Package 'LLMR'

May 21, 2026

Title Interface for Large Language Model APIs in R

Version 0.6.4

Depends R (>= 4.1.0)

Description Provides a unified interface to large language models across multiple providers. Supports text generation, tidy data workflows, structured output with optional JSON Schema validation, XML-like tag extraction, and embeddings. Includes chat sessions, consistent error handling, and parallel batch tools.

License MIT + file LICENSE

Encoding UTF-8

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

Suggests testthat (>= 3.0.0), roxygen2 (>= 7.1.2), httptest2, progressr, knitr, rmarkdown, ggplot2, R.rsp, jsonvalidate

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 2026-05-21 18:02:03 UTC

Contents

`bind_tools` *Bind tools to a config (provider-agnostic)*

Description

Bind tools to a config (provider-agnostic)

Usage

```
bind_tools(config, tools, tool_choice = NULL)
```

Arguments

`config`	llm_config
`tools`	list of tools (each with name, description, and parameters/input_schema)
`tool_choice`	optional tool_choice spec (provider-specific shape)

Value

modified llm_config

`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 (recycled against user prompts). Missing/NA values are ignored; 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. Extended thinking is supported: provide `thinking_budget` and `include_thoughts = TRUE` to include a `content` block of type `"thinking"` in the response; LLMR sets the beta header automatically.
- **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.
- **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)
a_cfg <- llm_config("anthropic", "claude-sonnet-4-6",
                    max_tokens = 5000,
                    thinking_budget = 16000,
                    include_thoughts = TRUE)
r2 <- call_llm(a_cfg, "Compute 87*93 in your head. Give only the final number.")
# thinking (if present): r2$raw$content[[1]]$thinking
# final text:            r2$raw$content[[2]]$text

## 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

A tibble with columns: message_index (metadata), provider, model, all model parameters, response_text, raw_response_json, success, error_message.

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 = "openai", model = "gpt-4.1-nano")

  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

A tibble with columns: config_index (metadata), provider, model, all varying model parameters, response_text, raw_response_json, success, error_message.

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,
  json_output = NULL,
  start_jitter = 5
)
```

Arguments

`experiments`	A tibble/data.frame with required list-columns 'config' (llm_config objects) and 'messages' (character vector OR message list).
`simplify`	Whether to cbind 'experiments' to the output data frame or not.
`tries`	Integer. Number of retries for each call. 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.
`max_workers`	Integer. Maximum number of parallel workers. If NULL, auto-detects.
`progress`	Logical. If TRUE, shows progress bar.

`json_output`	Deprecated. Raw JSON string is always included as raw_response_json. This parameter is kept for backward compatibility but has no effect.
`start_jitter`	Calls are made after a uniformly distributed delay between 0 and `start_jitter` seconds.

Value

A tibble containing all original columns plus:

- `response_text` – assistant text (or `NA` on failure)
- `raw_response_json` – raw JSON string
- `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
- `response` – the full `llmr_response` object (or `NA` 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()`.

See Also

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

`call_llm_robust` *Robustly Call LLM API (Simple Retry)*

Description

Wraps `call_llm` to handle rate-limit errors (HTTP 429 or related "Too Many Requests" messages). It retries the call a specified number of times, using exponential backoff. You can also choose to cache responses if you do not need fresh results each time.

Usage

```
call_llm_robust(
  config,
  messages,
  tries = 5,
  wait_seconds = 10,
  backoff_factor = 5,
  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. Number of retries before giving up. Default is 5.
wait_seconds	Numeric. Initial wait time (seconds) before the first retry. Default is 10.
backoff_factor	Numeric. Multiplier for wait time after each failure. Default is 5.
verbose	Logical. If TRUE, prints the full API response.
memoize	Logical. If TRUE, calls are cached to avoid repeated identical requests. Default is FALSE.

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. cache_llm_call for a memoised version that avoids repeated requests. llm_config to create the configuration object. chat_session for stateful, interactive conversations.

Examples

```
## Not run:
robust_resp <- call_llm_robust(
config = llm_config("openai", "gpt-5-nano"),
messages = list(list(role = "user", content = "Hello, LLM!")),
tries = 5,
wait_seconds = 10,
memoize = FALSE
)
print(robust_resp)
as.character(robust_resp)

## 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

A tibble with columns: swept_param_name, the varied parameter column, provider, model, all other model parameters, response_text, raw_response_json, success, error_message.

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 = "openai", model = "gpt-4.1-nano")

  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)
```

---

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

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), Anthropic, and Gemini.

Usage

```
enable_structured_output(
  config,
  schema = NULL,
  name = "llmr_schema",
  method = c("auto", "json_mode", "tool_call"),
  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".
method	One of c("auto","json_mode","tool_call"). "auto" chooses the best per provider. You rarely need to change this.
strict	Logical. Request strict validation when supported (OpenAI-compatible).

Value

Modified llm_config.

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()

---

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)
```

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.
verbose	Logical. If TRUE, prints progress messages. Default is TRUE.

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 embedding fails for certain texts, those rows will be filled with NA values. The matrix will always have the same number of rows as the input texts. Returns NULL if no embeddings were successfully generated.

See Also

llm_config to create the embedding configuration. parse_embeddings to convert the raw response to a numeric matrix.

Examples

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

  embed_cfg <- llm_config(
    provider = "voyage",
    model = "voyage-3.5-lite",
    embedding = TRUE,
    api_key = Sys.getenv("VOYAGE_API_KEY")
  )

  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.
- `finish_reason`: one of `"stop"`, `"length"`, `"filter"`, `"tool"`, `"other"`.
- `usage`: list with integers `sent`, `rec`, `total`, `reasoning` (if available).
- `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.

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)`. Missing values are `NA`.

`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)
## Not run:
fr <- finish_reason(r)

## End(Not run)
## Not run:
u <- tokens(r)
u$total

## End(Not run)
## Not run:
if (is_truncated(r)) message("Increase max_tokens")

## 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, ...)

## 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.
`...`	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, and invisibly return it.

`$send_structured(text, schema, ..., role = "user", .fields = NULL, .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.

`$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. One of: "openai", "anthropic", "gemini", "groq", "together", "voyage" (embeddings only), "deepseek", "xai", "ollama".
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	Character scalar. Provider API key.
troubleshooting	Logical. If TRUE, prints the full request payloads (including your API key!) for debugging. **Use with extreme caution.**
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. If NULL, LLMR 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 provider-specific parameters (e.g., temperature, top_p, max_tokens, top_k, repetition_penalty, reasoning_effort, api_url, etc.). Values are forwarded verbatim unless documented shims apply. For Anthropic extended thinking, supply thinking_budget (canonical; mapped to thinking.budget_tokens) together with include_thoughts = TRUE to request the thinking block in the response.

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).

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_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,
  ...,
  .return = c("text", "columns", "object")
)
```

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).
.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).

Value

For generative mode:

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

See Also

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

Examples

```
## Not run:
words <- c("excellent", "awful")
cfg <- llm_config("openai", "gpt-4.1-nano", 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
)
```

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.

See Also

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

`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"),
  .structured = FALSE,
  .schema = NULL,
  .fields = NULL,
  .tags = NULL,
  ...
)
```

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.
.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.
...	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.

Value

.data with the new column(s) appended.

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("openai", "gpt-4.1-nano",
                  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("inst/extdata/cat.png", "inst/extdata/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,
  ...
)
```

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

Shorthand syntax

Like llm_mutate(), this function supports shorthand syntax:

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

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,
  ...
)
```

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`.

... 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.

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("openai", "gpt-4.1-nano", 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_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".

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)
  voyage_config <- llm_config(
    provider = "voyage",
    model = "voyage-3.5-lite",
    api_key = Sys.getenv("VOYAGE_API_KEY")
  )

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

## 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)
```

Index