advance. No API key is required for the `"ollama"`
# backend.

data("example_writing_samples", package = "pairwiseLLM")
samples <- example_writing_samples[1:2, ]

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Single live comparison using the OpenAI backend and chat.completions
res_live <- llm_compare_pair(
  ID1               = samples$ID[1],
  text1             = samples$text[1],
  ID2               = samples$ID[2],
  text2             = samples$text[2],
  model             = "gpt-4.1",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  backend           = "openai",
  endpoint          = "chat.completions",
  temperature       = 0
)

res_live$better_id

# Using the OpenAI responses endpoint with gpt-5.1 and reasoning = "low"
res_live_gpt5 <- llm_compare_pair(
  ID1               = samples$ID[1],
  text1             = samples$text[1],
  ID2                = samples$ID[2],
  text2              = samples$text[2],
  model              = "gpt-5.1",
  trait_name         = td$name,
  trait_description = td$description,
  prompt_template    = tmpl,
  backend            = "openai",
  endpoint           = "responses",
  reasoning          = "low",
  include_thoughts   = TRUE,
  temperature        = NULL,
  top_p              = NULL,
  logprobs           = NULL,
  include_raw        = TRUE
)

str(res_live_gpt5$raw_response[[1]], max.level = 2)

# Example: single live comparison using a local Ollama backend
res_ollama <- llm_compare_pair(
  ID1 = samples$ID[1],
  text1 = samples$text[1],
  ID2 = samples$ID[2],
  text2 = samples$text[2],
  model = "mistral-small3.2:24b",
  trait_name = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  backend = "ollama",
  host = getOption(
    "pairwiseLLM.ollama_host",
    "http://127.0.0.1:11434"
  ),
  think = FALSE
)

res_ollama$better_id

## End(Not run)
```

llm_download_batch_results

Extract results from a pairwiseLLM batch object

Description

Helper to extract the parsed results tibble from a batch object returned by `llm_submit_pairs_batch()`. This is a thin wrapper around the `results` element returned by backend-specific batch pipelines and is designed to be forward-compatible with future, more asynchronous batch workflows.

Usage

```
llm_download_batch_results(x, ...)
```

Arguments

x	An object returned by `llm_submit_pairs_batch()` (class "pairwiseLLM_batch"), or a compatible list that contains a `results` element.
...	Reserved for future use; currently ignored.

Value

A tibble containing batch comparison results in the standard pairwiseLLM schema.

Examples

```
## Not run:
# Requires running a provider batch job first (API key + internet + cost).

batch <- llm_submit_pairs_batch(
  pairs             = tibble::tibble(
    ID1   = "S01",
    text1 = "Text 1",
    ID2   = "S02",
    text2 = "Text 2"
  ),
  backend           = "openai",
  model             = "gpt-4.1",
  trait_name        = trait_description("overall_quality")$name,
  trait_description = trait_description("overall_quality")$description,
  prompt_template   = set_prompt_template()
)

res <- llm_download_batch_results(batch)
res

## End(Not run)
```

---

llm_submit_pairs_batch

Submit pairs to an LLM backend via batch API

Description

`llm_submit_pairs_batch()` is a backend-agnostic front-end for running provider batch pipelines (OpenAI, Anthropic, Gemini). Together.ai and Ollama are supported only for live comparisons.

It mirrors `submit_llm_pairs()` but uses the provider batch APIs under the hood via `run_openai_batch_pipeline()`, `run_anthropic_batch_pipeline()`, and `run_gemini_batch_pipeline()`.

For OpenAI, this helper will by default:

- Use the `chat.completions` batch style for most models, and
- Automatically switch to the `responses` style endpoint when:
  - `model` starts with `"gpt-5.1"` or `"gpt-5.2"` (including date-stamped versions like `"gpt-5.2-2025-12-11"`) and
  - either `include_thoughts = TRUE` **or** a non-`"none"` `reasoning` effort is supplied in `...`.

Temperature Defaults: For OpenAI, if `temperature` is not specified in `...`:

- It defaults to `0` (deterministic) for standard models or when reasoning is disabled (`reasoning = "none"`) on supported models (5.1/5.2).
- It remains `NULL` (API default) when reasoning is enabled, as the API does not support temperature with reasoning.

For Anthropic, standard and date-stamped model names (e.g. `"claude-sonnet-4-5-20250929"`) are supported. This helper delegates temperature and extended-thinking behaviour to `run_anthropic_batch_pipeline()` and `build_anthropic_batch_requests()`, which apply the following rules:

- When `reasoning = "none"` (no extended thinking), the default temperature is `0` (deterministic) unless you explicitly supply a different `temperature` in `...`.
- When `reasoning = "enabled"` (extended thinking), Anthropic requires `temperature = 1`. If you supply a different value in `...`, an error is raised. Default values in this mode are `max_tokens = 2048` and `thinking_budget_tokens = 1024`, subject to `1024 <= thinking_budget_tokens < max_tok`
- Setting `include_thoughts = TRUE` while leaving `reasoning = "none"` causes `run_anthropic_batch_pipeline()` to upgrade to `reasoning = "enabled"`, which implies `temperature = 1` for the batch.

For Gemini, this helper simply forwards `include_thoughts` and other arguments to `run_gemini_batch_pipeline()`, which is responsible for interpreting any thinking-related options.

Currently, this function *synchronously* runs the full batch pipeline for each backend (build requests, create batch, poll until complete, download results, parse). The returned object contains both metadata and a normalized `results` tibble. See `llm_download_batch_results()` to extract the results.

Usage

```
llm_submit_pairs_batch(
  pairs,
  backend = c("openai", "anthropic", "gemini"),
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  include_thoughts = FALSE,
  include_raw = FALSE,
  ...
)
```

Arguments

`pairs`	A data frame or tibble of pairs with columns `ID1`, `text1`, `ID2`, and `text2`. Additional columns are allowed and will be carried through where supported.
`backend`	Character scalar; one of `"openai"`, `"anthropic"`, or `"gemini"`. Matching is case-insensitive.
`model`	Character scalar model name to use for the batch job. • For `"openai"`, use models like `"gpt-4.1"`, `"gpt-5.1"`, or `"gpt-5.2"` (including date-stamped versions like `"gpt-5.2-2025-12-11"`). • For `"anthropic"`, use provider names like `"claude-4-5-sonnet"` or date-stamped versions like `"claude-sonnet-4-5-20250929"`. • For `"gemini"`, use names like `"gemini-3-pro-preview"`.
`trait_name`	A short name for the trait being evaluated (e.g. `"overall_quality"`).
`trait_description`	A human-readable description of the trait.
`prompt_template`	A prompt template created by `set_prompt_template()` or a compatible character scalar.
`include_thoughts`	Logical; whether to request and parse model "thoughts" (where supported). • For OpenAI GPT-5.1/5.2, setting this to `TRUE` defaults to the `responses` endpoint. • For Anthropic, setting this to `TRUE` implies `reasoning = "enabled"` (unless overridden) and sets `temperature = 1`.
`include_raw`	Logical; whether to include raw provider responses in the result (where supported by backends).
`...`	Additional arguments passed through to the backend-specific `run_*_batch_pipeline()` functions. This can include provider-specific options such as temperature or batch configuration fields. For OpenAI, this may include `endpoint`, `temperature`, `top_p`, `logprobs`, `reasoning`, etc. For Anthropic, this may include `reasoning`, `max_tokens`, `temperature`, or `thinking_budget_tokens`.

Value

A list of class `"pairwiseLLM_batch"` containing at least:

- `backend`: the backend identifier (`"openai"`, `"anthropic"`, `"gemini"`),
- `batch_input_path`: path to the JSONL request file (if applicable),
- `batch_output_path`: path to the JSONL output file (if applicable),
- `batch`: provider-specific batch object (e.g., job metadata),
- `results`: a tibble of parsed comparison results in the standard pairwiseLLM schema.

Additional fields returned by the backend-specific pipeline functions are preserved.

Examples

```
# Requires:
# - Internet access
# - Provider API key set in your environment (OPENAI_API_KEY /
#   ANTHROPIC_API_KEY / GEMINI_API_KEY)
# - Billable API usage
## Not run:
pairs <- tibble::tibble(
  ID1   = c("S01", "S03"),
  text1 = c("Text 1", "Text 3"),
  ID2   = c("S02", "S04"),
  text2 = c("Text 2", "Text 4")
)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# OpenAI batch
batch_openai <- llm_submit_pairs_batch(
  pairs             = pairs,
  backend           = "openai",
  model             = "gpt-4.1",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  include_thoughts  = FALSE
)
res_openai <- llm_download_batch_results(batch_openai)

# Anthropic batch
batch_anthropic <- llm_submit_pairs_batch(
  pairs             = pairs,
  backend           = "anthropic",
  model             = "claude-4-5-sonnet",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  include_thoughts  = FALSE
)
res_anthropic <- llm_download_batch_results(batch_anthropic)

# Gemini batch
batch_gemini <- llm_submit_pairs_batch(
  pairs             = pairs,
  backend           = "gemini",
  model             = "gemini-3-pro-preview",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  include_thoughts  = TRUE
)
res_gemini <- llm_download_batch_results(batch_gemini)
```

```
## End(Not run)
```

---

make_pairs *Create all unordered pairs of writing samples*

Description

Given a data frame of samples with columns `ID` and `text`, this function generates all unordered pairs (combinations) of samples. Each pair appears exactly once, with `ID1` < `ID2` in lexicographic order.

Usage

```
make_pairs(samples)
```

Arguments

samples	A tibble or data frame with columns `ID` and `text`.

Value

A tibble with columns:

- `ID1`, `text1`
- `ID2`, `text2`

Examples

```
samples <- tibble::tibble(
  ID   = c("S1", "S2", "S3"),
  text = c("Sample 1", "Sample 2", "Sample 3")
)

pairs_all <- make_pairs(samples)
pairs_all

# Using the built-in example data
data("example_writing_samples")
pairs_example <- make_pairs(example_writing_samples)
nrow(pairs_example) # should be choose(10, 2) = 45
```

ollama_compare_pair_live

Live Ollama comparison for a single pair of samples

Description

`ollama_compare_pair_live()` sends a single pairwise comparison prompt to a local Ollama server and parses the result into the standard pairwiseLLM tibble format.

Usage

```
ollama_compare_pair_live(
  ID1,
  text1,
  ID2,
  text2,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  host = getOption("pairwiseLLM.ollama_host", "http://127.0.0.1:11434"),
  tag_prefix = "<BETTER_SAMPLE>",
  tag_suffix = "</BETTER_SAMPLE>",
  think = FALSE,
  num_ctx = 8192L,
  include_raw = FALSE,
  ...
)
```

Arguments

`ID1`	Character ID for the first sample.
`text1`	Character string containing the first sample's text.
`ID2`	Character ID for the second sample.
`text2`	Character string containing the second sample's text.
`model`	Ollama model name (for example `"mistral-small3.2:24b"`, `"qwen3:32b"`, `"gemma3:27b"`).
`trait_name`	Short label for the trait (for example `"Overall Quality"`).
`trait_description`	Full-text definition of the trait.
`prompt_template`	Prompt template string, typically from `set_prompt_template()`.
`host`	Base URL of the Ollama server. Defaults to the option `getOption("pairwiseLLM.ollama_host", "http://127.0.0.1:11434")`.
`tag_prefix`	Prefix for the better-sample tag. Defaults to `"<BETTER_SAMPLE>"`.

`tag_suffix`	Suffix for the better-sample tag. Defaults to `"</BETTER_SAMPLE>"`.
`think`	Logical; if `TRUE` and the model is a Qwen model (name starts with `"qwen"`), the temperature is set to `0.6`. Otherwise the temperature is `0`. The `think` argument does not itself modify the HTTP request body; it is used only for choosing the temperature, but the function will parse a `thinking` field from the response whenever one is present.
`num_ctx`	Integer; context window to use via `options$num_ctx`. The default is `8192L`.
`include_raw`	Logical; if `TRUE`, adds a list-column `raw_response` containing the parsed JSON body returned by Ollama (or `NULL` on parse failure). This is useful for debugging.
`...`	Reserved for future extensions.

Details

The function targets the `/api/generate` endpoint on a running Ollama instance and expects a single non-streaming response. Model names should match those available in your Ollama installation (for example `"mistral-small3.2:24b"`, `"qwen3:32b"`, `"gemma3:27b"`).

Temperature and context length are controlled as follows:

- By default, `temperature = 0` for all models.
- For Qwen models (model names beginning with `"qwen"`) and `think = TRUE`, `temperature` is set to `0.6`.
- The context window is set via `options$num_ctx`, which defaults to `8192L` but may be overridden via the `num_ctx` argument.

If the Ollama response includes a `thinking` field (as described in the Ollama API), that string is stored in the `thoughts` column of the returned tibble; otherwise `thoughts` is `NA`. This allows pairwiseLLM to consume Ollama's native thinking output in a way that is consistent with other backends that expose explicit reasoning traces.

The Ollama backend is intended to be compatible with the existing OpenAI, Anthropic, and Gemini backends, so the returned tibble can be used directly with downstream helpers such as `build_bt_data()` and `fit_bt_model()`.

In typical workflows, users will call `llm_compare_pair()` with `backend = "ollama"` rather than using `ollama_compare_pair_live()` directly. The direct helper is exported so that advanced users can work with Ollama in a more explicit and backend-specific way.

The function assumes that:

- An Ollama server is running and reachable at `host`.
- The requested `model` has already been pulled, for example via `ollama pull mistral-small3.2:24b` on the command line.

When the Ollama response includes a `thinking` field (as documented in the Ollama API), that string is copied into the `thoughts` column of the returned tibble; otherwise `thoughts` is `NA`. This parsed thinking output can be logged, inspected, or analyzed alongside the visible comparison decisions.

Value

A tibble with one row and columns:

- `custom_id` – ID string of the form `"LIVE_<ID1>_vs_<ID2>"`.
- `ID1`, `ID2` – the sample IDs supplied to the function.
- `model` – model name reported by the API (or the requested model).
- `object_type` – backend object type (for example `"ollama.generate"`).
- `status_code` – HTTP-style status code (`200` if successful).
- `error_message` – error message if something goes wrong; otherwise `NA`.
- `thoughts` – reasoning / thinking text when a `thinking` field is returned by Ollama; otherwise `NA`.
- `content` – visible response text from the model (from the `response` field).
- `better_sample` – `"SAMPLE_1"`, `"SAMPLE_2"`, or `NA`, based on tags found in `content`.
- `better_id` – `ID1` if `"SAMPLE_1"` is chosen, `ID2` if `"SAMPLE_2"` is chosen, otherwise `NA`.
- `prompt_tokens` – prompt / input token count (if reported).
- `completion_tokens` – completion / output token count (if reported).
- `total_tokens` – total token count (if reported).
- `raw_response` – optional list-column containing the parsed JSON body (present only when `include_raw = TRUE`).

See Also

- `submit_ollama_pairs_live()` for single-backend, row-wise comparisons.
- `llm_compare_pair()` for backend-agnostic single-pair comparisons.
- `submit_llm_pairs()` for backend-agnostic comparisons over tibbles of pairs.

Examples

```
## Not run:
# Requires a running Ollama server and locally available models.

data("example_writing_samples", package = "pairwiseLLM")

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

ID1 <- example_writing_samples$ID[1]
ID2 <- example_writing_samples$ID[2]
text1 <- example_writing_samples$text[1]
text2 <- example_writing_samples$text[2]

# Make sure an Ollama server is running

# mistral example
res_mistral <- ollama_compare_pair_live(
  ID1               = ID1,
  text1             = text1,
  ID2               = ID2,
  text2             = text2,
  model             = "mistral-small3.2:24b",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl
)

res_mistral$better_id

# qwen example with reasoning
res_qwen_think <- ollama_compare_pair_live(
  ID1               = ID1,
  text1             = text1,
  ID2               = ID2,
  text2             = text2,
  model             = "qwen3:32b",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  think             = TRUE,
  include_raw       = TRUE
)

res_qwen_think$better_id
res_qwen_think$thoughts

## End(Not run)
```

openai_compare_pair_live

Live OpenAI comparison for a single pair of samples

Description

This function sends a single pairwise comparison prompt to the OpenAI API and parses the result into a small tibble. It is the live / on-demand analogue of `build_openai_batch_requests` plus `parse_openai_batch_output`.

Usage

```
openai_compare_pair_live(
  ID1,
  text1,
  ID2,
  text2,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  endpoint = c("chat.completions", "responses"),
  tag_prefix = "<BETTER_SAMPLE>",
  tag_suffix = "</BETTER_SAMPLE>",
  api_key = NULL,
  include_raw = FALSE,
  ...
)
```

Arguments

`ID1`	Character ID for the first sample.
`text1`	Character string containing the first sample's text.
`ID2`	Character ID for the second sample.
`text2`	Character string containing the second sample's text.
`model`	OpenAI model name (e.g. "gpt-4.1", "gpt-5.2-2025-12-11").
`trait_name`	Short label for the trait (e.g. "Overall Quality").
`trait_description`	Full-text definition of the trait.
`prompt_template`	Prompt template string.
`endpoint`	Which OpenAI endpoint to use: `"chat.completions"` or `"responses"`.
`tag_prefix`	Prefix for the better-sample tag.
`tag_suffix`	Suffix for the better-sample tag.
`api_key`	Optional OpenAI API key.
`include_raw`	Logical; if TRUE, adds a `raw_response` column.
`...`	Additional OpenAI parameters, for example `temperature`, `top_p`, `logprobs`, `reasoning`, and (optionally) `include_thoughts`. The same validation rules for gpt-5 models are applied as in `build_openai_batch_requests`. When using the Responses endpoint with reasoning models, you can request reasoning summaries in the `thoughts` column by setting `endpoint = "responses"`, a non-"none" reasoning effort, and `include_thoughts = TRUE`.

Details

It supports both the Chat Completions endpoint ("/v1/chat/completions") and the Responses endpoint ("/v1/responses", for example gpt-5.1 with reasoning), using the same prompt template and model / parameter rules as the batch pipeline.

For the Responses endpoint, the function collects:

- Reasoning / "thoughts" text (if available) into the `thoughts` column.
- Visible assistant output into the `content` column.

Temperature Defaults: If `temperature` is not provided in `...`:

- It defaults to `0` (deterministic) for standard models or when reasoning is disabled.
- It remains `NULL` when reasoning is enabled, as the API does not support temperature in that mode.

Value

A tibble with one row and columns:

custom_id ID string of the form `"LIVE_<ID1>_vs_<ID2>"`.

ID1, ID2 The sample IDs you supplied.

model Model name reported by the API.

object_type OpenAI object type (for example "chat.completion" or "response").

status_code HTTP-style status code (200 if successful).

error_message Error message if something goes wrong; otherwise NA.

thoughts Reasoning / thinking summary text when available, otherwise NA.

content Concatenated text from the assistant's visible output. For the Responses endpoint this is taken from the `type = "message"` output items and does not include reasoning summaries.

better_sample "SAMPLE_1", "SAMPLE_2", or NA.

better_id ID1 if SAMPLE_1 is chosen, ID2 if SAMPLE_2 is chosen, otherwise NA.

prompt_tokens Prompt / input token count (if reported).

completion_tokens Completion / output token count (if reported).

total_tokens Total token count (if reported).

raw_response (Optional) list-column containing the parsed JSON body.

Examples

```
## Not run:
# Requires API key set and internet access

# 1. Standard comparison using GPT-4.1
res <- openai_compare_pair_live(
  ID1 = "A", text1 = "Text A...",
  ID2 = "B", text2 = "Text B...",
  model = "gpt-4.1",
  trait_name = "clarity",
  trait_description = "Which text is clearer?",
  temperature = 0
)

# 2. Reasoning comparison using GPT-5.2
res_reasoning <- openai_compare_pair_live(
  ID1 = "A", text1 = "Text A...",
  ID2 = "B", text2 = "Text B...",
  model = "gpt-5.2-2025-12-11",
  trait_name = "clarity",
  trait_description = "Which text is clearer?",
  endpoint = "responses",
  include_thoughts = TRUE,
  reasoning = "high"
)
print(res_reasoning$thoughts)

## End(Not run)
```

`openai_create_batch` *Create an OpenAI batch from an uploaded file*

Description

Creates and executes a batch based on a previously uploaded input file.

Usage

```
openai_create_batch(
  input_file_id,
  endpoint,
  completion_window = "24h",
  metadata = NULL,
  api_key = NULL
)
```

Arguments

`input_file_id`	The ID of the uploaded file (with purpose `"batch"`).
`endpoint`	The endpoint for the batch, e.g. `"/v1/chat/completions"` or `"/v1/responses"`.
`completion_window`	Time frame in which the batch should be processed. Currently only `"24h"` is supported by the API.
`metadata`	Optional named list of metadata key–value pairs.
`api_key`	Optional OpenAI API key.

Value

A list representing the Batch object.

Examples

```
## Not run:
# Requires OPENAI_API_KEY set in your environment and network access.

file_obj <- openai_upload_batch_file("batch_input.jsonl")
```

```
batch_obj <- openai_create_batch(
  input_file_id = file_obj$id,
  endpoint      = "/v1/chat/completions"
)

batch_obj$status

## End(Not run)
```

openai_download_batch_output

Download the output file for a completed batch

Description

Given a batch ID, retrieves the batch metadata, extracts the `output_file_id`, and downloads the corresponding file content to `path`.

Usage

```
openai_download_batch_output(batch_id, path, api_key = NULL)
```

Arguments

`batch_id`	The batch ID (e.g. `"batch_abc123"`).
`path`	Local file path to write the downloaded `.jsonl` output.
`api_key`	Optional OpenAI API key.

Value

Invisibly, the path to the downloaded file.

Examples

```
## Not run:
# Requires OPENAI_API_KEY and a completed batch with an output_file_id.

openai_download_batch_output("batch_abc123", "batch_output.jsonl")

# You can then parse the file
res <- parse_openai_batch_output("batch_output.jsonl")
head(res)

## End(Not run)
```

`openai_get_batch` *Retrieve an OpenAI batch*

Description

Retrieve an OpenAI batch

Usage

```
openai_get_batch(batch_id, api_key = NULL)
```

Arguments

`batch_id`	The batch ID (e.g. `"batch_abc123"`).
`api_key`	Optional OpenAI API key.

Value

A list representing the Batch object.

Examples

```
## Not run:
# Requires OPENAI_API_KEY and an existing batch ID.

batch <- openai_get_batch("batch_abc123")
batch$status

## End(Not run)
```

`openai_poll_batch_until_complete` *Poll an OpenAI batch until it completes or fails*

Description

Repeatedly calls `openai_get_batch()` until the batch reaches a terminal status (one of `"completed"`, `"failed"`, `"cancelled"`, `"expired"`), a timeout is reached, or `max_attempts` is exceeded.

Usage

```
openai_poll_batch_until_complete(
  batch_id,
  interval_seconds = 5,
  timeout_seconds = 600,
  max_attempts = Inf,
  api_key = NULL,
  verbose = TRUE
)
```

Arguments

`batch_id`	The batch ID.
`interval_seconds`	Number of seconds to wait between polling attempts.
`timeout_seconds`	Maximum total time to wait in seconds before giving up.
`max_attempts`	Maximum number of polling attempts. This is mainly useful for testing; default is `Inf`.
`api_key`	Optional OpenAI API key.
`verbose`	Logical; if `TRUE`, prints status messages to the console.

Details

This is a synchronous helper – it will block until one of the conditions above is met.

Value

The final Batch object (a list) as returned by `openai_get_batch()`.

Examples

```
## Not run:
# Requires OPENAI_API_KEY and a created batch that may still be running.

batch <- openai_create_batch("file_123", endpoint = "/v1/chat/completions")

final <- openai_poll_batch_until_complete(
  batch_id         = batch$id,
  interval_seconds = 10,
  timeout_seconds  = 3600
)

final$status

## End(Not run)
```

`openai_upload_batch_file` *Upload a JSONL batch file to OpenAI*

Description

Uploads a `.jsonl` file to the OpenAI Files API with purpose `"batch"`, which can then be used to create a Batch job.

Usage

```
openai_upload_batch_file(path, purpose = "batch", api_key = NULL)
```

Arguments

`path`	Path to the local `.jsonl` file to upload.
`purpose`	File purpose. For the Batch API this should be `"batch"`.
`api_key`	Optional OpenAI API key. Defaults to `Sys.getenv("OPENAI_API_KEY")`.

Value

A list representing the File object returned by the API, including `id`, `filename`, `bytes`, `purpose`, etc.

Examples

```
## Not run:
# Requires OPENAI_API_KEY set in your environment and network access

file_obj <- openai_upload_batch_file("batch_input.jsonl")
file_obj$id

## End(Not run)
```

`parse_anthropic_batch_output` *Parse Anthropic Message Batch output into a tibble*

Description

This function parses a `.jsonl` file produced by `anthropic_download_batch_results`. Each line in the file is a JSON object with at least:

Usage

```
parse_anthropic_batch_output(
  jsonl_path,
  tag_prefix = "<BETTER_SAMPLE>",
  tag_suffix = "</BETTER_SAMPLE>"
)
```

Arguments

`jsonl_path`	Path to a `.jsonl` file produced by `anthropic_download_batch_results`.
`tag_prefix`	Prefix for the better-sample tag. Defaults to `"<BETTER_SAMPLE>"`.
`tag_suffix`	Suffix for the better-sample tag. Defaults to `"</BETTER_SAMPLE>"`.

Details

```
{
  "custom_id": "ANTH_S01_vs_S02",
  "result": {
    "type": "succeeded" | "errored" | "canceled" | "expired",
    "message": { ... }  # when type == "succeeded"
    "error":   { ... }  # when type == "errored" (optional)
  }
}
```

Results may be returned in any order. This function uses the `custom_id` field to recover `ID1` and `ID2` and then applies the same parsing logic as `anthropic_compare_pair_live`, including extraction of extended thinking blocks (when enabled) into a separate `thoughts` column.

Value

A tibble with one row per result. The columns mirror `anthropic_compare_pair_live` with batch-specific additions:

custom_id Batch custom ID (for example `"ANTH_S01_vs_S02"`).

ID1, ID2 Sample IDs recovered from `custom_id`.

model Model name reported by Anthropic.

object_type Anthropic object type (for example `"message"`).

status_code HTTP-style status code (200 for succeeded results, `NA` otherwise).

result_type One of `"succeeded"`, `"errored"`, `"canceled"`, `"expired"`.

error_message Error message for non-succeeded results, otherwise NA.

thoughts Extended thinking text returned by Claude when reasoning is enabled (for example when `reasoning = "enabled"`), otherwise NA.

content Concatenated assistant text for succeeded results.

better_sample "SAMPLE_1", "SAMPLE_2", or NA.

better_id ID1 if SAMPLE_1 is chosen, ID2 if SAMPLE_2 is chosen, otherwise NA.

prompt_tokens Prompt / input token count (if reported).

completion_tokens Completion / output token count (if reported).

total_tokens Total token count (reported or computed upstream).

Examples

```
## Not run:
# Requires a completed Anthropic batch file
tbl <- parse_anthropic_batch_output("anthropic-results.jsonl")

## End(Not run)
```

---

parse_gemini_batch_output

Parse Gemini batch JSONL output into a tibble of pairwise results

---

Description

This reads a JSONL file created by `gemini_download_batch_results()` and converts each line into a row that mirrors the structure used for live Gemini calls, including a `thoughts` column when the batch was run with `include_thoughts = TRUE`.

Usage

```
parse_gemini_batch_output(results_path, requests_tbl)
```

Arguments

`results_path`	Path to the JSONL file produced by `gemini_download_batch_results()`.
`requests_tbl`	Tibble/data frame with at least columns `custom_id`, `ID1`, `ID2`, and (optionally) `request`. If a `request` list-column is present, it is used to detect whether `thinkingConfig.includeThoughts` was enabled for that pair.

Value

A tibble with one row per request and columns:

- `custom_id`, `ID1`, `ID2`
- `model`, `object_type`, `status_code`, `result_type`, `error_message`
- `thoughts`, `thought_signature`, `thoughts_token_count`
- `content`, `better_sample`, `better_id`
- `prompt_tokens`, `completion_tokens`, `total_tokens`

Examples

```
#' # This example assumes you have already:
# 1. Built Gemini batch requests with `build_gemini_batch_requests()`
# 2. Submitted and completed a batch job via the Gemini API
# 3. Downloaded the results using `gemini_download_batch_results()`
## Not run:
# Path to a JSONL file created by `gemini_download_batch_results()`
results_path <- "gemini_batch_results.jsonl"

# Requests table used to build the batch (must contain custom_id, ID1, ID2)
# as returned by `build_gemini_batch_requests()`
requests_tbl <- readRDS("gemini_batch_requests.rds")

# Parse batch output into a tidy tibble of pairwise results
results <- parse_gemini_batch_output(
  results_path = results_path,
  requests_tbl = requests_tbl
)

results

## End(Not run)
```

`parse_openai_batch_output` *Parse an OpenAI Batch output JSONL file*

Description

This function reads an OpenAI Batch API output file (JSONL) and extracts pairwise comparison results for use with Bradley–Terry models. It supports both the Chat Completions endpoint (where `object = "chat.completion"`) and the Responses endpoint (where `object = "response"`), including GPT-5.1 with reasoning.

Usage

```
parse_openai_batch_output(
  path,
  tag_prefix = "<BETTER_SAMPLE>",
  tag_suffix = "</BETTER_SAMPLE>"
)
```

Arguments

`path`	Path to a JSONL output file downloaded from the OpenAI Batch API.
`tag_prefix`	Character string marking the start of the better-sample tag. Defaults to `"<BETTER_SAMPLE>"`.
`tag_suffix`	Character string marking the end of the better-sample tag. Defaults to `"</BETTER_SAMPLE>"`.

Details

For each line, the function:

- extracts `custom_id` and parses `ID1` and `ID2` from the pattern `"<prefix>ID1_vs_ID2"`,
- pulls the raw LLM content containing the `<BETTER_SAMPLE>...</BETTER_SAMPLE>` tag,
- determines whether `SAMPLE_1` or `SAMPLE_2` was selected and maps that to `better_id`,
- collects model name and token usage statistics (including reasoning tokens for GPT-5.1 Responses),
- when using the Responses endpoint with reasoning, separates reasoning summaries into the `thoughts` column and visible assistant output into `content`.

The returned data frame is suitable as input for `build_bt_data`.

Value

A tibble with one row per successfully parsed comparison and columns:

custom_id The `custom_id` from the batch request.

ID1, ID2 Sample IDs inferred from `custom_id`.

model The model name reported by the API.

object_type The OpenAI response object type (e.g., `"chat.completion"` or `"response"`).

status_code HTTP-style status code from the batch output.

error_message Error message, if present; otherwise `NA`.

thoughts Reasoning / thinking summary text when available (for Responses with reasoning); otherwise `NA`.

content The raw assistant visible content string (the LLM's output), used to locate the `<BETTER_SAMPLE>` tag. For Responses with reasoning this does not include reasoning summaries, which are kept in `thoughts`.

better_sample Either `"SAMPLE_1"`, `"SAMPLE_2"`, or `NA` if the tag was not found.

better_id `ID1` if `SAMPLE_1` was chosen, `ID2` if `SAMPLE_2` was chosen, or `NA`.

prompt_tokens Prompt/input token count (if reported).

completion_tokens Completion/output token count (if reported).

total_tokens Total tokens (if reported).

prompt_cached_tokens Cached prompt tokens (if reported via `input_tokens_details$cached_tokens`); otherwise `NA`.

reasoning_tokens Reasoning tokens (if reported via `output_tokens_details$reasoning_tokens`); otherwise `NA`.

Examples

```
# Create a temporary JSONL file containing a simulated OpenAI batch result
tf <- tempfile(fileext = ".jsonl")

# A single line of JSON representing a successful Chat Completion
# custom_id implies "LIVE_" prefix, ID1="A", ID2="B"
json_line <- paste0(
  '{"custom_id": "LIVE_A_vs_B", ',
  '"response": {"status_code": 200, "body": {',
  '"object": "chat.completion", ',
  '"model": "gpt-4", ',
  '"choices": [{"message": {"content": "<BETTER_SAMPLE>SAMPLE_1</BETTER_SAMPLE>"}}], ',
  '"usage": {"prompt_tokens": 50, "completion_tokens": 10, "total_tokens": 60}}}}'
)

writeLines(json_line, tf)

# Parse the output
res <- parse_openai_batch_output(tf)

# Inspect the result
print(res$better_id)
print(res$prompt_tokens)

# Clean up
unlink(tf)
```

`randomize_pair_order` *Randomly assign samples to positions SAMPLE_1 and SAMPLE_2*

Description

This helper takes a table of paired writing samples (with columns `ID1`, `text1`, `ID2`, and `text2`) and, for each row, randomly decides whether to keep the current order or swap the two samples. The result is that approximately half of the pairs will have the original order and half will be reversed, on average.

Usage

```
randomize_pair_order(pairs, seed = NULL)
```

Arguments

`pairs`	A data frame or tibble with columns `ID1`, `text1`, `ID2`, and `text2`. Typically created by `make_pairs` (optionally followed by `sample_pairs`).
`seed`	Optional integer seed for reproducible randomization. If `NULL` (default), the current RNG state is used and not modified.

Details

This is useful for reducing position biases in LLM-based paired comparisons, while still allowing reverse-order consistency checks via `sample_reverse_pairs` and `compute_reverse_consistency`.

If you want a *deterministic* alternation of positions (for example, first pair as-is, second pair swapped, third pair as-is, and so on), use `alternate_pair_order` instead of this function.

Value

A tibble with the same columns as `pairs`, but with some rows' `ID1`/`text1` and `ID2`/`text2` swapped at random.

See Also

`alternate_pair_order` for deterministic alternating order, `sample_reverse_pairs` and `compute_reverse_consistency` for reverse-order checks.

Examples

```
data("example_writing_samples", package = "pairwiseLLM")

# Build all pairs
pairs_all <- make_pairs(example_writing_samples)

# Randomly flip the order within pairs
set.seed(123)
pairs_rand <- randomize_pair_order(pairs_all, seed = 123)

head(pairs_all[, c("ID1", "ID2")])
head(pairs_rand[, c("ID1", "ID2")])
```

`read_samples_df` *Read writing samples from a data frame*

Description

This function extracts ID and text columns from a data frame and enforces that IDs are unique. By default, it assumes the first column is the ID and the second column is the text.

Usage

```
read_samples_df(df, id_col = 1, text_col = 2)
```

Arguments

`df`	A data frame or tibble containing at least two columns.
`id_col`	Column specifying the IDs. Can be a column name (string) or a column index (integer). Defaults to 1.
`text_col`	Column specifying the writing samples (character). Can be a column name or index. Defaults to 2.

Value

A tibble with columns:

- `ID`: character ID for each sample
- `text`: character string of the writing sample

Any remaining columns in `df` are retained unchanged.

Examples

```
df <- data.frame(
  StudentID = c("S1", "S2"),
  Response = c("This is sample 1.", "This is sample 2."),
  Grade = c(8, 9),
  stringsAsFactors = FALSE
)

samples <- read_samples_df(df, id_col = "StudentID", text_col = "Response")
samples

# Using the built-in example dataset
data("example_writing_samples")
samples2 <- read_samples_df(
  example_writing_samples[, c("ID", "text")],
  id_col   = "ID",
  text_col = "text"
)
head(samples2)
```

`read_samples_dir` *Read writing samples from a directory of .txt files*

Description

This function reads all text files in a directory and uses the filename (without extension) as the sample ID and the file contents as the text.

Usage

```
read_samples_dir(path = ".", pattern = "\\.txt$")
```

Arguments

`path`	Directory containing .txt files.
`pattern`	A regular expression used to match file names. Defaults to `"\\.txt$"`, meaning all files ending in `.txt`.

Value

A tibble with columns:

- `ID`: filename without extension
- `text`: file contents as a single character string

Examples

```
## Not run:
# Suppose the working directory contains S1.txt and S2.txt
samples <- read_samples_dir(path = ".", pattern = "\\\\.txt$")
samples

## End(Not run)
```

---

register_prompt_template
Register a named prompt template

Description

This function validates a template (or reads it from a file) and stores it under a user-provided name for reuse in the current R session. Registered templates live in a package-internal registry.

Usage

```
register_prompt_template(name, template = NULL, file = NULL, overwrite = FALSE)
```

Arguments

`name`	Character scalar; name under which to store the template.
`template`	Optional character string containing a custom template. If `NULL`, the template is read from `file`, or the package default is used when both `template` and `file` are `NULL`.
`file`	Optional path to a text file containing a template. Ignored if `template` is not `NULL`.
`overwrite`	Logical; if `FALSE` (default), an error is thrown when `name` already exists in the registry.

Details

To make templates persistent across sessions, call this function in your `.Rprofile` or in a project startup script.

Any template must contain the placeholders `{TRAIT_NAME}`, `{TRAIT_DESCRIPTION}`, `{SAMPLE_1}`, and `{SAMPLE_2}`.

Value

Invisibly, the validated template string.

Examples

```
# Register a custom template for this session
custom <- "
You are an expert writing assessor for {TRAIT_NAME}.

{TRAIT_NAME} is defined as {TRAIT_DESCRIPTION}.

Which of the samples below is better on {TRAIT_NAME}?

SAMPLE 1:
{SAMPLE_1}

SAMPLE 2:
{SAMPLE_2}

<BETTER_SAMPLE>SAMPLE_1</BETTER_SAMPLE> or
<BETTER_SAMPLE>SAMPLE_2</BETTER_SAMPLE>
"

register_prompt_template("my_custom", template = custom)

# Retrieve and inspect it
tmpl <- get_prompt_template("my_custom")
cat(substr(tmpl, 1, 160), "...\n")
```

`remove_prompt_template` *Remove a registered prompt template*

Description

This function removes a template from the user registry created by `register_prompt_template`. It does not affect built-in templates stored under `inst/templates`.

Usage

```
remove_prompt_template(name, quiet = FALSE)
```

Arguments

`name`	Character scalar; name of the template to remove.
`quiet`	Logical; if `FALSE` (default), an error is thrown when `name` is not found in the user registry. When `TRUE`, the function simply returns `FALSE` in that case.

Value

Invisibly, `TRUE` if a template was removed, `FALSE` otherwise.

See Also

`register_prompt_template`, `get_prompt_template`, `list_prompt_templates`

Examples

```
# Register and then remove a template
register_prompt_template("to_delete", template = set_prompt_template())
remove_prompt_template("to_delete")
```

---

run_anthropic_batch_pipeline

Run an Anthropic batch pipeline for pairwise comparisons

Description

This high-level helper mirrors `run_openai_batch_pipeline` but targets Anthropic's *Message Batches API.* It:

Usage

```
run_anthropic_batch_pipeline(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  reasoning = c("none", "enabled"),
  include_thoughts = FALSE,
  batch_input_path = NULL,
  batch_output_path = NULL,
  poll = TRUE,
  interval_seconds = 60,
  timeout_seconds = 86400,
  api_key = Sys.getenv("ANTHROPIC_API_KEY"),
  anthropic_version = "2023-06-01",
  verbose = TRUE,
  ...
)
```

Arguments

`pairs`	Tibble or data frame with at least columns `ID1`, `text1`, `ID2`, `text2`.
`model`	Anthropic model name (for example `"claude-sonnet-4-5"`).
`trait_name`	Trait name to pass to `build_anthropic_batch_requests`.
`trait_description`	Trait description to pass to `build_anthropic_batch_requests`.
`prompt_template`	Prompt template string, typically from `set_prompt_template`.
`reasoning`	Character scalar; one of `"none"` or `"enabled"`. See details above for how `include_thoughts` influences this value and how temperature defaults are derived.
`include_thoughts`	Logical; if `TRUE`, requests extended thinking from Claude (by setting `reasoning = "enabled"` when necessary) and parses any thinking blocks into a `thoughts` column in the batch results.
`batch_input_path`	Path to write the JSON file containing the `requests` object. Defaults to a temporary file with suffix `".json"`.
`batch_output_path`	Path to write the downloaded `.jsonl` results if `poll = TRUE`. Defaults to a temporary file with suffix `".jsonl"`.
`poll`	Logical; if `TRUE`, the function will poll the batch until it reaches `processing_status = "ended"` using `anthropic_poll_batch_until_complete` and then download and parse the output. If `FALSE`, it stops after creating the batch and returns without polling or parsing.
`interval_seconds`	Polling interval in seconds (used when `poll = TRUE`).
`timeout_seconds`	Maximum total time in seconds for polling before giving up (used when `poll = TRUE`).
`api_key`	Optional Anthropic API key. Defaults to `Sys.getenv("ANTHROPIC_API_KEY")`.
`anthropic_version`	Anthropic API version string passed as the `anthropic-version` HTTP header. Defaults to `"2023-06-01"`.
`verbose`	Logical; if `TRUE`, prints progress messages while polling.
`...`	Additional Anthropic parameters forwarded to `build_anthropic_batch_requests` (for example `max_tokens`, `temperature`, `top_p`, `thinking_budget_tokens`).

Details

1. Builds Anthropic batch requests from a tibble of pairs using `build_anthropic_batch_requests`.
2. Writes a JSON file containing the `requests` object for reproducibility.
3. Creates a Message Batch via `anthropic_create_batch`.
4. Optionally polls until the batch reaches `processing_status = "ended"` using `anthropic_poll_batch_until_comple`

5. If polling is enabled, downloads the `.jsonl` result file with `anthropic_download_batch_results` and parses it via `parse_anthropic_batch_output`.

It is the Anthropic analogue of `run_openai_batch_pipeline` and returns a list with the same overall structure so that downstream code can treat the two backends uniformly.

When `include_thoughts = TRUE` and `reasoning` is left at its default of `"none"`, this function automatically upgrades `reasoning` to `"enabled"` so that Claude's extended thinking blocks are returned and parsed into the `thoughts` column by `parse_anthropic_batch_output`.

Temperature and reasoning defaults

Temperature and thinking-mode behaviour are controlled by `build_anthropic_batch_requests`:

- When `reasoning = "none"` (no extended thinking):
  - The default `temperature` is `0` (deterministic), unless you explicitly supply a `temperature` argument via `...`.
  - The default `max_tokens` is `768`, unless you override it via `max_tokens` in `...`.
- When `reasoning = "enabled"` (extended thinking enabled):
  - `temperature` **must** be `1`. If you supply a different value in `...`, `build_anthropic_batch_requests()` will throw an error.
  - By default, `max_tokens = 2048` and `thinking_budget_tokens = 1024`, subject to the constraint `1024 <= thinking_budget_tokens < max_tokens`. Violations of this constraint also produce an error.

Therefore, when you run batches without extended thinking (the usual case), the effective default is a temperature of `0`. When you explicitly use extended thinking (either by setting `reasoning = "enabled"` or by using `include_thoughts = TRUE`), Anthropic's requirement of `temperature = 1` is enforced.

Value

A list with elements (aligned with `run_openai_batch_pipeline`):

batch_input_path Path to the JSON file containing the batch `requests` object.

batch_output_path Path to the downloaded `.jsonl` results file if `poll = TRUE`, otherwise `NULL`.

file Always `NULL` for Anthropic batches (OpenAI uses a File object here). Included for structural compatibility.

batch Message Batch object; if `poll = TRUE`, this is the final batch after polling, otherwise the initial batch returned by `anthropic_create_batch`.

results Parsed tibble from `parse_anthropic_batch_output` if `poll = TRUE`, otherwise `NULL`.

Examples

```
## Not run:
# Requires ANTHROPIC_API_KEY and network access.
library(pairwiseLLM)

data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
```

```
    make_pairs() |>
    sample_pairs(n_pairs = 5, seed = 123) |>
    randomize_pair_order(seed = 456)

  td <- trait_description("overall_quality")
  tmpl <- set_prompt_template()

  # Standard batch without extended thinking
  pipeline_none <- run_anthropic_batch_pipeline(
    pairs             = pairs,
    model             = "claude-sonnet-4-5",
    trait_name        = td$name,
    trait_description = td$description,
    prompt_template   = tmpl,
    reasoning         = "none",
    include_thoughts  = FALSE,
    interval_seconds  = 60,
    timeout_seconds   = 3600,
    verbose           = TRUE
  )

  pipeline_none$batch$processing_status
  head(pipeline_none$results)

  # Batch with extended thinking and thoughts column
  pipeline_thoughts <- run_anthropic_batch_pipeline(
    pairs             = pairs,
    model             = "claude-sonnet-4-5",
    trait_name        = td$name,
    trait_description = td$description,
    prompt_template   = tmpl,
    include_thoughts  = TRUE,
    interval_seconds  = 60,
    timeout_seconds   = 3600,
    verbose           = TRUE
  )

  pipeline_thoughts$batch$processing_status
  head(pipeline_thoughts$results)

  ## End(Not run)
```

---

run_gemini_batch_pipeline
Run a Gemini batch pipeline for pairwise comparisons

Description

This helper ties together the core batch operations:

1. Build batch requests from a tibble of pairs.
2. Create a Batch job via gemini_create_batch.
3. Optionally poll for completion and download results.
4. Parse the JSONL results into a tibble via parse_gemini_batch_output.

Usage

```
run_gemini_batch_pipeline(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  thinking_level = c("low", "medium", "high"),
 batch_input_path = tempfile(pattern = "gemini-batch-input-", fileext = ".json"),
 batch_output_path = tempfile(pattern = "gemini-batch-output-", fileext = ".jsonl"),
  poll = TRUE,
  interval_seconds = 60,
  timeout_seconds = 86400,
  api_key = Sys.getenv("GEMINI_API_KEY"),
  api_version = "v1beta",
  verbose = TRUE,
  include_thoughts = FALSE,
  ...
)
```

Arguments

`pairs`	Tibble/data frame of pairs.
`model`	Gemini model name, for example "gemini-3-pro-preview".
`trait_name`	Trait name.
`trait_description`	Trait description.
`prompt_template`	Prompt template string.
`thinking_level`	One of "low", "medium", or "high".
`batch_input_path`	Path where the batch input JSON should be written.
`batch_output_path`	Path where the batch output JSONL should be written (only used if poll = TRUE).
`poll`	Logical; if TRUE, poll the batch until completion and parse results. If FALSE, only create the batch and write the input file.
`interval_seconds`	Polling interval when poll = TRUE.

`timeout_seconds`	Maximum total waiting time when `poll = TRUE`.
`api_key`	Optional Gemini API key.
`api_version`	API version string.
`verbose`	Logical; if `TRUE`, prints progress messages.
`include_thoughts`	Logical; if `TRUE`, sets `thinkingConfig.includeThoughts = TRUE` in each request, mirroring `gemini_compare_pair_live()`. Parsed results will include a `thoughts` column when visible thoughts are returned by the API (currently batch typically only exposes `thoughtSignature` + `thoughtsTokenCount`).
`...`	Additional arguments forwarded to `build_gemini_batch_requests` (for example `temperature`, `top_p`, `top_k`, `max_output_tokens`).

Details

The returned list mirrors the structure of `run_openai_batch_pipeline` and `run_anthropic_batch_pipeline`.

Value

A list with elements:

batch_input_path Path to the written batch input JSON.

batch_output_path Path to the batch output JSONL (or `NULL` when `poll = FALSE`).

file Reserved for parity with OpenAI/Anthropic; always `NULL` for Gemini inline batches.

batch The created Batch job object.

results Parsed tibble of results (or `NULL` when `poll = FALSE`).

Examples

```
# This example requires:
# - A valid Gemini API key (set in GEMINI_API_KEY)
# - Internet access
# - Billable Gemini API usage
## Not run:
# Example pairwise data
data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 5, seed = 123)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Run the full Gemini batch pipeline
res <- run_gemini_batch_pipeline(
  pairs             = pairs,
  model             = "gemini-3-pro-preview",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  thinking_level    = "low",
  poll              = TRUE,
  include_thoughts  = FALSE
)

# Parsed pairwise comparison results
res$results

# Inspect batch metadata
res$batch

# Paths to saved input/output files
res$batch_input_path
res$batch_output_path

## End(Not run)
```

---

run_openai_batch_pipeline

Run a full OpenAI batch pipeline for pairwise comparisons

Description

This helper wires together the existing pieces:

- `build_openai_batch_requests()`
- `write_openai_batch_file()`
- `openai_upload_batch_file()`
- `openai_create_batch()`
- optionally `openai_poll_batch_until_complete()`
- optionally `openai_download_batch_output()`
- optionally `parse_openai_batch_output()`

Usage

```
run_openai_batch_pipeline(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  include_thoughts = FALSE,
  include_raw = FALSE,
  endpoint = NULL,
  batch_input_path = tempfile("openai_batch_input_", fileext = ".jsonl"),
  batch_output_path = tempfile("openai_batch_output_", fileext = ".jsonl"),
  poll = TRUE,
  interval_seconds = 5,
  timeout_seconds = 600,
  max_attempts = Inf,
  metadata = NULL,
  api_key = NULL,
  ...
)
```

Arguments

`pairs`	Tibble of pairs with at least `ID1`, `text1`, `ID2`, `text2`. Typically produced by `make_pairs()`, `sample_pairs()`, and `randomize_pair_order()`.
`model`	OpenAI model name (e.g. `"gpt-4.1"`, `"gpt-5.1"`).
`trait_name`	Trait name to pass to `build_openai_batch_requests()`.
`trait_description`	Trait description to pass to `build_openai_batch_requests()`.
`prompt_template`	Prompt template string, typically from `set_prompt_template()`.
`include_thoughts`	Logical; if `TRUE` and using `endpoint = "responses"`, requests reasoning-style summaries to populate the `thoughts` column in the parsed output. When `endpoint` is not supplied, `include_thoughts = TRUE` causes the `responses` endpoint to be selected automatically.
`include_raw`	Logical; if `TRUE`, attaches the raw model response as a list-column `raw_response` in the parsed results.
`endpoint`	One of `"chat.completions"` or `"responses"`. If `NULL` (or omitted), it is chosen automatically as described above.
`batch_input_path`	Path to write the batch input `.jsonl` file. Defaults to a temporary file.
`batch_output_path`	Path to write the batch output `.jsonl` file if `poll = TRUE`. Defaults to a temporary file.
`poll`	Logical; if `TRUE`, the function will poll the batch until it reaches a terminal status using `openai_poll_batch_until_complete()` and then download and parse the output. If `FALSE`, it stops after creating the batch and returns without polling or parsing.
`interval_seconds`	Polling interval in seconds (used when `poll = TRUE`).
`timeout_seconds`	Maximum total time in seconds for polling before giving up (used when `poll = TRUE`).
`max_attempts`	Maximum number of polling attempts (primarily useful for testing).

`metadata`	Optional named list of metadata key–value pairs to pass to `openai_create_batch()`.
`api_key`	Optional OpenAI API key. Defaults to `Sys.getenv("OPENAI_API_KEY")`.
`...`	Additional arguments passed through to `build_openai_batch_requests()`, e.g. `temperature`, `top_p`, `logprobs`, `reasoning`.

Details

It is a convenience wrapper around these smaller functions and is intended for end-to-end batch runs on a set of pairwise comparisons. For more control (or testing), you can call the components directly.

When `endpoint` is not specified, it is chosen automatically:

- if `include_thoughts = TRUE`, the `"responses"` endpoint is used and, for `"gpt-5.1"`, a default reasoning effort of `"low"` is applied (unless overridden via `reasoning`).
- otherwise, `"chat.completions"` is used.

Value

A list with elements:

- `batch_input_path` – path to the input `.jsonl` file.
- `batch_output_path` – path to the output `.jsonl` file (or `NULL` if `poll = FALSE`).
- `file` – File object returned by `openai_upload_batch_file()`.
- `batch` – Batch object; if `poll = TRUE`, this is the final batch after polling, otherwise the initial batch returned by `openai_create_batch()`.
- `results` – Parsed tibble from `parse_openai_batch_output()` if `poll = TRUE`, otherwise `NULL`.

Examples

```
# The OpenAI batch pipeline requires:
# - Internet access
# - A valid OpenAI API key in OPENAI_API_KEY (or supplied via `api_key`)
# - Billable API usage
#
## Not run:
data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 2, seed = 123) |>
  randomize_pair_order(seed = 456)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Run a small batch using chat.completions
out <- run_openai_batch_pipeline(
  pairs             = pairs,
  model             = "gpt-4.1",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  endpoint          = "chat.completions",
  poll              = TRUE,
  interval_seconds  = 5,
  timeout_seconds   = 600
)

print(out$batch$status)
print(utils::head(out$results))

## End(Not run)
```

sample_pairs *Randomly sample pairs of writing samples*

Description

This function samples a subset of rows from a pairs data frame returned by `make_pairs`. You can specify either the proportion of pairs to retain (`pair_pct`), the absolute number of pairs (`n_pairs`), or both (in which case the minimum of the two is used).

Usage

```
sample_pairs(pairs, pair_pct = 1, n_pairs = NULL, seed = NULL)
```

Arguments

`pairs`	A tibble with columns `ID1`, `text1`, `ID2`, and `text2`.
`pair_pct`	Proportion of pairs to sample (between 0 and 1). Defaults to 1 (all pairs).
`n_pairs`	Optional integer specifying the maximum number of pairs to sample.
`seed`	Optional integer seed for reproducible sampling.

Value

A tibble containing the sampled rows of `pairs`.

Examples

```
samples <- tibble::tibble(
  ID   = c("S1", "S2", "S3", "S4"),
  text = paste("Sample", 1:4)
)
pairs_all <- make_pairs(samples)
```

```
# Sample 50% of all pairs
sample_pairs(pairs_all, pair_pct = 0.5, seed = 123)

# Sample exactly 3 pairs
sample_pairs(pairs_all, n_pairs = 3, seed = 123)

# Using built-in examples and sample 10% of all pairs
data("example_writing_samples")
pairs_ex <- make_pairs(example_writing_samples)
pairs_ex_sample <- sample_pairs(pairs_ex, pair_pct = 0.10, seed = 1)
nrow(pairs_ex_sample)
```

`sample_reverse_pairs` *Sample reversed versions of a subset of pairs*

Description

Given a table of pairs with columns `ID1`, `text1`, `ID2`, and `text2`, this function selects a subset of rows and returns a new tibble where the order of each selected pair is reversed.

Usage

```
sample_reverse_pairs(pairs, reverse_pct = NULL, n_reverse = NULL, seed = NULL)
```

Arguments

`pairs`	A data frame or tibble with columns `ID1`, `text1`, `ID2`, and `text2`.
`reverse_pct`	Optional proportion of rows to reverse (between 0 and 1). If `n_reverse` is also supplied, `n_reverse` takes precedence and `reverse_pct` is ignored.
`n_reverse`	Optional absolute number of rows to reverse. If supplied, this takes precedence over `reverse_pct`.
`seed`	Optional integer seed for reproducible sampling.

Value

A tibble containing the reversed pairs only (i.e., with `ID1` swapped with `ID2` and `text1` swapped with `text2`).

Examples

```
data("example_writing_samples")
pairs <- make_pairs(example_writing_samples)

# Reverse 20% of the pairs
rev20 <- sample_reverse_pairs(pairs, reverse_pct = 0.2, seed = 123)
```

set_prompt_template *Get or set a prompt template for pairwise comparisons*

Description

This function returns a default prompt template that includes placeholders for the trait name, trait description, and two writing samples. Any custom template must contain the placeholders `{TRAIT_NAME}`, `{TRAIT_DESCRIPTION}`, `{SAMPLE_1}`, and `{SAMPLE_2}`.

Usage

```
set_prompt_template(template = NULL, file = NULL)
```

Arguments

`template`	Optional character string containing a custom template. If `NULL`, a default template is returned.
`file`	Optional path to a text file containing a template. Ignored if `template` is not `NULL`.

Details

The default template is stored as a plain-text file in `inst/templates/default.txt` and loaded at run time. This makes it easy to inspect and modify the prompt text without changing the R code.

Value

A character string containing the prompt template.

Examples

```
# Get the default template shipped with the package
tmpl <- set_prompt_template()
cat(substr(tmpl, 1, 200), "...\n")

# Use a custom template defined in-line
custom <- "
You are an expert writing assessor for {TRAIT_NAME}.

{TRAIT_NAME} is defined as {TRAIT_DESCRIPTION}.

Which of the samples below is better on {TRAIT_NAME}?

SAMPLE 1:
{SAMPLE_1}

SAMPLE 2:
{SAMPLE_2}
```

```
<BETTER_SAMPLE>SAMPLE_1</BETTER_SAMPLE> or
<BETTER_SAMPLE>SAMPLE_2</BETTER_SAMPLE>
"

tmpl2 <- set_prompt_template(template = custom)
cat(substr(tmpl2, 1, 120), "...\n")
```

---

submit_anthropic_pairs_live

Live Anthropic (Claude) comparisons for a tibble of pairs

Description

This is a thin row-wise wrapper around `anthropic_compare_pair_live`. It takes a tibble of pairs (ID1 / text1 / ID2 / text2), submits each pair to the Anthropic Messages API, and binds the results into a single tibble.

Usage

```
submit_anthropic_pairs_live(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  api_key = NULL,
  anthropic_version = "2023-06-01",
  reasoning = c("none", "enabled"),
  verbose = TRUE,
  status_every = 1,
  progress = TRUE,
  include_raw = FALSE,
  include_thoughts = NULL,
  ...
)
```

Arguments

`pairs`	Tibble or data frame with at least columns `ID1`, `text1`, `ID2`, `text2`. Typically created by `make_pairs`, `sample_pairs`, and `randomize_pair_order`.
`model`	Anthropic model name (for example `"claude-sonnet-4-5"`, `"claude-haiku-4-5"`, or `"claude-opus-4-5"`).
`trait_name`	Trait name to pass to `anthropic_compare_pair_live`.
`trait_description`	Trait description to pass to `anthropic_compare_pair_live`.

`prompt_template`	Prompt template string, typically from `set_prompt_template`.
`api_key`	Optional Anthropic API key. Defaults to `Sys.getenv("ANTHROPIC_API_KEY")`.
`anthropic_version`	Anthropic API version string passed as the `anthropic-version` HTTP header. Defaults to `"2023-06-01"`.
`reasoning`	Character scalar passed to `anthropic_compare_pair_live` (one of `"none"` or `"enabled"`).
`verbose`	Logical; if `TRUE`, prints status, timing, and result summaries.
`status_every`	Integer; print status / timing for every `status_every`-th pair. Defaults to 1 (every pair). Errors are always printed.
`progress`	Logical; if `TRUE`, shows a textual progress bar.
`include_raw`	Logical; if `TRUE`, each row of the returned tibble will include a `raw_response` list-column with the parsed JSON body from Anthropic.
`include_thoughts`	Logical or `NULL`; forwarded to `anthropic_compare_pair_live`. When `TRUE` and `reasoning = "none"`, the underlying calls upgrade to extended thinking mode (`reasoning = "enabled"`), which implies `temperature = 1` and adds a `thinking` block. When `FALSE` or `NULL`, `reasoning` is used as-is.
`...`	Additional Anthropic parameters (for example `temperature`, `top_p`, `max_tokens`) passed on to `anthropic_compare_pair_live`.

Details

The output has the same columns as `anthropic_compare_pair_live`, with one row per pair, making it easy to pass into `build_bt_data` and `fit_bt_model`.

Temperature and reasoning behaviour

Temperature and extended-thinking behaviour are controlled by `anthropic_compare_pair_live`:

- When `reasoning = "none"` (no extended thinking), the default `temperature` is `0` (deterministic) unless you explicitly supply a different `temperature` via `...`.
- When `reasoning = "enabled"` (extended thinking), Anthropic requires `temperature = 1`. If you supply a different value, an error is raised by `anthropic_compare_pair_live`.

If you set `include_thoughts = TRUE` while `reasoning = "none"`, the underlying calls upgrade to `reasoning = "enabled"`, which in turn implies `temperature = 1` and adds a `thinking` block to the API request. When `include_thoughts = FALSE` (the default), and you leave `reasoning = "none"`, the effective default temperature is `0`.

Value

A tibble with one row per pair and the same columns as `anthropic_compare_pair_live`.

Examples

```
## Not run:
# Requires ANTHROPIC_API_KEY and network access.
library(pairwiseLLM)

data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 5, seed = 123) |>
  randomize_pair_order(seed = 456)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Deterministic comparisons with no extended thinking and temperature = 0
res_claude <- submit_anthropic_pairs_live(
  pairs             = pairs,
  model             = "claude-sonnet-4-5",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  reasoning         = "none",
  verbose           = TRUE,
  status_every      = 2,
  progress          = TRUE,
  include_raw       = FALSE
)

res_claude$better_id

# Comparisons with extended thinking and temperature = 1
res_claude_reason <- submit_anthropic_pairs_live(
  pairs             = pairs,
  model             = "claude-sonnet-4-5",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  reasoning         = "enabled",
  include_thoughts  = TRUE,
  verbose           = TRUE,
  status_every      = 2,
  progress          = TRUE,
  include_raw       = TRUE
)

res_claude_reason$better_id

## End(Not run)
```

submit_gemini_pairs_live

Live Google Gemini comparisons for a tibble of pairs

Description

This is a thin row-wise wrapper around `gemini_compare_pair_live()`. It takes a tibble of pairs (`ID1` / `text1` / `ID2` / `text2`), submits each pair to Gemini 3 Pro, and binds the results into a single tibble.

Usage

```
submit_gemini_pairs_live(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  api_key = NULL,
  thinking_level = c("low", "medium", "high"),
  temperature = NULL,
  top_p = NULL,
  top_k = NULL,
  max_output_tokens = NULL,
  api_version = "v1beta",
  verbose = TRUE,
  status_every = 1L,
  progress = TRUE,
  include_raw = FALSE,
  include_thoughts = FALSE,
  ...
)
```

Arguments

`pairs`	Tibble/data frame with columns `ID1`, `text1`, `ID2`, `text2`.
`model`	Gemini model name (e.g. `"gemini-3-pro-preview"`).
`trait_name`	Trait name.
`trait_description`	Trait description.
`prompt_template`	Prompt template string, typically from `set_prompt_template()`.
`api_key`	Optional Gemini API key.
`thinking_level`	Default `"low"`; see `gemini_compare_pair_live()`.

`temperature`	Optional numeric temperature; forwarded to `gemini_compare_pair_live()`. See Gemini docs; if `NULL` (default), the model uses its own default.
`top_p`	Optional numeric; forwarded to `gemini_compare_pair_live()`.
`top_k`	Optional numeric; forwarded to `gemini_compare_pair_live()`.
`max_output_tokens`	Optional integer; forwarded to `gemini_compare_pair_live()`.
`api_version`	API version; default `"v1beta"`.
`verbose`	Logical; print status/timing every `status_every` pairs.
`status_every`	Integer; how often to print status (default 1 = every pair).
`progress`	Logical; show a text progress bar.
`include_raw`	Logical; include `raw_response` list-column.
`include_thoughts`	Logical; if `TRUE`, requests explicit reasoning output from Gemini and stores it in the `thoughts` column of the result, mirroring `gemini_compare_pair_live()`.
`...`	Reserved for future extensions; passed through to `gemini_compare_pair_live()` (but `thinking_budget` is ignored there).

Details

The output has one row per pair and the same columns as `gemini_compare_pair_live()`, making it easy to pass into downstream Bradley-Terry / BTM pipelines.

Value

A tibble of results (one row per pair).

Examples

```
# Requires:
# - GEMINI_API_KEY set in your environment
# - Internet access
# - Billable Gemini API usage
## Not run:
# Example pair data
pairs <- tibble::tibble(
  ID1   = c("S01", "S03"),
  text1 = c("Text 1", "Text 3"),
  ID2   = c("S02", "S04"),
  text2 = c("Text 2", "Text 4")
)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Submit multiple live Gemini comparisons
res <- submit_gemini_pairs_live(
  pairs             = pairs,
  model             = "gemini-3-pro-preview",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  thinking_level    = "low",
  include_thoughts  = FALSE,
  progress          = TRUE
)

res
res$better_id

## End(Not run)
```

`submit_llm_pairs` *Backend-agnostic live comparisons for a tibble of pairs*

Description

`submit_llm_pairs()` is a backend-neutral wrapper around row-wise comparison for multiple pairs. It takes a tibble of pairs (`ID1`, `text1`, `ID2`, `text2`), submits each pair to the selected backend, and binds the results into a single tibble.

Usage

```
submit_llm_pairs(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  backend = c("openai", "anthropic", "gemini", "together", "ollama"),
  endpoint = c("chat.completions", "responses"),
  api_key = NULL,
  verbose = TRUE,
  status_every = 1,
  progress = TRUE,
  include_raw = FALSE,
  ...
)
```

Arguments

`pairs`	Tibble or data frame with at least columns `ID1`, `text1`, `ID2`, `text2`. Typically created by `make_pairs()`, `sample_pairs()`, and `randomize_pair_order()`.
`model`	Model identifier for the chosen backend. For `"openai"` this should be an OpenAI model name (for example `"gpt-4.1"`, `"gpt-5.1"`). For `"anthropic"` and `"gemini"`, use the corresponding provider model names (for example `"claude-4-5-sonnet"`

	or "gemini-3-pro-preview"). For "together", use Together.ai model identifiers such as "deepseek-ai/DeepSeek-R1" or "deepseek-ai/DeepSeek-V3". For "ollama", use a local model name known to the Ollama server (for example "mistral-small3.2:24b", "qwen3:32b", "gemma3:27b").
`trait_name`	Trait name to pass through to the backend-specific comparison function (for example "Overall Quality").
`trait_description`	Full-text trait description passed to the backend.
`prompt_template`	Prompt template string, typically from `set_prompt_template()`.
`backend`	Character scalar indicating which LLM provider to use. One of "openai", "anthropic", "gemini", "together", or "ollama".
`endpoint`	Character scalar specifying which endpoint family to use for backends that support multiple live APIs. For the "openai" backend this must be one of "chat.completions" or "responses", matching `submit_openai_pairs_live()`. For "anthropic", "gemini", "together", and "ollama", this is currently ignored.
`api_key`	Optional API key for the selected backend. If `NULL`, the backend-specific helper will use its own default environment variable. For "ollama", this argument is ignored (no API key is required for local inference).
`verbose`	Logical; if `TRUE`, prints status, timing, and result summaries (for backends that support it).
`status_every`	Integer; print status and timing for every `status_every`-th pair. Defaults to 1 (every pair). Errors are always printed.
`progress`	Logical; if `TRUE`, shows a textual progress bar for backends that support it.
`include_raw`	Logical; if `TRUE`, each row of the returned tibble will include a `raw_response` list-column with the parsed JSON body from the backend (for backends that support this).
`...`	Additional backend-specific parameters. For "openai" these are forwarded to `submit_openai_pairs_live()` (and ultimately `openai_compare_pair_live()`) and typically include `temperature`, `top_p`, `logprobs`, `reasoning`, and `include_thoughts`. For "anthropic" and "gemini", they are forwarded to `submit_anthropic_pairs_live()` or `submit_gemini_pairs_live()` and may include options such as `max_output_tokens`, `include_thoughts`, and provider-specific controls. For "ollama", arguments are forwarded to `submit_ollama_pairs_live()` and may include `host`, `think`, `num_ctx`, and other Ollama-specific options.

Details

At present, the following backends are implemented:

- "openai" → `submit_openai_pairs_live()`
- "anthropic" → `submit_anthropic_pairs_live()`
- "gemini" → `submit_gemini_pairs_live()`
- "together" → `together_compare_pair_live()`

- `"ollama"` → `submit_ollama_pairs_live()`

Each backend-specific helper returns a tibble with one row per pair and a compatible set of columns, including a `thoughts` column (reasoning / thinking text when available), `content` (visible assistant output), `better_sample`, `better_id`, and token usage fields.

Value

A tibble with one row per pair and the same columns as the underlying backend-specific helper for the selected backend. All backends are intended to return a compatible structure suitable for `build_bt_data()` and `fit_bt_model()`.

See Also

- `submit_openai_pairs_live()`, `submit_anthropic_pairs_live()`, `submit_gemini_pairs_live()`, `submit_together_pairs_live()`, and `submit_ollama_pairs_live()` for backend-specific implementations.
- `llm_compare_pair()` for single-pair comparisons.
- `build_bt_data()` and `fit_bt_model()` for Bradley–Terry modelling of comparison results.

Examples

```
## Not run:
# Requires an API key for the chosen cloud backend. For OpenAI, set
# OPENAI_API_KEY in your environment. Running these examples will incur
# API usage costs.
#
# For local Ollama use, an Ollama server must be running and the models
# must be pulled in advance. No API key is required for the `"ollama"`
# backend.

data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 5, seed = 123) |>
  randomize_pair_order(seed = 456)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Live comparisons for multiple pairs using the OpenAI backend
res_live <- submit_llm_pairs(
  pairs             = pairs,
  model             = "gpt-4.1",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  backend           = "openai",
  endpoint          = "chat.completions",
  temperature       = 0,
  verbose           = TRUE,
  status_every      = 2,
  progress          = TRUE,
  include_raw       = FALSE
)

res_live$better_id

# Live comparisons using a local Ollama backend

res_ollama <- submit_llm_pairs(
  pairs             = pairs,
  model             = "mistral-small3.2:24b",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  backend           = "ollama",
  verbose           = TRUE,
  status_every      = 2,
  progress          = TRUE,
  include_raw       = FALSE,
  think             = FALSE,
  num_ctx           = 8192
)

res_ollama$better_id

## End(Not run)
```

submit_ollama_pairs_live

Live Ollama comparisons for a tibble of pairs

Description

`submit_ollama_pairs_live()` is a thin row-wise wrapper around `ollama_compare_pair_live()`. It takes a tibble of pairs (`ID1` / `text1` / `ID2` / `text2`), submits each pair to a local Ollama server, and binds the results into a single tibble.

Usage

```
submit_ollama_pairs_live(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  host = getOption("pairwiseLLM.ollama_host", "http://127.0.0.1:11434"),
  verbose = TRUE,
  status_every = 1,
  progress = TRUE,
  think = FALSE,
  num_ctx = 8192L,
  include_raw = FALSE,
  ...
)
```

Arguments

`pairs`	Tibble or data frame with at least columns `ID1`, `text1`, `ID2`, `text2`. Typically created by `make_pairs()`, `sample_pairs()`, and `randomize_pair_order()`.
`model`	Ollama model name (for example `"mistral-small3.2:24b"`, `"qwen3:32b"`, `"gemma3:27b"`).
`trait_name`	Trait name to pass to `ollama_compare_pair_live()`.
`trait_description`	Trait description to pass to `ollama_compare_pair_live()`.
`prompt_template`	Prompt template string, typically from `set_prompt_template()`.
`host`	Base URL of the Ollama server. Defaults to the option `getOption("pairwiseLLM.ollama_host", "http://127.0.0.1:11434")`.
`verbose`	Logical; if `TRUE`, prints status, timing, and result summaries.
`status_every`	Integer; print status and timing for every `status_every`-th pair. Defaults to 1 (every pair). Errors are always printed.
`progress`	Logical; if `TRUE`, shows a textual progress bar.
`think`	Logical; see `ollama_compare_pair_live()` for behavior. When `TRUE` and the model name starts with `"qwen"`, the temperature is set to `0.6`; otherwise the temperature remains `0`.
`num_ctx`	Integer; context window to use via `options$num_ctx`. The default is `8192L`.
`include_raw`	Logical; if `TRUE`, each row of the returned tibble will include a `raw_response` list-column with the parsed JSON body from Ollama.
`...`	Reserved for future extensions and forwarded to `ollama_compare_pair_live()`.

Details

This helper mirrors `submit_openai_pairs_live()` but targets a local Ollama instance rather than a cloud API. It is intended to offer a similar interface and return shape, so results can be passed directly into `build_bt_data()` and `fit_bt_model()`.

Temperature and context length are controlled as follows:

- By default, `temperature = 0` for all models.
- For Qwen models (model names beginning with `"qwen"`) and `think = TRUE`, `temperature` is set to `0.6`.

- The context window is set via `options$num_ctx`, which defaults to `8192` but may be overridden via the `num_ctx` argument.

In most user-facing workflows, it is more convenient to call `submit_llm_pairs()` with `backend = "ollama"` rather than using `submit_ollama_pairs_live()` directly. The backend-neutral wrapper will route arguments to the appropriate backend helper and ensure a consistent return shape.

As with `ollama_compare_pair_live()`, this function assumes that:

- An Ollama server is running and reachable at `host`.
- The requested models have been pulled in advance (for example `ollama pull mistral-small3.2:24b`).

Value

A tibble with one row per pair and the same columns as `ollama_compare_pair_live()`, including an optional `raw_response` column when `include_raw = TRUE`.

See Also

- `ollama_compare_pair_live()` for single-pair Ollama comparisons.
- `submit_llm_pairs()` for backend-agnostic comparisons over tibbles of pairs.
- `submit_openai_pairs_live()`, `submit_anthropic_pairs_live()`, and `submit_gemini_pairs_live()` for other backend-specific implementations.

Examples

```
## Not run:
# Requires a running Ollama server and locally available models.

data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 5, seed = 123) |>
  randomize_pair_order(seed = 456)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Live comparisons for multiple pairs using a Mistral model via Ollama
res_mistral <- submit_ollama_pairs_live(
  pairs             = pairs,
  model             = "mistral-small3.2:24b",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  verbose           = TRUE,
  status_every      = 2,
  progress          = TRUE
)

res_mistral$better_id
```

```
# Qwen with thinking enabled
res_qwen_think <- submit_ollama_pairs_live(
  pairs             = pairs,
  model             = "qwen3:32b",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  think             = TRUE,
  num_ctx           = 16384,
  verbose           = FALSE,
  progress          = FALSE
)

res_qwen_think$better_id

## End(Not run)
```

---

submit_openai_pairs_live

Live OpenAI comparisons for a tibble of pairs

Description

This is a thin row-wise wrapper around `openai_compare_pair_live`. It takes a tibble of pairs (ID1 / text1 / ID2 / text2), submits each pair to the OpenAI API, and binds the results into a single tibble.

Usage

```
submit_openai_pairs_live(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  endpoint = c("chat.completions", "responses"),
  api_key = NULL,
  verbose = TRUE,
  status_every = 1,
  progress = TRUE,
  include_raw = FALSE,
  ...
)
```

Arguments

`pairs`	Tibble or data frame with at least columns `ID1`, `text1`, `ID2`, `text2`. Typically created by `make_pairs`, `sample_pairs`, and `randomize_pair_order`.

`model`	OpenAI model name (for example "gpt-4.1", "gpt-5.1").
`trait_name`	Trait name to pass to `openai_compare_pair_live`.
`trait_description`	Trait description to pass to `openai_compare_pair_live`.
`prompt_template`	Prompt template string, typically from `set_prompt_template`.
`endpoint`	Which OpenAI endpoint to target. One of `"chat.completions"` or `"responses"`.
`api_key`	Optional OpenAI API key.
`verbose`	Logical; if TRUE, prints status, timing, and result summaries.
`status_every`	Integer; print status / timing for every `status_every`-th pair. Defaults to 1 (every pair). Errors are always printed.
`progress`	Logical; if TRUE, shows a textual progress bar.
`include_raw`	Logical; if TRUE, each row of the returned tibble will include a `raw_response` list-column with the parsed JSON body from OpenAI.
`...`	Additional OpenAI parameters (temperature, top_p, logprobs, reasoning, and so on) passed on to `openai_compare_pair_live`.

Details

The output has the same columns as `openai_compare_pair_live`, with one row per pair, making it easy to pass into `build_bt_data` and `fit_bt_model`.

Value

A tibble with one row per pair and the same columns as `openai_compare_pair_live`, including a `thoughts` column for reasoning summaries (when available).

Examples

```
## Not run:
# Requires API key set and internet access

data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 5, seed = 123) |>
  randomize_pair_order(seed = 456)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Live comparisons for multiple pairs
res_live <- submit_openai_pairs_live(
  pairs             = pairs,
  model             = "gpt-4.1",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  endpoint          = "chat.completions",
  temperature       = 0,
  verbose           = TRUE,
  status_every      = 2,
  progress          = TRUE,
  include_raw       = FALSE
)

res_live$better_id

# Using gpt-5.1 with reasoning on the responses endpoint
res_live_gpt5 <- submit_openai_pairs_live(
  pairs             = pairs,
  model             = "gpt-5.1",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  endpoint          = "responses",
  reasoning         = "low",
  temperature       = NULL,
  top_p             = NULL,
  logprobs          = NULL,
  verbose           = TRUE,
  status_every      = 3,
  progress          = TRUE,
  include_raw       = TRUE
)

str(res_live_gpt5$raw_response[[1]], max.level = 2)

## End(Not run)
```

---

submit_together_pairs_live *Live Together.ai comparisons for a tibble of pairs*

Description

`submit_together_pairs_live()` is a thin row-wise wrapper around `together_compare_pair_live()`. It takes a tibble of pairs (`ID1`, `text1`, `ID2`, `text2`), submits each pair to the Together.ai Chat Completions API, and binds the results into a single tibble.

Usage

```
submit_together_pairs_live(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  api_key = NULL,
  verbose = TRUE,
  status_every = 1,
  progress = TRUE,
  include_raw = FALSE,
  ...
)
```

Arguments

`pairs`	Tibble or data frame with at least columns `ID1`, `text1`, `ID2`, `text2`. Typically created by `make_pairs()`, `sample_pairs()`, and `randomize_pair_order()`.
`model`	Together.ai model name, for example `"deepseek-ai/DeepSeek-R1"`, `"moonshotai/Kimi-K2-Instruct` `"Qwen/Qwen3-235B-A22B-Instruct-2507-tput"`, `"deepseek-ai/DeepSeek-V3"`.
`trait_name`	Trait name to pass to `together_compare_pair_live()`.
`trait_description`	Trait description to pass to `together_compare_pair_live()`.
`prompt_template`	Prompt template string, typically from `set_prompt_template()`.
`api_key`	Optional Together.ai API key. If `NULL` or empty, falls back to `TOGETHER_API_KEY` via `.together_api_key()`.
`verbose`	Logical; if `TRUE`, prints status, timing, and result summaries.
`status_every`	Integer; print status / timing for every `status_every`-th pair. Defaults to 1 (every pair). Errors are always printed.
`progress`	Logical; if `TRUE`, shows a textual progress bar.
`include_raw`	Logical; if `TRUE`, each row of the returned tibble will include a `raw_response` list-column with the parsed JSON body from Together.ai.
`...`	Additional Together.ai parameters, such as `temperature`, `top_p`, or other provider-specific options. These are forwarded to `together_compare_pair_live()`.

Details

The output has the same columns as `together_compare_pair_live()`, with one row per pair, making it easy to pass into `build_bt_data()` and `fit_bt_model()`.

Value

A tibble with one row per pair and the same columns as `together_compare_pair_live()`.

Examples

```
## Not run:
# Requires TOGETHER_API_KEY and network access.

data("example_writing_samples", package = "pairwiseLLM")
```

```
pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 5, seed = 123) |>
  randomize_pair_order(seed = 456)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Live comparisons for multiple pairs using DeepSeek-R1
res_live <- submit_together_pairs_live(
  pairs             = pairs,
  model             = "deepseek-ai/DeepSeek-R1",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl,
  temperature       = 0.6,
  verbose           = TRUE,
  status_every      = 2,
  progress          = TRUE,
  include_raw       = FALSE
)

res_live$better_id

## End(Not run)
```

summarize_bt_fit *Summarize a Bradley–Terry model fit*

Description

This helper takes the object returned by `fit_bt_model` and returns a tibble with one row per object (e.g., writing sample), including:

- `ID`: object identifier
- `theta`: estimated ability parameter
- `se`: standard error of `theta`
- `rank`: rank order of `theta` (1 = highest by default)
- `engine`: modeling engine used ("sirt" or "BradleyTerry2")
- `reliability`: MLE reliability (for **sirt**) or `NA`

Usage

```
summarize_bt_fit(fit, decreasing = TRUE, verbose = TRUE)
```

Arguments

fit	A list returned by fit_bt_model.
decreasing	Logical; should higher theta values receive lower rank numbers? If TRUE (default), the highest theta gets rank = 1.
verbose	Logical. If TRUE (default), emit warnings when coercing. If FALSE, suppress coercion warnings during ranking.

Value

A tibble with columns:

ID Object identifier.

theta Estimated ability parameter.

se Standard error of theta.

rank Rank of theta; 1 = highest (if decreasing = TRUE).

engine Modeling engine used ("sirt" or "BradleyTerry2").

reliability MLE reliability (numeric scalar) repeated on each row.

Examples

```
# Example using built-in comparison data
data("example_writing_pairs")
bt <- build_bt_data(example_writing_pairs)

fit1 <- fit_bt_model(bt, engine = "sirt")
fit2 <- fit_bt_model(bt, engine = "BradleyTerry2")

summarize_bt_fit(fit1)
summarize_bt_fit(fit2)
```

---

together_compare_pair_live

Live Together.ai comparison for a single pair of samples

Description

together_compare_pair_live() sends a single pairwise comparison prompt to the Together.ai Chat Completions API (/v1/chat/completions) and parses the result into a small tibble. It is the Together.ai analogue of openai_compare_pair_live() and uses the same prompt template and tag conventions (for example <BETTER_SAMPLE>...</BETTER_SAMPLE>).

Usage

```
together_compare_pair_live(
  ID1,
  text1,
  ID2,
  text2,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  tag_prefix = "<BETTER_SAMPLE>",
  tag_suffix = "</BETTER_SAMPLE>",
  api_key = NULL,
  include_raw = FALSE,
  ...
)
```

Arguments

`ID1`	Character ID for the first sample.
`text1`	Character string containing the first sample's text.
`ID2`	Character ID for the second sample.
`text2`	Character string containing the second sample's text.
`model`	Together.ai model name (for example `"deepseek-ai/DeepSeek-R1"`, `"moonshotai/Kimi-K2-Instruct` `"Qwen/Qwen3-235B-A22B-Instruct-2507-tput"`, `"deepseek-ai/DeepSeek-V3"`).
`trait_name`	Short label for the trait (for example "Overall Quality").
`trait_description`	Full-text definition of the trait.
`prompt_template`	Prompt template string, typically from `set_prompt_template()`.
`tag_prefix`	Prefix for the better-sample tag. Defaults to `"<BETTER_SAMPLE>"`.
`tag_suffix`	Suffix for the better-sample tag. Defaults to `"</BETTER_SAMPLE>"`.
`api_key`	Optional Together.ai API key. If `NULL` or empty, the helper falls back to the `TOGETHER_API_KEY` environment variable via `.together_api_key()`.
`include_raw`	Logical; if `TRUE`, adds a list-column `raw_response` containing the parsed JSON body returned by Together.ai (or `NULL` on parse failure). This is useful for debugging parsing problems.
`...`	Additional Together.ai parameters, typically including `temperature`, `top_p`, and provider-specific options. These are passed through to the JSON request body as top-level fields. If `temperature` is omitted, the function uses backend defaults (0.6 for `"deepseek-ai/DeepSeek-R1"`, 0 for all other models).

Details

For models such as `"deepseek-ai/DeepSeek-R1"` that emit internal reasoning wrapped in `<think>...</think>` tags, this helper will:

- Extract the `<think>...</think>` block into the `thoughts` column.
- Remove the `<think>...</think>` block from the visible `content` column, so `content` contains only the user-facing answer.

Other Together.ai models (for example `"moonshotai/Kimi-K2-Instruct-0905"`, `"Qwen/Qwen3-235B-A22B-Instruct-25`
`"deepseek-ai/DeepSeek-V3"`) are supported via the same API but may not use `<think>` tags; in those cases, `thoughts` will be `NA` and the full model output will appear in `content`.

Temperature handling:

- If `temperature` is **not** supplied in `...`, the function applies backend defaults:
  - `"deepseek-ai/DeepSeek-R1"` → `temperature = 0.6`.
  - All other models → `temperature = 0`.
- If `temperature` is included in `...`, that value is used and the defaults are not applied.

Value

A tibble with one row and columns:

custom_id ID string of the form `"LIVE_<ID1>_vs_<ID2>"`.

ID1, ID2 The sample IDs you supplied.

model Model name reported by the API.

object_type API object type, typically `"chat.completion"`.

status_code HTTP-style status code (200 if successful).

error_message Error message if something goes wrong; otherwise `NA`.

thoughts Internal reasoning text, for example `<think>...</think>` blocks from models like `"deepseek-ai/DeepSeek-R1`

content Concatenated visible assistant output (without `<think>` blocks).

better_sample "SAMPLE_1", "SAMPLE_2", or `NA`, based on the `<BETTER_SAMPLE>` tag.

better_id `ID1` if `"SAMPLE_1"` is chosen, `ID2` if `"SAMPLE_2"` is chosen, otherwise `NA`.

prompt_tokens Prompt / input token count (if reported).

completion_tokens Completion / output token count (if reported).

total_tokens Total token count (if reported).

raw_response (Optional) list-column containing the parsed JSON body.

Examples

```
## Not run:
# Requires TOGETHER_API_KEY set in your environment and network access.

data("example_writing_samples", package = "pairwiseLLM")
samples <- example_writing_samples[1:2, ]

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

# Example: DeepSeek-R1 with default temperature = 0.6 if not supplied
```

```
res_deepseek <- together_compare_pair_live(
  ID1               = samples$ID[1],
  text1             = samples$text[1],
  ID2               = samples$ID[2],
  text2             = samples$text[2],
  model             = "deepseek-ai/DeepSeek-R1",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl
)

res_deepseek$better_id
res_deepseek$thoughts

# Example: Kimi-K2 with default temperature = 0 unless overridden
res_kimi <- together_compare_pair_live(
  ID1               = samples$ID[1],
  text1             = samples$text[1],
  ID2               = samples$ID[2],
  text2             = samples$text[2],
  model             = "moonshotai/Kimi-K2-Instruct-0905",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl
)

res_kimi$better_id

## End(Not run)
```

`trait_description` *Get a trait name and description for prompts*

Description

This helper returns both a short display name and a longer description for a scoring trait. These can be inserted into the prompt template via the `{TRAIT_NAME}` and `{TRAIT_DESCRIPTION}` placeholders.

Usage

```
trait_description(
  name = c("overall_quality", "organization"),
  custom_name = NULL,
  custom_description = NULL
)
```

Arguments

`name`	Character identifier for a built-in trait. One of `"overall_quality"` or `"organization"`. Ignored if `custom_description` is supplied.
`custom_name`	Optional short label to use when supplying a `custom_description`. Defaults to "Custom trait" if `custom_description` is provided but `custom_name` is `NULL`.
`custom_description`	Optional full-text definition of a custom trait. When supplied, built-in `name` values are ignored and this text is returned instead.

Value

A list with two elements:

name Short display label for the trait (e.g., "Overall Quality").

description Full-text definition of the trait, suitable for inclusion in the prompt.

Examples

```
td <- trait_description("overall_quality")
td$name
td$description

custom_td <- trait_description(
  custom_name = "Ideas",
  custom_description = "Quality and development of ideas in the writing."
)
custom_td$name
custom_td$description
```

`write_openai_batch_file` *Write an OpenAI batch table to a JSONL file*

Description

This helper takes the output of `build_openai_batch_requests` (or a compatible table) and writes one JSON object per line, in the format expected by the OpenAI batch API.

Usage

```
write_openai_batch_file(batch_tbl, path)
```

Arguments

`batch_tbl`	A data frame or tibble, typically the result of `build_openai_batch_requests`.
`path`	File path where the JSONL file should be written.

Details

The input can either:

- Already contain a character column `jsonl` (one JSON string per row), in which case that column is used directly, or
- Contain the columns `custom_id`, `method`, `url`, and `body`, in which case the JSON strings are constructed automatically.

Value

Invisibly returns `path`.

Examples

```
## Not run:
# Requires OPENAI_API_KEY and network access.
data("example_writing_samples")
pairs_all <- make_pairs(example_writing_samples)
pairs_small <- sample_pairs(pairs_all, n_pairs = 5, seed = 1)

td <- trait_description("overall_quality")
tmpl <- set_prompt_template()

batch_tbl <- build_openai_batch_requests(
  pairs             = pairs_small,
  model             = "gpt-4.1",
  trait_name        = td$name,
  trait_description = td$description,
  prompt_template   = tmpl
)

write_openai_batch_file(batch_tbl, "batch_forward.jsonl")

## End(Not run)
```

Index