Package 'LLMR'

July 24, 2026

Title Interface for Large Language Model APIs in R

Version 0.8.11

Depends R (>= 4.1.0)

LazyData true

Description One interface to many large language model providers: a single configuration object and a single calling function serve chat and embedding models alike, so research code does not change when the provider does. The same calls scale to multi-model and multi-condition studies.

License MIT + file LICENSE

Encoding UTF-8

Imports httr2 (>= 1.0.0), curl, purrr, dplyr, tidyr, rlang, memoise, future, future.apply, tibble, base64enc, mime, glue (>= 1.6.0), cli (>= 3.6.0), jsonlite, vctrs, digest, tidyselect

Suggests testthat (>= 3.0.0), roxygen2 (>= 7.1.2), progressr, knitr, rmarkdown, ggplot2, jsonvalidate, stringi, kableExtra, withr

RoxygenNote 7.3.3

Config/testthat/edition 3

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

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

VignetteBuilder knitr

NeedsCompilation no

Author Ali Sanaei [aut, cre]

Maintainer Ali Sanaei <sanaei@uchicago.edu>

Repository CRAN

Date/Publication 2026-07-24 18:40:02 UTC

Contents

`anes_2024_personas` *Example participant profiles derived from ANES 2024*

Description

A ready-to-use set of 100 participant profiles, used across the LLMR family to build example synthetic panels and discussions without downloading anything. Each row is one respondent from the American National Election Studies (ANES) 2024 Time Series Study, with demographics and a broad battery of political and social attitudes decoded to their value labels. It is an ordinary data frame: select columns and filter rows with dplyr or base R, then hand the result to a consumer (for example `FocusGroup::create_agents_from_data()` or `LLMRpanel::panel_from_personas()`). The helpers `llm_persona_split()`, `llm_persona_overview()`, and `llm_persona_demographic_fields()` read the frame.

Usage

```
anes_2024_personas
```

Format

A data frame with 100 rows and 125 columns. Column names are short, tidy-select-friendly handles. Demographics use a `demo_` prefix (`demo_age`, `demo_race_ethnicity`, ...). Attitudes use an `att_<block>_` prefix grouping them into a few blocks, so `dplyr::select(starts_with("att_iss_"))` grabs a whole block:

- `att_id_` – political identity (party id, ideology, vote)

- `att_aff_` – feeling thermometers (candidates, parties, groups)
- `att_eval_` – approval, the economy, personal finances, national mood
- `att_iss_` – issue positions (taxes, immigration, guns, climate, health care, abortion, crime, race, foreign policy, trade, ...)
- `att_val_` – trust, social trust, values, democracy

One numeric column, `ideology_score`, is a single conservative–liberal dimension (see Details); the rows are sorted by it. Attitude values are character labels (`NA` when missing or not applicable). The data frame carries a `"dictionary"` attribute (a data frame mapping each handle to its question wording, ANES variable code, and block) and a `"demographic_fields"` attribute marking the demographic columns.

Details

The 100 respondents were chosen by diversity sampling (a greedy maximin pass over a Gower distance, after dropping respondents missing more than a quarter of the fields), so the set spans the range of demographic and attitudinal profiles rather than reproducing population proportions. It is example material; it is NOT a representative sample of the United States, carries no survey weights, and should not be used for population inference. Each respondent's own bundle of answers is kept intact (answers are not shuffled across people), which is what makes a profile read as a coherent person. Demographics are coarsened (age in bands, broad income and race categories, census region rather than state); no respondent identifiers, granular geography, open-ended text, or restricted-use variables are included. Items concerning sexual orientation and gender identity are excluded by design.

`ideology_score` is a unidimensional ideal point estimated from a graded- response item-response model over the ordinal attitude items, standardized to roughly mean 0 and unit scale, and oriented so that low is liberal and high is conservative (it agrees with the first principal component at r ~= 0.99).

The reproducibility scripts are under `inst/anes-data-prep/` in the installed package; the raw ANES file is not bundled (it is freely available from ANES).

Source

Derived from the American National Election Studies. 2025. *ANES 2024 Time Series Study Full Release* [dataset and documentation]. August 8, 2025. https://electionstudies.org/data-center/2024-time-series-study/. A derived product of the ANES public release, distributed for example use; it contains no ANES respondent identifiers and no restricted-use data. Work that uses these personas should cite ANES as above. The ANES bears no responsibility for the analyses or interpretations presented here.

Examples

```
data(anes_2024_personas, package = "LLMR")
nrow(anes_2024_personas)
# the data frame is the interface: filter rows, select columns

  if (requireNamespace("dplyr", quietly = TRUE)) {
    library(dplyr)
    conservative <- dplyr::filter(anes_2024_personas, ideology_score > 0.5)
```

```
  nrow(conservative)
}
```

`build_factorial_experiments` *Build Factorial Experiment Design*

Description

Creates a tibble of experiments for factorial designs where you want to test all combinations of configs, messages, and repetitions with automatic metadata.

Usage

```
build_factorial_experiments(
  configs,
  user_prompts,
  system_prompts = NULL,
  repetitions = 1,
  config_labels = NULL,
  user_prompt_labels = NULL,
  system_prompt_labels = NULL
)
```

Arguments

`configs`	List of llm_config objects to test.
`user_prompts`	Character vector (or list) of user-turn prompts.
`system_prompts`	Optional character vector of system messages. These are fully crossed with the user prompts (every combination appears), like the other factors. Missing/NA values are ignored; those messages are user-only.
`repetitions`	Integer. Number of repetitions per combination. Default is 1.
`config_labels`	Character vector of labels for configs. If NULL, uses "provider_model".
`user_prompt_labels`	Optional labels for the user prompts.
`system_prompt_labels`	Optional labels for the system prompts.

Value

A tibble with columns: config (list-column), messages (list-column), config_label, user_prompt_label, system_prompt_label, and repetition. Ready for use with call_llm_par().

Examples

```
## Not run:
  # Factorial design: 3 configs x 2 user prompts x 10 reps = 60 experiments
  configs <- list(gpt4_config, claude_config, llama_config)
  user_prompts <- c("Control prompt", "Treatment prompt")

  experiments <- build_factorial_experiments(
    configs = configs,
    user_prompts = user_prompts,
    repetitions = 10,
    config_labels = c("gpt4", "claude", "llama"),
    user_prompt_labels = c("control", "treatment")
  )

  # Use with call_llm_par
  results <- call_llm_par(experiments, progress = TRUE)

## End(Not run)
```

`call_llm` *Call an LLM (chat/completions or embeddings) with optional multimodal input*

Description

`call_llm()` dispatches to the correct provider implementation based on `config$provider`. It supports both generative chat/completions and embeddings, plus a simple multimodal shortcut for local files.

Usage

```
call_llm(config, messages, verbose = FALSE)

## S3 method for class 'ollama'
call_llm(config, messages, verbose = FALSE)
```

Arguments

`config`	An `llm_config` object.
`messages`	One of: • Plain character vector - each element becomes a `"user"` message. • Named character vector - names are roles (`"system"`, `"user"`, `"assistant"`). **Multimodal shortcut:** include one or more elements named `"file"` whose values are local paths; consecutive `{user | file}` entries are combined into one user turn and files are inlined (base64) for capable providers. • List of message objects: `list(role=..., content=...)`. For multimodal content, set `content` to a list of parts like `list(list(type="text", text="..."), list(type="file", path="..."))`.
`verbose`	Logical. If `TRUE`, prints the full parsed API response.

Value

- Generative mode: an `llmr_response` object. Use `as.character(x)` to get just the text; `print(x)` shows text plus a status line; use helpers `finish_reason(x)` and `tokens(x)`.
- Embedding mode: provider-native list with an element `data`; convert with `parse_embeddings()`.

Provider notes

- **OpenAI-compatible:** On a server 400 that identifies the bad parameter as `max_tokens`, LLMR will, unless `no_change=TRUE`, retry once replacing `max_tokens` with `max_completion_tokens` (and inform via a `cli_alert_info`). The former experimental "uncapped retry on empty content" is *disabled* by default to avoid unexpected costs.
- **Anthropic:** `max_tokens` is required; if omitted LLMR uses `2048` and warns. Multimodal images are inlined as base64 and PDFs as document blocks. Extended thinking is supported: provide `thinking_budget` (which must stay below `max_tokens`) and the response will carry `content` blocks of type `"thinking"`, also exposed as the `thinking` field of the result. Beta features can be requested by passing `anthropic_beta = "..."`, sent as the `anthropic-beta` header.
- **Gemini (REST):** `systemInstruction` is supported; user parts use `text`/`inlineData(mimeType,data)`; responses are set to `responseMimeType = "text/plain"`. For Vertex AI, use `provider = "gemini", vertex = TRUE,`
- **Ollama (local):** OpenAI-compatible endpoints on `http://localhost:11434/v1/*`; no Authorization header is required. Override with `api_url` as needed.
- **Alibaba / Moonshot regions:** Defaults target the *international* endpoints (`dashscope-intl.aliyuncs.com` and `api.moonshot.ai`). China-region accounts must pass `api_url` for the mainland hosts (`dashscope.aliyuncs.com` and `api.moonshot.cn`); using the wrong region returns HTTP 401.
- **OpenRouter:** one key (`OPENROUTER_API_KEY`) reaches many hosted models; its optional attribution headers (`HTTP-Referer`, `X-Title`) can be added through the `req_builder` hook of `llm_config()`. No embeddings or batch API.
- **Error handling:** HTTP errors raise structured conditions with classes like `llmr_api_param_error`, `llmr_api_rate_limit_error`, `llmr_api_server_error`; see the condition fields for status, code, request id, and (where supplied) the offending parameter.

Message normalization

See the *"multimodal shortcut"* described under `messages`. Internally, LLMR expands these into the provider's native request shape and tilde-expands local file paths.

Using a local Ollama server

Ollama provides an OpenAI-compatible HTTP API on localhost by default. Start the daemon and pull a model first (terminal): `ollama serve` (in background) and `ollama pull llama3`. Then configure LLMR with `llm_config("ollama", "llama3", embedding = FALSE)` for chat or `llm_config("ollama", "nomic-embed-text", embedding = TRUE)` for embeddings. Override the endpoint with `api_url` if not using the default `http://localhost:11434/v1/*`.

See Also

`llm_config`, `call_llm_robust`, `llm_chat_session`, `parse_embeddings`, `finish_reason`, `tokens`

Examples

```
## Not run:
## 1) Basic generative call
cfg <- llm_config("openai", "gpt-5-nano")
call_llm(cfg, "Say hello in Greek.")

## 2) Generative with rich return
r <- call_llm(cfg, "Say hello in Greek.")
r
as.character(r)
finish_reason(r); tokens(r)

## 3) Anthropic extended thinking (single example)
## max_tokens must cover the thinking budget plus the visible reply.
a_cfg <- llm_config("anthropic", "claude-sonnet-4-6",
                    max_tokens = 20000,
                    thinking_budget = 16000)
r2 <- call_llm(a_cfg, "Compute 87*93 in your head. Give only the final number.")
# reasoning text: r2$thinking
# final text:     as.character(r2)

## 4) Multimodal (named-vector shortcut)
msg <- c(
  system = "Answer briefly.",
  user   = "Describe this image in one sentence.",
  file   = "~/Pictures/example.png"
)
call_llm(cfg, msg)

## 5) Embeddings
e_cfg <- llm_config("voyage", "voyage-3.5-lite",
                    embedding = TRUE)
emb_raw <- call_llm(e_cfg, c("first", "second"))
emb_mat <- parse_embeddings(emb_raw)

## 6) With a chat session
ch <- chat_session(cfg)
ch$send("Say hello in Greek.")   # prints the same status line as `print.llmr_response`
ch$history()

## End(Not run)
```

`call_llm_broadcast` *Parallel API calls: Fixed Config, Multiple Messages*

Description

Broadcasts different messages using the same configuration in parallel. Perfect for batch processing different prompts with consistent settings. Use `setup_llm_parallel()` when you want explicit control over workers.

Usage

```
call_llm_broadcast(config, messages, ...)
```

Arguments

config	Single llm_config object to use for all calls.
messages	A character vector (each element is a prompt) OR a list where each element is a pre-formatted message list.
...	Additional arguments passed to call_llm_par (e.g., tries, verbose, progress).

Value

An llmr_experiment tibble with columns: message_index (metadata), the config list-column (so llm_par_resume() can re-run failures), provider, model, all model parameters, response_text, raw_response_json, success, error_message, and the other diagnostics documented in call_llm_par().

Parallel Workflow

Recommended workflow:

1. Call setup_llm_parallel() once at the start of your script.
2. Run one or more parallel experiments (e.g., call_llm_broadcast()).
3. Call reset_llm_parallel() at the end to restore sequential processing. If the active future plan is sequential, call_llm_par() temporarily switches to multisession for the duration of the call.

See Also

setup_llm_parallel, reset_llm_parallel, call_llm_par, llm_fn, llm_mutate

Examples

```
## Not run:
  # Broadcast different questions
  config <- llm_config(provider = "groq", model = "openai/gpt-oss-20b")

  messages <- list(
    list(list(role = "user", content = "What is 2+2?")),
    list(list(role = "user", content = "What is 3*5?")),
    list(list(role = "user", content = "What is 10/2?"))
  )

  setup_llm_parallel(workers = 4, verbose = TRUE)
  results <- call_llm_broadcast(config, messages)
  reset_llm_parallel(verbose = TRUE)

## End(Not run)
```

`call_llm_compare` *Parallel API calls: Multiple Configs, Fixed Message*

Description

Compares different configurations (models, providers, settings) using the same message. Perfect for benchmarking across different models or providers. Use `setup_llm_parallel()` when you want explicit control over workers.

Usage

```
call_llm_compare(configs_list, messages, ...)
```

Arguments

`configs_list`	A list of llm_config objects to compare.
`messages`	A character vector or a list of message objects (same for all configs).
`...`	Additional arguments passed to `call_llm_par` (e.g., tries, verbose, progress).

Value

An `llmr_experiment` tibble with columns: config_index (metadata), the `config` list-column (so `llm_par_resume()` can re-run failures), provider, model, all varying model parameters, response_text, raw_response_json, success, error_message, and the other diagnostics documented in `call_llm_par()`.

Parallel Workflow

Recommended workflow:

1. Call `setup_llm_parallel()` once at the start of your script.
2. Run one or more parallel experiments (e.g., `call_llm_broadcast()`).
3. Call `reset_llm_parallel()` at the end to restore sequential processing. If the active future plan is sequential, `call_llm_par()` temporarily switches to `multisession` for the duration of the call.

See Also

`setup_llm_parallel`, `reset_llm_parallel`, `call_llm_par`

Examples

```
## Not run:
  # Compare different models
  config1 <- llm_config(provider = "openai", model = "gpt-5-nano")
  config2 <- llm_config(provider = "groq", model = "openai/gpt-oss-20b")

  configs_list <- list(config1, config2)
```

```
  messages <- "Explain quantum computing"

  setup_llm_parallel(workers = 4, verbose = TRUE)
  results <- call_llm_compare(configs_list, messages)
  reset_llm_parallel(verbose = TRUE)

## End(Not run)
```

`call_llm_par` *Parallel LLM Processing with Tibble-Based Experiments (Core Engine)*

Description

Processes experiments from a tibble where each row contains a config and message pair. This is the core parallel processing function. Metadata columns are preserved. Use `setup_llm_parallel()` when you want explicit control over workers.

Usage

```
call_llm_par(
  experiments,
  simplify = TRUE,
  tries = 10,
  wait_seconds = 2,
  backoff_factor = 120^(1/tries),
  verbose = FALSE,
  memoize = FALSE,
  max_workers = NULL,
  progress = FALSE,
  start_jitter = 0,
  .request_hash = FALSE
)
```

Arguments

`experiments`	A tibble/data.frame with required list-columns 'config' (llm_config objects) and 'messages' (character vector OR message list).
`simplify`	If TRUE (default), provider, model, and the model parameters stored in each row's config are unnested into regular columns for easy filtering and grouping.
`tries`	Integer. Total number of attempts per call (first call plus retries). Default is 10.
`wait_seconds`	Numeric. Initial wait time (seconds) before retry. Default is 2.
`backoff_factor`	Numeric. Multiplier for wait time after each failure. Default is 3.
`verbose`	Logical. If TRUE, prints progress and debug information.
`memoize`	Logical. If TRUE, enables caching for identical requests. Note that under a multisession plan each worker process keeps its own cache, so deduplication is per worker, not global.

max_workers	Integer. Maximum number of parallel workers. If NULL, auto-detects.
progress	Logical. If TRUE, shows progress bar.
start_jitter	Each call starts after a uniformly distributed delay between 0 and start_jitter seconds. The default is 0 (no delay); set a few seconds when launching very large runs against a provider with strict burst limits.
.request_hash	Logical. If TRUE, append a request_hash column (the same key llm_request_hash() produces) so the results can be joined to the audit log. Default FALSE; the column is omitted unless requested.

Value

An llmr_experiment tibble containing all original columns plus:

- response_text - assistant text (or NA on failure)
- raw_response_json - raw JSON string (on failure: the provider's error body when available)
- success, error_message
- finish_reason - e.g. "stop", "length", "filter", "tool", or "error:category"
- sent_tokens, rec_tokens, total_tokens, reasoning_tokens
- response_id
- duration - seconds
- status_code, error_code, bad_param - error diagnostics (NA on success)
- response - the full llmr_response object (or NULL on failure)

The response column holds llmr_response objects on success, or NULL on failure.

Parallel Workflow

Recommended workflow:

1. Call setup_llm_parallel() once at the start of your script.
2. Run one or more parallel experiments (e.g., call_llm_broadcast()).
3. Call reset_llm_parallel() at the end to restore sequential processing. If the active future plan is sequential, this function temporarily switches to multisession for the duration of the call.

See Also

For setting up the environment: setup_llm_parallel, reset_llm_parallel. For simpler, pre-configured parallel tasks: call_llm_broadcast, call_llm_sweep, call_llm_compare. For creating experiment designs: build_factorial_experiments.

Examples

```
## Not run:
# Simple example: Compare two models on one prompt
cfg1 <- llm_config("openai", "gpt-4.1-nano")
cfg2 <- llm_config("groq", "openai/gpt-oss-20b")

experiments <- tibble::tibble(
  model_id = c("gpt-4.1-nano", "groq-gpt-oss-20b"),
  config = list(cfg1, cfg2),
  messages = "Count the number of the letter e in this word: Freundschaftsbeziehungen "
)

setup_llm_parallel(workers = 2)
results <- call_llm_par(experiments, progress = TRUE)
reset_llm_parallel()

print(results[, c("model_id", "response_text")])


## End(Not run)
```

`call_llm_par_structured` *Parallel experiments with structured parsing*

Description

Enables structured output on each config (if not already set), runs, then parses JSON.

Usage

```
call_llm_par_structured(experiments, schema = NULL, .fields = NULL, ...)
```

Arguments

`experiments`	Tibble with `config` and `messages` list-columns.
`schema`	Optional JSON Schema list.
`.fields`	Optional fields to hoist from parsed JSON (supports nested paths).
`...`	Passed to `call_llm_par()`.

Value

A tibble of class `llmr_experiment`: the `call_llm_par()` result with structured output parsed by `llm_parse_structured_col()` (adds `structured_ok`, `structured_data`, and one column per hoisted field).

See Also

`call_llm_par()`, `llm_parse_structured_col()`, `enable_structured_output()`

call_llm_par_tags *Parallel experiments with tag parsing*

Description

Injects tag instructions into each experiment row, runs `call_llm_par()`, then parses XML-like tags from each response via `llm_parse_tags_col()`.

Usage

```
call_llm_par_tags(experiments, .tags, .fields = NULL, ...)
```

Arguments

`experiments`	Tibble with `config` and `messages` list-columns.
`.tags`	Character vector of tag names to request and parse.
`.fields`	`NULL` to extract all tags, a character vector of tags, a named vector such as `c(person_age = "age")`, or `FALSE` to skip field extraction.
`...`	Passed to `call_llm_par()`.

Value

A tibble of class `llmr_experiment`: the `call_llm_par()` result with XML-like tags parsed by `llm_parse_tags_col()` (adds `tags_ok`, `tags_data`, and one column per requested tag or field).

See Also

`call_llm_par()`, `llm_parse_tags_col()`, `llm_fn_tags()`, `llm_mutate_tags()`

call_llm_robust *Robustly Call LLM API (Simple Retry)*

Description

Wraps `call_llm` so that transient failures are retried while permanent ones fail fast. Retried conditions are rate limits (HTTP 429), server errors (HTTP 5xx and 408), and network-level interruptions (timeouts, connection resets, DNS failures). Errors that retrying cannot fix, such as an invalid parameter (400), a missing key (401/403), or a prompt that exceeds the context window, are raised immediately.

Usage

```
call_llm_robust(
  config,
  messages,
  tries = 5,
  wait_seconds = 2,
  backoff_factor = 3,
  verbose = FALSE,
  memoize = FALSE
)
```

Arguments

config	An llm_config object from llm_config.
messages	A list of message objects (or character vector for embeddings).
tries	Integer. Total number of attempts (the first call plus retries) before giving up. Default is 5.
wait_seconds	Numeric. Initial wait time (seconds) before the first retry. Default is 2.
backoff_factor	Numeric. Multiplier for wait time after each failure. Default is 3.
verbose	Logical. If TRUE, prints the full API response.
memoize	Logical. If TRUE, calls are cached to avoid repeated identical requests. Default is FALSE.

Details

When the provider supplies a Retry-After header with a 429, the wait honors it; otherwise waits grow exponentially with a little jitter so that parallel workers do not retry in lockstep.

Value

The successful result from call_llm, or an error if all retries fail.

See Also

call_llm for the underlying, non-robust API call. llm_config to create the configuration object. chat_session for stateful, interactive conversations.

Examples

```
## Not run:
robust_resp <- call_llm_robust(
config = llm_config("groq", "openai/gpt-oss-20b"),
messages = list(list(role = "user", content = "Hello, LLM!")),
tries = 5,
wait_seconds = 2,
memoize = FALSE
)
print(robust_resp)
```

```
as.character(robust_resp)

## End(Not run)
```

`call_llm_stream` *Stream a chat completion token by token*

Description

Like `call_llm()`, but the reply arrives incrementally: `callback` is invoked with each text chunk as it is generated, and the complete llmr_response is returned at the end. Streaming keeps long generations responsive and avoids HTTP timeouts on slow, lengthy completions.

Usage

```
call_llm_stream(
  config,
  messages,
  callback = function(chunk) cat(chunk),
  verbose = FALSE
)
```

Arguments

`config`	An llm_config for a generative model.
`messages`	Messages as in `call_llm()`.
`callback`	Function called with each text chunk (a character scalar) as it arrives. The default prints chunks to the console with `cat()`. Reasoning deltas (when a provider streams them separately) are not passed to `callback`; they are collected into the result's `thinking` field.
`verbose`	Print the assembled response object at the end.

Details

Supported providers: all OpenAI-compatible chat APIs (openai, groq, together, deepseek, xai, alibaba, zhipu, moonshot, xiaomi, ollama), Anthropic, and Gemini. The request body is built by the same internals as `call_llm()`, so parameters and structured output behave identically; the `request_modifier` and `req_builder` hooks and the request timeout apply as well. Only the transport differs. The `response_modifier` hook is the one exception: it rewrites a full parsed response, which has no per-chunk meaning for a stream, so it is not applied here.

Value

An llmr_response assembled from the stream (invisibly). Token usage is filled when the provider reports it in the stream; otherwise it is `NA`.

See Also

call_llm(), chat_session()

Examples

```
## Not run:
cfg <- llm_config("groq", "openai/gpt-oss-20b")
r <- call_llm_stream(cfg, "Tell a 100-word story about a lighthouse.")
tokens(r)

## End(Not run)
```

---

call_llm_sweep *Parallel API calls: Parameter Sweep - Vary One Parameter, Fixed Message*

Description

Sweeps through different values of a single parameter while keeping the message constant. Perfect for hyperparameter tuning, temperature experiments, etc. Use setup_llm_parallel() when you want explicit control over workers.

Usage

```
call_llm_sweep(base_config, param_name, param_values, messages, ...)
```

Arguments

base_config	Base llm_config object to modify.
param_name	Character. Name of the parameter to vary (e.g., "temperature", "max_tokens").
param_values	Vector. Values to test for the parameter.
messages	A character vector or a list of message objects (same for all calls).
...	Additional arguments passed to call_llm_par (e.g., tries, verbose, progress).

Value

An llmr_experiment tibble with columns: swept_param_name, the varied parameter column, the config list-column (so llm_par_resume() can re-run failures), provider, model, all other model parameters, response_text, raw_response_json, success, error_message, and the other diagnostics documented in call_llm_par().

Parallel Workflow

Recommended workflow:

1. Call `setup_llm_parallel()` once at the start of your script.
2. Run one or more parallel experiments (e.g., `call_llm_broadcast()`).
3. Call `reset_llm_parallel()` at the end to restore sequential processing. If the active future plan is sequential, `call_llm_par()` temporarily switches to `multisession` for the duration of the call.

See Also

`setup_llm_parallel`, `reset_llm_parallel`, `call_llm_par`

Examples

```
## Not run:
  # Temperature sweep
  config <- llm_config(provider = "groq", model = "openai/gpt-oss-20b")

  messages <- "What is 15 * 23?"
  temperatures <- c(0, 0.3, 0.7, 1.0, 1.5)

  setup_llm_parallel(workers = 4, verbose = TRUE)
  results <- call_llm_sweep(config, "temperature", temperatures, messages)
  results |> dplyr::select(temperature, response_text)
  reset_llm_parallel(verbose = TRUE)

## End(Not run)
```

`call_llm_tools` *Call an LLM with tools and run the tool loop*

Description

Sends `messages` together with native tool definitions, executes every tool the model calls, feeds the results back, and repeats until the model answers in plain text (or `max_rounds` is reached). Supported for OpenAI-compatible providers (openai, groq, together, deepseek, xai, alibaba, zhipu, moonshot, xiaomi, ollama) and Anthropic.

Usage

```
call_llm_tools(
  config,
  messages,
  tools,
  max_rounds = 8L,
  max_tool_calls = Inf,
  verbose = FALSE,
```

```
  tries = 3L,
  wait_seconds = 2
)
```

Arguments

config	An llm_config for a generative model.
messages	Messages as in call_llm().
tools	One llm_tool() or a list of them.
max_rounds	Maximum model turns (a turn may contain several tool calls). When reached, the last response is returned as-is with a warning.
max_tool_calls	Maximum tool executions across the whole loop. Exceeding it raises a condition of class llmr_tool_limit rather than continuing to spend; the default is unlimited. Callers enforcing spend ceilings (e.g. agent frameworks) can catch that class.
verbose	Print each tool invocation as it happens.
tries, wait_seconds	Retry controls passed to call_llm_robust().

Value

The final llmr_response. Its messages field contains the full conversation, including tool results; tool_history is a tibble of executed calls with columns round, name, arguments (JSON), and result; and tool_loop is a list with model_calls, sent, rec (token totals over every internal model call, NA when the provider reported none), and tool_calls. The same values remain available as attributes for backward compatibility. Note that tokens(x) alone covers only the final model call.

See Also

llm_tool(), tool_calls(), call_llm()

Examples

```
## Not run:
weather <- llm_tool(
  function(city) paste0("22C and clear in ", city),
  name = "get_weather",
  description = "Current weather for a city.",
  parameters = list(city = list(type = "string"))
)
cfg <- llm_config("groq", "openai/gpt-oss-20b", temperature = 0)
r <- call_llm_tools(cfg, "What is the weather in Tunis?", tools = weather)
as.character(r)
r$tool_history

## End(Not run)
```

`diagnostics` *Machine-readable diagnostics for an LLMR-family result object*

Description

A shared generic across the LLMR method packages. It returns the small, machine-readable set of health numbers behind a result object (the part you would assert in a test or drop into a table), as distinct from `report()`, which drafts methods-section prose.

Usage

```
diagnostics(x, ...)
```

Arguments

`x`	An LLMR experiment or an object returned by an LLMR method package.
`...`	Passed to methods.

Details

LLMR defines the generic and an erroring default only. The method packages (LLMRcontent, LLMRpanel) provide the implementations, each returning a tibble of the key numbers for its own result classes.

Value

A method-defined object, by convention a tibble of diagnostic values.

See Also

`report()`

Examples

```
## Not run:
# LLMRcontent, for instance, returns one row of stability and fragility numbers:
diagnostics(audit)

## End(Not run)
```

disable_structured_output

Disable Structured Output (clean provider toggles)

Description

Removes response_format/response_schema/response_mime_type and schema tool if present. Keeps user tools intact.

Usage

```
disable_structured_output(config)
```

Arguments

config	llm_config

Value

The `llm_config` with the structured-output request fields removed.

See Also

`enable_structured_output()`

enable_structured_output

Enable Structured Output (Provider-Agnostic)

Description

Turn on structured output for a model configuration. Supports OpenAI-compatible providers (OpenAI, Groq, Together, x.ai, DeepSeek, Xiaomi, Alibaba (Qwen), Zhipu, Moonshot, OpenRouter), Anthropic, and Gemini.

Usage

```
enable_structured_output(
  config,
  schema = NULL,
  name = "llmr_schema",
  strict = TRUE
)
```

Arguments

config	An llm_config object.
schema	A named list representing a JSON Schema. If NULL, OpenAI-compatible providers enforce a JSON object; Gemini switches to JSON mime type; Anthropic only injects a tool when a schema is supplied.
name	Character. Schema/tool name for providers requiring one. Default "llmr_schema".
strict	Logical. Request strict validation when supported (OpenAI-compatible). Strict mode has formal requirements of its own: every object must set additionalProperties: false and list all its properties as required. LLMR fills those in automatically where your schema leaves them unspecified (your explicit settings are never overridden); pass strict = FALSE to send the schema verbatim.

Value

Modified llm_config.

Server-side enforcement by provider

OpenAI, Groq, Together, x.ai, OpenRouter, and Ollama accept a strict json_schema response format. DeepSeek, Alibaba (Qwen), Zhipu, Moonshot, and Xiaomi accept only JSON-object mode; for them the supplied schema drives local parsing and validation, so the prompt itself should describe the desired fields. Anthropic enforcement runs through a forced tool call; Gemini through responseJsonSchema.

Gemini

A supplied schema is sent as responseJsonSchema (standard JSON Schema, supported by Gemini 2.5+ models) together with the JSON mime type. For an older model that rejects it, set gemini_enable_response_schema = FALSE in the config to fall back to JSON-mime-type-only mode (the reply is still parsed and can be validated locally).

When to use tags instead

For tasks where strict JSON schema is unnecessary or unsupported, consider llm_mutate() with .tags or llm_mutate_tags() for soft structured output.

See Also

disable_structured_output(), llm_parse_structured(), llm_parse_structured_col(), llm_mutate_structured(
llm_mutate_tags()

`ensure_alternating_messages` *Make a message array provider-safe (alternating, user-leading)*

Description

Some providers (Anthropic, Gemini) reject a message array whose first non-system turn is not `user`, or that contains two consecutive turns of the same role. `ensure_alternating_messages()` repairs an assembled array so it satisfies both constraints: it merges all leading `system` messages into one system block, merges any run of adjacent same-role messages into one (joining their content), and guarantees the first non-system message has role `"user"`.

Usage

```
ensure_alternating_messages(messages)
```

Arguments

`messages`	A list of message objects, each `list(role=, content=)`. Any number of leading `system` messages is allowed; they are merged into one system block (most providers take system as a single top-level field).

Details

This is the normalizer that makes hand-built multi-turn arrays (for example from `transcript_as_messages()`) safe to send to any supported provider. It is a pure transformation: no network or file I/O.

Merging assumes scalar character content: if two adjacent same-role messages carry non-scalar content (for example multimodal/list content), they cannot be concatenated and the function errors rather than corrupt them. Supply an array that is already alternating in that case.

Value

A list of message objects that strictly alternates roles after the merged leading `system` block, beginning with `user`.

See Also

`transcript_as_messages()`, `call_llm()`

Examples

```
msgs <- list(
  list(role = "system", content = "be terse"),
  list(role = "user",   content = "Ana: hello"),
  list(role = "user",   content = "Ben: hi"),     # adjacent user -> merged
  list(role = "assistant", content = "my prior reply"),
  list(role = "user",   content = "your turn")
)
ensure_alternating_messages(msgs)
```

get_batched_embeddings *Generate Embeddings in Batches*

Description

A wrapper function that processes a list of texts in batches to generate embeddings, avoiding rate limits. This function calls call_llm_robust for each batch and stitches the results together and parses them (using parse_embeddings) to return a numeric matrix.

Usage

```
get_batched_embeddings(
  texts,
  embed_config,
  batch_size = 50,
  verbose = FALSE,
  tries = 5,
  wait_seconds = 2,
  backoff_factor = 3
)
```

Arguments

texts	Character vector of texts to embed. If named, the names will be used as row names in the output matrix.
embed_config	An llm_config object configured for embeddings.
batch_size	Integer. Number of texts to process in each batch. Default is 50. (Gemini's developer API embeds at most 100 texts per request; larger batches are split automatically.)
verbose	Logical. If TRUE, prints progress messages. Default is FALSE.
tries, wait_seconds, backoff_factor	Retry controls forwarded to call_llm_robust for each batch.

Value

A numeric matrix where each row is an embedding vector for the corresponding text. Columns are named v1, v2, ..., vK where K is the embedding dimension. If a transient failure exhausts its retries after another batch succeeds, or a successful response omits particular vectors, the affected rows are filled with NA. Authentication, invalid-request, and transport errors propagate, and the function errors if no batch yields an embedding. The matrix otherwise has the same number of rows as the input texts.

Batching, chunking, and row packing

"Batch" appears in three distinct senses in LLMR, and they are easy to keep apart once you note which path each belongs to. (1) The asynchronous provider Batch API (`llm_batch_submit()` and friends) defers a whole job for later delivery at a reduced price; it is the only deferred path here. (2) In the embedding path, `get_batched_embeddings()`'s `batch_size` sets how many texts go in one synchronous embedding request, bounded by the provider's per-call limit. (3) In the generative path, the `.rows_per_prompt` argument here packs several data rows into one generative prompt and parses them back into rows. Senses (2) and (3) are synchronous: `batch_size` and `.rows_per_prompt` control how work is grouped into requests, not the per-token rate, so the synchronous helpers do not themselves apply a provider discount. Pricing is a separate axis: several providers bill batched *embeddings* at a reduced rate through a dedicated async/batch tier (the embedding analogue of (1)), so embeddings are not categorically full-price; that discount is a property of the endpoint, not of `batch_size`. Row packing (3) can still lower total tokens by amortizing shared prompt overhead across rows, again without changing the rate.

See Also

`llm_config` to create the embedding configuration. `parse_embeddings` to convert the raw response to a numeric matrix. Here `batch_size` is embedding chunking, the synchronous sense; it is not the asynchronous provider Batch API (`llm_batch_submit`) nor the generative row packing of `llm_mutate` (`.rows_per_prompt`).

Examples

```
## Not run:
  # Basic usage
  texts <- c("Hello world", "How are you?", "Machine learning is great")
  names(texts) <- c("greeting", "question", "statement")

  # The key is read from the VOYAGE_API_KEY environment variable.
  embed_cfg <- llm_config(
    provider = "voyage",
    model = "voyage-3.5-lite",
    embedding = TRUE
  )

  embeddings <- get_batched_embeddings(
    texts = texts,
    embed_config = embed_cfg,
    batch_size = 2
  )

## End(Not run)
```

`llmr_response` *LLMR Response Object*

Description

A lightweight S3 container for **generative** model calls. It standardizes finish reasons and token usage across providers and keeps the raw response for advanced users.

Returns the standardized finish reason for an `llmr_response`.

Returns a list with token counts for an `llmr_response`.

Convenience check for truncation due to token limits.

Usage

```
finish_reason(x)

tokens(x)

is_truncated(x)

## S3 method for class 'llmr_response'
as.character(x, ...)

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

Arguments

`x`	An `llmr_response` object.
`...`	Ignored.

Details

Fields:

- `text`: character scalar. Assistant reply.
- `provider`: character. Provider id (e.g., `"openai"`, `"gemini"`).
- `model`: character. Model id as requested in the config.
- `model_version`: character. The model identifier the server reports having served (e.g., a dated snapshot). Useful for reproducibility records; `NA` when the provider does not echo it.
- `finish_reason`: one of `"stop"`, `"length"`, `"filter"`, `"tool"`, `"other"`.
- `usage`: list with integers `sent`, `rec`, `total`, `reasoning`, and `cached` (tokens read from the provider's prompt cache; `NA` when not reported).
- `thinking`: character. Reasoning text when the provider returns it separately (e.g., Anthropic thinking blocks, Gemini thought parts, DeepSeek `reasoning_content`); `NA` otherwise.
- `response_id`: provider's response identifier if present.
- `duration_s`: numeric seconds from request to parse.
- `raw`: parsed provider JSON (list).
- `raw_json`: raw JSON string.
- `messages`, `tool_history`, `tool_loop`: tool-loop provenance present on responses returned by `call_llm_tools()`; respectively, the full conversation, executed-call table, and aggregate loop usage.

Printing:

`print()` shows the text, then a compact status line with model, finish reason, token counts, and a terse hint if truncated or filtered.

Coercion:

`as.character()` extracts `text` so the object remains drop-in for code that expects a character return.

Value

A length-1 character vector or `NA_character_`.

A list `list(sent, rec, total, reasoning, cached)`. Missing values are `NA`. `cached` counts prompt tokens the provider read from its cache (cheaper than fresh input tokens); it is `NA` for providers that do not report cache usage.

`TRUE` if truncated, otherwise `FALSE`.

See also

`call_llm()`, `call_llm_robust()`, `llm_chat_session()`, `llm_config()`, `llm_mutate()`, `llm_fn()`

Examples

```
# Minimal fabricated example (no network):
r <- structure(
  list(
    text = "Hello!",
    provider = "openai",
    model = "demo",
    finish_reason = "stop",
    usage = list(sent = 12L, rec = 5L, total = 17L, reasoning = NA_integer_),
    response_id = "resp_123",
    duration_s = 0.012,
    raw = list(choices = list(list(message = list(content = "Hello!")))),
    raw_json = "{}"
  ),
  class = "llmr_response"
)
as.character(r)
finish_reason(r)
tokens(r)
print(r)
r <- structure(list(text="hi", model="demo", finish_reason="stop",
  usage=list(sent=1L, rec=1L, total=2L, reasoning=NA_integer_, cached=NA_integer_),
  duration_s=0.01), class="llmr_response")
finish_reason(r)
r <- structure(list(text="hi", model="demo", finish_reason="stop",
  usage=list(sent=1L, rec=1L, total=2L, reasoning=NA_integer_, cached=NA_integer_),
  duration_s=0.01), class="llmr_response")
tokens(r)
r <- structure(list(text="hi", model="demo", finish_reason="stop",
  usage=list(sent=1L, rec=1L, total=2L, reasoning=NA_integer_, cached=NA_integer_),
```

```
  duration_s=0.01), class="llmr_response")
is_truncated(r)
```

`llm_agreement` *Agreement across replicated LLM annotations*

Description

Computes per-row majority labels and overall reliability for replicate columns produced by `llm_replicate()` (or any set of columns holding repeated codings of the same units, including codings by different models or by humans). Reliability is reported as average pairwise percent agreement and Krippendorff's alpha for nominal data, the statistic reviewers most often ask for; alpha handles missing values (failed calls) gracefully.

Usage

```
llm_agreement(
  .data,
  cols = NULL,
  prefix = NULL,
  normalize = TRUE,
  metric = c("nominal", "ordinal", "interval"),
  levels = NULL
)
```

Arguments

`.data`	A data frame holding the replicate columns.
`cols`	Character vector naming the replicate columns. Alternatively supply `prefix`.
`prefix`	Base name: columns matching `<prefix>_1`, `<prefix>_2`, ... are used.
`normalize`	If `TRUE` (default), values are compared after trimming whitespace and lowercasing, so "Positive" and " positive" agree. Set to `FALSE` for exact string comparison.
`metric`	Difference function for Krippendorff's alpha: `"nominal"` (default, categories either match or do not), `"ordinal"` (ordered categories), or `"interval"` (numeric distance). For `"ordinal"` and `"interval"`, labels must parse as numbers, be ordered factors, or have an explicit order supplied through `levels`. Explicit categorical levels are treated as equally spaced for the interval metric. The default reproduces the previous nominal-only behavior exactly.
`levels`	Optional character vector giving the category order for `metric = "ordinal"` or `"interval"`. When omitted, a common ordered-factor level set is used if present; otherwise every observed label must parse as numeric.

Value

An object of class `llmr_agreement`: a list with

`by_row` a tibble with one row per unit: `majority` (modal label, `NA` on ties), `share` (modal share of non-missing replicates), `n_distinct`, `unanimous`, `tie`, `n_missing`.

`summary` a one-row tibble: `n_units`, `n_replicates`, `mean_pairwise_agreement`, `krippendorff_alpha`, `n_unanimous`, `n_ties`.

Printing shows the summary.

References

Krippendorff, K. (2019). Content Analysis: An Introduction to Its Methodology (4th ed.), chapter 12. The alpha implemented here is the nominal-data form with missing values allowed.

See Also

`llm_replicate()`

`llm_api_key_env` *Declare an API key sourced from an environment variable*

Description

Reference an API key by the name of the environment variable that holds it, so the secret never appears in your R code or saved objects. Store the key in your shell profile or in `~/.Renviron` (e.g. `OPENAI_API_KEY=sk-...`).

Usage

```
llm_api_key_env(var, required = TRUE, default = NULL)
```

Arguments

`var`	Name of the environment variable (e.g., "OPENAI_API_KEY"). A character vector is also accepted; the variables are tried in order and the first one that is set wins, which is convenient when a key may live under more than one name (e.g., `c("GROQ_API_KEY", "GROQ_KEY")`).
`required`	If TRUE, a missing variable raises an authentication error at call time. If FALSE, a missing variable resolves to an empty key, which is appropriate for providers that do not require authentication (e.g., a local Ollama server).
`default`	Optional default used if the environment variable is not set.

Details

Best practice is to not pass a key explicitly at all: `llm_config()` already looks up the standard variable for each provider (`<PROVIDER>_API_KEY`, then `<PROVIDER>_KEY`). Use `llm_api_key_env()` only when your variable has a non-standard name.

Value

A secret handle to pass as `api_key = llm_api_key_env("VARNAME")` in `llm_config()`.

See Also

`llm_config()`

Examples

```
cfg <- llm_config(
  "openai", "gpt-4o-mini",
  api_key = llm_api_key_env("MY_OPENAI_KEY")
)
```

---

llm_batch_cancel *Cancel a batch job*

Description

Cancel a batch job

Usage

```
llm_batch_cancel(job)
```

Arguments

`job`	An `llmr_batch_job` from `llm_batch_submit()`, or the path to one saved via `state_path`.

Value

The provider's response, invisibly.

---

llm_batch_fetch *Fetch the results of a batch job*

Description

Retrieves a finished job and returns one row per submitted request, in submission order, with the same diagnostic columns as `call_llm_par()` (response text, success, finish reason, token counts including cached tokens, response id, raw JSON). Rows whose requests failed carry the provider's error message. Parse structured replies afterwards with `llm_parse_structured_col()` or `llm_parse_tags_col()`, exactly as for live results.

Usage

```
llm_batch_fetch(job)
```

Arguments

job	An llmr_batch_job from llm_batch_submit(), or the path to one saved via state_path.

Value

A tibble with custom_id plus the diagnostic columns described above. If the job is not finished yet, an error is raised; check with llm_batch_status() first.

---

llm_batch_status *Check the status of a batch job*

Description

Check the status of a batch job

Usage

```
llm_batch_status(job)
```

Arguments

job	An llmr_batch_job from llm_batch_submit(), or the path to one saved via state_path.

Value

A one-row tibble: provider, batch_id, status (provider's wording; "completed"/"ended"/"done" mean ready), n_total, n_completed, n_failed (NA where a provider does not report counts).

See Also

llm_batch_submit(), llm_batch_fetch()

`llm_batch_submit` *Submit a batch job to a provider's batch API*

Description

Provider batch APIs run large jobs asynchronously at a reduced price (typically half) in exchange for delayed delivery (minutes up to 24 hours). `llm_batch_submit()` packages one request per element of `messages` and submits the job; `llm_batch_status()` polls it; `llm_batch_fetch()` retrieves results as a tidy tibble aligned with the inputs.

Usage

```
llm_batch_submit(config, messages, state_path = NULL)
```

Arguments

`config`	An llm_config for a generative model.
`messages`	An unnamed character vector (each element becomes one request's user message); a named character vector like `c(system = ..., user = ...)` (a single multi-role request); or a list with one element per request, where each element is any messages form accepted by `call_llm()`. Multimodal file parts are not supported in batch jobs.
`state_path`	Optional file path; when given, the job object is also saved there as RDS (and the path is remembered for convenience).

Details

Supported providers: `"openai"` and `"groq"` (Files + Batches protocol), `"anthropic"` (Message Batches), and `"gemini"` (`batchGenerateContent` with inline requests; developer API, not Vertex). All request-shaping features of `llm_config()` apply: sampling parameters, structured output, tools, and hooks shape each request exactly as a live `call_llm()` would.

A job whose config stores an environment-variable reference (the default) contains no secrets: it can be saved to disk, shared, and fetched later or from another machine with the same environment variables set. Pass `state_path` to save it automatically. When the config carries a literal key string instead, `state_path` is refused rather than writing the key to disk; the in-memory job still works for the current session.

Value

An `llmr_batch_job` object.

Batching, chunking, and row packing

"Batch" appears in three distinct senses in LLMR, and they are easy to keep apart once you note which path each belongs to. (1) The asynchronous provider Batch API (`llm_batch_submit()` and friends) defers a whole job for later delivery at a reduced price; it is the only deferred path here. (2) In the embedding path, `get_batched_embeddings()`'s `batch_size` sets how many texts go

in one synchronous embedding request, bounded by the provider's per-call limit. (3) In the generative path, the `.rows_per_prompt` argument here packs several data rows into one generative prompt and parses them back into rows. Senses (2) and (3) are synchronous: `batch_size` and `.rows_per_prompt` control how work is grouped into requests, not the per-token rate, so the synchronous helpers do not themselves apply a provider discount. Pricing is a separate axis: several providers bill batched *embeddings* at a reduced rate through a dedicated async/batch tier (the embedding analogue of (1)), so embeddings are not categorically full-price; that discount is a property of the endpoint, not of `batch_size`. Row packing (3) can still lower total tokens by amortizing shared prompt overhead across rows, again without changing the rate.

See Also

`llm_batch_status()`, `llm_batch_fetch()`, `llm_batch_cancel()`; and, for the two synchronous senses of "batch", `get_batched_embeddings()` (embedding chunking) and `llm_mutate()` (`.rows_per_prompt` row packing).

Examples

```
## Not run:
cfg <- llm_config("groq", "openai/gpt-oss-20b", temperature = 0)
job <- llm_batch_submit(cfg, c("2+2?", "Capital of Chile?"),
                        state_path = "my_batch.rds")
llm_batch_status(job)
# ... later, possibly in a new session:
res <- llm_batch_fetch("my_batch.rds")

## End(Not run)
```

`llm_chat_session` *Chat Session Object and Methods*

Description

Create and interact with a stateful chat session object that retains message history. This documentation page covers the constructor function `chat_session()` as well as all S3 methods for the `llm_chat_session` class.

Usage

```
chat_session(config, system = NULL, quiet = FALSE, ...)

## S3 method for class 'llm_chat_session'
as.data.frame(x, ...)

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

## S3 method for class 'llm_chat_session'
```

```
head(x, n = 6L, width = getOption("width") - 15, ...)

## S3 method for class 'llm_chat_session'
tail(x, n = 6L, width = getOption("width") - 15, ...)

## S3 method for class 'llm_chat_session'
print(x, width = getOption("width") - 15, ...)
```

Arguments

config	An llm_config **for a generative model** (embedding = FALSE).
system	Optional system prompt inserted once at the beginning.
quiet	Logical. If TRUE, $send() and friends do not print the reply; they still return it invisibly. Useful inside scripts and loops.
...	Default arguments forwarded to every call_llm_robust() call (e.g. verbose = TRUE).
x, object	An llm_chat_session object.
n	Number of turns to display.
width	Character width for truncating long messages.

Details

The chat_session object provides a simple way to hold a conversation with a generative model. It wraps call_llm_robust() to benefit from retry logic, caching, and error logging.

Value

For chat_session(), an object of class llm_chat_session. Other methods return what their titles state.

How it works

1. A private environment stores the running list of list(role, content) messages.
2. At each $send() the history is sent *in full* to the model.
3. Provider-agnostic token counts are extracted from the JSON response.

Public methods

$send(text, ..., role = "user") Append a message (default role "user"), query the model, print the assistant's reply (unless quiet = TRUE), and invisibly return it. text may also be a named character vector using the same multimodal shortcut as call_llm(), e.g. chat$send(c(user = "Describe this image.", file = "plot.png")).

$send_structured(text, schema, ..., role = "user", .validate_local = TRUE) Send a message with structured-output enabled using schema, append the assistant's reply, parse JSON (and optionally validate locally when .validate_local = TRUE), returning the parsed result invisibly.

`$send_tags(text, .tags, ..., role = "user")` Send a message with XML-like tag instructions injected, append the assistant's reply, parse the requested tags, and invisibly return the parsed list.

`$history()` Raw list of messages.

`$history_df()` Two-column data frame (`role`, `content`).

`$tokens_sent()`/`$tokens_received()` Running token totals.

`$reset()` Clear history (retains the optional system message).

See Also

`llm_config()`, `call_llm()`, `call_llm_robust()`, `llm_fn()`, `llm_mutate()`

Examples

```
if (interactive()) {
  cfg  <- llm_config("openai", "gpt-5-nano")
  chat <- chat_session(cfg, system = "Be concise.")
  chat$send("Who invented the moon?")
  chat$send("Explain why in one short sentence.")
  chat           # print() shows a summary and first 10 turns
  summary(chat)  # stats
  tail(chat, 2)
  as.data.frame(chat)
}
```

---

llm_config *Create an LLM configuration (provider-agnostic)*

Description

`llm_config()` builds a provider-agnostic configuration object that `call_llm()` (and friends) understand. You can pass provider-specific parameters via `...`; LLMR forwards them as-is, with a few safe conveniences.

Usage

```
llm_config(
  provider,
  model,
  api_key = NULL,
  troubleshooting = FALSE,
  base_url = NULL,
  embedding = NULL,
  no_change = FALSE,
  ...
)
```

Arguments

provider	Character scalar naming the backend. Known providers: "openai", "anthropic", "gemini", "groq", "together", "deepseek", "xai", "xiaomi", "alibaba" (Alibaba Cloud DashScope, OpenAI-compatible mode; serves the Qwen models), "zhipu", "moonshot", "openrouter" (an aggregator routing one OpenAI-compatible interface to many hosted models; model ids like "openai/gpt-4o-mini"), "voyage" (embeddings only), and "ollama" (local server, usually keyless). Other names are accepted and routed through the OpenAI-compatible path. When api_key is omitted, LLMR reads the key from the environment using a formulaic default: it tries <PROVIDER>_API_KEY and then <PROVIDER>_KEY, upper-cased (e.g. OPENAI_API_KEY, or ALIBABA_API_KEY/ALIBABA_KEY). The one exception is Gemini with vertex = TRUE, which reads VERTEX_ACCESS_TOKEN.
model	Character scalar. Model name understood by the chosen provider. (e.g., "gpt-4.1-nano", "gpt-5-nano", "gemini-2.5-flash-lite", "openai/gpt-oss-20b", etc.)
api_key	Provider API key. Preferred form is llm_api_key_env("VAR"), referencing an environment variable by name (see provider for the formulaic defaults). A bare environment-variable name or "env:VAR" string also works, as does a character vector of variable names tried in order. Supplying a literal key string is accepted but discouraged and triggers a warning. When omitted (or given as an empty string, which is what Sys.getenv() returns for an unset variable), the provider default is used. Printing a config never reveals the key.
troubleshooting	Logical. If TRUE, prints the messages and the config for debugging. The API key is masked in this output, not shown.
base_url	Optional character. Back-compat alias; if supplied it is stored as api_url in model_params and overrides the default endpoint.
embedding	NULL (default), TRUE, or FALSE. If TRUE, the call is routed to the provider's embeddings API; if FALSE, to the chat API. Voyage is always routed to embeddings because it is an embeddings-only provider. For other providers, NULL infers embeddings when model contains "embedding".
no_change	Logical. If TRUE, LLMR **never** auto-renames/adjusts provider parameters. If FALSE (default), well-known compatibility shims may apply (e.g., renaming OpenAI's max_tokens -> max_completion_tokens after a server hint; see call_llm() notes).
...	Additional model parameters. LLMR understands a small canonical set spelled the OpenAI way and translates it per provider, so you can keep one vocabulary across backends: • temperature, top_p, top_k, max_tokens, frequency_penalty, presence_penalty, repetition_penalty – sampling controls. Parameters a provider does not accept are dropped with a console note (e.g., repetition_penalty for Gemini); spellings a provider renames are renamed (e.g., max_tokens becomes maxOutputTokens for Gemini). • seed – request reproducible sampling where supported (OpenAI-compatible providers and Gemini; Anthropic has no seed). Determinism is not guaranteed; record model_version from the response for the full picture.

- logprobs, top_logprobs – token log-probabilities where supported (OpenAI-compatible chat APIs and Gemini). Retrieve them tidily with llm_logprobs().
- thinking_budget, include_thoughts – reasoning controls for Gemini and Anthropic. thinking_budget caps reasoning tokens (thinkingConfig.thinkingBudget on Gemini, thinking.budget_tokens on Anthropic, where it must be smaller than max_tokens). include_thoughts = TRUE asks Gemini to return its reasoning; Anthropic returns thinking blocks whenever thinking is on. Returned reasoning lands in the response's thinking field.
- timeout – total request timeout in seconds (default 600; also settable globally via options(llmr.timeout = ...)).
- cache – set cache = TRUE to mark the system prompt and tools as cacheable for Anthropic (prompt caching). OpenAI, Gemini, DeepSeek, and several compatible providers cache long prompt prefixes automatically; cached token counts are reported in tokens(x)$cached either way.
- Anything else (e.g., reasoning_effort, api_url, provider-specific flags) is forwarded verbatim on the OpenAI-compatible providers, so new provider features work without waiting for an LLMR release. Anthropic and Gemini have stricter request shapes: their builders send recognized fields only, and quietly note (once per session) anything they drop. The req_builder / request_modifier hooks remain the escape hatch for arbitrary fields on those providers.

Value

An object of class c("llm_config", provider). Fields: provider, model, api_key, troubleshooting, embedding, no_change, and model_params (a named list of extras). print() masks the API key.

Advanced hooks

Three optional functions in ... customize the HTTP exchange when a provider needs something unusual (a gateway header, an exotic body field, a nonstandard response envelope). All are applied on every request for every provider:

- request_modifier: function(body) -> body, edits the JSON body before serialization (OpenAI-compatible chat paths).
- req_builder: function(req) -> req, edits the httr2 request (headers, URL, auth) just before it is performed.
- response_modifier: function(content) -> content, edits the parsed JSON before LLMR interprets it.

Temperature range clamping

Anthropic temperatures must be in [0, 1]; others in [0, 2]. Out-of-range values are clamped with a warning. Reasoning or thinking-oriented models may reject custom temperature values; omit temperature unless the selected model accepts it.

Endpoint overrides

You can pass api_url (or base_url= alias) in ... to point to gateways or compatible proxies.

Vertex Gemini

Use `provider = "gemini", vertex = TRUE` for Gemini on Vertex AI. Supply `project` and optionally `location`; when `api_key` is omitted, LLMR looks for `VERTEX_ACCESS_TOKEN` and sends it as a Bearer token.

See Also

`call_llm`, `call_llm_robust`, `llm_chat_session`, `call_llm_par`, `get_batched_embeddings`

Examples

```
## Not run:
# Basic OpenAI config
cfg <- llm_config("openai", "gpt-4.1-nano",
temperature = 0.7, max_tokens = 300)

# Generative call returns an llmr_response object
r <- call_llm(cfg, "Say hello in Greek.")
print(r)
as.character(r)

# Embeddings (inferred from the model name)
e_cfg <- llm_config("gemini", "gemini-embedding-001")

# Force embeddings even if model name does not contain "embedding"
e_cfg2 <- llm_config("voyage", "voyage-3.5-lite", embedding = TRUE)

# Gemini through Vertex AI. VERTEX_ACCESS_TOKEN should contain a Bearer token.
v_cfg <- llm_config(
  "gemini", "gemini-2.5-flash-lite",
  vertex = TRUE,
  project = "my-gcp-project",
  location = "us-central1",
  api_key = "VERTEX_ACCESS_TOKEN"
)

## End(Not run)
```

`llm_failures` *List the rows of an LLM run that failed or were truncated*

Description

Returns one row per problem: a hard failure (`success` not `TRUE`) or a truncated / content-filtered completion (`finish_reason` `"length"` or `"filter"`), with the diagnostic detail needed to act. Works on both `call_llm_par()` and `llm_mutate()` results. For a `call_llm_par()` result (which still carries `config` and `messages`), pass the original frame to `llm_par_resume()` to re-run only these rows.

Usage

```
llm_failures(x, prefix = NULL, include = c("all", "failed", "truncated"))
```

Arguments

x	A data frame from call_llm_par() or llm_mutate().
prefix	For an llm_mutate() result, the output column name whose diagnostics to summarize (e.g. "answer"). Inferred automatically when a single diagnostic block is present; required when several are.
include	One of "failed" (hard failures only), "truncated" (length/filter only), or "all" (default; both).

Value

A tibble (zero rows if nothing matched) with: row (index into x), success, finish_reason, status_code, error_code, bad_param, error_message, response_id. Columns absent from x are filled with NA.

See Also

llm_par_resume() to re-run failed rows, llm_usage().

Examples

```
res <- tibble::tibble(
  success = c(TRUE, FALSE, TRUE),
  finish_reason = c("stop", "error:server", "length"),
  error_message = c(NA, "HTTP 503", NA)
)
llm_failures(res)
```

---

llm_fn *Apply an LLM prompt over vectors/data frames*

Description

Apply an LLM prompt over vectors/data frames

Usage

```
llm_fn(
  x,
  prompt,
  .config,
  .system_prompt = NULL,
  ...,
  .tags = NULL,
```

```
  .fields = NULL,
  .return = c("text", "columns", "object"),
  .na_action = c("send", "skip", "error"),
  .rows_per_prompt = 1L,
  .rowpack_payload = c("user", "system"),
 .rowpack_recovery = c("halve_recursive", "halve_once", "singletons", "retry_same",
    "none")
)
```

Arguments

x	A character vector *or* a data.frame/tibble.
prompt	A glue template string. With a data-frame you may reference columns (`{col}`); with a vector the placeholder is `{x}`.
.config	An llm_config object.
.system_prompt	Optional system message (character scalar).
...	Passed unchanged to `call_llm_broadcast()` (e.g. `tries`, `progress`, `verbose`).
.tags	Optional character vector of XML-like tag names to request and parse. When supplied, delegates to `llm_fn_tags()` for tag-based extraction.
.fields	Optional field selector for tag extraction (see `llm_fn_tags()`).
.return	One of `c("text","columns","object")`. `"columns"` returns a tibble of diagnostic columns; `"text"` returns a character vector; `"object"` returns a list of `llmr_response` (or `NA` on failure).
.na_action	What to do with elements whose template references an `NA` value. `"send"` (default) renders `NA` as an empty string and sends the prompt anyway; `"skip"` does not call the API for those elements and returns `NA` for them (`finish_reason = "skipped"` in column mode); `"error"` stops before any call is made. `"skip"`/`"error"` are not available together with `.tags`.
.rows_per_prompt	Integer scalar, or `Inf`. Number of input elements packed into a single generative request. The default, `1`, sends one request per element (the historical behavior). When greater than `1`, elements are grouped and transmitted in one call wrapped in numbered `<row_1>...</row_1>` tags (see *Row batching* below); `Inf` sends all elements in a single call. Ignored for embedding configurations, which use `get_batched_embeddings()` instead.
.rowpack_payload	One of `c("user","system")`. Channel to which the `<row_i>` data block is appended when batching. The imperative instruction is always placed in the system message; this argument controls only where the data goes. The default, `"user"`, keeps a static system prompt cacheable.
.rowpack_recovery	How to handle rows that a batched call leaves unresolved (dropped, malformed, or truncated). One of: `"halve_recursive"` (default) re-issue the unresolved rows at half the batch size, recursing down to single rows.

`"halve_once"` re-issue at half the batch size exactly once, then give up on any still-unresolved rows.

`"singletons"` re-issue each unresolved row on its own.

`"retry_same"` re-issue the failed batch once at the same size.

`"none"` do not recover; unresolved rows are returned as `NA`.

Recovery is bounded by an internal call budget so it always terminates.

Value

For generative mode:

- `.return = "text"`: character vector
- `.return = "columns"`: tibble with diagnostics
- `.return = "object"`: list of `llmr_response` (or `NA` on failure; unavailable when `.rows_per_prompt > 1`) For embedding mode, always a numeric matrix.

Batching, chunking, and row packing

"Batch" appears in three distinct senses in LLMR, and they are easy to keep apart once you note which path each belongs to. (1) The asynchronous provider Batch API (`llm_batch_submit()` and friends) defers a whole job for later delivery at a reduced price; it is the only deferred path here. (2) In the embedding path, `get_batched_embeddings()`'s `batch_size` sets how many texts go in one synchronous embedding request, bounded by the provider's per-call limit. (3) In the generative path, the `.rows_per_prompt` argument here packs several data rows into one generative prompt and parses them back into rows. Senses (2) and (3) are synchronous: `batch_size` and `.rows_per_prompt` control how work is grouped into requests, not the per-token rate, so the synchronous helpers do not themselves apply a provider discount. Pricing is a separate axis: several providers bill batched *embeddings* at a reduced rate through a dedicated async/batch tier (the embedding analogue of (1)), so embeddings are not categorically full-price; that discount is a property of the endpoint, not of `batch_size`. Row packing (3) can still lower total tokens by amortizing shared prompt overhead across rows, again without changing the rate.

Row packing

With `.rows_per_prompt > 1`, several input elements travel in one generative request: LLMR wraps each element's prompt in a numbered tag, `<row_1>...</row_1>`, `<row_2>...</row_2>`, and so on, appends that block to the message (see `.rowpack_payload`), and instructs the model to answer each item inside a matching numbered tag. The reply is split back into the original elements by those numbers. Batching trades a smaller number of (larger) requests for some dependence on the model following the protocol; it is most useful with capable models at `temperature = 0`, and it is a net loss when the model ignores the wrapping. Results are deterministic given the model's outputs: partitioning and parsing add no randomness. Rows the model drops, reorders, duplicates, or truncates are detected and re-issued according to `.rowpack_recovery`. Because a batch shares one underlying call, token counts are reported once per batch (on its first resolved row, `NA` elsewhere), as is the wall-clock duration, so that summing those columns is correct. When a batch reply is entirely unusable and its rows succeed only through recovery calls, the failed call's spend has no successful row to land on, so sums can slightly undercount in heavy-recovery runs.

See Also

llm_mutate(), llm_fn_structured(), llm_fn_tags(), setup_llm_parallel(), call_llm_broadcast(), get_batched_embeddings()

Examples

```
## Not run:
words <- c("excellent", "awful")
cfg <- llm_config("groq", "openai/gpt-oss-20b", temperature = 0)
llm_fn(words, "Classify '{x}' as Positive/Negative.", cfg, .return = "text")

df <- tibble::tibble(text = words, source = c("review", "review"))
llm_fn(df, "Classify '{text}' from {source}.", cfg, .return = "columns")

## End(Not run)
```

---

llm_fn_structured *Vectorized structured-output LLM*

Description

Schema-first variant of llm_fn(). It enables structured output on the config, calls the model via call_llm_broadcast(), parses JSON, and optionally validates.

Usage

```
llm_fn_structured(
  x,
  prompt,
  .config,
  .system_prompt = NULL,
  ...,
  .schema = NULL,
  .fields = NULL,
  .local_only = FALSE,
  .validate_local = TRUE,
  .rows_per_prompt = 1L,
  .rowpack_payload = c("user", "system"),
 .rowpack_recovery = c("halve_recursive", "halve_once", "singletons", "retry_same",
    "none")
)
```

Arguments

x	A character vector *or* a data.frame/tibble.
prompt	A glue template string. With a data-frame you may reference columns ({col}); with a vector the placeholder is {x}.

`.config` An llm_config object.

`.system_prompt` Optional system message (character scalar).

`...` Passed unchanged to `call_llm_broadcast()` (e.g. `tries`, `progress`, `verbose`).

`.schema` Optional JSON Schema list; if `NULL`, only JSON object is enforced.

`.fields` Optional fields to hoist from parsed JSON (supports nested paths).

`.local_only` If TRUE, do not send schema to the provider (parse/validate locally).

`.validate_local`
If TRUE and `.schema` provided, validate locally.

`.rows_per_prompt`
Integer scalar, or `Inf`. Number of input elements packed into a single generative request. The default, `1`, sends one request per element (the historical behavior). When greater than `1`, elements are grouped and transmitted in one call wrapped in numbered `<row_1>...</row_1>` tags (see *Row batching* below); `Inf` sends all elements in a single call. Ignored for embedding configurations, which use `get_batched_embeddings()` instead.

`.rowpack_payload`
One of `c("user","system")`. Channel to which the `<row_i>` data block is appended when batching. The imperative instruction is always placed in the system message; this argument controls only where the data goes. The default, `"user"`, keeps a static system prompt cacheable.

`.rowpack_recovery`
How to handle rows that a batched call leaves unresolved (dropped, malformed, or truncated). One of:

`"halve_recursive"` (default) re-issue the unresolved rows at half the batch size, recursing down to single rows.

`"halve_once"` re-issue at half the batch size exactly once, then give up on any still-unresolved rows.

`"singletons"` re-issue each unresolved row on its own.

`"retry_same"` re-issue the failed batch once at the same size.

`"none"` do not recover; unresolved rows are returned as `NA`.

Recovery is bounded by an internal call budget so it always terminates.

Value

A tibble: the `call_llm_broadcast()` diagnostics plus the parsed structured columns (`structured_ok`, `structured_data`, one column per hoisted field, and `structured_valid`/`structured_error` when `.schema` is validated locally).

See Also

`llm_fn()`, `llm_mutate_structured()`, `enable_structured_output()`, `llm_parse_structured_col()`

llm_fn_tags *Vectorized LLM with tag extraction*

Description

Tags-first variant of `llm_fn()`. Injects tag instructions, calls the model via `call_llm_broadcast()`, then parses XML-like tags from each response.

Usage

```
llm_fn_tags(
  x,
  prompt,
  .config,
  .system_prompt = NULL,
  ...,
  .tags,
  .fields = NULL,
  .return = c("columns", "text", "object"),
  .rows_per_prompt = 1L,
  .rowpack_payload = c("user", "system"),
 .rowpack_recovery = c("halve_recursive", "halve_once", "singletons", "retry_same",
    "none")
)
```

Arguments

`x`	A character vector *or* a data.frame/tibble.
`prompt`	A glue template string. With a data-frame you may reference columns (`{col}`); with a vector the placeholder is `{x}`.
`.config`	An llm_config object.
`.system_prompt`	Optional system message (character scalar).
`...`	Passed unchanged to `call_llm_broadcast()` (e.g. `tries`, `progress`, `verbose`).
`.tags`	Character vector of tag names to request and parse.
`.fields`	`NULL` to extract all tags, a character vector of tags, a named vector such as `c(person_age = "age")`, or `FALSE` to skip field extraction.
`.return`	One of `c("columns","text","object")`. `"columns"` returns a tibble with the parsed tag columns and diagnostics; `"text"` returns the raw response text. Unlike `llm_fn()`, `"object"` here returns the parsed tag data (a list, one element per row), not `llmr_response` objects; this form is also supported together with `.rows_per_prompt > 1`.
`.rows_per_prompt`	Integer scalar, or `Inf`. Number of input elements packed into a single generative request. The default, `1`, sends one request per element (the historical behavior). When greater than `1`, elements are grouped and transmitted in one call wrapped

in numbered <row_1>...</row_1> tags (see *Row batching* below); Inf sends all elements in a single call. Ignored for embedding configurations, which use get_batched_embeddings() instead.

.rowpack_payload

One of c("user","system"). Channel to which the <row_i> data block is appended when batching. The imperative instruction is always placed in the system message; this argument controls only where the data goes. The default, "user", keeps a static system prompt cacheable.

.rowpack_recovery

How to handle rows that a batched call leaves unresolved (dropped, malformed, or truncated). One of:

"halve_recursive" (default) re-issue the unresolved rows at half the batch size, recursing down to single rows.

"halve_once" re-issue at half the batch size exactly once, then give up on any still-unresolved rows.

"singletons" re-issue each unresolved row on its own.

"retry_same" re-issue the failed batch once at the same size.

"none" do not recover; unresolved rows are returned as NA.

Recovery is bounded by an internal call budget so it always terminates.

Value

Depends on .return: "columns" (default) a tibble with the parsed tag columns and diagnostics; "text" a character vector of the raw responses; "object" a list (one element per row) of parsed tag data.

See Also

llm_fn(), llm_mutate_tags(), llm_parse_tags_col(), call_llm_par_tags()

---

llm_hash *Content hash for research artifacts*

Description

One hash convention for the whole LLMR ecosystem: prompts, codebooks, coding protocols, archived requests. The object is reduced to a canonical form (list classes stripped, named lists sorted by name, functions replaced by their deparsed source), rendered as canonical JSON, and hashed with SHA-256 over the UTF-8 bytes of that string. Two consequences worth stating: equal content hashes equally regardless of construction order or list class, and the hash does not depend on R's serialization format, so a value recorded in a paper today is checkable on any machine later. Atomic vectors hash by their canonical JSON rendering, class included: a Date hashes as its date string, not as its unclassed integer.

Usage

```
llm_hash(x)
```

Arguments

x	An R object: list, character, config, codebook – anything whose canonical JSON form is well defined. Environments are not hashable.

Details

Downstream packages treat these hashes as identifiers of record (`LLMRcontent::protocol_lock()`, `LLMRcontent::archive_build()`); the convention is versioned by this function's documentation – any future change would be a new function, not a silent edit.

Value

A 64-character lowercase SHA-256 hex string.

Examples

```
llm_hash(list(model = "gpt-oss-20b", temperature = 0))
# construction order does not matter:
identical(llm_hash(list(a = 1, b = 2)), llm_hash(list(b = 2, a = 1)))
# any content change does:
identical(llm_hash(list(a = 1)), llm_hash(list(a = 2)))
```

---

llm_judge *LLM-as-a-Judge Evaluation*

Description

Evaluates outputs in a target column against a custom prompt using `llm_mutate_tags()` for clean tag-based extraction. The target column value is available in the prompt template as `{.target}`.

Usage

```
llm_judge(
  .data,
  .target,
  .config,
  prompt,
  .tags = c("reasoning", "score"),
  .output = "judge_res",
  ...
)
```

Arguments

.data	Data frame of experiment results.
.target	Bare column name containing the output to evaluate.
.config	The judge llm_config.
prompt	Evaluation prompt template. Use {.target} to reference the target column value (other data columns are also available).
.tags	Tags to extract from the judge response. Defaults to c("reasoning", "score").
.output	Name of the column that receives the judge's raw response. Default "judge_res".
...	Passed to llm_mutate_tags().

Value

.data with judge output columns appended.

See Also

llm_mutate_tags(), llm_parse_tags()

Examples

```
## Not run:
results |>
  llm_judge(
    .target = response_text,
    .config = judge_cfg,
    prompt = "Rate this answer on a 1-5 scale:\n{.target}",
    .tags = c("reasoning", "score")
  )

## End(Not run)
```

---

llm_logprobs *Extract token log-probabilities from a response*

Description

Token-level log-probabilities turn a classification into a measurement: the probability the model assigned to its own answer is a confidence score you can calibrate, threshold, or carry into downstream models as a soft label. Request them at config time (llm_config(..., logprobs = TRUE, top_logprobs = 5)); this helper then returns them tidily.

Usage

```
llm_logprobs(x)
```

Arguments

x	An llmr_response object (from call_llm() and friends), or a result frame from call_llm_par() (whose response list-column holds the response objects).

Value

For a single response: a tibble with one row per generated token: token (character), logprob (double), and top_logprobs (a list-column of data frames with the k most likely alternatives at that position, when requested). Returns a zero-row tibble when the response carries no logprobs. For a result frame: a list of such tibbles, one per row.

See Also

llm_config(), tokens()

Examples

```
## Not run:
# Provider support varies; deepseek-chat and OpenAI expose logprobs,
# Anthropic does not, and several hosts reject the flag model by model.
cfg <- llm_config("deepseek", "deepseek-chat",
                  logprobs = TRUE, top_logprobs = 3, temperature = 0)
r <- call_llm(cfg, "Answer with one word: is water wet?")
llm_logprobs(r)

# Confidence of the first answer token:
exp(llm_logprobs(r)$logprob[1])

## End(Not run)
```

---

llm_log_enable *Record every LLM call in a local audit log*

Description

llm_log_enable() turns on a session-wide audit log: each API call made through LLMR (including those issued by llm_fn(), llm_mutate(), call_llm_par(), and chat_session()) appends one JSON object to path. llm_log_disable() turns logging off. llm_log_status() reports the current destination, if any.

Usage

```
llm_log_enable(path = "llmr_log.jsonl", include_messages = TRUE)

llm_log_disable()

llm_log_status()

llm_log_active()
```

Arguments

path	File path for the log. It is opened for append when logging is enabled, creating the file if needed; an unusable destination errors before logging is activated.
include_messages	Logical. If TRUE (default), the request body and the reply text are stored. If FALSE, only metadata is stored.

Details

Methodological guidance for LLM-assisted research asks authors to retain, for every call: the model and provider, the full prompt, the inference settings, the output, and identifiers that allow an exact lookup later. The audit log records precisely that:

- ts: ISO-8601 timestamp with timezone.
- schema_version: the version of this record layout (currently "1.0"), so downstream tools that parse the log can rely on a stable contract.
- provider, model: as configured; model_version: the identifier the server reports having served (when echoed), which catches silent model updates.
- request: the JSON body sent to the provider, including all sampling parameters and the rendered messages. Inline file data (base64) is replaced by a short placeholder so logs stay small.
- text, finish_reason, usage: the reply, why it stopped, and token counts (including cached tokens when reported).
- response_id, status, duration_s: provider request id, HTTP status, and wall-clock seconds.
- Failed calls are logged too (kind = "error"), with the provider's error message.

Records are appended line by line. Under multisession parallel execution, workers write per-process shard files, which LLMR folds back into the main log after the parallel call completes. The log contains your prompts and the model's replies in clear text. It never contains API keys.

Set include_messages = FALSE to omit request bodies and reply text (keeping only metadata, parameters, usage, and identifiers), e.g. when prompts contain confidential data.

Value

llm_log_enable() and llm_log_disable() return the previous log path invisibly. llm_log_status() returns the active path or NULL, invisibly, after printing a one-line status.

llm_log_active() returns a list with active (logical), path (the destination or NULL), and include_messages (logical), for code that needs to read the logging state without printing.

See Also

llm_usage() for token summaries, report() for a draft methods paragraph.

Examples

```
## Not run:
llm_log_enable("annotation_run.jsonl")
cfg <- llm_config("groq", "openai/gpt-oss-20b")
call_llm(cfg, "One word: capital of France?")
llm_log_disable()

# Read the log back as a data frame
log_df <- jsonlite::stream_in(file("annotation_run.jsonl"), verbose = FALSE)

## End(Not run)
```

---

llm_log_read *Read an LLMR audit log into records and a manifest*

Description

Parses a JSONL audit log written by `llm_log_enable()` into its records and a per-record manifest tibble. Each manifest row carries the call's metadata, a `record_hash` over the verbatim log line (tamper evidence), and, where the request body was logged, a `request_hash` identifying the call. The `request_hash` is computed through `llm_request_hash()` over the call's canonical turns and generation parameters, so a call read from a log and the same call described by a config hash identically.

Usage

```
llm_log_read(log)
```

Arguments

`log`	Path to a JSONL audit log.

Value

A list with `records` (a list of `list(raw=, rec=)`, one per line) and `manifest` (a tibble with `idx`, `ts`, `kind`, `provider`, `model`, `model_version`, `status`, `schema_version`, `has_payload`, `request_hash`, `record_hash`).

See Also

`llm_request_hash()`, `llm_log_enable()`

Examples

```
# In practice the log comes from llm_log_enable("study.jsonl"); here one
# record written by hand.
log <- tempfile(fileext = ".jsonl")
writeLines(paste0(
  '{"ts":"2026-06-01T10:00:01+0000","schema_version":"1.0","kind":"call",',
  '"provider":"openai","model":"gpt-4o-mini","status":200,',
  '"request":{"messages":[{"role":"user","content":"Capital of France?"}],',
  '"temperature":0},"usage":{"sent":5,"rec":1},',
  '"response_id":"r-1","text":"Paris"}'), log)
read <- llm_log_read(log)
read$manifest
```

`llm_mutate` *Mutate a data frame with LLM output*

Description

Adds one or more columns to `.data` that are produced by a Large-Language-Model.

Usage

```
llm_mutate(
  .data,
  output,
  prompt = NULL,
  .messages = NULL,
  .config,
  .system_prompt = NULL,
  .before = NULL,
  .after = NULL,
  .return = c("columns", "text", "object"),
  .na_action = c("send", "skip", "error"),
  .structured = FALSE,
  .schema = NULL,
  .fields = NULL,
  .tags = NULL,
  .rows_per_prompt = 1L,
  .rowpack_payload = c("user", "system"),
  .rowpack_recovery = c("halve_recursive", "halve_once", "singletons", "retry_same",
    "none"),
  ...
)
```

Arguments

`.data`	A data.frame / tibble.

`output` Unquoted name that becomes **the new column** (generative) *or* **the prefix** for embedding columns. In shorthand form, omit this argument and pass `newcol = "<glue prompt>"` or `newcol = c(system = "...", user = "...")` through `...`.

`prompt` Optional glue template string for a single user turn; reference any columns in `.data` (e.g. `"{id}. {question}\nContext: {context}"`). Ignored if `.messages` is supplied.

`.messages` Optional **named** character vector of glue templates to build a multi-turn message, using roles in `c("system","user","assistant","file")`. Values are glue templates evaluated per-row; all can reference multiple columns. For multimodal, use role `"file"` with a column containing a path template.

`.config` An llm_config object (generative or embedding).

`.system_prompt` Optional system message sent with every request when `.messages` does not include a `system` entry.

`.before, .after` Standard dplyr::relocate helpers controlling where the generated column(s) are placed.

`.return` One of `c("columns","text","object")`. For generative mode, controls how results are added. `"columns"` (default) adds text plus diagnostic columns; `"text"` adds a single text column; `"object"` adds a list-column of `llmr_response` objects named `<output>_obj`. Ignored (with a warning) when `.structured = TRUE` or `.tags` is supplied, which always return parsed columns.

`.na_action` What to do with rows whose template references an `NA` value. `"send"` (default) renders `NA` as an empty string and sends the prompt anyway; `"skip"` does not call the API for those rows (the output column is `NA` and `finish_reason` is `"skipped"`); `"error"` stops before any call is made. With `.structured` or `.tags`, only `"send"` and `"error"` are available.

`.structured` Logical. If `TRUE`, enables structured JSON output with automatic parsing. When enabled, this is equivalent to calling `llm_mutate_structured()`. Default is `FALSE`.

`.schema` Optional JSON Schema (R list). When `.structured = TRUE`, this schema is sent to the provider for validation and used for local parsing. When `NULL`, only JSON mode is enabled (no strict schema validation).

`.fields` Optional character vector of fields to extract from parsed JSON or tag output. In JSON mode, supports nested paths (e.g., `"user.name"` or `"/data/items/0"`). When `NULL` and `.schema` is provided, auto-extracts all top-level schema properties. In tag mode, `NULL` extracts all `.tags`. Set to `FALSE` to skip field extraction entirely.

`.tags` Optional character vector of XML-like tag names to request and parse, such as `c("age", "job")`. When supplied, `llm_mutate()` delegates to `llm_mutate_tags()` and adds `tags_ok`, `tags_data`, and one column per tag unless `.fields = FALSE`.

`.rows_per_prompt` Integer scalar, or `Inf`. Number of rows packed into a single generative request. The default, `1`, sends one request per row (the historical behavior). When greater than `1`, rows are grouped and sent in one call wrapped in numbered `<row_1>...</row_1>` tags (see *Row batching* below); `Inf` sends all rows at once. Works in generative, tag, and structured modes; not applicable to embedding configurations.

.rowpack_payload
: One of c("user","system"). Channel to which the <row_i> data block is appended when batching. The default "user" keeps a static system prompt cacheable; the imperative instruction is always placed in the system message.

.rowpack_recovery
: How to handle rows a batched call leaves unresolved. One of "halve_recursive" (default), "halve_once", "singletons", "retry_same", or "none"; see llm_fn() for the precise meaning of each.

...
: Passed to the underlying calls: call_llm_broadcast() in generative mode, get_batched_embeddings() in embedding mode.

Details

- **Multi-column injection:** templating is NA-safe (NA -> empty string).
- **Multi-turn templating:** supply .messages = c(system=..., user=..., file=...). Duplicate role names are allowed (e.g., two user turns).
- **Generative mode:** one request per row via call_llm_broadcast().
- **Parallelism:** calls call_llm_broadcast(), which uses call_llm_robust() under the hood. If no future plan is active, workers are auto-configured; call setup_llm_parallel() to set worker count explicitly.
- **Embedding mode:** the per-row text is embedded via get_batched_embeddings(). Result expands to numeric columns named paste0(<output>, 1:N). If all rows fail to embed, a single <output>1 column of NA is returned.
- Diagnostic columns use suffixes: _finish, _sent, _rec, _tot, _reason, _ok, _err, _id, _status, _ecode, _param, _t.
- **Row packing:** with .rows_per_prompt > 1, three further columns are added (_rowpack, _rpn, _rpi: the row-pack identifier, the size of the resolving call, and the within-call position). They appear only when packing actually groups rows, so the default schema is unchanged at .rows_per_prompt = 1.

Value

.data with the new column(s) appended.

Row batching

With .rows_per_prompt > 1, several rows travel in one generative request. LLMR wraps each row's prompt in a numbered tag, <row_1>...</row_1>, <row_2>...</row_2>, and so on, appends that block to the message (see .rowpack_payload), and instructs the model to answer each item inside a matching numbered tag; the reply is split back into rows by those numbers. This also composes with .tags (each <row_i> then wraps the requested field tags) and with .structured = TRUE (rows are returned as one JSON object {"results":[{"row":i, ...}]}, de-multiplexed by the integer row field; a one-time warning notes that this relies on the model honoring the protocol and that strict provider-side schema validation is replaced by local parsing). Batching is most useful with capable models at temperature = 0 and is a net loss when the model ignores the wrapping. Dropped, reordered, duplicated, or truncated rows are detected and re-issued per .rowpack_recovery; token counts are reported once per batch so that summing token columns stays correct.

Shorthand

You can supply the output column and prompt in one argument:

```
df |> llm_mutate(answer = "{question} (hint: {hint})", .config = cfg)
df |> llm_mutate(answer = c(system = "One word.", user = "{question}"), .config = cfg)
df |> llm_mutate(country = "Where is {city}? Answer with only the country.", .config = cfg)
```

This is equivalent to:

```
df |> llm_mutate(answer, prompt = "{question} (hint: {hint})", .config = cfg)
df |> llm_mutate(answer, .messages = c(system = "One word.", user = "{question}"), .config = cfg)
```

Structured modes

- `.structured = TRUE` delegates to `llm_mutate_structured()` for JSON.
- `.tags` delegates to `llm_mutate_tags()` for XML-like tags. If both are supplied, `.structured` takes precedence.

See Also

`llm_fn()`, `llm_mutate_structured()`, `llm_mutate_tags()`, `llm_parse_structured_col()`, `llm_parse_tags_col()`, `call_llm_broadcast()`, `setup_llm_parallel()`

Examples

```
## Not run:
library(dplyr)

df <- tibble::tibble(
  id       = 1:2,
  question = c("Capital of France?", "Author of 1984?"),
  hint     = c("European city", "English novelist")
)

cfg <- llm_config("groq", "openai/gpt-oss-20b",
                  temperature = 0)

# Generative: single-turn with multi-column injection
df |>
  llm_mutate(
    answer,
    prompt = "{question} (hint: {hint})",
    .config = cfg,
    .system_prompt = "Respond in one word."
  )

# Generative: multi-turn via .messages (system + user)
df |>
  llm_mutate(
    advice,
```

```
    .messages = c(
      system = "You are a helpful zoologist. Keep answers short.",
      user   = "What is a key fact about this? {question} (hint: {hint})"
    ),
    .config = cfg
  )

# Multimodal: include an image path with role 'file'
pics <- tibble::tibble(
  img    = c("path/to/cat.png", "path/to/dog.jpg"),
  prompt = c("Describe the image.", "Describe the image.")
)
pics |>
  llm_mutate(
    vision_desc,
    .messages = c(user = "{prompt}", file = "{img}"),
    .config = llm_config("openai","gpt-4.1-mini")
  )

# Embeddings: output name becomes the prefix of embedding columns
emb_cfg <- llm_config("voyage", "voyage-3.5-lite",
                      embedding = TRUE)
df |>
  llm_mutate(
    vec,
    prompt  = "{question}",
    .config = emb_cfg,
    .after  = id
  )

# Structured output: using .structured = TRUE (equivalent to llm_mutate_structured)
schema <- list(
  type = "object",
  properties = list(
    answer = list(type = "string"),
    confidence = list(type = "number")
  ),
  required = list("answer", "confidence")
)

df |>
  llm_mutate(
    result,
    prompt = "{question}",
    .config = cfg,
    .structured = TRUE,
    .schema = schema
  )

# Structured with shorthand
df |>
  llm_mutate(
    result = "{question}",
```

```
    .config = cfg,
    .structured = TRUE,
    .schema = schema
  )

# Soft structured output with XML-like tags
df |>
  llm_mutate(
    result = "Extract the person's age and job from: {question}",
    .config = cfg,
    .tags = c("age", "job")
  )

cities <- tibble::tibble(city = c("Cairo", "Lima"))
cities |>
  llm_mutate(
    geo = "Where is {city}? Give country and continent in their own tags.",
    .config = cfg,
    .system_prompt = paste(
      "Use XML tags for different parts of the answer, but do not nest tags.",
      "Return <country>...</country> and <continent>...</continent>."
    ),
    .tags = c("country", "continent")
  )

## End(Not run)
```

`llm_mutate_structured` *Data-frame mutate with structured output*

Description

Drop-in schema-first variant of `llm_mutate()`. Produces parsed columns.

Usage

```
llm_mutate_structured(
  .data,
  output,
  prompt = NULL,
  .messages = NULL,
  .config,
  .system_prompt = NULL,
  .before = NULL,
  .after = NULL,
  .schema = NULL,
  .fields = NULL,
  .validate_local = TRUE,
  .rows_per_prompt = 1L,
```

```
  .rowpack_payload = c("user", "system"),
  .rowpack_recovery = c("halve_recursive", "halve_once", "singletons", "retry_same",
    "none"),
  ...
)
```

Arguments

`.data`	A data.frame / tibble.
`output`	Unquoted name that becomes **the new column** (generative) *or* **the prefix** for embedding columns. In shorthand form, omit this argument and pass `newcol = "<glue prompt>"` or `newcol = c(system = "...", user = "...")` through `...`.
`prompt`	Optional glue template string for a single user turn; reference any columns in `.data` (e.g. `"{id}. {question}\nContext: {context}"`). Ignored if `.messages` is supplied.
`.messages`	Optional **named** character vector of glue templates to build a multi-turn message, using roles in `c("system","user","assistant","file")`. Values are glue templates evaluated per-row; all can reference multiple columns. For multimodal, use role `"file"` with a column containing a path template.
`.config`	An llm_config object (generative or embedding).
`.system_prompt`	Optional system message sent with every request when `.messages` does not include a `system` entry.
`.before, .after`	Standard dplyr::relocate helpers controlling where the generated column(s) are placed.
`.schema`	Optional JSON Schema (R list). When provided, this schema is sent to the provider for strict validation and used for local parsing. When `NULL`, only JSON mode is enabled (no strict schema validation). The schema should follow JSON Schema specification (e.g., with `type`, `properties`, `required`).
`.fields`	Optional character vector of fields to extract from parsed JSON. Supports: • Character vector: `c("name", "score")` - extract these fields • Named vector: `c(person_name = "name", rating = "score")` - extract and rename • Nested paths: `c("user.name", "/data/items/0")` - dot notation or JSON Pointer • `NULL` (default): auto-extracts all top-level properties from `.schema` • `FALSE`: skip field extraction (keep only `structured_data` list-column)
`.validate_local`	If TRUE (default) and `.schema` is provided, each parsed object is validated locally against the schema (requires the `jsonvalidate` package), adding `structured_valid` and `structured_error` columns, exactly as `llm_fn_structured()` does.
`.rows_per_prompt`	Integer scalar, or `Inf`. Number of rows packed into a single generative request. The default, `1`, sends one request per row (the historical behavior). When greater than `1`, rows are grouped and sent in one call wrapped in numbered `<row_1>...</row_1>` tags (see *Row batching* below); `Inf` sends all rows at

once. Works in generative, tag, and structured modes; not applicable to embedding configurations.

`.rowpack_payload`
: One of `c("user","system")`. Channel to which the `<row_i>` data block is appended when batching. The default `"user"` keeps a static system prompt cacheable; the imperative instruction is always placed in the system message.

`.rowpack_recovery`
: How to handle rows a batched call leaves unresolved. One of `"halve_recursive"` (default), `"halve_once"`, `"singletons"`, `"retry_same"`, or `"none"`; see `llm_fn()` for the precise meaning of each.

`...`
: Passed to the underlying calls: `call_llm_broadcast()` in generative mode, `get_batched_embeddings()` in embedding mode.

Value

`.data` with the output column, the diagnostic columns, and the parsed structured columns (`structured_ok`, `structured_data`, one column per hoisted field, and `structured_valid`/`structured_error` when `.schema` is validated locally).

Shorthand syntax

Like `llm_mutate()`, this function supports shorthand syntax:

```
df |> llm_mutate_structured(result = "{text}", .config = cfg, .schema = schema)
df |> llm_mutate_structured(result = c(system = "Be brief.", user = "{text}"), .config = cfg, .schema = s
```

See Also

`llm_mutate()`, `llm_fn_structured()`, `enable_structured_output()`, `llm_parse_structured_col()`, `llm_mutate_tags()`

---

llm_mutate_tags *Data-frame mutate with XML-like tag output*

Description

Soft structured variant of `llm_mutate()`. It asks the model to return simple XML-like tags, then parses those tags into columns.

Usage

```
llm_mutate_tags(
  .data,
  output,
  prompt = NULL,
  .messages = NULL,
  .config,
```

```
  .system_prompt = NULL,
  .before = NULL,
  .after = NULL,
  .tags,
  .fields = NULL,
  .rows_per_prompt = 1L,
  .rowpack_payload = c("user", "system"),
 .rowpack_recovery = c("halve_recursive", "halve_once", "singletons", "retry_same",
    "none"),
  ...
)
```

Arguments

`.data`	A data.frame / tibble.
`output`	Unquoted name that becomes **the new column** (generative) *or* **the prefix** for embedding columns. In shorthand form, omit this argument and pass `newcol = "<glue prompt>"` or `newcol = c(system = "...", user = "...")` through `....`
`prompt`	Optional glue template string for a single user turn; reference any columns in `.data` (e.g. `"{id}. {question}\nContext: {context}"`). Ignored if `.messages` is supplied.
`.messages`	Optional **named** character vector of glue templates to build a multi-turn message, using roles in `c("system","user","assistant","file")`. Values are glue templates evaluated per-row; all can reference multiple columns. For multimodal, use role `"file"` with a column containing a path template.
`.config`	An llm_config object (generative or embedding).
`.system_prompt`	Optional system message sent with every request when `.messages` does not include a `system` entry.
`.before, .after`	Standard dplyr::relocate helpers controlling where the generated column(s) are placed.
`.tags`	Character vector of tag names to request and parse.
`.fields`	`NULL` to extract all tags, a character vector of tags, a named vector such as `c(person_age = "age")`, or `FALSE` to keep only `tags_data`.
`.rows_per_prompt`	Integer scalar, or `Inf`. Number of rows packed into a single generative request. The default, `1`, sends one request per row (the historical behavior). When greater than `1`, rows are grouped and sent in one call wrapped in numbered `<row_1>...</row_1>` tags (see *Row batching* below); `Inf` sends all rows at once. Works in generative, tag, and structured modes; not applicable to embedding configurations.
`.rowpack_payload`	One of `c("user","system")`. Channel to which the `<row_i>` data block is appended when batching. The default `"user"` keeps a static system prompt cacheable; the imperative instruction is always placed in the system message.

`.rowpack_recovery`
: How to handle rows a batched call leaves unresolved. One of `"halve_recursive"` (default), `"halve_once"`, `"singletons"`, `"retry_same"`, or `"none"`; see `llm_fn()` for the precise meaning of each.

`...`
: Passed to the underlying calls: `call_llm_broadcast()` in generative mode, `get_batched_embeddings()` in embedding mode.

Details

Returns the mutated data frame plus:

`tags_ok` `TRUE` when all requested tags were found.

`tags_data` A list-column of parsed tag lists.

tag columns One column per requested tag or field. Scalar columns are coerced to numeric or logical when all non-missing values allow it.

Value

`.data` with the output column, the diagnostic columns, `tags_ok`, `tags_data`, and one column per requested tag or field.

Shorthand syntax

```
df |> llm_mutate_tags(result = "{text}", .tags = c("age", "job"), .config = cfg)
```

See Also

`llm_mutate()`, `llm_parse_tags()`, `llm_parse_tags_col()`, `llm_mutate_structured()`, `llm_parse_structured_col(`

Examples

```
## Not run:
df <- tibble::tibble(city = c("Cairo", "Lima"))
cfg <- llm_config("groq", "openai/gpt-oss-20b", temperature = 0)

df |>
  llm_mutate_tags(
    geo = "Where is {city}? Give country and continent in their own tags.",
    .config = cfg,
    .system_prompt = paste(
      "Use XML tags for different parts of the answer, but do not nest tags.",
      "Return <country>...</country> and <continent>...</continent>."
    ),
    .tags = c("country", "continent")
  )

## End(Not run)
```

`llm_parse_structured` *Parse structured output emitted by an LLM*

Description

Robustly parses an LLM's structured output (JSON). Works on character scalars or an llmr_response. Strips code fences first, then tries strict parsing, then attempts to extract the largest balanced {...} or [...].

Usage

```
llm_parse_structured(x, strict_only = FALSE, simplify = FALSE)
```

Arguments

`x`	Character or llmr_response.
`strict_only`	If TRUE, do not attempt recovery via substring extraction.
`simplify`	Logical passed to jsonlite::fromJSON (`simplifyVector = FALSE` when FALSE).

Details

The return contract is list-or-NULL; scalar-only JSON is treated as failure.

Numerics are coerced to double for stability.

Value

A parsed R object (list), or NULL on failure.

See Also

`llm_parse_structured_col()`, `llm_fn_structured()`, `llm_mutate_structured()`, `llm_parse_tags()`

Examples

```
llm_parse_structured('{"score": 5, "label": "good"}')
```

`llm_parse_structured_col` *Parse structured fields from a column into typed vectors*

Description

Extracts fields from a column containing structured JSON (string or list) and appends them as new columns. Adds `structured_ok` (logical) and `structured_data` (list).

Usage

```
llm_parse_structured_col(
  .data,
  fields,
  structured_col = "response_text",
  prefix = "",
  allow_list = TRUE
)
```

Arguments

`.data`	data.frame/tibble
`fields`	Character vector of fields or named vector (dest_name = path).
`structured_col`	Column name to parse from. Default "response_text".
`prefix`	Optional prefix for new columns.
`allow_list`	Logical. If TRUE (default), non-scalar values (arrays/objects) are hoisted as list-columns instead of being dropped. If FALSE, only scalar fields are hoisted and non-scalars become NA.

Details

- Supports nested-path extraction via dot/bracket paths (e.g., `a.b[0].c`) or JSON Pointer (`/a/b/0/c`).
- When `allow_list = TRUE`, non-scalar values become list-columns; otherwise they yield `NA` and only scalars are hoisted.

Value

`.data` with diagnostics and one new column per requested field.

See Also

`llm_parse_structured()`, `llm_mutate_structured()`, `llm_parse_tags_col()`

Examples

```
df <- data.frame(response_text = '{"score": 5, "label": "good"}')
llm_parse_structured_col(df, fields = c("score", "label"))
```

llm_parse_tags *Parse XML-like tags emitted by an LLM*

Description

Extracts simple XML-like tags from a character scalar or llmr_response, such as <age>21</age> and <job>student</job>. This is intended for soft structured output, not full XML validation.

Usage

```
llm_parse_tags(x, tags)
```

Arguments

x	Character scalar or llmr_response.
tags	Character vector of tag names to extract.

Value

A named list of extracted tag values, or NULL when no requested tag is found.

See Also

llm_parse_tags_col(), llm_mutate_tags()

Examples

```
llm_parse_tags("<age>21</age><job>student</job>", tags = c("age", "job"))
```

llm_parse_tags_col *Parse XML-like tag fields from a column*

Description

Appends tags_ok, tags_data, and one column per requested tag or field.

Usage

```
llm_parse_tags_col(
  .data,
  tags,
  tags_col = "response_text",
  fields = NULL,
  prefix = ""
)
```

Arguments

.data	data.frame/tibble.
tags	Character vector of tag names to parse.
tags_col	Column name to parse from. Default "response_text".
fields	NULL to extract all tags, a character vector of tags, a named vector such as c(person_age = "age"), or FALSE to skip field extraction.
prefix	Optional prefix for extracted columns.

Value

.data with tag diagnostics and extracted columns.

See Also

llm_parse_tags(), llm_mutate_tags(), llm_parse_structured_col()

Examples

```
df <- data.frame(response_text = "<age>21</age><job>student</job>")
llm_parse_tags_col(df, tags = c("age", "job"))
llm_parse_tags_col(df, tags = c("age", "job"), fields = c(person_age = "age"))
```

---

llm_par_resume *Resume failed parallel LLM calls*

Description

Finds rows where success is FALSE or NA in the output of call_llm_par(), re-runs them, and patches the results back into the original data frame.

Usage

```
llm_par_resume(results, tries = 3, ...)
```

Arguments

results	Output from call_llm_par() (must contain config, messages, and success columns).
tries	Number of retries per call. Default 3.
...	Passed to call_llm_par().

Value

The patched data frame with re-run results filled in.

See Also

call_llm_par()

Examples

```
## Not run:
results <- call_llm_par(experiments)
results <- llm_par_resume(results, tries = 3)

## End(Not run)
```

`llm_persona_demographic_fields` *Which columns of a persona frame are demographics*

Description

Returns the demographic column names. A persona frame should mark these in a `demographic_fields` attribute; when it does not, a small set of common demographic names found in the frame is used as a fallback.

Usage

```
llm_persona_demographic_fields(x)
```

Arguments

`x`	A persona data frame.

Value

A character vector of column names present in `x`.

See Also

`llm_persona_split()`.

llm_persona_overview *A compact overview of a persona frame for display*

Description

Builds a small data frame summarizing the personas, suitable for a browse or selection table. By default it surfaces a few widely useful fields (an ideology score when present, plus party, age, race, region, and religion when the dictionary names them); pass `columns` to choose your own.

Usage

```
llm_persona_overview(x, columns = NULL)
```

Arguments

`x`	A persona data frame.
`columns`	Optional character vector of either column handles or question wordings to include. When `NULL`, a default set is used.

Value

A data frame with one row per persona.

See Also

anes_2024_personas.

llm_persona_split *Split one persona into labeled demographics and labeled answers*

Description

Extracts row `i` of a persona frame into two named character vectors: the demographic fields and the survey/attitude answers. Names are the question wording when the frame carries a `dictionary`, otherwise the column handles. Missing values are dropped, and any score/index column (e.g. `ideology_score`) is treated as metadata, not an answer. This is the shared field-extraction step; how the result becomes prompt text is up to the caller.

Usage

```
llm_persona_split(x, i, drop = "ideology_score")
```

Arguments

`x`	A persona data frame (see anes_2024_personas).
`i`	Integer row index.
`drop`	Character vector of column names to exclude from both parts (defaults to `"ideology_score"`).

Value

A list with two named character vectors, `demographics` and `responses`.

See Also

`llm_persona_demographic_fields()`, anes_2024_personas.

Examples

```
data(anes_2024_personas, package = "LLMR")
parts <- llm_persona_split(anes_2024_personas, 1)
names(parts$demographics)[1:3]
```

---

llm_preview *Preview a tidy LLM call without spending anything*

Description

Renders every row exactly as `llm_fn()` / `llm_mutate()` would (no API call, no file I/O), then reports a tidy, row-level summary: the rendered text, the roles, character counts, file presence and existence, the batch plan, and a list-column of `issues`. Problems that would only surface mid-run (a missing file, a `"file"` role combined with `.rows_per_prompt > 1`, an embedding config with row batching, `.return = "object"` with batching, a schema supplied without `.structured`, a template that references `NA` values or renders an empty prompt, a file part with no accompanying user text, or a tag name that collides with the batched `<row_N>` protocol) are collected per row so you see all of them at once rather than hitting the first error.

Usage

```
llm_preview(
  .data,
  prompt = NULL,
  .messages = NULL,
  .system_prompt = NULL,
  .config = NULL,
  .structured = FALSE,
  .schema = NULL,
  .tags = NULL,
  .return = c("columns", "text", "object"),
  .rows_per_prompt = 1L,
  rows = NULL,
  max_chars = 500L
)
```

Arguments

.data	A data.frame/tibble whose columns feed the glue templates.
prompt	A single glue template string (mutually exclusive with .messages).
.messages	A character vector of glue templates, optionally named by role ("system", "user", "assistant", "file"). Unnamed entries default to "user".
.system_prompt	Optional system string, prepended when a row has no "system" message.
.config	Optional llm_config(). When supplied, preview checks the embedding-vs-row-batching conflict; otherwise config-dependent checks are skipped.
.structured	Logical; if TRUE, the call would request structured JSON output. Used only to validate .schema/.tags combinations.
.schema	Optional JSON schema (list). Flagged if supplied without .structured = TRUE.
.tags	Optional character vector of tag names. Flagged if combined with .structured = TRUE (structured takes precedence).
.return	One of "columns", "text", "object". Only used to flag the unsupported "object" + batching combination.
.rows_per_prompt	Rows per call. 1 (default) means one call per row; > 1 or Inf packs rows into batched calls.
rows	Optional integer vector selecting which rows to render (default: all rows).
max_chars	Truncate each row's rendered preview to this many characters (default 500). Set higher to see full prompts.

Details

Batched data travels inside numbered <row_i>...</row_i> tags; the rowpack_id / rows_per_prompt / rowpack_row columns show how rows would be grouped into calls at the given .rows_per_prompt.

Value

A tibble of class llmr_preview, one row per previewed input row, with columns: row, ok (no issues), roles, rendered_preview, chars, has_file, file_ok, rowpack_id, rows_per_prompt, rowpack_row, and issues (a list-column of character vectors).

See Also

llm_render_messages(), llm_usage(), llm_failures().

Examples

```
df <- data.frame(text = c("a", "b", "c"), stringsAsFactors = FALSE)
llm_preview(df, prompt = "Classify: {text}", .rows_per_prompt = 2)
```

`llm_render_messages` *Render tidy messages without calling any API*

Description

Returns the per-row message objects that `llm_fn()` / `llm_mutate()` would build from `prompt` or `.messages`, using the same internal renderer they use. No request is sent and no file is read or encoded; a `"file"` role stays a (glued) path string. Use this to inspect templating, roles, and multimodal wiring before spending anything.

Usage

```
llm_render_messages(
  .data,
  prompt = NULL,
  .messages = NULL,
  .system_prompt = NULL,
  rows = NULL
)
```

Arguments

`.data`	A data.frame/tibble whose columns feed the `glue` templates.
`prompt`	A single `glue` template string (mutually exclusive with `.messages`).
`.messages`	A character vector of `glue` templates, optionally named by role (`"system"`, `"user"`, `"assistant"`, `"file"`). Unnamed entries default to `"user"`.
`.system_prompt`	Optional system string, prepended when a row has no `"system"` message.
`rows`	Optional integer vector selecting which rows to render (default: all rows).

Value

A list of length `length(rows)` (default `nrow(.data)`). Each element is either a bare character scalar (prompt only, no system) or a role-named character vector, identical to what the call path would dispatch.

See Also

`llm_preview()` for a row-level summary with issue flags and the batch plan; `llm_fn()`, `llm_mutate()`.

Examples

```
df <- data.frame(text = c("good", "bad"), stringsAsFactors = FALSE)
llm_render_messages(df, prompt = "Sentiment of: {text}")
llm_render_messages(
  df,
  .messages = c(system = "Be terse.", user = "Rate: {text}")
)
```

`llm_replicate` *Run the same prompt several times per row*

Description

Calls the model `.times` times for every row of `.data` (all replicates run through the parallel engine in one pass) and appends one column per replicate: `<output>_1`, `<output>_2`, ... Feed the result to `llm_agreement()` for per-row majority labels and overall reliability.

Usage

```
llm_replicate(
  .data,
  output,
  prompt,
  .config,
  .times = 3L,
  .system_prompt = NULL,
  ...
)
```

Arguments

`.data`	A data.frame / tibble.
`output`	Unquoted base name for the replicate columns.
`prompt`	A glue template string evaluated against the columns of `.data` (as in `llm_mutate()`).
`.config`	An llm_config object (generative).
`.times`	Number of replicates (default 3).
`.system_prompt`	Optional system message.
`...`	Passed to `call_llm_par()` (e.g. `tries`, `progress`).

Details

Replication only measures sampling variability if the model can vary: with `temperature = 0` (or a fixed `seed`) most providers return nearly identical draws, which inflates agreement. Conversely, for measurement purposes you may want exactly that check: high disagreement at low temperature signals a prompt the model finds genuinely ambiguous.

Value

`.data` with `.times` new character columns, `<output>_1 ... <output>_<.times>` (`NA` where a call failed).

See Also

`llm_agreement()`, `llm_mutate()`, `call_llm_par()`

Examples

```
## Not run:
cfg <- llm_config("groq", "openai/gpt-oss-20b", temperature = 1)
df <- tibble::tibble(text = c("I loved it", "Meh", "Terrible service"))
reps <- df |>
  llm_replicate(sentiment,
              prompt = "Sentiment of '{text}'. One word: positive, negative, or neutral.",
                 .config = cfg, .times = 5)
llm_agreement(reps, prefix = "sentiment")

## End(Not run)
```

---

llm_request_from_log *Rebuild a callable request from a logged record*

Description

Turns one parsed log record (the `rec` element of a `llm_log_read()` entry) back into the `config` and `messages` an experiments row needs, so an archived call can be re-issued live or matched against a config-driven call. The provider-specific request body is canonicalized to provider-neutral role/content turns and its generation parameters are recovered onto a reconstructed `llm_config()`.

Usage

```
llm_request_from_log(record, on_unsupported = c("error", "warn", "quiet"))
```

Arguments

`record`	A parsed log record: a list with at least `provider`, `model`, and `request` (as found in `llm_log_read(x)$records[[i]]$rec`).
`on_unsupported`	What to do when the request body uses a shape this reconstructor does not cover (so the rebuilt messages would be empty or lossy) – chiefly the OpenAI Responses API (`input`/`instructions`) and non-text content blocks (images, files). `"error"` (default) aborts; `"warn"` warns and returns the best-effort result; `"quiet"` returns it silently. Failing loudly is the default because a silently empty replay is worse than none.

Value

A list with `config` (an `llm_config()`), `messages` (a named character vector of role/content turns), and `complete` (a logical: `TRUE` when the body was fully reconstructed, `FALSE` when content was dropped). `NULL` when the record carries no request body.

See Also

`llm_log_read()`, `llm_request_hash()`

Examples

```
rec <- list(provider = "openai", model = "gpt-4o-mini",
            request = list(
              messages = list(list(role = "user", content = "Hi")),
              temperature = 0))
req <- llm_request_from_log(rec)
req$messages
```

`llm_request_hash` *Stable request hash for an LLM call*

Description

A request hash identifies the question asked. It keys on provider, model, the canonical role/content turns, attachment-content digests, and the generation parameters (`model_params` minus transport and internal knobs), plus any explicit tool, response-format, or schema signature. It is the key the archive layer uses for deduplication and replay, and the value to record when reproducibility of a specific call matters.

Usage

```
llm_request_hash(
  config = NULL,
  messages = NULL,
  provider = NULL,
  model = NULL,
  tools = NULL,
  response_format = NULL,
  schema = NULL,
  extra = NULL
)
```

Arguments

`config`	An `llm_config()` (its `provider`, `model`, and the answer-relevant entries of `model_params` are used), or `NULL` to supply `provider`/`model` directly and the generation parameters through `extra$params`.
`messages`	The messages or prompt sent. A character scalar, a named character vector of roles, or a list of `list(role=, content=)` turns – whatever was passed to `call_llm()`. Canonicalized to a provider-neutral list of role/content turns before hashing, so the message *shape* (a bare string vs a named vector vs a turn list) does not change the hash; only the roles and text do. File and image parts are keyed by a SHA-256 digest of their decoded bytes, not by their path or provider-specific wrapper.
`provider, model`	Provider and model ids. Taken from `config` when it is given; supply them directly (with `extra$params`) when hashing a call read from an audit log, where there is no `config`.

`tools` Optional provider tool definitions. Their signature is included when present.

`response_format` Optional response-format directive (e.g. a JSON-mode flag or object) that constrains the output.

`schema` Optional JSON Schema used for structured output.

`extra` Optional named list folded into the hashed object. The log side passes the call's generation parameters here as `extra$params` (a named list), which are consumed into the param key exactly as a config's `model_params` would be; a config, when given, takes precedence over `extra$params`. Any other entries are hashed as-is, to add identity-relevant fields the standard arguments do not cover.

Details

The hash is built over a canonical object and passed to `llm_hash()`, so it inherits that convention: construction order and S3 class do not matter, and the value does not depend on R's serialization format. API keys and volatile fields (timestamps, request ids) are never part of the input. Transport and internal knobs in `model_params` (see the source for the full list – e.g. `req_builder`, `response_modifier`, `timeout`, `api_url`, `max_tries`, `request_modifier`, `cache`, `use_responses_api`, `vertex`) are excluded, since they change how a call is issued, not what is asked.

Value

A 64-character lowercase SHA-256 hex string (see `llm_hash()`).

Scope

The hash keys on the canonical turns and generation parameters. It is designed so a call described by an `llm_config()` and the same call read from a logged provider request body agree for the common chat/completions path. It does *not* attempt to reconstruct every provider's exact translated body: calls that differ only in features outside this key – the OpenAI Responses API (`input`/`instructions`), full tool-call transcripts, or provider-injected defaults – may hash the same across the config and log sides. The archive layer's collision detection is the backstop for those cases: it flags when one replay key gathers records whose stored request hashes disagree.

The config side hashes the request as declared. A declared parameter that a provider builder discards as unsupported (with its printed note, e.g. `top_k` on OpenAI) is still part of the declared identity, so the config-side hash and the logged transmitted body then differ; set `no_change = TRUE` to send the parameter as declared, or leave it out of the config.

See Also

`llm_hash()`, `llm_response_record()`

Examples

```
cfg <- llm_config("openai", "gpt-4.1-mini", temperature = 0)
h1 <- llm_request_hash(cfg, "Hello")
# message order/content matters; temperature matters:
cfg2 <- llm_config("openai", "gpt-4.1-mini", temperature = 1)
identical(h1, llm_request_hash(cfg2, "Hello"))
```

`llm_response_record` *Flatten one LLM response to a provenance row*

Description

Turns a single llmr_response (or a caught error) into a one-row tibble with the columns the LLMR family treats as the response contract: `response_text`, `response_id`, `provider`, `model`, `model_version`, `finish_reason`, `sent_tokens`, `rec_tokens`, `total_tokens`, `reasoning_tokens`, `cached_tokens`, `success`, `error_message`, `duration_s`, `created_at`, and `request_hash`. It is the canonical flattening path: reuse it rather than reaching into a response's fields, so a sealed archive or an audit reconstructs the same columns everywhere.

Usage

```
llm_response_record(
  x,
  request = NULL,
  config = NULL,
  error = NULL,
  started_at = NULL,
  ended_at = NULL
)
```

Arguments

`x`	An llmr_response. If `x` is a condition (a caught error) it is treated as a failed call. Any other object yields a failed row whose message notes the unexpected class.
`request`	The messages or prompt that produced `x`, used together with `config` to fill `request_hash` via `llm_request_hash()`. `NULL` leaves `request_hash` as `NA`.
`config`	The `llm_config()` used for the call, for `request_hash` and to backfill `provider`/`model` when the response does not carry them.
`error`	Optional caught condition to record instead of (or alongside) a missing response; when supplied, the row is a failure carrying its message.
`started_at, ended_at`	Optional `POSIXct`/numeric timestamps. When both are given and the response does not report a duration, `duration_s` is their difference; `created_at` is `ended_at` when supplied.

Details

A failure is a record, not a gap. Pass a caught condition (or any non-response object) as `error`, or as `x`, and the row carries `success = FALSE` with the message in `error_message`; the call is never silently dropped.

Value

A one-row tibble with the response-contract columns.

See Also

`llm_request_hash()`, llmr_response, `llm_usage()`

Examples

```
r <- structure(
  list(text = "Hello!", provider = "openai", model = "demo",
       model_version = NA_character_, finish_reason = "stop",
       usage = list(sent = 12L, rec = 5L, total = 17L,
                    reasoning = NA_integer_, cached = NA_integer_),
       response_id = "resp_123", duration_s = 0.012),
  class = "llmr_response")
llm_response_record(r)
# a caught error is a row too:
llm_response_record(simpleError("boom"))
```

---

llm_tool *Define a tool the model may call*

Description

Wraps an R function with the name, description, and JSON-Schema argument specification that providers need for native tool calling. Pass a list of tools to `call_llm_tools()` (or to a `chat_session()` via that function), which executes the calls the model makes and returns the model's final answer.

Usage

```
llm_tool(fn, name, description, parameters = NULL, required = NULL)
```

Arguments

`fn`	The R function to expose. It is called with the model's arguments matched by name, so use the same parameter names as in `parameters`.
`name`	Tool name shown to the model (letters, digits, _, -).
`description`	One or two sentences telling the model what the tool does and when to use it. Write it for the model, not for a human reader; it is the only documentation the model sees.
`parameters`	Either a named list of JSON-Schema property definitions, e.g. `list(city = list(type = "string", description = "City name"))`, or a complete JSON-Schema object (a list with `type = "object"` and `properties`). A tool with no arguments may omit it.
`required`	Character vector of required argument names. Defaults to all parameter names when `parameters` is a property list.

Value

An object of class `llmr_tool`.

See Also

call_llm_tools(), tool_calls()

Examples

```
weather <- llm_tool(
  fn = function(city) paste0("22C and clear in ", city),
  name = "get_weather",
  description = "Current weather for a city.",
  parameters = list(city = list(type = "string", description = "City name"))
)
```

---

llm_tool_signature *A stable signature for a tool's identity*

Description

Returns a SHA-256 hash over a tool's name, description, and parameter schema (its implementation function is deliberately excluded, so the signature tracks the contract a model sees, not the R code). Use it to detect when a tool's advertised interface has changed, for example to pin a tool schema and notice a silent redefinition.

Usage

```
llm_tool_signature(tool)
```

Arguments

tool	An llmr_tool (from llm_tool()), or a list carrying name, description, and schema.

Value

A 64-character lowercase hex string.

See Also

llm_tool(), llm_hash().

Examples

```
tl <- llm_tool(function(x) x, "echo", "Echo a value",
               parameters = list(x = list(type = "string")))
llm_tool_signature(tl)
```

`llm_usage` *Summarize token usage and outcomes of an LLM run*

Description

Reads the diagnostic columns produced by `call_llm_par()` (and `call_llm_broadcast()` / `llm_fn()` with `.return = "columns"`) or by `llm_mutate()`, and returns a one-row tibble of counts and token totals. It reports **tokens, not money**: sent, received, total, and reasoning tokens are summed with `na.rm = TRUE` (correct under row batching, which attributes a batch's tokens to its first row and leaves the rest `NA`). To estimate cost, multiply these by your provider's current per-token prices yourself.

Usage

```
llm_usage(x, prefix = NULL, price_table = NULL)
```

Arguments

`x`	A data frame from `call_llm_par()` or `llm_mutate()`.
`prefix`	For an `llm_mutate()` result, the output column name whose diagnostics to summarize (e.g. `"answer"`). Inferred automatically when a single diagnostic block is present; required when several are.
`price_table`	Optional data frame you supply with your provider's current prices, holding columns `model`, `input`, and `output` (US dollars per **million** tokens), and optionally `cached` (price per million cached prompt tokens). When given, a `cost_estimate` column is added: cached tokens are billed at the `cached` rate (or the `input` rate if no `cached` column), the remaining sent tokens at `input`, and received tokens at `output`. LLMR ships no price list on purpose; prices change, and a stale bundled table would mislead silently.

Value

A one-row tibble: `n`, `n_ok`, `n_failed`, `ok_rate`, `n_truncated` (finish `"length"`), `n_filtered` (finish `"filter"`), `sent_tokens`, `rec_tokens`, `total_tokens`, `reasoning_tokens`, `cached_tokens` (prompt tokens served from the provider's cache, when reported), `n_unknown_tokens` (successful rows for which the provider reported no token usage, so the token sums above understate the truth), `duration_s`, (when a batch id column is present) `rowpack_calls` and `rows_per_rowpack`, and (when `price_table` is supplied) `cost_estimate` in the table's currency.

See Also

`llm_failures()`, `llm_preview()`, `llm_par_resume()`.

Examples

```
res <- tibble::tibble(
  success = c(TRUE, TRUE, FALSE),
  finish_reason = c("stop", "length", "error:rate_limit"),
  sent_tokens = c(10L, 12L, NA_integer_),
  rec_tokens = c(5L, 7L, NA_integer_),
  total_tokens = c(15L, 19L, NA_integer_),
  reasoning_tokens = c(NA_integer_, NA_integer_, NA_integer_),
  duration = c(0.4, 0.5, 0.1)
)
llm_usage(res)
```

---

llm_validate_persona_frame

Check that a data frame follows the persona contract

Description

Verifies the minimum a persona frame needs: it is a data frame with at least one row. It also reports, via a message when `verbose`, whether the optional `dictionary` and `demographic_fields` attributes are present. It is tolerant: a frame without those attributes is still usable (handles stand in for questions, and demographics are guessed from common names).

Usage

```
llm_validate_persona_frame(x, verbose = FALSE)
```

Arguments

`x`	A candidate persona data frame.
`verbose`	Logical; if `TRUE`, report which optional attributes are present.

Value

`TRUE` invisibly if `x` is a usable persona frame; otherwise it raises an error.

See Also

`llm_persona_split()`, anes_2024_personas.

llm_validate_structured_col *Validate structured JSON objects against a JSON Schema (locally)*

Description

Adds `structured_valid` (logical) and `structured_error` (chr) by validating each row's `structured_data` against `schema`. No provider calls are made.

Usage

```
llm_validate_structured_col(
  .data,
  schema,
  structured_list_col = "structured_data"
)
```

Arguments

`.data`	A data.frame with a `structured_data` list-column.
`schema`	JSON Schema (R list)
`structured_list_col`	Column name with parsed JSON. Default "structured_data".

Value

`.data` with two added columns: `structured_valid` (logical) and `structured_error` (character; `NA` when valid).

See Also

`llm_parse_structured_col()`, `llm_fn_structured()`

parse_embeddings *Parse Embedding Response into a Numeric Matrix*

Description

Converts the embedding response data to a numeric matrix.

Usage

```
parse_embeddings(embedding_response)
```

Arguments

embedding_response
: The response returned from an embedding API call.

Value

A numeric matrix of embeddings with column names as sequence numbers.

Examples

```
## Not run:
  text_input <- c("Political science is a useful subject",
                  "We love sociology",
                  "German elections are different",
                  "A student was always curious.")

  # Configure the embedding API provider (example with Voyage API).
  # The key is read from the VOYAGE_API_KEY environment variable.
  voyage_config <- llm_config(
    provider = "voyage",
    model = "voyage-3.5-lite"
  )

  embedding_response <- call_llm(voyage_config, text_input)
  embeddings <- parse_embeddings(embedding_response)
  # Additional processing:
  embeddings |> cor() |> print()

## End(Not run)
```

---

print.llm_config *Print an LLM configuration with the API key masked*

Description

Configurations never print their key: a literal key shows as <llmr_secret: literal> and an environment reference as <llmr_secret: env:VARNAME>, so configs are safe to print in scripts, logs, and rendered documents.

Usage

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

## S3 method for class 'llm_config'
format(x, ...)
```

Arguments

x	An llm_config object.
...	Ignored.

Value

x invisibly (for print); a character vector (for format).

---

report *Draft a methods-section report from an LLMR-family result object*

Description

A shared generic across the LLMR method packages. It returns the methods-section prose and tables for a result object (what a paper's appendix would print), as distinct from diagnostics(), which returns the machine-readable numbers.

Usage

```
report(x, ...)

## S3 method for class 'llmr_experiment'
report(x, prefix = NULL, task = NULL, ...)
```

Arguments

x	An LLMR experiment or an object returned by an LLMR method package.
...	Passed to methods (some methods require extra arguments, e.g. some LLMRcontent report methods require the gold set and protocol).
prefix	For an llmr_experiment, the output-column prefix when the object uses llm_mutate() diagnostic names. It is inferred when possible.
task	For an llmr_experiment, an optional clause describing the task, such as "to classify open-ended responses".

Details

LLMR implements this generic for llmr_experiment objects returned by call_llm_par() and its wrappers. The method packages (LLMRcontent, LLMRpanel) provide methods for their own result classes.

Value

A method-defined report object, by convention a character vector with a print method. The llmr_experiment method returns a character scalar containing a draft methods paragraph.

See Also

diagnostics(), llm_usage(), llm_log_enable()

Examples

```
## Not run:
results <- call_llm_broadcast(cfg, c("First prompt", "Second prompt"))
cat(report(results, task = "to classify short texts"))

## End(Not run)
```

---

reset *Reset a stateful object to its initial position*

Description

A shared generic across the LLMR method packages, for objects that carry consumable state. LLMR defines the generic and an erroring default only; a method package (for example, an archive replayer) registers the concrete method that restores its state.

Usage

```
reset(x, ...)
```

Arguments

x	An object with resettable state.
...	Passed to methods.

Value

x, invisibly, with its state restored.

Examples

```
## Not run:
# An archive replayer whose queue has advanced can be rewound:
reset(replayer)

## End(Not run)
```

reset_llm_parallel *Reset Parallel Environment*

Description

Resets the future plan to sequential processing.

Usage

```
reset_llm_parallel(verbose = FALSE)
```

Arguments

verbose	Logical. If TRUE, prints reset information.

Value

Invisibly returns the future plan that was in place before resetting to sequential.

Examples

```
## Not run:
  # Setup parallel processing
  old_plan <- setup_llm_parallel(workers = 2)

  # Do some parallel work...

  # Reset to sequential
  reset_llm_parallel(verbose = TRUE)

  # Optionally restore the specific old_plan if it was non-sequential
  # future::plan(old_plan)

## End(Not run)
```

setup_llm_parallel *Setup Parallel Environment for LLM Processing*

Description

Convenience function to set up the future plan for optimal LLM parallel processing. Automatically detects system capabilities and sets appropriate defaults.

Usage

```
setup_llm_parallel(workers = NULL, strategy = NULL, verbose = FALSE)
```

Arguments

workers	Integer. Number of workers to use. If NULL, auto-detects optimal number (availableCores - 1, capped at 8). If called as `setup_llm_parallel(4)`, the single numeric positional argument is interpreted as `workers`.
strategy	Character. The future strategy to use. Options: "multisession", "multicore", "sequential". If NULL (default), automatically chooses "multisession".
verbose	Logical. If TRUE, prints setup information.

Value

Invisibly returns the previous future plan.

Examples

```
## Not run:
  # Automatic setup
  setup_llm_parallel()

  # Manual setup with specific workers
  setup_llm_parallel(workers = 4, verbose = TRUE)

  # Force sequential processing for debugging
  setup_llm_parallel(strategy = "sequential")

  # Restore old plan if needed
  reset_llm_parallel()

## End(Not run)
```

---

tool_calls *Extract tool calls from a response*

Description

When a model decides to call tools, `finish_reason(x)` is `"tool"` and this helper returns what it asked for. `call_llm_tools()` uses it internally; it is exported so custom loops can be built on it.

Usage

```
tool_calls(x)
```

Arguments

x	An llmr_response object.

Value

A list with one element per requested call: `list(id =, name =, arguments =)` where `arguments` is a named list. `list()` when the response contains no tool calls.

See Also

call_llm_tools(), llm_tool()

---

transcript_as_messages *Build a role-flipped message array from a multi-speaker transcript*

Description

Renders a shared, speaker-attributed transcript into a message array from one speaker's point of view, the way a stateful chat is post-trained to read it: the current speaker's own past turns become `assistant` messages (their own voice), and every other speaker's turns become `user` messages labeled with the speaker's name. An optional `system` block (persona, instructions) leads the array, and an optional `instruction` (the current question or "your turn" cue) closes it as a trailing `user` message. The result is passed through `ensure_alternating_messages()` so it is provider-safe.

Usage

```
transcript_as_messages(
  transcript,
  speaker,
  system = NULL,
  instruction = NULL,
  label = function(id) paste0(id, ": "),
  sanitize = TRUE,
  ensure_final_user = TRUE
)
```

Arguments

`transcript`	A data.frame/tibble with character columns `speaker` and `text`, in chronological order.
`speaker`	The current speaker's id (matched against `transcript$speaker`). Rows with this speaker become `assistant` turns; all others become labeled `user` turns.
`system`	Optional system text (persona, standing instructions) placed as the leading `system` message. `NULL` to omit.
`instruction`	Optional trailing instruction (current question, "your turn" cue) placed as the final `user` message. `NULL` to omit.
`label`	A function `(speaker_id) -> prefix` producing the in-content label for other speakers' turns; the default renders `"<id>: "`. The speaker's own turns are never labeled.
`sanitize`	If `TRUE` (default), strip forged role headers and chat control tokens from other speakers' text before embedding.

ensure_final_user
: If TRUE (default) and no instruction is supplied and the last turn is the speaker's own (an assistant turn), append a minimal user cue so the array ends on a user turn (a sound next-turn generation boundary). Set FALSE for verbatim archival conversion. Missing text is coerced to ""; a missing speaker is an error.

Details

Putting the speaker's own turns in the assistant role gives the model a structural handle on "what I already said", which reduces self-repetition in multi-agent conversations relative to flattening the whole transcript into one user block. Other speakers' text stays in user turns (never assistant), which preserves the trust boundary and is sanitized against forged role headers.

Value

A list of message objects (list(role=, content=)) suitable for call_llm(), guaranteed alternating and user-leading after any system.

See Also

ensure_alternating_messages(), call_llm()

Examples

```
tx <- data.frame(
  speaker = c("Ana", "Ben", "Cy"),
  text    = c("I think we ban it.", "Too costly for renters.", "Phase it in."),
  stringsAsFactors = FALSE
)
# From Ben's perspective: his line is assistant, Ana's and Cy's are user.
transcript_as_messages(tx, speaker = "Ben",
                       system = "You are Ben, a renter.",
                       instruction = "Your turn, Ben.")
```

Index