

Package: pairwiseLLM (via r-universe)

October 19, 2026

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

Version 1.6.0

Description Provides a unified framework for generating, submitting, and analyzing pairwise comparisons of writing quality using large language models (LLMs). The package supports live and/or batch evaluation workflows across multiple providers ('OpenAI', 'Anthropic', 'Google Gemini', 'Together AI', 'TypeSafe', 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. The Bayesian writing quality scores can be converted to distribution-matched performance levels or calibrated to ordered rubric scores using linear or monotone ordinal models. 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

Roxygen list(markdown = TRUE)

Imports cli, curl, dplyr, httr2 (>= 1.1.0), jsonlite, openssl, rlang (>= 1.3.0), stats, tibble, tidyselect, tools, utils

Suggests BradleyTerry2, brglm2, cmdstanr, e1071, EloChoice, filelock, future, future.apply, glmnet, knitr, later, mgcv (>= 1.9-4), mockery, ordinal, parallelly, pkgload, pls, posterior,

promises, purrr, readr, rmarkdown, reticulate (≥ 1.41), sirt, stringr, testthat ($\geq 3.0.0$), tidyr, vctrs, withr ($\geq 3.0.3$)

Additional repositories <https://stan-dev.r-universe.dev/>

SystemRequirements CmdStan and a C++17 toolchain (optional; needed only for Bayesian and adaptive modeling); Python 3.12.3 with spaCy, TextDescriptives, textstat and English resources (optional; needed only for warm-start text feature extraction)

Config/testthat/edition 3

URL <https://github.com/shmercer/pairwiseLLM>,
<https://shmercer.github.io/pairwiseLLM/>

BugReports <https://github.com/shmercer/pairwiseLLM/issues>

Depends R (≥ 4.4)

VignetteBuilder knitr

Collate 'adaptive_benchmark_metrics.R' 'adaptive_bootstrap.R'
'adaptive_btl_refit.R' 'adaptive_candidates.R'
'adaptive_constraints.R' 'adaptive_linking_calibration.R'
'adaptive_linking_phase_a.R' 'adaptive_logs.R'
'adaptive_pairing_strategy.R' 'adaptive_persist.R'
'adaptive_predictive_tree.R' 'adaptive_print.R'
'adaptive_rank.R' 'adaptive_replay.R'
'adaptive_replay_reservoir.R' 'adaptive_replay_selection.R'
'adaptive_round_candidates.R' 'adaptive_run.R'
'adaptive_schemas.R' 'adaptive_select.R'
'adaptive_simulation_harness.R' 'adaptive_starvation.R'
'adaptive_state.R' 'adaptive_step.R' 'adaptive_trueskill.R'
'adaptive_utility.R' 'anthropic_batch_api.R' 'anthropic_live.R'
'api_keys.R' 'batch_retrieval.R' 'bayes_btl_mcmc.R'
'bayes_btl_mcmc_adaptive.R' 'bayes_btl_summarize.R'
'bt_alpha.R' 'bt_bootstrap.R' 'bt_bootstrap_contracts.R'
'bt_bootstrap_schedule.R' 'bt_common.R' 'bt_firth.R'
'bt_helpers.R' 'bt_lapse.R' 'bt_lapse_likelihood.R'
'bt_lapse_optimizer.R' 'bt_model.R' 'bt_provenance.R'
'btl_evidence_identity.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_resources.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' 'decision_bulk.R'
'decision_checkpoint.R' 'decision_live.R' 'decision_prompts.R'
'decision_results.R' 'draws_sanitize.R' 'elo_model.R'
'gemini_batch_api.R' 'gemini_live.R' 'gemini_params.R'
'http2_retry.R' 'link_contract.R' 'link_coordinates.R'
'link_fixed_shape.R' 'link_gaussian_bridge.R'
'link_joint_mcmc.R' 'link_joint_offset.R' 'link_legacy.R'

'link_likelihood.R' 'link_orchestration.R'
 'link_phase_a_draws.R' 'link_phase_a_evidence.R'
 'link_phase_a_points.R' 'link_probe_quality.R'
 'link_reporting.R' 'link_result.R' 'link_session.R'
 'llm_backends.R' 'llm_batch.R' 'llm_multi_batch.R'
 'normalize_results.R' 'ollama_live.R' 'openai_batch_api.R'
 'openai_batch_cache.R' 'openai_decisions.R' 'openai_live.R'
 'openai_params.R' 'pairing.R' 'pairwiseLLM-package.R'
 'prompt_template.R' 'reverse_consistency.R'
 'rubric_calibration.R' 'rubric_cj_contract.R'
 'rubric_cj_linked.R' 'rubric_diagnostics.R'
 'rubric_linked_anchors.R' 'rubric_metrics.R'
 'rubric_ordinal_linear.R' 'rubric_ordinal_monotone.R'
 'rubric_percentile.R' 'rubric_reference.R'
 'rubric_validation.R' 'seed_helpers.R' 'together_live.R'
 'traits.R' 'typesafe_live.R' 'utils-null-coalesce.R'
 'vertex_live.R' 'warm_start_adaptive.R'
 'warm_start_algorithm_ensemble.R' 'warm_start_bundle.R'
 'warm_start_calibration.R' 'warm_start_candidate_validity.R'
 'warm_start_coefficients.R' 'warm_start_cv.R'
 'warm_start_cv_plan.R' 'warm_start_engine.R'
 'warm_start_ensemble.R' 'warm_start_feature_schema.R'
 'warm_start_features.R' 'warm_start_format3.R'
 'warm_start_model.R' 'warm_start_model_artifact.R'
 'warm_start_model_io.R' 'warm_start_model_registry.R'
 'warm_start_pls.R' 'warm_start_pls_validation.R'
 'warm_start_predict.R' 'warm_start_predictions.R'
 'warm_start_preprocess.R' 'warm_start_prior.R'
 'warm_start_python.R' 'warm_start_svr.R'
 'warm_start_svr_validation.R' 'warm_start_tune.R'
 'warm_start_validation.R'

Config/roxygen2/version 8.1.0

Config/pak/sysreqs libssl-dev python3

Repository <https://shmercer.r-universe.dev>

Date/Publication 2026-10-19 19:15:35 UTC

RemoteUrl <https://github.com/shmercer/pairwisellm>

RemoteRef HEAD

RemoteSha 3cf49a9a2559f341e30f2e77e8df2ad4883ab7f5

Contents

adaptive_get_logs	6
adaptive_item_log	7
adaptive_rank	8
adaptive_rank_resume	20
adaptive_rank_run_live	22

adaptive_rank_start	29
adaptive_results_history	33
adaptive_round_log	34
adaptive_step_log	35
alternate_pair_order	37
anthropic_compare_pair_live	38
anthropic_create_batch	42
anthropic_download_batch_results	44
anthropic_get_batch	45
anthropic_poll_batch_until_complete	46
bootstrap_bt_model	48
build_anthropic_batch_requests	51
build_bt_data	54
build_btl_results_data	55
build_elo_data	56
build_gemini_batch_requests	57
build_openai_batch_requests	60
build_prompt	64
check_llm_api_keys	65
check_positional_bias	66
compute_reverse_consistency	68
decision_bulk	69
ensemble_warm_start_algorithms	73
ensemble_warm_start_models	74
ensure_only_ollama_model_loaded	76
estimate_llm_pairs_cost	78
evaluate_rubric_predictions	81
example_openai_batch_output	83
example_writing_pairs	84
example_writing_results	85
example_writing_samples	86
example_writing_samples1000	87
extract_warm_start_features	88
fit_bayes_btl_mcmc	89
fit_bt_model	92
fit_elo_model	99
fit_link	101
fit_rubric_calibration	104
fit_warm_start_model	108
gemini_compare_pair_live	112
gemini_create_batch	116
gemini_download_batch_results	118
gemini_get_batch	120
gemini_poll_batch_until_complete	121
get_prompt_template	123
list_prompt_templates	124
llm_compare_pair	125
llm_download_batch_results	129

llm_resume_multi_batches	131
llm_submit_pairs_batch	133
llm_submit_pairs_multi_batch	138
load_adaptive_session	141
make_adaptive_judge_llm	143
make_adaptive_judge_replay	145
make_adaptive_replay_reservoir	146
make_pairs	147
make_warm_start_cv_plan	149
make_warm_start_prior	150
ollama_compare_pair_live	151
openai_compare_pair_live	155
openai_create_batch	159
openai_download_batch_errors	161
openai_download_batch_output	162
openai_get_batch	163
openai_poll_batch_until_complete	164
openai_upload_batch_file	166
pairwiseLLM_warm_model	167
parse_anthropic_batch_output	170
parse_gemini_batch_output	172
parse_openai_batch_output	173
predict.pairwiseLLM_bt_alpha	175
predict.pairwiseLLM_bt_firth	176
predict.pairwiseLLM_bt_lapse	177
predict.pairwiseLLM_rubric_calibration	178
predict.pairwiseLLM_warm_algorithm_ensemble	179
predict.pairwiseLLM_warm_ensemble	181
predict.pairwiseLLM_warm_model	182
predict_link	184
prepare_link_input	185
prepare_linked_rubric_reference	189
prepare_warm_start_model	190
print.adaptive_state	192
print.pairwiseLLM_cost_estimate	193
randomize_pair_order	194
read_samples_df	195
read_samples_dir	196
register_prompt_template	197
register_warm_start_model	199
remove_prompt_template	201
run_anthropic_batch_pipeline	202
run_gemini_batch_pipeline	205
run_openai_batch_pipeline	209
sample_pairs	213
sample_reverse_pairs	214
save_adaptive_session	215
save_link_session	216

save_warm_start_model	217
scale_separation_reliability	219
set_prompt_template	220
start_link_session	221
submit_anthropic_pairs_live	222
submit_gemini_pairs_live	225
submit_llm_pairs	229
submit_ollama_pairs_live	233
submit_openai_pairs_live	236
submit_together_pairs_live	240
submit_typesafe_pairs_live	243
submit_vertex_pairs_live	245
summarize_adaptive	247
summarize_bt_fit	249
summarize_items	250
summarize_refits	252
summary.pairwiseLLM_link_result	254
summary.pairwiseLLM_warm_algorithm_ensemble	255
summary.pairwiseLLM_warm_ensemble	256
summary.pairwiseLLM_warm_predictions	258
together_compare_pair_live	259
trait_description	262
typesafe_compare_pair_live	263
validate_adaptive_replay	267
validate_session_dir	268
vertex_compare_pair_live	269
warm_start_coefficients	271
warm_start_feature_schema	273
warm_start_python_status	275
write_openai_batch_file	276

Index	278
--------------	------------

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

<code>state</code>	Adaptive state.
<code>refit_id</code>	Optional refit index.
<code>stack</code>	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,
  warm_start_model = NULL,
  warm_start_prior = NULL,
  warm_start_features = NULL,
  warm_start_python = NULL,
  warm_start_prior_sd = NULL,
  warm_start_mode = NULL,
  replay_reservoir = NULL,
  warm_start_trueskill = NULL,
  bootstrap_policy = "shuffled_connected"
)
```

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 <code>read_samples_df()</code> . Default is 2.
backend	Backend passed to <code>make_adaptive_judge_llm()</code> . Choices are "openai", "anthropic", "gemini", "vertex", "together", and "ollama". Default is "openai".
model	Model passed to <code>make_adaptive_judge_llm()</code> . 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: pairing_strategy Post-bootstrap strategy: hybrid (default), random, trueskill_p50, or trueskill_pollitt. Direct strategies currently require run_mode = "within_set". The default bootstrap is connected and shuffled; predictive graphs require Pollitt. Direct strategies choose a focal item from sorted IDs uniformly at minimum committed degree, using the run seed and committed count; invalid judgments retry the same draw. Among legal partners, random chooses uniformly, trueskill_p50 minimizes distance to TrueSkill probability 0.50, and trueskill_pollitt minimizes distance to 1/3 or 2/3, with item-ID tie breaking. This is a Pollitt-inspired strategy using TrueSkill probabilities; the earlier article used BTL probabilities, so this is not an exact replication. Direct strategies allow at most two observations per unordered pair, with canonical presentation balancing and reversal on repeat. They stop on focal partner exhaustion and do not use hybrid

stage quotas or coverage overrides. Step logs identify `direct_pairing`, `pairing_strategy`, and `target_distance`; `i_id` is the focal item and `p_ij` is the pre-judgment TrueSkill probability for presented A over B. Target distance is symmetric under reversal and is NA for random pairing. BTL estimation, refit cadence, and stopping remain unchanged. On resume, omit this field or supply the saved strategy; changing strategy requires a new session.

- `dup_max_obs_relaxed` Hybrid's maximum observations per unordered pair under the relaxed near-tie fallback: 3L (historical default) or 2L. For directed-table replay, use 2L with `make_adaptive_judge_replay()` for two collected orientations. The ordinary ceiling remains two; direct strategies already cap at two. Automatic Phase B selection is unavailable. This setting persists; sparse reservoirs independently enforce a one-use ceiling. A requested budget may be unreachable even below the arithmetic multiplicity ceiling because third observations are conditional and other filters remain active. Use `summarize_adaptive()` with `include_starvation = TRUE` for recorded terminal hybrid evidence.
- `global_identified_reliability_min` Global EAP reliability threshold used to mark the run as globally identified after a refit. Default is 0.80.
- `global_identified_rank_corr_min` 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.
- `p_long_low` Lower bound for long-link win probability gating after global identification. Within-set/Phase-A hybrid uses TrueSkill throughout. Automatic Phase B selection is unavailable. Default is 0.10.
- `p_long_high` Upper bound for the same long-link probability gate. Default is 0.90; bounds are inclusive.
- `long_taper_mult` Multiplier controlling long-link quota tapering after global identifiability. Default is 0.25.
- `long_frac_floor` Floor fraction for long-link quota after tapering. Default is 0.02.
- `mid_bonus_frac` 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.

link_estimation_mode Explicit estimator identity for linking Phase A preparation: `fixed_shape_offset`, `gaussian_posterior_bridge`, or `joint_offset`. E1 needs earlier scores, E2 needs joint posterior draws, and E3 needs original within-set comparisons under compatible model/prior settings. There is no default. Adaptive Phase B selection is unavailable pending validation. Use `prepare_link_input()`, `fit_link()`, and `start_link_session()` for explicit-evidence linking. Legacy Phase B posteriors cannot be migrated; restart from compatible Phase A inputs.

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 runtime 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. Within-set resume reuses the saved configuration when omitted; an explicit list resolves against the defaults. 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-
hat R . 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. Saved predictive mode, prior, TrueSkill state, bootstrap queue, and pairing strategy are retained without loading a model or regenerating predictions. Omit all predictive initialization arguments on resume. Other supported controller overrides remain available. 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.
warm_start_model	Optional calibrated model, cross-task or same-task algorithm ensemble, path string, or loader reference list (name/source or path). Mutually exclusive with warm_start_prior. Resolve and predict once when creating an assessment.
warm_start_prior	Optional <code>make_warm_start_prior()</code> object covering all items. Saved numeric scores are centered within each BTL refit scope.
warm_start_features	Optional precomputed feature rows for model input; otherwise use item texts. Precomputed prediction needs neither Python nor a fitting backend.
warm_start_python	Explicit Python interpreter for text extraction only.
warm_start_prior_sd	Optional user-chosen raw theta prior SD for model input; scalar or per-item vector, default 0.5. Supplied prior objects retain their SDs. With warm_start_trueskill = "predictive_distribution", explicit SD is required for model input, including trueskill_only, and also initializes TrueSkill sigma. Otherwise trueskill_only rejects this argument.
warm_start_mode	Predictive destination: cold (neither model), btl_only (BTL prior), trueskill_only (TrueSkill locations), or both (both models). Omitted/NULL mode defaults to btl_only with predictive input, otherwise cold. Request both explicitly to initialize both models. In TrueSkill-warm modes, exact item-ID alignment precedes $\mu = \mu_0 + \sigma_0 * \text{prior_mean}$, with $\mu_0 = 25$, $\sigma_0 = 25/3$, fixed

multiplier 1, and unchanged sigma unless warm_start_trueskill explicitly requests distribution initialization. Explicit cold with predictive input, or a non-cold mode without it, errors.

replay_reservoir

Optional `make_adaptive_replay_reservoir()` object. Requires ordinary within-set mode and a matching reservoir replay judge. Uses a seeded spanning-tree bootstrap and at most one committed observation per allowed unordered edge, always in its frozen observed orientation. On resume, omit this argument or supply the identical reservoir.

warm_start_trueskill

Optional explicit uncertainty policy: NULL retains legacy initialization; "predictive_distribution" requires trueskill_only or both and maps $\mu = 25 + (25/3) * \text{prior_mean}$ and $\sigma = (25/3) * \text{prior_sd}$, with $\beta = 25/6$. It overrides constructor means and sigmas, aligns by exact item ID, and never clips SD. Model input requires explicit warm_start_prior_sd; prior objects use their stored SD. Named SD vectors align by ID; unnamed vectors follow prediction-input order. Omit this argument on resume; the saved distribution policy is authoritative.

bootstrap_policy

Initial graph policy, default "shuffled_connected". "predictive_connected" requires a selectable replay_reservoir, ordinary within-set mode, warm_start_trueskill = "predictive_distribution", and adaptive_config = list(pairing_strategy = "trueskill_pollitt"). Build the reservoir from selectable primary observations only, excluding held-out edges and reversal audits. The graph uses only manifest endpoints, frozen initial TrueSkill means/SDs, and the seed, never outcomes. The queue is built once before judging and retained across updates and resume. On wrapper resume, omit this argument or supply the saved policy; a different policy or explicit predictive initialization seed is rejected.

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. An explicit `link_estimation_mode` is required. Adaptive Phase B is unavailable pending selector validation; use `start_link_session()` with explicit evidence for E1–E3 estimation. 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: selection uses one-pair transactional steps after the configured connected bootstrap. In the default hybrid strategy, TrueSkill supplies live ranks, strata, pair probabilities, base utility, and rolling anchors throughout within-set and Phase-A work. Rolling anchors use current TrueSkill ranks, 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)`.

Hybrid exposure and repeat routing: under-represented routing is degree-based ($\text{deg} \leq D_{\min} + 1$), 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 supply item estimates, posterior uncertainty, EAP reliability, diagnostics, stopping, and the existing `global_identified` signal. That signal can change later hybrid tapering and routing, so selection is not wholly independent of BTL. The within-set/Phase-A long-link gate uses TrueSkill probabilities throughout. Automatic linking Phase B selection is unavailable pending validation; use `prepare_link_input()` and `start_link_session()`.

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.

Predictive destinations and initial connectivity are separate choices. By default, every warm mode retains the same seeded shuffled bootstrap of $N - 1$ valid comparisons. The explicit predictive graph policy uses a frozen allowed spanning tree with Pollitt probability targets and degree-cap relaxation. Both policies preserve invalid-result retries and recorded reservoir orientation. Later pairing retains the configured strategy. For budgets expressed as mean comparison exposures per item, each committed pair counts twice. The synthetic five-arm example uses $\text{floor}(B * N / 2)$ total committed pairs, including bootstrap, and reports realized exposure $2 * \text{committed_pairs} / N$. $B = 0$ is prior-only; $B = 0.5$ and $B = 1$ precede connectivity. The tree completes at $N - 1$ successful commits; comparison N uses the post-bootstrap strategy. `n_steps` counts attempts, so invalid or retried judgments must not be counted as additional committed evidence. See `vignette("adaptive-warm-start")` for count and continuation examples. Without the distribution opt-in, BTL prior SD does not determine TrueSkill sigma. Ensemble disagreement never supplies SD automatically. No historical training-score units are restored.

Distribution initialization assumes the supplied BTL prior SD and TrueSkill uncertainty describe comparable latent scales under the documented affine convention. BTL priors apply to `theta_raw`; centering induces dependence, so this is not the marginal SD of centered BTL effects and does not equate the models' posteriors. Existing BTL identifiability and prior rules are unchanged. Upstream predictive workflows must establish uncertainty calibration using training data only; supplying SD explicitly does not establish calibration. Scalar SD supports sensitivity analyses, not essay-specific calibration. Saved metadata records the versioned mapping, SD source and scalar/per-item rule, and integrity digests; calibration is recorded as upstream, unverified.

Predictive BTL priors apply only in `btl_only` and both, including run-required linking Phase A. TrueSkill initialization applies in `trueskill_only` and both. Imported Phase-A artifacts retain their own generation identity and are not rerun because predictive input exists. Linking and pooled judge refits keep their existing prior rules; predictive evidence is not injected into Phase B priors, D-optimal selection, or probes. Custom BTL fit functions should consume `state$predictive_prior` only when `state$meta$warm_start_mode` is `btl_only` or both; its presence alone does not imply BTL warming. Resume preserves saved predictions, current TrueSkill state, mode, strategy, and bootstrap progress; omit all warm-start arguments.

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()`

`make_warm_start_prior()`, `fit_warm_start_model()` and the [integrated CJ workflow](#)

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

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)

```

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; must be empty. Resume uses persisted predictive priors.

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. The saved predictive mode, prior, pairing strategy, current TrueSkill state, bootstrap policy, and frozen queue are authoritative. Resume does not reload a predictive model, regenerate predictions, initialize ratings, or rebuild a predictive tree. SHA-256 validates the frozen predictive inputs and queue against saved session metadata. Legacy shuffled sessions retain their original behavior.

Value

An `adaptive_state` object restored from disk.

See Also

`make_warm_start_prior()`, `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()`, `make_adaptive_judge_replay()`, `make_adaptive_replay_reservoir()`, `summarize_adaptive()`, `validate_adaptive_replay()`

Examples

```
dir <- tempfile("pwillm-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. A resumed session retains its saved pairing strategy: omit <code>pairing_strategy</code> or supply the same value. Other supported controller overrides remain available. 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. Within-set continuation and resume reuse the saved configuration when this argument is omitted; an explicit list resolves against the defaults. 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/Phase-A hybrid routing uses TrueSkill ranks, strata, rolling anchors, pair probabilities, and utility

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

. The long-link probability gate uses TrueSkill throughout within-set/Phase-A hybrid selection. Direct strategies apply their partner targets after the same connected bootstrap and currently require ordinary within-set mode. Adaptive Phase B selection is unavailable pending separate validation. Use `prepare_link_input()`, `fit_link()`, and `start_link_session()` with explicit cross-set evidence and an explicit E1–E3 estimator. Legacy Phase B posteriors cannot be resumed or migrated into a new estimator. Exploration/exploitation routing and fallback handling are recorded in `step_log`.

Hybrid 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.

Hybrid exposure and repeat handling are soft, stage-local constraints: under-represented exploration uses degree set $\text{deg} \leq D_{\min} + 1$, 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 by committed comparisons 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 hybrid pair routing and quotas through the existing `global_identified` signal. BTL supplies item estimates, posterior uncertainty, EAP reliability, diagnostics, and stopping. Automatic Phase B selection is unavailable pending separate validation. 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.

Phase B linking restriction

Adaptive Phase B D-optimal selection is unavailable pending a separate selector validation study. This includes all E1–E3 engines and legacy D-optimal aliases; execution fails before selecting or judging a Phase B pair. Phase A and ordinary within-set ranking remain available. For linking, use `prepare_link_input()`, `fit_link()`, and `start_link_session()` with explicit cross-set evidence and an explicit estimator choice.

See Also

`make_warm_start_prior()`, `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()`, `make_adaptive_judge_replay()`, `make_adaptive_replay_reservoir()`, `summarize_adaptive()`, `validate_adaptive_replay()`

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("pwllm-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(), "pwlmlm-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	<i>Adaptive ranking</i>
---------------------	-------------------------

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,
  warm_start_model = NULL,
  warm_start_prior = NULL,
  warm_start_features = NULL,
  warm_start_python = NULL,
  warm_start_prior_sd = NULL,
  warm_start_mode = NULL,
  replay_reservoir = NULL,
  warm_start_trueskill = NULL,
  bootstrap_policy = "shuffled_connected"
)
```

Arguments

<code>items</code>	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.
<code>seed</code>	Integer seed used for deterministic connected-bootstrap choices and selection randomness. Default is 1L.
<code>session_dir</code>	Optional directory for saving session artifacts. Default is NULL.
<code>persist_item_log</code>	Logical; when TRUE, write per-refit item logs to disk. Default is FALSE.
<code>...</code>	Internal/testing only. Supply <code>now_fn</code> to override the clock used for timestamps.
<code>adaptive_config</code>	Optional named list of adaptive controller overrides. <code>pairing_strategy</code> defaults to <code>hybrid</code> ; <code>random</code> , <code>trueskill_p50</code> , and <code>trueskill_pollitt</code> select direct pairs after the configured connected bootstrap and currently require <code>run_mode</code>

	= "within_set". 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.
warm_start_model	Optional calibrated model, cross-task or same-task algorithm ensemble, path string, or loader reference list (name/source or path). Mutually exclusive with warm_start_prior. Resolve and predict once when creating an assessment.
warm_start_prior	Optional make_warm_start_prior() object covering all items. Saved numeric scores are centered within each BTL refit scope.
warm_start_features	Optional precomputed feature rows for model input; otherwise use item texts. Precomputed prediction needs neither Python nor a fitting backend.
warm_start_python	Explicit Python interpreter for text extraction only.
warm_start_prior_sd	Optional user-chosen raw theta prior SD for model input; scalar or per-item vector, default 0.5. Supplied prior objects retain their SDs. With warm_start_trueskill = "predictive_distribution", explicit SD is required for model input, including trueskill_only, and also initializes TrueSkill sigma. Otherwise trueskill_only rejects this argument.
warm_start_mode	Predictive destination: cold (neither model), btl_only (BTL prior), trueskill_only (TrueSkill locations), or both (both models). Omitted/NULL mode defaults to btl_only with predictive input, otherwise cold. Request both explicitly to initialize both models. In TrueSkill-warm modes, exact item-ID alignment precedes $\mu = \mu_0 + \sigma_0 * \text{prior_mean}$, with $\mu_0 = 25$, $\sigma_0 = 25/3$, fixed multiplier 1, and unchanged sigma unless warm_start_trueskill explicitly requests distribution initialization. Explicit cold with predictive input, or a non-cold mode without it, errors.
replay_reservoir	Optional make_adaptive_replay_reservoir() object. Requires ordinary within-set mode and a matching reservoir replay judge. Uses a seeded spanning-tree bootstrap and at most one committed observation per allowed unordered edge, always in its frozen observed orientation. On resume, omit this argument or supply the identical reservoir.
warm_start_trueskill	Optional explicit uncertainty policy: NULL retains legacy initialization; "predictive_distribution" requires trueskill_only or both and maps $\mu = 25 + (25/3) * \text{prior_mean}$ and $\sigma = (25/3) * \text{prior_sd}$, with $\beta = 25/6$. It overrides constructor means and sigmas, aligns by exact item ID, and never clips SD. Model input requires explicit warm_start_prior_sd; prior objects use their stored SD. Named SD vectors align by ID; unnamed vectors follow prediction-input order. Omit this argument on resume; the saved distribution policy is authoritative.

bootstrap_policy

Initial graph policy, default "shuffled_connected". "predictive_connected" requires a selectable replay_reservoir, ordinary within-set mode, warm_start_trueskill = "predictive_distribution", and adaptive_config = list(pairing_strategy = "trueskill_pollitt"). Build the reservoir from selectable primary observations only, excluding held-out edges and reversal audits. The graph uses only manifest endpoints, frozen initial TrueSkill means/SDs, and the seed, never outcomes. The queue is built once before judging and retained across updates and resume. On wrapper resume, omit this argument or supply the saved policy; a different policy or explicit predictive initialization seed is rejected.

Details

This function creates the stepwise controller state and seeds all canonical logs used in the adaptive pairing workflow. The default connected bootstrap uses a seeded shuffled chain, or a shuffled allowed tree for replay reservoirs. Either graph policy connects all items after $N - 1$ committed comparisons.

Pair selection in this framework is stepwise and uncertainty-aware. Within-set/Phase-A hybrid routing uses TrueSkill ranks, strata, rolling anchors, pair probabilities, and base utility

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

where p_{ij} is the current TrueSkill win probability for pair $\{i, j\}$. Linking Phase A preparation requires an explicit adaptive_config\$link_estimation_mode. Adaptive Phase B selection remains unavailable pending separate validation. Use `prepare_link_input()`, `fit_link()`, and `start_link_session()` with explicit cross-set evidence and frozen shared judge parameters for E1–E3 linking. The within-set/Phase-A hybrid long-link gate uses TrueSkill throughout. Bayesian BTL supplies item estimates, posterior uncertainty, EAP reliability, diagnostics, stopping, and the existing global_identified signal. This signal can affect later hybrid tapering and routing; selection is not wholly independent of BTL. Direct within-set strategies use their documented partner targets after the common bootstrap. Automatic Phase B selection is unavailable pending separate validation. 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()`.

Predictive destinations and initial connectivity are separate choices. By default, every warm mode retains the same seeded shuffled bootstrap of $N - 1$ valid comparisons. The explicit predictive graph policy uses a frozen allowed spanning tree with Pollitt probability targets and degree-cap relaxation. Both policies preserve invalid-result retries and recorded reservoir orientation. Later pairing retains the configured strategy. When expressing budgets as mean comparison exposures per item, each committed pair contributes two exposures. A prespecified rounding-down convention gives $\text{floor}(B * N / 2)$ total pairs, including bootstrap, with realized exposure $2 * \text{committed_pairs} / N$. $B = 0$ is prior-only; $B = 0.5$ and $B = 1$ precede connectivity and are not fully connected comparative-judgment estimates. The tree completes at $N - 1$ successful commits; comparison N

uses the post-bootstrap strategy. Invalid attempts and retries do not add evidence. See `vignette("adaptive-warm-start")` for the five-arm synthetic count example. Without the distribution opt-in, BTL prior SD does not determine TrueSkill sigma. Ensemble disagreement never supplies SD automatically. No historical training-score units are restored.

Distribution initialization assumes the supplied BTL prior SD and TrueSkill uncertainty describe comparable latent scales under the documented affine convention. BTL priors apply to `theta_raw`; centering induces dependence, so this is not the marginal SD of centered BTL effects and does not equate the models' posteriors. Existing BTL identifiability and prior rules are unchanged. Upstream predictive workflows must establish uncertainty calibration using training data only; supplying SD explicitly does not establish calibration. Scalar SD supports sensitivity analyses, not essay-specific calibration. Saved metadata records the versioned mapping, SD source and scalar/per-item rule, and integrity digests; calibration is recorded as upstream, unverified.

Predictive BTL priors apply only in `bt1_only` and `both`, including run-required linking Phase A. TrueSkill initialization applies in `trueskill_only` and `both`. Imported Phase-A artifacts retain their own generation identity and are not rerun because predictive input exists. Linking and pooled judge refits keep their existing prior rules; predictive evidence is not injected into Phase B priors, D-optimal selection, or probes. Custom BTL fit functions should consume `state$predictive_prior` only when `state$meta$warm_start_mode` is `bt1_only` or `both`; its presence alone does not imply BTL warming. Resume preserves saved predictions, current TrueSkill state, mode, strategy, and bootstrap progress; omit all warm-start arguments.

Value

An adaptive state object containing `step_log`, `round_log`, and `item_log`. The object includes class `"adaptive_state"`, item ID mappings, TrueSkill state, frozen bootstrap policy/queue, refit metadata, and runtime configuration.

Phase B linking restriction

Adaptive Phase B D-optimal selection is unavailable pending a separate selector validation study. This includes all E1–E3 engines and legacy D-optimal aliases; execution fails before selecting or judging a Phase B pair. Phase A and ordinary within-set ranking remain available. For linking, use `prepare_link_input()`, `fit_link()`, and `start_link_session()` with explicit cross-set evidence and an explicit estimator choice.

See Also

`make_warm_start_prior()`, `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()`, `make_adaptive_judge_replay()`, `make_adaptive_replay_reservoir()`, `summarize_adaptive()`, `validate_adaptive_replay()`

Examples

```
state <- adaptive_rank_start(c("a", "b", "c"), seed = 11)
summarize_adaptive(state)
# Prior-only exposure checkpoint: B=0 commits no comparisons.
B <- c(0, 0.5, 1, 2)
```

```
data.frame(B = B, target_pairs = floor(B * length(state$item_ids) / 2))
```

```
adaptive_results_history
```

Adaptive results history ready for BT fitting

Description

Adaptive results history ready for BT fitting

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 accepted by `fit_bt_model()` (object1, object2, result). Pass the result directly to `fit_bt_model()`; it is already in the output format of `build_bt_data()` and needs no further conversion. 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

`fit_bt_model()`, `build_bt_data()`, `adaptive_step_log()`, [integrated CJ workflow](#)

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)
```

adaptive_round_log	<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_cores_available, mcmc_parallel_chains_requested, mcmc_concurrency_budget, mcmc_concurrency_used, mcmc_cmdstanr_version. Requested parallel chains are NA for automatic scheduling; the budget and usage count CPU slots across chains and threads. Historical fits without allocation metadata report NA for those fields.
- 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, pairing_strategy, stage_committed_so_far, stage_quota. Direct strategies use direct_pairing rather than hybrid stages; hybrid-only diagnostics are missing where inapplicable.
- 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, target_distance. For direct strategies, i_id is the focal item and p_ij is the pre-judgment TrueSkill probability for presented A over B. target_distance is missing for random selection.
- 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
<i>Live Anthropic (Claude) comparison for a single pair of samples</i>

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: <ul style="list-style-type: none"> • "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. <p>When reasoning = "none":</p> <ul style="list-style-type: none"> • Omitted temperature and top_p values are not sent, so the model/provider defaults apply. • max_tokens = 768 unless you supply max_tokens. <p>When reasoning = "enabled" (extended thinking), the Anthropic API imposes additional constraints:</p> <ul style="list-style-type: none"> • 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_typesafe_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [typesafe_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 = tmpl,
  reasoning    = "none",
  include_raw  = FALSE
)

res_claude$better_id

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

```

    prompt_template = 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

<code>requests</code>	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> .
<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

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_errors()`, `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 = tmpl
)

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

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

## End(Not run)
```

anthropic_download_batch_results

Download Anthropic Message Batch results (.jsonl)

Description

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

Usage

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

Arguments

<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`.

Retrieval retries

Batch metadata and result-file GET requests retry HTTP 408, 429, all 5xx responses, and transport failures, with at most three total HTTP attempts per GET. Valid `Retry-After` seconds or HTTP dates take precedence; otherwise retries use exponential backoff starting at 0.5 seconds plus up to 0.25 seconds of jitter, capped at 30 seconds. Other HTTP errors fail immediately. Exhaustion raises the original error with the additional class `pairwiseLLM_batch_retry_exhausted`. These retries retrieve the same batch; they do not resubmit comparisons or create scientific failed-attempt rows.

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_errors()`, `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)
```

anthropic_get_batch	<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

<code>batch_id</code>	Character scalar giving the batch ID (for example <code>"msgbatch_01HkcTjaV5uDC8jWR4ZsDV8d"</code>).
<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> .

Value

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

Retrieval retries

Batch metadata and result-file GET requests retry HTTP 408, 429, all 5xx responses, and transport failures, with at most three total HTTP attempts per GET. Valid Retry-After seconds or HTTP dates take precedence; otherwise retries use exponential backoff starting at 0.5 seconds plus up to 0.25 seconds of jitter, capped at 30 seconds. Other HTTP errors fail immediately. Exhaustion raises the original error with the additional class `pairwiseLLM_batch_retry_exhausted`. These retries retrieve the same batch; they do not resubmit comparisons or create scientific failed-attempt rows.

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_errors()`, `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

<code>batch_id</code>	Character scalar giving the batch ID.
<code>interval_seconds</code>	Polling interval in seconds. Defaults to 60.
<code>timeout_seconds</code>	Maximum total waiting time in seconds. Defaults to 24 hours (86400 seconds).
<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 "2023-06-01".
<code>verbose</code>	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.

Polling limits

HTTP retries occur within a logical status poll. Elapsed time includes retry waits, but `timeout_seconds` is checked between status requests; an in-flight GET and its retries can finish after that limit. Existing terminal-status and timeout return behavior is preserved.

Retrieval retries

Batch metadata and result-file GET requests retry HTTP 408, 429, all 5xx responses, and transport failures, with at most three total HTTP attempts per GET. Valid `Retry-After` seconds or HTTP dates take precedence; otherwise retries use exponential backoff starting at 0.5 seconds plus up to 0.25 seconds of jitter, capped at 30 seconds. Other HTTP errors fail immediately. Exhaustion raises the original error with the additional class `pairwiseLLM_batch_retry_exhausted`. These retries retrieve the same batch; they do not resubmit comparisons or create scientific failed-attempt rows.

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_errors(), 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)

```

bootstrap_bt_model	<i>Schedule-aware parametric bootstrap bias correction for Bradley-Terry scores</i>
--------------------	---

Description

Simulate binary comparisons from fitted BT scores, refit the same estimator, and estimate itemwise bias. Adaptive runs repeat the actual package scheduling algorithm, including changes driven by simulated earlier outcomes.

Usage

```

bootstrap_bt_model(
  object,
  mode,
  n_rep,
  seed,
  schedule = NULL,
  initial_state = NULL,
  budget = NULL,
  estimator = NULL,
  estimator_args = list(),
  btl_config = NULL,
  schedule_fit_fn = NULL,
  min_success = n_rep,
  workers = 1L,
  keep = c("summary", "theta", "full"),
  executor = NULL
)

```

Arguments

object	A fit from <code>fit_bt_model()</code> or a named finite numeric vector of initial/generating BT scores. Item IDs must be unique. The initial estimate also supplies the generating probabilities; a separate generating model is not fitted. Positional, lapse, and tie models are outside this interface.
mode	Required: "fixed" for an outcome-independent frozen schedule or "adaptive" to rerun within-set selection. Do not freeze an observed outcome-dependent adaptive schedule and describe it as schedule-aware.
n_rep	Required integer number of bootstrap replicates, at least two. Choose this for the precision and cost of your application.
seed	Required integer master seed, from zero to <code>.Machine\$integer.max</code> .
schedule	Fixed mode: a data frame with <code>object1/object2</code> or <code>A_id/B_id</code> . Row order, presentations and repeated pairs are retained. Other columns, including observed outcomes, are ignored. If omitted, recover <code>object\$comparisons</code> when available.
initial_state	Adaptive mode: a pristine <code>adaptive_rank_start()</code> state, before any comparison attempts or refits. Its connected $N - 1$ initial pairs, initial presentations, predictive initialization, pairing strategy and constraints remain fixed. Predictive scores must be outcome-independent. Sparse reservoir membership and presentations are retained, but original reservoir outcomes are not used. Sessions are never written to disk.
budget	Total number of committed comparisons, including initialization. Required in adaptive mode; in fixed mode defaults to the number of schedule rows and must equal that count.
estimator	"alpha", "brglm2", or a function with arguments <code>bt_data</code> and <code>item_ids</code> , plus any <code>estimator_args</code> . A function returns a named finite theta vector or a fit with a theta data frame (ID, theta). It must raise an error for unsuccessful estimation. Reported failed convergence/uncertainty diagnostics invalidate a returned fit. For alpha/Firth fitted objects, recover the original estimator/settings and reject conflicting overrides. Other inputs require an explicit estimator. No automatic engine fallback.
estimator_args	Named estimator arguments. Alpha requires an explicit alpha; numerical controls follow <code>fit_bt_model()</code> .
bt1_config	Adaptive scheduling-refit configuration, passed to <code>adaptive_rank_run_live()</code> . Defaults to the saved configuration or canonical defaults. This is separate from the final estimator. Refits are never silently disabled: the default scheduling fitter requires optional CmdStan. An explicitly larger-than-budget <code>refit_pairs_target</code> describes a design without scheduled refits during this budget.
schedule_fit_fn	Optional adaptive scheduling fitter with the existing <code>function(state, config)</code> posterior-fit contract. Omitted means the canonical Bayesian fitter. Bootstrap supplies deterministic <code>config\$cmdstan\$seed</code> at each refit. Custom functions must honor it for external RNGs and must be deterministic, provider-free, and independent of mutable external state.

<code>min_success</code>	Minimum successful replicates, from two through <code>n_rep</code> . Default requires all replicates. Failures are never included in averages.
<code>workers</code>	Positive worker count, default one (serial). More than one uses optional <code>future</code> and <code>future.apply</code> with a temporary multisession plan. The previous plan and caller RNG state are restored.
<code>keep</code>	Retention: "summary" (default) keeps summaries and compact diagnostics; "theta" also keeps aligned replicate estimates; "full" additionally keeps comparisons, adaptive states and detailed failure diagnostics (which can include dense matrices). Full artifacts can be large.
<code>executor</code>	Optional ordered map function with arguments <code>X</code> , <code>FUN</code> , and <code>...</code> , for example an adapter to a caller-owned worker pool. It must apply <code>FUN</code> to each element of <code>X</code> , forwarding <code>...</code> , and return a list of results in input order, without retries or seed replacement. <code>workers</code> bounds the number of chunks submitted at once; the caller must allocate consistent CPU/memory capacity, including any native sampler chains. <code>NULL</code> retains the existing serial/temporary-multisession behavior. A supplied executor owns its resources; this function neither changes its plan nor closes its pool.

Details

Each outcome has probability `plogis(theta[A] - theta[B])`. Initial scores and refits are aligned by ID and centered to sum zero. With successful refits only, `bias = mean(theta_boot) - theta_initial` and `theta_corrected = theta_initial - bias`. Corrections can change rankings and need not improve every item or every application.

Adaptive p50, Pollitt-inspired, hybrid and random strategies use the canonical live runner with a synthetic judge. Initialization stays frozen, while later selector randomness changes by replicate. Scheduled refits, TrueSkill updates, hybrid identifiability/tapering and constraints remain active. The existing `max_pairs_after_stop` continuation control is set to budget so statistical stopping cannot truncate the requested fixed budget. Terminal exhaustion is a failed replicate; it never triggers replacement with another strategy.

RNG algorithm is Mersenne-Twister, with Inversion normals and Rejection sampling. For replicate `b` and stage `s` (selector, outcomes, scheduling refits, final estimator, numbered 1–4), the integer seed is `max(1, floor((seed * 1000003 + b * 10007 + s * 101 + 304) %% 2147483647))`. Scheduling-refit seeds use the canonical stage derivation again, with the scheduling seed, committed comparison count, stage 1, and offset zero. Serial and parallel results use the same seeds and aggregation order. Exact reproducibility requires compatible R/package/engine versions and deterministic callbacks. There is no checkpoint/resume API; restarting repeats the run. Built-in final estimators reuse immutable item contrasts within this call. Fixed schedules also reuse pair indexes and designs; `alpha` retains its all-pairs pseudo-count design, including unobserved pairs. Outcome counts, estimation and numerical validity checks are recomputed for every replicate. Worker chunks hold at most eight replicates for summary/theta retention and two for full retention. Designs are constructed inside each chunk, never transferred from the parent or retained in a global cache. An executor is an execution resource and is not stored in scientific provenance. Its transport errors propagate without replacement replicates or a serial fallback.

`bootstrap_sd` describes replicate-estimate dispersion, whereas `mcse_bias` is `bootstrap_sd / sqrt(n_success)`: simulation error in the estimated bias. Neither is automatically a standard error for the bias-corrected estimator. This function does not construct confidence intervals or corrected SSR.

An unmet `min_success` raises `pairwiseLLM_bt_bootstrap_error` with a `result` field containing diagnostics and missing bias/corrected scores. Tolerated failures raise a warning and remain in the diagnostic table. Numerical warnings from successful fits are recorded per replicate and signaled once at completion.

Value

A `pairwiseLLM_bt_bootstrap` list containing `theta` (ID, initial theta, bootstrap mean, bias, corrected theta, bootstrap SD and Monte Carlo error), `n_success`, `n_failed`, `status`, `replicates` (seeds, failures and schedule diagnostics), `failures` (available error diagnostics), optional `draws` and `artifacts`, and `provenance`. `Provenance` includes normalized generating parameters, `schedule/initial state`, `estimator/settings`, `master seed`, `RNG convention`, `scheduling configuration`, `callbacks` and `software versions`.

See Also

`fit_bt_model()`, `adaptive_rank_start()`, `adaptive_rank_run_live()` and the [integrated CJ workflow](#)

Other frequentist models: `build_bt_data()`, `build_elo_data()`, `fit_bt_model()`, `fit_elo_model()`, `predict.pairwiseLLM_bt_alpha()`, `predict.pairwiseLLM_bt_firth()`, `predict.pairwiseLLM_bt_lapse()`, `scale_separation_reliability()`, `summarize_bt_fit()`

Examples

```
dat <- data.frame(object1 = rep("a", 8), object2 = rep("b", 8),
                  result = c(rep(1, 6), rep(0, 2)))
fit <- fit_bt_model(dat, engine = "alpha", alpha = 1, verbose = FALSE)
boot <- bootstrap_bt_model(fit, mode = "fixed", n_rep = 20, seed = 304)
boot$theta
```

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

<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>	Anthropic Claude model name, for example "claude-sonnet-4-5", "claude-haiku-4-5", or "claude-opus-4-5".
<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>reasoning</code>	Character scalar indicating whether to allow extended thinking; one of "none" or "enabled". See details above.
<code>custom_id_prefix</code>	Prefix for the custom_id field. Defaults to "ANTH" so that IDs take the form "ANTH_<ID1>_vs_<ID2>".
<code>...</code>	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 ≤ thinking_budget_tokens < 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_errors()`, `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_bt_data

Build Bradley-Terry comparison data from pairwise results

Description

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

Usage

```
build_bt_data(results)
```

Arguments

`results` A data frame or tibble with 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: [bootstrap_bt_model\(\)](#), [build_elo_data\(\)](#), [fit_bt_model\(\)](#), [fit_elo_model\(\)](#), [predict.pairwiseLLM_bt_alpha\(\)](#), [predict.pairwiseLLM_bt_firth\(\)](#), [predict.pairwiseLLM_bt_lapse\(\)](#), [scale_separation_reliability\(\)](#), [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_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

<code>results</code>	A data frame or tibble containing columns ID1, ID2, and better_id.
<code>phase</code>	Length-1 phase label for all rows. Must be one of "phase1", "phase2", or "phase3". Defaults to "phase2".
<code>backend</code>	Length-1 backend label to record in output metadata.
<code>model</code>	Length-1 model label to record in output metadata.
<code>iter_start</code>	Integer starting value for iter. Defaults to 1L.
<code>received_at_start</code>	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_elo_data

Build EloChoice comparison data from pairwise results

Description

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

Usage

```
build_elo_data(results)
```

Arguments

results A data frame or tibble with 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: [bootstrap_bt_model\(\)](#), [build_bt_data\(\)](#), [fit_bt_model\(\)](#), [fit_elo_model\(\)](#), [predict.pairwiseLLM_bt_alpha\(\)](#), [predict.pairwiseLLM_bt_firth\(\)](#), [predict.pairwiseLLM_bt_lapse\(\)](#), [scale_separation_reliability\(\)](#), [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> . <ul style="list-style-type: none"> 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.

top_p	Optional nucleus sampling parameter. If NULL, omitted.
top_k	Optional top-k sampling parameter. If NULL, omitted.
max_output_tokens	Optional integer. If NULL, omitted.
service_tier	Gemini Developer API service tier. Use "standard" (default) or NULL for provider default behavior, or "flex" / "priority" to encode the documented Gemini serviceTier request field.
include_thoughts	Logical; if TRUE, sets thinkingConfig.includeThoughts = TRUE so that Gemini returns visible chain-of-thought. For most pairwise scoring use cases this should remain FALSE.
...	Additional parameters. store accepts NULL or one non-missing logical value. TRUE enables and FALSE disables request logging for every batch request. Omission or NULL preserves the provider/project logging default. This controls Gemini Developer API logging, not all data retention. Any thinking_budget 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_errors()`, `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",
store = NULL,
max_output_tokens = NULL,
prompt_caching = NULL,
...
)

```

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".
<code>temperature</code>	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.
<code>top_p</code>	Optional top-p parameter. If NULL, it is omitted so the model/provider default applies.
<code>logprobs</code>	Optional logprobs parameter.
<code>reasoning</code>	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".
<code>include_thoughts</code>	Logical; if TRUE and using responses endpoint with reasoning, requests a summary. Defaults reasoning to "low" for GPT-5 series models if not specified.
<code>request_id_prefix</code>	String prefix for custom_id; the full ID takes the form "<prefix>_<ID1>_vs_<ID2>".
<code>store</code>	Optional logical scalar for either endpoint. TRUE and FALSE are forwarded unchanged; omitted or NULL leaves the field absent.

max_output_tokens	Optional positive whole numeric scalar, at most <code>.Machine\$integer.max</code> , for the Responses endpoint only. Includes visible output and reasoning tokens. Omitted or NULL leaves the field absent.
prompt_caching	Batch caching policy: NULL (model-aware default), "disabled", or "implicit". See the compatibility rules below.
...	Reserved; must be empty. Raw provider caching controls are not accepted. Use prompt_caching instead.

Details

Invalid controls fail even for empty pairs. Responses are stored for later API retrieval by default; set `store = FALSE` to disable response storage. For Chat Completions, `store` controls storage for distillation or evaluations. Omitting it preserves that endpoint's provider default. This parameter does not control Batch input/output/error file retention and does not imply zero data retention. See <https://developers.openai.com/api/docs/guides/your-data>.

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.

Batch prompt caching

The default disables prompt caching for these verified model IDs: `gpt-5.6-luna`, `gpt-5.6-terra`, `gpt-5.6-sol`, `gpt-6-luna`, `gpt-6-sol`, `gpt-6-astra`, and `gpt-6.1-sol`. Each request body gets `prompt_cache_options = list(mode = "explicit")`, with no breakpoints, on both endpoints. Set `prompt_caching = "implicit"` to omit that field and retain provider caching when prefix reuse is expected to be worthwhile.

Recognized earlier OpenAI models retain their historical request bodies when the policy is omitted or NULL; explicitly disabling caching on those models errors. Unknown IDs (including unlisted snapshots, future models and custom aliases) require an explicit "implicit" opt-in or a verified model ID. Support is an offline allowlist; version numbers and snapshot suffixes never establish support for disabling caching. Older GPT-3.5/4, GPT-5 through GPT-5.5 and o-series naming forms, including recognized legacy date-suffixed names, retain provider defaults.

Invalid policies and manual cache controls fail before submission, even with zero pairs. Raw `prompt_cache_options`, `prompt_cache_breakpoint`, `prompt_cache_key`, `retention` and `TTL` controls are not accepted by these builders. Keys affect cache routing/accounting; `retention` and `TTL` affect lifetime; `store = FALSE` affects response storage. None substitutes for disabling implicit cache breakpoints. Prompt text and keys are not rewritten. This policy affects future package-built Batch requests only, not live requests or already submitted jobs. See the [prompt-caching guide](#) and vignette("advanced-batch-workflows") for usage-based cost comparisons.

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_errors(), 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",
  store = FALSE,
  max_output_tokens = 1024
)

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, VERTEX_API_KEY, TOGETHER_API_KEY, and TYPESAFE_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", "typesafe".

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

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\(\)](#), [typesafe_compare_pair_live\(\)](#)

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_typesafe_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [typesafe_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: <ul style="list-style-type: none"> • 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
```

Description

Collect and resume OpenAI Decisions and TypeSafe Jev comparisons through `submit_llm_pairs()`, `submit_openai_pairs_live()` or `submit_typesafe_pairs_live()`. Each pair occurrence is one live request; these endpoints do not use provider-managed Batch jobs.

Execution controls

The bulk wrappers accept `execution_control`, a uniquely named list. Unknown names, invalid scalars and unsupported provider options fail before HTTP. `parallel = FALSE` or `workers = 1` selects sequential execution. `parallel = TRUE`, `workers > 1` uses optional **promises** and **later** through `httr2`'s experimental public asynchronous API. Missing packages cause an informative error before requests; no packages are installed automatically. Workers bound simultaneous in-flight requests, not R processes. One private event loop and curl pool belong to each session and are cleaned up on exit. The coordinator explicitly implements retries and admission; asynchronous `httr2` retry/throttle policies are not assumed to apply.

Supported controls and defaults are:

`requests_per_minute = NULL` Optional positive integer maximum transmission reservations in a rolling 60-second window.

`tokens_per_minute = NULL` Optional positive integer maximum estimated token reservations in that window. Requires `token_estimates`.

`token_estimates = NULL` Positive integer scalar or vector in input row order, including an output allowance. A reservation must fit the token limit. No tokenizer or missing provider usage is guessed. This bounds reservations, not actual usage when estimates are too small.

`max_attempts = 3L` Persistent total send-reservation budget per occurrence, including initial, transient and ambiguous-recovery sends. It is fixed in the saved manifest; restarting does not replenish it.

`timeout_seconds = 120` Positive finite timeout per transmission.

`retry_ambiguous = TRUE` Allow at most one automatic reissue of an interrupted occurrence across all restarts, also within `max_attempts`. `FALSE` leaves ambiguous work unresolved. Toggling it never replenishes the consumed allowance.

`lock_timeout_ms = 10000` Nonnegative finite milliseconds to wait for an exclusive saved-run lock before failing without provider calls.

Rate limits are caller configuration, not assumed provider quotas. Recent reservations, retry deadlines and shared provider cooldowns survive restart. Limits apply to this saved run; other sessions sharing an account can consume the same provider quota. Limits and timeout may change on resume without changing scientific identity; prior reservations remain counted.

Retries and ambiguous requests

Bulk OpenAI retries HTTP 408, 409, 429 and all 5xx statuses; TypeSafe retries 408, 429 and all 5xx, including 529. Recognized OpenAI insufficient-quota or billing-limit errors are terminal. Timeouts receive bounded transient retries; other transport failures and permanent HTTP errors remain unresolved. Exponential delays start at 0.5 seconds, double up to 30 seconds and add uniform jitter of up to 0.25 seconds. Valid `Retry-After` seconds or HTTP dates are a minimum delay, with

shared cooldowns for 429, 503 and 529. Invalid headers use backoff. Single-pair retry defaults are unchanged.

A send reservation is committed before transmission. An interruption before transmission may therefore conservatively consume an attempt. Reservations without a committed response have an unknown provider outcome. Recovery first checks local committed evidence. The currently supported endpoint contracts provide no documented remote result lookup/idempotency mechanism: reconciliation is recorded as unsupported, without speculative retrieval requests. One automatic ambiguous reissue is permitted by default, across all restarts. Exhaustion leaves the occurrence in `failed_pairs` for explicit operator action; an operator can deliberately start a separate run containing only unresolved source pairs. Preserve the old journal rather than editing it or resetting budgets. Timeout retries can also incur duplicate charges. Neither locking nor checkpoints establish exactly-once provider execution.

Malformed/refused decisions, incomplete probabilities, invalid confidence and choice/probability inconsistencies are recorded without automatic retry and never become completed evidence. Only normalized valid results enter modeling.

Request identity

All four input ID/text columns must be nonmissing character columns. Supplied `pair_uid` values must be unique and nonempty. Otherwise deterministic scientific-fingerprint IDs plus occurrence suffixes distinguish repeats and reversed presentations. Supply explicit IDs when scheduled occurrences need independent labels. IDs are never matched by row position alone.

The saved workset must have the same occurrence membership, scientific payload and provider wire controls. Matching occurrences can be reordered. Adding or removing occurrences, or changing texts, orientation, trait, instructions, choices, model/provider/endpoint or OpenAI cache controls requires a separate save path. Omitted decision prompts select the exact `decision_cj_v1` wording; custom instructions follow [decision_request](#). Generative templates fail.

Every current request is rebuilt and validated even when its result is already saved. The shared scientific validator checks body, ordered evidence hashes, approved question core, provenance and fingerprint. Serialization and attached raw UTF-8 bytes are checked before every transmission, including retries. `scientific_payload_sha256` hashes the serialized scientific body; `wire_body_sha256` hashes the exact transmitted JSON body, including supported cache controls. These digests exclude authentication headers and timeout. Digests detect identity/integrity mismatches, not malicious modification with recomputed checksums. Requested and provider-reported resolved models remain separate. Moving aliases may change weights; saved results retain the originally reported resolution, and no availability or alias-resolution calls are made.

Persistence and output

A decision `save_path` names a CSV export and requires optional **filelock**. The dedicated `<save_path>.lock` is locked before checkpoints are read and held through all asynchronous calls, retries and writes. The OS lock releases on exit/process termination; its file is retained. Only one writer is admitted. `<save_path>.checkpoint/` is the authoritative lossless RDS journal. Its immutable checksummed manifest and sequenced records are written through same-directory temporary files, flushed/closed, then renamed to new final names. Each response is validated and committed immediately, independently of other in-flight work. Completed committed occurrences are never automatically submitted again. Uncommitted temporary files are ignored; corrupt committed files, missing interior records

and unsupported schemas stop execution before HTTP. CSV alone cannot establish completion and is not imported as a resume journal.

CSV contains valid results with stable scalar columns, including all probability and provenance columns. Numeric values use 17 significant digits to round-trip R doubles with a correctly rounded decimal reader. Some readers (including base numeric conversion on macOS ARM in qualification) can change the last bit. For exact numeric CSV recovery, read columns as character and decode probability cells with `jsonlite::parse_json()`; use RDS for exact native recovery. Read IDs as character to preserve leading zeros. A missing or torn CSV export is rebuilt from RDS on resume. Journal envelopes are RDS lists with a SHA-256 and serialized raw data; `unserialize(readRDS(record)$data)` exposes the versioned event and its state/attempt records. Treat these as trusted local files, not a public cross-version interchange format.

The returned list retains results, `failed_pairs` and `failed_attempts`. Results preserve [decision_results](#) fields, exact parsed R double probabilities, full `probabilities_json`, independent confidence, usage and model provenance. Missing usage/resolution stays missing; no price or cost estimate is fabricated. Additional diagnostics include payload digests, controls, local `attempt_id`, `attempt_count`, `ambiguous_reissues`, `possible_duplicate_charge`, `elapsed_seconds` since admission, and `recovery_status`. Failed attempts retain unique local IDs, reason (`initial`, `transient_retry`, `ambiguous_recovery`), timestamps, available original/new provider request IDs, provenance, and reconciliation status. Provider IDs unknown at interruption remain missing. Successful outcomes and failed attempts remain separate; history is not re-added on every restart. Probabilities never become BT/adaptive weights, fractional wins or extra comparisons.

No durability is promised without `save_path`. Saved-run guarantees concern process interruption on a filesystem supporting OS locks and atomic rename; they are not power-loss/fsync guarantees. Network/cloud-synchronized storage and cross-machine locking require filesystem guarantees beyond this package.

Privacy

Samples go to the selected provider. Normal messages and scalar checkpoints omit API keys and sample text. Provenance retains complete custom instructions and trait definitions, which can themselves be sensitive. CSV excludes input text columns and arbitrary caller metadata. `include_raw = TRUE` explicitly retains parsed provider responses in RDS and may contain sensitive content; `FALSE` omits them from new persisted records, including failures. Existing opted-in records are not erased. Increasing raw retention on resume cannot recreate responses previously omitted and never triggers resubmission. In-memory failure objects may contain raw responses even with `include_raw = FALSE`. Explicitly saving or serializing the complete returned object (for example with `saveRDS()`) can persist sensitive text. Successful normalized results can also retain source text columns. The default journal/export privacy policy does not sanitize caller-created stores. Keep journals and returned diagnostics private.

See Also

[submit_llm_pairs\(\)](#), [decision_request](#), [decision_results](#)

Examples

```
## Not run:
# Requires provider credentials; incurs usage charges.
```

```

pairs <- data.frame(ID1 = c("A", "B"), ID2 = c("B", "A"),
                   text1 = c("Synthetic A.", "Synthetic B."),
                   text2 = c("Synthetic B.", "Synthetic A."),
                   pair_uid = c("forward", "reverse"))
result <- submit_llm_pairs(
  pairs, model = "jev-1.13.0", backend = "typesafe",
  trait_name = "Organization", trait_description = "Logical sequencing.",
  parallel = TRUE, workers = 4, save_path = "decisions.csv",
  execution_control = list(requests_per_minute = 60,
                           retry_ambiguous = FALSE)
)
# Repeat the same call to resume; preserve its .checkpoint directory.
result$failed_pairs
result$failed_attempts

## End(Not run)

```

ensemble_warm_start_algorithms

Average algorithms trained on the same task

Description

Unlike `ensemble_warm_start_models()`, this constructor requires auditable shared training outcomes and evaluation partitions. Validation averages the untouched outer-held-out calibrated predictions, not full-fit predictions or component validation metrics.

Usage

```
ensemble_warm_start_algorithms(...)
```

Arguments

... At least two explicitly named, fully audited format-3 `fit_warm_start_model()` objects. Names must be nonblank and unique. Load saved components before calling this constructor. Nested ensembles fail.

Details

Components must share the exact task, ordered IDs, original outcomes, schema, CV plan and outer observed values/folds. Every component requires learned `oof_linear` calibration. Legacy and reduced models cannot establish the necessary audit evidence and must not be used to construct a new ensemble. Deployment averages each component's calibrated prediction with equal weight. There are no learned weights, automatic component selection or ensemble-level recalibration. Validation calibration intercept/slope are diagnostics only. Component names identify members; distinct engine names are not required.

`prepare_warm_start_model()` can reduce an existing ensemble to summary-only audit status. Its portable component payloads and identity survive, but the stored validation summaries cannot then be recomputed. Between-component SD is diagnostic disagreement, never a Bayesian prior SD.

Value

A standalone pairwiseLLM_warm_algorithm_ensemble, format 1, with artifact_type = "algorithm_ensemble", ordered components, common schema, outcome and CV identity, and honest nested-CV ensemble validation.

See Also

`make_warm_start_cv_plan()`, `make_warm_start_prior()`, `predict.pairwiseLLM_warm_algorithm_ensemble()`

Other adaptive warm start: `ensemble_warm_start_models()`, `extract_warm_start_features()`, `fit_warm_start_model()`, `make_warm_start_cv_plan()`, `make_warm_start_prior()`, `pairwiseLLM_warm_model`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_ensemble()`, `predict.pairwiseLLM_warm_model()`, `prepare_warm_start_model()`, `register_warm_start_model()`, `save_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_ensemble()`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_coefficients()`, `warm_start_feature_schema()`, `warm_start_python_status()`

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("pls", quietly = TRUE) && requireNamespace("withr", quietly = TRUE)) {
  local({
    withr::local_seed(259L)
    fields <- warm_start_feature_schema()$feature
    x <- as.data.frame(matrix(runif(20 * length(fields)), nrow = 20))
    names(x) <- fields
    x$n_tokens <- 11:30
    x <- data.frame(item_id = as.character(1:20), x)
    attr(x, "warm_start_schema") <- "writing_features_v1"
    theta <- x$n_tokens - 3 * x$token_length_mean
    plan <- make_warm_start_cv_plan(x$item_id, theta, "synthetic", seed = 259L)
    en <- fit_warm_start_model(x$item_id, theta, "synthetic", features = x,
      cv_plan = plan, alpha_grid = c(0, 1))
    pls <- fit_warm_start_model(x$item_id, theta, "synthetic", features = x,
      cv_plan = plan, engine = "pls", engine_control = list(ncomp = 1L))
    ensemble <- ensemble_warm_start_algorithms(elastic_net = en, pls = pls)
    print(ensemble)
    summary(ensemble)$validation$metrics
    predictions <- predict(ensemble, x)
    make_warm_start_prior(predictions, prior_sd = 0.5)
  })
}
```

ensemble_warm_start_models

Combine independently trained warm-start models

Description

An ensemble averages calibrated standardized predictions from separate task models. Each task standardizes its own outcome, so original BT/BTL scales need not be linked. Callers are responsible for independently sourced training data; task labels cannot establish independence.

Usage

```
ensemble_warm_start_models(...)
```

Arguments

... Individual calibrated models, path strings, or reference lists with path, or name and optional source, as in [load_warm_start_model\(\)](#). Supply argument names for explicit component identities. Unnamed arguments receive component_<position>; duplicate final names and blank supplied names are rejected. Order and supplied names are preserved. Nested ensembles fail.

Details

Components must have learned OOF calibration and compatible frozen schemas. Legacy model formats 1/2 and full or summary-only format-3 models can be mixed. Each component keeps its own preprocessing, calibration, and training metadata. References are loaded once; prediction uses stored components. Character references always mean paths, never implicit registry names. Registry names retain existing normalization and source ambiguity rules; component names do not use registry normalization. Custom or learned weights are not supported.

Repeated models and task labels remain accepted; neither establishes independent training data. Validation reports component metrics only. Use [ensemble_warm_start_algorithms\(\)](#) for aligned validation of algorithms trained on one assessment.

[prepare_warm_start_model\(\)](#) can add ensemble metadata or explicitly reduce all component audits. Ordinary save/load is lossless. Ensemble format 1 is independent of component formats and remains 1 after reduction.

Value

A portable pairwiseLLM_warm_ensemble list with ensemble format version 1, named components, common schema, original features, standardized outcome definition/sample-SD convention, and weighting = "equal".

See Also

[fit_warm_start_model\(\)](#), [predict.pairwiseLLM_warm_ensemble\(\)](#), [ensemble_warm_start_algorithms\(\)](#), [warm_start_coefficients\(\)](#), [summary.pairwiseLLM_warm_ensemble\(\)](#)

Other adaptive warm start: [ensemble_warm_start_algorithms\(\)](#), [extract_warm_start_features\(\)](#), [fit_warm_start_model\(\)](#), [make_warm_start_cv_plan\(\)](#), [make_warm_start_prior\(\)](#), [pairwiseLLM_warm_model](#), [predict.pairwiseLLM_warm_algorithm_ensemble\(\)](#), [predict.pairwiseLLM_warm_ensemble\(\)](#), [predict.pairwiseLLM_warm_model\(\)](#), [prepare_warm_start_model\(\)](#), [register_warm_start_model\(\)](#), [save_warm_start_model\(\)](#), [summary.pairwiseLLM_warm_algorithm_ensemble\(\)](#), [summary.pairwiseLLM_warm_ensemble\(\)](#), [summary.pairwiseLLM_warm_predictions\(\)](#), [warm_start_coefficients\(\)](#), [warm_start_feature_schema\(\)](#), [warm_start_python_status\(\)](#)

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  local({
    # Synthetic features illustrate the interface, not predictive validity.
    example_features <- function(seed) {
      withr::local_seed(seed)
      fields <- warm_start_feature_schema()$feature
      x <- as.data.frame(matrix(runif(15 * length(fields)), nrow = 15))
      names(x) <- fields
      x$n_tokens <- 11:25
      x$token_length_mean <- 2 + 10 * x$token_length_mean
      x$token_length_std <- 0.2 + x$token_length_std
      x$dale_chall_readability_score <- 5 + 20 * x$dale_chall_readability_score
      x <- data.frame(item_id = as.character(1:15), x)
      attr(x, "warm_start_schema") <- "writing_features_v1"
      x
    }
    features <- example_features(3103)
    theta <- 10 + 0.4 * features$n_tokens - 2 * features$token_length_mean
    # A small alpha grid keeps this example fast; the default has 41 values.
    model <- fit_warm_start_model(features$item_id, theta, "synthetic-a",
      features = features, alpha_grid = c(0, 1))
    features_b <- example_features(3104)
    theta_b <- 30 + features_b$n_tokens - 3 * features_b$token_length_mean
    model_b <- fit_warm_start_model(features_b$item_id, theta_b, "synthetic-b",
      features = features_b, alpha_grid = c(0, 1))
    ensemble <- ensemble_warm_start_models(assessment_a = model, assessment_b = model_b)
    print(ensemble)
  })
}
```

ensure_only_ollama_model_loaded

Ensure only one Ollama model is loaded in memory

Description

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

Usage

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

Arguments

model	Character scalar giving the Ollama model name that should remain loaded (for example "mistral-small3.2:24b", "qwen3:32b", "gemma3:27b"). All other models currently reported by <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 ID1, text1, ID2, text2. 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 .
<code>backend</code>	Backend for the pilot run; one of "openai", "anthropic", "gemini", "vertex", or "together".
<code>endpoint</code>	OpenAI endpoint; one of "chat.completions" or "responses". Ignored for other backends.
<code>mode</code>	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.
<code>n_test</code>	Number of pilot pairs to run live. Defaults to 25 or fewer if fewer pairs are supplied.
<code>test_strategy</code>	Strategy for selecting pilot pairs: "stratified_prompt_bytes" (default), "random", or "first".
<code>seed</code>	Optional integer seed used for pilot sampling when test_strategy is not "first".
<code>cost_per_million_input</code>	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.
<code>cost_per_million_output</code>	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.
<code>batch_discount</code>	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.
<code>budget_quantile</code>	Quantile used for the "budget" output-token estimate for remaining pairs. Defaults to 0.9 (p90).
<code>return_test_results</code>	Logical; if TRUE, include pilot results in the returned object so you can reuse them and avoid paying twice.
<code>return_remaining_pairs</code>	Logical; if TRUE, include the remaining pairs (excluding pilot pairs) in the returned object.
<code>...</code>	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")
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
  est$summary

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

  ## End(Not run)

```

```
evaluate_rubric_predictions
```

Evaluate rubric predictions on observed ordered labels

Description

Compute rubric prediction metrics conditional on completed CJ point scores. Normalized ranked probability score (RPS) is the primary ordinal probability metric. Additional assumption diagnostics run only when requested here.

Usage

```

evaluate_rubric_predictions(
  object,
  rubric,
  newdata = NULL,
  hard_score = c("median", "mode"),
  bins = 10L,
  diagnostics = FALSE
)

```

Arguments

object	A fitted pairwiseLLM_rubric_calibration object.
rubric	Data frame with unique item_id and rubric_score, optionally trait. Labels align by ID and use the fitted level order. Missing labels are excluded and counted. Evaluation need not represent every category.

newdata	As in <code>predict.pairwiseLLM_rubric_calibration()</code> : NULL predicts original items; linked targets require accepted Phase B results.
hard_score	Hard category rule, default "median", optionally "mode".
bins	Positive integer number of equal-width bins (default 10) for cumulative calibration summaries. Empty bins are omitted; ties stay together.
diagnostics	Logical; request additional training-model diagnostics. Default FALSE avoids diagnostic refits. Evaluation labels never refit the production model or enter its training-model diagnostics.

Details

For K levels, RPS for item i is $\sum((F[i, k] - I(Y[i] \leq k))^2) / (K - 1)$ over boundaries $k = 1, \dots, K-1$. This is normalized RPS, not the unnormalized sum; lower is better and zero is perfect. Log loss is $-\log(p[i, Y[i]])$. Observed-category probabilities are floored at `.Machine$double.xmin` before taking logs, solely during evaluation to handle zero/underflow probabilities. Stored predictions are unchanged. Both metrics are averaged over observed labels.

Hard metrics are exact accuracy, within-one accuracy, mean absolute category error, and quadratic weighted kappa. Distances use internal indices $1:K$, not numeric gaps between user labels. Kappa is one minus observed divided by expected weighted disagreement, with weights $(i-j)^2 / (K-1)^2$ and independent empirical marginals. Zero expected disagreement returns NA with a reason. Percentile scoring supplies only hard metrics; no probabilities are invented.

Cumulative summaries compare $P(Y \leq k)$ with observed cumulative frequencies within equal-width probability bins on $[0, 1]$ and theta bins over the evaluated range. Internal edges enter the higher bin; the upper endpoint stays in the last bin. A constant theta range occupies one bin. These descriptive tables require no plotting dependency and do not constitute formal goodness-of-fit tests. Training-label overlap is reported; apparent performance is not out-of-sample validation.

Requested diagnostics expose stored numerical/uncertainty information and training-only cumulative summaries. For linear fits, an explicit threshold-varying `ordinal::clm` alternative supplies a proportional-odds likelihood-ratio diagnostic. Convergence/identification failures produce an unavailable status. A p-value below .05 triggers an exploratory review warning, not automatic selection. This test is distinct from functional-form assessment. Monotone common-effect checks are descriptive boundary-specific binomial fits against the estimated latent effect (expected cumulative slope -1), with no omnibus p-value or automatic adequacy threshold. Diagnostic failures preserve the fitted calibration. Optional backends are required only when used.

Internal rubric-label cross-validation holds out labels while keeping the completed CJ evidence and theta fixed. Each fold re-estimates calibration scaling and parameters from training labels alone. Linked-reference CV assesses reference calibration; externally labeled Phase B targets assess transport. Invalid training folds are reported without collapsing categories, and prevent an overall CV estimate. If resamples are used to choose methods or tuning, unbiased post-selection assessment requires an outer validation layer. No automatic method selection is supplied. Core fits condition on point theta; retained CJ uncertainty is not propagated through a posterior/bootstrap engine.

Value

A list with `metrics` (one-row tibble), `per_item` (predictions, observed labels, uncertainty, overlap flags and individual losses), `calibration` (cumulative summaries, or NULL for percentile), `diagnostics`, and `metadata`. Metrics are `n`, `rps`, `log_loss`, `exact_accuracy`, `within_one_accuracy`,

mae, and quadratic_weighted_kappa. Unavailable probability metrics are NA. Summary columns are group_by, boundary, rubric_level, bin, n, theta_min, theta_max, probability_min, probability_max, predicted, observed, and residual (observed minus predicted). Metadata records exclusions, training overlap, extrapolation, metric conventions, unavailable reasons, and original linking metadata when present. Requested diagnostics contain status, stored, functional_form, and common_effect; their statuses distinguish formal, descriptive and unavailable results. No input object or upstream CJ state is modified.

See Also

`fit_rubric_calibration()`, `predict.pairwiseLLM_rubric_calibration()`

Other rubric calibration: `fit_rubric_calibration()`, `predict.pairwiseLLM_rubric_calibration()`, `prepare_linked_rubric_reference()`

Examples

```
## Not run:
# Rubric labels align with items in an already completed CJ result.
if (requireNamespace("ordinal", quietly = TRUE)) {
  fit <- fit_rubric_calibration(completed_cj, training_labels,
    trait = "organization", levels = c("developing", "proficient", "advanced"))
  assessment <- evaluate_rubric_predictions(fit, evaluation_labels, diagnostics = TRUE)
  assessment$metrics
  assessment$calibration
  assessment$metadata$n_training_label_overlap
  assessment$diagnostics$common_effect
}

## 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-row/example_writing_samples.R` in the source repository.

Examples

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

```
example_writing_samples
```

Example dataset of writing samples

Description

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

Usage

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

Format

A tibble with 20 rows and 3 variables:

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

text Character string with the writing sample.

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

Source

Package-author-created synthetic examples. Reproduction code is in `data-row/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)
```

```
extract_warm_start_features
```

Extract frozen writing features for warm-start prediction

Description

Warm-start prediction uses text features to predict relative writing quality before collecting pairwise comparisons. This function extracts the frozen English writing features; it does not train a model or estimate BTL scores.

Usage

```
extract_warm_start_features(
  ids,
  texts,
  schema = "writing_features_v1",
  python = NULL
)
```

Arguments

<code>ids</code>	Unique, nonmissing character or finite numeric item IDs. IDs are returned as character strings; blank IDs are not allowed.
<code>texts</code>	A nonempty character vector of the same length as <code>ids</code> , without missing values. Empty strings are allowed. Text is never trimmed or normalized.
<code>schema</code>	The frozen schema identifier: "writing_features_v1" (default) or "writing_features_v2".
<code>python</code>	Optional path to an existing Python interpreter. With <code>NULL</code> , use an existing environment selected through reticulate. Automatic environment creation is disabled. Conflicting interpreter selections require correction or a fresh R session; this function never switches an initialized interpreter.

Details

Extraction requires optional reticulate and the audited Python 3.12.3 stack. See [warm_start_python_status\(\)](#) and the installed setup instructions located by `system.file("python", "README.md", package = "pairwiseLLM")`. Software and resources must be installed explicitly before extraction. Package loading, schema inspection, and later prediction from precomputed features do not require Python. The tested environment is Linux x86_64; Windows and macOS have not been validated. Features are not evidence of predictive validity.

The default English spaCy model, resource contents and package versions are checked before extraction. Missing resources fail without downloads. Entropy is divided by all spaCy tokens, including punctuation and whitespace, rather than the filtered `n_tokens` feature. Undefined values and valid zeros follow `warm_start_feature_schema()`. Zero-vector coherence preserves upstream values and warnings. No feature is imputed or replaced with zero. Version 2 enables TextDescriptives readability and preserves undefined filtered-token summaries and second-order coherence (fewer than three sentences). `textstat` methods use their audited defaults, including first-100-word Linsear–Write and unique difficult-word counts. Valid negative readability scores remain unchanged. Model preprocessing learns missing-value handling from each training split.

Value

A tibble with character `item_id` and the 20 (v1) or 46 (v2) numeric features in schema order, in the requested ID order. The `warm_start_schema` attribute records the schema identifier. Document-level undefined values remain NA.

See Also

`warm_start_python_status()`, `warm_start_feature_schema()`, `fit_warm_start_model()`

Other adaptive warm start: `ensemble_warm_start_algorithms()`, `ensemble_warm_start_models()`, `fit_warm_start_model()`, `make_warm_start_cv_plan()`, `make_warm_start_prior()`, `pairwiseLLM_warm_model`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_ensemble()`, `predict.pairwiseLLM_warm_model()`, `prepare_warm_start_model()`, `register_warm_start_model()`, `save_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_ensemble()`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_coefficients()`, `warm_start_feature_schema()`, `warm_start_python_status()`

Examples

```
## Not run:
# Explicitly provision the documented environment first.
extract_warm_start_features(c("a", "b"), c("A short text.", "Another text."),
  schema = "writing_features_v2", python = "/path/to/venv/bin/python")

## End(Not run)
```

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,
  warm_start_prior = NULL
)
```

Arguments

- | | |
|---------------|--|
| results | Canonical results_tbl with A_id, B_id, and better_id (plus the standard adaptive results columns). See <code>validate_results_tbl()</code> 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 | <p>List of CmdStan settings. Common fields:</p> <p>chains Number of chains (defaults to <code>min(8, physical_cores)</code> via internal resolution).</p> <p>parallel_chains Maximum concurrent chains. Automatic scheduling uses at most two CPU slots within the available allocation and <code>core_fraction</code>. An explicit value bypasses that fraction and is capped at chains; its product with <code>threads_per_chain</code> must fit the available allocation.</p> <p>threads_per_chain Threads per chain (default 1). The bundled models do not use within-chain parallel likelihoods, so higher values do not accelerate their likelihood evaluation.</p> <p>iter_warmup Warmup iterations (default 1000).</p> <p>iter_sampling Sampling iterations (default 1000).</p> <p>seed Optional integer seed forwarded to CmdStan (default NULL).</p> <p>core_fraction Fraction of available CPU slots for automatic parallelization (default 0.8); at least one slot is retained.</p> <p>output_dir Optional directory for CmdStan output.</p> <p>Sampling requires the suggested parallelly package for allocation-aware detection. Scheduler, container, and check constraints limit concurrency, without changing the number of chains. Explicit requests above two CPU slots are reported and permitted within the allocation, except under an active <code>_R_CHECK_LIMIT_CORES_</code> constraint. Excessive requests fail with guidance. If detection fails, a warning reports the one-slot fallback. When launching multiple fits independently, divide the allocation among them yourself.</p> |

<code>pair_counts</code>	Optional integer vector of subset sizes (e.g., <code>c(200, 500, 1000)</code>). When provided, the model is fit once per subset size and the round log contains one row per fit. If <code>NULL</code> , a single fit is run using all rows in results.
<code>subset_method</code>	Subset strategy when <code>pair_counts</code> is provided: "first" (default) uses the first <code>n</code> rows of results for each refit; "sample" draws a random permutation once and then takes the first <code>n</code> rows of that permutation for each refit.
<code>seed</code>	Optional integer seed for deterministic subset selection when <code>subset_method = "sample"</code> . When <code>NULL</code> , falls back to <code>cmdstan\$seed</code> if provided.
<code>inference_contract</code>	Optional list of inference-routing semantics to attach to each fit contract. When omitted, values are inferred from <code>results\$phase</code> and optional <code>results\$judge_scope</code> .
<code>warm_start_prior</code>	Optional <code>make_warm_start_prior()</code> object. Defaults to no predictive prior (raw theta prior mean 0, SD 1). IDs must match <code>ids</code> .

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). Each records the actual per-item raw theta prior in `theta_prior`; predictive fits also include compact provenance in `predictive_prior`. Each new fit also records `evidence_identity` (normalized fitted comparisons, item domain, hash and count) and `reference_fit_config`. These let `prepare_linked_rubric_reference()` check the original comparisons before saving a reusable rubric reference. Subset fits record only their fitted rows.

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

See Also

`build_btl_results_data()`, `make_warm_start_prior()`, `adaptive_rank()` and the [integrated CJ workflow](#)

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 optional frequentist engines

Description

This function fits a Bradley–Terry paired-comparison model to data prepared by [build_bt_data](#). It supports five modeling engines:

- **sirt**: [btm](#) — the default 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.
- **brglm2**: explicit Firth mean bias reduction for random or nonadaptive schedules, with centered covariance and SSR.
- **alpha**: explicit alpha adjustment motivated by adaptive schedules, using base R with centered covariance and SSR.
- **lapse**: experimental unpenalized model matching with estimated first-position bias and lapse probability, using base R.

Usage

```
fit_bt_model(
  bt_data,
  engine = c("auto", "sirt", "BradleyTerry2", "brglm2", "alpha", "lapse"),
  verbose = TRUE,
  ...,
  sirt_eps = NULL,
  alpha = NULL
)
```

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", "BradleyTerry2", "brglm2", "alpha", or "lapse". Automatic selection never chooses Firth, alpha, or lapse.
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> . For <code>brglm2</code> , only a named control list is accepted, with numerical settings <code>epsilon</code> (default $1e-10$), <code>maxit</code> (200), <code>slowit</code> (1), <code>max_step_factor</code> (12), and <code>trace</code> (FALSE). <code>verbose = FALSE</code> disables tracing; numerical warnings are retained. The mean-bias-reduction method cannot be changed through controls. For <code>alpha</code> , only a named control list is accepted: <code>epsilon</code> (default $1e-12$), <code>maxit</code> (200), <code>gradient_tol</code> ($1e-7$), <code>step_tol</code> ($1e-7$), <code>min_rcond</code> ($1e-12$), and <code>trace</code> (FALSE). See the <code>alpha</code> section below. For <code>lapse</code> , only a named control list is accepted: <code>maxit</code> (2000), <code>reltol</code> ($1e-12$), <code>gradient_tol</code> ($1e-7$), <code>step_tol</code> ($1e-7$), <code>min_rcond</code> ($1e-12$), and <code>trace</code> (FALSE). See the <code>lapse</code> section below.
sirt_eps	Optional finite, nonnegative epsilon adjustment for <code>sirt</code> , supplied by exact name. NULL preserves the engine default or legacy <code>eps</code> in Supplying both forms raises an error. This argument is valid only with <code>engine = "sirt"</code> or <code>"auto"</code> ; on automatic fallback it remains recorded as requested but is not applied to <code>BradleyTerry2</code> .
alpha	Explicit finite nonnegative numeric scalar, supplied by exact name, required only for <code>engine = "alpha"</code> . There is no default penalty and no tuning from outcomes. Values 0.30 and 0.50 are supported alongside other nonnegative values. Zero requests ordinary unpenalized estimation and requires a strongly connected directed win graph. NULL is only accepted for other engines.

Details

When `engine = "auto"` (the default), the function attempts **sirt** first and automatically falls back to **BradleyTerry2** on availability or execution failure. Explicit engine requests never fall back. Invalid inputs, disconnected graphs, and invalid reliability results raise errors even in automatic mode. 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. Higher values mean stronger relative performance on the assessed trait. Zero is not a pass mark, and these estimates are not rubric grades or automatically comparable across independently fitted sets. Standard errors are included for all modeling engines when their uncertainty checks pass. Raw MLE reliability is available from **sirt**; Firth and alpha fits return independently calculated SSR.

For `sirt`, `$ssr` independently calculates $1 - \text{mean}(\text{se}^2) / \text{stats::var}(\text{theta})$ from all returned items, using sample variance. Agreement with `$fit$mle.rel` is required within $1e-12 * \max(1, \text{abs}(\text{engine_reliability}), \text{abs}(\text{ssr}))$. Negative SSR is retained. Nonfinite `theta`/SEs, negative SEs, zero score variance, and nonfinite calculations raise errors without dropping items. This includes `sirt` fits with missing SEs from fixed `theta` or extreme scores when `epsilon` is zero. See [scale_separation_reliability\(\)](#) for the component definitions.

SSR depends on estimated score variance and the SE convention: it is not an estimator-free measure of recovery. `BradleyTerry2` uses engine contrasts (normally a reference item); this wrapper does not calculate SSR from its reference-based SEs. Its legacy `$reliability` remains NA.

The `sirt` default `epsilon` is resolved from the installed engine (0.3 in `sirt` 4.2.133), and the returned `epsilon` is checked and recorded. Other estimator defaults are unchanged, including `sirt`'s tie and positional parameters. `effective_settings` records resolved arguments; for `sirt`, `fix.delta_requested` is separated from `returned_parameters` because `sirt` 4.2.133 accepts but does not apply `fix.delta`. Other engine versions are marked unverified for that argument. No fix for the upstream estimator is applied here.

Connectivity is checked before any engine is called, including ties removed by `ignore.ties = TRUE`, and `BradleyTerry2` subsets/zero weights. Disconnected data cannot identify global BT scores or SSR. Missing outcomes, invalid IDs, and self-comparisons raise errors rather than being dropped. Direct `sirt` inputs may include ties coded 0.5; the `BradleyTerry2` wrapper requires binary outcomes.

`sirt` provides iterations but no explicit convergence flag. Early termination is recorded as `stopping_criterion_met`; reaching `maxiter` is recorded as `iteration_limit_reached` with `converged = NA`, not as proven convergence or nonconvergence. `BradleyTerry2`'s reported convergence is preserved. Reliability validity describes the arithmetic, not proof of convergence.

Install an optional engine before fitting, for example with `install.packages("sirt")`. Pairwise data preparation does not need that engine. See the [offline walkthrough](#) for fitting and interpreting bundled synthetic comparisons.

Firth fits use binomial-logit `brglm2::brglmFit` with `type = "AS_mean"`, equivalent to adding half the log determinant of expected information to the log likelihood. This is a genuine Firth estimator, not `sirt` `epsilon` adjustment. It is an explicit option for random/nonadaptive schedules; it is not recommended here as the primary adaptive-schedule correction. No schedule type is inferred from outcomes. See Hamilton and Tawn, [doi:10.1111/jedm.70022](#), and the `brglm2` mean-bias-reduction documentation.

The Firth design has no intercept, tie, positional, or lapse parameter. Binary comparisons are aggregated in deterministic item/pair order. Internal contrasts use the last radix-sorted item as reference,

then both estimates and covariance are transformed to sum-to-zero coordinates. `$vcov` is the model-based inverse expected information at the bias-reduced estimate, transformed to item coordinates; it is not a penalized-Hessian or bootstrap covariance. Its rank is the number of items minus one because of centering. SEs are square roots of its diagonal. Separation and undefeated or winless items are supported when the comparison graph is connected.

Firth fits must converge with finite estimates and valid covariance. Failures error without fallback. A valid fit with zero score variance is retained: `$reliability` is NA and `$ssr$status` is "zero_score_variance". Other invalid theta/SE or reliability arithmetic raises an error. Use `predict.pairwiseLLM_bt_firth()` for first-item win probabilities.

Value

A list with the following elements:

engine The engine actually used ("sirt", "BradleyTerry2", "brglm2", "alpha", or "lapse").

fit The fitted model object.

theta A tibble with columns:

- ID: object identifier
- theta: estimated ability parameter
- se: standard error of theta; NA for valid lapse-boundary fits

reliability Raw MLE reliability for sirt or calculated SSR for Firth/alpha. NA for **BradleyTerry2** and lapse models or a zero-variance Firth/alpha fit.

ssr For sirt, the `scale_separation_reliability()` decomposition plus `engine_reliability`, `agrees`, `absolute_difference`, and `tolerance`. For **BradleyTerry2**, `ssr` and `engine_reliability` are NA, `valid` is FALSE, `agrees` is NA, and `status` is "unavailable_se_convention". For Firth/alpha, the helper decomposition, or `valid = FALSE` and `status = "zero_score_variance"` when undefined.

provenance A list recording engine, requested_engine, loaded engine_version and package_version, supplied_arguments, requested_sirt_eps, effective_settings, adjustment, identification, convergence (status, converged, iterations), theta_finite, se_finite, reliability_valid, reliability_status, and fallback_reason (NULL unless automatic fallback occurred). Identification records sirt centering or BradleyTerry2's contrasts, reference category and player levels. Save the full object to retain these settings; the legacy summary tibble is unchanged. Firth provenance also records the coordinate transformation and covariance convention. Alpha adds `engine_package = "stats"`, the explicit penalty, solver, parameter ordering, convergence code/message and uncertainty scope.

vcov Firth/alpha/lapse: centered item covariance matrix, with row/column labels in the same order as `theta$ID`; NULL for valid lapse-boundary fits.

comparisons Firth/alpha/lapse: original item pairs for default prediction.

beta, epsilon, model_variant Lapse only: first-position bias, guessing probability, and exact variant "bt1_e_b".

parameter_vcov Lapse only: joint centered theta/beta/epsilon covariance, ordered by `theta:ID` labels followed by `beta` and `epsilon`; NULL at the zero boundary.

log_likelihood, objective Lapse only: unpenalized log likelihood and its negative. No penalty or parameter-transformation Jacobian is added.

alpha Alpha engine only: the requested penalty strength.

diagnostics Alpha: objective components, item scores, reduced-coordinate gradient, penalized Hessian and unpenalized information, matrix checks, Newton correction, optimizer status, and numerical warnings. Lapse: ordered-pair probabilities, natural-coordinate score and observed Hessian, information/matrix checks, Newton correction, and all optimizer attempts including the epsilon-zero boundary and epsilon-one objective.

Experimental frequentist lapse model matching

engine = "lapse" matches the Bayesian `bt1_e_b` likelihood:

$$p(A \text{ wins}) = (1 - \epsilon) \text{logit}^{-1}(\theta_A - \theta_B + \beta) + \epsilon/2.$$

Positive beta favors the first presented item; epsilon in $[0, 1]$ is the probability of guessing, distinct from sirt's epsilon adjustment. The fit estimates all three parameter groups jointly, with sum-to-zero theta, no penalty and no Bayesian priors. It is a model-matching prototype, not a replacement for Bayesian estimation or a production recommendation.

Binary rows are aggregated by ordered pair. Joint optimization uses the existing centered reference contrasts and natural epsilon constrained to $[0, 1]$, with L-BFGS-B. Starts have zero item contrasts and beta, and epsilon 0.001, 0.05, 0.2, 0.5 and 0.9. The objective and gradient are divided by the comparison count for optimization; reported likelihoods, Hessians and stationarity checks use the unscaled sum. L-BFGS-B uses `factr = reltol / .Machine$double.eps`, and its projected-score tolerance is `gradient_tol` divided by the comparison count. Up to ten damped Newton steps polish successful attempts in natural coordinates, within the bounds. The highest-likelihood candidate is checked independently; a materially worse solution is never selected to obtain valid uncertainty. Native convergence, natural-coordinate scores and centered Newton corrections must satisfy the numerical controls. All attempts, including failed ones, remain in diagnostics.

The epsilon-zero face is optimized separately with BFGS, and the epsilon-one constant-probability likelihood is evaluated exactly. The zero-face solution must pass its own theta/beta convergence, stationarity and curvature checks even when an interior fit is selected. It can be selected when its objective is lower or within $100 * .Machine$double.eps * \max(1, \text{abs}(\text{objective}))$ of the joint candidate and its one-sided negative-log-likelihood epsilon score is at least `-gradient_tol`. A small positive fitted epsilon alone does not establish a boundary optimum. There are no bounds on item strengths or beta, penalties, or clipping of probabilities, epsilon, or Hessian eigenvalues. Out-of-domain optimizer evaluations are retained as failed attempts.

Connectedness alone does not establish identification of lapse and bias. The ordered design must have sufficient rank, and full natural-parameter expected information must be positive definite with reciprocal condition number at least `min_rcond`. Interior fits also require a positive definite, well-conditioned joint observed negative-log-likelihood Hessian. Its inverse is transformed to centered theta, beta and natural epsilon; theta SEs include estimation of both nuisance parameters. The full covariance is positive semidefinite and singular only because theta sums to zero. SEs are local, model-based approximations conditional on the realized graph; these checks do not establish repeated-sample calibration.

An identified epsilon-zero optimum is a valid point-estimate fit. It returns theta, beta, exactly zero epsilon, likelihoods and predictions, with `provenance$convergence$status = "converged_boundary"` and `converged = TRUE`. Required observed curvature is positive definite along the theta/beta face; when the epsilon score is within `gradient_tol` of zero, the full joint curvature must also pass. Ordinary joint lapse-model uncertainty is not reported: `theta$se` is NA, `vcov` and `parameter_vcov` are

NULL, and `provenance$uncertainty` has `method = "none"`, `valid = FALSE`, and `status = "nonregular_boundary"`. Interior fits have uncertainty status "valid". Epsilon one leaves theta and beta unidentified and is rejected.

Genuine failed checks raise `pairwiseLLM_bt_lapse_error`, also a BT validation error, retaining available theta (without SEs), beta, epsilon, diagnostics, provenance and `failure_reason`. No regularization or simple-BT fallback is introduced after failure. Conventional SSR is unavailable for this prototype (`reliability = NA`, `ssr$valid = FALSE`), including valid boundary and interior fits. `bootstrap_bt_model()` rejects these fit objects. Boundary intervals and bootstrap uncertainty are outside this prototype.

The original frozen issue-305 audit recovered 324 of 450 cases and rejected 100 boundary candidates and 26 numerical failures. The boundary follow-up repeats the identical cases and recovery tolerances: 352 interior-valid fits and 98 boundary-valid fits recovered, with zero numerical/identification failures and zero recovery failures among valid fits. Read its results with `readLines(system.file("valid", "bt-lapse-305-boundary.txt", package = "pairwiseLLM"))`. The original `bt-lapse-305.txt` and accompanying evidence are preserved. This synthetic qualification does not establish production validity, boundary interval coverage or repeated-sample uncertainty calibration. See `predict.pairwiseLLM_bt_lapse()` for ordered plug-in probabilities.

Alpha-adjusted estimation

Hamilton and Tawn ([doi:10.1111/jedm.70022](https://doi.org/10.1111/jedm.70022), equation 3) define an adjustment to the score equation for item r :

$$a_r = \alpha \left(1 - \frac{2}{N-1} \sum_{j \neq r} p_{rj} \right).$$

With $p_{ij} = \text{logit}^{-1}(\theta_i - \theta_j)$, observed win counts w_{ij} , and $c = \alpha/(N-1)$, the implemented objective is

$$\ell_\alpha(\theta) = \sum_{i < j} \{w_{ij} \log p_{ij} + w_{ji} \log(1 - p_{ij})\} + c \sum_{i < j} \log\{p_{ij}(1 - p_{ij})\}.$$

The penalty covers every unordered pair, including unobserved pairs, and adds c pseudo-wins in each direction. It differs from sirt's conventional epsilon adjustment, which uses observed win proportions. Neither method changes which pairs are selected. Alpha adjustment is motivated by adaptive scheduling; it is not universally preferred or a guarantee of unbiased SSR. Firth remains the intended modern comparator for random schedules. `bootstrap_bt_model()` separately provides schedule-aware bootstrap bias correction; it does not turn these conditional SEs into corrected-score SEs.

The alpha engine shares Firth's binary input and sum-to-zero convention. Items are radix-sorted; coefficient i is the contrast of item i to the last item, for $i = 1, \dots, N-1$. If B is the centered reference map, $\theta = B \beta$. For pair design row x and total observed comparisons m , the negative objective Hessian is $H_\alpha = \sum_{i < j} (m_{ij} + 2c)p_{ij}(1 - p_{ij})xx^T$. The original-data information is $I = \sum_{i < j} m_{ij}p_{ij}(1 - p_{ij})xx^T$. The returned covariance is $BI^{-1}B^T$, evaluated at the alpha estimate, with SEs from its diagonal. These are model-based SEs conditional on the realized comparison graph, not schedule-aware uncertainty. The penalized Hessian, inverse penalized curvature, and sandwich covariance are not used for reported SEs or SSR. Centering makes the item covariance rank $N-1$.

A single `stats::glm.fit` IWLS fit uses weighted binary rows for the augmented counts, zero starts, and a quasibinomial-logit working family with dispersion fixed at one. This supplies the exact binomial-logit estimating equations without warnings about fractional pseudo-counts; no dispersion estimate or GLM covariance is used. The objective and derivatives are evaluated independently. Native convergence and full rank are necessary but not sufficient: the maximum absolute adjusted item score must be at most `gradient_tol`, and the maximum absolute item-coordinate Newton correction at most `step_tol`. Both reduced matrices must be positive definite with reciprocal condition number at least `min_rcond`. All numeric controls must be positive and finite; `maxit` must be an integer, `min_rcond` less than one, and `trace` logical.

Fits never change alpha or solver after failure. Numerical validation errors have class `pairwiseLLM_bt_alpha_error` (also a BT validation error). Their `theta`, `provenance`, `diagnostics`, and `failure_reason` fields preserve available results for auditing, including converged theta if uncertainty fails. No theta-only public fit is returned. A valid equal-strength fit is retained with NA SSR and `zero_score_variance` status, as for Firth. When every item's total wins equal its total losses, zero is the exact stationary solution. `$diagnostics$exact_zero_solution` records its use; `$diagnostics$coefficients` are the effective contrasts, while `$fit` retains the raw IWLS output. This is an exact count-based identity, not rounding small estimates to zero. Native convergence and uncertainty checks still apply. Extremely small/large penalties can exceed numerical resolution and error. The dense all-pair design has no large-scale sparse-optimization guarantee. Use `predict.pairwiseLLM_bt_alpha()` for plug-in pair probabilities.

See Also

`build_bt_data()`, `summarize_bt_fit()`, `scale_separation_reliability()`, `bootstrap_bt_model()`, [integrated CJ workflow](#)

Other frequentist models: `bootstrap_bt_model()`, `build_bt_data()`, `build_elo_data()`, `fit_elo_model()`, `predict.pairwiseLLM_bt_alpha()`, `predict.pairwiseLLM_bt_firth()`, `predict.pairwiseLLM_bt_lapse()`, `scale_separation_reliability()`, `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", sirt_eps = 0.3, verbose = FALSE)
  fit1$ssr
  fit1$provenance$adjustment
}
if (requireNamespace("BradleyTerry2", quietly = TRUE)) {
  fit2 <- fit_bt_model(bt, engine = "BradleyTerry2", verbose = FALSE)
}
```

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 <code>build_elo_data</code> .
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: `bootstrap_bt_model()`, `build_bt_data()`, `build_elo_data()`, `fit_bt_model()`, `predict.pairwiseLLM_bt_alpha()`, `predict.pairwiseLLM_bt_firth()`, `predict.pairwiseLLM_bt_lapse()`, `scale_separation_reliability()`, `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
}
```

fit_link

*Fit a linker using prepared explicit evidence***Description**

Fit a linker using prepared explicit evidence

Usage

```
fit_link(input, previous = NULL)
```

Arguments

input	A validated object from <code>prepare_link_input()</code> .
previous	Optional common linking result for numerical warm starting. Supply the complete cumulative cross-set evidence in input. The old prefix, identities, Phase A inputs, judge surface, and offset prior must be unchanged. Only the previous numerical mode is passed to an engine, never its posterior as a new prior. Invalid previous fits supply no warm start.

Details

E1 (fixed_shape_offset) uses adaptive one-dimensional quadrature. E2 (gaussian_posterior_bridge) and E3 (joint_offset) use MAP/Laplace. E3 additionally supports explicit MCMC; there is no default estimator or fallback. Result schema version 1 uses $\theta_H = H_H u_H$ and $\theta_S = \delta + H_S u_S$, 95 percent interval endpoints, descending ranks with average ties, and free coordinates delta, hub shape, then spoke shape. E1 has only the delta free coordinate. Prediction and continuation state contain serializable data, with functions resolved by estimator ID/version.

E1 holds separately centered Phase A EAP shapes fixed and updates only delta under the configured Normal prior and the supplied cross-set likelihood. Reported means, SDs, and equal-tailed 95 percent intervals come from adaptive Gauss-Kronrod quadrature, with CDF integration and root finding for interval endpoints. Both infinite tails are integrated using rational coordinate maps; there is no finite-domain truncation or Laplace approximation. Monotone likelihood bounds guard against missed distant mass. With zero edges, or epsilon equal to one, summaries use the exact Normal prior. Positive-budget epsilon-one fits report unidentified.

E1 uncertainty is conditional on fixed shapes: hub variance is truly zero, every spoke SD equals delta SD, and all spoke uncertainty is perfectly correlated. It does not propagate Phase A estimation uncertainty. E1 is useful with strong Phase A estimates or when computation must be inexpensive. Numerical initial values are ignored; continuation recomputes the posterior from the original prior and cumulative evidence, deterministically.

E2 is a staged, joint-uncertainty-propagating approximation: each separately centered Phase A posterior is Gaussianized using its full reduced-coordinate mean and sample covariance. Only cross-set observations enter its likelihood; historical Phase A raw judgments need not be replayed. The offset has its own Normal prior, independent of both centered shapes. Bridge covariances use the smallest successful Cholesky jitter from $c(0, 1e-12, 1e-10, 1e-8, 1e-6) * s$, where s is the maximum positive diagonal variance (one only if no positive scale exists). Failure beyond this ladder is

explicit; neither diagonal approximation nor dimension removal is allowed. Recorded jitter is a numerical adjustment to the bridge.

E2 fits the exact Gaussian penalties and lapse-mixture likelihood in bridge-whitened coordinates. Five BFGS starts use the bridge means and delta at the prior mean and plus/minus one and two prior SDs. Up to ten damped Newton steps polish a converged BFGS fit to the gradient tolerance. The lowest-objective converged stationary solution is selected; this finite deterministic search does not guarantee a global mode for a multimodal lapse-mixture posterior. Its observed Hessian must be positive definite without additional regularization. The inverse Hessian is transformed to free coordinates and then item scale; reported means are the Laplace center and intervals are normal 95 percent approximations. At zero edges, E2 returns the two independent stabilized bridges and the offset prior exactly. With epsilon one the same distribution applies, labeled unidentified at positive budget. Continuation refits cumulative evidence from the original bridges, never from an earlier Phase B posterior. E3 uses each raw hub Phase A, spoke Phase A, and cross-set observation exactly once, with fixed beta/epsilon. It jointly re-estimates the centered shapes with independent standard-Normal reduced-coordinate priors and an independent Normal offset prior. This is a complete joint model with greater data-retention and computational requirements than staged E2. E3 MAP uses the same five deterministic starts and convergence/PD-Hessian requirements as E2, in prior-whitened coordinates. Reported means are the MAP/Laplace center and intervals are normal 95 percent approximations. No Hessian repair or fallback is applied. The same finite-multistart/global-mode limitation applies. Initial values do not alter the deterministic search.

E3-MCMC (`control$estimator$engine = "mcmc"`) is an audit/reference engine, never invoked by ordinary MAP/Laplace fitting. It samples the mathematically identical model, returns empirical means/covariance and equal-tailed 95 percent intervals, and retains chain-indexed free-coordinate draws in `prediction$state`. Item draws are `free_draws %>% t(item_transform)`. No live CmdStan object or CSV file is required for saved-result prediction. Compiled models are cached in the writable package user cache, keyed by source, CmdStan/CmdStanR versions, and compilation options.

At zero cross edges, E3-MCMC reports exact Normal offset-prior moments and intervals, with theoretical zero between-block covariances and sampled within-shape moments. Spoke intervals convolve empirical shape draws with the independent Normal offset prior. Raw draws remain unmodified for audit and posterior-average prediction, so their sample moments can differ from these factorized summaries. With epsilon one, all reported moments/intervals use the exact independent Gaussian priors. Zero-edge identification is `prior_only`; positive-budget epsilon-one identification is `unidentified`.

Sampler diagnostics include per-coordinate rank-normalized R-hat, bulk/tail ESS, MCSE(mean)/SD, per-chain divergences, E-BFMI and treedepth hits/fractions, effective controls, and the frozen D005 audit gate. Valid MCMC fits require zero divergences, $R\text{-hat} \leq 1.01$, bulk ESS ≥ 1000 , tail ESS ≥ 500 , E-BFMI $\geq .30$ in every chain, overall treedepth-hit fraction $\leq .01$, and MCSE/SD $\leq .05$ for every free coordinate. Missing diagnostics fail the gate. Failed gates retain finite summaries, covariance, draws, and diagnostics, but set `fit_valid = FALSE` and `failure_code = "mcmc_audit_gate_failed"`; prediction requires a valid fit. Sampling errors instead return typed missing summaries. Repairs use a new explicitly configured call; the package does not automatically escalate sampling or change the model. MCMC does not supply a continuation mode, and no engine reuses a posterior as a prior.

Value

A `pairwiseLLM_link_result` containing common-scale item summaries, offset summaries, joint uncertainty, prediction data, diagnostics, provenance, and continuation inputs. Unavailable uncertainty is typed missing, never implicitly zero. Numerical failures remain invalid method-specific results.

Result fields

`items` contains `set_id`, `item_id`, `global_item_id`, `theta_link_mean`, `theta_link_sd`, `theta_link_lower`, `theta_link_upper`, and `rank_link`. `theta_link_eap` is a compatibility alias of `theta_link_mean`; it is a posterior mean for E1 and E3-MCMC, and a MAP location for E2/E3 Laplace. `estimator_id` and `uncertainty_scope` accompany every item. `offset` contains `delta_mean`, `delta_sd`, `delta_lower`, `delta_upper`, and `identification` (`prior_only`, `cross_set`, `unidentified`, or `failed`). At zero cross edges a valid result retains the configured Normal offset prior and reports `prior_only`. E1 retains fixed shapes; E2 retains independent full Gaussian Phase A bridges; E3 fits only the centered Phase A shape posterior.

`uncertainty` contains the named free-coordinate covariance (or NULL), basis, and `item_transform`. For transform T and covariance V , item-scale covariance is $T \%* \% V \%* \% t(T)$. E1's item transform has only a delta column; its fixed shapes supply the deterministic part of item means.

`diagnostics` records `fit_attempted`, `fit_valid`, `convergence_code`, `finite_objective`, `finite_gradient`, `hessian_pd`, `covariance_valid`, `covariance_jitter`, `warning_code`, `failure_code`, `uncertainty_scope`, `sampler`, `elapsed_seconds`, `cpu_seconds`, `peak_memory_bytes`, and `n_parameters`. Undefined fields use typed missing values. Invalid fits carry a failure code and cannot be used for prediction. E1 adds optional quadrature diagnostics: `method`, `domain`, `coordinate`, `effective_controls`, `integration_partitions`, `subdivision_count`, `log_normalizer`, `estimated_mass/moment_errors`, `CDF_error` and `root_brackets` when computed, `summary_method`, and `status`. Partial diagnostics remain available after numerical failure. Quadrature errors are numerical estimates, not posterior SDs. Valid E1 scope is `offset_only_conditional_on_fixed_shapes`; valid E2/E3 scope is `joint_shapes_and_offset`. A true zero conditional variance is allowed. E2 adds bridge diagnostics for each set (`draw_count`, `reduced_mean`, `stabilized_covariance`, `variance_scale`, and `applied_jitter`) and optimization diagnostics (`effective_controls`, `starts`, `convergence`, `objectives`, `gradients`, `observed Hessians`, `selected_attempt`, and `reciprocal_condition_estimates`). Gradients and Hessians are in bridge-whitened coordinates; result covariance is in the named common free coordinates. `covariance_jitter` is the largest absolute bridge jitter. Hessians are never jittered. Failure messages and completed bridge/optimization diagnostics remain available on invalid fits.

`provenance` retains package/source versions, all input hashes and counts, Phase A sources, and the judge source. `prediction` binds serializable estimator data to estimator ID/version and input hash. `continuation` retains the original normalized input and optional free-coordinate numerical mode. Results can be round-tripped with `saveRDS()` and `readRDS()`.

See Also

`prepare_link_input()`, `predict_link()`

`prepare_link_input()`, `fit_link()`, `predict_link()`, `start_link_session()`

Other linking: `predict_link()`, `prepare_link_input()`, `save_link_session()`, `start_link_session()`, `summary.pairwiseLLM_link_result()`

fit_rubric_calibration

Fit a rubric calibration to completed comparative judgments

Description

Establish a trait-specific rubric conversion from completed Bayesian BTL results. Percentile scoring assigns deterministic, distribution-matched performance levels. Linear ordinal calibration fits human rubric labels with a proportional-odds cumulative-logit model. Monotone ordinal calibration replaces its linear effect with a nondecreasing penalized cubic spline.

Usage

```
fit_rubric_calibration(
  cj,
  rubric = NULL,
  method = "ordinal_linear",
  calibration_design = "same_set",
  trait = NULL,
  levels = NULL,
  K = NULL,
  target_distribution = NULL,
  ...
)
```

Arguments

<code>cj</code>	A completed <code>fit_bayes_btl_mcmc()</code> result, a completed within-set <code>adaptive_rank()</code> result (or its <code>adaptive_state</code>), or an import-ready Phase A artifact list, or a frozen <code>prepare_linked_rubric_reference()</code> result. Fixed results with several refits use the last refit. Raw score tables and intermediate adaptive states are not accepted.
<code>rubric</code>	Data frame with <code>item_id</code> and <code>rubric_score</code> , optionally <code>trait</code> . Rows align by ID, never position. Missing scores denote unlabeled items. Use global item IDs for adaptive results and Phase A artifacts that provide them; otherwise use the original fixed-fit IDs. Omit for percentile scoring.
<code>method</code>	One of "ordinal_linear" (default), "ordinal_monotone", or "percentile". Percentile scoring is norm-referenced/distribution-matched.
<code>calibration_design</code>	"same_set" for one completed CJ scale, or "linked_anchors" for a reusable Phase A rubric reference artifact. Both ordinal methods support both designs. Percentile scoring supports only "same_set".
<code>trait</code>	Single trait identifier. Required when absent from CJ metadata. Each analytic trait requires its own CJ analysis and calibration.

levels	Ordered original rubric labels, from lowest to highest quality. When omitted, use ordered-factor levels or ascending numeric rubric labels. Character and unordered-factor scores require explicit levels.
K	Optional number of levels, an integer ≥ 3 . Must agree with levels and any target proportions. Typically 3–6.
target_distribution	Optional positive proportions in level order, summing to one, for percentile scoring. Omission requests equal proportions. The absolute sum tolerance is $1e-8$; accepted proportions are normalized to sum to one for computation, while the requested values are retained. Named proportions must match the labels in their specified order.
...	For ordinal_monotone only, uniquely named k and sp controls. k is the cubic basis dimension (default 6, integer ≥ 5). It is reduced with a warning to the number of unique labeled CJ scores if necessary; fewer than five unique scores fail. sp = NULL (default) estimates smoothing by extended Fellner-Schall (EFS); a finite positive scalar fixes the penalty parameter. Other methods require empty dots. Unknown controls fail.

Details

Higher CJ locations and higher ordered rubric levels must mean better performance; no orientation is silently reversed. All requested categories must be observed for ordinal calibration.

Fitting `ordinal_linear` requires the optional package **ordinal**. Install it with `install.packages("ordinal")`. Percentile scoring and prediction from an already fitted linear calibration do not require it.

Monotone ordinal fitting and prediction require optional **mgcv** $\geq 1.9-4$ (`install.packages("mgcv")`). Fitting also requires optional **withr** (`install.packages("withr")`) to isolate backend RNG use with a fixed internal seed. Saved monotone models require **mgcv** for spline prediction.

For both ordinal methods, calibration items must belong to one completed trait-specific CJ fit. Only labeled rows estimate the calibration: $z = (\text{theta} - \mu_{\text{cal}}) / \sigma_{\text{cal}}$, where μ_{cal} is their mean and σ_{cal} their sample standard deviation. Both are stored and reused. `ordinal::clm()` fits $\text{logit } P(Y \leq k \mid z) = \tau_k - \beta * z$ with flexible ordered thresholds. Zero or negative slopes produce a diagnostic warning; coding is never reversed. Missing categories or degenerate calibration scores fail. Singleton categories and numerical convergence/identification problems produce warnings; no universal minimum calibration sample size is enforced. Finite ordered estimates can be retained with warnings even when convergence or uncertainty is unreliable. Such warnings require review before use. Standard errors condition on estimated CJ point locations; CJ measurement uncertainty is not propagated. `evaluate_rubric_predictions()` supplies metrics and optional assumption diagnostics; diagnostic refits run only when requested during evaluation. Internal rubric-label validation holds CJ fixed and refits calibration from training labels alone.

`ordinal_monotone` uses `mgcv::scasm()` with $s(z, \text{bs} = "sc", \text{xt} = "m+", k = k)$ and `mgcv::ocat()` with integer categories $1:K$. The model is $\text{logit } P(Y \leq k \mid z) = \tau_k - \eta(z)$, where η includes an intercept and a nondecreasing smooth. The backend fixes the first threshold at -1 for identification; its identity link describes the latent location, while category cumulative probabilities follow the logistic distribution. Bootstrap is disabled. Smoothing is estimated by EFS unless sp is fixed. A 1,001-point calibration grid verifies nondecreasing latent locations and nonincreasing cumulative probabilities with tolerance $1e-8$. Invalid probabilities or monotonicity failures abort; no unconstrained fallback is used. Essentially flat effects, singleton categories, and convergence

problems warn. Small bases and some reversed/separated label patterns can fail in the backend; these return contextual fit errors. The five-unique-score requirement is a backend feasibility guard, not a recommended calibration sample size. Threshold uncertainty is unavailable from this wrapper (the fixed first threshold has standard error zero; the remaining entries are NA). Numerical median cutpoints are reported only for unique crossings within the calibration range. Flat or unresolved crossings and thresholds outside the range have NA cutpoints and an explanatory status; scoring always uses probabilities. Predictions beyond the labeled range are flagged and retain the backend's spline extrapolation; their calibration is not established.

Percentile scoring uses `stats::quantile()` with type 8 at cumulative target proportions. This quantile type is fixed. A score equal to a cutpoint enters the higher category: category is one plus the number of cutpoints less than or equal to the score. Repeated cutpoints can leave categories empty; all exact-score ties receive the same category, even when every score is equal (in which case all items enter the highest category). Requested proportions may be unattainable because of ties or finite sample size. Cutpoint tie counts count all source observations exactly equal to each cutpoint, including a count of one when only one observation equals that cutpoint.

These are norm-referenced CJ-derived performance levels, not criterion-referenced scores. Matching a historical rubric distribution does not establish agreement with rubric raters. Recomputing cutpoints on another cohort changes the reference distribution. Stored percentile cutpoints support reuse of the original completed result only; independent cohorts and linked target prediction are not supported. Scoring conditions on accepted point locations, even when CJ posterior draws are available. Percentile category probabilities and uncertainty propagation are unavailable.

Phase A artifacts are checked using existing import-readiness rules, including the existing explicit quality-gate override. Import readiness does not assert that the originating adaptive run terminated. Existing CJ diagnostics are retained; failed diagnostics produce a warning.

With `linked_anchors`, first obtain an import-ready Phase A artifact for the human-scored `rubric_reference_set`, or prepare a completed standalone Bayesian ranking with `prepare_linked_rubric_reference()`, and fit its ordinal calibration. Standalone references require evidence identity recorded during fitting; older fits must be refitted from their original comparisons before reuse as a reference. Their existing same-set uses remain supported. Next run Phase A for the target set, then explicitly select an E1–E3 estimator in `prepare_link_input()`, supplying the reference artifact or the prepared standalone reference's matching hub payload and source metadata. Pass the `fit_link()` result or `start_link_session()` session to `predict()`. E1 needs points, E2 needs posterior draws, and E3 needs raw within-set rows. Rubric labels are used for calibration; Phase B does not require them. A rubric reference set contains externally scored material; a Phase B **hub anchor** is an item selected for routing recurring comparisons.

Historical prediction validates the stored reference set, stable item IDs, exact reference locations and uncertainty, within-set evidence, trait, orientation, and canonical fit contract. A configuration hash alone does not establish metric identity. Original hashes are retained as provenance; compatible legacy hashes follow the existing Phase A import rules. Target-only Phase A scores cannot be used with the stored calibration. Prediction consumes accepted Phase B common-scale scores and reuses the stored reference transformation, never the target cohort's mean or SD. `theta_link_eap` aliases the estimator's point estimate: posterior mean for E1/E3-MCMC, MAP for E2/E3 Laplace. Prediction propagates the reported uncertainty scope (conditional on fixed shapes for E1; shapes and offset for E2/E3). Scores are translated from the centered linking origin to the original reference origin by adding the frozen hub mean, without rescaling or using target labels. E2/E3 hub posteriors may update; calibration stays attached to the original Phase A hub artifact. Changed artifact identity requires a new link. Invalid or unidentified fits and legacy anchored-joint sessions fail. Rubric

scoring is downstream of CJ estimation and does not run comparisons or change Phase B estimation. Ordinal calibration conditions on accepted CJ locations; joint CJ/rubric likelihood estimation is outside this API.

Value

For percentile scoring, a fitted `pairwiseLLM_rubric_calibration` containing labels, K, normalized `cj`, source `calibration_range`, `trait`, `provenance`, `diagnostics`, and an identity transformation (center 0, scale 1). Its backend stores `quantile_type`, `cutpoints`, `cumulative_probs`, `requested_proportions`, `normalized_effective_proportions`, `achieved_proportions`, `achieved_counts`, and `cutpoint_tie_counts`. Category summaries include every requested level, including empty levels. `target_distribution` retains the requested proportions (or equal defaults). `calibration_data` and `category_counts` remain NULL because no human labels are fitted. `diagnostics$category_probabilities_available` is FALSE. For linear ordinal calibration, the same class retains ID-aligned `calibration_data` (including unlabeled items), `category_counts`, and the labeled `calibration_range`. `transformation$center` and `$scale` store `mu_cal` and `sigma_cal`. The backend contains `name`, `version`, `link`, `threshold`, the fitted model, `thresholds`, `slope`, `vcov`, `standard_errors`, and `convergence`. With positive slope, `cutpoints_z` stores τ / β and `cutpoints_theta` maps these to the original scale. Otherwise cutpoints are omitted. Unavailable coefficient uncertainty is represented by NA, with a warning. `diagnostics$ordinal` reports numerical convergence, gradient, Hessian condition, covariance availability, nonpositive slope, singleton categories, and conditioning on CJ locations. `warnings` retains ordinal diagnostic messages and `diagnostics$category_probabilities_available` is TRUE. Monotone ordinal calibration uses the same class, data, transformation, and probability availability flag. Its backend stores `name`, `version`, `cumulative_link`, `latent_link`, fitted model, `thresholds`, `intercept`, `basis` (requested/effective dimension, constraint, order, knots), `smoothing` (method, requested/fitted penalty parameters, penalty matrices), `total_edf`, `smooth_edf`, `convergence`, `threshold_standard_errors`, `cutpoints_z`, `cutpoints_theta`, and `cutpoint_status` ("unique", "outside_range", or "flat_or_unresolved"). `diagnostics$ordinal` stores convergence, singleton categories, conditioning on CJ, unavailable threshold uncertainty, and numerical monotonicity results. Linked calibrations additionally retain reference: the reference `set_id`, sorted stable item IDs with original Phase A locations/SDs, canonical `fit_contract`, original `fit_contract_hash`, and within-set evidence/hash. Original reference SDs are retained separately from estimator-specific linked SDs.

See Also

`predict.pairwiseLLM_rubric_calibration()`, `evaluate_rubric_predictions()`, `vignette("rubric-calibration")` (package = "pairwiseLLM") for practical workflows.

Other rubric calibration: `evaluate_rubric_predictions()`, `predict.pairwiseLLM_rubric_calibration()`, `prepare_linked_rubric_reference()`

Examples

```
## Not run:
# Starting from an already completed CJ result; no sampling occurs here.
fit <- fit_rubric_calibration(completed_cj, method = "percentile",
  trait = "organization", levels = c("developing", "proficient", "advanced"),
  target_distribution = c(0.2, 0.5, 0.3))
predict(fit)
```

```

fit$backend$sachieved_proportions
fit$backend$cutpoint_tie_counts
# Human labels cover a subset of this same completed CJ result.
if (requireNamespace("ordinal", quietly = TRUE)) {
  ordinal_fit <- fit_rubric_calibration(completed_cj, rubric = rubric_labels,
    trait = "organization", levels = c("developing", "proficient", "advanced"))
  predictions <- predict(ordinal_fit)
  predictions$probabilities
  predictions$expected_level
}
if (requireNamespace("mgcv", quietly = TRUE) &&
  packageVersion("mgcv") >= "1.9.4" && requireNamespace("withr", quietly = TRUE)) {
  monotone_fit <- fit_rubric_calibration(completed_cj, rubric = rubric_labels,
    method = "ordinal_monotone", trait = "organization",
    levels = c("developing", "proficient", "advanced"), k = 6)
  predict(monotone_fit)$probabilities
}
# Reference and target Phase A, followed by Phase B, are completed upstream.
if (requireNamespace("ordinal", quietly = TRUE)) {
  linked_fit <- fit_rubric_calibration(reference_phase_a, reference_labels,
    calibration_design = "linked_anchors", trait = "organization",
    levels = c("developing", "proficient", "advanced"))
  predict(linked_fit) # Original reference items.
  target_scores <- predict(linked_fit, completed_phase_b)
  target_scores$set_id
  attr(target_scores, "linking")$diagnostics
}

## End(Not run)

```

fit_warm_start_model *Train a task-specific warm-start model with nested validation*

Description

A warm-start model predicts relative writing quality before pairwise comparisons. One fitting call represents one assessment; do not pool unlinked BT/BTL scales.

Usage

```

fit_warm_start_model(
  ids,
  theta,
  task_id,
  texts = NULL,
  features = NULL,
  schema = "writing_features_v1",
  python = NULL,
  seed = 1L,

```

```

outer_folds = 5L,
inner_folds = 5L,
alpha_grid = seq(0, 1, by = 0.025),
lambda_rule = c("lambda.1se", "lambda.min"),
engine = c("glmnet", "pls", "svr_rbf"),
cv_plan = NULL,
engine_control = NULL
)

```

Arguments

ids	Unique item IDs, aligned positionally with theta and texts.
theta	Finite numeric BT/BTL scores from one assessment, one per ID.
task_id	One nonblank assessment label, stored as provenance. It does not group rows, affect fitting, or identify a registered model.
texts	Optional character vector of texts. Supply exactly one of texts and features. Extraction runs once before resampling.
features	Optional precomputed feature table accepted by predict.pairwiseLLM_warm_model() , aligned to ids by its item_id column.
schema	Frozen feature schema identifier.
python	Existing Python interpreter path for text extraction only.
seed	Integer from zero through .Machine\$integer.max. Local randomization preserves the caller's RNG state. Defaults to 1.
outer_folds, inner_folds	Integer fold counts, at least two; defaults are five each. Every fitting split needs three rows and a nonconstant outcome. Requested fold counts are never reduced automatically.
alpha_grid	Unique finite alpha candidates in $[0, 1]$, sorted internally. Defaults to 41 values from ridge to lasso in increments of 0.025.
lambda_rule	Deployment rule: "lambda.1se" (default) or expert override "lambda.min". The rule applies to outer models and the final model alike.
engine	Fitting engine: "glmnet" (default), "pls", or "svr_rbf". Explicit alpha_grid or lambda_rule arguments are accepted only for glmnet.
cv_plan	Optional make_warm_start_cv_plan() object with exactly matching task, ordered IDs and outcomes. Omitted seed/fold arguments defer to the plan; explicitly conflicting arguments fail. A supplied plan is never regenerated.
engine_control	For PLS, an optional named list containing ncomp, a vector of unique positive integer candidate component counts, at most 10. Every candidate must be legal in every required inner fit and context refit. The default uses all counts from one through the common legal maximum in each tuning context. For SVR, only cost and gamma_multiplier are accepted: unique finite positive candidate vectors, defaulting independently to $2^{(-2:4)}$ and $2^{(-2:2)}$. Epsilon is fixed at 0.10. Unknown controls fail. For glmnet use alpha_grid and lambda_rule instead.

Details

New public fits use engine-neutral format 3 with explicit full audit status, a reusable CV plan, and a numeric deployment payload. Legacy formats 1/2 remain supported. Construct a plan once to reuse partitions across feature schemas.

Folds balance outcome ranges using consecutive outcome-ranked blocks, with randomized ties and distinct randomized fold labels within each block. Predictor missingness filtering, median imputation, near-zero-variance removal and sample-SD scaling are learned separately in every inner training split. Each outer training set defines its outcome mean and sample SD; both its inner fits and its held-out outcomes use that scale. See [pairwiseLLM_warm_model](#) for preprocessing defaults. No PCA or outcome-based feature screening is used.

For each alpha, glmnet constructs a reference path on the current tuning dataset with its own preprocessing: 100 requested lambdas, minimum ratio 0.01 if retained predictors outnumber rows, otherwise 0.0001. The actual returned path is stored. Each inner training split fits those exact penalties with its own preprocessing; reference preprocessing is not used to transform inner rows. Candidate penalties use the entire tuning dataset, including its inner holdouts. No interpolation, extrapolation, or omitted fold is accepted for an eligible candidate. When an inner fit reports an iteration-limit failure and returns a verified prefix of the requested path, only its missing tail is invalidated. A lambda is eligible only if it converges in every inner fold. CV summaries and both lambda rules use eligible candidates only; an alpha with none is excluded. If all alphas are excluded, tuning fails. Reference-path generation, malformed engine output, other engine errors, and selected-lambda re-fit failures remain fatal. Full audits retain versioned candidate-validity records with requested and returned fold lambdas, convergence codes, counts, masks, invalid-tail counts, alpha eligibility and whether selection reached the smallest eligible lambda. Selection indices refer to the original path; unavailable losses and summaries are NA, never imputed. Historical fully converged audits remain valid.

CV error is the observation-count-weighted mean of fold mean squared errors. Its SE is $\sqrt{\text{weighted.mean}((\text{fold_mse} - \text{cvm})^2, \text{fold_sizes}) / (K - 1)}$. Errors within $1e-10 * \max(1, \text{abs}(a), \text{abs}(b))$ are numerical ties. Lambda minimum ties favor the largest penalty. Alpha selection compares CV error at each alpha's lambda minimum and favors smaller alpha in a tie. The default selected penalty is the largest lambda within one SE of that alpha's minimum error.

PLS uses the optional pls package with explicit method = "kernelpls", scale = FALSE, validation = "none", and centering. Preprocessing remains owned by this package. Each tuning context uses a common component grid bounded by every inner training matrix and the context refit: centered QR rank at tolerance $1e-7$, retained predictor count, training row count minus one, and 10. Explicit candidates are never silently dropped. Weighted MSE and SE follow the rules above; minimum-error ties and eligible 1-SE choices favor fewer components. Nonfinite or degenerate fits fail with context, without fallback. All candidate OOF predictions, fold losses, rank bounds and selection evidence are retained. Stored rank bounds are checked for consistency; recomputing rank itself requires the original feature table, which is not stored in the model.

RBF-SVR uses optional e1071, with type = "eps-regression", kernel = "radial", scale = FALSE, cross = 0, probability = FALSE, and fixed epsilon = 0.10. Every fit divides its gamma multiplier by its own retained predictor count. The complete Cartesian grid is tuned with the weighted MSE/SE above. Both minimum-error ties and eligible 1-SE choices favor lower cost, then lower gamma multiplier. Full audits retain candidate OOF values, fold losses, retained counts and actual gammas. Failed candidates are never omitted.

OOF (out-of-fold) predictions are predictions for rows excluded from their corresponding engine

fit. Calibration regresses standardized outcomes on selected-hyperparameter OOF predictions using ordinary least squares. Those same folds select hyperparameters: calibration fit statistics are not independent performance estimates. Only untouched outer predictions define validation metrics. Calibration needs at least three rows and full rank at QR tolerance 1e-7; constant/degenerate predictions or nonfinite coefficients cause an explicit error. Finite negative slopes are allowed. Undefined validation diagnostics are recorded as NA with reasons, without an identity-calibration fallback.

After outer validation, full-data tuning and OOF calibration precede the final all-row engine refit. Raw and calibrated predictions use within-task standardized units, not original BTL units. They are not Bayesian prior SDs. The selected engine package and withr are required only for model development; Python is required only for the text-input path. This function never installs software or downloads data.

Value

A portable `pairwiseLLM_warm_model` with deployment preprocessing, numeric engine parameters and OOF calibration, plus tuning traces, fold assignments, outer raw/calibrated predictions, transformed held-out outcomes, validation metrics and warnings. Audit records include item IDs and outcomes but no raw training texts. Each outer record includes its tuning, scaling, preprocessing, calibration and hyperparameters (linear models also record nonzero counts). Final tuning metadata is separate from outer validation. No backend fit is retained. PLS coefficients and $\bar{Y}$ means - $\bar{X}$ means β reproduce backend predictions on the stored preprocessed predictor scale without requiring pls at deployment. SVR stores numeric support vectors, dual coefficients, rho and actual gamma; prediction needs no e1071. Its linear coefficients/intercept are NULL.

See Also

```
predict.pairwiseLLM_warm_model(), ensemble_warm_start_models(), ensemble_warm_start_algorithms(),
save_warm_start_model()
```

```
Other adaptive warm start: ensemble_warm_start_algorithms(), ensemble_warm_start_models(),
extract_warm_start_features(), make_warm_start_cv_plan(), make_warm_start_prior(),
pairwiseLLM_warm_model, predict.pairwiseLLM_warm_algorithm_ensemble(), predict.pairwiseLLM_warm_ensemble(),
predict.pairwiseLLM_warm_model(), prepare_warm_start_model(), register_warm_start_model(),
save_warm_start_model(), summary.pairwiseLLM_warm_algorithm_ensemble(), summary.pairwiseLLM_warm_ensemble(),
summary.pairwiseLLM_warm_predictions(), warm_start_coefficients(), warm_start_feature_schema(),
warm_start_python_status()
```

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  local({
    # Synthetic features illustrate the interface, not predictive validity.
    example_features <- function(seed) {
      withr::local_seed(seed)
      fields <- warm_start_feature_schema()$feature
      x <- as.data.frame(matrix(runif(15 * length(fields)), nrow = 15))
      names(x) <- fields
      x$n_tokens <- 11:25
    }
  })
}
```

```

x$token_length_mean <- 2 + 10 * x$token_length_mean
x$token_length_std <- 0.2 + x$token_length_std
x$dale_chall_readability_score <- 5 + 20 * x$dale_chall_readability_score
x <- data.frame(item_id = as.character(1:15), x)
attr(x, "warm_start_schema") <- "writing_features_v1"
x
}
features <- example_features(3103)
theta <- 10 + 0.4 * features$n_tokens - 2 * features$token_length_mean
# A small alpha grid keeps this example fast; the default has 41 values.
model <- fit_warm_start_model(features$item_id, theta, "synthetic-a",
  features = features, alpha_grid = c(0, 1))
summary(model)
model$validation$metrics
})
}

```

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

ID1	Character ID for the first sample.
text1	Character containing the first sample text.
ID2	Character ID for the second sample.
text2	Character containing the second sample text.
model	Gemini model identifier (for example "gemini-3.5-flash-lite" or "gemini-3-flash-preview"). The value is interpolated into the path "{api_version}/models/<model>:generateContent".
trait_name	Short label for the trait (e.g. "Overall Quality").
trait_description	Full-text trait / rubric description.
prompt_template	Prompt template string, typically from set_prompt_template() . The template should embed <BETTER_SAMPLE> tags.
api_key	Optional Gemini API key (defaults to <code>Sys.getenv("GEMINI_API_KEY")</code>).
thinking_level	One of "minimal", "low", "medium", or "high". This controls the maximum depth of internal reasoning. <ul style="list-style-type: none"> • 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.

<code>include_thoughts</code>	Logical; if TRUE, requests explicit reasoning output from Gemini via <code>generationConfig\$thinkingConfig</code> and stores the first text part as <code>thoughts</code> , with subsequent parts collapsed into <code>content</code> . If FALSE (default), all text parts are collapsed into <code>content</code> and <code>thoughts</code> is NA.
<code>pair_uid</code>	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>").
<code>...</code>	Additional parameters. <code>store</code> accepts NULL or one non-missing logical value. TRUE enables and FALSE disables request logging; omission or NULL preserves the provider/project logging default. This controls Gemini Developer API logging, not all data retention. Any <code>thinking_budget</code> entry in <code>...</code> is ignored (and a warning is emitted) because Gemini 3 does not allow <code>thinking_budget</code> and <code>thinking_level</code> 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_typesafe_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `typesafe_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 <code>GenerateContent</code> 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 <code>"gemini-3.5-flash-lite"</code> .
<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 <code>"v1beta"</code> .
<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_errors()`, `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

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

Retrieval retries

Batch metadata and result-file GET requests retry HTTP 408, 429, all 5xx responses, and transport failures, with at most three total HTTP attempts per GET. Valid Retry-After seconds or HTTP dates take precedence; otherwise retries use exponential backoff starting at 0.5 seconds plus up to 0.25 seconds of jitter, capped at 30 seconds. Other HTTP errors fail immediately. Exhaustion raises the original error with the additional class `pairwiseLLM_batch_retry_exhausted`. These retries retrieve the same batch; they do not resubmit comparisons or create scientific failed-attempt rows.

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_errors()`, `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	<i>Retrieve a Gemini Batch job by name</i>
------------------	--

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 "v1beta".

Details

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

Value

A list representing the Batch job object.

Retrieval retries

Batch metadata and result-file GET requests retry HTTP 408, 429, all 5xx responses, and transport failures, with at most three total HTTP attempts per GET. Valid `Retry-After` seconds or HTTP dates take precedence; otherwise retries use exponential backoff starting at 0.5 seconds plus up to 0.25 seconds of jitter, capped at 30 seconds. Other HTTP errors fail immediately. Exhaustion raises the original error with the additional class `pairwiseLLM_batch_retry_exhausted`. These retries retrieve the same batch; they do not resubmit comparisons or create scientific failed-attempt rows.

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_errors()`, `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

<code>batch_name</code>	Character scalar giving the batch name.
<code>interval_seconds</code>	Polling interval in seconds. Defaults to 60.
<code>timeout_seconds</code>	Maximum total waiting time in seconds. Defaults to 24 hours (86400 seconds).
<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>verbose</code>	Logical; if TRUE, prints progress messages.

Value

The final Batch job object as returned by [gemini_get_batch](#).

Polling limits

HTTP retries occur within a logical status poll. Elapsed time includes retry waits, but `timeout_seconds` is checked between status requests; an in-flight GET and its retries can finish after that limit. Existing terminal-status and timeout return behavior is preserved.

Retrieval retries

Batch metadata and result-file GET requests retry HTTP 408, 429, all 5xx responses, and transport failures, with at most three total HTTP attempts per GET. Valid Retry-After seconds or HTTP dates take precedence; otherwise retries use exponential backoff starting at 0.5 seconds plus up to 0.25 seconds of jitter, capped at 30 seconds. Other HTTP errors fail immediately. Exhaustion raises the original error with the additional class `pairwiseLLM_batch_retry_exhausted`. These retries retrieve the same batch; they do not resubmit comparisons or create scientific failed-attempt rows.

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_errors\(\)](#), [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	<i>Retrieve a named prompt template</i>
---------------------	---

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", "ollama", and "typesafe" 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\(\)](#)
- "typesafe" → [typesafe_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",  
             "typesafe"),  
  endpoint = c("chat.completions", "responses", "decisions"),  
  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-small3.2:24b", "qwen3:32b", "gemma3:27b"). For "typesafe", omission defaults to "jev-1.13.0"; explicit versioned Jev identifiers and documented aliases are accepted without an availability guarantee or automatic fallback. Other backends still require model.
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> . For Decisions, omit or use NULL; custom decision instructions belong in <code>decision_template</code> in ... The same rule applies to TypeSafe; its omitted generative default is never evaluated.
backend	Character scalar indicating which LLM provider to use. One of "openai", "anthropic", "gemini", "vertex", "together", "ollama", or "typesafe".
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", "responses", or "decisions", matching <code>openai_compare_pair_live()</code> . For "anthropic", "gemini", "vertex", "together", and "ollama", this argument is currently ignored. For "typesafe", omit it or use "systemone"; other values fail.
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, TYPESAFE_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 <code>raw_response</code> 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 <code>temperature</code> , <code>top_p</code> , <code>logprobs</code> , <code>reasoning</code> , and <code>include_thoughts</code> . For "anthropic", "gemini", and "vertex" they are forwarded to the corresponding live helper and may include parameters such as <code>reasoning</code> , <code>include_thoughts</code> , <code>max_output_tokens</code> , <code>service_tier</code> , <code>thinking_level</code> , or provider-specific options. For "ollama", arguments are forwarded to <code>ollama_compare_pair_live()</code> and may include <code>host</code> , <code>think</code> , <code>num_ctx</code> , and other Ollama-specific controls. <code>service_tier</code> 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". For both OpenAI generative live

endpoints, "standard" and "default" send "default"; "auto", "flex", and "priority" are sent unchanged. Omission or NULL allows OpenAI's project-configured "auto" behavior. Model/tier availability is validated by OpenAI.

For OpenAI and Gemini Developer API, store accepts NULL, TRUE, or FALSE. It controls OpenAI response storage or Gemini request logging; omission or NULL preserves provider/project defaults. See the selected provider helper for details; this is not a cross-provider retention policy. Decisions accepts only pair_uid, decision_template, and prompt_cache_options in ...; generative, storage, and tier controls fail before HTTP. Use model = "gpt-6-luna". Its omitted prompt is never replaced with the generative default. Use [submit_llm_pairs\(\)](#) for resumable bulk Decisions execution. TypeSafe accepts only pair_uid and decision_template in ..., including rejection of unsupported settings explicitly supplied as NULL. Both decision backends retain independent confidence and full request provenance; probabilities never change binary BT/adaptive winner semantics. See [decision_bulk](#) for bulk execution; managed Batch remains unsupported.

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"), Responses API ("responses"), or opt-in Decisions API ("decisions") is used. Decisions uses the approved decision instructions and retains the complete binary probability distribution; see [openai_compare_pair_live\(\)](#) for its supported controls, return fields, validation, and errors. TypeSafe uses its Choice endpoint and the same decision instructions and probability-preserving contracts; see [typesafe_compare_pair_live\(\)](#). 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.

Decision failure retention

Decision failures may retain raw provider responses in memory even when include_raw = FALSE; saving the complete return value may persist sensitive sample text. See [decision_bulk](#) for default checkpoint/export retention.

See Also

- `openai_compare_pair_live()`, `anthropic_compare_pair_live()`, `gemini_compare_pair_live()`, `vertex_compare_pair_live()`, `together_compare_pair_live()`, `ollama_compare_pair_live()`, and `typesafe_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_typesafe_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `typesafe_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 = tmpl,
  backend       = "openai",
  endpoint      = "chat.completions",
  temperature   = 0
)

res_live$better_id

# Using the OpenAI responses endpoint with gpt-5.1 and reasoning = "low"
res_live_gpt5 <- llm_compare_pair(
  ID1          = samples$ID[1],
  text1        = samples$text[1],
```

```

    ID2          = samples$ID[2],
    text2        = samples$text[2],
    model        = "gpt-5.1",
    trait_name    = td$name,
    trait_description = td$description,
    prompt_template = tmpl,
    backend       = "openai",
    endpoint      = "responses",
    reasoning     = "low",
    include_thoughts = TRUE,
    temperature   = NULL,
    top_p         = NULL,
    logprobs      = NULL,
    include_raw   = TRUE
  )

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

# Example: single live comparison using a local Ollama backend
res_ollama <- llm_compare_pair(
  ID1 = samples$ID[1],
  text1 = samples$text[1],
  ID2 = samples$ID[2],
  text2 = samples$text[2],
  model = "mistral-small13.2:24b",
  trait_name = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  backend = "ollama",
  host = getOption(
    "pairwiseLLM.ollama_host",
    "http://127.0.0.1:11434"
  ),
  think = FALSE
)

res_ollama$better_id

## End(Not run)

```

llm_download_batch_results

Extract results from a pairwiseLLM batch object

Description

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

Usage

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

Arguments

x An object returned by `llm_submit_pairs_batch()` (class "pairwiseLLM_batch"), or a compatible list that contains a results element.

... Reserved for future use; currently ignored.

Value

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

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_errors()`, `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. Each retrieval GET has a bounded HTTP retry budget. Exhausted transient retrievals leave the job unfinished for the next polling round; permanent HTTP errors, parsing errors and local file errors propagate immediately. There is no overall round limit. Retrieval retries do not resubmit comparisons or add scientific failed-attempt rows.

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.

<code>per_job_delay</code>	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.
<code>write_results_csv</code>	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.
<code>keep_jsonl</code>	Logical; if FALSE, the <code>.jsonl</code> input and output files will be deleted after the job results have been parsed. Defaults to TRUE.
<code>write_registry</code>	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 llm_submit_pairs_multi_batch() for additional details on the registry format.
<code>tag_prefix</code> , <code>tag_suffix</code>	Character strings passed to parse_anthropic_batch_output() and parse_gemini_batch_output() . These tags mark the start and end of the "better" sample in each provider's output. The defaults match those used in the vignette.
<code>verbose</code>	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.
<code>write_combined_csv</code>	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.
<code>combined_csv_path</code>	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> .
<code>openai_max_retries</code>	Positive integer giving the total HTTP attempt budget per GET when downloading OpenAI batch output, including its metadata lookup. Defaults to 3. Retries honor <code>Retry-After</code> or use exponential backoff with jitter; <code>per_job_delay</code> controls spacing between jobs, not HTTP retries. There is no additional outer download retry loop. Status polling and other providers' retrievals use three HTTP attempts per GET.

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_errors()`, `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)
```

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. OpenAI endpoint = "decisions" is not a managed Batch endpoint and is rejected before entering a provider pipeline. Use `llm_compare_pair()` for single-pair Decisions; bulk decision execution is deferred.

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 <code>ID1</code> , <code>text1</code> , <code>ID2</code> , and <code>text2</code> . 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. <ul style="list-style-type: none"> 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). <ul style="list-style-type: none"> 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, store, prompt_caching, and Responses-only max_output_tokens. Select endpoint = "responses" explicitly for output limits. OpenAI Batch does not accept service_tier through this helper. 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. Gemini also accepts store = TRUE or FALSE to control logging for each request; omission or NULL preserves provider/project defaults. OpenAI accepts store on both endpoints; omission or NULL preserves provider defaults, including Responses storage for later retrieval. store = FALSE does not disable Batch file

retention or all data retention. Use `openai_download_batch_errors()` to retrieve OpenAI request failures.

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.

Batch prompt caching

The default disables prompt caching for these verified model IDs: gpt-5.6-luna, gpt-5.6-terra, gpt-5.6-sol, gpt-6-luna, gpt-6-sol, gpt-6-astra, and gpt-6.1-sol. Each request body gets `prompt_cache_options = list(mode = "explicit")`, with no breakpoints, on both endpoints. Set `prompt_caching = "implicit"` to omit that field and retain provider caching when prefix reuse is expected to be worthwhile.

Recognized earlier OpenAI models retain their historical request bodies when the policy is omitted or NULL; explicitly disabling caching on those models errors. Unknown IDs (including unlisted snapshots, future models and custom aliases) require an explicit "implicit" opt-in or a verified model ID. Support is an offline allowlist; version numbers and snapshot suffixes never establish support for disabling caching. Older GPT-3.5/4, GPT-5 through GPT-5.5 and o-series naming forms, including recognized legacy date-suffixed names, retain provider defaults.

Invalid policies and manual cache controls fail before submission, even with zero pairs. Raw `prompt_cache_options`, `prompt_cache_breakpoint`, `prompt_cache_key`, retention and TTL controls are not accepted by these builders. Keys affect cache routing/accounting; retention and TTL affect lifetime; `store = FALSE` affects response storage. None substitutes for disabling implicit cache breakpoints. Prompt text and keys are not rewritten. This policy affects future package-built Batch requests only, not live requests or already submitted jobs. See the [prompt-caching guide](#) and vignette("advanced-batch-workflows") for usage-based cost comparisons.

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_errors()`, `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")
tmpl <- 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,
  endpoint = "responses",
  store = FALSE,
  max_output_tokens = 1024
)
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() .
--------------------	---

model	Model identifier for the chosen backend. Passed through to the corresponding <code>run*_batch_pipeline()</code> function.
trait_name, trait_description, prompt_template	Parameters forwarded to <code>run_openai_batch_pipeline()</code> , <code>run_anthropic_batch_pipeline()</code> , or <code>run_gemini_batch_pipeline()</code> . See those functions for details.
backend	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.
batch_size	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> .
n_segments	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.
output_dir	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.
write_registry	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 <code>readr::read_csv()</code> and passed to <code>llm_resume_multi_batches()</code> for polling and resumption. If FALSE, the registry is returned in memory only.
keep_jsonl	Logical; if FALSE, the .jsonl 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.
verbose	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.
...	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. For OpenAI, <code>prompt_caching</code> is validated before creating files and its effective policy is reused across all segments and submission retries.
openai_max_retries	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.

Batch prompt caching

The default disables prompt caching for these verified model IDs: `gpt-5.6-luna`, `gpt-5.6-terra`, `gpt-5.6-sol`, `gpt-6-luna`, `gpt-6-sol`, `gpt-6-astra`, and `gpt-6.1-sol`. Each request body gets `prompt_cache_options = list(mode = "explicit")`, with no breakpoints, on both endpoints. Set `prompt_caching = "implicit"` to omit that field and retain provider caching when prefix reuse is expected to be worthwhile.

Recognized earlier OpenAI models retain their historical request bodies when the policy is omitted or `NULL`; explicitly disabling caching on those models errors. Unknown IDs (including unlisted snapshots, future models and custom aliases) require an explicit `"implicit"` opt-in or a verified model ID. Support is an offline allowlist; version numbers and snapshot suffixes never establish support for disabling caching. Older GPT-3.5/4, GPT-5 through GPT-5.5 and o-series naming forms, including recognized legacy date-suffixed names, retain provider defaults.

Invalid policies and manual cache controls fail before submission, even with zero pairs. Raw `prompt_cache_options`, `prompt_cache_breakpoint`, `prompt_cache_key`, retention and TTL controls are not accepted by these builders. Keys affect cache routing/accounting; retention and TTL affect lifetime; `store = FALSE` affects response storage. None substitutes for disabling implicit cache breakpoints. Prompt text and keys are not rewritten. This policy affects future package-built Batch requests only, not live requests or already submitted jobs. See the [prompt-caching guide](#) and vignette("advanced-batch-workflows") for usage-based cost comparisons.

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_errors()`, `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.

Legacy sessions without a saved predictive mode migrate to cold when no predictive prior exists, and `btl_only` otherwise. An absent pairing strategy migrates to hybrid. Saved TrueSkill values, the connected bootstrap policy, frozen queue and its index, and round progress remain authoritative; loading never recomputes predictions or initializes TrueSkill again.

Predictive-distribution sessions also record a versioned TrueSkill mapping, SD provenance, and integrity digests in state and session metadata. Loading validates their agreement without recomputing predictions or resetting ratings. Legacy sessions retain their existing migration rules and fixed-sigma policy.

`metadata.rds` records effective `warm_start_mode` and `pairing_strategy`, plus bootstrap policy/version and the predictive SHA-256 identity for session-level audit. Predictive queues are verified from frozen inputs and the digest without repeating graph selection. Missing legacy bootstrap policy fields retain shuffled behavior. `summarize_adaptive()` with `include_bootstrap = TRUE` exposes the saved bootstrap audit. Direct step logs record `pairing_strategy`, the presented A-over-B TrueSkill probability `p_ij`, and `target_distance` (missing for random pairing). Predictive vectors and provenance are retained once in `state$predictive_prior`.

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("pwllm-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 backend = "openai"; 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()`, `make_adaptive_judge_replay()`, `make_adaptive_replay_reservoir()`, `summarize_adaptive()`, `validate_adaptive_replay()`

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_adaptive_judge_replay

Create an offline judge from frozen directed outcomes

Description

Replay the stored result for the exact presented (A_id, B_id). Reverse judgments are independent stored observations: they are never inferred by complementing the forward result. No provider calls, random draws, or step-dependent outcomes are used.

Usage

```
make_adaptive_judge_replay(
  outcomes,
  item_ids,
  strict_use = TRUE,
  complete = TRUE
)
```

Arguments

outcomes	A directed outcome data frame (see validate_adaptive_replay()) or a make_adaptive_replay_reservoir object.
item_ids	Panel IDs. Required for data-frame input; inferred from a reservoir when omitted, or checked for agreement when supplied.
strict_use	Logical; reject repeated use of an exact ordered judgment. Default TRUE. FALSE permits repeated lookups for non-study inspection.
complete	Logical; require all $N * (N - 1)$ ordered pairs, including both orientations of every unordered pair. Default TRUE. FALSE permits a partial table; requests for absent orientations still fail during replay.

Details

For directed-table studies, set `adaptive_config = list(dup_max_obs_relaxed = 2L)` when creating the adaptive state. This prevents hybrid's relaxed third observation at selection time. Normal presentation balancing and repeat reversal remain active. Direct strategies already cap unordered pairs at two observations.

Create a fresh judge for each independent replicate. Strict use records each successful lookup in the closure, even if the caller subsequently discards the updated state. The judge is not saved in an adaptive session. To resume, create a new judge from the same matrix and pass the loaded state to the runner; strict use also rejects keys already present in that state's committed history. The matrix and its provenance must be retained separately by the caller.

A [make_adaptive_replay_reservoir\(\)](#) object instead enables sparse, single-observation replay. Bind that object through `replay_reservoir` when creating state. The judge then requires matching reservoir identity, uses committed unordered-edge history for consumption, and preserves the stored

orientation. Discarding an updated state does not consume an observation. Recreate a matching judge after loading a session; state contains only the outcome-free manifest. `complete` applies only to directed data-frame input; `strict_use = FALSE` is unsupported for reservoirs.

Value

A function `judge(A, B, state = NULL, ...)` compatible with `adaptive_rank_run_live()`. `A` and `B` are one-row data frames with `item_id`. The result contains `is_valid = TRUE`, integer `Y`, and `judge_backend = "replay"`. Missing keys and strict reuse raise errors.

See Also

`validate_adaptive_replay()`, `adaptive_rank()`

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

Examples

```
ids <- c("a", "b", "c")
outcomes <- expand.grid(A_id = ids, B_id = ids, stringsAsFactors = FALSE)
outcomes[outcomes$A_id != outcomes$B_id, ]
outcomes$Y <- as.integer(outcomes$A_id < outcomes$B_id)
judge <- make_adaptive_judge_replay(outcomes, ids)
state <- adaptive_rank_start(ids, seed = 42,
  adaptive_config = list(dup_max_obs_relaxed = 2L))
state <- adaptive_rank_run_live(state, judge, n_steps = 3L, progress = "none")
```

`make_adaptive_replay_reservoir`

Create a sparse frozen adaptive replay reservoir

Description

Each allowed unordered edge has exactly one observed presentation and binary outcome. Subset to the primary observation layer before construction; exclude held-out edges and separate reversal audits. The allowed graph must connect every panel item. Reservoir replay supports ordinary within-set runs only.

Usage

```
make_adaptive_replay_reservoir(outcomes, item_ids)
```

Arguments

<code>outcomes</code>	Data frame with character <code>A_id</code> , character <code>B_id</code> , and binary <code>Y</code> (one means presented <code>A</code> wins). <code>Y</code> accepts logical, numeric zero/one, or character <code>"0"/"1"</code> ; factors, missing values, and other values are rejected.
<code>item_ids</code>	Unique non-blank character IDs for the active panel, with at least two items. IDs must match the adaptive state's item IDs.

Details

By default, all strategies and warm-start modes share a shuffled seeded tree of $N - 1$ allowed edges. Opt into a frozen predictive tree with `bootstrap_policy = "predictive_connected"` in `adaptive_rank_start()`, with coherent predictive TrueSkill initialization and Pollitt subsequent pairing. Subsequent selection uses unused allowed edges, and commits their stored presentation without reversing or complementing outcomes. Consumption follows committed history, so discarded/failed transactions do not consume observations. Existing statistical stopping rules still apply.

State stores an outcome-free manifest and identity; retain the reservoir to recreate the judge on resume. Identity includes the panel, edge membership, presentation and Y, but excludes row order and ancillary metadata. Changed identities are rejected before replay. Do not edit a constructed reservoir. IDs that make distinct allowed edges collide in the existing colon-separated adaptive history keys are rejected; use unambiguous panel IDs in that case.

Value

A pairwiseLLM_replay_reservoir object for replay_reservoir in `adaptive_rank_start()` or `adaptive_rank()`, and for `make_adaptive_judge_replay()`.

See Also

`make_adaptive_judge_replay()`, `adaptive_rank_start()`, `adaptive_rank_resume()`

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

Examples

```
ids <- c("a", "b", "c", "d")
frozen <- data.frame(A_id = c("b", "a", "d", "c"),
  B_id = c("a", "c", "a", "d"), Y = c(1L, 0L, 1L, 1L))
reservoir <- make_adaptive_replay_reservoir(frozen, ids)
state <- adaptive_rank_start(ids, seed = 42, replay_reservoir = reservoir,
  adaptive_config = list(pairing_strategy = "random"))
state <- adaptive_rank_run_live(state, make_adaptive_judge_replay(reservoir),
  n_steps = 4L, progress = "none")
```

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. The first and second positions follow the input row order; IDs are not sorted lexicographically. This function only prepares data and makes no provider requests.

Usage

```
make_pairs(samples)
```

Arguments

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

Details

With N samples there are $\text{choose}(N, 2)$ pairs: 20 samples produce 190 pairs, while 100 produce 4,950. Use [randomize_pair_order\(\)](#) to vary which sample appears first, and plan the comparison budget before submission.

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(20, 2) = 190
```

`make_warm_start_cv_plan`*Construct reusable warm-start cross-validation partitions*

Description

Construct reusable warm-start cross-validation partitions

Usage

```
make_warm_start_cv_plan(  
  ids,  
  theta,  
  task_id,  
  seed = 1L,  
  outer_folds = 5L,  
  inner_folds = 5L  
)
```

Arguments

<code>ids</code>	Unique item IDs in the same order as <code>theta</code> .
<code>theta</code>	Finite numeric scores from one assessment.
<code>task_id</code>	One nonblank assessment label.
<code>seed</code>	Integer random seed, default 1.
<code>outer_folds, inner_folds</code>	Fold counts, each at least two. Counts are never reduced automatically; every training split needs three nonconstant outcomes.

Details

Partitions use outcome-ranked blocks with randomized ties and fold labels. Draw order is outer folds, inner folds for outer training sets in fold order, then full-data inner folds. The caller's RNG kind and seed state are preserved. Plans are independent of feature schema and engine. A supplied plan is checked before extraction or fitting and is never regenerated or silently realigned. Its digest detects accidental changes, not authorship. Plans contain outcomes and item IDs; they are development evidence, not anonymized artifacts.

Value

A portable `pairwiseLLM_warm_cv_plan` list, format 1. It contains exact ordered IDs and outcomes, task identity, seed/RNG provenance, named outer and inner assignments, and an integrity digest. Save with `saveRDS()`.

See Also

`fit_warm_start_model()`

Other adaptive warm start: `ensemble_warm_start_algorithms()`, `ensemble_warm_start_models()`, `extract_warm_start_features()`, `fit_warm_start_model()`, `make_warm_start_prior()`, `pairwiseLLM_warm_model()`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_ensemble()`, `predict.pairwiseLLM_warm_model()`, `prepare_warm_start_model()`, `register_warm_start_model()`, `save_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_ensemble()`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_coefficients()`, `warm_start_feature_schema()`, `warm_start_python_status()`

Examples

```
if (requireNamespace("withr", quietly = TRUE)) {
  plan <- make_warm_start_cv_plan(as.character(1:20), seq_len(20), "example")
  plan$outer_foldid
}
```

`make_warm_start_prior` *Convert warm-start predictions to Bayesian BTL priors*

Description

Convert warm-start predictions to Bayesian BTL priors

Usage

```
make_warm_start_prior(predictions, ids = NULL, prior_sd = 0.5)
```

Arguments

<code>predictions</code>	Calibrated single-model or ensemble predictions, or finite numeric scores with names or explicit ids. Numeric input is an expert choice of relative prior location, not an automatic calibration method.
<code>ids</code>	Active item IDs. For prediction tables and named scores these must match the input ID set exactly and determine output order. For unnamed scores they identify input positions. Defaults to the IDs in the input.
<code>prior_sd</code>	Positive finite scalar or vector. An unnamed vector follows input order; a named vector aligns by ID. Defaults to 0.5. This default is a chosen prior width, not learned uncertainty calibration.

Details

Single-model values must have learned OOF calibration. Ensembles use their equal-weight calibrated mean. Scores are centered over the active items in R; adaptive scoped refits subset the saved scores and center again in that scope. Calibration is not applied twice and original training BTL units are not used. Ensemble component predictions and sample disagreement SD remain diagnostics; disagreement never supplies the Bayesian prior SD automatically.

The normal prior applies to `theta_raw`; Stan centers this to obtain `theta`. Centering induces dependence, so the supplied SD is not the marginal SD of centered `theta`. Without predictive input, BTL retains raw prior mean 0 and SD 1. Adaptive BTL consumes this prior only in `btl_only` and both modes. Adaptive `trueskill_only` and both also use its centered `prior_mean` to initialize TrueSkill locations as $25 + (25/3) * \text{prior_mean}$, with unchanged sigma by default. Explicit `warm_start_trueskill = "predictive_distribution"` additionally maps $\text{sigma} = (25/3) * \text{prior_sd}$, keeping $\text{beta} = 25/6$, without clipping. A prior object supplies its stored SD; model input requires explicit SD in that mode. This assumes comparable latent scales and does not turn raw-`theta` SD into the marginal SD of centered effects. Upstream workflows must establish calibration; explicit SD alone does not prove it. Scalar SD is a sensitivity assumption, not essay-specific calibration. Ensemble disagreement never supplies SD automatically. By default, every adaptive mode retains the same seeded connected shuffled bootstrap. The separate `bootstrap_policy = "predictive_connected"` opt-in uses frozen mapped means and SDs to build an allowed spanning tree; see `adaptive_rank_start()` for its reservoir and selection requirements. Standalone `fit_bayes_btl_mcmc()` retains its `warm_start_prior` interface without the adaptive four-mode argument.

Value

A version-1 `pairwiseLLM_warm_prior` list containing `item_id`, `scores`, `prior_mean`, `prior_sd`, compact diagnostics, provenance, and an integrity digest. Pass this object to `fit_bayes_btl_mcmc()` or `adaptive_rank_start()`.

See Also

`predict.pairwiseLLM_warm_model()`, `predict.pairwiseLLM_warm_ensemble()`, `ensemble_warm_start_algorithms`, `fit_bayes_btl_mcmc()`, `adaptive_rank_start()`, `adaptive_rank_resume()`

Other adaptive warm start: `ensemble_warm_start_algorithms()`, `ensemble_warm_start_models()`, `extract_warm_start_features()`, `fit_warm_start_model()`, `make_warm_start_cv_plan()`, `pairwiseLLM_warm_model`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_ensemble`, `predict.pairwiseLLM_warm_model()`, `prepare_warm_start_model()`, `register_warm_start_model()`, `save_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_ensemble`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_coefficients()`, `warm_start_feature_schema()`, `warm_start_python_status()`

Examples

```
prior <- make_warm_start_prior(c(a = -1, b = 0, c = 1))
prior$prior_mean
```

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_typesafe_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [typesafe_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", "decisions"),
  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"). Decisions currently accepts only "gpt-6-luna"; other model IDs fail before HTTP.
trait_name	Short label for the trait (e.g. "Overall Quality").
trait_description	Full-text definition of the trait.
prompt_template	Generative prompt template string. For Decisions, omit this argument or use NULL; explicitly supplied non-NULL templates fail. Use decision_template in ... for custom decision instructions.
endpoint	Which OpenAI endpoint to use: "chat.completions" (the unchanged default), "responses", or "decisions".
tag_prefix	Prefix for the better-sample tag. Must be omitted for Decisions.
tag_suffix	Suffix for the better-sample tag. Must be omitted for Decisions.
api_key	Optional OpenAI API key.
include_raw	Logical; if TRUE, adds a raw_response column.
...	Additional OpenAI parameters, for example temperature, top_p, logprobs, reasoning, service_tier, max_output_tokens, store, pair_uid, and (optionally) include_thoughts. max_output_tokens must be a positive integer and is supported only by the Responses endpoint. When pair_uid is supplied, it is used verbatim as custom_id. store accepts NULL or one non-missing logical value for either generative endpoint. TRUE/FALSE are sent unchanged; omission or NULL leaves the field absent and preserves the OpenAI default. For either generative endpoint, service_tier = "standard" or "default" sends "default"; explicit "auto", "flex", and "priority" are sent unchanged. Omission or NULL leaves the field absent, allowing OpenAI's project-configured "auto" behavior. Model/tier availability is validated by OpenAI. For Responses, store = FALSE disables response storage for later retrieval. This is not a general data-retention guarantee. The same validation rules for gpt-5 models are applied as in build_openai_batch_requests . When using the Responses endpoint with reasoning models, you can request reasoning summaries in the thoughts column by setting endpoint = "responses", a non-"none" reasoning effort, and include_thoughts = TRUE. For Decisions, only pair_uid, decision_template, and prompt_cache_options are supported in ...; see below. Duplicate, unnamed, unsupported, or misspelled settings fail before credentials or HTTP.

Details

It supports 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. The opt-in Decisions endpoint ("/v1/decisions") instead uses one structured binary choice question; see the Decisions section below.

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. Failed Decisions always retain their response, including invalid JSON text, regardless of include_raw.

Decisions additionally returns the columns documented in [decision_results](#), including integer winner_pos, pair_uid, both probabilities, p_chosen, independent decision_confidence, full_probabilities_json, scientific_provenance/fingerprint, requested/resolved model, and provider x-request-id. object_type is "decision"; thoughts and content are NA. prompt_cached_tokens, prompt_cache_write_tokens, and reasoning_tokens retain reported usage details, including zero; unavailable counts are NA. request_controls_json records submitted prompt_cache_options (or null) and effective_prompt_cache_options with effective mode and TTL. These transport settings are separate from the scientific fingerprint. retry_failures is a list-column of transient attempt diagnostics, consumed by the existing normalizer for failed_attempts.

Decisions

Decisions is single-pair, single-trait and binary only. input is the exact shared JSON-string evidence built by [decision_request](#); questions contains one named choice question, `writing_trait`, with the ordered `SAMPLE_1` and `SAMPLE_2` values and symmetric descriptions. Omission of the generative prompt does not evaluate [set_prompt_template\(\)](#). The default instructions are the exact approved `decision_cj_v1`. A custom `decision_template` must contain `{TRAIT_NAME}` and `{TRAIT_DESCRIPTION}`, and cannot contain sample placeholders or generative output tags. The complete trait definition is inserted verbatim and fingerprinted. No instructions request generated probabilities, reasoning traces, or XML.

The selected choice is preserved, never replaced with an argmax. Both probabilities must be finite numbers in $[0, 1]$ summing to 1 within $1e-6$; exact 0, 0.5 and 1 are kept. Independent confidence is required in $[0, 1]$. A selection below the alternative by more than $1e-8$ is a diagnostic failure, retaining the original choice and probabilities. Malformed JSON/answers, refusals, and incomplete distributions return classified failed rows; they are never valid binary evidence or automatically retried. See [decision_results](#) for error codes and failure retention. HTTP errors/timeouts use the existing transport/retry conditions and attempt metadata; no generative fallback occurs.

`prompt_cache_options` is NULL or a nonempty named list with only `mode` ("implicit", "explicit", or NULL) and/or `t1` ("30m"). Omitted values use implicit mode and a 30-minute minimum TTL. Explicit mode disables the implicit breakpoint; this helper adds no explicit breakpoints, so explicit mode disables prompt caching for this request. Cache entries can persist longer than the TTL. This is caching, not request storage or a general data-retention guarantee. Account/project data controls still apply. `store`, `service/processing tiers`, `cache keys`, `breakpoints`, and `Batch cache policy controls` are unsupported here. All supplied generative controls, including `temperature`, `top_p`, `logprobs`, `reasoning`, `max_output_tokens`, and `include_thoughts`, fail even when NULL or FALSE.

Before request construction, the scientific body is checked independently against provenance, exact ordered text/evidence hashes, instructions, question, and choices. The serialized JSON is checked again, attached as exact UTF-8 bytes, and verified before every HTTP attempt, including retries. Inconsistent requests abort with `pairwiseLLM_decision_request_error` and sample-free diagnostics before HTTP; no automatic repair is performed. A fingerprint checks identity/integrity and is not an authenticated signature.

Requests contain the complete sample texts. Provenance excludes samples but includes full trait/custom instruction text, which can itself be sensitive. Failed raw responses can also contain sensitive content; do not log publicly. These probabilities do not establish validated educational assessment accuracy and do not change binary BT/adaptive evidence. Use [submit_llm_pairs\(\)](#) or [submit_openai_pairs_live\(\)](#) for live bulk execution; managed Batch remains unsupported for Decisions. A live smoke run on 2026-10-16 exercised `gpt-6-luna` on 20 synthetic samples, 60 primary pairs and 10 reverse audits; see the model-compatibility vignette for the tested commit and limits. Probabilities are not established calibrated probabilities of assessment correctness. API availability remains provider/account dependent. The schema and controls follow the [OpenAI Decisions reference](#).

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_typesafe_pairs_live(), submit_vertex_pairs_live(), together_compare_pair_live(),
typesafe_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)

# 3. One decision with both choice probabilities (also incurs API costs)
res_decision <- openai_compare_pair_live(
  "A", "Synthetic sample A.", "B", "Synthetic sample B.",
  model = "gpt-6-luna", trait_name = "Organization",
  trait_description = "Logical sequencing and connections between ideas.",
  endpoint = "decisions", pair_uid = "example-forward",
  prompt_cache_options = list(mode = "explicit")
)
res_decision[c("better_id", "p_sample_1", "p_sample_2", "decision_confidence")]

## 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

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

Download the error file for an OpenAI batch

Description

Retrieves batch metadata and downloads the raw JSONL file identified by `error_file_id`. This works even when the batch has no `output_file_id`. An error is raised if no valid error file ID is available; the message includes the batch ID and status. HTTP and local file-write errors propagate. An existing local file is overwritten only after its content is downloaded.

Usage

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

Arguments

<code>batch_id</code>	The batch ID (e.g. "batch_abc123").
<code>path</code>	Local file path to write the downloaded error .jsonl file.
<code>api_key</code>	Optional OpenAI API key. Defaults to <code>Sys.getenv("OPENAI_API_KEY")</code> .

Details

Successful requests are retrieved with `openai_download_batch_output()`. Reconcile both files against submitted requests by `custom_id`, not line order. This helper does not parse errors or retry failed comparisons.

Value

Invisibly, the path to the downloaded file.

Retrieval retries

Batch metadata and result-file GET requests retry HTTP 408, 429, all 5xx responses, and transport failures, with at most three total HTTP attempts per GET. Valid `Retry-After` seconds or HTTP dates take precedence; otherwise retries use exponential backoff starting at 0.5 seconds plus up to 0.25 seconds of jitter, capped at 30 seconds. Other HTTP errors fail immediately. Exhaustion raises the original error with the additional class `pairwiseLLM_batch_retry_exhausted`. These retries retrieve the same batch; they do not resubmit comparisons or create scientific failed-attempt rows.

See Also

[openai_get_batch\(\)](#), [openai_download_batch_output\(\)](#)

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\(\)](#), [write_openai_batch_file\(\)](#)

Examples

```
## Not run:
# Requires OPENAI_API_KEY and a batch with an error_file_id.
openai_download_batch_errors("batch_abc123", "batch_errors.jsonl")
errors <- lapply(readLines("batch_errors.jsonl"), jsonlite::fromJSON)

## 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. Request failures are stored separately; use [openai_download_batch_errors\(\)](#) to retrieve them and reconcile records by `custom_id`, not line order.

Usage

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

Arguments

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

Value

Invisibly, the path to the downloaded file.

Retrieval retries

Batch metadata and result-file GET requests retry HTTP 408, 429, all 5xx responses, and transport failures, with at most three total HTTP attempts per GET. Valid Retry-After seconds or HTTP dates take precedence; otherwise retries use exponential backoff starting at 0.5 seconds plus up to 0.25 seconds of jitter, capped at 30 seconds. Other HTTP errors fail immediately. Exhaustion raises the original error with the additional class `pairwiseLLM_batch_retry_exhausted`. These retries retrieve the same batch; they do not resubmit comparisons or create scientific failed-attempt rows.

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_errors()`, `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 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.

Retrieval retries

Batch metadata and result-file GET requests retry HTTP 408, 429, all 5xx responses, and transport failures, with at most three total HTTP attempts per GET. Valid Retry-After seconds or HTTP dates take precedence; otherwise retries use exponential backoff starting at 0.5 seconds plus up to 0.25 seconds of jitter, capped at 30 seconds. Other HTTP errors fail immediately. Exhaustion raises the original error with the additional class `pairwiseLLM_batch_retry_exhausted`. These retries retrieve the same batch; they do not resubmit comparisons or create scientific failed-attempt rows.

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_errors()`, `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 logical status polls, excluding internal HTTP retries. This is mainly useful for testing; default is <code>Inf</code> .
<code>api_key</code>	Optional OpenAI API key.
<code>verbose</code>	Logical; if <code>TRUE</code> , 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()`.

Polling limits

HTTP retries occur within a logical status poll. Elapsed time includes retry waits, but `timeout_seconds` is checked between status requests; an in-flight GET and its retries can finish after that limit. Existing terminal-status and timeout return behavior is preserved.

Retrieval retries

Batch metadata and result-file GET requests retry HTTP 408, 429, all 5xx responses, and transport failures, with at most three total HTTP attempts per GET. Valid `Retry-After` seconds or HTTP dates take precedence; otherwise retries use exponential backoff starting at 0.5 seconds plus up to 0.25 seconds of jitter, capped at 30 seconds. Other HTTP errors fail immediately. Exhaustion raises the original error with the additional class `pairwiseLLM_batch_retry_exhausted`. These retries retrieve the same batch; they do not resubmit comparisons or create scientific failed-attempt rows.

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_errors()`, `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

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

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_errors()`, `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)
```

pairwiseLLM_warm_model

Portable task-specific warm-start models

Description

A warm-start model predicts relative quality from frozen writing features before pairwise comparisons are collected. Each model represents one task or assessment; unlinked BT/BTL scores from different tasks must not be pooled.

Usage

```
## S3 method for class 'pairwiseLLM_warm_model'
summary(object, ...)

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

Arguments

object	A pairwiseLLM_warm_model object.
...	Reserved for future extensions; must be empty.
x	A pairwiseLLM_warm_model object.

Details

The fixed-hyperparameter core is internal. It does not tune hyperparameters, run cross-validation, or learn calibration. `fit_warm_start_model()` adds nested validation and OOF calibration. Core models remain explicitly uncalibrated.

Format version 1 is an S3 list with these deployment fields:

- `format_version`: integer 1L.
- `schema` and `features`: frozen schema identity and original feature order, including predictors removed during training.
- `preprocessing`: original/retained feature names, named removal reasons, training missing fractions, retained medians, centers, sample SDs, thresholds, training row count, and SD convention.
- `coefficients` and `intercept`: finite linear coefficients on the stored preprocessed predictor scale, in retained feature order, and intercept.
- `outcome`: `definition = "within_task_z"`, original training mean and sample SD, and `sd_convention = "sample"`. The target is $(\text{theta} - \text{mean}) / \text{sd}$.
- `calibration`: `status = "uncalibrated"`, with `intercept = NULL` and `slope = NULL` for core fits. Public fits store `status = "oof_linear"`, learned intercept/slope, calibration row count, method, QR tolerance and OOF source.
- `training`: task ID, training row count, requested alpha/lambda, nonzero coefficient count, engine/version, and package version.
- `tuning` and `validation`: `NULL` for core fits; public fits retain exact alpha/lambda traces, fold preprocessing, OOF calibration inputs, outer held-out predictions and metrics, and warning messages. See `fit_warm_start_model()`.

New public fits use format 3, with `audit_status = "full"`, a validated `cv_plan`, compact `cv_identity`, and a typed numeric `engine_payload`. Linear coefficients/intercept remain checked compatibility views. Training also records selected hyperparameters. The complete plan binds exact ordered IDs, original outcomes and every nested partition. No backend fit object is stored. Format-3 audit omission keeps format 3 and compact identity but removes the plan. PLS models record `training$hyperparameters$ncomp`, with complete component-grid, rank-bound, fold-loss and candidate OOF evidence in full models. Reduced PLS artifacts retain the selected count and tuning settings, but omit row evidence. SVR stores `type = "rbf_svr"` payloads with a retained-feature-ordered numeric support_vectors matrix, numeric dual, rho, and actual gamma. Linear coefficients/intercept are `NULL`; no linear nonzero count is reported. Training hyperparameters record cost, gamma multiplier, fixed epsilon and actual gamma. Full audits retain all candidate OOF/loss values and split-specific gammas; reduced models retain deployment parameters and tuning settings. Numeric support vectors remain necessary deployment data; reduction is not anonymization. `summary()` and `print()` report components for PLS, alpha/lambda for glmnet, and hyperparameters/support-vector count for SVR.

For legacy models, `prepare_warm_start_model()` audit omission creates format 2 with `audit_status = "summary_only"`. It preserves deployment parameters and validation summaries but omits row-level evidence. Those summaries cannot be recomputed from the reduced artifact. Format 1 remains fully audited.

Preprocessing is fitted only on the supplied training rows. It first removes all-missing columns and columns with missing fraction strictly above 0.20, then median-imputes remaining columns. It removes constants and columns with unique-value fraction at most 0.10 AND most-common/second-most-common frequency ratio strictly above 19. Frequencies are calculated after imputation. Retained columns are centered and divided by sample SD (denominator $n - 1$). Internal controls permit changing these thresholds; fitted values are stored. No surviving predictors is an error. With one survivor, an excluded zero column satisfies glmnet's two-column input requirement only during fitting; it is never a schema feature or deployment coefficient.

The internal fixed fit requires at least three observations, explicit alpha in $[0, 1]$, and finite lambda greater than or equal to zero. It uses Gaussian glmnet with an intercept and `standardize = FALSE`, with solver threshold $1e-12$ and maximum 10000000 iterations, directly at the requested lambda. Outcomes are standardized before fitting. No PCA or feature screening based on outcomes is used, including when predictors outnumber observations.

Prediction uses only the stored preprocessing and numeric engine payload, never a serialized back-end object. Python is optional for extraction; glmnet, pls and e1071 are optional development engines. None is needed to inspect or predict from a deployment object with precomputed features. Schema metadata is an input contract, not verified extraction provenance or evidence of predictive validity.

Value

`print()` invisibly returns the model. `summary()` returns a named list describing the task, target, preprocessing, hyperparameters, and calibration status and stored nested-validation metrics when available.

See Also

`fit_warm_start_model()`, `predict.pairwiseLLM_warm_model()`, `warm_start_coefficients()`, `prepare_warm_start_model()`

Other adaptive warm start: `ensemble_warm_start_algorithms()`, `ensemble_warm_start_models()`, `extract_warm_start_features()`, `fit_warm_start_model()`, `make_warm_start_cv_plan()`, `make_warm_start_prior()`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_ensemble()`, `predict.pairwiseLLM_warm_model()`, `prepare_warm_start_model()`, `register_warm_start_model()`, `save_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_ensemble()`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_coefficients()`, `warm_start_feature_schema()`, `warm_start_python_status()`

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  local({
    # Synthetic features illustrate the interface, not predictive validity.
    example_features <- function(seed) {
```

```

    withr::local_seed(seed)
    fields <- warm_start_feature_schema()$feature
    x <- as.data.frame(matrix(runif(15 * length(fields)), nrow = 15))
    names(x) <- fields
    x$n_tokens <- 11:25
    x$token_length_mean <- 2 + 10 * x$token_length_mean
    x$token_length_std <- 0.2 + x$token_length_std
    x$dale_chall_readability_score <- 5 + 20 * x$dale_chall_readability_score
    x <- data.frame(item_id = as.character(1:15), x)
    attr(x, "warm_start_schema") <- "writing_features_v1"
    x
  }
  features <- example_features(3103)
  theta <- 10 + 0.4 * features$n_tokens - 2 * features$token_length_mean
  # A small alpha grid keeps this example fast; the default has 41 values.
  model <- fit_warm_start_model(features$item_id, theta, "synthetic-a",
    features = features, alpha_grid = c(0, 1))
  print(model)
  summary(model)
})
}

```

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$cached_tokens` or Chat Completions' `prompt_tokens_details$cached_tokens`); otherwise NA.

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

prompt_cache_write_tokens Prompt tokens written to cache, from `input_tokens_details$cache_write_tokens` or `prompt_tokens_details$cache_write_tokens`; otherwise NA. Missing counts are not inferred from the request policy.

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)
```

predict.pairwiseLLM_bt_alpha

Predict pairwise win probabilities from an alpha-adjusted Bradley-Terry fit

Description

Calculate $\text{plogis}(\theta_1 - \theta_2)$, the probability that the first item wins. These plug-in probabilities do not integrate over uncertainty and have no tie, lapse, or positional parameter.

Usage

```
## S3 method for class 'pairwiseLLM_bt_alpha'
predict(object, newdata = NULL, ...)
```

Arguments

object An alpha-adjusted fit from `fit_bt_model()` with engine = "alpha".

newdata	A data frame containing object1 and object2 item IDs. NULL uses the original comparisons in their original order. IDs must be nonmissing and present in the fitted model. Repeated pairs are allowed; comparing an item with itself returns 0.5.
...	Reserved; additional arguments are rejected.

Value

A numeric vector of first-item win probabilities in input row order.

See Also

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

Other frequentist models: [bootstrap_bt_model\(\)](#), [build_bt_data\(\)](#), [build_elo_data\(\)](#), [fit_bt_model\(\)](#), [fit_elo_model\(\)](#), [predict.pairwiseLLM_bt_firth\(\)](#), [predict.pairwiseLLM_bt_lapse\(\)](#), [scale_separation_reliability\(\)](#), [summarize_bt_fit\(\)](#)

Examples

```
comparisons <- data.frame(object1 = c("a", "a", "b"),
                          object2 = c("b", "c", "c"), result = c(1, 1, 1))
fit <- fit_bt_model(comparisons, engine = "alpha", alpha = 0.5)
predict(fit)
predict(fit, data.frame(object1 = "c", object2 = "a"))
```

predict.pairwiseLLM_bt_firth

Predict pairwise win probabilities from a Firth Bradley-Terry fit

Description

Calculate $\text{plogis}(\theta_1 - \theta_2)$, the probability that the first item wins. Predictions use the fitted item strengths, without a positional, lapse, or tie parameter. They do not integrate over estimation uncertainty.

Usage

```
## S3 method for class 'pairwiseLLM_bt_firth'
predict(object, newdata = NULL, ...)
```

Arguments

object	A Firth fit returned by fit_bt_model() with engine = "brglm2".
newdata	A data frame containing object1 and object2 item IDs. NULL uses the original comparisons in their original order. IDs must be nonmissing and present in the fitted model. Repeated pairs are allowed; comparing an item with itself returns 0.5.
...	Reserved; additional arguments are rejected.

Value

A numeric vector of first-item win probabilities in input row order. An empty data frame returns `numeric(0)`.

See Also

`fit_bt_model()`, `summarize_bt_fit()`

Other frequentist models: `bootstrap_bt_model()`, `build_bt_data()`, `build_elo_data()`, `fit_bt_model()`, `fit_elo_model()`, `predict.pairwiseLLM_bt_alpha()`, `predict.pairwiseLLM_bt_lapse()`, `scale_separation_reli`, `summarize_bt_fit()`

Examples

```
if (requireNamespace("brglm2", quietly = TRUE)) {
  comparisons <- data.frame(object1 = c("a", "a", "b"),
                             object2 = c("b", "c", "c"), result = c(1, 1, 1))
  fit <- fit_bt_model(comparisons, engine = "brglm2")
  predict(fit)
  predict(fit, data.frame(object1 = "c", object2 = "a"))
}
```

`predict.pairwiseLLM_bt_lapse`

Predict ordered comparisons from an experimental lapse BTL fit

Description

Calculate $(1 - \epsilon) * \text{plogis}(\theta_1 - \theta_2 + \beta) + \epsilon / 2$. Positive β favors the first presented item. Swapping the items generally does not give complementary probabilities unless β is zero. These are plug-in probabilities, without integration over estimation uncertainty. Valid epsilon-zero boundary fits also support prediction; their unavailable joint uncertainty does not invalidate the point estimates.

Usage

```
## S3 method for class 'pairwiseLLM_bt_lapse'
predict(object, newdata = NULL, ...)
```

Arguments

<code>object</code>	A validated experimental fit from <code>fit_bt_model()</code> with <code>engine = "lapse"</code> .
<code>newdata</code>	A data frame containing <code>object1</code> and <code>object2</code> IDs known to the fit. <code>NULL</code> uses the original comparisons in their original order. Repeated pairs and self-predictions are allowed; a self-prediction includes positional bias and need not equal 0.5. Empty input returns <code>numeric(0)</code> .
<code>...</code>	Reserved; additional arguments are rejected.

Value

Numeric first-item win probabilities in input row order.

See Also

`fit_bt_model()`, [integrated CJ workflow](#)

Other frequentist models: `bootstrap_bt_model()`, `build_bt_data()`, `build_elo_data()`, `fit_bt_model()`, `fit_elo_model()`, `predict.pairwiseLLM_bt_alpha()`, `predict.pairwiseLLM_bt_firth()`, `scale_separation_reli`, `summarize_bt_fit()`

<code>predict.pairwiseLLM_rubric_calibration</code>
<i>Predict rubric scores from a calibration</i>

Description

Predict rubric scores from a calibration

Usage

```
## S3 method for class 'pairwiseLLM_rubric_calibration'
predict(object, newdata = NULL, hard_score = c("median", "mode"), ...)
```

Arguments

<code>object</code>	A <code>pairwiseLLM_rubric_calibration</code> object.
<code>newdata</code>	Completed CJ result to predict, or <code>NULL</code> for the original items. Same-set scoring accepts the original completed result with unchanged item IDs, exact accepted scores and uncertainty, and matching fit/reference evidence. Item reordering is allowed; collection provenance does not affect scoring. Raw tables, independent cohorts, refits, and Phase B targets are not supported for same-set prediction. For <code>linked_anchors</code> , <code>NULL</code> scores the original reference items; explicit <code>newdata</code> must be a completed Phase B result (or its state) on the stored reference scale. It returns only target items across all spokes, in their input order. Every spoke must have an accepted Phase B refit and committed active hub-spoke evidence.
<code>hard_score</code>	Ordinal hard-score rule: <code>"median"</code> (default) or <code>"mode"</code> . The median is the lowest category whose cumulative probability is at least 0.5; equality at a median cutpoint therefore selects the lower category. Modal ties select the lowest category. Ordinal output always retains all category probabilities, both decision rules, and expected level. Both choices give the same deterministic category for percentile scoring.
<code>...</code>	Reserved arguments; currently must be empty.

Value

For percentile scoring, a tibble with item_id, accepted source theta, integer category in 1:K, original-label rubric_score, and extrapolated (outside the fitted CJ range). Under the original-result restriction, extrapolation flags are always false. Stored cutpoints are reused unchanged. No category probabilities or probabilistic summaries are returned. Both ordinal methods include the same five columns, plus probabilities, a list-column of numeric K-vectors named by the original ordered labels; integer median_category and modal_category; and expected_level = sum(k * P(Y = k)) for internal indices k in 1:K, regardless of the original labels' numerical spacing. This expected rubric level is a continuous summary, not a replacement for the ordinal result. category and rubric_score use the requested hard_score rule. Ordinal extrapolated flags use the labeled calibration range, so unlabeled source items can be extrapolated; endpoints are included in the range. Stored standardization is reused. Unfitted objects fail clearly. Linked target output adds set_id (target spoke), source_item_id, global_item_id, and theta_sd (accepted Phase B uncertainty, possibly NA). Its linking attribute contains normalized reference, fit_contract, provenance (including hub/spoke IDs and linking stage logs), diagnostics, and reliability. Provenance estimation/uncertainty methods reflect validated spoke contracts, including accepted-state reuse. Each is a single string when all spokes agree, otherwise the distinct strings in spoke order; per-spoke contracts retain the complete attribution. Failed upstream diagnostics warn and remain available; accepted scores do not assert adequate precision or successful stopping. CJ uncertainty is retained as metadata and is not propagated into category probabilities. Reference predictions and same-set output keep their existing columns and have no linking attribute.

See Also

[fit_rubric_calibration\(\)](#), [evaluate_rubric_predictions\(\)](#)

Other rubric calibration: [evaluate_rubric_predictions\(\)](#), [fit_rubric_calibration\(\)](#), [prepare_linked_rubric_ref](#)

predict.pairwiseLLM_warm_algorithm_ensemble
<i>Predict with a same-task algorithm ensemble</i>

Description

Predict with a same-task algorithm ensemble

Usage

```
## S3 method for class 'pairwiseLLM_warm_algorithm_ensemble'
predict(object, newdata = NULL, ..., texts = NULL, ids = NULL, python = NULL)
```

Arguments

object	A full or summary-only ensemble_warm_start_algorithms() artifact.
newdata	Precomputed frozen feature rows, including item_id and warm_start_schema metadata. Mutually exclusive with texts.
...	Reserved; must be empty.

texts	Character texts, paired positionally with explicit ids.
ids	Item IDs for text input only.
python	Optional explicitly selected Python interpreter for text extraction only; see extract_warm_start_features()

Value

A `pairwiseLLM_warm_predictions` tibble with ordered component calibrated predictions, their `ensemble_mean`, and diagnostic sample `ensemble_sd`. Component results and shared CV identity are retained as metadata. Prediction uses stored numeric payloads without loading any fitting backend. Text input extracts features once; precomputed features require no Python.

See Also

[ensemble_warm_start_algorithms\(\)](#), [make_warm_start_prior\(\)](#)

Other adaptive warm start: [ensemble_warm_start_algorithms\(\)](#), [ensemble_warm_start_models\(\)](#), [extract_warm_start_features\(\)](#), [fit_warm_start_model\(\)](#), [make_warm_start_cv_plan\(\)](#), [make_warm_start_prior\(\)](#), [pairwiseLLM_warm_model](#), [predict.pairwiseLLM_warm_ensemble\(\)](#), [predict.pairwiseLLM_warm_model\(\)](#), [prepare_warm_start_model\(\)](#), [register_warm_start_model\(\)](#), [save_warm_start_model\(\)](#), [summary.pairwiseLLM_warm_algorithm_ensemble\(\)](#), [summary.pairwiseLLM_warm_ensemble\(\)](#), [summary.pairwiseLLM_warm_predictions\(\)](#), [warm_start_coefficients\(\)](#), [warm_start_feature_schema\(\)](#), [warm_start_python_status\(\)](#)

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("pls", quietly = TRUE) && requireNamespace("withr", quietly = TRUE)) {
  local({
    withr::local_seed(259L)
    fields <- warm_start_feature_schema()$feature
    x <- as.data.frame(matrix(runif(20 * length(fields)), nrow = 20))
    names(x) <- fields
    x$n_tokens <- 11:30
    x <- data.frame(item_id = as.character(1:20), x)
    attr(x, "warm_start_schema") <- "writing_features_v1"
    theta <- x$n_tokens - 3 * x$token_length_mean
    plan <- make_warm_start_cv_plan(x$item_id, theta, "synthetic", seed = 259L)
    en <- fit_warm_start_model(x$item_id, theta, "synthetic", features = x,
      cv_plan = plan, alpha_grid = c(0, 1))
    pls <- fit_warm_start_model(x$item_id, theta, "synthetic", features = x,
      cv_plan = plan, engine = "pls", engine_control = list(ncomp = 1L))
    ensemble <- ensemble_warm_start_algorithms(elastic_net = en, pls = pls)
    print(ensemble)
    summary(ensemble)$validation$metrics
    predictions <- predict(ensemble, x)
    make_warm_start_prior(predictions, prior_sd = 0.5)
  })
}
```

predict.pairwiseLLM_warm_ensemble

Predict with an equal-weight warm-start ensemble

Description

Predict with an equal-weight warm-start ensemble

Usage

```
## S3 method for class 'pairwiseLLM_warm_ensemble'
predict(object, newdata = NULL, ..., texts = NULL, ids = NULL, python = NULL)
```

Arguments

object	A valid ensemble_warm_start_models() ensemble.
newdata	Precomputed frozen feature rows, including item_id and warm_start_schema metadata. Mutually exclusive with texts.
...	Reserved; must be empty.
texts	Character texts, paired positionally with explicit ids.
ids	Item IDs for text input only.
python	Optional explicitly selected Python interpreter for text extraction only; see extract_warm_start_features()

Details

Text features are extracted once. Each model independently applies its stored preprocessing and OOF calibration to the same raw feature rows. Original BTL scales never enter the average; calibration is not applied a second time. Every component must return exactly the input IDs in order and finite numeric predictions. A failure names the component; no model or row is dropped. Pre-computed prediction needs neither Python nor a fitting backend.

Ensemble SD measures between-model disagreement. It is diagnostic, not a Bayesian prior SD or a calibrated uncertainty estimate. These predictions do not perform BTL prior conversion. No ensemble validation metric is inferred from component validation metrics.

Value

A pairwiseLLM_warm_predictions tibble in input order with item_id, one component_<name> calibrated prediction column per component, ensemble_mean, and ensemble_sd. SD is the sample SD (denominator k - 1) across all k calibrated predictions, including for exactly two components. Attributes warm_start_schema and warm_start_model retain schema and ensemble/component identity and format metadata. component_predictions retains complete named single-model results, including raw predictions; component_columns maps component names to output columns.

See Also

`ensemble_warm_start_models()`, `summary.pairwiseLLM_warm_predictions()`, `make_warm_start_prior()`

Other adaptive warm start: `ensemble_warm_start_algorithms()`, `ensemble_warm_start_models()`, `extract_warm_start_features()`, `fit_warm_start_model()`, `make_warm_start_cv_plan()`, `make_warm_start_prior()`, `pairwiseLLM_warm_model`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_model()`, `prepare_warm_start_model()`, `register_warm_start_model()`, `save_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_ensemble()`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_coefficients()`, `warm_start_feature_schema()`, `warm_start_python_status()`

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  local({
    # Synthetic features illustrate the interface, not predictive validity.
    example_features <- function(seed) {
      withr::local_seed(seed)
      fields <- warm_start_feature_schema()$feature
      x <- as.data.frame(matrix(runif(15 * length(fields)), nrow = 15))
      names(x) <- fields
      x$n_tokens <- 11:25
      x$token_length_mean <- 2 + 10 * x$token_length_mean
      x$token_length_std <- 0.2 + x$token_length_std
      x$dale_chall_readability_score <- 5 + 20 * x$dale_chall_readability_score
      x <- data.frame(item_id = as.character(1:15), x)
      attr(x, "warm_start_schema") <- "writing_features_v1"
      x
    }
    features <- example_features(3103)
    theta <- 10 + 0.4 * features$n_tokens - 2 * features$token_length_mean
    # A small alpha grid keeps this example fast; the default has 41 values.
    model <- fit_warm_start_model(features$item_id, theta, "synthetic-a",
      features = features, alpha_grid = c(0, 1))
    features_b <- example_features(3104)
    theta_b <- 30 + features_b$n_tokens - 3 * features_b$token_length_mean
    model_b <- fit_warm_start_model(features_b$item_id, theta_b, "synthetic-b",
      features = features_b, alpha_grid = c(0, 1))
    ensemble <- ensemble_warm_start_models(assessment_a = model, assessment_b = model_b)
    predictions <- predict(ensemble, features)
    head(predictions)
  })
}
```

`predict.pairwiseLLM_warm_model`

Predict relative quality from a portable warm-start model

Description

Predict relative quality from a portable warm-start model

Usage

```
## S3 method for class 'pairwiseLLM_warm_model'
predict(object, newdata, ...)
```

Arguments

object	A valid pairwiseLLM_warm_model object.
newdata	Precomputed feature rows with item_id, all original frozen schema columns, and matching warm_start_schema metadata. IDs must be unique, nonmissing and nonblank. Numeric IDs normalize to character. Extra columns are ignored; required columns cannot be omitted even if training removed them.
...	Reserved for future extensions; must be empty.

Details

Prediction applies training medians, centers, and sample SDs, followed by the stored numeric engine payload. Linear engines use intercept and coefficients; SVR uses $\exp(-\gamma * \text{squared_distance})$ $^{**} \text{dual} - \rho$. Prediction does not recompute preprocessing, train a model, load glmnet/pls/e1071, initialize Python, or check a Python environment. Predictions are not calibrated Bayesian prior means or prior standard deviations. See [pairwiseLLM_warm_model](#) for the portable model contract.

Value

A tibble in newdata row order with character item_id, numeric raw_prediction, and numeric calibrated_prediction. Raw predictions are on the within-task standardized outcome scale, not the original BT/BTL scale. Public fits apply the stored OOF calibration intercept and slope to raw values. For uncalibrated core fits, calibrated predictions are NA_real_, never identity-calibrated substitutes. Attributes warm_start_schema and warm_start_model record schema identity and model metadata (format version, task ID, outcome definition, and calibration status), respectively.

See Also

[fit_warm_start_model\(\)](#), [predict.pairwiseLLM_warm_ensemble\(\)](#), [make_warm_start_prior\(\)](#)

Other adaptive warm start: [ensemble_warm_start_algorithms\(\)](#), [ensemble_warm_start_models\(\)](#), [extract_warm_start_features\(\)](#), [fit_warm_start_model\(\)](#), [make_warm_start_cv_plan\(\)](#), [make_warm_start_prior\(\)](#), [pairwiseLLM_warm_model](#), [predict.pairwiseLLM_warm_algorithm_ensemble\(\)](#), [predict.pairwiseLLM_warm_ensemble\(\)](#), [prepare_warm_start_model\(\)](#), [register_warm_start_model\(\)](#), [save_warm_start_model\(\)](#), [summary.pairwiseLLM_warm_algorithm_ensemble\(\)](#), [summary.pairwiseLLM_warm_ensemble\(\)](#), [summary.pairwiseLLM_warm_predictions\(\)](#), [warm_start_coefficients\(\)](#), [warm_start_feature_schema\(\)](#), [warm_start_python_status\(\)](#)

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  local({
    # Synthetic features illustrate the interface, not predictive validity.
    example_features <- function(seed) {
      withr::local_seed(seed)
      fields <- warm_start_feature_schema()$feature
      x <- as.data.frame(matrix(runif(15 * length(fields)), nrow = 15))
      names(x) <- fields
      x$n_tokens <- 11:25
      x$token_length_mean <- 2 + 10 * x$token_length_mean
      x$token_length_std <- 0.2 + x$token_length_std
      x$dale_chall_readability_score <- 5 + 20 * x$dale_chall_readability_score
      x <- data.frame(item_id = as.character(1:15), x)
      attr(x, "warm_start_schema") <- "writing_features_v1"
      x
    }
    features <- example_features(3103)
    theta <- 10 + 0.4 * features$n_tokens - 2 * features$token_length_mean
    # A small alpha grid keeps this example fast; the default has 41 values.
    model <- fit_warm_start_model(features$item_id, theta, "synthetic-a",
      features = features, alpha_grid = c(0, 1))
    predictions <- predict(model, features)
    head(predictions)
  })
}
```

predict_link

Predict oriented hub-spoke comparison probabilities

Description

Predict oriented hub-spoke comparison probabilities

Usage

```
predict_link(result, pairs)
```

Arguments

result	A valid common result returned by fit_link() .
pairs	Data frame with observation_id, A_set, A_item, B_set, and B_item, using the identities in prepare_link_input() . No outcomes are accepted. A and B encode presentation orientation; both directions are legal.

Details

Conditional probabilities are $(1 - \epsilon) * \text{plogis}(\theta_A - \theta_B + \beta) + \epsilon / 2$. Reversing presentation is complementary only when β is zero. Prediction requires no refit or provider calls. E1 integrates over δ using the serialized posterior quadrature nodes. It refines numerical integration for a requested pair when necessary, without updating the posterior or modifying the result. There is no plug-in-mean prediction mode. Integration failure raises `pairwiseLLM_e1_numerical_error`, rather than substituting another prediction. E2 integrates the oriented Gaussian pair contrast under its joint Laplace approximation, including all item/offset cross-covariances. Deterministic adaptive quadrature over the standard Normal variable splits at zero and the logistic transition (bounded to plus/minus eight SDs to avoid excessively long finite intervals); both infinite tails are integrated. Local relative and absolute error budgets divide the requested global tolerances by the segment count and, successively, 4, 16, 64, and 256. The first valid attempt whose summed error estimate meets the original global absolute/relative tolerance is returned; exhaustion raises a numerical error. Refinement does not change saved controls or fit state. Degenerate zero-variance contrasts use the exact conditional probability. Repeated calls do not sample or mutate state. Integration failure raises `pairwiseLLM_e2_numerical_error`. E3 MAP uses the same integration with `pairwiseLLM_e3_numerical_error` on failure. E3-MCMC averages the conditional probability over its retained raw draws, without new sampling.

Value

A double vector of probabilities that presented A wins, aligned with the input rows. The estimator's prediction hook integrates its uncertainty; the dispatcher never silently substitutes probabilities at posterior means.

See Also

`prepare_link_input()`, `fit_link()`, `predict_link()`, `start_link_session()`

Other linking: `fit_link()`, `prepare_link_input()`, `save_link_session()`, `start_link_session()`, `summary.pairwiseLLM_link_result()`

<code>prepare_link_input</code>	<i>Prepare explicit evidence for a linking estimator</i>
---------------------------------	--

Description

These development interfaces define the common contract for E1–E3. They do not select pairs, contact providers, or depend on adaptive state. E1 is implemented with deterministic quadrature; E2 uses a Gaussian posterior bridge with MAP/Laplace inference. E3 jointly refits raw evidence using MAP/Laplace or an explicitly requested MCMC audit/reference engine.

Usage

```
prepare_link_input(
  estimator,
  hub,
```

```

    spoke,
    phase_a,
    cross,
    judge,
    control = list(),
    provenance = list()
)

```

Arguments

estimator	Required exact ID: fixed_shape_offset, gaussian_posterior_bridge, or joint_offset. There is no default.
hub, spoke	Lists with scalar set_id and an items data frame containing unique item_id and optional global_item_id. Local IDs may overlap between sets; supplied global IDs must be unique across both sets. Items are ordered hub first, then spoke, and by bitwise item ID within each set.
phase_a	Named list with hub and spoke entries. Each entry contains exactly one statistical payload: E1 points (named numeric vector), E2 draws (draws by named item columns), or E3 observations (table as below). Points and each draw are separately centered, with removed means recorded. Optional source metadata contains artifact_hash, reference_hash, evidence_hash, and n_observations, optional trait and orientation (higher_is_better); unavailable values remain typed missing. External source hashes are assertions of provenance, distinct from computed payload hashes. Use the source from prepare_linked_rubric_reference() for a frozen standalone hub; rubric prediction checks its identity and estimator payload. E1 also accepts list(artifact = artifact) in either set entry, mutually exclusive with points. Supply an in-memory canonical Phase A artifact (use readRDS() explicitly for files). Its set_id, fit_model_id, n_items, n_pairs_committed, and items are checked. Item-aligned items\$theta_raw_mean values are the EAP source; global IDs must match when supplied. Phase B summaries are rejected. The original artifact is hashed, and its declared within-set evidence hash/count are retained as provenance; raw outcomes, posterior draws, and marginal SDs are not used for E1 inference or retained in normalized input. Artifact source fields, if supplied, must agree with the extracted metadata. This extracts statistical inputs; it does not run adaptive Phase A quality/reliability gates. E2 also accepts list(artifact = artifact), mutually exclusive with draws. The same identity/model/scope checks apply, but the required statistical payload is posterior_draws, a finite matrix with at least two rows and named item columns. EAP summaries and marginal SDs cannot replace draws. E2 retains the centered draw matrix and source hashes/counts, without raw Phase A outcomes. Each set is Gaussianized in reduced centered coordinates using its sample mean and full sample covariance. E3 requires raw within-set observations for both sets. It retains all raw evidence and jointly re-estimates both centered shapes and the offset. Every observation enters once; points/draws cannot accompany raw evidence. E3 also accepts list(artifact = artifact) with exact phase_a_within_set_evidence (or historical within_set_evidence) rows. Historical rows use unique positive integer pair_id and step_id, item endpoints and binary y_A; judgment IDs

are generated from set and pair ID. Explicit observation tables use the common columns below. Recorded row counts and any declared raw-evidence hash must reconcile exactly. Artifact means, SDs and draws are ignored for inference. All three artifact adapters preserve exact artifact hashes, including historical compatible artifacts.

cross	Explicit active cross-set observations, including an empty table at zero budget. Evidence tables contain observation_id, A_set, A_item, B_set, B_item, and numeric binary y_A (one means A won). IDs identify judgments, not pairs, and must be unique across all raw evidence supplied. Repeated judgments of a pair use different IDs. All rows are used in supplied order; invalid rows are rejected rather than dropped. Phase A observations use the same columns and must be within the corresponding set.
judge	Named list with finite beta, epsilon in $[0, 1]$, model_variant (btl, btl_b, btl_e, or btl_e_b), link = "logit", and scalar source. Omitted model components must have value zero.
control	List with delta_prior = list(mean = 0, sd = 5), numerical estimator controls, and optional named initial free-coordinate vector. E1 accepts positive rel_tol = 1e-9, abs_tol = 1e-11, quantile_tol = 1e-8, and integer subdivisions = 1000L. The subdivision limit bounds the number of quadrature panels and CDF integration/root iterations. Tolerances apply to normalized mass and moments in prior-SD coordinates; quantile_tol is in delta units. Effective defaults are logged with every fit. E2 accepts positive rel_tol = 1e-10, gradient_tol = 1e-6, prediction_rel_tol = 1e-9, prediction_abs_tol = 1e-11, and positive integers maxit = 2000L and subdivisions = 1000L. maxit limits BFGS iterations per deterministic start; gradient_tol bounds the absolute gradient in whitened coordinates. Prediction tolerances apply to the integrated logistic probability, and subdivisions limits integration on each interval. E3 accepts the same numerical controls/defaults, with gradients measured in prior-whitened coordinates, and engine = "map_laplace" (default) or engine = "mcmc". Only explicit MCMC accepts a nested cmdstan list: chains = 4L, iter_warmup = 1000L, iter_sampling = 1000L, adapt_delta = .90, max_treedepth = 12L, optional seed and output_dir, and parallel_chains, threads_per_chain, core_fraction following <code>fit_bayes_btl_mcmc()</code> CPU-allocation policy. Missing seed uses CmdStan's random seed; supply a seed for reproducibility. All effective controls and resource allocation are recorded. MCMC uses optional cmdstanr and posterior; MAP/Laplace requires neither. Sampling repairs are separate explicit calls. Initial values are optimization hints only; the current engines ignore them to keep fresh and cumulative-evidence fits on the same numerical path. E3 shape priors are always independent standard Normals in reduced coordinates; Phase A posterior summaries cannot supply data-derived priors.
provenance	List with optional source_commit (package source revision; unknown is NA_character_) and expected, a named subset of the computed hashes and counts lists against which to reconcile inputs. Hashes use a versioned rlang::hash() scheme; the producing rlang version is recorded. Hashes produced before rlang 1.3.0 cannot be verified with its new algorithm. Validation reports this known incompatibility without changing saved hashes. See <code>save_link_session()</code> for recovery guidance.

Value

A versioned pairwiseLLM_link_input list with normalized identities, evidence, coordinates, controls, hashes, counts, and provenance. Raw Phase A likelihood counts are zero for E1/E2; source observation counts are separate.

See Also

`prepare_link_input()`, `fit_link()`, `predict_link()`, `start_link_session()`

Other linking: `fit_link()`, `predict_link()`, `save_link_session()`, `start_link_session()`, `summary.pairwiseLLM_link_result()`

Examples

```
hub <- list(set_id = "H", items = data.frame(item_id = c("h1", "h2")))
spoke <- list(set_id = "S", items = data.frame(item_id = c("s1", "s2")))
cross <- data.frame(observation_id = character(), A_set = character(),
  A_item = character(), B_set = character(), B_item = character(), y_A = integer())
input <- prepare_link_input("fixed_shape_offset", hub, spoke,
  phase_a = list(hub = list(points = c(h1 = -1, h2 = 1)),
    spoke = list(points = c(s1 = -.5, s2 = .5))), cross = cross,
  judge = list(beta = 0, epsilon = 0, model_variant = "btl",
    link = "logit", source = "frozen Phase A"))
input$counts
fit <- fit_link(input)
fit$offset
predict_link(fit, data.frame(observation_id = "held-out-1",
  A_set = "H", A_item = "h1", B_set = "S", B_item = "s2"))

# E2 carries the full joint Phase A draw information forward.
hub_draws <- cbind(h1 = c(-1, -2, -.5, -1.5), h2 = c(1, 2, .5, 1.5))
spoke_draws <- cbind(s1 = c(-.3, -.7, -.4, -.6), s2 = c(.3, .7, .4, .6))
bridge <- prepare_link_input("gaussian_posterior_bridge", hub, spoke,
  phase_a = list(hub = list(draws = hub_draws), spoke = list(draws = spoke_draws)),
  cross = cross, judge = input$judge)
bridge_fit <- fit_link(bridge)
bridge_fit$offset
predict_link(bridge_fit, data.frame(observation_id = "bridge-held-out",
  A_set = "H", A_item = "h1", B_set = "S", B_item = "s2"))

# E3: raw within-set outcomes, with the same explicit cross evidence.
within <- function(set, id, a, b) data.frame(observation_id = id, A_set = set,
  A_item = a, B_set = set, B_item = b, y_A = 1L)
joint <- prepare_link_input("joint_offset", hub, spoke,
  phase_a = list(hub = list(observations = within("H", "h1", "h1", "h2")),
    spoke = list(observations = within("S", "s1", "s1", "s2"))),
  cross = cross, judge = input$judge)
joint_fit <- fit_link(joint)
```

```
prepare_linked_rubric_reference
```

Save a standalone Bayesian ranking as a reusable rubric reference

Description

Prepare a completed reference ranking before scoring any new samples. This supports reference rankings fitted from pooled historical comparisons as well as a single collection of comparisons. The original scores and comparisons are saved together so later linking can check that it uses the same reference.

Usage

```
prepare_linked_rubric_reference(
  cj,
  evidence,
  set_id,
  items = NULL,
  trait = NULL,
  provenance = list()
)
```

Arguments

<code>cj</code>	A completed <code>fit_bayes_btl_mcmc()</code> result. With several refits, the last refit is used. It must record evidence identity at fit time. Older results must be refitted from their original comparisons first; they can still be used for ordinary same-set rubric calibration.
<code>evidence</code>	The exact comparison rows used for that refit, in the same results-table format supplied to <code>fit_bayes_btl_mcmc()</code> . For a subset fit, supply only that subset. See <code>build_btl_results_data()</code> .
<code>set_id</code>	A single reference-set identifier, as in <code>prepare_link_input()</code> .
<code>items</code>	Optional data frame with <code>item_id</code> and <code>global_item_id</code> , mapping every fitted item to a unique ID across sets. When omitted, fitted item IDs are also used as global IDs.
<code>trait</code>	The trait being ranked, such as "organization". Required when the completed result does not already record it.
<code>provenance</code>	Optional named list of serializable study/source metadata, such as <code>source_commit</code> . Stored separately from computed identity fields.

Details

The supplied comparisons must match the evidence recorded when the fit was made. Compatible item names alone are insufficient. Changed comparisons require a new fit and a new reference. Save the result with `saveRDS()` and restore it with `readRDS()`; no live sampler is needed for later

prediction. Hash-engine provenance is recorded for new references. Older references without it still require every identity hash to verify. Hashes from before rlang 1.3.0 use a different algorithm; see [save_link_session\(\)](#) for error and recovery behavior. A completed fit with unverifiable fit-time evidence identity cannot be reused automatically to rebuild a reference.

This constructor supports standalone fits with shared judge parameters. Import-ready adaptive Phase A artifacts continue to work directly with [fit_rubric_calibration\(\)](#). Neither route changes the original reference scores to match the distribution of a new set.

Value

A serializable `pairwiseLLM_linked_rubric_reference`. Supply it to [fit_rubric_calibration\(\)](#) with `calibration_design = "linked_anchors"`. `items` holds global IDs, source IDs, frozen scores and SDs. For linking, `hub` supplies the set/item mapping; `points`, `posterior_draws`, and `evidence` supply the E1, E2, and E3 hub inputs, respectively. Attach `source` to each input to carry the same computed `reference_hash`. `judge` contains the shared bias/lapse settings from the accepted fit. Diagnostics and reliability are retained; failed diagnostics warn as in ordinary calibration.

See Also

[prepare_link_input\(\)](#), [fit_rubric_calibration\(\)](#)
Other rubric calibration: [evaluate_rubric_predictions\(\)](#), [fit_rubric_calibration\(\)](#), [predict.pairwiseLLM_rubric](#)

<code>prepare_warm_start_model</code>	<i>Prepare metadata or a summary-only warm-start artifact</i>
---------------------------------------	---

Description

Prepare metadata or a summary-only warm-start artifact

Usage

```
prepare_warm_start_model(model, metadata = list(), omit_audit = FALSE)
```

Arguments

<code>model</code>	A valid pairwiseLLM_warm_model , ensemble_warm_start_models() ensemble, or ensemble_warm_start_algorithms() ensemble.
<code>metadata</code>	Named list with optional scalar character name, version, domain, notes, license, <code>prepared_at</code> , <code>preparation_package_version</code> , and <code>extraction_provenance</code> (a named character vector). Supplied fields replace existing fields. Timestamps use UTC YYYY-MM-DDTHH:MM:SSZ strings.
<code>omit_audit</code>	Explicitly omit row-level development evidence. Default FALSE.

Details

Preparation adds a preparation timestamp/package version when absent, not a training date. Extraction provenance defaults to `c(status = "unavailable")`; user-supplied records are not independently verified. Schema identity or a later Python status check is not extraction provenance.

Ordinary preparation preserves audit records. Explicit omission creates format 2 for legacy models; format-3 models remain format 3. Both use summary-only audit status, retaining deployment parameters, tuning settings and nested-validation summaries. IDs, OOF rows, fold records, tuning traces and contextual warning messages are omitted; warning counts remain. Format-3 artifacts retain compact CV identity digests, not the original plan. Summaries cannot be recomputed without the original evidence. An already reduced artifact cannot recover its audit through this function. SVR support vectors retain numeric matrix dimensions and feature order as necessary deployment data; audit omission does not anonymize them.

Ensembles retain ensemble format 1; audit omission recursively reduces each component, preserving its format generation and existing metadata. Supplied preparation metadata applies to the ensemble only. Same-task algorithm ensembles also retain their common CV identity and honest ensemble validation summaries, dropping ensemble row evidence and marking the artifact summary-only.

No raw texts are added. Review task labels, notes, domain, and provenance for restricted information before bundling; this is not a general anonymizer. User models do not need complete publication metadata. Task-specific outcome scales remain standardized independently; storage does not link BTL scales.

Value

A prepared copy using the same model class and prediction method.

See Also

`save_warm_start_model()`, `register_warm_start_model()`

Other adaptive warm start: `ensemble_warm_start_algorithms()`, `ensemble_warm_start_models()`, `extract_warm_start_features()`, `fit_warm_start_model()`, `make_warm_start_cv_plan()`, `make_warm_start_prior()`, `pairwiseLLM_warm_model`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_ensemble()`, `predict.pairwiseLLM_warm_model()`, `register_warm_start_model()`, `save_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_ensemble()`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_coefficients()`, `warm_start_feature_schema()`, `warm_start_python_status()`

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  local({
    # Synthetic features illustrate the interface, not predictive validity.
    example_features <- function(seed) {
      withr::local_seed(seed)
      fields <- warm_start_feature_schema()$feature
      x <- as.data.frame(matrix(runif(15 * length(fields)), nrow = 15))
      names(x) <- fields
    }
  })
}
```

```

x$n_tokens <- 11:25
x$token_length_mean <- 2 + 10 * x$token_length_mean
x$token_length_std <- 0.2 + x$token_length_std
x$dale_chall_readability_score <- 5 + 20 * x$dale_chall_readability_score
x <- data.frame(item_id = as.character(1:15), x)
attr(x, "warm_start_schema") <- "writing_features_v1"
x
}
features <- example_features(3103)
theta <- 10 + 0.4 * features$n_tokens - 2 * features$token_length_mean
# A small alpha grid keeps this example fast; the default has 41 values.
model <- fit_warm_start_model(features$item_id, theta, "synthetic-a",
  features = features, alpha_grid = c(0, 1))
deployment <- prepare_warm_start_model(model,
  metadata = list(name = "example", domain = "Synthetic demonstration"),
  omit_audit = TRUE)
predict(deployment, features)
})
}

```

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

randomize_pair_order *Randomly assign samples to positions SAMPLE_1 and SAMPLE_2*

Description

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

Usage

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

Arguments

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

Details

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

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

Value

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

See Also

[alternate_pair_order](#) for deterministic alternating order, [sample_reverse_pairs](#) and [compute_reverse_consistency](#) for reverse-order checks.

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.

Details

Use one row per writing sample and keep IDs stable across collection and analysis. Map spreadsheet column names with `id_col` and `text_col`. For a CSV, first use `read.csv("writing.csv", colClasses = "character")` to preserve identifiers such as "001". Inspect blank/missing IDs and texts yourself; this reader checks duplicates, not all data-quality problems. See the [getting started guide](#) for a complete example.

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)
```

`read_samples_dir`

Read writing samples from a directory of .txt files

Description

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

Usage

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

Arguments

<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 <code>.txt</code> .

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")
```

```
register_warm_start_model
```

Register, inspect, or remove user warm-start models

Description

Register, inspect, or remove user warm-start models

Usage

```
register_warm_start_model(model, name, overwrite = FALSE)
```

```
remove_warm_start_model(name)
```

```
list_warm_start_models(source = c("all", "user", "bundled"))
```

Arguments

model	A valid <code>pairwiseLLM_warm_model</code> , <code>ensemble_warm_start_models()</code> ensemble, or <code>ensemble_warm_start_algorithms()</code> ensemble.
name	Model registry name, separate from the assessment task ID.
overwrite	Explicitly replace an existing user entry. Default FALSE.
source	Which registries to list.

Details

User models live in the `models` subdirectory of `tools::R_user_dir("pairwiseLLM", "data")`. Only explicit registration creates this directory. Names are trimmed, ASCII-lowercased, and spaces/underscores become hyphens. The result must contain alphanumeric segments separated by single hyphens. Dots, path separators, traversal, and escaping symlinks are rejected. The normalized name determines collisions and the `<name>.rds` file.

Installed bundled models are read-only `models/<name>.rds` resources. Both sources use the same model validator and prediction method. Listing reads and validates artifacts without a fitting backend or Python; corrupt entries produce errors naming their paths. Missing registries return empty results and are not created. Bundled lookup/listing additionally require manifest version 1, matching file inventory, MD5 checksum, size, and artifact metadata. Checksums detect changes; they do not authenticate publishers. User artifacts do not require a manifest.

Registration preserves full audit evidence unless explicitly reduced beforehand. Compressed files replace entries only with explicit overwrite; no backup history accumulates. Use listing and removal to manage obsolete user models. Models are not automatically removed based on age. Removal never affects bundled models. Tests and examples must redirect `R_USER_DATA_DIR` to a temporary directory. No user models are written into the installed package tree.

Value

Registration/removal invisibly return the entry path. Listing returns a tibble with name, source, path, version, format_version, schema, target, n, calibration, audit_status, size_bytes, metadata, and validation. Metadata and validation are list columns; unspecified metadata versions are NA character values. Additional columns artifact_type and component_count distinguish ensembles. engine, engine_version, component_engines, and component_engine_versions identify fitting algorithms and backend versions. Cross-task ensemble n is NA (no pooled sample size), calibration is component_oof_linear, and audit status is full, summary_only, or mixed. Ensemble validation contains named component metrics, not ensemble-performance estimates. Same-task algorithm rows have artifact_type = "algorithm_ensemble", the common training n, and honest ensemble outer-validation metrics. Their audit is full or summary_only.

See Also

`save_warm_start_model()`, `load_warm_start_model()`, `ensemble_warm_start_models()`

Other adaptive warm start: `ensemble_warm_start_algorithms()`, `ensemble_warm_start_models()`, `extract_warm_start_features()`, `fit_warm_start_model()`, `make_warm_start_cv_plan()`, `make_warm_start_prior()`, `pairwiseLLM_warm_model`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_ensemble()`, `predict.pairwiseLLM_warm_model()`, `prepare_warm_start_model()`, `save_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_ensemble()`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_coefficients()`, `warm_start_feature_schema()`, `warm_start_python_status()`

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  local({
    # Synthetic features illustrate the interface, not predictive validity.
    example_features <- function(seed) {
      withr::local_seed(seed)
      fields <- warm_start_feature_schema()$feature
      x <- as.data.frame(matrix(runif(15 * length(fields)), nrow = 15))
      names(x) <- fields
      x$n_tokens <- 11:25
      x$token_length_mean <- 2 + 10 * x$token_length_mean
      x$token_length_std <- 0.2 + x$token_length_std
      x$dale_chall_readability_score <- 5 + 20 * x$dale_chall_readability_score
      x <- data.frame(item_id = as.character(1:15), x)
      attr(x, "warm_start_schema") <- "writing_features_v1"
      x
    }
    features <- example_features(3103)
    theta <- 10 + 0.4 * features$n_tokens - 2 * features$token_length_mean
    # A small alpha grid keeps this example fast; the default has 41 values.
    model <- fit_warm_start_model(features$item_id, theta, "synthetic-a",
      features = features, alpha_grid = c(0, 1))
    registry <- withr::local_tempdir()
    withr::local_envvar(c(R_USER_DATA_DIR = registry))
    register_warm_start_model(model, name = "example")
  })
}
```

```
list_warm_start_models(source = "user")
restored <- load_warm_start_model(name = "example", source = "user")
remove_warm_start_model("example")
})
}
```

`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

pairs	Tibble or data frame with at least columns ID1, text1, ID2, text2.
model	Anthropic model name (for example "claude-sonnet-4-5").
trait_name	Trait name to pass to build_anthropic_batch_requests .
trait_description	Trait description to pass to build_anthropic_batch_requests .
prompt_template	Prompt template string, typically from set_prompt_template .
reasoning	Character scalar; one of "none" or "enabled". See details above for how include_thoughts influences this value and how temperature defaults are derived.
include_thoughts	Logical; if TRUE, requests extended thinking from Claude (by setting reasoning = "enabled" when necessary) and parses any thinking blocks into a thoughts column in the batch results.

batch_input_path	Path to write the JSON file containing the requests object. Defaults to a temporary file with suffix ".json".
batch_output_path	Path to write the downloaded .jsonl results if poll = TRUE. Defaults to a temporary file with suffix ".jsonl".
poll	Logical; if TRUE, the function will poll the batch until it reaches processing_status = "ended" using anthropic_poll_batch_until_complete and then download and parse the output. If FALSE, it stops after creating the batch and returns without polling or parsing.
interval_seconds	Polling interval in seconds (used when poll = TRUE).
timeout_seconds	Maximum total time in seconds for polling before giving up (used when poll = TRUE).
api_key	Optional Anthropic API key. Defaults to Sys.getenv("ANTHROPIC_API_KEY").
anthropic_version	Anthropic API version string passed as the anthropic-version HTTP header. Defaults to "2023-06-01".
verbose	Logical; if TRUE, prints progress messages while polling.
...	Additional Anthropic parameters forwarded to build_anthropic_batch_requests (for example max_tokens, temperature, top_p, thinking_budget_tokens).

Details

1. Builds Anthropic batch requests from a tibble of pairs using [build_anthropic_batch_requests](#).
2. Writes a JSON file containing the requests object for reproducibility.
3. Creates a Message Batch via [anthropic_create_batch](#).
4. Optionally polls until the batch reaches processing_status = "ended" using [anthropic_poll_batch_until_complete](#).
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 $1024 \leq \text{thinking_budget_tokens} < \text{max_tokens}$. 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_errors\(\)](#), [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> . <ul style="list-style-type: none"> • 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.
<code>verbose</code>	Logical; if TRUE, prints progress messages.
<code>include_thoughts</code>	Logical; if TRUE, sets <code>thinkingConfig.includeThoughts = TRUE</code> in each request, mirroring <code>gemini_compare_pair_live()</code> . Parsed results will include a <code>thoughts</code> column when visible thoughts are returned by the API (currently batch typically only exposes <code>thoughtSignature + thoughtsTokenCount</code>).
<code>service_tier</code>	Gemini Developer API service tier forwarded to <code>build_gemini_batch_requests</code> . Use "standard" (default) or NULL for provider default behavior, or "flex" / "priority" to request the documented Gemini service tier.
<code>...</code>	Additional arguments forwarded to <code>build_gemini_batch_requests</code> (for example <code>temperature</code> , <code>top_p</code> , <code>top_k</code> , <code>max_output_tokens</code> , <code>store</code>). <code>store = TRUE</code> or <code>FALSE</code> controls logging for every batch request; omission or NULL preserves the provider/project logging default.

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_errors()`, `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 <code>make_pairs()</code> , <code>sample_pairs()</code> , and <code>randomize_pair_order()</code> .
<code>model</code>	OpenAI model name (e.g. "gpt-4.1", "gpt-5.6-sol").
<code>trait_name</code>	Trait name to pass to <code>build_openai_batch_requests()</code> .
<code>trait_description</code>	Trait description to pass to <code>build_openai_batch_requests()</code> .
<code>prompt_template</code>	Prompt template string, typically from <code>set_prompt_template()</code> .
<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, include_thoughts = TRUE causes the responses endpoint to be selected automatically.
<code>include_raw</code>	Logical; if TRUE, attaches the raw model response as a list-column raw_response in the parsed results.
<code>endpoint</code>	One of "chat.completions" or "responses". If NULL (or omitted), it is chosen automatically as described above.
<code>batch_input_path</code>	Path to write the batch input .jsonl file. Defaults to a temporary file.
<code>batch_output_path</code>	Path to write the batch output .jsonl file if poll = TRUE. Defaults to a temporary file.
<code>poll</code>	Logical; if TRUE, 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 FALSE, it stops after creating the batch and returns without polling or parsing.
<code>interval_seconds</code>	Polling interval in seconds (used when poll = TRUE).
<code>timeout_seconds</code>	Maximum total time in seconds for polling before giving up (used when poll = TRUE).
<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. temperature, top_p, logprobs, reasoning, store, prompt_caching, and max_output_tokens. Explicitly select endpoint = "responses" when supplying max_output_tokens.

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.

Omitted or NULL store preserves provider defaults. Responses are stored for later retrieval by default; `store = FALSE` disables that response storage, not Batch input/output/error file retention or all data retention. See `build_openai_batch_requests()` for endpoint-specific details. Use `openai_download_batch_errors()` separately to retrieve request failures. For an errors-only batch, use `poll = FALSE`, then `openai_get_batch()` or `openai_poll_batch_until_complete()` and the error downloader; this pipeline's `poll = TRUE` path still requires an output file.

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.

Batch prompt caching

The default disables prompt caching for these verified model IDs: `gpt-5.6-luna`, `gpt-5.6-terra`, `gpt-5.6-sol`, `gpt-6-luna`, `gpt-6-sol`, `gpt-6-astra`, and `gpt-6.1-sol`. Each request body gets `prompt_cache_options = list(mode = "explicit")`, with no breakpoints, on both endpoints. Set `prompt_caching = "implicit"` to omit that field and retain provider caching when prefix reuse is expected to be worthwhile.

Recognized earlier OpenAI models retain their historical request bodies when the policy is omitted or NULL; explicitly disabling caching on those models errors. Unknown IDs (including unlisted snapshots, future models and custom aliases) require an explicit "implicit" opt-in or a verified model ID. Support is an offline allowlist; version numbers and snapshot suffixes never establish support for disabling caching. Older GPT-3.5/4, GPT-5 through GPT-5.5 and o-series naming forms, including recognized legacy date-suffixed names, retain provider defaults.

Invalid policies and manual cache controls fail before submission, even with zero pairs. Raw `prompt_cache_options`, `prompt_cache_breakpoint`, `prompt_cache_key`, retention and TTL controls are not accepted by these builders. Keys affect cache routing/accounting; retention and TTL affect lifetime; `store = FALSE` affects response storage. None substitutes for disabling implicit cache breakpoints. Prompt text and keys are not rewritten. This policy affects future package-built Batch requests only, not live requests or already submitted jobs. See the [prompt-caching guide](#) and vignette("advanced-batch-workflows") for usage-based cost comparisons.

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_errors()`, `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.

Details

A random subset does not guarantee that every item participates or that all items are connected through comparisons. Inspect the design before using the collected outcomes to rank a complete set.

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 <code>round(nrow(pairs) * reverse_pct)</code> .
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 or an explicit-evidence linking session. Linking sessions are saved exactly in link-session.rds using save_link_session() .
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("pwillm-session-")
state <- adaptive_rank_start(c("a", "b", "c"), seed = 1)
save_adaptive_session(state, dir, overwrite = TRUE)
```

save_link_session	<i>Save and load exact linking sessions</i>
-------------------	---

Description

RDS preserves exact types, item/evidence order, hashes, covariance, numerical modes and prediction data, including E3-MCMC draws. Reading never refits or migrates a posterior. Only open trusted RDS files. Unknown external Phase A provenance remains missing; computed payload hashes are always present.

Usage

```
save_link_session(state, path, overwrite = FALSE)
```

```
load_link_session(path, input = NULL)
```

Arguments

state	A start_link_session() result.
path	File path for an RDS session.
overwrite	Whether to replace an existing file.
input	Optional expected prepared input (or list of inputs) on load. It must match the saved input exactly. Use resume_link_session() to append new evidence after this identity check.

Details

Hashes are verified using the current rlang engine. A failed check with recorded link-v1-rlang provenance from before rlang 1.3.0 raises `pairwiseLLM_link_hash_incompatibility` (also a linking-contract error), with `stored_hash_scheme`, `stored_hash_engine_version`, and `current_hash_engine_version` condition fields. Other mismatches remain integrity/contract errors; a different version alone is not proof of incompatibility. Missing engine provenance cannot establish the cause.

Keep the original artifact and hashes. Explicitly prepare new inputs from the original saved judgments and compatible Phase A points/draws using [prepare_link_input\(\)](#). This preparation needs no new LLM judgments or sampling. A new standalone reference can reuse a saved completed fit through [prepare_linked_rubric_reference\(\)](#) only if its fit-time evidence identity verifies. Otherwise investigate in its producing environment before deciding separately whether a refit is needed. Creating a new E3 result itself refits the saved outcomes; MCMC remains an explicit, potentially expensive choice. Loading/validation never rebuilds, migrates hashes, overwrites artifacts, contacts providers, or starts sampling.

Value

Saving invisibly returns path; loading returns the identical session.

See Also

[prepare_link_input\(\)](#), [fit_link\(\)](#), [start_link_session\(\)](#), [save_link_session\(\)](#)

Other linking: [fit_link\(\)](#), [predict_link\(\)](#), [prepare_link_input\(\)](#), [start_link_session\(\)](#), [summary.pairwiseLLM_link_result\(\)](#)

save_warm_start_model *Save or load a portable warm-start model*

Description

Save or load a portable warm-start model

Usage

```
save_warm_start_model(model, path, overwrite = FALSE)

load_warm_start_model(
  path = NULL,
  name = NULL,
  source = c("auto", "user", "bundled")
)
```

Arguments

model	A valid pairwiseLLM_warm_model , ensemble_warm_start_models() ensemble, or ensemble_warm_start_algorithms() ensemble.
path	Explicit file path. Its parent must already exist when saving.
overwrite	Allow replacement of an existing artifact. Default FALSE.
name	Registered model name, mutually exclusive with path.
source	Registry to search. auto errors if user and bundled names collide.

Details

Artifacts are compressed RDS objects, without an envelope or serialized backend fit. Legacy formats 1 (full audit) and 2 (explicit summary-only), and format 3 (explicit full or summary-only audit status), are supported independently of package version. Cross-task ensembles use their own format 1 and may contain any supported single-model format. Same-task algorithm ensembles have a distinct format-1 contract containing only format-3 components with matching audit status. Use [prepare_warm_start_model\(\)](#) to add metadata or explicitly omit audit records before saving. Saving never strips records or adds timestamps. Neither loading nor prediction from precomputed features needs a fitting backend or Python.

Exactly one of path or name is required for loading. Positional input means a path; a missing file never falls back to a registry search. Explicit paths require source = "auto". Registry names follow [register_warm_start_model\(\)](#). Named bundled lookup verifies the installed manifest and

artifact checksum. Explicit file paths use ordinary artifact validation without a manifest. Load only trusted RDS files; contract validation is not a serialization sandbox.

Writes are staged in the destination directory and validated before publishing. Failed writes clean up staging files. Replacement uses filesystem rename; if the platform cannot replace an existing file this way, the operation fails and leaves that file intact. No persistent backup history is created.

Value

Saving invisibly returns the normalized destination path. Loading returns the validated model, unchanged from its serialized representation.

See Also

`prepare_warm_start_model()`, `register_warm_start_model()`, `list_warm_start_models()`

Other adaptive warm start: `ensemble_warm_start_algorithms()`, `ensemble_warm_start_models()`, `extract_warm_start_features()`, `fit_warm_start_model()`, `make_warm_start_cv_plan()`, `make_warm_start_prior()`, `pairwiseLLM_warm_model`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_ensemble()`, `predict.pairwiseLLM_warm_model()`, `prepare_warm_start_model()`, `register_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_model`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_coefficients()`, `warm_start_feature_schema()`, `warm_start_python_status()`

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  local({
    # Synthetic features illustrate the interface, not predictive validity.
    example_features <- function(seed) {
      withr::local_seed(seed)
      fields <- warm_start_feature_schema()$feature
      x <- as.data.frame(matrix(runif(15 * length(fields)), nrow = 15))
      names(x) <- fields
      x$n_tokens <- 11:25
      x$token_length_mean <- 2 + 10 * x$token_length_mean
      x$token_length_std <- 0.2 + x$token_length_std
      x$dale_chall_readability_score <- 5 + 20 * x$dale_chall_readability_score
      x <- data.frame(item_id = as.character(1:15), x)
      attr(x, "warm_start_schema") <- "writing_features_v1"
      x
    }
    features <- example_features(3103)
    theta <- 10 + 0.4 * features$n_tokens - 2 * features$token_length_mean
    # A small alpha grid keeps this example fast; the default has 41 values.
    model <- fit_warm_start_model(features$item_id, theta, "synthetic-a",
      features = features, alpha_grid = c(0, 1))
    path <- tempfile(fileext = ".rds")
    on.exit(unlink(path), add = TRUE)
    save_warm_start_model(model, path = path)
    restored <- load_warm_start_model(path = path)
    predict(restored, features)
```

```
    })
  }
```

scale_separation_reliability

Calculate scale-separation reliability

Description

Calculate conventional SSR as $1 - \text{mean}(\text{se}^2) / \text{stats::var}(\text{theta})$ and expose its variance components. The observed variance uses the sample denominator $n - 1$. Estimates and SEs must use the same scale and item order.

Usage

```
scale_separation_reliability(theta, se)
```

Arguments

theta	Real numeric vector of finite item estimates, with at least two entries and positive sample variance.
se	Real numeric vector of finite, nonnegative standard errors, in the same order and of the same length as theta. Names are not used to align the vectors.

Details

Negative SSR and negative estimated true-score variance are retained, not clipped. They indicate that mean squared uncertainty exceeds observed score variance. SSR depends on the estimator, its SE convention, and the estimated score variance; it is not an estimator-free measure of recovery or accuracy.

Value

A list with observed_variance, mean_squared_se, true_score_variance (observed variance minus mean squared SE), ssr, n_items, n_finite, valid, and status. Successful calculations have valid = TRUE and status "ok" or "negative_true_score_variance". Invalid inputs or non-finite calculated components raise an error; no items are dropped and no coefficient is returned for an undefined calculation.

See Also

[fit_bt_model\(\)](#), [integrated CJ workflow](#)

Other frequentist models: [bootstrap_bt_model\(\)](#), [build_bt_data\(\)](#), [build_elo_data\(\)](#), [fit_bt_model\(\)](#), [fit_elo_model\(\)](#), [predict.pairwiseLLM_bt_alpha\(\)](#), [predict.pairwiseLLM_bt_firth\(\)](#), [predict.pairwiseLLM_bt_summarize_bt_fit\(\)](#)

Examples

```
scale_separation_reliability(c(-1, 0, 1), c(0.2, 0.3, 0.4))
```

set_prompt_template	<i>Read or validate 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

Save the returned text and pass it as prompt_template to a comparison function. Despite its name, this function does not change a global active template. Keep custom template files in your own project, rather than editing the installed package. Use [register_prompt_template\(\)](#) and [get_prompt_template\(\)](#) to manage named templates for the current session.

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>
"
```

```
tmp12 <- set_prompt_template(template = custom)
cat(substr(tmp12, 1, 120), "...\\n")
```

start_link_session	<i>Run and resume explicit-evidence linking sessions</i>
--------------------	--

Description

Sessions retain common E1–E3 results under `linking$estimator`, including estimator ID/version, per-spoke continuation, diagnostics and exact provenance. Pair selection and stopping remain external. No estimator is selected by default. Adaptive Phase B D-optimal execution is unavailable for all estimators pending a separate selector validation study. Posterior covariance does not establish selector validity. Use these explicit-evidence sessions for E1–E3 comparisons; E3-MCMC remains an explicitly requested audit/reference engine.

Usage

```
start_link_session(input, status = "active")

resume_link_session(state, input, status = NULL)
```

Arguments

input	A prepare_link_input() object, or (for <code>start_link_session</code>) a list of such objects with distinct spokes and identical hub evidence/judge.
status	Explicit reporting status: active, probe, or frozen. This labels controller status only: all observations in <code>input\$cross</code> remain active estimation evidence. Held-out probe outcomes must not enter that table.
state	A linking session from <code>start_link_session()</code> or <code>load_link_session()</code> .

Details

Resume requires the same estimator, version, item order, Phase A, judge, numerical configuration and unchanged old cross-evidence prefix. New rows may only be appended. Passing the identical input is an exact no-op (including MCMC); extended evidence refits using the saved numerical mode, never the previous posterior as a new prior. A frozen spoke cannot append evidence until the caller explicitly sets `status = "active"`.

With multiple spokes, each estimator fits its own hub/spoke pair. E2/E3 hub posteriors can differ across spokes. Item summaries retain `link_spoke_id` and do not average or silently select one hub posterior. Ranks are per fit.

Value

A `pairwiseLLM_link_session` with exact data-only results and a stage log.

See Also

`prepare_link_input()`, `fit_link()`, `start_link_session()`, `save_link_session()`

Other linking: `fit_link()`, `predict_link()`, `prepare_link_input()`, `save_link_session()`, `summary.pairwiseLLM_link_result()`

`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

<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>	Anthropic model name (for example "claude-sonnet-4-5", "claude-haiku-4-5", or "claude-opus-4-5").
<code>trait_name</code>	Trait name to pass to anthropic_compare_pair_live .
<code>trait_description</code>	Trait description to pass to anthropic_compare_pair_live .
<code>prompt_template</code>	Prompt template string, typically from set_prompt_template .
<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 anthropic-version HTTP header. Defaults to "2023-06-01".
<code>reasoning</code>	Character scalar passed to anthropic_compare_pair_live (one of "none" or "enabled").
<code>verbose</code>	Logical; if TRUE, prints status, timing, and result summaries.
<code>status_every</code>	Integer; print status / timing for every status_every-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 raw_response list-column with the parsed JSON body from Anthropic. Note: Raw responses are not saved to the incremental CSV file.
<code>include_thoughts</code>	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.
<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 readr package.
<code>parallel</code>	Logical; if TRUE, enables parallel processing using <code>future.apply</code> . Requires the future and future.apply packages.
<code>workers</code>	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.
<code>...</code>	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_typesafe_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [typesafe_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 <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: Use no more than 2 workers to avoid HTTP 429 errors and respect shared check-farm resources.
...	Additional parameters passed through to <code>gemini_compare_pair_live()</code> . store = TRUE or FALSE controls request logging for every submitted pair; omission or NULL preserves the provider/project logging default. thinking_budget is ignored by the single-pair helper.

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_typesafe_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `typesafe_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

*Backend-agnostic live comparisons for a tibble of pairs***Description**

submit_llm_pairs() is a backend-neutral wrapper around row-wise comparison for multiple pairs. It takes a tibble of pairs (ID1, text1, ID2, text2), submits each pair to the selected backend, and 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",
    "typesafe"),
  endpoint = c("chat.completions", "responses", "decisions"),
  api_key = NULL,
  verbose = TRUE,
  status_every = 1,
  progress = TRUE,
  include_raw = FALSE,
  save_path = NULL,
  parallel = FALSE,
  workers = 1,
  execution_control = list(),
  ...
)

```

Arguments

pairs	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> .
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 <code>set_prompt_template()</code> .
backend	Character scalar indicating which LLM provider to use. One of "openai", "anthropic", "gemini", "vertex", "together", "ollama", or "typesafe" (Jev decisions).
endpoint	Character scalar specifying which endpoint family to use for backends that support multiple live APIs. For the "openai" backend this must be "chat.completions", "responses", or "decisions", matching <code>submit_openai_pairs_live()</code> . For "anthropic", "gemini", "vertex", "together", and "ollama", this is currently ignored. For TypeSafe, omit it or use "systemone".
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> for generative backends, requiring future and future.apply . Decision concurrency uses optional promises/later.
workers	Integer; the number of parallel workers (threads) to use if parallel = TRUE. Defaults to 1.

`execution_control`

Named list of decision-only scheduling, retry, timeout and lock controls; defaults to `list()`. See [decision_bulk](#) for all defaults, persistent budgets, probability storage and recovery guarantees.

...

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 both OpenAI live endpoints, `service_tier = "standard"` or `"default"` sends `"default"`; `"auto"`, `"flex"`, and `"priority"` are sent unchanged. Omission or `NULL` allows OpenAI's project-configured `"auto"` behavior. Model/tier availability is validated by OpenAI. 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.

For OpenAI and Gemini Developer API, `store` accepts `NULL`, `TRUE`, or `FALSE`. It controls OpenAI response storage or Gemini request logging; omission or `NULL` preserves provider/project defaults. See the selected provider helper for details; this is not a cross-provider retention policy.

Details

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

Start with `parallel = FALSE` and a small collection example. Cloud calls transmit sample text to the chosen provider and may incur charges. Inspect all three returned components before modeling; a successful retry can leave an entry in `failed_attempts` even when the pair has a valid result. Use a separate `save_path` for each study/model/trait configuration. See [Getting Started](#) and the [recovery guide](#).

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()`
- "typesafe" → `submit_typesafe_pairs_live()`

For OpenAI Decisions and TypeSafe Jev, omitted `prompt_template` selects the frozen decision instructions. Explicit generative templates fail. See [decision_bulk](#) for concurrent live execution, occurrence IDs, strict resume identity, atomic RDS checkpoints, CSV exports, raw-response privacy and the one-time automatic ambiguous reissue. Concurrent decision execution uses optional promises/late, with workers bounding in-flight requests; sequential decision execution requires neither. Other backends retain their existing future-based execution and CSV resumption behavior.

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_typesafe_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [typesafe_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()

# Small sequential collection example (paid requests, not a full ranking design)
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       = FALSE,
  save_path      = "results_openai.csv"
)

# Inspect successes, unresolved pairs, and failed attempts separately.
res_live$results
res_live$failed_pairs
```

```

res_live$failed_attempts
# After inspecting failures and collecting an adequate comparison design:
bt_data <- build_bt_data(res_live$results)

# Optional: parallel = TRUE, workers = 2 needs future and future.apply.

# Live comparisons using a local Ollama backend with incremental saving
res_ollama <- submit_llm_pairs(
  pairs          = pairs,
  model          = "mistral-small3.2:24b",
  trait_name     = td$name,
  trait_description = td$description,
  prompt_template = tmpl,
  backend        = "ollama",
  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 make_pairs() , sample_pairs() , and randomize_pair_order() .
<code>model</code>	Ollama model name (for example "mistral-small13.2:24b", "qwen3:32b", "gemma3:27b").
<code>trait_name</code>	Trait name to pass to ollama_compare_pair_live() .
<code>trait_description</code>	Trait description to pass to ollama_compare_pair_live() .
<code>prompt_template</code>	Prompt template string, typically from set_prompt_template() .
<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 ollama_compare_pair_live() for behavior.
<code>num_ctx</code>	Integer; context window to use via <code>options\$num_ctx</code> . The default is 8192L.
<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 Ollama. 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. Defaults to FALSE.
<code>workers</code>	Integer; the number of parallel workers (threads) to use if <code>parallel = TRUE</code> . Defaults to 1.
<code>...</code>	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_typesafe_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `typesafe_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-small3.2:24b",
  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", "decisions"),
```

```

    api_key = NULL,
    verbose = TRUE,
    status_every = 1,
    progress = TRUE,
    include_raw = FALSE,
    save_path = NULL,
    parallel = FALSE,
    workers = 1,
    execution_control = list(),
    ...
  )

```

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.
<code>workers</code>	Integer; the number of parallel workers (threads) to use if <code>parallel = TRUE</code> . 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.
<code>execution_control</code>	Named list used only with <code>endpoint = "decisions"</code> ; see decision_bulk for controls, defaults, durable budgets and recovery.

... Additional OpenAI parameters (temperature, top_p, logprobs, reasoning, service_tier, store, and so on) passed on to `openai_compare_pair_live`. `store` accepts `NULL`, `TRUE`, or `FALSE` for either endpoint and is forwarded to every submitted pair. Omission or `NULL` preserves the OpenAI default. For either endpoint, `service_tier = "standard"` or `"default"` sends `"default"`; `"auto"`, `"flex"`, and `"priority"` are sent unchanged. Omission or `NULL` allows OpenAI's project-configured `"auto"` behavior. Model/tier availability is validated by OpenAI.

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.

With `endpoint = "decisions"`, omitted `prompt_template` selects the frozen decision prompt. Concurrent execution uses optional `promises/late`; `workers` bounds in-flight requests. Saved decisions require filelock and an atomic RDS journal alongside the CSV export. See [decision_bulk](#) for complete scientific identity, probability storage, raw-response privacy and recovery contracts. Historical generative execution and resumption are unchanged.

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_typesafe_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [typesafe_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

<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>	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.
<code>trait_name</code>	Trait name to pass to <code>together_compare_pair_live()</code> .
<code>trait_description</code>	Trait description to pass to <code>together_compare_pair_live()</code> .

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 <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: 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_typesafe_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `typesafe_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_typesafe_pairs_live

Collect and resume live TypeSafe Jev decisions

Description

Collect multiple binary decisions with the same scientific contract as [typesafe_compare_pair_live\(\)](#). Use [submit_llm_pairs\(\)](#) for provider-neutral dispatch. These are live requests, not managed Batch jobs.

Usage

```

submit_typesafe_pairs_live(
  pairs,
  model = "jev-1.13.0",
  trait_name,
  trait_description,
  prompt_template = NULL,
  api_key = NULL,
  verbose = TRUE,
  status_every = 1,
  progress = TRUE,
  include_raw = FALSE,
  save_path = NULL,
  parallel = FALSE,
  workers = 1,
  execution_control = list(),
  ...
)

```

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	Requested Jev identifier; defaults to versioned "jev-1.13.0".
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	Must be omitted or NULL; custom decision instructions are supplied with decision_template in
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 future.apply for generative backends, requiring future and future.apply . Decision concurrency uses optional promises/later.
workers	Integer; the number of parallel workers (threads) to use if parallel = TRUE. Defaults to 1.
execution_control	Named list of decision-only scheduling, retry, timeout and lock controls; defaults to list(). See decision_bulk for all defaults, persistent budgets, probability storage and recovery guarantees.
...	Only decision_template is supported. See typesafe_compare_pair_live() for the complete scientific/provider contract.

Details

See [decision_bulk](#) for execution controls, persistent retry budgets, locking, probability storage, and interruption/recovery guarantees.

Value

The results, failed_pairs, and failed_attempts list documented in [submit_llm_pairs\(\)](#).

See Also

[submit_llm_pairs\(\)](#), [submit_openai_pairs_live\(\)](#), [decision_bulk](#)

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\(\)](#), [typesafe_compare_pair_live\(\)](#), [vertex_compare_pair_live\(\)](#)

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 <code>set_prompt_template()</code> .
api_key	Optional Vertex API key.
temperature	Optional numeric temperature; forwarded to <code>vertex_compare_pair_live()</code> .
top_p	Optional numeric; forwarded to <code>vertex_compare_pair_live()</code> .
top_k	Optional numeric; forwarded to <code>vertex_compare_pair_live()</code> .
max_output_tokens	Optional integer; forwarded to <code>vertex_compare_pair_live()</code> .
thinking_level	Optional Gemini 3 thinking level; forwarded to <code>vertex_compare_pair_live()</code> .
thinking_budget	Optional integer; forwarded to <code>vertex_compare_pair_live()</code> .
service_tier	Vertex AI service tier forwarded to <code>vertex_compare_pair_live()</code> .
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 <code>vertex_compare_pair_live()</code> .

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()`, `submit_typesafe_pairs_live()`, `together_compare_pair_live()`, `typesafe_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,
  include_starvation = FALSE,
  include_bootstrap = FALSE
)
```

Arguments

<code>state</code>	Adaptive state.
<code>include_starvation</code>	Logical; append a <code>starvation_diagnostic</code> list-column containing terminal hybrid exhaustion evidence. Default FALSE.

include_bootstrap

Logical; add a bootstrap list-column with policy, version, initialization seed, integrity identities, and saved tree diagnostics. Default FALSE retains the historical summary columns. Legacy sessions report the shuffled policy and no predictive bootstrap digest or diagnostics.

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. For an ordinary within-set run with N items, mean comparison exposure is $2 * \text{committed_pairs} / N$. This count includes bootstrap comparisons and excludes invalid attempts. A planned exposure budget B can use the explicit rounding convention $\text{floor}(B * N / 2)$; this summary does not enforce budgets or imply connectivity before $N - 1$ successful bootstrap comparisons.

The optional diagnostic is NULL when no current terminal evidence is available, including sessions saved before this diagnostic was introduced. Otherwise it contains a classification, originating step, committed count, active item-set scope, maximum observations per pair, and a remaining arithmetic capacity upper bound. The bound includes bootstrap history and ignores pairing restrictions; it is not a feasibility forecast. Sparse replay counts only unused allowed edges. The attempt table retains each failed stage's fallback policies, filter counts, pre-exposure hard-filter boundary, admissible candidate count, and bounded-search flag. Counts refer to examined pools; overlapping pools must not be summed. A missing bounded-search flag means its extent was not recorded. Classification distinguishes pair-capacity exhaustion, duplicate-policy exhaustion, exposure/star-cap exhaustion, other or mixed restrictions, unknown evidence, and selection inconsistency (surviving candidates despite starvation). Only zero arithmetic capacity establishes global pair-capacity exhaustion. Other classifications describe observed filter collapse, not proof of global infeasibility. Existing stop reasons and canonical logs are unchanged.

Value

A one-row tibble with columns `n_items`, `steps_attempted`, `committed_pairs`, `n_refits`, `last_stop_decision`, and `last_stop_reason`, plus the optional `starvation_diagnostic` and bootstrap list-columns.

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()`, `make_adaptive_judge_replay()`, `make_adaptive_replay_replay()`, `validate_adaptive_replay()`

Examples

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

summarize_bt_fit	<i>Summarize a Bradley–Terry model fit</i>
------------------	--

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", "BradleyTerry2", "brglm2", "alpha", or "lapse")
- **reliability**: raw sirt reliability, calculated Firth/alpha SSR, or NA

Usage

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

Arguments

fit	A list returned by <code>fit_bt_model</code> .
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.

Details

Standard errors describe model uncertainty; small differences in estimates should not be interpreted without considering that uncertainty. Reliability is a study-level summary repeated on each row, not an item-specific score. The returned rows retain input order; sort explicitly by rank or theta when preparing a ranked report. Valid lapse-boundary fits retain point estimates and ranks with missing SEs; reliability is unavailable for every lapse fit.

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", "BradleyTerry2", "brglm2", "alpha", or "lapse").

reliability Reliability (numeric scalar, or NA) repeated on each row.

See Also

`build_bt_data()`, `fit_bt_model()`

Other frequentist models: `bootstrap_bt_model()`, `build_bt_data()`, `build_elo_data()`, `fit_bt_model()`, `fit_elo_model()`, `predict.pairwiseLLM_bt_alpha()`, `predict.pairwiseLLM_bt_firth()`, `predict.pairwiseLLM_bt_logit()`, `scale_separation_reliability()`

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

summarize_items	<i>Summarize adaptive items</i>
-----------------	---------------------------------

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

- `state` An `adaptive_state` or list containing adaptive logs.
- `posterior` Optional `item_log_list` (list of item log tables) or an item log table. When `NULL`, uses `state$logs$item_log_list` when available.

<code>refit</code>	Optional refit index. When NULL, the most recent refit is returned; when set, the k-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\(\)](#) and the [integrated CJ workflow](#)

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)
```

```
summary.pairwiseLLM_link_result
```

Inspect explicit-evidence linking results and sessions

Description

Inspect explicit-evidence linking results and sessions

Usage

```
## S3 method for class 'pairwiseLLM_link_result'
summary(object, ...)

## S3 method for class 'pairwiseLLM_link_session'
summary(object, ...)

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

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

Arguments

object, x	A common linking result or linking session.
...	Unused.

Details

theta_link_eap is a compatibility alias for theta_link_mean: a posterior mean for E1 quadrature and E3-MCMC, and the MAP location for E2/E3 Laplace. Unavailable uncertainty remains NA. E1 uncertainty is conditional on fixed Phase A shapes; E2/E3 include shapes and offset.

Value

summary() returns one row per spoke, including offset, uncertainty scope, evidence counts, validity and controller status. print() returns its input invisibly. [summarize_items\(\)](#) supplies linked item means, SDs, intervals, ranks, estimator ID and uncertainty scope.

See Also

[fit_link\(\)](#), [start_link_session\(\)](#), [summarize_items\(\)](#), [summarize_refits\(\)](#)

Other linking: [fit_link\(\)](#), [predict_link\(\)](#), [prepare_link_input\(\)](#), [save_link_session\(\)](#), [start_link_session\(\)](#)

summary.pairwiseLLM_warm_algorithm_ensemble

Inspect a same-task algorithm ensemble

Description

Inspect a same-task algorithm ensemble

Usage

```
## S3 method for class 'pairwiseLLM_warm_algorithm_ensemble'
summary(object, ...)
```

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

Arguments

object, x	A full or summary-only ensemble_warm_start_algorithms() artifact.
...	Reserved; must be empty.

Value

`summary()` returns shared identity, component summaries, audit status and honest outer ensemble validation metrics. Summary-only metrics are explicitly labeled and cannot be recomputed without the original audit. `print()` returns its input invisibly.

See Also

Other adaptive warm start: [ensemble_warm_start_algorithms\(\)](#), [ensemble_warm_start_models\(\)](#), [extract_warm_start_features\(\)](#), [fit_warm_start_model\(\)](#), [make_warm_start_cv_plan\(\)](#), [make_warm_start_prior\(\)](#), [pairwiseLLM_warm_model](#), [predict.pairwiseLLM_warm_algorithm_ensemble\(\)](#), [predict.pairwiseLLM_warm_ensemble\(\)](#), [predict.pairwiseLLM_warm_model\(\)](#), [prepare_warm_start_model\(\)](#), [register_warm_start_model\(\)](#), [save_warm_start_model\(\)](#), [summary.pairwiseLLM_warm_ensemble\(\)](#), [summary.pairwiseLLM_warm_predictions\(\)](#), [warm_start_coefficients\(\)](#), [warm_start_feature_schema\(\)](#), [warm_start_python_status\(\)](#)

Examples

```

if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("pls", quietly = TRUE) && requireNamespace("withr", quietly = TRUE)) {
  local({
    withr::local_seed(259L)
    fields <- warm_start_feature_schema()$feature
    x <- as.data.frame(matrix(runif(20 * length(fields)), nrow = 20))
    names(x) <- fields
    x$n_tokens <- 11:30
    x <- data.frame(item_id = as.character(1:20), x)
    attr(x, "warm_start_schema") <- "writing_features_v1"
    theta <- x$n_tokens - 3 * x$token_length_mean
    plan <- make_warm_start_cv_plan(x$item_id, theta, "synthetic", seed = 259L)
    en <- fit_warm_start_model(x$item_id, theta, "synthetic", features = x,
      cv_plan = plan, alpha_grid = c(0, 1))
    pls <- fit_warm_start_model(x$item_id, theta, "synthetic", features = x,
      cv_plan = plan, engine = "pls", engine_control = list(ncomp = 1L))
    ensemble <- ensemble_warm_start_algorithms(elastic_net = en, pls = pls)
    print(ensemble)
    summary(ensemble)$validation$metrics
    predictions <- predict(ensemble, x)
    make_warm_start_prior(predictions, prior_sd = 0.5)
  })
}

```

```
summary.pairwiseLLM_warm_ensemble
```

Inspect a warm-start ensemble

Description

Inspect a warm-start ensemble

Usage

```

## S3 method for class 'pairwiseLLM_warm_ensemble'
summary(object, ...)

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

```

Arguments

object, x	A pairwiseLLM_warm_ensemble.
...	Reserved; must be empty.

Value

summary() returns component summaries and ensemble contract details. print() returns its input invisibly. Component metrics do not validate the ensemble. Between-model SD is diagnostic disagreement, not Bayesian prior SD.

See Also

[ensemble_warm_start_models\(\)](#), [predict.pairwiseLLM_warm_ensemble\(\)](#)

Other adaptive warm start: [ensemble_warm_start_algorithms\(\)](#), [ensemble_warm_start_models\(\)](#), [extract_warm_start_features\(\)](#), [fit_warm_start_model\(\)](#), [make_warm_start_cv_plan\(\)](#), [make_warm_start_prior\(\)](#), [pairwiseLLM_warm_model](#), [predict.pairwiseLLM_warm_algorithm_ensemble\(\)](#), [predict.pairwiseLLM_warm_ensemble\(\)](#), [predict.pairwiseLLM_warm_model\(\)](#), [prepare_warm_start_model\(\)](#), [register_warm_start_model\(\)](#), [save_warm_start_model\(\)](#), [summary.pairwiseLLM_warm_algorithm_ensemble\(\)](#), [summary.pairwiseLLM_warm_predictions\(\)](#), [warm_start_coefficients\(\)](#), [warm_start_feature_schema\(\)](#), [warm_start_python_status\(\)](#)

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  local({
    # Synthetic features illustrate the interface, not predictive validity.
    example_features <- function(seed) {
      withr::local_seed(seed)
      fields <- warm_start_feature_schema()$feature
      x <- as.data.frame(matrix(runif(15 * length(fields)), nrow = 15))
      names(x) <- fields
      x$n_tokens <- 11:25
      x$token_length_mean <- 2 + 10 * x$token_length_mean
      x$token_length_std <- 0.2 + x$token_length_std
      x$dale_chall_readability_score <- 5 + 20 * x$dale_chall_readability_score
      x <- data.frame(item_id = as.character(1:15), x)
      attr(x, "warm_start_schema") <- "writing_features_v1"
      x
    }
    features <- example_features(3103)
    theta <- 10 + 0.4 * features$n_tokens - 2 * features$token_length_mean
    # A small alpha grid keeps this example fast; the default has 41 values.
    model <- fit_warm_start_model(features$item_id, theta, "synthetic-a",
      features = features, alpha_grid = c(0, 1))
    features_b <- example_features(3104)
    theta_b <- 30 + features_b$n_tokens - 3 * features_b$token_length_mean
    model_b <- fit_warm_start_model(features_b$item_id, theta_b, "synthetic-b",
      features = features_b, alpha_grid = c(0, 1))
    ensemble <- ensemble_warm_start_models(assessment_a = model, assessment_b = model_b)
    summary(ensemble)
    print(ensemble)
  })
}
```

```
summary.pairwiseLLM_warm_predictions
```

Inspect ensemble predictions

Description

Inspect ensemble predictions

Usage

```
## S3 method for class 'pairwiseLLM_warm_predictions'
summary(object, ...)
```

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

Arguments

object, x	A pairwiseLLM_warm_predictions tibble.
...	Passed to tibble printing; summary arguments must be empty.

Value

summary() returns item count, component names, and summaries of the mean and diagnostic sample SD. print() invisibly returns its input.

See Also

[predict.pairwiseLLM_warm_ensemble\(\)](#), [make_warm_start_prior\(\)](#)

Other adaptive warm start: [ensemble_warm_start_algorithms\(\)](#), [ensemble_warm_start_models\(\)](#), [extract_warm_start_features\(\)](#), [fit_warm_start_model\(\)](#), [make_warm_start_cv_plan\(\)](#), [make_warm_start_prior\(\)](#), [pairwiseLLM_warm_model](#), [predict.pairwiseLLM_warm_algorithm_ensemble\(\)](#), [predict.pairwiseLLM_warm_ensemble\(\)](#), [predict.pairwiseLLM_warm_model\(\)](#), [prepare_warm_start_model\(\)](#), [register_warm_start_model\(\)](#), [save_warm_start_model\(\)](#), [summary.pairwiseLLM_warm_algorithm_ensemble\(\)](#), [summary.pairwiseLLM_warm_ensemble\(\)](#), [warm_start_coefficients\(\)](#), [warm_start_feature_schema\(\)](#), [warm_start_python_status\(\)](#)

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  local({
    # Synthetic features illustrate the interface, not predictive validity.
    example_features <- function(seed) {
      withr::local_seed(seed)
      fields <- warm_start_feature_schema()$feature
      x <- as.data.frame(matrix(runif(15 * length(fields)), nrow = 15))
      names(x) <- fields
    }
  })
}
```

```

    x$n_tokens <- 11:25
    x$token_length_mean <- 2 + 10 * x$token_length_mean
    x$token_length_std <- 0.2 + x$token_length_std
    x$dale_chall_readability_score <- 5 + 20 * x$dale_chall_readability_score
    x <- data.frame(item_id = as.character(1:15), x)
    attr(x, "warm_start_schema") <- "writing_features_v1"
  }
  x
}
features <- example_features(3103)
theta <- 10 + 0.4 * features$n_tokens - 2 * features$token_length_mean
# A small alpha grid keeps this example fast; the default has 41 values.
model <- fit_warm_start_model(features$item_id, theta, "synthetic-a",
  features = features, alpha_grid = c(0, 1))
features_b <- example_features(3104)
theta_b <- 30 + features_b$n_tokens - 3 * features_b$token_length_mean
model_b <- fit_warm_start_model(features_b$item_id, theta_b, "synthetic-b",
  features = features_b, alpha_grid = c(0, 1))
ensemble <- ensemble_warm_start_models(assessment_a = model, assessment_b = model_b)
predictions <- predict(ensemble, features)
summary(predictions)
print(predictions)
})
}

```

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 <code>set_prompt_template()</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>".
api_key	Optional Together.ai API key. If NULL or empty, the helper falls back to the TOGETHER_API_KEY environment variable via <code>.together_api_key()</code> .
include_raw	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.
...	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 ..., 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_typesafe_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [typesafe_compare_pair_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 = tpl
)

res_deepseek$better_id
res_deepseek$thoughts

## End(Not run)
```

trait_description	Get a trait name and description for prompts
-------------------	--

Description

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

Usage

```
trait_description(
  name = c("overall_quality", "organization", "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
```

typesafe_compare_pair_live

Live TypeSafe Jev comparison for a single pair of samples

Description

Compare two writing samples on one trait using TypeSafe's binary Choice API, POST https://api.typesafe.ai/v1/systems/decision_cj_v1. The default is the exact approved `decision_cj_v1` instructions, with the complete trait description preserved. This opt-in helper uses the same scientific request, integrity validator, response parser, and binary winner semantics as OpenAI Decisions.

Usage

```
typesafe_compare_pair_live(
  ID1,
  text1,
  ID2,
  text2,
  model = "jev-1.13.0",
  trait_name,
  trait_description,
  prompt_template = NULL,
  api_key = NULL,
  include_raw = FALSE,
  ...
)
```

Arguments

ID1, ID2	Distinct nonempty character scalar sample IDs, in presentation order. Reversing the presentation reverses the sample-to-position mapping.
text1, text2	Nonmissing character scalar sample texts. Empty strings are allowed. Whitespace, Unicode characters, and line endings are preserved.
model	Requested model identifier. Defaults to the explicit version "jev-1.13.0". Other jev-<major>.<minor>.<patch> identifiers and the documented aliases "jev-latest" and "jev-preview" are accepted. Identifiers are checked for syntax, not availability; TypeSafe determines whether the account can use them. No model-list preflight, version allowlist, substitution, or automatic fallback is used. Prefer versioned identifiers for reproducible research: aliases can resolve differently over time.
trait_name	Nonempty character scalar trait label.
trait_description	Nonempty character scalar containing the full trait definition, including construct boundaries; inserted verbatim.
prompt_template	Must be omitted or NULL. Generative templates are rejected. Use decision_template in ... for custom decision instructions.
api_key	Optional character scalar TypeSafe key. NULL or "" uses TYPESAFE_API_KEY through the shared credential resolver. Explicit keys take precedence; missing, malformed, whitespace-containing, or control-character keys fail without printing their values. Persist credentials in .Renviron and restart R, or use Sys.setenv(TYPESAFE_API_KEY = "YOUR_KEY_HERE") for the current session. check_llm_api_keys() reports presence only.
include_raw	Logical scalar. Retain successful parsed responses when TRUE. Failed decision responses are always retained for diagnosis.
...	Only pair_uid and decision_template are supported, and must be uniquely named. pair_uid identifies the scheduled pair occurrence; its default is "LIVE_<ID1>_vs_<ID2>". Custom decision_template instructions must contain {TRAIT_NAME} and {TRAIT_DESCRIPTION}, and must not contain sample placeholders or generative output tags. Custom instructions are fingerprinted. Sampling, reasoning, token limits, caching, storage, tiers, arbitrary URLs/headers, and long-context/image controls are unsupported, including explicitly supplied NULL settings. Unknown settings fail before credential resolution or HTTP.

Details

The wire request contains model, state (a JSON string with exactly the ordered SAMPLE_1 and SAMPLE_2 texts), and a questions map with one writing_trait entry. Its type is choice, with rendered instructions and symmetric SAMPLE_1/SAMPLE_2 criteria descriptions. Responses use an answers map with choice, a complete probabilities map, and confidence.

The shared integrity validator checks the complete scientific body against recorded provenance and its SHA-256 fingerprint before provider operations. The serialized JSON is checked again, attached as raw UTF-8 bytes, and checked with the URL/content type before every HTTP attempt. Altered

requests fail with `pairwiseLLM_decision_request_error`, without repair or HTTP. This is an integrity/reproducibility guard, not an authenticated signature.

Exactly two finite probabilities in $[0, 1]$, summing to 1 within $1e-6$, and finite confidence in $[0, 1]$ are required. Values are never renormalized, clipped, rounded, complemented, or fabricated. Exact 0, 0.5, and 1 survive. The explicit choice is retained; if the alternative exceeds its probability by more than $1e-8$, the result fails with `choice_probability_mismatch`. Missing confidence, malformed/unknown answer types, incomplete distributions, and invalid usage are classified by the shared parser. TypeSafe has no documented refusal variant; unknown types are malformed. These response failures are not retried and are excluded by the shared normalizer before modeling. Only successful normalized results are model evidence.

HTTP 408, 429, all 5xx statuses (including 529), and timeouts receive up to three total attempts with exponential delays of 0.5 and 1 second. Requests retain identical validated bytes. Other transport failures and permanent HTTP errors, including authentication, unavailable models, and schema errors, abort. Conditions have sample-free/key-free messages and `retry_failures`; HTTP conditions also carry `status_code`. Backoff uses the package's shared retry helper; it does not currently interpret `Retry-After` headers.

Only the standard text endpoint is supported: no SDK is required, and no compression, long-context/image opt-in, or managed Batch is implemented here. Use `submit_typesafe_pairs_live()` for resumable live bulk collection. The provider enforces body/context limits. An owner-authorized live smoke run on 2026-10-16 exercised `jev-1.13.0` through this endpoint on 20 synthetic samples (60 primary pairs and 10 reverse audits); other versions/aliases were not qualified. See the model-compatibility vignette for the tested commit, design and limits. Model probabilities do not establish validated educational assessment accuracy and do not become BT/adaptive weights, fractional wins, or judgments.

Sample text is sent to TypeSafe. Keys are omitted from diagnostics. Request bodies and raw responses can contain private text; sample-free provenance still includes possibly sensitive custom instructions and trait definitions. Avoid logging them publicly. This helper provides no data-retention control.

Value

A one-row tibble with the shared `decision_results` columns:

ID1, ID2, pair_uid, custom_id Presentation and scheduled pair identity.

better_sample, better_id, winner_pos Explicit selected sample label, corresponding ID, and integer presentation position.

p_sample_1, p_sample_2, p_chosen, probabilities_json Both provider probabilities, the selected probability, and the full binary map.

decision_confidence Independent provider-reported confidence; never inferred from or substituted for the selected probability.

requested_model, resolved_model, model Requested identifier, provider-reported identifier (NA when absent), and resolved identifier when supplied, otherwise the requested identifier. Missing resolution metadata does not establish which model weights served an alias.

prompt_tokens, completion_tokens, total_tokens Reported input, output, and total usage; unavailable counts are NA, not inferred. Shared cached/write/reasoning token columns are also retained if supplied.

endpoint, decision_question, decision_type, decision_template_version, request_fingerprint, request_provenance_json Scientific identity. Provenance includes provider, full trait/instructions/criteria, ordered IDs, and exact text hashes.

provider_request_id Optional x-request-id header, otherwise NA.

object_type, status_code, error_message "decision", HTTP status, and sample-free decision failure detail (NA on success).

decision_error_code, decision_error_detail Shared response-validation failure classification and detail; NA on success.

thoughts, content NA; this endpoint does not generate assessment prose.

retry_failures List-column of preceding transient attempt records.

raw_response Optional parsed response, always present on decision failure, including malformed output and inconsistent probabilities.

See Also

[llm_compare_pair\(\)](#), [check_llm_api_keys\(\)](#), [decision_request](#), [decision_results](#), [openai_compare_pair_live\(\)](#), [TypeSafe API reference](#), [TypeSafe models](#)

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_typesafe_pairs_live\(\)](#), [submit_vertex_pairs_live\(\)](#), [together_compare_pair_live\(\)](#), [vertex_compare_pair_live\(\)](#)

Examples

```
## Not run:
# Requires TYPESAFE_API_KEY and incurs provider usage charges.
result <- typesafe_compare_pair_live(
  "A", "Synthetic sample A.", "B", "Synthetic sample B.",
  trait_name = "Organization",
  trait_description = "Logical sequencing and connections between ideas."
)
result$better_id
result$p_sample_1

# The same backend through the public dispatcher:
llm_compare_pair(
  "A", "Synthetic sample A.", "B", "Synthetic sample B.",
  model = "jev-1.13.0", backend = "typesafe",
  trait_name = "Organization",
  trait_description = "Logical sequencing and connections between ideas."
)

## End(Not run)
```

 validate_adaptive_replay

Validate frozen directed judgments for an adaptive panel

Description

Validate one primary observation per ordered pair. No rows are dropped or reordered; optional metadata columns are retained and do not determine lookup. Subset to one panel and one primary judgment layer before validation.

Usage

```
validate_adaptive_replay(outcomes, item_ids, complete = TRUE)
```

Arguments

outcomes	Data frame with character A_id, character B_id, and binary Y (one means presented A wins). Y accepts logical, numeric zero/one, or character "0"/"1"; factors, missing values, and other values are rejected.
item_ids	Unique non-blank character IDs for the active panel, with at least two items. IDs must match the adaptive state's item IDs.
complete	Logical; require all $N * (N - 1)$ ordered pairs, including both orientations of every unordered pair. Default TRUE. FALSE permits a partial table; requests for absent orientations still fail during replay.

Value

A tibble retaining the input rows and metadata, with integer Y.

See Also

[make_adaptive_judge_replay\(\)](#)

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

Examples

```
outcomes <- data.frame(A_id = c("a", "b"), B_id = c("b", "a"), Y = c(1L, 1L))
validate_adaptive_replay(outcomes, item_ids = c("a", "b"))
```

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: known legacy fields are backfilled before checking canonical columns and types. Other added/removed/reordered columns abort validation. Saved mode/strategy metadata must agree with the authoritative state.

Value

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

See Also

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

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

Examples

```
dir <- tempfile("pwillm-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
<i>Live Vertex AI Gemini comparison for a single pair of samples</i>

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_typesafe_pairs_live()`, `submit_vertex_pairs_live()`, `together_compare_pair_live()`, `typesafe_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)
```

warm_start_coefficients

Inspect calibrated standardized warm-start coefficients

Description

`warm_start_coefficients()` reports fitted elastic-net or PLS coefficients after applying the model's learned OOF linear calibration. It supports individual calibrated warm-start models and equal-weight warm-start ensembles.

Usage

```
warm_start_coefficients(object, ...)

## S3 method for class 'pairwiseLLM_warm_model'
warm_start_coefficients(object, ...)

## S3 method for class 'pairwiseLLM_warm_ensemble'
warm_start_coefficients(object, ...)

## S3 method for class 'pairwiseLLM_warm_algorithm_ensemble'
warm_start_coefficients(object, ...)
```

Arguments

<code>object</code>	A calibrated <code>pairwiseLLM_warm_model</code> or <code>pairwiseLLM_warm_ensemble</code> or <code>pairwiseLLM_warm_algorithm_ensemble</code> object.
<code>...</code>	Reserved for future extensions; must be empty.

Details

Predictors use each component model's fitted training medians, centers, and sample SDs. The fitted target is within-task standardized BT/BTL quality. For each retained feature, the reported value is the stored elastic-net or PLS coefficient multiplied by the learned OOF calibration slope. Thus, holding the other included predictors fixed, it is the change in calibrated within-task standardized prediction for a one-training-SD increase in that feature.

`retained = FALSE` with an NA coefficient means preprocessing removed the feature and no fitted standardized coefficient exists. `retained = TRUE` with coefficient zero means the feature survived preprocessing but has zero calibrated weight at the selected engine hyperparameters.

Positive and negative signs describe fitted direction conditional on the other included predictors. Correlated predictors can share or trade fitted weight, so coefficient magnitude is not a unique measure of predictive importance, causal influence, or explained variance.

Nonlinear SVR models have no linear feature coefficients. Requests raise `pairwiseLLM_warm_nonlinear_coefficients`, also naming the nonlinear component when an ensemble contains SVR. Dual coefficients are kernel weights, not linear feature effects.

Ensemble columns show component coefficients side by side. Each component standardized predictors using its own training distribution, so columns do not imply one common raw-feature SD. The table exposes fitted direction, magnitude, and stability across component models; it neither defines an aggregate coefficient nor changes equal prediction weighting. Inspection from an existing portable artifact needs neither Python nor a fitting backend.

Value

For an individual model, a tibble with exactly `feature`, `retained`, and `calibrated_std_coefficient`, in the model's frozen feature order. For an ensemble, a tibble with `feature` followed by one `<component>_std_coefficient` column per component, in component order.

See Also

```
fit_warm_start_model(), ensemble_warm_start_models(), ensemble_warm_start_algorithms(),
predict.pairwiseLLM_warm_model()
```

Other adaptive warm start: `ensemble_warm_start_algorithms()`, `ensemble_warm_start_models()`, `extract_warm_start_features()`, `fit_warm_start_model()`, `make_warm_start_cv_plan()`, `make_warm_start_prior()`, `pairwiseLLM_warm_model`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_ensemble()`, `predict.pairwiseLLM_warm_model()`, `prepare_warm_start_model()`, `register_warm_start_model()`, `save_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_ensemble()`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_feature_schema`, `warm_start_python_status()`

Examples

```
if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  local({
    withr::local_seed(3103)
    fields <- warm_start_feature_schema()$feature
    x <- as.data.frame(matrix(runif(15 * length(fields)), nrow = 15))
    names(x) <- fields
    x$n_tokens <- 11:25
    x$token_length_mean <- 2 + 10 * x$token_length_mean
    x$token_length_std <- 0.2 + x$token_length_std
    x$dale_chall_readability_score <- 5 + 20 * x$dale_chall_readability_score
    x <- data.frame(item_id = as.character(1:15), x)
    attr(x, "warm_start_schema") <- "writing_features_v1"
    theta <- 10 + 0.4 * x$n_tokens - 2 * x$token_length_mean
    model <- fit_warm_start_model(x$item_id, theta, "synthetic-example",
      features = x, alpha_grid = c(0, 1))
    warm_start_coefficients(model)
  })
}
```

```
warm_start_feature_schema
```

Inspect the frozen warm-start writing feature schema

Description

Warm-start prediction uses text features to predict relative writing quality before collecting pairwise comparisons. This function lists the frozen predictor definitions; it does not extract features or require Python.

Usage

```
warm_start_feature_schema(schema = "writing_features_v1")
```

Arguments

schema A single schema identifier: "writing_features_v1" (default, 20 features) or "writing_features_v2" (46 features).

Details

Version 1 describes English writing using TextDescriptives and a supplementary textstat readability measure. The schema records definitions, not evidence of predictive validity for any particular writing population. Version 2 retains all version 1 features first, followed by 26 source-audited scalar additions in inventory order. It adds counts, medians, syllable summaries, AUX/DET/PART proportions, dependency variation, second-order coherence and readability measures using the same pinned Python environment. Conceptual overlap is intentional; selection did not use training outcomes. Inspect the installed python/schema-writing-v2.json for its CSV SHA-256.

upstream_entropy_per_token divides TextDescriptives' entropy by the number of all spaCy tokens, including punctuation and whitespace tokens. This is an average of upstream probability-weighted contributions, not normalized document Shannon entropy or language-model cross-entropy. Its denominator differs from the filtered n_tokens feature. Zero tokens give a missing value.

Definitions, settings, membership, and ordering are frozen. Changes require a new schema identifier rather than silently modifying an existing schema. Optional Python software is needed only for extraction, not schema inspection or prediction from precomputed features.

Value

A tibble with one row per feature in fixed predictor order. Columns:

- schema, position, feature: version, integer order, and canonical name.
- source_package, source_version, component, upstream_field: audited upstream mapping.
- family, type, unit, interpretation: meaning of the feature.
- configuration, definition, requirements, missing_behavior: fixed calculation settings, formula, capabilities, and undefined values.
- source_url: upstream implementation reference.

See Also

`extract_warm_start_features()`, `fit_warm_start_model()`

Other adaptive warm start: `ensemble_warm_start_algorithms()`, `ensemble_warm_start_models()`, `extract_warm_start_features()`, `fit_warm_start_model()`, `make_warm_start_cv_plan()`, `make_warm_start_prior()`, `pairwiseLLM_warm_model`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_ensemble()`, `predict.pairwiseLLM_warm_model()`, `prepare_warm_start_model()`, `register_warm_start_model()`, `save_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_ensemble()`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_coefficients()`, `warm_start_python_status()`

Examples

```
features <- warm_start_feature_schema()
features[c("position", "feature", "family")]
```

warm_start_python_status

Check the optional warm-start feature environment

Description

Check the optional warm-start feature environment

Usage

```
warm_start_python_status(schema = "writing_features_v1", python = NULL)
```

Arguments

schema	The frozen schema identifier: "writing_features_v1" (default) or "writing_features_v2".
python	Optional path to an existing Python interpreter. With NULL, use an existing environment selected through reticulate. Automatic environment creation is disabled. Conflicting interpreter selections require correction or a fresh R session; this function never switches an initialized interpreter.

Details

This explicitly invoked check may initialize reticulate's Python interpreter and load the spaCy model to verify its capabilities. It never installs software or downloads resources. Once Python is initialized, selecting a different interpreter requires restarting R. Use `reticulate::use_virtualenv()` or `reticulate::use_condaenv()` with `required = TRUE` before this call, or supply `python` directly. See the installed `python/README.md` for explicit setup.

Value

A list with available, python, expected, observed, and problems. Availability requires the pinned versions, resource contents, and pipeline capabilities. Missing prerequisites are reported without throwing an error; malformed arguments or missing installed metadata are errors.

See Also

`extract_warm_start_features()`, `warm_start_feature_schema()`

Other adaptive warm start: `ensemble_warm_start_algorithms()`, `ensemble_warm_start_models()`, `extract_warm_start_features()`, `fit_warm_start_model()`, `make_warm_start_cv_plan()`, `make_warm_start_prior()`, `pairwiseLLM_warm_model`, `predict.pairwiseLLM_warm_algorithm_ensemble()`, `predict.pairwiseLLM_warm_ensemble()`, `predict.pairwiseLLM_warm_model()`, `prepare_warm_start_model()`, `register_warm_start_model()`, `save_warm_start_model()`, `summary.pairwiseLLM_warm_algorithm_ensemble()`, `summary.pairwiseLLM_warm_ensemble()`, `summary.pairwiseLLM_warm_predictions()`, `warm_start_coefficients()`, `warm_start_feature_schema()`

Examples

```
## Not run:
warm_start_python_status(python = "/path/to/venv/bin/python")

## 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

<code>batch_tbl</code>	A data frame or tibble, typically the result of build_openai_batch_requests .
<code>path</code>	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_errors\(\)](#), [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, [55](#)
 - fit_bayes_btl_mcmc, [89](#)
- * **Ollama resource management**
 - ensure_only_ollama_model_loaded, [76](#)
- * **adaptive logs**
 - adaptive_get_logs, [6](#)
 - adaptive_item_log, [7](#)
 - adaptive_results_history, [33](#)
 - adaptive_round_log, [34](#)
 - adaptive_step_log, [35](#)
 - summarize_items, [250](#)
 - summarize_refits, [252](#)
- * **adaptive persistence**
 - load_adaptive_session, [141](#)
 - save_adaptive_session, [215](#)
 - validate_session_dir, [268](#)
- * **adaptive ranking**
 - adaptive_rank, [8](#)
 - adaptive_rank_resume, [20](#)
 - adaptive_rank_run_live, [22](#)
 - adaptive_rank_start, [29](#)
 - make_adaptive_judge_llm, [143](#)
 - make_adaptive_judge_replay, [145](#)
 - make_adaptive_replay_reservoir, [146](#)
 - summarize_adaptive, [247](#)
 - validate_adaptive_replay, [267](#)
- * **adaptive warm start**
 - ensemble_warm_start_algorithms, [73](#)
 - ensemble_warm_start_models, [74](#)
 - extract_warm_start_features, [88](#)
 - fit_warm_start_model, [108](#)
 - make_warm_start_cv_plan, [149](#)
 - make_warm_start_prior, [150](#)
 - pairwiseLLM_warm_model, [167](#)
 - predict.pairwiseLLM_warm_algorithm_ensemble, [179](#)
 - predict.pairwiseLLM_warm_ensemble, [181](#)
 - predict.pairwiseLLM_warm_model, [182](#)
 - prepare_warm_start_model, [190](#)
 - register_warm_start_model, [199](#)
 - save_warm_start_model, [217](#)
 - summary.pairwiseLLM_warm_algorithm_ensemble, [255](#)
 - summary.pairwiseLLM_warm_ensemble, [256](#)
 - summary.pairwiseLLM_warm_predictions, [258](#)
 - warm_start_coefficients, [271](#)
 - warm_start_feature_schema, [273](#)
 - warm_start_python_status, [275](#)
- * **batch backends**
 - anthropic_create_batch, [42](#)
 - anthropic_download_batch_results, [44](#)
 - anthropic_get_batch, [45](#)
 - anthropic_poll_batch_until_complete, [46](#)
 - build_anthropic_batch_requests, [51](#)
 - build_gemini_batch_requests, [57](#)
 - build_openai_batch_requests, [60](#)
 - gemini_create_batch, [116](#)
 - gemini_download_batch_results, [118](#)
 - gemini_get_batch, [120](#)
 - gemini_poll_batch_until_complete, [121](#)
 - llm_download_batch_results, [129](#)
 - llm_resume_multi_batches, [131](#)
 - llm_submit_pairs_batch, [133](#)
 - llm_submit_pairs_multi_batch, [138](#)
 - openai_create_batch, [159](#)
 - openai_download_batch_errors, [161](#)
 - openai_download_batch_output, [162](#)
 - openai_get_batch, [163](#)

- openai_poll_batch_until_complete, 164
- openai_upload_batch_file, 166
- run_anthropic_batch_pipeline, 202
- run_gemini_batch_pipeline, 205
- run_openai_batch_pipeline, 209
- write_openai_batch_file, 276
- * **bias and consistency**
 - check_positional_bias, 66
 - compute_reverse_consistency, 68
- * **cost estimation**
 - estimate_llm_pairs_cost, 78
- * **datasets**
 - example_openai_batch_output, 83
 - example_writing_pairs, 84
 - example_writing_results, 85
 - example_writing_samples, 86
 - example_writing_samples1000, 87
- * **frequentist models**
 - bootstrap_bt_model, 48
 - build_bt_data, 54
 - build_elo_data, 56
 - fit_bt_model, 92
 - fit_elo_model, 99
 - predict.pairwiseLLM_bt_alpha, 175
 - predict.pairwiseLLM_bt_firth, 176
 - predict.pairwiseLLM_bt_lapse, 177
 - scale_separation_reliability, 219
 - summarize_bt_fit, 249
- * **linking**
 - fit_link, 101
 - predict_link, 184
 - prepare_link_input, 185
 - save_link_session, 216
 - start_link_session, 221
 - summary.pairwiseLLM_link_result, 254
- * **live backends**
 - anthropic_compare_pair_live, 38
 - check_llm_api_keys, 65
 - gemini_compare_pair_live, 112
 - llm_compare_pair, 125
 - ollama_compare_pair_live, 151
 - openai_compare_pair_live, 155
 - submit_anthropic_pairs_live, 222
 - submit_gemini_pairs_live, 225
 - submit_llm_pairs, 229
 - submit_ollama_pairs_live, 233
 - submit_openai_pairs_live, 236
 - submit_together_pairs_live, 240
 - submit_typesafe_pairs_live, 243
 - submit_vertex_pairs_live, 245
 - together_compare_pair_live, 259
 - typesafe_compare_pair_live, 263
 - vertex_compare_pair_live, 269
- * **pairing and data**
 - alternate_pair_order, 37
 - make_pairs, 147
 - randomize_pair_order, 194
 - read_samples_df, 195
 - read_samples_dir, 196
 - sample_pairs, 213
 - sample_reverse_pairs, 214
- * **prompts and traits**
 - build_prompt, 64
 - get_prompt_template, 123
 - list_prompt_templates, 124
 - register_prompt_template, 197
 - remove_prompt_template, 201
 - set_prompt_template, 220
 - trait_description, 262
- * **result normalization**
 - parse_anthropic_batch_output, 170
 - parse_gemini_batch_output, 172
 - parse_openai_batch_output, 173
- * **rubric calibration**
 - evaluate_rubric_predictions, 81
 - fit_rubric_calibration, 104
 - predict.pairwiseLLM_rubric_calibration, 178
 - prepare_linked_rubric_reference, 189
- adaptive_get_logs, 6
- adaptive_get_logs(), 8, 18, 33, 35, 36, 248, 251, 253
- adaptive_item_log, 7
- adaptive_item_log(), 7, 25, 32, 33, 35, 36, 251, 253
- adaptive_rank, 8
- adaptive_rank(), 21, 22, 25, 30, 32, 91, 104, 144, 146, 147, 248, 267
- adaptive_rank_resume, 20
- adaptive_rank_resume(), 18, 25, 32, 142, 144, 146, 147, 151, 248, 267
- adaptive_rank_run_live, 22

- `adaptive_rank_run_live()`, 8, 10, 14–16, 18, 21, 32, 35, 36, 49, 51, 143, 144, 146, 147, 248, 267
- `adaptive_rank_start`, 29
- `adaptive_rank_start()`, 18, 21, 22, 25, 49, 51, 144, 146, 147, 151, 248, 267
- `adaptive_results_history`, 33
- `adaptive_results_history()`, 7, 8, 35, 36, 251, 253
- `adaptive_round_log`, 34
- `adaptive_round_log()`, 7, 8, 25, 32, 33, 36, 251, 253
- `adaptive_step_log`, 35
- `adaptive_step_log()`, 7, 8, 25, 32, 33, 35, 251, 253
- `alternate_pair_order`, 37, 194
- `alternate_pair_order()`, 148, 194, 196, 197, 213, 214
- `anthropic_compare_pair_live`, 38, 52, 171, 222–224
- `anthropic_compare_pair_live()`, 65, 115, 125, 128, 154, 158, 224, 228, 232, 235, 238, 242, 244, 247, 261, 266, 271
- `anthropic_create_batch`, 42, 203, 204
- `anthropic_create_batch()`, 45–47, 53, 59, 63, 117, 119, 121, 122, 130, 133, 136, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- `anthropic_download_batch_results`, 44, 170, 203
- `anthropic_download_batch_results()`, 43, 46, 47, 53, 59, 63, 117, 119, 121, 122, 130, 133, 136, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- `anthropic_get_batch`, 45, 46, 47
- `anthropic_get_batch()`, 43, 45, 47, 53, 59, 63, 117, 119, 121, 122, 130, 133, 136, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- `anthropic_poll_batch_until_complete`, 46, 203
- `anthropic_poll_batch_until_complete()`, 43, 45, 46, 53, 59, 63, 117, 119, 121, 122, 130, 133, 136, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- `base::print()`, 248
- `bootstrap_bt_model`, 48
- `bootstrap_bt_model()`, 54, 57, 97, 98, 100, 176–178, 219, 250
- `BTm`, 92
- `btm`, 85, 92
- `build_anthropic_batch_requests`, 42, 51, 202, 203
- `build_anthropic_batch_requests()`, 43, 45–47, 59, 63, 117, 119, 121, 122, 130, 133, 134, 136, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- `build_bt_data`, 54, 92, 93, 174
- `build_bt_data()`, 33, 51, 57, 98, 100, 128, 153, 176–178, 219, 250
- `build_btl_results_data`, 55
- `build_btl_results_data()`, 90, 91, 189
- `build_elo_data`, 56, 99
- `build_elo_data()`, 51, 54, 98, 100, 176–178, 219, 250
- `build_gemini_batch_requests`, 57, 116, 207
- `build_gemini_batch_requests()`, 43, 45–47, 53, 63, 117, 119, 121, 122, 130, 133, 136, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- `build_openai_batch_requests`, 52, 60, 155, 156, 276
- `build_openai_batch_requests()`, 43, 45, 46, 48, 53, 59, 117, 119, 121, 122, 130, 133, 136, 140, 160, 162–164, 166, 167, 204, 207, 209–212, 276
- `build_prompt`, 64
- `build_prompt()`, 123, 124, 198, 201, 220, 263
- `check_llm_api_keys`, 65
- `check_llm_api_keys()`, 41, 115, 128, 154, 158, 224, 228, 232, 235, 238, 242, 244, 247, 261, 264, 266, 271
- `check_positional_bias`, 66, 69
- `check_positional_bias()`, 69
- `compute_reverse_consistency`, 66, 68, 194
- `compute_reverse_consistency()`, 67
- `decision_bulk`, 69, 127, 231, 237, 238, 244
- `decision_request`, 71, 72, 158, 266
- `decision_results`, 72, 157, 158, 265, 266

- elochoice, 99
- ensemble_warm_start_algorithms, 73
- ensemble_warm_start_algorithms(), 75, 89, 111, 150, 151, 169, 179, 180, 182, 183, 190, 191, 199, 200, 217, 218, 255, 257, 258, 273–275
- ensemble_warm_start_models, 74
- ensemble_warm_start_models(), 73, 74, 89, 111, 150, 151, 169, 180–183, 190, 191, 199, 200, 217, 218, 255, 257, 258, 273–275
- ensure_only_ollama_model_loaded, 76
- estimate_llm_pairs_cost, 78, 193
- evaluate_rubric_predictions, 81
- evaluate_rubric_predictions(), 105, 107, 179, 190
- example_openai_batch_output, 83
- example_writing_pairs, 84, 85
- example_writing_results, 85
- example_writing_samples, 84, 86
- example_writing_samples1000, 87
- extract_warm_start_features, 88
- extract_warm_start_features(), 74, 75, 111, 150, 151, 169, 180–183, 191, 200, 218, 255, 257, 258, 273–275
- fit_bayes_btl_mcmc, 85, 89
- fit_bayes_btl_mcmc(), 55, 56, 104, 151, 187, 189
- fit_bt_model, 92, 99, 249
- fit_bt_model(), 33, 49, 51, 54, 57, 100, 128, 153, 175–178, 219, 250
- fit_elo_model, 99
- fit_elo_model(), 51, 54, 57, 98, 176–178, 219, 250
- fit_link, 101
- fit_link(), 12, 24, 25, 31, 32, 103, 106, 184, 185, 188, 217, 222, 255
- fit_rubric_calibration, 104
- fit_rubric_calibration(), 83, 179, 190
- fit_warm_start_model, 108
- fit_warm_start_model(), 18, 73–75, 89, 150, 151, 168, 169, 180, 182, 183, 191, 200, 218, 255, 257, 258, 273–275
- gemini_compare_pair_live, 112
- gemini_compare_pair_live(), 41, 65, 125, 128, 154, 158, 207, 224–228, 232, 235, 238, 242, 244, 247, 261, 266, 271
- gemini_create_batch, 116, 120, 205
- gemini_create_batch(), 43, 45, 46, 48, 53, 59, 63, 119, 121, 122, 130, 133, 136, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- gemini_download_batch_results, 118
- gemini_download_batch_results(), 43, 45, 46, 48, 53, 59, 63, 117, 121, 122, 130, 133, 136, 140, 160, 162–164, 166, 167, 172, 204, 207, 212, 276
- gemini_get_batch, 120, 121, 122
- gemini_get_batch(), 43, 45, 46, 48, 53, 59, 63, 117, 119, 122, 130, 133, 136, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- gemini_poll_batch_until_complete, 121
- gemini_poll_batch_until_complete(), 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 130, 133, 136, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- get_prompt_template, 123, 201
- get_prompt_template(), 64, 124, 198, 201, 220, 263
- list_prompt_templates, 123, 124, 201
- list_prompt_templates(), 64, 123, 198, 201, 220, 263
- list_warm_start_models
 - (register_warm_start_model), 199
- list_warm_start_models(), 218
- llm_compare_pair, 125
- llm_compare_pair(), 10, 16, 18, 41, 65, 115, 134, 143, 144, 153, 154, 158, 224, 228, 232, 235, 238, 242, 244, 247, 261, 266, 271
- llm_download_batch_results, 129
- llm_download_batch_results(), 43, 45–48, 53, 59, 63, 117, 119, 121, 122, 133, 134, 136, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- llm_resume_multi_batches, 131
- llm_resume_multi_batches(), 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130, 136, 139, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- llm_submit_pairs_batch, 133

- `llm_submit_pairs_batch()`, 43, 45–48, 53, 59, 63, 80, 117, 119, 121, 122, 129, 130, 133, 140, 160, 162–164, 166, 167, 204, 207, 212, 276
- `llm_submit_pairs_multi_batch`, 138
- `llm_submit_pairs_multi_batch()`, 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130–133, 136, 160, 162–164, 166, 167, 204, 207, 212, 276
- `load_adaptive_session`, 141
- `load_adaptive_session()`, 21, 215, 268
- `load_link_session(save_link_session)`, 216
- `load_warm_start_model`
(`save_warm_start_model`), 217
- `load_warm_start_model()`, 75, 200

- `make_adaptive_judge_llm`, 143
- `make_adaptive_judge_llm()`, 10, 16, 18, 21, 25, 32, 146, 147, 248, 267
- `make_adaptive_judge_replay`, 145
- `make_adaptive_judge_replay()`, 11, 18, 21, 25, 32, 144, 147, 248, 267
- `make_adaptive_replay_reservoir`, 146
- `make_adaptive_replay_reservoir()`, 16, 18, 21, 25, 30, 32, 144–146, 248, 267
- `make_pairs`, 52, 58, 79, 147, 194, 213, 223, 237
- `make_pairs()`, 37, 80, 138, 194, 196, 197, 210, 213, 214, 230, 234, 240, 243
- `make_warm_start_cv_plan`, 149
- `make_warm_start_cv_plan()`, 74, 75, 89, 109, 111, 151, 169, 180, 182, 183, 191, 200, 218, 255, 257, 258, 273–275
- `make_warm_start_prior`, 150
- `make_warm_start_prior()`, 15, 18, 21, 25, 30, 32, 74, 75, 89, 91, 111, 150, 169, 180, 182, 183, 191, 200, 218, 255, 257, 258, 273–275
- `mgcv::ocat()`, 105
- `mgcv::scasm()`, 105

- `ollama_compare_pair_live`, 151
- `ollama_compare_pair_live()`, 41, 65, 77, 115, 125, 126, 128, 158, 224, 228, 232–235, 238, 242, 244, 247, 261, 266, 271
- `openai_compare_pair_live`, 39, 155, 236, 238
- `openai_compare_pair_live()`, 41, 65, 115, 125–128, 154, 224, 228, 232, 235, 238, 242, 244, 247, 259, 261, 266, 271
- `openai_create_batch`, 159
- `openai_create_batch()`, 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130, 133, 136, 140, 162–164, 166, 167, 204, 207, 209–212, 276
- `openai_download_batch_errors`, 161
- `openai_download_batch_errors()`, 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130, 133, 136, 140, 160, 162–164, 166, 167, 204, 207, 211, 212, 276
- `openai_download_batch_output`, 162
- `openai_download_batch_output()`, 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130, 133, 136, 140, 160–162, 164, 166, 167, 204, 207, 209, 212, 276
- `openai_get_batch`, 163
- `openai_get_batch()`, 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130, 133, 137, 140, 160, 162, 163, 165–167, 204, 207, 211, 212, 276
- `openai_poll_batch_until_complete`, 164
- `openai_poll_batch_until_complete()`, 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130, 133, 137, 140, 160, 162–164, 167, 204, 207, 209–212, 276
- `openai_upload_batch_file`, 166
- `openai_upload_batch_file()`, 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130, 133, 137, 140, 160, 162–164, 166, 204, 207, 209, 211, 212, 276
- `ordinal::clm()`, 105

- `pairwiseLLM_warm_model`, 74, 75, 89, 110, 111, 150, 151, 167, 180, 182, 183, 190, 191, 199, 200, 217, 218, 255, 257, 258, 273–275
- `parse_anthropic_batch_output`, 170, 203, 204
- `parse_anthropic_batch_output()`, 132, 172, 174

- `parse_gemini_batch_output`, 172, 205
- `parse_gemini_batch_output()`, 132, 171, 174
- `parse_openai_batch_output`, 155, 173
- `parse_openai_batch_output()`, 171, 172, 209, 211
- `predict.pairwiseLLM_bt_alpha`, 175
- `predict.pairwiseLLM_bt_alpha()`, 51, 54, 57, 98, 100, 177, 178, 219, 250
- `predict.pairwiseLLM_bt_firth`, 176
- `predict.pairwiseLLM_bt_firth()`, 51, 54, 57, 95, 98, 100, 176, 178, 219, 250
- `predict.pairwiseLLM_bt_lapse`, 177
- `predict.pairwiseLLM_bt_lapse()`, 51, 54, 57, 97, 98, 100, 176, 177, 219, 250
- `predict.pairwiseLLM_rubric_calibration`, 178
- `predict.pairwiseLLM_rubric_calibration()`, 82, 83, 107, 190
- `predict.pairwiseLLM_warm_algorithm_ensemble`, 179
- `predict.pairwiseLLM_warm_algorithm_ensemble()`, 74, 75, 89, 111, 150, 151, 169, 182, 183, 191, 200, 218, 255, 257, 258, 273–275
- `predict.pairwiseLLM_warm_ensemble`, 181
- `predict.pairwiseLLM_warm_ensemble()`, 74, 75, 89, 111, 150, 151, 169, 180, 183, 191, 200, 218, 255, 257, 258, 273–275
- `predict.pairwiseLLM_warm_model`, 182
- `predict.pairwiseLLM_warm_model()`, 74, 75, 89, 109, 111, 150, 151, 169, 180, 182, 191, 200, 218, 255, 257, 258, 273–275
- `predict_link`, 184
- `predict_link()`, 103, 185, 188, 217, 222, 255
- `prepare_link_input`, 185
- `prepare_link_input()`, 12, 17, 24, 25, 31, 32, 101, 103, 106, 184, 185, 188–190, 216, 217, 221, 222, 255
- `prepare_linked_rubric_reference`, 189
- `prepare_linked_rubric_reference()`, 83, 91, 104, 106, 107, 179, 186, 216
- `prepare_warm_start_model`, 190
- `prepare_warm_start_model()`, 73–75, 89, 111, 150, 151, 169, 180, 182, 183, 200, 217, 218, 255, 257, 258, 273–275
- `print.adaptive_state`, 192
- `print.pairwiseLLM_cost_estimate`, 193
- `print.pairwiseLLM_link_result`
(`summary.pairwiseLLM_link_result`), 254
- `print.pairwiseLLM_link_session`
(`summary.pairwiseLLM_link_result`), 254
- `print.pairwiseLLM_warm_algorithm_ensemble`
(`summary.pairwiseLLM_warm_algorithm_ensemble`), 255
- `print.pairwiseLLM_warm_ensemble`
(`summary.pairwiseLLM_warm_ensemble`), 256
- `print.pairwiseLLM_warm_model`
(`pairwiseLLM_warm_model`), 167
- `print.pairwiseLLM_warm_predictions`
(`summary.pairwiseLLM_warm_predictions`), 258
- `randomize_pair_order`, 52, 58, 79, 194, 223, 237
- `randomize_pair_order()`, 37, 138, 148, 196, 197, 210, 213, 214, 230, 234, 240, 243
- `read_samples_df`, 195
- `read_samples_df()`, 9, 10, 37, 148, 194, 197, 213, 214
- `read_samples_dir`, 196
- `read_samples_dir()`, 37, 148, 194, 196, 213, 214
- `readr::read_csv()`, 139
- `readRDS()`, 103, 186, 189
- `register_prompt_template`, 123, 197, 201
- `register_prompt_template()`, 64, 123, 124, 201, 220, 263
- `register_warm_start_model`, 199
- `register_warm_start_model()`, 74, 75, 89, 111, 150, 151, 169, 180, 182, 183, 191, 217, 218, 255, 257, 258, 273–275
- `reliability`, 99
- `remove_prompt_template`, 123, 201
- `remove_prompt_template()`, 64, 123, 124, 198, 220, 263
- `remove_warm_start_model`
(`register_warm_start_model`), 199

- resume_link_session
 - (start_link_session), 221
- resume_link_session(), 216
- run_anthropic_batch_pipeline, 42, 202, 207
- run_anthropic_batch_pipeline(), 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130, 133, 134, 137–140, 160, 162–164, 166, 167, 207, 212, 276
- run_gemini_batch_pipeline, 116, 205
- run_gemini_batch_pipeline(), 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130, 133, 134, 137, 139, 140, 160, 162–164, 166, 167, 204, 212, 276
- run_openai_batch_pipeline, 202–204, 207, 209
- run_openai_batch_pipeline(), 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130, 133, 137–140, 160, 162–164, 166, 167, 204, 207, 276
- sample_pairs, 52, 58, 79, 194, 213, 223, 237
- sample_pairs(), 37, 138, 148, 194, 196, 197, 210, 214, 230, 234, 240, 243
- sample_reverse_pairs, 194, 214
- sample_reverse_pairs(), 37, 148, 194, 196, 197, 213
- save_adaptive_session, 215
- save_adaptive_session(), 21, 31, 142, 268
- save_link_session, 216
- save_link_session(), 103, 185, 187, 188, 190, 215, 217, 222, 255
- save_warm_start_model, 217
- save_warm_start_model(), 74, 75, 89, 111, 150, 151, 169, 180, 182, 183, 191, 200, 255, 257, 258, 273–275
- saveRDS(), 103, 149, 189
- scale_separation_reliability, 219
- scale_separation_reliability(), 51, 54, 57, 94, 95, 98, 100, 176–178, 250
- set_prompt_template, 38, 52, 58, 64, 79, 123, 202, 220, 223, 237
- set_prompt_template(), 10, 64, 113, 123, 124, 126, 135, 143, 152, 158, 198, 201, 210, 226, 230, 234, 241, 246, 260, 263, 270
- start_link_session, 221
- start_link_session(), 12, 17, 24, 25, 31, 32, 103, 106, 185, 188, 216, 217, 222, 255
- stats::quantile(), 106
- submit_anthropic_pairs_live, 222
- submit_anthropic_pairs_live(), 41, 65, 115, 128, 154, 158, 228, 231, 232, 235, 238, 242, 244, 247, 261, 266, 271
- submit_gemini_pairs_live, 225
- submit_gemini_pairs_live(), 41, 65, 115, 128, 154, 158, 224, 231, 232, 235, 238, 242, 244, 247, 261, 266, 271
- submit_llm_pairs, 79, 229
- submit_llm_pairs(), 41, 65, 70, 72, 80, 115, 127, 128, 134, 154, 158, 224, 228, 235, 238, 242–244, 247, 261, 266, 271
- submit_ollama_pairs_live, 233
- submit_ollama_pairs_live(), 41, 65, 77, 115, 128, 154, 158, 224, 228, 231, 232, 238, 242, 244, 247, 261, 266, 271
- submit_openai_pairs_live, 236
- submit_openai_pairs_live(), 41, 65, 70, 115, 128, 154, 158, 224, 228, 230–232, 235, 242, 244, 247, 261, 266, 271
- submit_together_pairs_live, 240
- submit_together_pairs_live(), 41, 65, 115, 128, 154, 158, 224, 228, 231, 232, 235, 238, 244, 247, 261, 266, 271
- submit_typesafe_pairs_live, 243
- submit_typesafe_pairs_live(), 41, 65, 70, 115, 128, 154, 159, 224, 228, 231, 232, 235, 238, 242, 247, 261, 265, 266, 271
- submit_vertex_pairs_live, 245
- submit_vertex_pairs_live(), 41, 65, 115, 128, 154, 159, 224, 228, 231, 232, 235, 238, 242, 244, 261, 266, 271
- summarize_adaptive, 247
- summarize_adaptive(), 11, 18, 21, 25, 32, 142, 144, 146, 147, 192, 267
- summarize_bt_fit, 249
- summarize_bt_fit(), 51, 54, 57, 98, 100, 176–178, 219
- summarize_items, 250
- summarize_items(), 7, 8, 18, 33, 35, 36, 89,

- [91, 253–255](#)
- [summarize_refits, 252](#)
- [summarize_refits\(\), 7, 8, 18, 33, 35, 36, 89, 251, 255](#)
- [summary.pairwiseLLM_link_result, 254](#)
- [summary.pairwiseLLM_link_result\(\), 103, 185, 188, 217, 222](#)
- [summary.pairwiseLLM_link_session \(summary.pairwiseLLM_link_result\), 254](#)
- [summary.pairwiseLLM_warm_algorithm_ensemble, 255](#)
- [summary.pairwiseLLM_warm_algorithm_ensemble\(\), 74, 75, 89, 111, 150, 151, 169, 180, 182, 183, 191, 200, 218, 257, 258, 273–275](#)
- [summary.pairwiseLLM_warm_ensemble, 256](#)
- [summary.pairwiseLLM_warm_ensemble\(\), 74, 75, 89, 111, 150, 151, 169, 180, 182, 183, 191, 200, 218, 255, 258, 273–275](#)
- [summary.pairwiseLLM_warm_model \(pairwiseLLM_warm_model\), 167](#)
- [summary.pairwiseLLM_warm_predictions, 258](#)
- [summary.pairwiseLLM_warm_predictions\(\), 74, 75, 89, 111, 150, 151, 169, 180, 182, 183, 191, 200, 218, 255, 257, 273–275](#)
- [together_compare_pair_live, 259](#)
- [together_compare_pair_live\(\), 41, 65, 115, 125, 128, 154, 159, 224, 228, 232, 235, 238, 240–242, 244, 247, 266, 271](#)
- [trait_description, 262](#)
- [trait_description\(\), 64, 123, 124, 198, 201, 220](#)
- [typesafe_compare_pair_live, 263](#)
- [typesafe_compare_pair_live\(\), 41, 65, 115, 125, 127, 128, 154, 159, 224, 228, 232, 235, 238, 242–244, 247, 261, 271](#)
- [validate_adaptive_replay, 267](#)
- [validate_adaptive_replay\(\), 18, 21, 25, 32, 144–147, 248](#)
- [validate_session_dir, 268](#)
- [validate_session_dir\(\), 142, 215](#)
- [vertex_compare_pair_live, 269](#)
- [vertex_compare_pair_live\(\), 41, 65, 115, 125, 128, 154, 159, 224, 228, 232, 235, 238, 242, 244–247, 261, 266](#)
- [warm_start_coefficients, 271](#)
- [warm_start_coefficients\(\), 74, 75, 89, 111, 150, 151, 169, 180, 182, 183, 191, 200, 218, 255, 257, 258, 274, 275](#)
- [warm_start_feature_schema, 273](#)
- [warm_start_feature_schema\(\), 74, 75, 89, 111, 150, 151, 169, 180, 182, 183, 191, 200, 218, 255, 257, 258, 273, 275](#)
- [warm_start_python_status, 275](#)
- [warm_start_python_status\(\), 74, 75, 88, 89, 111, 150, 151, 169, 180, 182, 183, 191, 200, 218, 255, 257, 258, 273, 274](#)
- [write_openai_batch_file, 276](#)
- [write_openai_batch_file\(\), 43, 45, 46, 48, 53, 59, 63, 117, 119, 121, 122, 130, 133, 137, 140, 160, 162–164, 166, 167, 204, 207, 209, 212](#)