

Package ‘pairwiseLLM’

September 7, 2026

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

Version 1.3.1

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. The package additionally supports adaptive pairing workflows that iteratively select comparisons based on model uncertainty to improve ranking efficiency. Results can be modeled using frequentist or Bayesian Bradley–Terry–Luce models (Bradley & Terry, 1952 <[doi:10.2307/2334029](https://doi.org/10.2307/2334029)>; see also Caron & Doucet, 2012 <[doi:10.1080/10618600.2012.638220](https://doi.org/10.1080/10618600.2012.638220)>) or Elo rating methods (see Clark et al., 2018 <[doi:10.1371/journal.pone.0190393](https://doi.org/10.1371/journal.pone.0190393)>) to derive writing quality scores. For information on pairwise comparisons and comparative judgement, see Thurstone (1927) <[doi:10.1037/h0070288](https://doi.org/10.1037/h0070288)> and Heldsinger & Humphry (2010) <[doi:10.1007/BF03216919](https://doi.org/10.1007/BF03216919)>.

License MIT + file LICENSE

Encoding UTF-8

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

Suggests BradleyTerry2, cmdstanr, EloChoice, future, future.apply, knitr, mockery, pkgload, purrr, readr, rmarkdown, sirt, stringr, testthat (>= 3.0.0), tidyr, vctrs, withr

Additional_repositories <https://stan-dev.r-universe.dev/>

SystemRequirements CmdStan and a C++17 toolchain (optional; needed only for Bayesian and adaptive modeling)

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

Collate 'adaptive_benchmark_metrics.R' 'adaptive_btl_refit.R'
 'adaptive_candidates.R' 'adaptive_constraints.R'
 'adaptive_linking_calibration.R' 'adaptive_linking_phase_a.R'
 'adaptive_logs.R' 'adaptive_persist.R' 'adaptive_print.R'
 'adaptive_rank.R' 'adaptive_round_candidates.R'
 'adaptive_run.R' 'adaptive_schemas.R' 'adaptive_select.R'
 'adaptive_simulation_harness.R' 'adaptive_state.R'
 'adaptive_step.R' 'adaptive_trueskill.R' 'adaptive_utility.R'
 'anthropic_batch_api.R' 'anthropic_live.R' 'api_keys.R'
 'bayes_btl_mcmc.R' 'bayes_btl_mcmc_adaptive.R'
 'bayes_btl_summarize.R' 'bt_helpers.R' 'bt_model.R'
 'btl_mcmc_constraints.R' 'btl_mcmc_contracts.R'
 'btl_mcmc_fit_contracts.R' 'btl_mcmc_ingest.R'
 'btl_mcmc_model_variant.R' 'btl_mcmc_state.R'
 'btl_mcmc_stopping.R' 'btl_mcmc_summaries.R'
 'btl_mcmc_theta_summary_stubs.R' 'core_budget.R'
 'cost_estimator.R' 'custom_id.R' 'data-example_writing.R'
 'data_import.R' 'draws_sanitize.R' 'elo_model.R'
 'gemini_batch_api.R' 'gemini_live.R' 'gemini_params.R'
 'htrr2_retry.R' 'llm_backends.R' 'llm_batch.R'
 'llm_multi_batch.R' 'normalize_results.R' 'ollama_live.R'
 'openai_batch_api.R' 'openai_live.R' 'openai_params.R'
 'pairing.R' 'pairwiseLLM-package.R' 'prompt_template.R'
 'reverse_consistency.R' 'seed_helpers.R' 'together_live.R'
 'traits.R' 'utils-null-coalesce.R' 'vertex_live.R'

Config/roxygen2/version 8.1.0

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2026-09-07 06:40:11 UTC

Contents

adaptive_get_logs	4
adaptive_item_log	5
adaptive_rank	6
adaptive_rank_resume	16
adaptive_rank_run_live	17
adaptive_rank_start	23
adaptive_results_history	25

adaptive_round_log	26
adaptive_step_log	28
alternate_pair_order	29
anthropic_compare_pair_live	30
anthropic_create_batch	34
anthropic_download_batch_results	36
anthropic_get_batch	37
anthropic_poll_batch_until_complete	38
build_anthropic_batch_requests	40
build_btl_results_data	42
build_bt_data	44
build_elo_data	45
build_gemini_batch_requests	46
build_openai_batch_requests	49
build_prompt	51
check_llm_api_keys	52
check_positional_bias	53
compute_reverse_consistency	56
ensure_only_ollama_model_loaded	57
estimate_llm_pairs_cost	59
example_openai_batch_output	62
example_writing_pairs	63
example_writing_results	64
example_writing_samples	65
example_writing_samples1000	65
fit_bayes_btl_mcmc	67
fit_bt_model	69
fit_elo_model	71
gemini_compare_pair_live	73
gemini_create_batch	76
gemini_download_batch_results	78
gemini_get_batch	80
gemini_poll_batch_until_complete	81
get_prompt_template	83
list_prompt_templates	84
llm_compare_pair	85
llm_download_batch_results	89
llm_resume_multi_batches	90
llm_submit_pairs_batch	93
llm_submit_pairs_multi_batch	96
load_adaptive_session	99
make_adaptive_judge_llm	100
make_pairs	102
ollama_compare_pair_live	103
openai_compare_pair_live	107
openai_create_batch	110
openai_download_batch_output	111
openai_get_batch	112

openai_poll_batch_until_complete	113
openai_upload_batch_file	114
parse_anthropic_batch_output	115
parse_gemini_batch_output	117
parse_openai_batch_output	118
print.adaptive_state	120
print.pairwiseLLM_cost_estimate	121
randomize_pair_order	122
read_samples_df	123
read_samples_dir	124
register_prompt_template	125
remove_prompt_template	127
run_anthropic_batch_pipeline	127
run_gemini_batch_pipeline	131
run_openai_batch_pipeline	134
sample_pairs	138
sample_reverse_pairs	139
save_adaptive_session	140
set_prompt_template	141
submit_anthropic_pairs_live	142
submit_gemini_pairs_live	145
submit_llm_pairs	149
submit_ollama_pairs_live	152
submit_openai_pairs_live	155
submit_together_pairs_live	158
submit_vertex_pairs_live	162
summarize_adaptive	164
summarize_bt_fit	165
summarize_items	166
summarize_refits	168
together_compare_pair_live	170
trait_description	172
validate_session_dir	174
vertex_compare_pair_live	175
write_openai_batch_file	177

Index 180

adaptive_get_logs	<i>Retrieve canonical adaptive logs.</i>
-------------------	--

Description

Retrieve canonical adaptive logs.

Usage

adaptive_get_logs(state)

Arguments

state Adaptive state.

Details

Returns the canonical adaptive logs as currently held in memory: `step_log`, `round_log`, `item_log`, and `link_stage_log`. These correspond to step attempts, posterior refit rounds, item-level refit summaries, and per-refit linking summaries.

Value

A named list with four elements:

step_log A tibble with one row per attempted step.

round_log A tibble with one row per BTL refit round.

item_log A list of per-refit item tibbles.

link_stage_log A tibble with one row per (`refit_id`, `spoke_id`) linking summary when linking mode is active.

See Also

[adaptive_step_log\(\)](#), [adaptive_round_log\(\)](#), [adaptive_item_log\(\)](#)

Other adaptive logs: [adaptive_item_log\(\)](#), [adaptive_results_history\(\)](#), [adaptive_round_log\(\)](#), [adaptive_step_log\(\)](#), [summarize_items\(\)](#), [summarize_refits\(\)](#)

Examples

```
state <- adaptive_rank_start(c("a", "b", "c"), seed = 1)
logs <- adaptive_get_logs(state)
names(logs)
```

adaptive_item_log	<i>Adaptive item log accessor.</i>
-------------------	------------------------------------

Description

Adaptive item log accessor.

Usage

```
adaptive_item_log(state, refit_id = NULL, stack = FALSE)
```

Arguments

state Adaptive state.

refit_id Optional refit index.

stack When TRUE, stack all refits.

Details

`item_log` stores per-item posterior summaries by refit. The underlying state stores a list of refit tables; this accessor can return one refit table (default: most recent) or stack all refits into a single tibble.

Item-level summaries are domain-explicit:

- `theta_raw_*`: raw/within-set posterior summaries (EAP, fixed quantiles, SD, rank) at the current refit.
- `theta_link_*`: linked/global posterior summaries (EAP, fixed quantiles, SD, rank) after transform application.
- During linking Phase A (`phase_scope = "phase_a_set"`), `theta_link_*` is typed NA by design.
- `phase_scope`, `phase_scope_set_id`, and `in_phase_scope` indicate which item domain is currently optimized.

Value

A tibble of item-level summaries. When `stack = FALSE`, one row per item for the selected refit. When `stack = TRUE`, one row per item per refit with `refit_id` identifying source refit.

See Also

`adaptive_get_logs()`, `summarize_items()`, `adaptive_round_log()`

Other adaptive logs: `adaptive_get_logs()`, `adaptive_results_history()`, `adaptive_round_log()`, `adaptive_step_log()`, `summarize_items()`, `summarize_refits()`

Examples

```
state <- adaptive_rank_start(c("a", "b", "c"), seed = 1)
adaptive_item_log(state)
```

adaptive_rank

Run adaptive ranking end-to-end from data and model settings

Description

High-level workflow wrapper that reads sample data, constructs an LLM judge, starts or resumes adaptive state, runs `adaptive_rank_run_live()`, and returns state plus summary outputs.

Usage

```
adaptive_rank(
  data,
  id_col = 1,
  text_col = 2,
  backend = c("openai", "anthropic", "gemini", "vertex", "together", "ollama"),
  model = NULL,
  trait = "overall_quality",
  trait_name = NULL,
  trait_description = NULL,
  prompt_template = set_prompt_template(),
  endpoint = "chat.completions",
  api_key = NULL,
  include_raw = FALSE,
  judge_args = list(),
  judge_call_args = list(),
  n_steps = 1L,
  fit_fn = NULL,
  adaptive_config = NULL,
  btl_config = NULL,
  session_dir = NULL,
  persist_item_log = FALSE,
  checkpoint_every_steps = NULL,
  resume = TRUE,
  seed = 1L,
  progress = c("all", "refits", "steps", "none"),
  progress_redraw_every = 10L,
  progress_show_events = TRUE,
  progress_errors = TRUE,
  save_outputs = FALSE,
  output_file = NULL,
  judge = NULL
)
```

Arguments

data	Data source: a data frame/tibble, a file path (.csv, .tsv, .txt, .rds), or a directory containing .txt files.
id_col	ID column selector for tabular inputs. Passed to read_samples_df() . Default is 1.
text_col	Text column selector for tabular inputs. Passed to read_samples_df() . Default is 2.
backend	Backend passed to make_adaptive_judge_llm() . Choices are "openai", "anthropic", "gemini", "vertex", "together", and "ollama". Default is "openai".
model	Model passed to make_adaptive_judge_llm() . Required when judge is NULL. Default is NULL.

trait	Built-in trait key used when no custom trait is supplied. Ignored when both trait_name and trait_description are supplied. Default is "overall_quality".
trait_name	Optional custom trait display name.
trait_description	Optional custom trait definition.
prompt_template	Prompt template string. Defaults to <code>set_prompt_template()</code> .
endpoint	Endpoint family passed to <code>make_adaptive_judge_llm()</code> . Only used when backend = "openai"; choices are "chat.completions" and "responses". Default is "chat.completions". Ignored for other backends.
api_key	Optional API key passed to <code>make_adaptive_judge_llm()</code> . Default is NULL.
include_raw	Logical; forwarded to <code>make_adaptive_judge_llm()</code> . Default is FALSE.
judge_args	Named list of fixed additional arguments forwarded to <code>llm_compare_pair()</code> by the generated judge. Default is <code>list()</code> .
judge_call_args	Named list of additional arguments forwarded to the judge at run time through <code>adaptive_rank_run_live()</code> . Default is <code>list()</code> .
n_steps	Maximum number of attempted adaptive steps to execute in this call. The run may return earlier due to candidate starvation or BTL stop criteria. Attempted invalid steps also count toward this limit.
fit_fn	Optional fit override passed to <code>adaptive_rank_run_live()</code> .
adaptive_config	Optional named list of adaptive controller overrides. Unknown fields and invalid values abort with actionable errors. Supported keys (with defaults) include: <code>global_identified_reliability_min</code> Global EAP reliability threshold used to mark the run as globally identified after a refit. Default is 0.80. <code>global_identified_rank_corr_min</code> Minimum Spearman correlation between the TrueSkill rank proxy and the BTL posterior mean ranks used to mark the run as globally identified after a refit. Default is 0.90. <code>p_long_low</code> Lower bound for long-link posterior win probability gating after global identifiability when an accepted posterior refit is available. Before posterior availability, the gate falls back deterministically to TrueSkill. Default is 0.10. <code>p_long_high</code> Upper bound for long-link posterior win probability gating after global identifiability when an accepted posterior refit is available. Before posterior availability, the gate falls back deterministically to TrueSkill. Default is 0.90. <code>long_taper_mult</code> Multiplier controlling long-link quota tapering after global identifiability. Default is 0.25. <code>long_frac_floor</code> Floor fraction for long-link quota after tapering. Default is 0.02. <code>mid_bonus_frac</code> Fraction of tapered long-link quota reallocated to mid-links. Default is 0.20.

explore_taper_mult Multiplier controlling exploration tapering after global identifiability. Default is 0.50.
boundary_k Top/bottom band size used by boundary-priority routing after global identifiability. Default is 20L.
boundary_window Lookback window (steps) used by boundary-priority routing after global identifiability. Default is $\max(10L, \text{ceiling}(0.05 * N))$ where N is the number of items.
boundary_frac Fraction of local-stage steps eligible for boundary-priority routing after global identifiability. Default is 0.15.
p_star_override_margin Near-tie probability margin for star-cap override consideration. Default is 0.05.
star_override_budget_per_round Per-round budget of star-cap overrides allowed by the near-tie rule. Default is 1L.
run_mode Run mode. Choices are "within_set" (single-set), "link_one_spoke" (hub + one spoke), and "link_multi_spoke" (hub + multiple spokes). Default is "within_set". Linking modes require multi-set inputs with `set_id` and `global_item_id` in data.
hub_id Hub `set_id` for linking modes. Default is 1L.
Phase B estimation Linking modes use anchored-joint estimation with a hard-locked hub, global-shared judge parameters, concurrent spokes, and fail-fast Phase A artifact import. Historical transform/free-lock config fields are normalized only when loading older sessions or Phase A artifacts; they are not accepted as new `adaptive_config` keys.
anchored_joint_spoke_prior_scale Scale multiplier for anchored-joint spoke priors. Default is 1.0.
anchored_joint_sd_floor Lower bound applied to anchored-joint spoke prior SDs derived from Phase A artifacts. Default is 0.02.
anchored_joint_spoke_prior_fallback_sd Fallback anchored-joint spoke prior SD used when artifact-level SDs are unavailable. Default is 1.0.
link_identified_reliability_min Minimum `reliability_link_global` value on the linking-active item domain used to mark a spoke as identified. Default is 0.80.
link_stop_reliability_min Minimum `reliability_link_global` value on the linking-active item domain used to permit linking stop. Default is 0.90.
link_rank_corr_min Minimum Spearman rank correlation between TrueSkill and transformed BTL posterior mean ranks on the linking-active item domain. Default is 0.90.
max_pairs_after_stop Stop-boundary budget: when 0L, the run stops immediately after the first refit with `stop_decision = TRUE`. Values > 0L allow that many additional committed comparisons after the first stop boundary before deterministic termination. Default is 0L.
probe_panel_edges Optional explicit planned held-out probe target per spoke. When omitted in linking modes, the default scales with the largest spoke: $\max(160L, \text{ceiling}(0.12 * \text{max_spoke_items}))$. When supplied, the value must be a positive integer and becomes the canonical planned target recorded in Phase B logs.

`probe_pairs_per_refit_per_spoke` Base held-out probe collection cap per spoke per refit window while the spoke remains active in Phase B. If omitted in linking modes, the default scales with the largest spoke: $\max(4L, \text{ceiling}(0.0035 * \text{max_spoke_items}))$. The runtime uses this as a fixed per-refit cap and does not apply bootstrap or sole-blocker probe acceleration.

`probe_edges_min_for_stop` Minimum realized held-out probe edges required before Phase B stop or escalation can be evaluated. If omitted in linking modes, the default scales with the largest spoke: $\max(120L, \text{ceiling}(0.106 * \text{max_spoke_items}))$.

`probe_near_boundary_min_frac, probe_extreme_max_frac, probe_midrange_min_frac, probe_un`
Held-out probe quality gates used by Phase B stop decisions to require useful probability spread, hub/spoke item coverage, rank-bin coverage, near-boundary Brier calibration, and calibration ECE.

`probe_brier_delta_min` Minimum held-out probe Brier improvement required by the Phase B probe quality gate. Default is 0.005.

`probe_brier_max` Maximum held-out probe Brier score allowed by the Phase B stop gate. Default is 0.19.

`probe_pred_rmse_max` Maximum lagged held-out probe prediction RMSE allowed by the Phase B stop gate. Default is 0.015.

`theta_global_rmse_max` Maximum lagged transformed-score RMSE on the direct-evidence spoke scope allowed by the Phase B stop gate. Default is 0.05.

`stability_window_refits` Number of eligible refits retained in the rolling stop window. Default is 3L.

`stability_passes_required` Minimum number of passing eligible refits required within the rolling stop window. Default is 2L.

`min_refits_in_phase_b` Minimum refit index within Phase B before linking stop can be evaluated. Default is 3L.

`reliability_var_mu_epsilon` Degeneracy guard for the active-domain variance of posterior transformed-score means used in linking reliability. Default is $1e-6$.

`reliability_total_var_epsilon` Degeneracy guard for the total active-domain transformed-score variance used in linking reliability. Default is $1e-6$.

`hub_anchor_required_phase_b` Controls the normative HubEligible domain used for Phase B held-out probe construction. When TRUE (default), planned probes are drawn from the hub anchor pool; when FALSE, they are drawn from the full hub set.

`spoke_quantile_coverage_bins` Cross-set coverage control: number of quantile bins used to ensure spoke items across the score distribution receive cross-set exposure within each refit window. Default is 3L.

`spoke_quantile_coverage_min_per_bin_per_refit` Cross-set coverage control: minimum cross-set comparisons per quantile bin per refit window. Default is 1L.

`min_cross_set_pairs_per_spoke_per_refit` Only used in concurrent multi-spoke linking. Minimum cross-set committed comparisons per spoke per refit window. Default is 5L.

phase_a_mode Phase A handling for linking modes. Choices are "run" (compute within-set Phase A artifacts in-run), "import" (require user-supplied artifacts), and "mixed" (import where provided, otherwise run). Default is "run".

phase_a_required_reliability_min Minimum within-set EAP reliability required for Phase A artifacts to be considered ready (unless an imported artifact is explicitly marked quality_gate_accepted = TRUE as a trusted external quality override). Default is 0.80.

phase_a_artifacts Named list mapping set_id to an imported Phase A artifact (list) or a .rds path containing one. On the wrapper surface, this field also accepts a prior adaptive_rank() phase_a return, an out\$phase_a\$manifest, a saved session directory, or a phase_a_artifacts/ directory, and normalizes those inputs back to the canonical named-list form before run-time validation. Imported artifacts must match the current normalized BTL model_variant; all four canonical variants ("btl", "btl_e", "btl_b", "btl_e_b") are supported when the artifact and run variants match. Default is list().

Wrapper preflight validates linking mode combinations against supplied data and aborts early for incompatible run_mode/set structure combinations.

btl_config Optional named list passed to [adaptive_rank_run_live\(\)](#) to control BTL refit cadence, stopping diagnostics, and selected round-log diagnostics. Supported fields:

refit_pairs_target Minimum new committed comparisons required before the next BTL refit. Default is $\text{ceiling}(N / 2)$ clamped to [20L, 5000L]. In linking Phase A, N is the active Phase A set size. In concurrent linking Phase B, the effective target is raised when needed so each active spoke can satisfy the configured active probe floor plus the base per-refit probe cap.

model_variant BTL likelihood variant used for inference only. Choices are "btl" (no lapse, no position bias), "btl_e" (lapse), "btl_b" (position bias), and "btl_e_b" (lapse + position bias). Default is "btl_e_b".

ess_bulk_min Minimum bulk effective sample size required for diagnostics to pass. Default is $\max(400, \text{round}(20 * \sqrt{N}))$.

ess_bulk_min_near_stop Stricter bulk ESS requirement used when a run is close to stopping. Default is $\max(1000, \text{round}(50 * \sqrt{N}))$.

max_rhat Maximum allowed split-hatR. Default is 1.01.

divergences_max Maximum allowed divergent transitions. Default is 0L.

eap_reliability_min Minimum EAP reliability required to permit stopping. Default is 0.90.

stability_lag Lag (in refits) used for stability checks. Default is 2L.

theta_corr_min Minimum lagged correlation of posterior means required by stability checks. Default is 0.95.

theta_sd_rel_change_max Maximum relative change in posterior SD allowed by stability checks. Default is 0.10.

rank_spearman_min Minimum lagged Spearman rank correlation required by stability checks. Default is 0.95.

	near_tie_p_low	Lower bound of the near-tie probability band used for round logging only. Default is 0.40.
	near_tie_p_high	Upper bound of the near-tie probability band used for round logging only. Default is 0.60.
	Defaults depend on the current item count N and are merged with user overrides.	
session_dir	Optional session directory for persistence/resume. Default is NULL.	
persist_item_log	Logical; write per-refit item logs when TRUE. Default is FALSE.	
checkpoint_every_steps	Optional positive integer checkpoint cadence for ordinary live persistence. New sessions default to 100L; resumed sessions reuse the persisted cadence unless overridden.	
resume	Logical; when TRUE and session_dir contains a valid session, resume from disk; otherwise initialize a new state. Default is TRUE.	
seed	Integer seed used when creating a new adaptive state. Default is 1L.	
progress	Progress mode for <code>adaptive_rank_run_live()</code> . Choices are "all", "refits", "steps", and "none". Default is "all".	
progress_redraw_every	Redraw interval for progress output. Default is 10L.	
progress_show_events	Logical; show step events. Default is TRUE.	
progress_errors	Logical; show invalid-step events. Default is TRUE.	
save_outputs	Logical; when TRUE, save returned outputs as .rds. Default is FALSE.	
output_file	Optional output .rds path. If NULL and save_outputs = TRUE, defaults to <code>file.path(session_dir, "adaptive_outputs.rds")</code> when session_dir is set, otherwise to a temporary file.	
judge	Optional prebuilt judge function with contract <code>judge(A, B, state, ...)</code> . If supplied, model/trait/template options are ignored and this function is used directly.	

Details

This helper is designed for end users who want one entry point for adaptive runs. It supports:

- data input from a data frame, file (.csv, .tsv, .txt, .rds), or a directory of .txt files;
- model/backend configuration through `make_adaptive_judge_llm()`;
- all adaptive runtime controls exposed by `adaptive_rank_run_live()`;
- wrapper-visible phase_a reuse surfaces (manifest, artifact_dir, and per-set status) for separate-run then later-link workflows;
- resumability via session_dir and resume;
- optional saving of run outputs to an .rds artifact.

Model options: use `judge_args` (fixed) and `judge_call_args` (per-run overrides) to pass any additional `llm_compare_pair()` arguments, including provider-specific controls such as reasoning, service_tier, temperature, top_p, logprobs, include_thoughts, or host.

Adaptive options: all key controls from `adaptive_rank_run_live()` are available directly: `n_steps`, `fit_fn`, `adaptive_config`, `btl_config`, `progress`, `progress_redraw_every`, `progress_show_events`, `progress_errors`, `session_dir`, and `persist_item_log`. Use `adaptive_config` for identifiability-gated controller behavior and `btl_config` for inference/diagnostics cadence only.

Linking run modes: `run_mode = "within_set"` is the single-set workflow. `run_mode = "link_one_spoke"` and `run_mode = "link_multi_spoke"` require multi-set input (`set_id/global_item_id`), enforce hub->spoke routing defaults, and preserve Phase A artifact gating before Phase B cross-set comparisons begin. Phase B uses anchored-joint estimation with a hard-locked hub and global-shared judge parameters. Every wrapper call returns canonical phase_a outputs that can be fed back into a later linking run through `adaptive_config$phase_a_artifacts`.

Selection semantics: pair selection is TrueSkill-driven in one-pair transactional steps. Rolling anchors are refreshed from current score proxies and anchor-link routing compares exactly one anchor endpoint with one non-anchor endpoint. Long/mid-link routing excludes anchor-anchor and anchor-non-anchor pairs, while local-link routing admits same-stratum pairs and anchor-involving pairs according to stage bounds.

Wrapper-visible defaults include top-band refinement (`top_band_pct = 0.10`, `top_band_bins = 5`) with top-band size computed as `ceiling(top_band_pct * N)`.

Exposure and repeat routing: under-represented routing is degree-based (`deg <= D_min + 1`), while repeat-pressure gating is based on recent exposure (bottom-quantile `recent_deg` with quantile default `0.25`) and per-endpoint repeat slot accounting.

Inference separation: BTL refits are used for posterior inference, diagnostics, stop logic, and the long-link posterior gate after an accepted refit is available. They are not used to choose the next pair.

Resume behavior: when `resume = TRUE` and `session_dir` already contains adaptive artifacts, failed session loads abort with an actionable error instead of starting a fresh run silently.

Value

A list with:

state Final `adaptive_state`.

summary Run-level summary from `summarize_adaptive()`.

refits Per-refit summary from `summarize_refits()`.

items Item summary from `summarize_items()`, sorted by a usable canonical rank column (`rank_link` for linking runs when available, otherwise `rank_raw`).

logs Canonical logs from `adaptive_get_logs()`.

phase_a Canonical wrapper-visible Phase A discovery surface with per-set status, `artifact_dir`, `artifact_paths`, and a reusable manifest that can be fed back into a later linking run via `adaptive_config$phase_a_artifacts`.

output_file Saved output path when `save_outputs = TRUE`, otherwise `NULL`.

See Also

`make_adaptive_judge_llm()`, `adaptive_rank_run_live()`, `adaptive_rank_start()`, `adaptive_rank_resume()`, `llm_compare_pair()`

Other adaptive ranking: `adaptive_rank_resume()`, `adaptive_rank_run_live()`, `adaptive_rank_start()`, `make_adaptive_judge_llm()`, `summarize_adaptive()`

Examples

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

out <- adaptive_rank(
  data = example_writing_samples[1:8, c("ID", "text", "quality_score")],
  id_col = "ID",
  text_col = "text",
  model = "gpt-5.1",
  judge = function(A, B, state, ...) {
    y <- as.integer(A$quality_score[[1]] >= B$quality_score[[1]])
    list(is_valid = TRUE, Y = y, invalid_reason = NA_character_)
  },
  n_steps = 4,
  progress = "none"
)

out$summary
head(out$logs$step_log)

## Not run:
# Live run with OpenAI gpt-5.1 + lower-cost Flex processing.
live <- adaptive_rank(
  data = example_writing_samples[1:12, c("ID", "text")],
  backend = "openai",
  model = "gpt-5.1",
  endpoint = "responses",
  judge_args = list(
    reasoning = "low",
    service_tier = "flex",
    include_thoughts = FALSE
  ),
  btl_config = list(
    refit_pairs_target = 20L,
    ess_bulk_min = 500,
    eap_reliability_min = 0.92
  ),
  adaptive_config = list(
    explore_taper_mult = 0.40,
    star_override_budget_per_round = 2L
  ),
  n_steps = 120,
  session_dir = file.path(tempdir(), "adaptive-live"),
  persist_item_log = TRUE,
  resume = TRUE,
```

```

    progress = "all",
    save_outputs = TRUE
  )

  print(live$state)
  live$summary

  # Wrapper-driven linking workflow (hub + one spoke).
  linking_samples <- example_writing_samples[1:12, c("ID", "text")]
  linking_samples$set_id <- rep(c(1L, 2L), each = 6L)
  linking_samples$global_item_id <- paste0("g_", linking_samples$ID)

  link_out <- adaptive_rank(
    data = linking_samples,
    id_col = "ID",
    text_col = "text",
    backend = "openai",
    model = "gpt-5.1",
    adaptive_config = list(
      run_mode = "link_one_spoke",
      hub_id = 1L,
      phase_a_mode = "run",
      probe_panel_edges = 48L,
      hub_anchor_required_phase_b = TRUE,
      max_pairs_after_stop = 0L
    ),
    n_steps = 200,
    session_dir = file.path(tempdir(), "adaptive-link"),
    resume = TRUE,
    progress = "refits"
  )

  # Later linking from prior wrapper outputs:
  # hub_run <- adaptive_rank(
  #   data = linking_samples[linking_samples$set_id == 1L, c("ID", "text")],
  #   backend = "openai",
  #   model = "gpt-5.1",
  #   n_steps = 120,
  #   progress = "none"
  # )
  # spoke_run <- adaptive_rank(
  #   data = linking_samples[linking_samples$set_id == 2L, c("ID", "text")],
  #   backend = "openai",
  #   model = "gpt-5.1",
  #   n_steps = 120,
  #   progress = "none"
  # )
  #
  # link_out <- adaptive_rank(
  #   data = linking_samples,
  #   id_col = "ID",
  #   text_col = "text",
  #   backend = "openai",

```

```

# model = "gpt-5.1",
# adaptive_config = list(
#   run_mode = "link_one_spoke",
#   hub_id = 1L,
#   phase_a_mode = "import",
#   phase_a_artifacts = list(
#     `1` = hub_run$phase_a$manifest,
#     `2` = spoke_run$phase_a$artifact_dir
#   )
# ),
# n_steps = 200,
# progress = "refits"
# )

names(link_out$logs)

## End(Not run)

```

adaptive_rank_resume *Adaptive ranking resume*

Description

Resume a previously persisted adaptive pairing session.

Usage

```
adaptive_rank_resume(session_dir, ...)
```

Arguments

<code>session_dir</code>	Directory containing session artifacts.
<code>...</code>	Reserved for future extensions; currently unused.

Details

This is a thin wrapper around [load_adaptive_session\(\)](#) and performs schema and log-shape checks during load. Returned state preserves canonical `step_log`, `round_log`, and `item_log` contents used for adaptive auditability.

Value

An `adaptive_state` object restored from disk.

See Also

[adaptive_rank_start\(\)](#), [adaptive_rank_run_live\(\)](#), [save_adaptive_session\(\)](#), [load_adaptive_session\(\)](#)
 Other adaptive ranking: [adaptive_rank\(\)](#), [adaptive_rank_run_live\(\)](#), [adaptive_rank_start\(\)](#),
[make_adaptive_judge_llm\(\)](#), [summarize_adaptive\(\)](#)

Examples

```

dir <- tempfile("pwl1m-session-")
state <- adaptive_rank_start(c("a", "b", "c"), seed = 3)
save_adaptive_session(state, dir, overwrite = TRUE)
restored <- adaptive_rank_resume(dir)
summarize_adaptive(restored)

```

adaptive_rank_run_live

Adaptive ranking live runner

Description

Execute stepwise adaptive ranking with a user-supplied judge.

Usage

```

adaptive_rank_run_live(
  state,
  judge,
  n_steps = 1L,
  fit_fn = NULL,
  adaptive_config = NULL,
  btl_config = NULL,
  session_dir = NULL,
  persist_item_log = NULL,
  checkpoint_every_steps = NULL,
  progress = c("all", "refits", "steps", "none"),
  progress_redraw_every = 10L,
  progress_show_events = TRUE,
  progress_errors = TRUE,
  ...
)

```

Arguments

state	An adaptive state object created by adaptive_rank_start() .
judge	A function called as <code>judge(A, B, state, ...)</code> that returns a list with <code>is_valid = TRUE</code> and <code>Y</code> in <code>0/1</code> , or <code>is_valid = FALSE</code> with <code>invalid_reason</code> .
n_steps	Maximum number of attempted adaptive steps to execute in this call. The run may terminate earlier if candidate starvation is encountered or if BTL stopping criteria are met at a refit. Each attempted step counts toward this budget, including invalid judge responses.
fit_fn	Optional BTL fit function for deterministic testing; defaults to <code>default_btl_fit_fn()</code> when a refit is due.

adaptive_config

Optional named list overriding adaptive controller behavior. Unknown fields and invalid values abort with an actionable error. See `adaptive_rank()` for the full list of supported keys, detailed semantics, and defaults.

btl_config

Optional named list overriding BTL refit cadence, stopping thresholds, and selected round-log diagnostics. Supported fields:

`refit_pairs_target` Minimum new committed comparisons required before the next BTL refit. Default is $\text{ceiling}(N / 2)$ clamped to $[20L, 5000L]$ (Phase A linking uses the active set size).

`model_variant` BTL MCMC variant: "btl", "btl_e", "btl_b", or "btl_e_b". Default is "btl_e_b".

`ess_bulk_min` Minimum bulk ESS required for diagnostics to pass. Default is $\max(400, \text{round}(20 * \sqrt{N}))$.

`ess_bulk_min_near_stop` Stricter ESS requirement when a run is close to stopping. Default is $\max(1000, \text{round}(50 * \sqrt{N}))$.

`max_rhat` Maximum allowed split- $\hat{R}$ diagnostic value. Default is 1.01.

`divergences_max` Maximum allowed divergent transitions. Default is 0L.

`eap_reliability_min` Minimum EAP reliability to allow stopping. Default is 0.90.

`stability_lag` Lag (in refits) used for stability checks. Default is 2L.

`theta_corr_min` Minimum lagged correlation of posterior means. Default is 0.95.

`theta_sd_rel_change_max` Maximum relative change in posterior SD allowed by stability checks. Default is 0.10.

`rank_spearman_min` Minimum lagged Spearman rank correlation. Default is 0.95.

`near_tie_p_low`, `near_tie_p_high` Probability band used only for near-tie diagnostics in round logging (not used for stopping decisions). Defaults are 0.40 and 0.60.

`deferred_audit_max_draws` Maximum posterior draws used for report-only deferred round-log audit metrics such as near-tie and credible-interval width summaries. This does not affect CmdStan diagnostics or stop gates. Default is 400L; use Inf to use all draws.

`phase_b_refit_parallel`, `phase_b_refit_workers` Opt-in parallel execution for spoke-separable Phase B post-refit updates after the main BTL refit. Uses forked local workers and is only supported on Unix-like platforms. Defaults are FALSE and 1L.

Defaults are resolved from the current item count N , then merged with user overrides.

session_dir

Optional directory for saving session artifacts. If NULL, uses `state$config$session_dir`. Default is NULL.

persist_item_log

Logical; when TRUE, write per-refit item logs to disk. If NULL, uses `state$config$persist_item_log`. Default is NULL.

checkpoint_every_steps

Optional positive integer checkpoint cadence for ordinary live persistence. If NULL, uses the persisted state value when present, otherwise defaults to 100L.

progress Progress output: "all", "refits", "steps", or "none". Default is "all".
 progress_redraw_every Redraw progress bar every N steps. Default is 10L.
 progress_show_events Logical; when TRUE, print notable step events. Default is TRUE.
 progress_errors Logical; when TRUE, include invalid-step events. Default is TRUE.
 ... Additional arguments passed through to judge().

Details

Each iteration attempts at most one pair evaluation ("one-pair step"), then applies transactional updates if and only if the judge response is valid. Invalid responses produce a logged step with `pair_id = NA` and must not update committed-comparison state.

Within-set routing is TrueSkill-based with utility

$$U_0 = p_{ij}(1 - p_{ij})$$

. After an accepted posterior refit is available, the long-link gate uses the BTL posterior win probability for candidate eligibility; before that it falls back deterministically to TrueSkill. In linking Phase B, anchor/strata routing uses linking-global scores built from Phase A summaries and the accepted anchored-joint state. Linking Phase B routing ranks eligible cross-set candidates by ridge-stabilized D-optimal log-det information gain on the active linking parameter block using order-averaged Model D probabilities. Linking inference parameters remain inference-only (diagnostics and stopping) and are not direct pair-selection objectives. Phase B uses a hard-lock hub-fixed fit and a deterministic accepted state before the first linking refit. Exploration/exploitation routing and fallback handling are recorded in `step_log`.

Round scheduling uses stage-specific admissibility:

- rolling-anchor links compare one anchor and one non-anchor endpoint;
- long/mid links exclude anchor endpoints and enforce stratum-distance bounds;
- local-link routing admits same-stratum pairs and anchor-involving pairs within local stage bounds.

Exposure and repeat handling are soft, stage-local constraints: under-represented exploration uses degree set `deg <= D_min + 1`, while repeat-pressure gating uses bottom-quantile `recent_deg` (default quantile 0.25) and per-endpoint repeat-slot accounting against `repeat_in_round_budget`.

Top-band defaults for stratum construction are `top_band_pct = 0.10` and `top_band_bins = 5`, with top-band size `ceiling(top_band_pct * N)`.

Bayesian BTL refits are triggered on step-based cadence and evaluated with diagnostics gates (including ESS thresholds), reliability, and lagged stability criteria. Refit-level outcomes are appended to `round_log`; per-item posterior summaries are appended to `item_log`. Controller behavior can change after refits via identifiability-gated settings in `adaptive_config`; those controls affect pair routing and quotas, while BTL remains inference-only. If `adaptive_config$max_pairs_after_stop > 0`, the run records a stop boundary at the first refit with `stop_decision = TRUE` and allows at most that many additional committed comparisons before deterministic termination. Round logs record `max_pairs_after_stop` and `pairs_committed_after_stop`.

Value

An updated `adaptive_state`. The returned state includes appended `step_log` rows for attempted steps and, when refits occur, appended `round_log` and `item_log` entries.

See Also

`adaptive_rank_start()`, `adaptive_rank_resume()`, `adaptive_step_log()`, `adaptive_round_log()`, `adaptive_item_log()`

Other adaptive ranking: `adaptive_rank()`, `adaptive_rank_resume()`, `adaptive_rank_start()`, `make_adaptive_judge_llm()`, `summarize_adaptive()`

Examples

```
# -----
# Offline end-to-end workflow (fast, deterministic, CRAN-safe)
# -----
data("example_writing_samples", package = "pairwiseLLM")

items <- dplyr::rename(
  example_writing_samples[1:8, c("ID", "text", "quality_score")],
  item_id = ID
)

# Use the package defaults for trait and prompt template.
trait <- trait_description("overall_quality")
prompt_template <- set_prompt_template()

# Deterministic local judge based on fixture quality scores.
sim_judge <- function(A, B, state, ...) {
  y <- as.integer(A$quality_score[[1]] >= B$quality_score[[1]])
  list(is_valid = TRUE, Y = y, invalid_reason = NA_character_)
}

session_dir <- tempfile("pwillm-adaptive-session-")

state <- adaptive_rank_start(
  items = items,
  seed = 42,
  adaptive_config = list(
    global_identified_reliability_min = 0.85,
    star_override_budget_per_round = 2L
  ),
  session_dir = session_dir,
  persist_item_log = TRUE
)

state <- adaptive_rank_run_live(
  state = state,
  judge = sim_judge,
  n_steps = 6,
  btl_config = list(
```

```

    # Keep examples lightweight while showing custom stop config inputs.
    refit_pairs_target = 50L,
    ess_bulk_min = 400,
    eap_reliability_min = 0.90
  ),
  adaptive_config = list(
    explore_taper_mult = 0.40,
    boundary_frac = 0.20
  ),
  progress = "steps",
  progress_redraw_every = 1L,
  progress_show_events = TRUE,
  progress_errors = TRUE
)

# Print and inspect run outputs.
print(state)
run_summary <- summarize_adaptive(state)
step_view <- adaptive_step_log(state)
logs <- adaptive_get_logs(state)

run_summary
head(step_view)
names(logs)

# Resume from disk and continue.
resumed <- adaptive_rank_resume(session_dir)
resumed <- adaptive_rank_run_live(
  state = resumed,
  judge = sim_judge,
  n_steps = 4,
  progress = "none"
)
summarize_adaptive(resumed)

# -----
# Live OpenAI workflow via backend-agnostic llm_compare_pair()
# -----
## Not run:
# Requires network + OPENAI_API_KEY. This incurs API cost.
# check_llm_api_keys() is a quick preflight.
check_llm_api_keys()

data("example_writing_samples", package = "pairwiseLLM")
live_items <- dplyr::rename(
  example_writing_samples[1:12, c("ID", "text")],
  item_id = ID
)

# Default trait/template setup used by the backend-agnostic runner.
trait <- trait_description("overall_quality")
prompt_template <- set_prompt_template()

```

```

live_session_dir <- file.path(tempdir(), "pwllm-adaptive-openai")

judge_openai <- function(A, B, state, ...) {
  res <- llm_compare_pair(
    ID1 = A$item_id[[1]],
    text1 = A$text[[1]],
    ID2 = B$item_id[[1]],
    text2 = B$text[[1]],
    model = "gpt-5.1",
    trait_name = trait$name,
    trait_description = trait$description,
    prompt_template = prompt_template,
    backend = "openai",
    endpoint = "responses",
    reasoning = "low",
    service_tier = "flex",
    include_thoughts = FALSE,
    temperature = NULL,
    top_p = NULL,
    logprobs = NULL
  )

  better_id <- res$better_id[[1]]
  ok_ids <- c(A$item_id[[1]], B$item_id[[1]])
  if (is.na(better_id) || !(better_id %in% ok_ids)) {
    return(list(
      is_valid = FALSE,
      Y = NA_integer_,
      invalid_reason = "model_response_invalid"
    ))
  }

  list(
    is_valid = TRUE,
    Y = as.integer(identical(better_id, A$item_id[[1]])),
    invalid_reason = NA_character_
  )
}

state_live <- adaptive_rank_start(
  items = live_items,
  seed = 2026,
  session_dir = live_session_dir,
  persist_item_log = TRUE
)

state_live <- adaptive_rank_run_live(
  state = state_live,
  judge = judge_openai,
  n_steps = 120L,
  btl_config = list(
    refit_pairs_target = 20L,
    ess_bulk_min = 500,

```

```

    ess_bulk_min_near_stop = 1200,
    max_rhat = 1.01,
    divergences_max = 0L,
    eap_reliability_min = 0.92,
    stability_lag = 2L,
    theta_corr_min = 0.97,
    theta_sd_rel_change_max = 0.08,
    rank_spearman_min = 0.97
  ),
  progress = "all",
  progress_redraw_every = 1L,
  progress_show_events = TRUE,
  progress_errors = TRUE
)

# Reporting outputs for end users.
print(state_live)
run_summary <- summarize_adaptive(state_live)
refit_summary <- summarize_refits(state_live)
item_summary <- summarize_items(state_live)
logs <- adaptive_get_logs(state_live)

# Store outputs for audit/reproducibility.
saveRDS(
  list(
    run_summary = run_summary,
    refit_summary = refit_summary,
    item_summary = item_summary,
    logs = logs
  ),
  file.path(live_session_dir, "adaptive_outputs.rds")
)

# Resume from stored state and continue sampling.
state_live <- adaptive_rank_resume(live_session_dir)
state_live <- adaptive_rank_run_live(
  state = state_live,
  judge = judge_openai,
  n_steps = 40L,
  progress = "refits"
)
print(summarize_adaptive(state_live))

## End(Not run)

```

adaptive_rank_start *Adaptive ranking*

Description

Initialize an adaptive ranking session and canonical state object.

Usage

```
adaptive_rank_start(
  items,
  seed = 1L,
  session_dir = NULL,
  persist_item_log = FALSE,
  ...,
  adaptive_config = NULL,
  checkpoint_every_steps = NULL
)
```

Arguments

items	A vector or data frame of items. Data frames must include an <code>item_id</code> column (or <code>id/ID</code>). For linking run modes, items must also include integer <code>set_id</code> values and globally unique <code>global_item_id</code> values. Item IDs may be character; internal logs use integer indices derived from these IDs.
seed	Integer seed used for deterministic warm-start shuffling and selection randomness. Default is 1L.
session_dir	Optional directory for saving session artifacts. Default is NULL.
persist_item_log	Logical; when TRUE, write per-refit item logs to disk. Default is FALSE.
...	Internal/testing only. Supply <code>now_fn</code> to override the clock used for timestamps.
adaptive_config	Optional named list of adaptive controller overrides. Unknown fields and invalid values abort with an actionable error. See adaptive_rank() for the full list of supported keys, detailed semantics, and defaults.
checkpoint_every_steps	Optional positive integer checkpoint cadence for ordinary live persistence. If NULL, defaults to 100L.

Details

This function creates the stepwise controller state and seeds all canonical logs used in the adaptive pairing workflow. Warm start pair construction follows the shuffled chain design, which guarantees a connected comparison graph after $N - 1$ committed comparisons.

Pair selection in this framework is stepwise and uncertainty-aware. Within-set routing uses TrueSkill base utility

$$U_0 = p_{ij}(1 - p_{ij})$$

where p_{ij} is the current TrueSkill win probability for pair $\{i, j\}$. In linking Phase B, anchor/strata routing uses a linking-global score derived from Phase A raw summaries and the accepted Phase B linking state. In linking Phase B, eligible cross-set candidates are ranked by ridge-stabilized D-optimal log-det information gain on the active linking parameter block using order-averaged Model D probabilities. In the spoke free block with the hub fixed. Linking inference parameters are used for inference/diagnostics/stopping, not as direct selection objectives. Phase B uses pooled within-set Phase A judge-parameter estimates, using the configured BTL model variant, as the accepted shared

source for fixed beta/epsilon constants. Bayesian BTL posterior draws are not used as general pair-selection objectives; within-set pairing remains TrueSkill-routed, with accepted posterior refits contributing only to the long-link probability gate. Linking Phase B refits use Bayesian posterior estimation and posterior summaries/diagnostics are logged per spoke at each linking refit.

The returned state contains canonical logs:

- `step_log`: one row per attempted step,
- `round_log`: one row per posterior refit,
- `item_log`: per-item posterior summaries by refit.

If `session_dir` is supplied, the initialized state is persisted immediately using `save_adaptive_session()`.

Value

An adaptive state object containing `step_log`, `round_log`, and `item_log`. The object includes class `"adaptive_state"`, item ID mappings, TrueSkill state, warm-start queue, refit metadata, and runtime configuration.

See Also

`adaptive_rank_run_live()`, `adaptive_rank_resume()`, `adaptive_step_log()`, `adaptive_round_log()`, `adaptive_item_log()`

Other adaptive ranking: `adaptive_rank()`, `adaptive_rank_resume()`, `adaptive_rank_run_live()`, `make_adaptive_judge_llm()`, `summarize_adaptive()`

Examples

```
state <- adaptive_rank_start(c("a", "b", "c"), seed = 11)
summarize_adaptive(state)
```

`adaptive_results_history`

Adaptive results history in `build_bt_data()` format.

Description

Adaptive results history in `build_bt_data()` format.

Usage

```
adaptive_results_history(state, committed_only = TRUE)
```

Arguments

`state` Adaptive state.
`committed_only` Use only committed comparisons.

Details

Converts adaptive step outcomes into the three-column format used by `build_bt_data()` (`object1`, `object2`, `result`). With `committed_only = TRUE`, only committed steps (`pair_id` not missing) are retained. This preserves the transactional invariant that invalid steps do not contribute to inferred comparisons.

Value

A tibble with columns:

object1 Character item id shown in position A.

object2 Character item id shown in position B.

result Numeric outcome in $\{0, 1\}$ where 1 means `object1` wins.

See Also

`build_bt_data()`, `adaptive_step_log()`

Other adaptive logs: `adaptive_get_logs()`, `adaptive_item_log()`, `adaptive_round_log()`, `adaptive_step_log()`, `summarize_items()`, `summarize_refits()`

Examples

```
state <- adaptive_rank_start(c("a", "b", "c"), seed = 1)
adaptive_results_history(state)
```

<code>adaptive_round_log</code>	<i>Adaptive round log accessor.</i>
---------------------------------	-------------------------------------

Description

Adaptive round log accessor.

Usage

```
adaptive_round_log(state, reconstruct_deferred = FALSE)
```

Arguments

`state` Adaptive state.

`reconstruct_deferred`

Logical; when TRUE, reconstruct deferred audit-only posterior summaries from stored refit payloads when available. By default, returns the canonical stored `round_log` without reconstruction.

Details

round_log is the canonical per-refit audit log for the adaptive pairing workflow. Each row summarizes one Bayesian BTL refit and includes diagnostics, reliability, and stopping-gate fields used to justify stop decisions.

Core columns:

- Refit identity/state: refit_id, round_id_at_refit, step_id_at_refit, timestamp, model_variant, n_items, total_pairs_done, new_pairs_since_last_refit, n_unique_pairs_seen.
- Candidate health: proposed_pairs_mode, starve_rate_since_last_refit, fallback_rate_since_last_refit, fallback_used_mode, starvation_reason_mode.
- Identifiability/quota adaptation: global_identified, global_identified_reliability_min, global_identified_rank_corr_min, long_quota_raw, long_quota_effective, long_quota_removed, realloc_to_mid, realloc_to_local, phase_scope, phase_scope_set_id, phase_scope_n_items.
- Coverage/imbalance: mean_degree, min_degree, mean_degree_scope, min_degree_scope, pos_balance_sd, star_cap_rejects_since_last_refit, star_cap_reject_rate_since_last_refit, recent_deg_median_since_last_refit, recent_deg_max_since_last_refit.
- Posterior parameter summaries: epsilon_mean/percentiles and b_mean/percentiles.
- Audit diagnostics: ts_sigma_mean, ts_sigma_max, ts_degree_sigma_corr, ts_btl_theta_corr, ts_btl_rank_spearman, ci95_theta_width_*, near_tie_adj_frac, near_tie_adj_count, p_adj_median, cov_trace_theta, cov_logdet_diag_theta, post_sd_theta_p10, post_sd_theta_p50, post_sd_theta_p90, top20_boundary_entropy_*, nn_diff_sd_*
- Stopping diagnostics: diagnostics_pass, diagnostics_divergences_pass, diagnostics_rhat_pass, diagnostics_ess_pass, divergences, divergences_max_allowed, max_rhat, max_rhat_allowed, min_ess_bulk, ess_bulk_required, near_stop_active, reliability_EAP, reliability_EAP_scope, eap_reliability_min, eap_pass, eap_pass_scope, theta_sd_eap, theta_sd_eap_scope, rho_theta, rho_theta_scope, lag_eligible, lag_eligible_scope, theta_corr_min, theta_corr_pass, theta_corr_pass_scope, delta_sd_theta, delta_sd_theta_scope, theta_sd_rel_change_max, delta_sd_theta_pass, delta_sd_theta_pass_scope, rho_rank, rho_rank_scope, rank_spearman_min, rho_rank_pass, rho_rank_pass_scope.
- Refit execution metadata: mcmc_chains, mcmc_parallel_chains, mcmc_core_fraction, mcmc_cores_detected_physical, mcmc_cores_detected_logical, mcmc_threads_per_chain, mcmc_cmdstanr_version.
- Stop output: stop_decision, stop_reason, max_pairs_after_stop, pairs_committed_after_stop.

Value

A tibble with one row per completed posterior refit round.

See Also

[adaptive_get_logs\(\)](#), [summarize_refits\(\)](#), [adaptive_rank_run_live\(\)](#)

Other adaptive logs: [adaptive_get_logs\(\)](#), [adaptive_item_log\(\)](#), [adaptive_results_history\(\)](#), [adaptive_step_log\(\)](#), [summarize_items\(\)](#), [summarize_refits\(\)](#)

Examples

```
state <- adaptive_rank_start(c("a", "b", "c"), seed = 1)
adaptive_round_log(state)
```

adaptive_step_log	<i>Adaptive step log accessor.</i>
-------------------	------------------------------------

Description

Adaptive step log accessor.

Usage

```
adaptive_step_log(state)
```

Arguments

state	Adaptive state.
-------	-----------------

Details

step_log is the canonical per-step audit log for the adaptive workflow. It records candidate pipeline outcomes, selected pair/order, and commit status. A step with invalid judge response keeps committed fields as NA and must not update model state. The selected endpoints i/j are the pre-orientation item indices, while A/B are the displayed / judged item indices after order assignment. Y is defined relative to displayed order: Y = 1 means A wins and Y = 0 means B wins. For cross-run audit and reuse, prefer the stable *_id columns plus unordered_key/ordered_key rather than transient integer item positions from the live state. Judge provenance, token counts, and raw_response_json are canonical step-log outputs, with raw_response_json stored as serialized character data rather than a list-column.

Core columns:

- Identity/outcome: step_id, timestamp, pair_id, i, j, i_id, j_id, A, B, A_id, B_id, unordered_key, ordered_key, Y, status.
- Judge audit: judge_backend, judge_model, judge_endpoint, judge_valid, judge_invalid_reason, llm_status_code, llm_error_message, llm_custom_id, prompt_tokens, completion_tokens, total_tokens, raw_response_json.
- Routing/scheduling: round_id, round_stage, pair_type, stage_committed_so_far, stage_quota.
- Exposure/strata: used_in_round_i, used_in_round_j, is_anchor_i, is_anchor_j, stratum_i, stratum_j, dist_stratum.
- Candidate health: is_explore_step, explore_mode, explore_reason, explore_rate_used, local_priority_mode, long_gate_pass, long_gate_reason, star_override_used, star_override_reason, candidate_starved, fallback_used, fallback_path, starvation_reason.
- Candidate counts: n_candidates_generated, n_candidates_after_hard_filters, n_candidates_after_duplicates, n_candidates_after_star_caps, n_candidates_scored.

- Endpoint diagnostics: deg_i, deg_j, recent_deg_i, recent_deg_j, mu_i, mu_j, sigma_i, sigma_j, p_ij, U0_ij.
- Star-cap diagnostics: star_cap_rejects, star_cap_reject_items.

Value

A tibble with one row per attempted step, in execution order.

See Also

[adaptive_get_logs\(\)](#), [adaptive_round_log\(\)](#), [adaptive_rank_run_live\(\)](#)

Other adaptive logs: [adaptive_get_logs\(\)](#), [adaptive_item_log\(\)](#), [adaptive_results_history\(\)](#), [adaptive_round_log\(\)](#), [summarize_items\(\)](#), [summarize_refits\(\)](#)

Examples

```
state <- adaptive_rank_start(c("a", "b", "c"), seed = 1)
adaptive_step_log(state)
```

alternate_pair_order	<i>Deterministically alternate sample order in pairs</i>
----------------------	--

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.

See Also

`read_samples_df()`, `read_samples_dir()`

Other pairing and data: `make_pairs()`, `randomize_pair_order()`, `read_samples_df()`, `read_samples_dir()`, `sample_pairs()`, `sample_reverse_pairs()`

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 <code>Sys.getenv("ANTHROPIC_API_KEY")</code> .
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": • Omitted temperature and top_p values are not sent, so the model/provider defaults apply. • 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 model-default sampling 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, omitted sampling parameters use the model/provider defaults.

Value

A tibble with one row and columns:

custom_id Stable ID for the pair (pair_uid if supplied via ...; otherwise "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.

See Also

[check_llm_api_keys\(\)](#), [llm_compare_pair\(\)](#)

Other live backends: [check_llm_api_keys\(\)](#), [gemini_compare_pair_live\(\)](#), [llm_compare_pair\(\)](#), [ollama_compare_pair_live\(\)](#), [openai_compare_pair_live\(\)](#), [submit_anthropic_pairs_live\(\)](#), [submit_gemini_pairs_live\(\)](#), [submit_llm_pairs\(\)](#), [submit_ollama_pairs_live\(\)](#), [submit_openai_pairs_live\(\)](#), [submit_together_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [vertex_compare_pair_live\(\)](#)

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()

# Standard comparison with model-default sampling and no 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 = tpl,
    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 = tpl,
    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 <code>list(custom_id = <chr>, params = <list>)</code> . You can obtain this list from the output of build_anthropic_batch_requests via <code>split / Map</code> , or use <code>run_anthropic_batch_pipeline</code> .
api_key	Optional Anthropic API key. Defaults to <code>Sys.getenv("ANTHROPIC_API_KEY")</code> .
anthropic_version	Anthropic API version string passed as the <code>anthropic-version</code> HTTP header. Defaults to <code>"2023-06-01"</code> .

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`.

See Also

[llm_submit_pairs_batch\(\)](#), [llm_download_batch_results\(\)](#)

Other batch backends: [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_create_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [openai_upload_batch_file\(\)](#), [run_anthropic_batch_pipeline\(\)](#), [run_gemini_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#), [write_openai_batch_file\(\)](#)

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 = tmp1
)

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

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

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`.

See Also

```
llm_submit_pairs_batch(), llm_download_batch_results()
```

Other batch backends: `anthropic_create_batch()`, `anthropic_get_batch()`, `anthropic_poll_batch_until_complete()`, `build_anthropic_batch_requests()`, `build_gemini_batch_requests()`, `build_openai_batch_requests()`, `gemini_create_batch()`, `gemini_download_batch_results()`, `gemini_get_batch()`, `gemini_poll_batch_until_complete()`, `llm_download_batch_results()`, `llm_resume_multi_batches()`, `llm_submit_pairs_batch()`, `llm_submit_pairs_multi_batch()`, `openai_create_batch()`, `openai_download_batch_output()`, `openai_get_batch()`, `openai_poll_batch_until_complete()`, `openai_upload_batch_file()`, `run_anthropic_batch_pipeline()`, `run_gemini_batch_pipeline()`, `run_openai_batch_pipeline()`, `write_openai_batch_file()`

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)
```

<code>anthropic_get_batch</code>	<i>Retrieve an Anthropic Message Batch by ID</i>
----------------------------------	--

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 <code>Sys.getenv("ANTHROPIC_API_KEY")</code> .
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`.

See Also

`llm_submit_pairs_batch()`, `llm_download_batch_results()`

Other batch backends: `anthropic_create_batch()`, `anthropic_download_batch_results()`, `anthropic_poll_batch_until_complete()`, `build_anthropic_batch_requests()`, `build_gemini_batch_requests()`, `build_openai_batch_requests()`, `gemini_create_batch()`, `gemini_download_batch_results()`, `gemini_get_batch()`, `gemini_poll_batch_until_complete()`, `llm_download_batch_results()`, `llm_resume_multi_batches()`, `llm_submit_pairs_batch()`, `llm_submit_pairs_multi_batch()`, `openai_create_batch()`, `openai_download_batch_output()`, `openai_get_batch()`, `openai_poll_batch_until_complete()`, `openai_upload_batch_file()`, `run_anthropic_batch_pipeline()`, `run_gemini_batch_pipeline()`, `run_openai_batch_pipeline()`, `write_openai_batch_file()`

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.

See Also

[llm_submit_pairs_batch\(\)](#), [llm_download_batch_results\(\)](#)

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_create_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [openai_upload_batch_file\(\)](#), [run_anthropic_batch_pipeline\(\)](#), [run_gemini_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#), [write_openai_batch_file\(\)](#)

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":
 - Omitted temperature and top_p values are not sent, so the model/provider defaults apply.
 - 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 $1024 \leq \text{thinking_budget_tokens} < \text{max_tokens}$. Violations of this constraint produce an error.

As a result, batches without extended thinking (reasoning = "none") use model-default sampling. 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.

See Also

[llm_submit_pairs_batch\(\)](#), [llm_download_batch_results\(\)](#)

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_create_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [openai_upload_batch_file\(\)](#), [run_anthropic_batch_pipeline\(\)](#), [run_gemini_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#), [write_openai_batch_file\(\)](#)

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()

# 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

```

```
build_btl_results_data
```

Build canonical results_tbl data for Bayesian BTL MCMC

Description

Converts non-adaptive pairwise outcomes (for example, rows like `example_writing_pairs` with ID1, ID2, better_id) into the canonical `results_tbl` schema required by `fit_bayes_btl_mcmc()`.

Usage

```

build_btl_results_data(
  results,
  phase = "phase2",
  backend = "non_adaptive_import",
  model = "unknown",
  iter_start = 1L,
  received_at_start = as.POSIXct("1970-01-01 00:00:00", tz = "UTC")
)

```

Arguments

results	A data frame or tibble containing columns ID1, ID2, and better_id.
phase	Length-1 phase label for all rows. Must be one of "phase1", "phase2", or "phase3". Defaults to "phase2".
backend	Length-1 backend label to record in output metadata.
model	Length-1 model label to record in output metadata.
iter_start	Integer starting value for iter. Defaults to 1L.
received_at_start	Length-1 POSIXct timestamp for the first row. Subsequent rows increment by one second.

Details

The output is deterministic and schema-valid:

- stable unordered_key / ordered_key values,
- deterministic pair_uid as "<unordered_key>#<occurrence>",
- deterministic iter and received_at sequences.

Value

A tibble in canonical results_tbl format with columns: pair_uid, unordered_key, ordered_key, A_id, B_id, better_id, winner_pos, phase, iter, received_at, backend, model.

See Also

[fit_bayes_btl_mcmc\(\)](#)

Other Bayesian models: [fit_bayes_btl_mcmc\(\)](#)

Examples

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

results_tbl <- build_btl_results_data(example_writing_pairs)
head(results_tbl)

ids <- sort(unique(c(results_tbl$A_id, results_tbl$B_id)))
ids
```

build_bt_data	<i>Build Bradley-Terry comparison data from pairwise results</i>
---------------	--

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 either ID1/ID2/better_id or A_id/B_id/better_id.

Details

It accepts either:

- legacy columns ID1, ID2, better_id, or
- canonical columns A_id, B_id, better_id.

Rows where better_id does not match either side of the pair (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.

See Also

[fit_bt_model\(\)](#), [summarize_bt_fit\(\)](#)

Other frequentist models: [build_elo_data\(\)](#), [fit_bt_model\(\)](#), [fit_elo_model\(\)](#), [summarize_bt_fit\(\)](#)

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	<i>Build EloChoice comparison data from pairwise results</i>
----------------	--

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 either ID1/ID2/better_id or A_id/B_id/better_id.

Details

It accepts either:

- legacy columns ID1, ID2, better_id, or
- canonical columns A_id, B_id, better_id.

Rows where better_id does not match either side of the pair (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.

See Also

`build_bt_data()`, `fit_bt_model()`

Other frequentist models: `build_bt_data()`, `fit_bt_model()`, `fit_elo_model()`, `summarize_bt_fit()`

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 = "low",
  custom_id_prefix = "GEM",
  temperature = NULL,
  top_p = NULL,
  top_k = NULL,
  max_output_tokens = NULL,
  service_tier = "standard",
  include_thoughts = FALSE,
  ...
)
```

Arguments

<code>pairs</code>	Tibble or data frame with at least columns ID1, text1, ID2, text2. Typically created by make_pairs , sample_pairs , and randomize_pair_order .
<code>model</code>	Gemini model name, for example "gemini-3.5-flash-lite". 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.
<code>trait_name</code>	Short label for the trait (for example "Overall Quality").
<code>trait_description</code>	Full-text description of the trait or rubric.
<code>prompt_template</code>	Prompt template string, typically from set_prompt_template . The template should embed your full instructions, rubric text, and <BETTER_SAMPLE> tagging convention.
<code>thinking_level</code>	One of "minimal", "low", "medium", or "high". This is mapped to Gemini's <code>thinkingConfig.thinkingLevel</code> . - For Gemini 3 Flash models (for example "gemini-3-flash-preview"), "minimal" is supported and is passed through as "minimal". - For models not matched by the package's Gemini 3 Flash-name detector (for example "gemini-3.5-flash-lite"), "minimal" is not supported. - For backward compatibility with earlier Gemini 3 Pro usage, "low" maps to "low" and both "medium" and "high" map to "high". "Medium" currently behaves like "High".
<code>custom_id_prefix</code>	Prefix for the <code>custom_id</code> field. Defaults to "GEM" so that IDs take the form "GEM_<ID1>_vs_<ID2>".
<code>temperature</code>	Optional numeric temperature. If NULL, it is omitted and Gemini uses its own default.
<code>top_p</code>	Optional nucleus sampling parameter. If NULL, omitted.
<code>top_k</code>	Optional top-k sampling parameter. If NULL, omitted.
<code>max_output_tokens</code>	Optional integer. If NULL, omitted.
<code>service_tier</code>	Gemini Developer API service tier. Use "standard" (default) or NULL for provider default behavior, or "flex" / "priority" to encode the documented Gemini <code>serviceTier</code> request field.
<code>include_thoughts</code>	Logical; if TRUE, sets <code>thinkingConfig.includeThoughts = TRUE</code> so that Gemini returns visible chain-of-thought. For most pairwise scoring use cases this should remain FALSE.
<code>...</code>	Reserved for future extensions. Any <code>thinking_budget</code> entries are ignored (Gemini 3 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.

See Also

`llm_submit_pairs_batch()`, `llm_download_batch_results()`

Other batch backends: `anthropic_create_batch()`, `anthropic_download_batch_results()`, `anthropic_get_batch()`, `anthropic_poll_batch_until_complete()`, `build_anthropic_batch_requests()`, `build_openai_batch_requests()`, `gemini_create_batch()`, `gemini_download_batch_results()`, `gemini_get_batch()`, `gemini_poll_batch_until_complete()`, `llm_download_batch_results()`, `llm_resume_multi_batches()`, `llm_submit_pairs_batch()`, `llm_submit_pairs_multi_batch()`, `openai_create_batch()`, `openai_download_batch_output()`, `openai_get_batch()`, `openai_poll_batch_until_complete()`, `openai_upload_batch_file()`, `run_anthropic_batch_pipeline()`, `run_gemini_batch_pipeline()`, `run_openai_batch_pipeline()`, `write_openai_batch_file()`

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()
```

```
# Dated tested Gemini Developer API configuration
reqs <- build_gemini_batch_requests(
  pairs          = pairs,
  model          = "gemini-3.5-flash-lite",
  trait_name     = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  thinking_level = "low",
  include_thoughts = TRUE
)
```

```
reqs
```

```
# Gemini 3 Flash example (minimal thinking)
reqs_flash <- build_gemini_batch_requests(
  pairs          = pairs,
  model          = "gemini-3-flash-preview",
  trait_name     = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  thinking_level = "minimal",
```

```

    include_thoughts = FALSE
  )

  reqs_flash

```

```
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

<code>pairs</code>	A data frame or tibble with columns ID1, text1, ID2, and text2.
<code>model</code>	Character scalar giving the OpenAI model name. Supports standard names (e.g. "gpt-4.1", "gpt-5.6-sol") and date-stamped versions (e.g. "gpt-5.4-2026-01-15").
<code>trait_name</code>	Short label for the trait (e.g., "Overall Quality").
<code>trait_description</code>	Full-text definition of the trait.
<code>prompt_template</code>	Character template containing the placeholders {TRAIT_NAME}, {TRAIT_DESCRIPTION}, {SAMPLE_1}, and {SAMPLE_2}. Defaults to <code>set_prompt_template()</code> .
<code>endpoint</code>	Which OpenAI endpoint to target. One of "chat.completions" (default) or "responses".

temperature	Optional temperature parameter. If NULL, it is omitted so the model/provider default applies. Must be NULL for reasoning modes that do not support it.
top_p	Optional top-p parameter. If NULL, it is omitted so the model/provider default applies.
logprobs	Optional logprobs parameter.
reasoning	Optional reasoning effort for GPT-5 series when using the /v1/responses endpoint. For "gpt-5" and "gpt-5-mini", "none" is normalized to "minimal". For later GPT-5.x reasoning models, use model-supported efforts such as "none", "low", "medium", "high", "xhigh", or "max".
include_thoughts	Logical; if TRUE and using responses endpoint with reasoning, requests a summary. Defaults reasoning to "low" for GPT-5 series models 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.

See Also

[llm_submit_pairs_batch\(\)](#), [llm_resume_multi_batches\(\)](#)

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_create_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [openai_upload_batch_file\(\)](#), [run_anthropic_batch_pipeline\(\)](#), [run_gemini_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#), [write_openai_batch_file\(\)](#)

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()
```

```
# 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"
)

# 2. GPT-5.6 Sol Responses Batch with Reasoning
batch_tbl_resp <- build_openai_batch_requests(
  pairs = pairs,
  model = "gpt-5.6-sol",
  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
```

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.

See Also

`trait_description()`, `set_prompt_template()`

Other prompts and traits: `get_prompt_template()`, `list_prompt_templates()`, `register_prompt_template()`, `remove_prompt_template()`, `set_prompt_template()`, `trait_description()`

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	<i>Check configured API keys for LLM backends</i>
--------------------	---

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, GEMINI_API_KEY, and VERTEX_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", "vertex", "together".

service Human-readable service name, e.g. "OpenAI", "Anthropic", "Google Gemini", "Vertex AI Gemini API", "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.

See Also

`llm_compare_pair()`, `submit_llm_pairs()`

Other live backends: `anthropic_compare_pair_live()`, `gemini_compare_pair_live()`, `llm_compare_pair()`, `ollama_compare_pair_live()`, `openai_compare_pair_live()`, `submit_anthropic_pairs_live()`, `submit_gemini_pairs_live()`, `submit_llm_pairs()`, `submit_ollama_pairs_live()`, `submit_openai_pairs_live()`, `submit_together_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `vertex_compare_pair_live()`

Examples

```
# 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
```

check_positional_bias *Check positional preference and bootstrap reversal agreement*

Description

This function diagnoses positional preference in LLM-based paired comparison data and provides a bootstrapped confidence interval for the overall agreement 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 <code>compute_reverse_consistency()</code> that contains a <code>\$details</code> tibble; or • A tibble/data frame with columns <code>key</code>, <code>ID1_main</code>, <code>ID2_main</code>, <code>better_id_main</code>, <code>ID1_rev</code>, <code>ID2_rev</code>, <code>better_id_rev</code>, and <code>is_consistent</code>.
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 <code>NULL</code> (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.

Each row of `details` is one unordered pair after any duplicate judgments have been reduced to a per-direction majority by `compute_reverse_consistency()`. The agreement estimate and its percentile bootstrap interval therefore use unordered pairs as the unit of analysis and treat those rows as independently resampled units. This assumption may be inappropriate when pairs share items.

The direction-specific binomial tests likewise treat unordered-pair outcomes within a direction as independent. The overall test is paired: among inconsistent pairs, it compares the number for which position 1 won both presentations with the number for which position 2 won both. This is the exact conditional form of McNemar's test. It returns `NA` when there are no informative inconsistent pairs. A large p-value is not evidence that positional preference is absent.

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 the paired exact test that position-1 and position-2 inconsistencies are equally likely
- `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

See Also

[compute_reverse_consistency\(\)](#)

Other bias and consistency: [compute_reverse_consistency\(\)](#)

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 unordered pairs that were evaluated in both directions 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 directions. This function assumes each input contains columns ID1, ID2, and better_id, where better_id is the ID of the better sample (not "SAMPLE_1"/"SAMPLE_2").

Per-key majority agreement (duplicates supported). If a pair appears multiple times in main_results and/or reverse_results (e.g., submitted twice), this function aggregates each unordered pair key separately in each direction and takes the *majority* better_id. If there is a tie for the majority winner within a direction, that direction's majority winner is set to NA and the key is excluded from the consistency calculation.

The output details contains exactly one row per unordered pair key, which keeps it compatible with [check_positional_bias](#). Columns other than ID1, ID2, and better_id are not grouping variables. Split results by model, template, trait, reasoning or thinking condition, and any intended replicate unit before calling this function; otherwise their judgments are pooled in the per-key majority.

Value

A list with two elements:

- **summary:** a tibble with one row and columns n_pairs, n_consistent, and prop_consistent. Here, n_pairs counts unordered pair keys with a non-missing majority winner in both directions.
- **details:** a tibble with one row per unordered pair key, including columns key, ID1_main, ID2_main, ID1_rev, ID2_rev, better_id_main, better_id_rev, and is_consistent. Additional columns provide vote counts and tie flags.

See Also[check_positional_bias\(\)](#)Other bias and consistency: [check_positional_bias\(\)](#)**Examples**

```
main <- tibble::tibble(
  ID1      = c("A", "A", "X"),
  ID2      = c("B", "B", "Y"),
  better_id = c("A", "B", "X") # duplicate A-B with disagreement
)
rev <- tibble::tibble(
  ID1      = c("B"),
  ID2      = c("A"),
  better_id = c("A")
)
compute_reverse_consistency(main, rev)$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-small13.2:24b", "qwen3:32b", "gemma3:27b"). All other models currently reported by <code>ollama ps</code> 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)
```

estimate_llm_pairs_cost

Estimate LLM token usage and cost for a set of pairwise comparisons

Description

Estimate total token usage and cost for running a large set of pairwise comparisons by:

- running a small pilot on `n_test` pairs (live calls) to observe `prompt_tokens` and `completion_tokens`, and
- using the pilot to calibrate a prompt-bytes-to-input-token model for the remaining pairs, and
- prorating output tokens for the remaining pairs from the pilot distribution.

Usage

```
estimate_llm_pairs_cost(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  backend = c("openai", "anthropic", "gemini", "vertex", "together"),
  endpoint = c("chat.completions", "responses"),
  mode = c("live", "batch"),
  n_test = 25,
  test_strategy = c("stratified_prompt_bytes", "random", "first"),
  seed = NULL,
  cost_per_million_input,
  cost_per_million_output,
  batch_discount = 1,
  budget_quantile = 0.9,
  return_test_results = TRUE,
  return_remaining_pairs = TRUE,
  ...
)
```

Arguments

<code>pairs</code>	Tibble or data frame with at least columns <code>ID1</code> , <code>text1</code> , <code>ID2</code> , <code>text2</code> . Typically created by make_pairs , sample_pairs , and randomize_pair_order .
<code>model</code>	Model name to use for the pilot run (and for the target job).
<code>trait_name</code>	Short label for the trait (for example "Overall Quality").
<code>trait_description</code>	Full-text description of the trait or rubric.
<code>prompt_template</code>	Prompt template string, typically from set_prompt_template .

backend	Backend for the pilot run; one of "openai", "anthropic", "gemini", "vertex", or "together".
endpoint	OpenAI endpoint; one of "chat.completions" or "responses". Ignored for other backends.
mode	Target execution mode for the full job; one of "live" or "batch". The pilot is always run live. If mode = "batch", batch_discount is applied to the estimated cost for the remaining (non-pilot) pairs.
n_test	Number of pilot pairs to run live. Defaults to 25 or fewer if fewer pairs are supplied.
test_strategy	Strategy for selecting pilot pairs: "stratified_prompt_bytes" (default), "random", or "first".
seed	Optional integer seed used for pilot sampling when test_strategy is not "first".
cost_per_million_input	Cost per one million input tokens (prompt tokens), in your currency of choice. Supply a current price for the selected provider, model, endpoint, and execution mode.
cost_per_million_output	Cost per one million output tokens (completion tokens). Reasoning/thinking tokens are treated as output. Supply and verify this price as for cost_per_million_input.
batch_discount	Numeric scalar multiplier applied to the estimated cost for the remaining pairs when mode = "batch". For example, if batch pricing is 50 percent of live pricing, use batch_discount = 0.5.
budget_quantile	Quantile used for the "budget" output-token estimate for remaining pairs. Defaults to 0.9 (p90).
return_test_results	Logical; if TRUE, include pilot results in the returned object so you can reuse them and avoid paying twice.
return_remaining_pairs	Logical; if TRUE, include the remaining pairs (excluding pilot pairs) in the returned object.
...	Additional arguments forwarded to submit_llm_pairs for the pilot run (for example api_key, reasoning, include_thoughts, max_tokens, etc.).

Details

The estimator does not require a provider tokenizer. Input tokens are estimated from the byte length of the fully constructed prompt and calibrated on the pilot's observed prompt_tokens. Expected remaining completion tokens use the arithmetic mean of usable pilot completion counts. Budget completion tokens use the type-7 sample quantile selected by budget_quantile. These are token-use estimates; the function does not supply, validate, or refresh provider prices.

Pilot pairs are selected from pairs and are excluded from remaining_pairs whether their calls succeed or fail. Calibration uses successful normalized rows with non-missing input-token counts and a missing or 200 status. Completion estimates use the analogous usable completion-token rows. With no usable input counts, calibration falls back to one token per four prompt bytes; with one row

it uses a zero-intercept ratio; with multiple distinct byte lengths it uses ordinary least squares; and with multiple identical byte lengths it uses their mean token count as a constant prediction. With no usable completion counts, completion and total-cost estimates that depend on them are NA.

The observed pilot token totals are always priced as live calls. When mode = "batch", batch_discount applies only to estimated remaining input and output tokens. The returned pilot object and remaining pairs are not automatically merged into a later job result. Local Ollama models are unsupported because this function estimates token-billed API cost rather than local compute cost.

Value

An object of class "pairwiseLLM_cost_estimate", a list with:

summary A one-row tibble with expected and budget token and cost estimates (and pilot usage).

calibration A list describing the input-token calibration (coefficients and fit diagnostics).

test_pairs The pilot pair subset.

pilot Pilot results (when return_test_results = TRUE).

remaining_pairs Remaining pairs (when return_remaining_pairs = TRUE).

See Also

`make_pairs()`, `submit_llm_pairs()`, `llm_submit_pairs_batch()`

Examples

```
## Not run:
# Requires an API key and internet access.
data("example_writing_samples", package = "pairwiseLLM")

pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 50, seed = 123)

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

est <- estimate_llm_pairs_cost(
  pairs = pairs,
  backend = "openai",
  model = "gpt-4.1",
  endpoint = "chat.completions",
  trait_name = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  mode = "batch",
  batch_discount = 0.5,
  n_test = 10,
  cost_per_million_input = 0.15,
  cost_per_million_output = 0.60
)
```

```
est
est$summary

# Reuse pilot results and run only remaining pairs:
remaining <- est$remaining_pairs

## 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.

Source

Package-author-created fictional API response fixtures. Reproduction code is in `data-raw/example_writing_samples.R` in the source repository.

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.

Source

Deterministically derived from `example_writing_samples` by `data-raw/example_writing_samples.R` in the source repository.

Examples

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

```
example_writing_results
```

Example canonical results table for writing comparisons

Description

Canonical results_tbl representation of [example_writing_pairs](#), intended for direct use with [fit_bayes_btl_mcmc](#) and other functions that require adaptive schema-compatible results input.

Usage

```
data("example_writing_results")
```

Format

A tibble with 190 rows and 12 variables:

pair_uid Deterministic pair attempt ID.

unordered_key Unordered pair key ("min:max").

ordered_key Ordered pair key ("A_id:B_id").

A_id Character ID in first position.

B_id Character ID in second position.

better_id Character ID judged better in this comparison.

winner_pos Integer winner position (1L or 2L).

phase Phase label.

iter Integer step index.

received_at POSIXct timestamp in UTC.

backend Backend label for provenance.

model Model label for provenance.

Source

Deterministically derived from `example_writing_pairs` by `data-raw/example_writing_samples.R` in the source repository.

Examples

```
data("example_writing_results")
head(example_writing_results)
```

example_writing_samples
<i>Example dataset of writing samples</i>

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).

Source

Package-author-created synthetic examples. Reproduction code is in data-raw/example_writing_samples.R in the source repository.

Examples

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

example_writing_samples1000
<i>Synthetic Writing Samples with Controlled Quality Levels (N = 1000)</i>

Description

A synthetic dataset of 1,000 short writing samples generated by a large language model for use in pairwise comparison and ranking experiments.

Usage

```
data("example_writing_samples1000")
```

Format

A tibble with 1,000 rows and 7 variables:

ID Character. Unique sample identifier (S0001–S1000).

text Character. The writing sample (approximately 120–180 words).

quality_level Integer. Intended quality level used during generation (1–20).

theta_true Numeric. Centered latent-quality proxy derived from `quality_level`.

prompt_id Character. Identifier for the generation prompt template.

model Character. Language model used to generate the samples.

created_at POSIXct. Timestamp (UTC) when the samples were generated.

Details

Samples are generated in 20 discrete quality levels (1 = lowest, 20 = highest), with multiple responses per level. Quality levels are intended to represent overlapping ranges of overall writing quality rather than a strict total ordering, allowing for realistic noise and near-ties in pairwise judgments.

All samples respond to the same writing prompt to avoid topic effects. The dataset is primarily intended for benchmarking ranking models and for comparing random versus adaptive pair selection strategies under limited judgment budgets.

The column `theta_true` provides a centered numeric proxy for the latent quality dimension derived from `quality_level`. This proxy is intended for evaluation purposes (e.g., rank recovery or correlation) and does not imply a perfectly ordered ground truth at the individual-sample level.

Source

Generated via live OpenAI API calls using a controlled, bucketed quality prompt. See `data-raw/generate_example_writing` for details.

Examples

```
data(example_writing_samples1000)
head(example_writing_samples1000)
```

fit_bayes_btl_mcmc *Full Bayesian BTL inference via CmdStanR (adaptive-compatible)*

Description

Runs full Bayesian posterior inference for a Bradley–Terry–Luce (BTL) style model using the package’s CmdStan machinery, but in a standalone (non-adaptive) context. The function is designed so downstream diagnostics and reporting can reuse the existing adaptive summary tools (notably [summarize_items\(\)](#) and [summarize_refits\(\)](#)) without requiring new summary functions.

Usage

```
fit_bayes_btl_mcmc(
  results,
  ids,
  model_variant = "btl_e_b",
  cmdstan = list(iter_warmup = 1000, iter_sampling = 1000, seed = NULL, core_fraction =
    0.8),
  pair_counts = NULL,
  subset_method = c("first", "sample"),
  seed = NULL,
  inference_contract = NULL
)
```

Arguments

results	Canonical results_tbl with A_id, B_id, and better_id (plus the standard adaptive results columns). See validate_results_tbl() for required structure. For legacy ID1/ID2/better_id data, first use build_btl_results_data() .
ids	Character vector of all sample ids (length N).
model_variant	Model variant label: "btl", "btl_e", "btl_b", or "btl_e_b". Defaults to "btl_e_b".
cmdstan	List of CmdStan settings. Common fields: chains Number of chains (defaults to min(8, physical_cores) via internal resolution). iter_warmup Warmup iterations (default 1000). iter_sampling Sampling iterations (default 1000). seed Optional integer seed forwarded to CmdStan (default NULL). core_fraction Fraction of physical cores for parallelization (default 0.8). output_dir Optional directory for CmdStan output.
pair_counts	Optional integer vector of subset sizes (e.g., c(200, 500, 1000)). When provided, the model is fit once per subset size and the round log contains one row per fit. If NULL, a single fit is run using all rows in results.

subset_method	Subset strategy when pair_counts is provided: "first" (default) uses the first n rows of results for each refit; "sample" draws a random permutation once and then takes the first n rows of that permutation for each refit.
seed	Optional integer seed for deterministic subset selection when subset_method = "sample". When NULL, falls back to cmdstan\$seed if provided.
inference_contract	Optional list of inference-routing semantics to attach to each fit contract. When omitted, values are inferred from results\$phase and optional results\$judge_scope.

Details

Internally, the function can optionally refit the model on increasing subsets of the observed comparisons (via pair_counts). Each refit is treated as a "refit" in the adaptive logging sense, producing:

- one round-log row per refit (compatible with round_log_schema()),
- one item-log table per refit (compatible with .adaptive_item_log_schema()).

Value

A list with:

item_log_list List of item-log tables, one per refit, matching the canonical adaptive item log schema. This is the preferred structure for reuse with [summarize_items\(\)](#).

item_summary A single tibble formed by row-binding item_log_list (kept for backward compatibility). Each row corresponds to an item within a refit; refit_id identifies the refit.

round_log Tibble matching the canonical adaptive round log schema (one row per refit).

fits List of BTL fit contracts (one per refit).

fit Single fit contract (only when one refit is run).

See Also

[build_btl_results_data\(\)](#)

Other Bayesian models: [build_btl_results_data\(\)](#)

Examples

```
## Not run:
results <- tibble::tibble(
  pair_uid = "A:B#1",
  unordered_key = "A:B",
  ordered_key = "A:B",
  A_id = "A",
  B_id = "B",
  better_id = "A",
  winner_pos = 1L,
  phase = "phase2",
  iter = 1L,
  received_at = as.POSIXct("2026-01-01 00:00:00", tz = "UTC"),
  backend = "openai",
```

```

    model = "gpt-test"
  )

  fit <- fit_bayes_btl_mcmc(
    results,
    ids = c("A", "B"),
    model_variant = "btl_e_b"
  )

  # Generate summaries
  summarize_refits(fit)
  summarize_items(fit)

  ## End(Not run)

```

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 <code>sirt::btm()</code> or <code>BradleyTerry2::BTm()</code> .

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.

See Also

[build_bt_data\(\)](#), [summarize_bt_fit\(\)](#)

Other frequentist models: [build_bt_data\(\)](#), [build_elo_data\(\)](#), [fit_elo_model\(\)](#), [summarize_bt_fit\(\)](#)

Examples

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

if (requireNamespace("sirt", quietly = TRUE)) {
  fit1 <- fit_bt_model(bt, engine = "sirt")
}
if (requireNamespace("BradleyTerry2", quietly = TRUE)) {
  fit2 <- fit_bt_model(bt, engine = "BradleyTerry2")
}
```

fit_elo_model	<i>Fit an EloChoice model to pairwise comparison data</i>
---------------	---

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 <code>EloChoice::elochoice</code> . 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 <code>EloChoice::elochoice()</code> .

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.

See Also

[build_bt_data\(\)](#), [fit_bt_model\(\)](#)

Other frequentist models: [build_bt_data\(\)](#), [build_elo_data\(\)](#), [fit_bt_model\(\)](#), [summarize_bt_fit\(\)](#)

Examples

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

elo_data <- build_elo_data(example_writing_pairs)

if (requireNamespace("EloChoice", quietly = TRUE)) {
  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 / Flash) 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 = "low",  
  temperature = NULL,  
  top_p = NULL,  
  top_k = NULL,  
  max_output_tokens = NULL,  
  service_tier = "standard",  
  api_version = "v1beta",  
  include_raw = FALSE,  
  include_thoughts = FALSE,  
  pair_uid = NULL,  
  ...  
)
```

Arguments

<code>ID1</code>	Character ID for the first sample.
<code>text1</code>	Character containing the first sample text.
<code>ID2</code>	Character ID for the second sample.
<code>text2</code>	Character containing the second sample text.
<code>model</code>	Gemini model identifier (for example "gemini-3.5-flash-lite" or "gemini-3-flash-preview"). The value is interpolated into the path "{api_version}/models/<model>:generateContent".
<code>trait_name</code>	Short label for the trait (e.g. "Overall Quality").

trait_description	Full-text trait / rubric description.
prompt_template	Prompt template string, typically from <code>set_prompt_template()</code> . The template should embed <BETTER_SAMPLE> tags.
api_key	Optional Gemini API key (defaults to <code>Sys.getenv("GEMINI_API_KEY")</code>).
thinking_level	One of "minimal", "low", "medium", or "high". This controls the maximum depth of internal reasoning. • For Gemini 3 Flash models (for example "gemini-3-flash-preview"), "minimal" is supported and is passed through as "minimal". • For models not matched by the package's Gemini 3 Flash-name detector (for example "gemini-3.5-flash-lite"), "minimal" is not supported. • For backward compatibility with earlier Gemini 3 Pro usage, "low" maps to "low" and both "medium" and "high" map to "high". "Medium" currently behaves like "High".
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.
service_tier	Gemini Developer API service tier. Use "standard" (default) or NULL for the provider default request, or "flex" / "priority" to encode the documented Gemini serviceTier request field.
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\$thinkingConfig 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.
pair_uid	Optional stable per-pair identifier; when supplied, this value is used verbatim as custom_id (otherwise custom_id defaults to "LIVE_<ID1>-vs_<ID2>").
...	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` - stable ID for the pair (`pair_uid` if supplied).
- `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_.

See Also

`check_llm_api_keys()`, `llm_compare_pair()`

Other live backends: `anthropic_compare_pair_live()`, `check_llm_api_keys()`, `llm_compare_pair()`, `ollama_compare_pair_live()`, `openai_compare_pair_live()`, `submit_anthropic_pairs_live()`, `submit_gemini_pairs_live()`, `submit_llm_pairs()`, `submit_ollama_pairs_live()`, `submit_openai_pairs_live()`, `submit_together_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `vertex_compare_pair_live()`

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()

# Dated tested Gemini Developer API configuration
res <- gemini_compare_pair_live(
```

```

    ID1          = "S01",
    text1        = "Text 1",
    ID2          = "S02",
    text2        = "Text 2",
    model        = "gemini-3.5-flash-lite",
    trait_name    = td$name,
    trait_description = td$description,
    prompt_template = tmpl,
    thinking_level = "low",
    include_thoughts = FALSE,
    include_raw    = FALSE
  )

  res
  res$better_id

# Gemini 3 Flash example (minimal thinking)
res_flash <- gemini_compare_pair_live(
  ID1          = "S01",
  text1        = "Text 1",
  ID2          = "S02",
  text2        = "Text 2",
  model        = "gemini-3-flash-preview",
  trait_name    = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  thinking_level = "minimal",
  include_thoughts = FALSE,
  include_raw    = FALSE
)

res_flash

## 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

<code>requests</code>	List of GenerateContent request objects, each of the form <code>list(contents = ..., generationConfig = ...)</code> . You can obtain this list from the output of build_gemini_batch_requests via <code>batch\$request</code> .
<code>model</code>	Gemini model name, for example "gemini-3.5-flash-lite".
<code>api_key</code>	Optional Gemini API key. Defaults to <code>Sys.getenv("GEMINI_API_KEY")</code> .
<code>api_version</code>	API version string for the path; defaults to "v1beta".
<code>display_name</code>	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`.

See Also

[llm_submit_pairs_batch\(\)](#), [llm_download_batch_results\(\)](#)

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_create_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [openai_upload_batch_file\(\)](#), [run_anthropic_batch_pipeline\(\)](#), [run_gemini_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#), [write_openai_batch_file\(\)](#)

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-3.5-flash-lite",
  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-3.5-flash-lite"
)

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 <code>gemini_get_batch()</code>) or a character batch name such as "batches/123...".
requests_tbl	Tibble/data frame with a <code>custom_id</code> 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`.

See Also

`llm_submit_pairs_batch()`, `llm_download_batch_results()`

Other batch backends: `anthropic_create_batch()`, `anthropic_download_batch_results()`, `anthropic_get_batch()`, `anthropic_poll_batch_until_complete()`, `build_anthropic_batch_requests()`, `build_gemini_batch_requests()`, `build_openai_batch_requests()`, `gemini_create_batch()`, `gemini_get_batch()`, `gemini_poll_batch_until_complete()`, `llm_download_batch_results()`, `llm_resume_multi_batches()`, `llm_submit_pairs_batch()`, `llm_submit_pairs_multi_batch()`, `openai_create_batch()`, `openai_download_batch_output()`, `openai_get_batch()`, `openai_poll_batch_until_complete()`, `openai_upload_batch_file()`, `run_anthropic_batch_pipeline()`, `run_gemini_batch_pipeline()`, `run_openai_batch_pipeline()`, `write_openai_batch_file()`

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 <code>Sys.getenv("GEMINI_API_KEY")</code> .
api_version	API version string for the path; defaults to <code>"v1beta"</code> .

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.

See Also

```
llm_submit_pairs_batch(), llm_download_batch_results()
```

Other batch backends: `anthropic_create_batch()`, `anthropic_download_batch_results()`, `anthropic_get_batch()`, `anthropic_poll_batch_until_complete()`, `build_anthropic_batch_requests()`, `build_gemini_batch_requests()`, `build_openai_batch_requests()`, `gemini_create_batch()`, `gemini_download_batch_results()`, `gemini_poll_batch_until_complete()`, `llm_download_batch_results()`, `llm_resume_multi_batches()`, `llm_submit_pairs_batch()`, `llm_submit_pairs_multi_batch()`, `openai_create_batch()`, `openai_download_batch_output()`, `openai_get_batch()`, `openai_poll_batch_until_complete()`, `openai_upload_batch_file()`, `run_anthropic_batch_pipeline()`, `run_gemini_batch_pipeline()`, `run_openai_batch_pipeline()`, `write_openai_batch_file()`

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](#).

See Also

[llm_submit_pairs_batch\(\)](#), [llm_download_batch_results\(\)](#)

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_create_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [openai_upload_batch_file\(\)](#), [run_anthropic_batch_pipeline\(\)](#), [run_gemini_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#), [write_openai_batch_file\(\)](#)

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](#)

Other prompts and traits: [build_prompt\(\)](#), [list_prompt_templates\(\)](#), [register_prompt_template\(\)](#), [remove_prompt_template\(\)](#), [set_prompt_template\(\)](#), [trait_description\(\)](#)

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.

See Also

[trait_description\(\)](#), [set_prompt_template\(\)](#)

Other prompts and traits: [build_prompt\(\)](#), [get_prompt_template\(\)](#), [register_prompt_template\(\)](#), [remove_prompt_template\(\)](#), [set_prompt_template\(\)](#), [trait_description\(\)](#)

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", "vertex", "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\(\)](#)
- "vertex" → [vertex_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", "vertex", "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", use Anthropic model names such as "claude-sonnet-4-5". For "gemini" and "vertex", use the corresponding Gemini model names (for example "gemini-3.5-flash-lite")

or "gemini-2.5-flash"). For "together", use Together.ai model identifiers such as "deepseek-ai/DeepSeek-V4-Flash-0731". For "ollama", use a local model name known to the Ollama server (for example "mistral-small13.2:24b", "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 <code>set_prompt_template()</code> .
backend	Character scalar indicating which LLM provider to use. One of "openai", "anthropic", "gemini", "vertex", "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 <code>openai_compare_pair_live()</code> . For "anthropic", "gemini", "vertex", "together", 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, VERTEX_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 <code>openai_compare_pair_live()</code> and typically include arguments such as temperature, top_p, logprobs, reasoning, and include_thoughts. For "anthropic", "gemini", and "vertex" they are forwarded to the corresponding live helper and may include parameters such as reasoning, include_thoughts, max_output_tokens, service_tier, thinking_level, or provider-specific options. For "ollama", arguments are forwarded to <code>ollama_compare_pair_live()</code> and may include host, think, num_ctx, and other Ollama-specific controls. service_tier is provider-specific: OpenAI, Gemini Developer API, and Vertex each validate and encode it using their own request surfaces. For Gemini Developer API and Vertex, supported public values are "standard" (default / omitted), "flex", and "priority".

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", "vertex", "together", 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()`, `vertex_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.

Other live backends: `anthropic_compare_pair_live()`, `check_llm_api_keys()`, `gemini_compare_pair_live()`, `ollama_compare_pair_live()`, `openai_compare_pair_live()`, `submit_anthropic_pairs_live()`, `submit_gemini_pairs_live()`, `submit_llm_pairs()`, `submit_ollama_pairs_live()`, `submit_openai_pairs_live()`, `submit_together_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `vertex_compare_pair_live()`

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 = tml,
    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 = tml,
  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-small13.2:24b",
  trait_name = td$name,
  trait_description = td$description,
  prompt_template = tml,
  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

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

Value

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

See Also

`llm_submit_pairs_batch()`, `llm_submit_pairs_multi_batch()`

Other batch backends: `anthropic_create_batch()`, `anthropic_download_batch_results()`, `anthropic_get_batch()`, `anthropic_poll_batch_until_complete()`, `build_anthropic_batch_requests()`, `build_gemini_batch_requests()`, `build_openai_batch_requests()`, `gemini_create_batch()`, `gemini_download_batch_results()`, `gemini_get_batch()`, `gemini_poll_batch_until_complete()`, `llm_resume_multi_batches()`, `llm_submit_pairs_batch()`, `llm_submit_pairs_multi_batch()`, `openai_create_batch()`, `openai_download_batch_output()`, `openai_get_batch()`, `openai_poll_batch_until_complete()`, `openai_upload_batch_file()`, `run_anthropic_batch_pipeline()`, `run_gemini_batch_pipeline()`, `run_openai_batch_pipeline()`, `write_openai_batch_file()`

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_resume_multi_batches

Resume polling and download results for multiple batch jobs

Description

This function takes the output of `llm_submit_pairs_multi_batch()` (or a previously written registry CSV) and polls each batch until completion, downloading and parsing results as they finish. It implements a conservative polling loop with a configurable interval between rounds and a small delay between individual jobs to reduce the risk of API rate-limit errors. The `httr2` retry wrapper is still invoked for each API call, so transient HTTP errors will be retried with exponential back-off.

Usage

```

llm_resume_multi_batches(
  jobs = NULL,
  output_dir = NULL,
  interval_seconds = 60,
  per_job_delay = 2,
  write_results_csv = FALSE,
  keep_jsonl = TRUE,
  write_registry = FALSE,
  tag_prefix = "<BETTER_SAMPLE>",
  tag_suffix = "</BETTER_SAMPLE>",
  verbose = FALSE,
  write_combined_csv = FALSE,
  combined_csv_path = NULL,
  openai_max_retries = 3
)

```

Arguments

jobs	A list of job objects returned by <code>llm_submit_pairs_multi_batch()</code> . If NULL, a registry CSV is loaded from <code>output_dir</code> and converted into an internal jobs structure.
output_dir	Directory containing the batch files and (optionally) the registry CSV. If jobs is NULL, this directory must be supplied so that the registry can be loaded. When jobs is provided and <code>output_dir</code> is NULL, the directory is inferred from the first job's <code>batch_output_path</code> . When writing results CSVs or updating the registry, this directory is used.
interval_seconds	Number of seconds to wait between rounds of polling unfinished batches. The default (60) mirrors the example in the advanced vignette.
per_job_delay	Number of seconds to wait between polling individual jobs within a single round. A small delay (e.g. 2) can help prevent 429 (Too Many Requests) responses.
write_results_csv	Logical; if TRUE, each batch's parsed results are written to a CSV file (<code>csv_path</code>) in <code>output_dir</code> as soon as they are available. If FALSE (the default), results are kept in memory.
keep_jsonl	Logical; if FALSE, the <code>.jsonl</code> input and output files will be deleted after the job results have been parsed. Defaults to TRUE.
write_registry	Logical; if TRUE, a CSV registry of batch jobs will be written (or updated) at the end of polling. When reading jobs from a saved registry via <code>output_dir</code> , this argument can be used to control whether the registry is refreshed on disk as job statuses change. Defaults to FALSE. See <code>llm_submit_pairs_multi_batch()</code> for additional details on the registry format.
tag_prefix, tag_suffix	Character strings passed to <code>parse_anthropic_batch_output()</code> and <code>parse_gemini_batch_output()</code> . These tags mark the start and end of the "better" sample in each provider's output. The defaults match those used in the vignette.
verbose	Logical; if TRUE, prints progress messages during polling and result processing. Messages include the batch ID, provider, and current state on each polling round, as well as summary messages when combined results are written to disk. Defaults to FALSE.
write_combined_csv	Logical; if TRUE, the combined results tibble returned by the function will also be written to a CSV file. The path to write this file is determined by <code>combined_csv_path</code> . Defaults to FALSE.
combined_csv_path	Optional file path for the combined results CSV. If <code>write_combined_csv = TRUE</code> and <code>combined_csv_path</code> is NULL, the combined results will be written to <code>file.path(output_dir, "combined_results.csv")</code> . When a non-NULL value is supplied, it is treated as an absolute path if it begins with <code>"/"</code> , <code>"~/</code> ", or a Windows drive letter (e.g. <code>"C:"</code>), or if it contains a directory component (i.e. <code>dirname(combined_csv_path) != "."</code>). In that case it will be used exactly as given. Otherwise the file name is assumed to be relative to <code>output_dir</code> . This argument is ignored when <code>write_combined_csv = FALSE</code> .

openai_max_retries

Integer giving the maximum number of times to retry certain OpenAI API calls when a transient HTTP 5xx error occurs. In particular, when downloading batch output with `openai_download_batch_output()`, the function will attempt to fetch the output file up to `openai_max_retries` times if an `httr2_http_500` error is raised. Between retries the function sleeps for `per_job_delay` seconds. Set to a small positive value (e.g. 3) to automatically recover from occasional server errors. Defaults to 3.

Value

A list with four elements: `jobs`, the updated jobs list with each element containing parsed results and a done flag; `combined`, a tibble obtained by binding all completed results (NULL if no batches completed); `failed_attempts`, a tibble of failed attempts captured during normalization; and `batch_failures`, a tibble describing batches that reached a terminal non-success status. If `write_results_csv` is TRUE, the combined tibble is still returned in memory. If `write_combined_csv` is TRUE, the combined tibble is also written to a CSV file on disk (see `combined_csv_path` for details) but is still returned in memory.

See Also

`llm_submit_pairs_batch()`, `llm_download_batch_results()`

Other batch backends: `anthropic_create_batch()`, `anthropic_download_batch_results()`, `anthropic_get_batch()`, `anthropic_poll_batch_until_complete()`, `build_anthropic_batch_requests()`, `build_gemini_batch_requests()`, `build_openai_batch_requests()`, `gemini_create_batch()`, `gemini_download_batch_results()`, `gemini_get_batch()`, `gemini_poll_batch_until_complete()`, `llm_download_batch_results()`, `llm_submit_pairs_batch()`, `llm_submit_pairs_multi_batch()`, `openai_create_batch()`, `openai_download_batch_output()`, `openai_get_batch()`, `openai_poll_batch_until_complete()`, `openai_upload_batch_file()`, `run_anthropic_batch_pipeline()`, `run_gemini_batch_pipeline()`, `run_openai_batch_pipeline()`, `write_openai_batch_file()`

Examples

```
# Continuing the example from llm_submit_pairs_multi_batch():
# After submitting multiple batches, resume polling and combine the results.
## Not run:
# Suppose `outdir` is the directory where batch files were written and
# `jobs` is the list of job metadata returned by llm_submit_pairs_multi_batch().

results <- llm_resume_multi_batches(
  jobs          = jobs,
  output_dir    = outdir,
  interval_seconds = 60,
  per_job_delay  = 2,
  write_results_csv = TRUE,
  keep_jsonl     = FALSE,
  write_registry = TRUE,
  verbose        = TRUE,
  write_combined_csv = TRUE
)
```

```
# The combined results are available in the `combined` element
print(results$combined)

## 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). Vertex, Together.ai, and Ollama are supported only for live comparisons in this series.

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 is in the GPT-5 series (including `gpt-5`, `gpt-5-mini`, `gpt-5.6-sol`, `gpt-5.6-terra`, `gpt-5.6-luna`, and date-stamped GPT-5.x variants), and
 - either `include_thoughts = TRUE` **or** a reasoning effort is supplied in ... (for GPT-5, `reasoning = "none"` maps to "minimal").

Sampling defaults: For OpenAI, omitted temperature and `top_p` values are not added to the request, so the model/provider defaults apply. Reasoning modes that do not support sampling parameters continue to require them to be `NULL`.

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), omitted temperature and `top_p` values use the model/provider defaults.
- 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_tokens`.
- 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 meta-data 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

<code>pairs</code>	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.
<code>backend</code>	Character scalar; one of "openai", "anthropic", or "gemini". Matching is case-insensitive. If "vertex" is supplied, this function aborts explicitly because Vertex batch mode is not implemented in this series.
<code>model</code>	Character scalar model name to use for the batch job. - For "openai", use models like "gpt-4.1", "gpt-5", "gpt-5-mini", "gpt-5.6-sol", "gpt-5.6-terra", or "gpt-5.6-luna" (including date-stamped GPT-5.x versions where available). - For "anthropic", use provider names like "claude-sonnet-4-5" or date-stamped versions like "claude-sonnet-4-5-20250929". - For "gemini", use names like "gemini-3.5-flash-lite".
<code>trait_name</code>	A short name for the trait being evaluated (e.g. "overall_quality").
<code>trait_description</code>	A human-readable description of the trait.
<code>prompt_template</code>	A prompt template created by <code>set_prompt_template()</code> or a compatible character scalar.
<code>include_thoughts</code>	Logical; whether to request and parse model "thoughts" (where supported). - For OpenAI GPT-5 series, setting this to TRUE defaults to the responses endpoint. - For Anthropic, setting this to TRUE implies reasoning = "enabled" (unless overridden) and sets temperature = 1.
<code>include_raw</code>	Logical; whether to include raw provider responses in the result (where supported by backends).
<code>...</code>	Additional arguments passed through to the backend-specific <code>run*_batch_pipeline()</code> 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, service_tier, etc. For Anthropic, this may include reasoning, max_tokens, temperature, or thinking_budget_tokens.

For Gemini, this may include `thinking_level`, `temperature`, `top_p`, `top_k`, `max_output_tokens`, and `service_tier`.

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.
- `failed_attempts`: a tibble of failed attempts captured during normalization (empty when no failures are observed).

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

See Also

[llm_download_batch_results\(\)](#), [llm_submit_pairs_multi_batch\(\)](#)

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_create_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [openai_upload_batch_file\(\)](#), [run_anthropic_batch_pipeline\(\)](#), [run_gemini_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#), [write_openai_batch_file\(\)](#)

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")
tpl <- set_prompt_template()

# OpenAI batch
batch_openai <- llm_submit_pairs_batch(
  pairs = pairs,
```

```

    backend      = "openai",
    model         = "gpt-5-mini",
    trait_name    = td$name,
    trait_description = td$description,
    prompt_template = tmpl,
    include_thoughts = FALSE,
    service_tier  = "flex"
  )
  res_openai <- llm_download_batch_results(batch_openai)

  # Anthropic batch
  batch_anthropic <- llm_submit_pairs_batch(
    pairs      = pairs,
    backend    = "anthropic",
    model      = "claude-sonnet-4-5",
    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-3.5-flash-lite",
    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)

```

llm_submit_pairs_multi_batch

Multi-batch submission and polling wrappers

Description

These functions provide higher-level wrappers around the existing provider-specific batch APIs in **pairwiseLLM**. They allow a large tibble of pairwise comparisons to be automatically split into multiple batch jobs, submitted concurrently (without polling), recorded in a registry for safe resumption, and later polled until completion and merged into a single results data frame. They do not modify any of the underlying API functions such as [run_openai_batch_pipeline\(\)](#) or [run_anthropic_batch_pipeline\(\)](#), but orchestrate these calls to support resilient multi-batch workflows.

Usage

```
llm_submit_pairs_multi_batch(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  backend = c("openai", "anthropic", "gemini"),
  batch_size = NULL,
  n_segments = NULL,
  output_dir = tempfile("llm_multi_batch_"),
  write_registry = FALSE,
  keep_jsonl = TRUE,
  verbose = FALSE,
  ...,
  openai_max_retries = 3
)
```

Arguments

<code>pairs</code>	A tibble of pairs with columns ID1, text1, ID2, text2. Typically produced by make_pairs() , sample_pairs() , and randomize_pair_order() .
<code>model</code>	Model identifier for the chosen backend. Passed through to the corresponding <code>run*_batch_pipeline()</code> function.
<code>trait_name</code> , <code>trait_description</code> , <code>prompt_template</code>	Parameters forwarded to run_openai_batch_pipeline() , run_anthropic_batch_pipeline() , or run_gemini_batch_pipeline() . See those functions for details.
<code>backend</code>	One of "openai", "anthropic", or "gemini". Determines which provider pipeline is used for each batch. If "vertex" is supplied, this function aborts explicitly because Vertex batch mode is not implemented in this series.
<code>batch_size</code>	Integer giving the maximum number of pairs per batch. Exactly one of <code>batch_size</code> or <code>n_segments</code> must be supplied; if <code>batch_size</code> is supplied, the number of segments is computed as <code>ceiling(nrow(pairs) / batch_size)</code> . The final segment may contain fewer pairs than <code>batch_size</code> .
<code>n_segments</code>	Integer giving the number of segments to create. Exactly one of <code>batch_size</code> or <code>n_segments</code> must be supplied; if <code>n_segments</code> is supplied, each segment contains approximately <code>nrow(pairs) / n_segments</code> pairs. The last segment may be smaller.
<code>output_dir</code>	Directory in which to write all batch files, including the .jsonl input/output files, the optional registry CSV, and (if requested) parsed results CSVs. A temporary directory is created by default.
<code>write_registry</code>	Logical; if TRUE, a CSV registry of batch jobs is written to <code>file.path(output_dir, "jobs_registry.csv")</code> . The registry can be reloaded with readr::read_csv() and passed to llm_resume_multi_batches() for polling and resumption. If FALSE, the registry is returned in memory only.

<code>keep_jsonl</code>	Logical; if FALSE, the <code>.jsonl</code> input and output files for each batch will be deleted after the job results have been parsed in <code>llm_resume_multi_batches()</code> . Since the provider APIs require file paths, the files are always created during submission; this option controls whether to retain them on disk after completion.
<code>verbose</code>	Logical; if TRUE, prints progress messages during batch submission. Messages include the segment index, the number of pairs in each segment, the chosen provider, and confirmation that the batch has been created along with the input file path. Defaults to FALSE.
<code>...</code>	Additional arguments passed through to the provider-specific <code>run_*_batch_pipeline()</code> function. These may include arguments such as <code>include_thoughts</code> , <code>reasoning</code> , <code>include_raw</code> , <code>temperature</code> , etc.
<code>openai_max_retries</code>	Integer giving the maximum number of times to retry the initial OpenAI batch submission when a transient HTTP 5xx error occurs. When creating a segment on the OpenAI backend, <code>run_openai_batch_pipeline()</code> internally uploads the JSONL file and creates the batch. On rare occasions this call can return a 500 error; specifying a positive value here (e.g. 3) will automatically retry the submission up to that many times. Between retries, the function sleeps for a brief period proportional to the current attempt. Defaults to 3.

Value

A list with two elements: `jobs`, a list of per-batch metadata (similar to the example in the advanced vignette), and `registry`, a tibble summarising all jobs. The registry contains columns `segment_index`, `provider`, `model`, `batch_id`, `batch_input_path`, `batch_output_path`, `csv_path`, `pairs_path`, `done`, and `results` (initialized to NULL). If `write_registry` is TRUE, the tibble is also written to disk as `jobs_registry.csv`.

`llm_submit_pairs_multi_batch()`

Splits a tibble of comparison pairs into chunks and submits one batch per chunk using the appropriate provider pipeline. Each batch is created with `poll = FALSE`, so the function returns immediately after the batch jobs have been created. Metadata for each batch—including the `batch_id`, provider type, and input/output file paths—is collected and (optionally) written to a CSV registry for later resumption.

See Also

`llm_submit_pairs_batch()`, `llm_download_batch_results()`

Other batch backends: `anthropic_create_batch()`, `anthropic_download_batch_results()`, `anthropic_get_batch()`, `anthropic_poll_batch_until_complete()`, `build_anthropic_batch_requests()`, `build_gemini_batch_requests()`, `build_openai_batch_requests()`, `gemini_create_batch()`, `gemini_download_batch_results()`, `gemini_get_batch()`, `gemini_poll_batch_until_complete()`, `llm_download_batch_results()`, `llm_resume_multi_batches()`, `llm_submit_pairs_batch()`, `openai_create_batch()`, `openai_download_batch_output()`, `openai_get_batch()`, `openai_poll_batch_until_complete()`, `openai_upload_batch_file()`, `run_anthropic_batch_pipeline()`, `run_gemini_batch_pipeline()`, `run_openai_batch_pipeline()`, `write_openai_batch_file()`

Examples

```

# Example: split a small set of pairs into five segments, submit
# them to the Gemini backend, and then poll and combine the results.
# Requires a funded API key and internet access.
## Not run:
# Construct ten random pairs from the example writing samples
set.seed(123)
pairs <- sample_pairs(example_writing_samples, n_pairs = 10)

# Directory to store batch files and results
outdir <- tempfile("multi_batch_example_")

# Submit the pairs in five batches. We write the registry to disk
# and print progress messages as each batch is created.
job_info <- llm_submit_pairs_multi_batch(
  pairs          = pairs,
  model          = "gemini-3.5-flash-lite",
  trait_name     = "writing_quality",
  trait_description = "Which text shows better writing quality?",
  n_segments     = 5,
  output_dir     = outdir,
  write_registry = TRUE,
  verbose        = TRUE
)

# Resume polling until all batches complete. The per-batch and
# combined results are written to CSV files, the registry is
# refreshed on disk, and progress messages are printed.
results <- llm_resume_multi_batches(
  jobs           = job_info$jobs,
  output_dir     = outdir,
  interval_seconds = 60,
  per_job_delay  = 2,
  write_results_csv = TRUE,
  keep_jsonl     = FALSE,
  write_registry  = TRUE,
  verbose        = TRUE,
  write_combined_csv = TRUE
)

# Access the combined results tibble
head(results$combined)

## End(Not run)

```

Description

Load an adaptive session from disk.

Usage

```
load_adaptive_session(session_dir)
```

Arguments

`session_dir` Directory containing session artifacts.

Details

Restores a persisted Adaptive state and revalidates basic invariants such as schema version, required state fields, and index ranges in `step_log`. If per-refit item logs are found on disk, they are loaded into `state$item_log` and persistence is marked as enabled. Resume uses strict schema validation for canonical logs; incompatible saved schemas abort with explicit errors.

Value

An `adaptive_state` object ready for resume.

See Also

[save_adaptive_session\(\)](#), [validate_session_dir\(\)](#), [adaptive_rank_resume\(\)](#)

Other adaptive persistence: [save_adaptive_session\(\)](#), [validate_session_dir\(\)](#)

Examples

```
dir <- tempfile("pwillm-session-")
state <- adaptive_rank_start(c("a", "b", "c"), seed = 1)
save_adaptive_session(state, dir, overwrite = TRUE)
restored <- load_adaptive_session(dir)
summarize_adaptive(restored)
```

`make_adaptive_judge_llm`

Build an LLM judge function for adaptive ranking

Description

Creates a judge function compatible with [adaptive_rank_run_live\(\)](#) by wrapping [llm_compare_pair\(\)](#) and converting provider responses into adaptive binary outcomes (Y in $\{0, 1\}$).

Usage

```
make_adaptive_judge_llm(
  backend = c("openai", "anthropic", "gemini", "vertex", "together", "ollama"),
  model,
  trait = "overall_quality",
  trait_name = NULL,
  trait_description = NULL,
  prompt_template = set_prompt_template(),
  endpoint = "chat.completions",
  api_key = NULL,
  include_raw = FALSE,
  text_col = "text",
  judge_args = list()
)
```

Arguments

backend	Backend passed to <code>llm_compare_pair()</code> . Choices are "openai", "anthropic", "gemini", "vertex", "together", and "ollama". Default is "openai".
model	Model identifier passed to <code>llm_compare_pair()</code> . Required.
trait	Built-in trait key used when no custom trait is supplied. Ignored when both <code>trait_name</code> and <code>trait_description</code> are supplied. Default is "overall_quality".
trait_name	Optional custom trait display name.
trait_description	Optional custom trait definition.
prompt_template	Prompt template string. Defaults to <code>set_prompt_template()</code> .
endpoint	Endpoint family passed to <code>llm_compare_pair()</code> . Only used when <code>backend = "openai"</code> ; choices are "chat.completions" and "responses". Default is "chat.completions". Ignored for other backends.
api_key	Optional API key passed to <code>llm_compare_pair()</code> .
include_raw	Logical; forwarded to <code>llm_compare_pair()</code> . Default is FALSE.
text_col	Name of the text column expected in adaptive item rows. Default is "text".
judge_args	Named list of additional fixed arguments forwarded to <code>llm_compare_pair()</code> . Use this for provider-specific controls such as reasoning, service_tier, temperature, top_p, logprobs, host, or include_thoughts. Default is <code>list()</code> .

Details

The returned function has signature `judge(A, B, state, ...)` and enforces the adaptive transactional contract: it returns `is_valid = TRUE` with `Y` in `{0,1}` when the model response identifies one of the two presented items, and returns `is_valid = FALSE` otherwise. In addition to the required contract fields, the returned judge preserves canonical audit metadata from the first `llm_compare_pair()` row, including backend/model provenance, status/error fields, token counts, and a serialized `raw_response_json` payload when available.

Model configuration is split into:

- fixed build-time options via `judge_args`,
- per-run overrides via `judge_call_args` in `adaptive_rank()`,
- optional per-step overrides via `...` passed through `adaptive_rank_run_live()`.

Collectively this supports all `llm_compare_pair()` options, including backend-specific parameters such as OpenAI reasoning and `service_tier`.

Value

A function `judge(A, B, state, ...)` returning a list with required fields `is_valid`, `Y`, and `invalid_reason`, plus optional canonical audit fields such as `judge_backend`, `judge_model`, `judge_endpoint`, `llm_status_code`, `llm_error_message`, `llm_custom_id`, `prompt_tokens`, `completion_tokens`, `total_tokens`, and `raw_response_json`.

See Also

`adaptive_rank()`, `adaptive_rank_run_live()`, `llm_compare_pair()`

Other adaptive ranking: `adaptive_rank()`, `adaptive_rank_resume()`, `adaptive_rank_run_live()`, `adaptive_rank_start()`, `summarize_adaptive()`

Examples

```
judge <- make_adaptive_judge_llm(
  backend = "openai",
  model = "gpt-5.1",
  endpoint = "responses",
  judge_args = list(
    reasoning = "low",
    service_tier = "flex",
    include_thoughts = FALSE
  )
)
```

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

See Also

[read_samples_df\(\)](#), [read_samples_dir\(\)](#)

Other pairing and data: [alternate_pair_order\(\)](#), [randomize_pair_order\(\)](#), [read_samples_df\(\)](#), [read_samples_dir\(\)](#), [sample_pairs\(\)](#), [sample_reverse_pairs\(\)](#)

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-small13.2:24b", "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 <code>getOption("pairwiseLLM.ollama_host", "http://127.0.0.1:11434")</code> .
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; forwarded to Ollama's think request field. The function parses a thinking field from the response whenever one is present.
num_ctx	Integer; context window to use via <code>options\$num_ctx</code> . The default is 8192L.
include_raw	Logical; if TRUE, adds a list-column <code>raw_response</code> containing the parsed JSON body returned by Ollama (or NULL on parse failure). This is useful for debugging.
...	Reserved for future extensions. When <code>pair_uid</code> is supplied via ..., it is used verbatim as <code>custom_id</code> .

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-small13.2:24b", "qwen3:32b", "gemma3:27b").

Sampling and context length are controlled as follows:

- Sampling parameters are omitted so the local model default applies.
- 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-small13.2:24b` 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` – stable ID for the pair (`pair_uid` if supplied via ...; otherwise `"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.

Other live backends: `anthropic_compare_pair_live()`, `check_llm_api_keys()`, `gemini_compare_pair_live()`, `llm_compare_pair()`, `openai_compare_pair_live()`, `submit_anthropic_pairs_live()`, `submit_gemini_pairs_live()`, `submit_llm_pairs()`, `submit_ollama_pairs_live()`, `submit_openai_pairs_live()`, `submit_together_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `vertex_compare_pair_live()`

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-small13.2:24b",
  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-5.6-sol").
trait_name	Short label for the trait (e.g. "Overall Quality").

<code>trait_description</code>	Full-text definition of the trait.
<code>prompt_template</code>	Prompt template string.
<code>endpoint</code>	Which OpenAI endpoint to use: "chat.completions" or "responses".
<code>tag_prefix</code>	Prefix for the better-sample tag.
<code>tag_suffix</code>	Suffix for the better-sample tag.
<code>api_key</code>	Optional OpenAI API key.
<code>include_raw</code>	Logical; if TRUE, adds a <code>raw_response</code> column.
<code>...</code>	Additional OpenAI parameters, for example <code>temperature</code> , <code>top_p</code> , <code>logprobs</code> , <code>reasoning</code> , <code>service_tier</code> , <code>max_output_tokens</code> , <code>pair_uid</code> , and (optionally) <code>include_thoughts</code> . <code>max_output_tokens</code> must be a positive integer and is supported only by the Responses endpoint. When <code>pair_uid</code> is supplied, it is used verbatim as <code>custom_id</code> . 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 <code>thoughts</code> column by setting <code>endpoint = "responses"</code> , a non-"none" reasoning effort, and <code>include_thoughts = TRUE</code> .

Details

It supports both the Chat Completions endpoint ("/v1/chat/completions") and the Responses endpoint ("/v1/responses", for example gpt-5.6 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.

Sampling defaults: If `temperature` or `top_p` is not provided in `...`, the corresponding field is omitted so the model/provider default applies. Reasoning modes that do not support sampling parameters continue to require them to be `NULL`.

Value

A tibble with one row and columns:

custom_id Stable ID for the pair (`pair_uid` if supplied via `...`; otherwise "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.

See Also

[check_llm_api_keys\(\)](#), [llm_compare_pair\(\)](#)

Other live backends: [anthropic_compare_pair_live\(\)](#), [check_llm_api_keys\(\)](#), [gemini_compare_pair_live\(\)](#), [llm_compare_pair\(\)](#), [ollama_compare_pair_live\(\)](#), [submit_anthropic_pairs_live\(\)](#), [submit_gemini_pairs_live\(\)](#), [submit_llm_pairs\(\)](#), [submit_ollama_pairs_live\(\)](#), [submit_openai_pairs_live\(\)](#), [submit_together_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [vertex_compare_pair_live\(\)](#)

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?"
)

# 2. Reasoning comparison using GPT-5.6 Sol
res_reasoning <- openai_compare_pair_live(
  ID1 = "A", text1 = "Text A...",
  ID2 = "B", text2 = "Text B...",
  model = "gpt-5.6-sol",
  trait_name = "clarity",
  trait_description = "Which text is clearer?",
  endpoint = "responses",
  include_thoughts = TRUE,
  reasoning = "high",
  service_tier = "flex"
)
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. <code>"/v1/chat/completions"</code> or <code>"/v1/responses"</code> .
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.

See Also

[llm_submit_pairs_batch\(\)](#), [llm_download_batch_results\(\)](#)

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [openai_upload_batch_file\(\)](#), [run_anthropic_batch_pipeline\(\)](#), [run_gemini_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#), [write_openai_batch_file\(\)](#)

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.

See Also

```
llm_submit_pairs_batch(), llm_download_batch_results()
```

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#),

```
llm_submit_pairs_multi_batch(), openai_create_batch(), openai_get_batch(), openai_poll_batch_until_comp
openai_upload_batch_file(), run_anthropic_batch_pipeline(), run_gemini_batch_pipeline(),
run_openai_batch_pipeline(), write_openai_batch_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	<i>Retrieve an OpenAI batch</i>
------------------	---------------------------------

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.

See Also

```
llm_submit_pairs_batch(), llm_download_batch_results()
```

Other batch backends: `anthropic_create_batch()`, `anthropic_download_batch_results()`, `anthropic_get_batch()`, `anthropic_poll_batch_until_complete()`, `build_anthropic_batch_requests()`, `build_gemini_batch_requests()`, `build_openai_batch_requests()`, `gemini_create_batch()`, `gemini_download_batch_results()`, `gemini_get_batch()`, `gemini_poll_batch_until_complete()`, `llm_download_batch_results()`, `llm_resume_multi_batches()`, `llm_submit_pairs_batch()`, `llm_submit_pairs_multi_batch()`, `openai_create_batch()`, `openai_download_batch_output()`, `openai_poll_batch_until_complete()`, `openai_upload_batch_file()`, `run_anthropic_batch_pipeline()`, `run_gemini_batch_pipeline()`, `run_openai_batch_pipeline()`, `write_openai_batch_file()`

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

<code>batch_id</code>	The batch ID.
<code>interval_seconds</code>	Number of seconds to wait between polling attempts.
<code>timeout_seconds</code>	Maximum total time to wait in seconds before giving up.
<code>max_attempts</code>	Maximum number of polling attempts. This is mainly useful for testing; default is Inf.
<code>api_key</code>	Optional OpenAI API key.
<code>verbose</code>	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()`.

See Also

`llm_submit_pairs_batch()`, `llm_download_batch_results()`

Other batch backends: `anthropic_create_batch()`, `anthropic_download_batch_results()`, `anthropic_get_batch()`, `anthropic_poll_batch_until_complete()`, `build_anthropic_batch_requests()`, `build_gemini_batch_requests()`, `build_openai_batch_requests()`, `gemini_create_batch()`, `gemini_download_batch_results()`, `gemini_get_batch()`, `gemini_poll_batch_until_complete()`, `llm_download_batch_results()`, `llm_resume_multi_batches()`, `llm_submit_pairs_batch()`, `llm_submit_pairs_multi_batch()`, `openai_create_batch()`, `openai_download_batch_output()`, `openai_get_batch()`, `openai_upload_batch_file()`, `run_anthropic_batch_pipeline()`, `run_gemini_batch_pipeline()`, `run_openai_batch_pipeline()`, `write_openai_batch_file()`

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.

See Also

[llm_submit_pairs_batch\(\)](#), [llm_download_batch_results\(\)](#)

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_create_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [run_anthropic_batch_pipeline\(\)](#), [run_gemini_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#), [write_openai_batch_file\(\)](#)

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).

See Also

[parse_openai_batch_output\(\)](#), [parse_gemini_batch_output\(\)](#)

Other result normalization: [parse_gemini_batch_output\(\)](#), [parse_openai_batch_output\(\)](#)

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

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

See Also

[parse_openai_batch_output\(\)](#), [parse_anthropic_batch_output\(\)](#)

Other result normalization: [parse_anthropic_batch_output\(\)](#), [parse_openai_batch_output\(\)](#)

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.

See Also

[parse_anthropic_batch_output\(\)](#), [parse_gemini_batch_output\(\)](#)

Other result normalization: [parse_anthropic_batch_output\(\)](#), [parse_gemini_batch_output\(\)](#)

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)
```

print.adaptive_state *Print an adaptive state summary.*

Description

S3 method for printing adaptive_state objects.

Usage

```
## S3 method for class 'adaptive_state'
print(x, ...)
```

Arguments

x	An adaptive_state object.
...	Unused.

Value

x, invisibly.

See Also[summarize_adaptive\(\)](#)**Examples**

```
state <- adaptive_rank_start(c("a", "b", "c"), seed = 1)
print(state)
```

```
print.pairwiseLLM_cost_estimate
```

```
Print a pairwiseLLM cost estimate
```

Description

Prints a compact, human-readable summary of an object returned by [estimate_llm_pairs_cost](#). The print method reports the backend, model, pilot/remaining pair counts, estimated token totals, and both the expected and budget cost estimates.

Usage

```
## S3 method for class 'pairwiseLLM_cost_estimate'
print(x, ...)
```

Arguments

x	An object of class "pairwiseLLM_cost_estimate", typically returned by estimate_llm_pairs_cost .
...	Unused. Included for method compatibility.

Value

x, invisibly.

Examples

```
## Not run:
data("example_writing_samples", package = "pairwiseLLM")
pairs <- example_writing_samples |>
  make_pairs() |>
  sample_pairs(n_pairs = 50, seed = 123)

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

est <- estimate_llm_pairs_cost(
  pairs = pairs,
  backend = "openai",
  model = "gpt-4.1",
```

```

    endpoint = "chat.completions",
    trait_name = td$name,
    trait_description = td$description,
    prompt_template = tpl,
    mode = "batch",
    batch_discount = 0.5,
    n_test = 10,
    cost_per_million_input = 0.15,
    cost_per_million_output = 0.60
  )

  est

  ## End(Not run)

```

randomize_pair_order	<i>Randomly assign samples to positions SAMPLE_1 and SAMPLE_2</i>
----------------------	---

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

<code>pairs</code>	A data frame or tibble with columns ID1, text1, ID2, and text2. Typically created by make_pairs (optionally followed by sample_pairs).
<code>seed</code>	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.

Other pairing and data: [alternate_pair_order\(\)](#), [make_pairs\(\)](#), [read_samples_df\(\)](#), [read_samples_dir\(\)](#), [sample_pairs\(\)](#), [sample_reverse_pairs\(\)](#)

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	<i>Read writing samples from a data frame</i>
-----------------	---

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.

See Also

`read_samples_dir()`, `make_pairs()`

Other pairing and data: `alternate_pair_order()`, `make_pairs()`, `randomize_pair_order()`, `read_samples_dir()`, `sample_pairs()`, `sample_reverse_pairs()`

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)
```

<code>read_samples_dir</code>	<i>Read writing samples from a directory of .txt files</i>
-------------------------------	--

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

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

Value

A tibble with columns:

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

See Also

```
read_samples_df(), make_pairs()
```

Other pairing and data: `alternate_pair_order()`, `make_pairs()`, `randomize_pair_order()`, `read_samples_df()`, `sample_pairs()`, `sample_reverse_pairs()`

Examples

```
# Create a temporary directory with sample text files
samples_dir <- tempfile()
dir.create(samples_dir)

writeLines("This is sample A.", file.path(samples_dir, "A.txt"))
writeLines("This is sample B.", file.path(samples_dir, "B.txt"))

# Read samples into a tibble
samples <- read_samples_dir(samples_dir)

samples
```

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.

See Also

`trait_description()`, `set_prompt_template()`

Other prompts and traits: `build_prompt()`, `get_prompt_template()`, `list_prompt_templates()`, `remove_prompt_template()`, `set_prompt_template()`, `trait_description()`

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

<code>name</code>	Character scalar; name of the template to remove.
<code>quiet</code>	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](#)

Other prompts and traits: [build_prompt\(\)](#), [get_prompt_template\(\)](#), [list_prompt_templates\(\)](#), [register_prompt_template\(\)](#), [set_prompt_template\(\)](#), [trait_description\(\)](#)

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

<code>pairs</code>	Tibble or data frame with at least columns ID1, text1, ID2, text2.
<code>model</code>	Anthropic model name (for example "claude-sonnet-4-5").
<code>trait_name</code>	Trait name to pass to build_anthropic_batch_requests .
<code>trait_description</code>	Trait description to pass to build_anthropic_batch_requests .
<code>prompt_template</code>	Prompt template string, typically from set_prompt_template .
<code>reasoning</code>	Character scalar; one of "none" or "enabled". See details above for how <code>include_thoughts</code> influences this value and how temperature defaults are derived.
<code>include_thoughts</code>	Logical; if TRUE, requests extended thinking from Claude (by setting <code>reasoning = "enabled"</code> when necessary) and parses any thinking blocks into a <code>thoughts</code> column in the batch results.
<code>batch_input_path</code>	Path to write the JSON file containing the requests object. Defaults to a temporary file with suffix ".json".
<code>batch_output_path</code>	Path to write the downloaded .jsonl results if <code>poll = TRUE</code> . Defaults to a temporary file with suffix ".jsonl".
<code>poll</code>	Logical; if TRUE, the function will poll the batch until it reaches <code>processing_status = "ended"</code> 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 <code>Sys.getenv("ANTHROPIC_API_KEY")</code> .
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_completion](#).
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):
 - Omitted temperature and top_p values are not sent, so the model/provider defaults apply.
 - 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, batches without extended thinking use model-default sampling. 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.

See Also

[llm_submit_pairs_batch\(\)](#), [llm_download_batch_results\(\)](#)

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_create_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [openai_upload_batch_file\(\)](#), [run_gemini_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#), [write_openai_batch_file\(\)](#)

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 = "low",

```

```

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,
service_tier = "standard",
...
)

```

Arguments

<code>pairs</code>	Tibble/data frame of pairs.
<code>model</code>	Gemini model name, for example "gemini-3.5-flash-lite" or "gemini-3-flash-preview".
<code>trait_name</code>	Trait name.
<code>trait_description</code>	Trait description.
<code>prompt_template</code>	Prompt template string.
<code>thinking_level</code>	One of "minimal", "low", "medium", or "high". This controls the maximum depth of internal reasoning for Gemini batch requests via <code>generationConfig\$thinkingConfig\$thinkingLevel</code> . - For Gemini 3 Flash models (for example "gemini-3-flash-preview"), "minimal" is supported and is passed through as "minimal". - For models not matched by the package's Gemini 3 Flash-name detector (for example "gemini-3.5-flash-lite"), "minimal" is not supported. - For backward compatibility with earlier Gemini 3 Pro usage, "low" maps to "low" and both "medium" and "high" map to "high". "Medium" currently behaves like "High".
<code>batch_input_path</code>	Path where the batch input JSON should be written.
<code>batch_output_path</code>	Path where the batch output JSONL should be written (only used if <code>poll = TRUE</code>).
<code>poll</code>	Logical; if TRUE, poll the batch until completion and parse results. If FALSE, only create the batch and write the input file.
<code>interval_seconds</code>	Polling interval when <code>poll = TRUE</code> .
<code>timeout_seconds</code>	Maximum total waiting time when <code>poll = TRUE</code> .
<code>api_key</code>	Optional Gemini API key.
<code>api_version</code>	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).
service_tier	Gemini Developer API service tier forwarded to build_gemini_batch_requests . Use "standard" (default) or NULL for provider default behavior, or "flex" / "priority" to request the documented Gemini service tier.
...	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).

See Also

[llm_submit_pairs_batch\(\)](#), [llm_download_batch_results\(\)](#)

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_create_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [openai_upload_batch_file\(\)](#), [run_anthropic_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#), [write_openai_batch_file\(\)](#)

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-3.5-flash-lite",
  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

# Gemini 3 Flash example (minimal thinking)
res_flash <- run_gemini_batch_pipeline(
  pairs          = pairs,
  model          = "gemini-3-flash-preview",
  trait_name     = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  thinking_level = "minimal",
  poll          = TRUE,
  include_thoughts = FALSE
)

res_flash$results

## 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

<code>pairs</code>	Tibble of pairs with at least ID1, text1, ID2, text2. Typically produced by make_pairs() , sample_pairs() , and randomize_pair_order() .
<code>model</code>	OpenAI model name (e.g. "gpt-4.1", "gpt-5.6-sol").
<code>trait_name</code>	Trait name to pass to build_openai_batch_requests() .
<code>trait_description</code>	Trait description to pass to build_openai_batch_requests() .
<code>prompt_template</code>	Prompt template string, typically from set_prompt_template() .
<code>include_thoughts</code>	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, <code>include_thoughts = TRUE</code> causes the responses endpoint to be selected automatically.
<code>include_raw</code>	Logical; if <code>TRUE</code> , attaches the raw model response as a list-column <code>raw_response</code> in the parsed results.
<code>endpoint</code>	One of <code>"chat.completions"</code> or <code>"responses"</code> . If <code>NULL</code> (or omitted), it is chosen automatically as described above.
<code>batch_input_path</code>	Path to write the batch input <code>.jsonl</code> file. Defaults to a temporary file.
<code>batch_output_path</code>	Path to write the batch output <code>.jsonl</code> file if <code>poll = TRUE</code> . Defaults to a temporary file.
<code>poll</code>	Logical; if <code>TRUE</code> , the function will poll the batch until it reaches a terminal status using <code>openai_poll_batch_until_complete()</code> and then download and parse the output. If <code>FALSE</code> , it stops after creating the batch and returns without polling or parsing.
<code>interval_seconds</code>	Polling interval in seconds (used when <code>poll = TRUE</code>).
<code>timeout_seconds</code>	Maximum total time in seconds for polling before giving up (used when <code>poll = TRUE</code>).
<code>max_attempts</code>	Maximum number of polling attempts (primarily useful for testing).
<code>metadata</code>	Optional named list of metadata key–value pairs to pass to <code>openai_create_batch()</code> .
<code>api_key</code>	Optional OpenAI API key. Defaults to <code>Sys.getenv("OPENAI_API_KEY")</code> .
<code>...</code>	Additional arguments passed through to <code>build_openai_batch_requests()</code> , e.g. <code>temperature</code> , <code>top_p</code> , <code>logprobs</code> , <code>reasoning</code> .

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` or GPT-5 reasoning is requested, the `"responses"` endpoint is used and a default reasoning effort of `"low"` is applied for GPT-5 series models 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`.

See Also

`llm_submit_pairs_batch()`, `llm_resume_multi_batches()`

Other batch backends: `anthropic_create_batch()`, `anthropic_download_batch_results()`, `anthropic_get_batch()`, `anthropic_poll_batch_until_complete()`, `build_anthropic_batch_requests()`, `build_gemini_batch_requests()`, `build_openai_batch_requests()`, `gemini_create_batch()`, `gemini_download_batch_results()`, `gemini_get_batch()`, `gemini_poll_batch_until_complete()`, `llm_download_batch_results()`, `llm_resume_multi_batches()`, `llm_submit_pairs_batch()`, `llm_submit_pairs_multi_batch()`, `openai_create_batch()`, `openai_download_batch_output()`, `openai_get_batch()`, `openai_poll_batch_until_complete()`, `openai_upload_batch_file()`, `run_anthropic_batch_pipeline()`, `run_gemini_batch_pipeline()`, `write_openai_batch_file()`

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	<i>Randomly sample pairs of writing samples</i>
--------------	---

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

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

Value

A tibble containing the sampled rows of pairs.

See Also

[read_samples_df\(\)](#), [read_samples_dir\(\)](#)

Other pairing and data: [alternate_pair_order\(\)](#), [make_pairs\(\)](#), [randomize_pair_order\(\)](#), [read_samples_df\(\)](#), [read_samples_dir\(\)](#), [sample_reverse_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. For values strictly between 0 and 1, the row count is round(nrow(pairs) * reverse_pct).
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).

See Also

[read_samples_df\(\)](#), [read_samples_dir\(\)](#)

Other pairing and data: [alternate_pair_order\(\)](#), [make_pairs\(\)](#), [randomize_pair_order\(\)](#), [read_samples_df\(\)](#), [read_samples_dir\(\)](#), [sample_pairs\(\)](#)

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)
```

save_adaptive_session *Save an adaptive session to disk.*

Description

Save an adaptive session to disk.

Usage

```
save_adaptive_session(state, session_dir, overwrite = FALSE)
```

Arguments

state	Adaptive state.
session_dir	Directory to write session artifacts.
overwrite	Logical; overwrite existing artifacts.

Details

Saves canonical Adaptive artifacts under session_dir: state.rds, step_log.rds, round_log.rds, metadata.rds, optional btl_fit.rds, and optional per-refit item log files when state\$config\$persist_item_log is TRUE. Writes are atomic at file level to reduce partial-write risk. Persisted step_log/round_log files keep the full canonical schemas, so resume preserves expanded audit fields without recomputation.

Value

The session_dir path, invisibly.

See Also

[validate_session_dir\(\)](#), [load_adaptive_session\(\)](#)

Other adaptive persistence: [load_adaptive_session\(\)](#), [validate_session_dir\(\)](#)

Examples

```
dir <- tempfile("pwl1m-session-")
state <- adaptive_rank_start(c("a", "b", "c"), seed = 1)
save_adaptive_session(state, dir, overwrite = TRUE)
```

set_prompt_template	<i>Get or set a prompt template for pairwise comparisons</i>
---------------------	--

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.

See Also

[trait_description\(\)](#), [build_prompt\(\)](#)

Other prompts and traits: [build_prompt\(\)](#), [get_prompt_template\(\)](#), [list_prompt_templates\(\)](#), [register_prompt_template\(\)](#), [remove_prompt_template\(\)](#), [trait_description\(\)](#)

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 robust 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 collects the results.

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,
  save_path = NULL,
  parallel = FALSE,
  workers = 1,
  ...
)

```

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 <code>Sys.getenv("ANTHROPIC_API_KEY")</code> .
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).
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. Note: Raw responses are not saved to the incremental CSV file.
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.
save_path	Character string; optional file path (e.g., "output.csv") to save results incrementally. If the file exists, the function reads it to identify and skip pairs that have already been processed (resume mode). Requires the readr package.
parallel	Logical; if TRUE, enables parallel processing using <code>future.apply</code> . Requires the future and future.apply packages.
workers	Integer; the number of parallel workers (threads) to use if parallel = TRUE. Defaults to 1. Guidance: Anthropic rate limits vary significantly by tier. Start conservatively (at most 2 workers) to avoid HTTP 429 errors and respect shared check-farm resources.
...	Additional Anthropic parameters (for example temperature, top_p, max_tokens) passed on to anthropic_compare_pair_live . When pair_uid is supplied via ..., it is used verbatim as custom_id.

Details

This function offers:

- **Parallel Processing:** Uses the future package to process multiple pairs simultaneously.
- **Incremental Saving:** Writes results to a CSV file as they complete. If the process is interrupted, re-running the function with the same save_path will automatically skip pairs that were already successfully processed.
- **Error Separation:** Returns valid results and failed pairs separately, making it easier to debug or retry specific failures.

Temperature and reasoning behaviour

Temperature and extended-thinking behaviour are controlled by [anthropic_compare_pair_live](#):

- When reasoning = "none" (no extended thinking), omitted temperature and top_p values use model/provider defaults.
- 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", omitted sampling values use model defaults.

Value

A list containing three elements:

results A tibble with one row per successfully processed pair.

failed_pairs A tibble containing the rows from pairs that failed to process (due to API errors or timeouts), along with an error_message column.

failed_attempts A tibble of attempt-level failures (retries, timeouts, parse errors, invalid winners), separate from observed outcomes.

See Also

[check_llm_api_keys\(\)](#), [llm_compare_pair\(\)](#)

Other live backends: [anthropic_compare_pair_live\(\)](#), [check_llm_api_keys\(\)](#), [gemini_compare_pair_live\(\)](#), [llm_compare_pair\(\)](#), [ollama_compare_pair_live\(\)](#), [openai_compare_pair_live\(\)](#), [submit_gemini_pairs_live\(\)](#), [submit_llm_pairs\(\)](#), [submit_ollama_pairs_live\(\)](#), [submit_openai_pairs_live\(\)](#), [submit_together_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [vertex_compare_pair_live\(\)](#)

Examples

```
## Not run:
# Requires ANTHROPIC_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()

# 1. Sequential execution with incremental saving
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",
  save_path      = "results_seq.csv"
)

# 2. Parallel execution (faster)
res_par <- submit_anthropic_pairs_live(
  pairs          = pairs,
  model          = "claude-sonnet-4-5",
  trait_name     = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  save_path      = "results_par.csv",
  parallel       = TRUE,
  workers        = 2
)

# Inspect results
head(res_par$results)

# Check for failures
if (nrow(res_par$failed_pairs) > 0) {
  message("Some pairs failed:")
  print(res_par$failed_pairs)
}

## End(Not run)

```

submit_gemini_pairs_live

Live Google Gemini comparisons for a tibble of pairs

Description

This is a robust row-wise wrapper around `gemini_compare_pair_live()`. It takes a tibble of pairs (ID1 / text1 / ID2 / text2), submits each pair to the Google Gemini API, and collects the results.

Usage

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

Arguments

<code>pairs</code>	Tibble/data frame with columns ID1, text1, ID2, text2.
<code>model</code>	Gemini model name (e.g. "gemini-3.5-flash-lite" or "gemini-3-flash-preview").
<code>trait_name</code>	Trait name.
<code>trait_description</code>	Trait description.
<code>prompt_template</code>	Prompt template string, typically from set_prompt_template() .
<code>api_key</code>	Optional Gemini API key.
<code>thinking_level</code>	Default "low"; see gemini_compare_pair_live() . For Gemini 3 Flash models, "minimal" is also supported (e.g., <code>thinking_level = "minimal"</code> with <code>model = "gemini-3-flash-preview"</code>).
<code>temperature</code>	Optional numeric temperature; forwarded to gemini_compare_pair_live() . See Gemini docs; if NULL (default), the model uses its own default.
<code>top_p</code>	Optional numeric; forwarded to gemini_compare_pair_live() .
<code>top_k</code>	Optional numeric; forwarded to gemini_compare_pair_live() .
<code>max_output_tokens</code>	Optional integer; forwarded to gemini_compare_pair_live() .

service_tier	Gemini Developer API service tier forwarded to <code>gemini_compare_pair_live()</code> . Use "standard" (default) or NULL for provider default behavior, or "flex" / "priority" to request the documented Gemini service tier.
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; if TRUE, each row of the returned tibble will include a raw_response list-column with the parsed JSON body. Note: Raw responses are not saved to the incremental CSV file.
include_thoughts	Logical; if TRUE, requests explicit reasoning output from Gemini and stores it in the thoughts column of the result, mirroring <code>gemini_compare_pair_live()</code> .
save_path	Character string; optional file path (e.g., "output.csv") to save results incrementally. If the file exists, the function reads it to identify and skip pairs that have already been processed (resume mode). Requires the readr package.
parallel	Logical; if TRUE, enables parallel processing using future.apply. Requires the future and future.apply packages.
workers	Integer; the number of parallel workers (threads) to use if parallel = TRUE. Defaults to 1. Guidance: Use no more than 2 workers to avoid HTTP 429 errors and respect shared check-farm resources.
...	Reserved for future extensions; passed through to <code>gemini_compare_pair_live()</code> (but thinking_budget is ignored there).

Details

This function offers:

- **Parallel Processing:** Uses the future package to process multiple pairs simultaneously.
- **Incremental Saving:** Writes results to a CSV file as they complete. If the process is interrupted, re-running the function with the same save_path will automatically skip pairs that were already successfully processed.
- **Error Separation:** Returns valid results and failed pairs separately, making it easier to debug or retry specific failures.

Value

A list containing three elements:

results A tibble with one row per successfully processed pair.

failed_pairs A tibble containing the rows from pairs that failed to process (due to API errors or timeouts), along with an error_message column.

failed_attempts A tibble of attempt-level failures (retries, timeouts, parse errors, invalid winners), separate from observed outcomes.

See Also

`check_llm_api_keys()`, `llm_compare_pair()`

Other live backends: `anthropic_compare_pair_live()`, `check_llm_api_keys()`, `gemini_compare_pair_live()`, `llm_compare_pair()`, `ollama_compare_pair_live()`, `openai_compare_pair_live()`, `submit_anthropic_pairs_live()`, `submit_llm_pairs()`, `submit_ollama_pairs_live()`, `submit_openai_pairs_live()`, `submit_together_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `vertex_compare_pair_live()`

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()

# 1. Sequential execution with incremental saving
res_seq <- submit_gemini_pairs_live(
  pairs          = pairs,
  model          = "gemini-3.5-flash-lite",
  trait_name     = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  save_path      = "results_gemini_seq.csv"
)

# 2. Parallel execution (faster)
res_par <- submit_gemini_pairs_live(
  pairs          = pairs,
  model          = "gemini-3.5-flash-lite",
  trait_name     = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  save_path      = "results_gemini_par.csv",
  parallel       = TRUE,
  workers        = 2
)

# 3. Gemini 3 Flash example (minimal thinking)
res_flash <- submit_gemini_pairs_live(
  pairs          = pairs,
  model          = "gemini-3-flash-preview",
```

```

    trait_name      = td$name,
    trait_description = td$description,
    prompt_template = tmpl,
    thinking_level   = "minimal",
    save_path        = "results_gemini_flash.csv"
  )

# Inspect results
head(res_par$results)

## End(Not run)

```

submit_llm_pairs	<i>Backend-agnostic live comparisons for a tibble of pairs</i>
------------------	--

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 aggregates the results.

Usage

```

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

```

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", use Anthropic model names such as "claude-sonnet-4-5". For "gemini" and "vertex", use the corresponding Gemini model names (for example "gemini-3.5-flash-lite" or "gemini-2.5-flash"). For "together", use Together.ai model identifiers such as "deepseek-ai/DeepSeek-V4-Flash-0731". 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", "vertex", "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", "vertex", "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).
save_path	Character string; optional file path (e.g., "output.csv") to save results incrementally. If the file exists, the function reads it to identify and skip pairs that have already been processed (resume mode). Supported by all backends.
parallel	Logical; if TRUE, enables parallel processing using <code>future.apply</code> . Requires the future package. Supported by all backends (though defaults may vary).
workers	Integer; the number of parallel workers (threads) to use if parallel = TRUE. Defaults to 1.
...	Additional backend-specific parameters. For "openai" these are forwarded to submit_openai_pairs_live() and typically include temperature, top_p, logprobs, reasoning, service_tier, and include_thoughts. For "anthropic", "gemini", and "vertex", they are forwarded to submit_anthropic_pairs_live() , submit_gemini_pairs_live() , or submit_vertex_pairs_live() and may

include options such as `max_output_tokens`, `include_thoughts`, `service_tier`, `thinking_level`, 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

This function supports parallel processing, incremental saving, and resume capability for the "openai", "anthropic", "gemini", "vertex", "together", and "ollama" backends.

At present, the following backends are implemented:

- "openai" → `submit_openai_pairs_live()`
- "anthropic" → `submit_anthropic_pairs_live()`
- "gemini" → `submit_gemini_pairs_live()`
- "vertex" → `submit_vertex_pairs_live()`
- "together" → `submit_together_pairs_live()`
- "ollama" → `submit_ollama_pairs_live()`

Value

A list containing:

results A tibble with one row per successfully processed pair.

failed_pairs A tibble containing rows that failed to process (for supported backends).

failed_attempts A tibble containing normalized failure records (invalid winners, parse failures, HTTP/timeouts) suitable for debugging.

See Also

- `submit_openai_pairs_live()`, `submit_anthropic_pairs_live()`, `submit_gemini_pairs_live()`, `submit_vertex_pairs_live()`, `submit_together_pairs_live()`, and `submit_ollama_pairs_live()` for backend-specific implementations.

Other live backends: `anthropic_compare_pair_live()`, `check_llm_api_keys()`, `gemini_compare_pair_live()`, `llm_compare_pair()`, `ollama_compare_pair_live()`, `openai_compare_pair_live()`, `submit_anthropic_pairs_live()`, `submit_gemini_pairs_live()`, `submit_ollama_pairs_live()`, `submit_openai_pairs_live()`, `submit_together_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `vertex_compare_pair_live()`

Examples

```
## Not run:
# Requires an API key for the chosen cloud 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()

    # Parallel execution with OpenAI (requires future package)
    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",
      parallel        = TRUE,
      workers         = 2,
      save_path       = "results_openai.csv"
    )

    # Live comparisons using a local Ollama backend with incremental saving
    res_ollama <- submit_llm_pairs(
      pairs          = pairs,
      model           = "mistral-small13.2:24b",
      trait_name      = td$name,
      trait_description = td$description,
      prompt_template = tmpl,
      backend         = "ollama",
      save_path       = "results_ollama.csv",
      verbose         = TRUE
    )

    # GPT-5 live comparisons with service tier
    res_gpt5 <- submit_llm_pairs(
      pairs          = pairs,
      model           = "gpt-5",
      trait_name      = td$name,
      trait_description = td$description,
      backend         = "openai",
      endpoint        = "responses",
      reasoning       = "none",
      service_tier    = "flex"
    )

    res_ollama$results

    ## End(Not run)

```

submit_ollama_pairs_live

Live Ollama comparisons for a tibble of pairs

Description

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

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,
  save_path = NULL,
  parallel = FALSE,
  workers = 1,
  ...
)
```

Arguments

<code>pairs</code>	Tibble or data frame with at least columns ID1, text1, ID2, text2. Typically created by <code>make_pairs()</code> , <code>sample_pairs()</code> , and <code>randomize_pair_order()</code> .
<code>model</code>	Ollama model name (for example "mistral-small3.2:24b", "qwen3:32b", "gemma3:27b").
<code>trait_name</code>	Trait name to pass to <code>ollama_compare_pair_live()</code> .
<code>trait_description</code>	Trait description to pass to <code>ollama_compare_pair_live()</code> .
<code>prompt_template</code>	Prompt template string, typically from <code>set_prompt_template()</code> .
<code>host</code>	Base URL of the Ollama server. Defaults to the option <code>getOption("pairwiseLLM.ollama_host", "http://127.0.0.1:11434")</code> .
<code>verbose</code>	Logical; if TRUE, prints status, timing, and result summaries.
<code>status_every</code>	Integer; print status and timing for every <code>status_every</code> -th pair. Defaults to 1 (every pair). Errors are always printed.
<code>progress</code>	Logical; if TRUE, shows a textual progress bar.
<code>think</code>	Logical; see <code>ollama_compare_pair_live()</code> for behavior.
<code>num_ctx</code>	Integer; context window to use via <code>options\$num_ctx</code> . 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. Note: Raw responses are not saved to the incremental CSV file.
save_path	Character string; optional file path (e.g., "output.csv") to save results incrementally. If the file exists, the function reads it to identify and skip pairs that have already been processed (resume mode). Requires the readr package.
parallel	Logical; if TRUE, enables parallel processing using future.apply. Requires the future and future.apply packages. Defaults to FALSE.
workers	Integer; the number of parallel workers (threads) to use if parallel = TRUE. Defaults to 1.
...	Reserved for future extensions and forwarded to ollama_compare_pair_live().

Details

This function offers:

- **Incremental Saving:** Writes results to a CSV file as they complete. If the process is interrupted, re-running the function with the same save_path will automatically skip pairs that were already successfully processed.
- **Parallel Processing:** Uses the future package to process multiple pairs simultaneously. **Note:** Since Ollama typically runs locally on the GPU, parallel processing may degrade performance or cause out-of-memory errors unless the hardware can handle concurrent requests. Defaults are set to sequential processing.

Sampling and context length are controlled as follows:

- Sampling parameters are omitted so the local model default applies.
- 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.

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-small13.2:24b`).

Value

A list containing three elements:

results A tibble with one row per successfully processed pair.

failed_pairs A tibble containing the rows from pairs that failed to process (due to API errors or timeouts), along with an error_message column.

failed_attempts A tibble of attempt-level failures (retries, timeouts, parse errors, invalid winners), separate from observed outcomes.

See Also

- [ollama_compare_pair_live\(\)](#) for single-pair Ollama comparisons.
- [submit_llm_pairs\(\)](#) for backend-agnostic comparisons over tibbles of pairs.

Other live backends: [anthropic_compare_pair_live\(\)](#), [check_llm_api_keys\(\)](#), [gemini_compare_pair_live\(\)](#), [llm_compare_pair\(\)](#), [ollama_compare_pair_live\(\)](#), [openai_compare_pair_live\(\)](#), [submit_anthropic_pairs_live\(\)](#), [submit_gemini_pairs_live\(\)](#), [submit_llm_pairs\(\)](#), [submit_openai_pairs_live\(\)](#), [submit_together_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [vertex_compare_pair_live\(\)](#)

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 with incremental saving
res_mistral <- submit_ollama_pairs_live(
  pairs          = pairs,
  model          = "mistral-small13.2:24b",
  trait_name     = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  save_path      = "ollama_results.csv",
  verbose        = TRUE
)

# Access results
res_mistral$results

## End(Not run)
```

submit_openai_pairs_live

Live OpenAI comparisons for a tibble of pairs

Description

This is a robust 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 collects the results.

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,
  save_path = NULL,
  parallel = FALSE,
  workers = 1,
  ...
)
```

Arguments

<code>pairs</code>	Tibble or data frame with at least columns ID1, text1, ID2, text2. Typically created by make_pairs , sample_pairs , and randomize_pair_order .
<code>model</code>	OpenAI model name (for example "gpt-4.1", "gpt-5.1").
<code>trait_name</code>	Trait name to pass to <code>openai_compare_pair_live</code> .
<code>trait_description</code>	Trait description to pass to <code>openai_compare_pair_live</code> .
<code>prompt_template</code>	Prompt template string, typically from set_prompt_template .
<code>endpoint</code>	Which OpenAI endpoint to target. One of "chat.completions" or "responses".
<code>api_key</code>	Optional OpenAI API key.
<code>verbose</code>	Logical; if TRUE, prints status, timing, and result summaries.
<code>status_every</code>	Integer; print status / timing for every <code>status_every</code> -th pair. Defaults to 1 (every pair).
<code>progress</code>	Logical; if TRUE, shows a textual progress bar.
<code>include_raw</code>	Logical; if TRUE, each row of the returned tibble will include a <code>raw_response</code> list-column with the parsed JSON body from OpenAI. Note: Raw responses are not saved to the incremental CSV file.
<code>save_path</code>	Character string; optional file path (e.g., "output.csv") to save results incrementally. If the file exists, the function reads it to identify and skip pairs that have already been processed (resume mode). Requires the <code>readr</code> package.
<code>parallel</code>	Logical; if TRUE, enables parallel processing using <code>future.apply</code> . Requires the <code>future</code> and <code>future.apply</code> packages.

workers	Integer; the number of parallel workers (threads) to use if parallel = TRUE. Defaults to 1. Guidance: A value between 4 and 8 is usually safe. Setting this too high (e.g., >20) may trigger OpenAI rate limit errors (HTTP 429) depending on your usage tier.
...	Additional OpenAI parameters (temperature, top_p, logprobs, reasoning, service_tier, and so on) passed on to openai_compare_pair_live.

Details

This function improves upon simple looping by offering:

- **Parallel Processing:** Uses the future package to process multiple pairs simultaneously.
- **Incremental Saving:** Writes results to a CSV file as they complete. If the process is interrupted, re-running the function with the same save_path will automatically skip pairs that were already successfully processed.
- **Error Separation:** Returns valid results and failed pairs separately, making it easier to debug or retry specific failures.

Value

A list containing three elements:

results A tibble with one row per successfully processed pair and columns such as better_id, better_sample, thoughts, and content. See [openai_compare_pair_live](#) for details.

failed_pairs A tibble containing the rows from pairs that failed to process (due to API errors or timeouts), along with an error_message column. These can be easily re-submitted.

failed_attempts A tibble of attempt-level failures (retries, timeouts, parse errors, invalid winners), separate from observed outcomes.

See Also

[check_llm_api_keys\(\)](#), [llm_compare_pair\(\)](#)

Other live backends: [anthropic_compare_pair_live\(\)](#), [check_llm_api_keys\(\)](#), [gemini_compare_pair_live\(\)](#), [llm_compare_pair\(\)](#), [ollama_compare_pair_live\(\)](#), [openai_compare_pair_live\(\)](#), [submit_anthropic_pairs_live\(\)](#), [submit_gemini_pairs_live\(\)](#), [submit_llm_pairs\(\)](#), [submit_ollama_pairs_live\(\)](#), [submit_together_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [vertex_compare_pair_live\(\)](#)

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 = 10, seed = 123) |>
  randomize_pair_order(seed = 456)
```

```

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

# 1. Sequential execution with incremental saving
# If interrupted, running this again will resume progress.
res_seq <- submit_openai_pairs_live(
  pairs          = pairs,
  model          = "gpt-4.1",
  trait_name     = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  save_path      = "results_seq.csv"
)

# 2. Parallel execution (faster)
# Note: On Windows, this opens background R sessions.
res_par <- submit_openai_pairs_live(
  pairs          = pairs,
  model          = "gpt-4.1",
  trait_name     = td$name,
  trait_description = td$description,
  save_path      = "results_par.csv",
  parallel       = TRUE,
  workers        = 2
)

# Inspect results
head(res_par$results)

# Check for failures
if (nrow(res_par$failed_pairs) > 0) {
  message("Some pairs failed:")
  print(res_par$failed_pairs)
}

# 3. GPT-5 live run with service tier (Responses endpoint)
res_gpt5 <- submit_openai_pairs_live(
  pairs          = pairs,
  model          = "gpt-5",
  trait_name     = td$name,
  trait_description = td$description,
  endpoint       = "responses",
  reasoning      = "none",
  service_tier   = "priority"
)

## End(Not run)

```

submit_together_pairs_live

Live Together.ai comparisons for a tibble of pairs

Description

submit_together_pairs_live() is a robust 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 collects the results.

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,
  save_path = NULL,
  parallel = FALSE,
  workers = 1,
  ...
)
```

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 the dated tested identifier "deepseek-ai/DeepSeek-V4-Flash-073". Check the provider's current serverless catalog before use.
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 <code>.together_api_key()</code> .
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).
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. Note: Raw responses are not saved to the incremental CSV file.
save_path	Character string; optional file path (e.g., "output.csv") to save results incrementally. If the file exists, the function reads it to identify and skip pairs that have already been processed (resume mode). Requires the readr package.
parallel	Logical; if TRUE, enables parallel processing using future.apply. Requires the future and future.apply packages.
workers	Integer; the number of parallel workers (threads) to use if parallel = TRUE. Defaults to 1. Guidance: Together.ai rate limits vary by usage tier. Start with no more than 2 workers to avoid HTTP 429 errors and respect shared check-farm resources.
...	Additional Together.ai parameters, such as temperature, top_p, or other provider-specific options. These are forwarded to together_compare_pair_live() .

Details

This function improves upon simple looping by offering:

- **Parallel Processing:** Uses the future package to process multiple pairs simultaneously.
- **Incremental Saving:** Writes results to a CSV file as they complete. If the process is interrupted, re-running the function with the same save_path will automatically skip pairs that were already successfully processed.
- **Error Separation:** Returns valid results and failed pairs separately, making it easier to debug or retry specific failures.

Value

A list containing three elements:

results A tibble with one row per successfully processed pair and columns such as better_id, better_sample, thoughts, and content.

failed_pairs A tibble containing the rows from pairs that failed to process (due to API errors or timeouts), along with an error_message column. These can be easily re-submitted.

failed_attempts A tibble of attempt-level failures (retries, timeouts, parse errors, invalid winners), separate from observed outcomes.

See Also

[check_llm_api_keys\(\)](#), [llm_compare_pair\(\)](#)

Other live backends: [anthropic_compare_pair_live\(\)](#), [check_llm_api_keys\(\)](#), [gemini_compare_pair_live\(\)](#), [llm_compare_pair\(\)](#), [ollama_compare_pair_live\(\)](#), [openai_compare_pair_live\(\)](#), [submit_anthropic_pairs_live\(\)](#), [submit_gemini_pairs_live\(\)](#), [submit_llm_pairs\(\)](#), [submit_ollama_pairs_live\(\)](#), [submit_openai_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [vertex_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 = 10, seed = 123) |>
  randomize_pair_order(seed = 456)

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

# 1. Sequential execution with incremental saving
# If interrupted, running this again will resume progress.
res_seq <- submit_together_pairs_live(
  pairs          = pairs,
  model          = "deepseek-ai/DeepSeek-V4-Flash-0731",
  trait_name     = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  save_path      = "results_seq.csv"
)

# 2. Parallel execution (faster)
# Note: On Windows, this opens background R sessions.
res_par <- submit_together_pairs_live(
  pairs          = pairs,
  model          = "deepseek-ai/DeepSeek-V4-Flash-0731",
  trait_name     = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  save_path      = "results_par.csv",
  parallel       = TRUE,
  workers        = 2
)

# Inspect results
head(res_par$results)

# Check for failures
if (nrow(res_par$failed_pairs) > 0) {
  message("Some pairs failed:")
  print(res_par$failed_pairs)
}

## End(Not run)

```

submit_vertex_pairs_live

Live Vertex AI Gemini comparisons for a tibble of pairs

Description

This is a row-wise wrapper around `vertex_compare_pair_live()`. It takes a tibble of pairs (ID1 / text1 / ID2 / text2), submits each pair to the Vertex AI Gemini API, and collects the results with optional incremental saving and resume support.

Usage

```
submit_vertex_pairs_live(
  pairs,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  api_key = NULL,
  temperature = NULL,
  top_p = NULL,
  top_k = NULL,
  max_output_tokens = NULL,
  thinking_level = NULL,
  thinking_budget = NULL,
  service_tier = "standard",
  api_version = "v1",
  verbose = TRUE,
  status_every = 1L,
  progress = TRUE,
  include_raw = FALSE,
  include_thoughts = FALSE,
  save_path = NULL,
  parallel = FALSE,
  workers = 1,
  ...
)
```

Arguments

<code>pairs</code>	Tibble/data frame with columns ID1, text1, ID2, text2.
<code>model</code>	Vertex Gemini model name (for example "gemini-2.5-flash").
<code>trait_name</code>	Trait name.
<code>trait_description</code>	Trait description.

prompt_template	Prompt template string, typically from set_prompt_template() .
api_key	Optional Vertex API key.
temperature	Optional numeric temperature; forwarded to vertex_compare_pair_live() .
top_p	Optional numeric; forwarded to vertex_compare_pair_live() .
top_k	Optional numeric; forwarded to vertex_compare_pair_live() .
max_output_tokens	Optional integer; forwarded to vertex_compare_pair_live() .
thinking_level	Optional Gemini 3 thinking level; forwarded to vertex_compare_pair_live() .
thinking_budget	Optional integer; forwarded to vertex_compare_pair_live() .
service_tier	Vertex AI service tier forwarded to vertex_compare_pair_live() .
api_version	API version; default "v1".
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; if TRUE, each row of the returned tibble will include a raw_response list-column with the parsed JSON body.
include_thoughts	Logical; if TRUE, requests explicit reasoning output and stores it in the thoughts column of the result.
save_path	Character string; optional file path to save results incrementally. If the file exists, the function reads it to identify and skip pairs that have already been processed (resume mode). Requires the readr package.
parallel	Logical; if TRUE, enables parallel processing using future.apply. Requires the future and future.apply packages.
workers	Integer; the number of parallel workers to use if parallel = TRUE. Defaults to 1.
...	Reserved for future extensions; passed through to vertex_compare_pair_live() .

Value

A list containing three elements:

results A tibble with one row per successfully processed pair.

failed_pairs A tibble containing the rows from pairs that failed to process, along with an error_message column.

failed_attempts A tibble of attempt-level failures separate from observed outcomes.

See Also

[check_llm_api_keys\(\)](#), [llm_compare_pair\(\)](#)

Other live backends: [anthropic_compare_pair_live\(\)](#), [check_llm_api_keys\(\)](#), [gemini_compare_pair_live\(\)](#), [llm_compare_pair\(\)](#), [ollama_compare_pair_live\(\)](#), [openai_compare_pair_live\(\)](#), [submit_anthropic_pairs_live\(\)](#), [submit_gemini_pairs_live\(\)](#), [submit_llm_pairs\(\)](#), [submit_ollama_pairs_live\(\)](#), [submit_openai_pairs_live\(\)](#), [submit_together_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [vertex_compare_pair_live\(\)](#)

Examples

```
## Not run:
data("example_writing_samples", package = "pairwiseLLM")
pairs <- make_pairs(example_writing_samples[1:3, ])
td <- trait_description("overall_quality")
out <- submit_vertex_pairs_live(
  pairs = pairs,
  model = "gemini-3.8-flash",
  trait_name = td$name,
  trait_description = td$description,
  thinking_level = "low",
  parallel = FALSE
)
out$failed_pairs

## End(Not run)
```

summarize_adaptive	<i>Summarize an adaptive state.</i>
--------------------	-------------------------------------

Description

Summarize an adaptive state.

Usage

```
summarize_adaptive(state)
```

Arguments

state	Adaptive state.
-------	-----------------

Details

Returns a compact run-level summary from canonical logs: attempted steps, committed comparisons, refit count, and last stop decision/reason. This is a pure view and does not recompute model quantities.

Value

A one-row tibble with columns `n_items`, `steps_attempted`, `committed_pairs`, `n_refits`, `last_stop_decision`, and `last_stop_reason`.

See Also

[adaptive_get_logs\(\)](#), [base::print\(\)](#)

Other adaptive ranking: [adaptive_rank\(\)](#), [adaptive_rank_resume\(\)](#), [adaptive_rank_run_live\(\)](#), [adaptive_rank_start\(\)](#), [make_adaptive_judge_llm\(\)](#)

Examples

```
state <- adaptive_rank_start(c("a", "b", "c"), seed = 1)
summarize_adaptive(state)
```

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.

See Also

`build_bt_data()`, `fit_bt_model()`

Other frequentist models: `build_bt_data()`, `build_elo_data()`, `fit_bt_model()`, `fit_elo_model()`

Examples

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

if (requireNamespace("sirt", quietly = TRUE)) {
  fit1 <- fit_bt_model(bt, engine = "sirt")
  summarize_bt_fit(fit1)
}
if (requireNamespace("BradleyTerry2", quietly = TRUE)) {
  fit2 <- fit_bt_model(bt, engine = "BradleyTerry2")
  summarize_bt_fit(fit2)
}
```

summarize_items

Summarize adaptive items

Description

Build an item-level diagnostics summary from the canonical item logs. This is a pure view and does not recompute posterior quantities or exposure metrics.

Usage

```
summarize_items(
  state,
  posterior = NULL,
  refit = NULL,
  bind = FALSE,
  top_n = NULL,
  sort_by = NULL,
  include_optional = TRUE
)
```

Arguments

<code>state</code>	An <code>adaptive_state</code> or list containing adaptive logs.
<code>posterior</code>	Optional <code>item_log_list</code> (list of item log tables) or an item log table. When <code>NULL</code> , uses <code>state\$logs\$item_log_list</code> when available.
<code>refit</code>	Optional refit index. When <code>NULL</code> , the most recent refit is returned; when set, the <code>k</code> -th refit is returned.

<code>bind</code>	Logical; when TRUE, stack all refits into a single table.
<code>top_n</code>	Optional positive integer; return only the top n rows after sorting.
<code>sort_by</code>	Column used for sorting. When NULL, the first available column in <code>c("rank_link", "rank_raw", "rank_mean", "theta_link_eap", "theta_raw_eap", "theta_mean", "theta_sd", "degree", "pos_A_rate")</code> is used.
<code>include_optional</code>	Logical; include optional diagnostic columns.

Details

Rank percentiles are computed from the per-draw induced ranks (lower is better). Rank uncertainty grows when draws disagree on the ordering. Degree and position exposure metrics summarize how frequently each item was shown and whether it appeared as the first option (A position). When `refit = NULL`, the most recent refit is returned; when `refit = k`, the k-th refit is returned. When `bind = TRUE`, all refits are stacked into a single table and `refit` must be NULL.

Value

A tibble with one row per item per refit. Columns reflect the supplied item-log schema. Standalone and legacy logs use fields such as `ID`, `theta_mean`, `rank_mean`, `deg`, and `posA_prop`. Current adaptive logs use `item_id`, `theta_raw_eap`, `theta_raw_sd`, `rank_raw`, `degree`, `pos_count_A`, and `pos_count_B`; linking logs can also include `theta_link_eap`, `theta_link_sd`, and `rank_link`. The function is a view and does not rename these fields.

In standalone logs, `rank_mean` is the posterior mean of per-draw induced ranks. In current adaptive logs, `rank_raw` is the rank of the EAP scores. They are not the same statistic, although a request to sort a current log by the legacy name `"rank_mean"` maps to `"rank_raw"` for compatibility. Similarly, legacy sorting requests for `"theta_mean"` and `"theta_sd"` map to `"theta_raw_eap"` and `"theta_raw_sd"`. When `include_optional = FALSE`, optional columns such as repeated-pair or adjacency diagnostics are dropped if present.

See Also

[adaptive_get_logs\(\)](#), [adaptive_step_log\(\)](#)

Other adaptive logs: [adaptive_get_logs\(\)](#), [adaptive_item_log\(\)](#), [adaptive_results_history\(\)](#), [adaptive_round_log\(\)](#), [adaptive_step_log\(\)](#), [summarize_refits\(\)](#)

Examples

```
# summarize_items() expects an item_log_list (list of per-refit item tables).
# This example constructs a minimal logs object that matches what adaptive runs emit.
```

```
item_log_1 <- tibble::tibble(
  refit_id = 1L,
  ID = c("A", "B", "C"),
  theta_mean = c(0.4, 0.1, -0.2),
  theta_sd = c(0.2, 0.3, 0.25),
  rank_mean = c(1.2, 2.1, 2.7),
  degree = c(10L, 9L, 8L),
  pos_A_rate = c(0.55, 0.50, 0.48)
```

```

)

item_log_2 <- dplyr::mutate(
  item_log_1,
  refit_id = 2L,
  theta_mean = theta_mean + c(0.1, 0.05, 0.02),
  rank_mean = rank_mean + c(-0.1, 0.0, 0.1)
)

logs <- list(item_log_list = list(item_log_1, item_log_2))

# Default returns the most recent refit:
summarize_items(logs)

# Select a specific refit:
summarize_items(logs, refit = 1)

# Stack all refits into one table:
summarize_items(logs, bind = TRUE)

# Sort and take the top rows:
summarize_items(logs, sort_by = "rank_mean", top_n = 2)

```

summarize_refits	<i>Summarize adaptive refits</i>
------------------	----------------------------------

Description

Build a thin per-refit diagnostics summary from the adaptive round log. This is a pure view over round_log and does not recompute posterior quantities or stop metrics.

Usage

```
summarize_refits(state, last_n = NULL, include_optional = TRUE)
```

Arguments

state	An adaptive_state or list containing adaptive logs.
last_n	Optional positive integer; return only the last n rows.
include_optional	Logical; include optional diagnostic columns.

Details

The round log is the canonical stop-audit trail. This summary is a direct view over round_log with no recomputation.

Key fields include:

- identity: refit_id, round_id_at_refit, step_id_at_refit
- run scale: total_pairs_done, new_pairs_since_last_refit, new_active_pairs_since_last_refit, new_probe_pairs_since_last_refit, new_total_cross_pairs_since_last_refit, n_unique_pairs_seen
- candidate health: proposed_pairs_mode, starve_rate_since_last_refit, fallback_rate_since_last_refit, fallback_used_mode, starvation_reason_mode
- identifiability/quota adaptation: global_identified, global_identified_reliability_min, global_identified_rank_corr_min, long_quota_raw, long_quota_effective, long_quota_removed, realloc_to_mid, realloc_to_local
- diagnostics/stopping: diagnostics_pass, divergences, max_rhat, min_ess_bulk, ess_bulk_required, reliability_EAP, rho_theta, delta_sd_theta, rho_rank, stop_decision, stop_reason
- report-only uncertainty metrics: ci95_theta_width_*, near_tie_adj_*, cov_trace_theta, top20_boundary_entropy_*, nn_diff_sd_*

Value

A tibble with one row per refit (canonical round_log schema).

See Also

[adaptive_get_logs\(\)](#), [adaptive_step_log\(\)](#)

Other adaptive logs: [adaptive_get_logs\(\)](#), [adaptive_item_log\(\)](#), [adaptive_results_history\(\)](#), [adaptive_round_log\(\)](#), [adaptive_step_log\(\)](#), [summarize_items\(\)](#)

Examples

```
# These summaries work on either an adaptive_state or a plain list of logs.
logs <- list(
  round_log = tibble::tibble(
    refit_id = 1:2,
    round_id_at_refit = c(1L, 2L),
    step_id_at_refit = c(10L, 20L),
    new_pairs_since_last_refit = c(50L, 50L),
    total_pairs_done = c(50L, 100L),
    divergences = c(0L, 0L),
    max_rhat = c(1.01, 1.00),
    min_ess_bulk = c(800, 900),
    stop_decision = c(NA, TRUE),
    stop_reason = c(NA_character_, "btl_converged")
  )
)

# Full per-refit view:
summarize_refits(logs)

# Only the most recent refit row:
summarize_refits(logs, last_n = 1)

# Drop optional diagnostics if you want a compact core summary:
summarize_refits(logs, include_optional = FALSE)
```

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 the dated tested identifier "deepseek-ai/DeepSeek-V4-Flash-073"). Check the provider's current serverless catalog before use.
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>".

<code>api_key</code>	Optional Together.ai API key. If NULL or empty, the helper falls back to the <code>TOGETHER_API_KEY</code> environment variable via <code>.together_api_key()</code> .
<code>include_raw</code>	Logical; if TRUE, adds a list-column <code>raw_response</code> containing the parsed JSON body returned by Together.ai (or NULL on parse failure). This is useful for debugging parsing problems.
<code>...</code>	Additional Together.ai parameters, typically including <code>temperature</code> , <code>top_p</code> , and provider-specific options. These are passed through to the JSON request body as top-level fields. Omitted sampling controls use model/provider defaults. When <code>pair_uid</code> is supplied via <code>...</code> , it is used verbatim as <code>custom_id</code> .

Details

For models 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.

Models that do not use `<think>` tags return NA in `thoughts`, and their full output appears in `content`. Model identifiers are forwarded to the provider; see `vignette("model-compatibility")` for dated tested examples.

If `temperature` or `top_p` is omitted from `...`, the corresponding field is not sent and the model/provider default applies. Explicit values are passed through unchanged.

Value

A tibble with one row and columns:

custom_id Stable ID for the pair (`pair_uid` if supplied via `...`; otherwise `"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 from `<think>...</think>` blocks, when present.

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.

See Also

```
check_llm_api_keys(), llm_compare_pair()

Other live backends: anthropic_compare_pair_live(), check_llm_api_keys(), gemini_compare_pair_live(),
llm_compare_pair(), ollama_compare_pair_live(), openai_compare_pair_live(), submit_anthropic_pairs_live(),
submit_gemini_pairs_live(), submit_llm_pairs(), submit_ollama_pairs_live(), submit_openai_pairs_live(),
submit_together_pairs_live(), submit_vertex_pairs_live(), vertex_compare_pair_live()
```

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()

# Dated tested Together serverless configuration
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-V4-Flash-0731",
  trait_name    = td$name,
  trait_description = td$description,
  prompt_template = tmpl
)

res_deepseek$better_id
res_deepseek$thoughts

## End(Not run)
```

trait_description	<i>Get a trait name and description for prompts</i>
-------------------	---

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", "IRRC"),
  custom_name = NULL,
  custom_description = NULL
)
```

Arguments

name	Character identifier for a built-in trait. One of "overall_quality", "organization", or "IRRC". "IRRC" is case-sensitive and returns an overall-writing rubric spanning prompt task, development of explanation, organization, and language use. 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.

See Also

[set_prompt_template\(\)](#), [build_prompt\(\)](#)

Other prompts and traits: [build_prompt\(\)](#), [get_prompt_template\(\)](#), [list_prompt_templates\(\)](#), [register_prompt_template\(\)](#), [remove_prompt_template\(\)](#), [set_prompt_template\(\)](#)

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
```

validate_session_dir	<i>Validate an adaptive session directory.</i>
----------------------	--

Description

Validate an adaptive session directory.

Usage

```
validate_session_dir(session_dir)
```

Arguments

session_dir Directory containing session artifacts.

Details

Verifies that required session artifacts exist and that serialized logs match canonical schemas for `step_log` and `round_log`. This check is intended as a preflight for [load_adaptive_session\(\)](#) and enforces the canonical adaptive session metadata shape. Validation is strict: added/removed/reordered columns in persisted logs are treated as schema incompatibilities and abort resume.

Value

A metadata list containing at least `schema_version`, `package_version`, and `n_items`.

See Also

[save_adaptive_session\(\)](#), [load_adaptive_session\(\)](#)

Other adaptive persistence: [load_adaptive_session\(\)](#), [save_adaptive_session\(\)](#)

Examples

```
dir <- tempfile("pwl1m-session-")
state <- adaptive_rank_start(c("a", "b", "c"), seed = 1)
save_adaptive_session(state, dir, overwrite = TRUE)
validate_session_dir(dir)
```

vertex_compare_pair_live

Live Vertex AI Gemini comparison for a single pair of samples

Description

This function sends a single pairwise comparison prompt to the Vertex AI Gemini API using the express-mode REST generateContent endpoint and parses the result into a one-row tibble that mirrors the structure used by the other live backends.

Usage

```
vertex_compare_pair_live(
  ID1,
  text1,
  ID2,
  text2,
  model,
  trait_name,
  trait_description,
  prompt_template = set_prompt_template(),
  api_key = NULL,
  temperature = NULL,
  top_p = NULL,
  top_k = NULL,
  max_output_tokens = NULL,
  thinking_level = NULL,
  thinking_budget = NULL,
  service_tier = "standard",
  api_version = "v1",
  include_raw = FALSE,
  include_thoughts = FALSE,
  pair_uid = NULL,
  ...
)
```

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	Vertex Gemini model identifier. You may supply either a short model name such as "gemini-2.5-flash" or the fully qualified publisher model resource "publishers/google/models/gemini-2.5-flash".

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 <code>set_prompt_template()</code> .
api_key	Optional Vertex API key (defaults to <code>Sys.getenv("VERTEX_API_KEY")</code>).
temperature	Optional numeric temperature. If NULL (default), the parameter is omitted and Vertex uses the provider default.
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.
thinking_level	Optional Gemini 3 thinking level. Supported public values are "minimal", "low", "medium", and "high", but exact support varies by Gemini 3 model family. This parameter is only valid for Gemini 3 and later models. Do not supply it together with thinking_budget.
thinking_budget	Optional thinking budget in tokens. If supplied, the request includes <code>generationConfig\$thinkingConfig</code> . For models earlier than Gemini 3, this is the supported control surface. Do not supply it together with thinking_level on Gemini 3 models.
service_tier	Vertex AI service tier. Use "standard" (default) or NULL for provider default behavior. Use "flex" to request the documented shared flex headers or "priority" to request the documented dedicated request-type header.
api_version	API version to use, default "v1".
include_raw	Logical; if TRUE, the returned tibble includes a <code>raw_response</code> list-column with the parsed JSON body.
include_thoughts	Logical; if TRUE, requests explicit reasoning output via <code>generationConfig\$thinkingConfig\$includeThoughts</code> and stores the first returned text part as thoughts when available.
pair_uid	Optional stable per-pair identifier; when supplied, this value is used verbatim as <code>custom_id</code> (otherwise <code>custom_id</code> defaults to "LIVE_<ID1>_vs_<ID2>").
...	Reserved for future extensions.

Details

The prompt template should instruct the model to choose exactly one of `SAMPLE_1` or `SAMPLE_2` and wrap the decision in `<BETTER_SAMPLE>` tags.

Value

A tibble with one row and columns:

- `custom_id` - stable ID for the pair (`pair_uid` if supplied).
- `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 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_.

See Also

[check_llm_api_keys\(\)](#), [llm_compare_pair\(\)](#)

Other live backends: [anthropic_compare_pair_live\(\)](#), [check_llm_api_keys\(\)](#), [gemini_compare_pair_live\(\)](#), [llm_compare_pair\(\)](#), [ollama_compare_pair_live\(\)](#), [openai_compare_pair_live\(\)](#), [submit_anthropic_pairs_live\(\)](#), [submit_gemini_pairs_live\(\)](#), [submit_llm_pairs\(\)](#), [submit_ollama_pairs_live\(\)](#), [submit_openai_pairs_live\(\)](#), [submit_together_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#)

Examples

```
## Not run:
td <- trait_description("overall_quality")
vertex_compare_pair_live(
  ID1 = "A", text1 = "First response.",
  ID2 = "B", text2 = "Second response.",
  model = "gemini-3.8-flash",
  trait_name = td$name,
  trait_description = td$description,
  thinking_level = "low"
)

## End(Not run)
```

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.

See Also

[llm_submit_pairs_batch\(\)](#), [llm_download_batch_results\(\)](#)

Other batch backends: [anthropic_create_batch\(\)](#), [anthropic_download_batch_results\(\)](#), [anthropic_get_batch\(\)](#), [anthropic_poll_batch_until_complete\(\)](#), [build_anthropic_batch_requests\(\)](#), [build_gemini_batch_requests\(\)](#), [build_openai_batch_requests\(\)](#), [gemini_create_batch\(\)](#), [gemini_download_batch_results\(\)](#), [gemini_get_batch\(\)](#), [gemini_poll_batch_until_complete\(\)](#), [llm_download_batch_results\(\)](#), [llm_resume_multi_batches\(\)](#), [llm_submit_pairs_batch\(\)](#), [llm_submit_pairs_multi_batch\(\)](#), [openai_create_batch\(\)](#), [openai_download_batch_output\(\)](#), [openai_get_batch\(\)](#), [openai_poll_batch_until_complete\(\)](#), [openai_upload_batch_file\(\)](#), [run_anthropic_batch_pipeline\(\)](#), [run_gemini_batch_pipeline\(\)](#), [run_openai_batch_pipeline\(\)](#)

Examples

```
# Construct a minimal batch request tibble
requests <- tibble::tibble(
  custom_id = c("req1", "req2"),
  method = "POST",
  url = "/v1/chat/completions",
  body = list(
    list(
      model = "gpt-4o-mini",
      messages = list(
        list(role = "user", content = "Hello")
      )
    ),
    list(
      model = "gpt-4o-mini",
      messages = list(
```

```
        list(role = "user", content = "Goodbye")
      )
    )
  )

# Write to a temporary JSONL file
path <- tempfile(fileext = ".jsonl")
write_openai_batch_file(requests, path)

# Inspect the file contents
readLines(path)
```

Index

- * **Bayesian models**
 - build_btl_results_data, 42
 - fit_bayes_btl_mcmc, 67
- * **Ollama resource management**
 - ensure_only_ollama_model_loaded, 57
- * **adaptive logs**
 - adaptive_get_logs, 4
 - adaptive_item_log, 5
 - adaptive_results_history, 25
 - adaptive_round_log, 26
 - adaptive_step_log, 28
 - summarize_items, 166
 - summarize_refits, 168
- * **adaptive persistence**
 - load_adaptive_session, 99
 - save_adaptive_session, 140
 - validate_session_dir, 174
- * **adaptive ranking**
 - adaptive_rank, 6
 - adaptive_rank_resume, 16
 - adaptive_rank_run_live, 17
 - adaptive_rank_start, 23
 - make_adaptive_judge_llm, 100
 - summarize_adaptive, 164
- * **batch backends**
 - anthropic_create_batch, 34
 - anthropic_download_batch_results, 36
 - anthropic_get_batch, 37
 - anthropic_poll_batch_until_complete, 38
 - build_anthropic_batch_requests, 40
 - build_gemini_batch_requests, 46
 - build_openai_batch_requests, 49
 - gemini_create_batch, 76
 - gemini_download_batch_results, 78
 - gemini_get_batch, 80
 - gemini_poll_batch_until_complete, 81
 - llm_download_batch_results, 89
 - llm_resume_multi_batches, 90
 - llm_submit_pairs_batch, 93
 - llm_submit_pairs_multi_batch, 96
 - openai_create_batch, 110
 - openai_download_batch_output, 111
 - openai_get_batch, 112
 - openai_poll_batch_until_complete, 113
 - openai_upload_batch_file, 114
 - run_anthropic_batch_pipeline, 127
 - run_gemini_batch_pipeline, 131
 - run_openai_batch_pipeline, 134
 - write_openai_batch_file, 177
- * **bias and consistency**
 - check_positional_bias, 53
 - compute_reverse_consistency, 56
- * **cost estimation**
 - estimate_llm_pairs_cost, 59
- * **datasets**
 - example_openai_batch_output, 62
 - example_writing_pairs, 63
 - example_writing_results, 64
 - example_writing_samples, 65
 - example_writing_samples1000, 65
- * **frequentist models**
 - build_bt_data, 44
 - build_elo_data, 45
 - fit_bt_model, 69
 - fit_elo_model, 71
 - summarize_bt_fit, 165
- * **live backends**
 - anthropic_compare_pair_live, 30
 - check_llm_api_keys, 52
 - gemini_compare_pair_live, 73
 - llm_compare_pair, 85
 - ollama_compare_pair_live, 103
 - openai_compare_pair_live, 107

- submit_anthropic_pairs_live, 142
- submit_gemini_pairs_live, 145
- submit_llm_pairs, 149
- submit_ollama_pairs_live, 152
- submit_openai_pairs_live, 155
- submit_together_pairs_live, 159
- submit_vertex_pairs_live, 162
- together_compare_pair_live, 170
- vertex_compare_pair_live, 175
- * pairing and data**
 - alternate_pair_order, 29
 - make_pairs, 102
 - randomize_pair_order, 122
 - read_samples_df, 123
 - read_samples_dir, 124
 - sample_pairs, 138
 - sample_reverse_pairs, 139
- * prompts and traits**
 - build_prompt, 51
 - get_prompt_template, 83
 - list_prompt_templates, 84
 - register_prompt_template, 125
 - remove_prompt_template, 127
 - set_prompt_template, 141
 - trait_description, 172
- * result normalization**
 - parse_anthropic_batch_output, 115
 - parse_gemini_batch_output, 117
 - parse_openai_batch_output, 118
- adaptive_get_logs, 4
- adaptive_get_logs(), 6, 13, 26, 27, 29, 164, 167, 169
- adaptive_item_log, 5
- adaptive_item_log(), 5, 20, 25–27, 29, 167, 169
- adaptive_rank, 6
- adaptive_rank(), 16, 18, 20, 24, 25, 102, 164
- adaptive_rank_resume, 16
- adaptive_rank_resume(), 14, 20, 25, 100, 102, 164
- adaptive_rank_run_live, 17
- adaptive_rank_run_live(), 6, 8, 11–14, 16, 25, 27, 29, 100, 102, 164
- adaptive_rank_start, 23
- adaptive_rank_start(), 14, 16, 17, 20, 102, 164
- adaptive_results_history, 25
- adaptive_results_history(), 5, 6, 27, 29, 167, 169
- adaptive_round_log, 26
- adaptive_round_log(), 5, 6, 20, 25, 26, 29, 167, 169
- adaptive_step_log, 28
- adaptive_step_log(), 5, 6, 20, 25–27, 167, 169
- alternate_pair_order, 29, 122, 123
- alternate_pair_order(), 103, 123–125, 138, 139
- anthropic_compare_pair_live, 30, 41, 116, 142–144
- anthropic_compare_pair_live(), 53, 75, 85, 87, 106, 109, 144, 148, 151, 155, 157, 160, 163, 172, 177
- anthropic_create_batch, 34, 129, 130
- anthropic_create_batch(), 37–39, 41, 48, 50, 77, 79, 81, 82, 89, 92, 95, 98, 110–112, 114, 115, 130, 133, 137, 178
- anthropic_download_batch_results, 36, 115, 116, 129
- anthropic_download_batch_results(), 35, 38, 39, 41, 48, 50, 77, 79, 81, 82, 89, 92, 95, 98, 110–112, 114, 115, 130, 133, 137, 178
- anthropic_get_batch, 37, 38, 39
- anthropic_get_batch(), 35, 37, 39, 41, 48, 50, 77, 79, 81, 82, 89, 92, 95, 98, 110–112, 114, 115, 130, 133, 137, 178
- anthropic_poll_batch_until_complete, 38, 128, 129
- anthropic_poll_batch_until_complete(), 35, 37, 38, 41, 48, 50, 77, 79, 81, 82, 89, 92, 95, 98, 110–112, 114, 115, 130, 133, 137, 178
- base::print(), 164
- BTm, 69
- btm, 63, 69
- build_anthropic_batch_requests, 35, 40, 128, 129
- build_anthropic_batch_requests(), 35, 37–39, 48, 50, 77, 79, 81, 82, 89, 92, 93, 95, 98, 110–112, 114, 115, 130, 133, 137, 178
- build_bt_data, 44, 69, 119

- `build_bt_data()`, 26, 46, 70, 72, 87, 105, 166
- `build_btl_results_data`, 42
- `build_btl_results_data()`, 67, 68
- `build_elo_data`, 45, 71
- `build_elo_data()`, 44, 70, 72, 166
- `build_gemini_batch_requests`, 46, 77, 133
- `build_gemini_batch_requests()`, 35, 37–39, 41, 50, 77, 79, 81, 82, 89, 92, 95, 98, 110–112, 114, 115, 130, 133, 137, 178
- `build_openai_batch_requests`, 41, 49, 107, 108, 177, 178
- `build_openai_batch_requests()`, 35, 37–39, 41, 48, 77, 79, 81, 82, 89, 92, 95, 98, 110–112, 114, 115, 130, 133, 135–137, 178
- `build_prompt`, 51
- `build_prompt()`, 83, 84, 126, 127, 141, 173
- `check_llm_api_keys`, 52
- `check_llm_api_keys()`, 33, 75, 87, 106, 109, 144, 148, 151, 155, 157, 160, 163, 172, 177
- `check_positional_bias`, 53, 56
- `check_positional_bias()`, 57
- `compute_reverse_consistency`, 54, 56, 122, 123
- `compute_reverse_consistency()`, 55
- `elochoice`, 71
- `ensure_only_ollama_model_loaded`, 57
- `estimate_llm_pairs_cost`, 59, 121
- `example_openai_batch_output`, 62
- `example_writing_pairs`, 63, 64
- `example_writing_results`, 64
- `example_writing_samples`, 63, 65
- `example_writing_samples1000`, 65
- `fit_bayes_btl_mcmc`, 64, 67
- `fit_bayes_btl_mcmc()`, 42, 43
- `fit_bt_model`, 69, 71, 165
- `fit_bt_model()`, 44, 46, 72, 87, 105, 166
- `fit_elo_model`, 71
- `fit_elo_model()`, 44, 46, 70, 166
- `gemini_compare_pair_live`, 73
- `gemini_compare_pair_live()`, 33, 53, 85, 87, 106, 109, 133, 144–148, 151, 155, 157, 160, 163, 172, 177
- `gemini_create_batch`, 76, 80, 131
- `gemini_create_batch()`, 35, 37–39, 41, 48, 50, 79, 81, 82, 89, 92, 95, 98, 110–112, 114, 115, 130, 133, 137, 178
- `gemini_download_batch_results`, 78
- `gemini_download_batch_results()`, 35, 37–39, 41, 48, 50, 77, 81, 82, 89, 92, 95, 98, 110–112, 114, 115, 117, 130, 133, 137, 178
- `gemini_get_batch`, 80, 81, 82
- `gemini_get_batch()`, 35, 37–39, 41, 48, 50, 77, 79, 82, 89, 92, 95, 98, 110–112, 114, 115, 130, 133, 137, 178
- `gemini_poll_batch_until_complete`, 81
- `gemini_poll_batch_until_complete()`, 35, 37–39, 41, 48, 50, 77, 79, 81, 89, 92, 95, 98, 110–112, 114, 115, 130, 133, 137, 178
- `get_prompt_template`, 83, 127
- `get_prompt_template()`, 52, 84, 126, 127, 141, 173
- `list_prompt_templates`, 83, 84, 127
- `list_prompt_templates()`, 52, 83, 126, 127, 141, 173
- `llm_compare_pair`, 85
- `llm_compare_pair()`, 8, 13, 14, 33, 53, 75, 100–102, 105, 106, 109, 144, 148, 151, 155, 157, 160, 163, 172, 177
- `llm_download_batch_results`, 89
- `llm_download_batch_results()`, 35, 37–39, 41, 48, 50, 77, 79, 81, 82, 92, 93, 95, 98, 110–112, 114, 115, 130, 133, 137, 178
- `llm_resume_multi_batches`, 90
- `llm_resume_multi_batches()`, 35, 37–39, 41, 48, 50, 77, 79, 81, 82, 89, 95, 97, 98, 110–112, 114, 115, 130, 133, 137, 178
- `llm_submit_pairs_batch`, 93
- `llm_submit_pairs_batch()`, 35, 37–39, 41, 48, 50, 61, 77, 79, 81, 82, 89, 92, 98, 110–112, 114, 115, 130, 133, 137, 178
- `llm_submit_pairs_multi_batch`, 96
- `llm_submit_pairs_multi_batch()`, 35, 37–39, 41, 48, 50, 77, 79, 81, 82,

- [89–92, 95, 110, 112, 114, 115, 130, 133, 137, 178](#)
- `load_adaptive_session`, [99](#)
- `load_adaptive_session()`, [16, 140, 174](#)
- `make_adaptive_judge_llm`, [100](#)
- `make_adaptive_judge_llm()`, [7, 8, 12, 14, 16, 20, 25, 164](#)
- `make_pairs`, [40, 47, 59, 102, 122, 138, 143, 156](#)
- `make_pairs()`, [30, 61, 97, 123–125, 135, 138, 139, 149, 153, 159](#)
- `ollama_compare_pair_live`, [103](#)
- `ollama_compare_pair_live()`, [33, 53, 58, 75, 85–87, 109, 144, 148, 151, 153–155, 157, 160, 163, 172, 177](#)
- `openai_compare_pair_live`, [32, 107, 155, 157](#)
- `openai_compare_pair_live()`, [33, 53, 75, 85–87, 106, 144, 148, 151, 155, 157, 160, 163, 170, 172, 177](#)
- `openai_create_batch`, [110](#)
- `openai_create_batch()`, [35, 37–39, 41, 48, 50, 77, 79, 81, 82, 89, 92, 95, 98, 112, 114, 115, 130, 133, 135–137, 178](#)
- `openai_download_batch_output`, [111](#)
- `openai_download_batch_output()`, [35, 37–39, 41, 48, 50, 77, 79, 81, 82, 89, 92, 95, 98, 110, 112, 114, 115, 130, 133, 135, 137, 178](#)
- `openai_get_batch`, [112](#)
- `openai_get_batch()`, [35, 37–39, 41, 48, 50, 77, 79, 81, 82, 89, 92, 95, 98, 110, 112–115, 130, 133, 137, 178](#)
- `openai_poll_batch_until_complete`, [113](#)
- `openai_poll_batch_until_complete()`, [35, 37–39, 41, 48, 50, 77, 79, 81, 82, 89, 92, 95, 98, 110, 112, 115, 130, 133, 135–137, 178](#)
- `openai_upload_batch_file`, [114](#)
- `openai_upload_batch_file()`, [35, 37–39, 41, 48, 50, 77, 79, 81, 82, 89, 92, 95, 98, 110, 112, 114, 130, 133, 135–137, 178](#)
- `parse_anthropic_batch_output`, [115, 129, 130](#)
- `parse_anthropic_batch_output()`, [91, 117, 119](#)
- `parse_gemini_batch_output`, [117, 131](#)
- `parse_gemini_batch_output()`, [91, 116, 119](#)
- `parse_openai_batch_output`, [107, 118](#)
- `parse_openai_batch_output()`, [116, 117, 135, 137](#)
- `print.adaptive_state`, [120](#)
- `print.pairwiseLLM_cost_estimate`, [121](#)
- `randomize_pair_order`, [40, 47, 59, 122, 143, 156](#)
- `randomize_pair_order()`, [30, 97, 103, 124, 125, 135, 138, 139, 149, 153, 159](#)
- `read_samples_df`, [123](#)
- `read_samples_df()`, [7, 30, 103, 123, 125, 138, 139](#)
- `read_samples_dir`, [124](#)
- `read_samples_dir()`, [30, 103, 123, 124, 138, 139](#)
- `readr::read_csv()`, [97](#)
- `register_prompt_template`, [83, 125, 127](#)
- `register_prompt_template()`, [52, 83, 84, 127, 141, 173](#)
- `reliability`, [71](#)
- `remove_prompt_template`, [83, 127](#)
- `remove_prompt_template()`, [52, 83, 84, 126, 141, 173](#)
- `run_anthropic_batch_pipeline`, [35, 127, 133](#)
- `run_anthropic_batch_pipeline()`, [35, 37–39, 41, 48, 50, 77, 79, 81, 82, 89, 92, 93, 95–98, 110, 112, 114, 115, 133, 137, 178](#)
- `run_gemini_batch_pipeline`, [77, 131](#)
- `run_gemini_batch_pipeline()`, [35, 37–39, 41, 48, 50, 77, 79, 81, 82, 89, 92, 93, 95, 97, 98, 110, 112, 114, 115, 130, 137, 178](#)
- `run_openai_batch_pipeline`, [127, 129, 130, 133, 134](#)
- `run_openai_batch_pipeline()`, [35, 37–39, 41, 48, 50, 77, 79, 81, 82, 89, 92, 95–98, 110, 112, 114, 115, 130, 133, 178](#)
- `sample_pairs`, [40, 47, 59, 122, 138, 143, 156](#)

- `sample_pairs()`, [30](#), [97](#), [103](#), [123–125](#), [135](#),
[139](#), [149](#), [153](#), [159](#)
- `sample_reverse_pairs`, [122](#), [123](#), [139](#)
- `sample_reverse_pairs()`, [30](#), [103](#), [123–125](#),
[138](#)
- `save_adaptive_session`, [140](#)
- `save_adaptive_session()`, [16](#), [25](#), [100](#), [174](#)
- `set_prompt_template`, [31](#), [40](#), [47](#), [51](#), [59](#), [83](#),
[128](#), [141](#), [143](#), [156](#)
- `set_prompt_template()`, [8](#), [52](#), [74](#), [83](#), [84](#),
[86](#), [94](#), [101](#), [104](#), [126](#), [127](#), [135](#), [146](#),
[150](#), [153](#), [159](#), [163](#), [170](#), [173](#), [176](#)
- `submit_anthropic_pairs_live`, [142](#)
- `submit_anthropic_pairs_live()`, [33](#), [53](#),
[75](#), [87](#), [106](#), [109](#), [148](#), [150](#), [151](#), [155](#),
[157](#), [160](#), [163](#), [172](#), [177](#)
- `submit_gemini_pairs_live`, [145](#)
- `submit_gemini_pairs_live()`, [33](#), [53](#), [75](#),
[87](#), [106](#), [109](#), [144](#), [150](#), [151](#), [155](#),
[157](#), [160](#), [163](#), [172](#), [177](#)
- `submit_llm_pairs`, [60](#), [149](#)
- `submit_llm_pairs()`, [33](#), [53](#), [61](#), [75](#), [87](#), [93](#),
[106](#), [109](#), [144](#), [148](#), [154](#), [155](#), [157](#),
[160](#), [163](#), [172](#), [177](#)
- `submit_ollama_pairs_live`, [152](#)
- `submit_ollama_pairs_live()`, [33](#), [53](#), [58](#),
[75](#), [87](#), [106](#), [109](#), [144](#), [148](#), [151](#), [157](#),
[160](#), [163](#), [172](#), [177](#)
- `submit_openai_pairs_live`, [155](#)
- `submit_openai_pairs_live()`, [33](#), [53](#), [75](#),
[87](#), [106](#), [109](#), [144](#), [148](#), [150](#), [151](#),
[155](#), [160](#), [163](#), [172](#), [177](#)
- `submit_together_pairs_live`, [158](#)
- `submit_together_pairs_live()`, [33](#), [53](#), [75](#),
[87](#), [106](#), [109](#), [144](#), [148](#), [151](#), [155](#),
[157](#), [163](#), [172](#), [177](#)
- `submit_vertex_pairs_live`, [162](#)
- `submit_vertex_pairs_live()`, [33](#), [53](#), [75](#),
[87](#), [106](#), [109](#), [144](#), [148](#), [150](#), [151](#),
[155](#), [157](#), [160](#), [172](#), [177](#)
- `summarize_adaptive`, [164](#)
- `summarize_adaptive()`, [13](#), [14](#), [16](#), [20](#), [25](#),
[102](#), [121](#)
- `summarize_bt_fit`, [165](#)
- `summarize_bt_fit()`, [44](#), [46](#), [70](#), [72](#)
- `summarize_items`, [166](#)
- `summarize_items()`, [5](#), [6](#), [13](#), [26](#), [27](#), [29](#), [67](#),
[68](#), [169](#)
- `summarize_refits`, [168](#)
- `summarize_refits()`, [5](#), [6](#), [13](#), [26](#), [27](#), [29](#), [67](#),
[167](#)
- `together_compare_pair_live`, [170](#)
- `together_compare_pair_live()`, [33](#), [53](#), [75](#),
[85](#), [87](#), [106](#), [109](#), [144](#), [148](#), [151](#), [155](#),
[157](#), [159](#), [160](#), [163](#), [177](#)
- `trait_description`, [172](#)
- `trait_description()`, [52](#), [83](#), [84](#), [126](#), [127](#),
[141](#)
- `validate_session_dir`, [174](#)
- `validate_session_dir()`, [100](#), [140](#)
- `vertex_compare_pair_live`, [175](#)
- `vertex_compare_pair_live()`, [33](#), [53](#), [75](#),
[85](#), [87](#), [106](#), [109](#), [144](#), [148](#), [151](#), [155](#),
[157](#), [160](#), [162](#), [163](#), [172](#)
- `write_openai_batch_file`, [177](#)
- `write_openai_batch_file()`, [35](#), [37–39](#), [41](#),
[48](#), [50](#), [77](#), [79](#), [81](#), [82](#), [89](#), [92](#), [95](#), [98](#),
[110](#), [112](#), [114](#), [115](#), [130](#), [133](#), [135](#),
[137](#)