

Package ‘LLMR’

August 26, 2025

Title Interface for Large Language Model APIs in R

Version 0.6.0

Depends R (>= 4.1.0)

Description Provides a unified interface to large language models across multiple providers. Supports text generation, structured output with optional JSON Schema validation, and embeddings. Includes tidyverse-friendly helpers, chat session, 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 7.3.2

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 2025-08-26 06:20:02 UTC

Contents

bind_tools	2
build_factorial_experiments	3

cache_llm_call	4
call_llm	5
call_llm_broadcast	7
call_llm_compare	8
call_llm_par	9
call_llm_par_structured	11
call_llm_robust	12
call_llm_sweep	13
disable_structured_output	14
enable_structured_output	14
get_batched_embeddings	15
llmr_response	16
llm_api_key_env	18
llm_chat_session	19
llm_config	21
llm_fn	22
llm_fn_structured	24
llm_mutate	25
llm_mutate_structured	27
llm_parse_structured	28
llm_parse_structured_col	29
llm_validate_structured_col	30
log_llm_error	30
parse_embeddings	31
reset_llm_parallel	32
setup_llm_parallel	33
Index	34

bind_tools	<i>Bind tools to a config (provider-agnostic)</i>
------------	---

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
<i>Build Factorial Experiment Design</i>

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, message_label, and repetition. Ready for use with call_llm_par().

Examples

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

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

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

## End(Not run)
```

cache_llm_call

Cache LLM API Calls

Description

A memoised version of [call_llm](#) to avoid repeated identical requests.

Usage

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

Arguments

config	An llm_config object from llm_config .
messages	A list of message objects or character vector for embeddings.
verbose	Logical. If TRUE, prints the full API response (passed to call_llm).

Details

- Requires the memoise package. Add memoise to your package's DESCRIPTION.
- Clearing the cache can be done via `memoise::forget(cache_llm_call)` or by restarting your R session.

Value

The (memoised) response object from [call_llm](#).

Examples

```
## Not run:
# Using cache_llm_call:
response1 <- cache_llm_call(my_config, list(list(role="user", content="Hello!")))
# Subsequent identical calls won't hit the API unless we clear the cache.
response2 <- cache_llm_call(my_config, list(list(role="user", content="Hello!")))

## End(Not run)
```

call_llm	<i>Call an LLM (chat/completions or embeddings) with optional multi-modal input</i>
----------	---

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)
```

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.
- **Gemini (REST):** `systemInstruction` is supported; user parts use `text/inlineData(mimeType,data)`; responses are set to `responseMimeType = "text/plain"`.
- **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.

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

## 4) Embeddings
e_cfg <- llm_config("voyage", "voyage-large-2",
  embedding = TRUE)
emb_raw <- call_llm(e_cfg, c("first", "second"))
emb_mat <- parse_embeddings(emb_raw)

## 5) 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	<i>Parallel API calls: Fixed Config, Multiple Messages</i>
--------------------	--

Description

Broadcasts different messages using the same configuration in parallel. Perfect for batch processing different prompts with consistent settings. This function requires setting up the parallel environment using `setup_llm_parallel`.

Usage

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

Arguments

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

Value

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

Parallel Workflow

All parallel functions require the future backend to be configured. The recommended workflow is:

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.

See Also

[setup_llm_parallel](#), [reset_llm_parallel](#)

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. This function requires setting up the parallel environment using `setup_llm_parallel`.

Usage

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

Arguments

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

Value

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

Parallel Workflow

All parallel functions require the future backend to be configured. The recommended workflow is:

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.

See Also

[setup_llm_parallel](#), [reset_llm_parallel](#)

Examples

```
## Not run:
# Compare different models
config1 <- llm_config(provider = "openai", model = "gpt-4o-mini")
config2 <- llm_config(provider = "openai", model = "gpt-4.1-nano")

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. This function requires setting up the parallel environment using `setup_llm_parallel`.

Usage

```
call_llm_par(
  experiments,
  simplify = TRUE,
  tries = 10,
  wait_seconds = 2,
  backoff_factor = 3,
  verbose = FALSE,
  memoize = FALSE,
  max_workers = NULL,
  progress = FALSE,
  json_output = NULL
)
```

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 2.
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.

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

All parallel functions require the future backend to be configured. The recommended workflow is:

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.

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", "llama-3.3-70b-versatile")

experiments <- tibble::tibble(
  model_id = c("gpt-4.1-nano", "groq-llama-3.3"),
  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() .

`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

<code>config</code>	An <code>llm_config</code> object from <code>llm_config</code> .
<code>messages</code>	A list of message objects (or character vector for embeddings).
<code>tries</code>	Integer. Number of retries before giving up. Default is 5.
<code>wait_seconds</code>	Numeric. Initial wait time (seconds) before the first retry. Default is 10.
<code>backoff_factor</code>	Numeric. Multiplier for wait time after each failure. Default is 5.
<code>verbose</code>	Logical. If TRUE, prints the full API response.
<code>memoize</code>	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-4o-mini"),
  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	<i>Parallel API calls: Parameter Sweep - Vary One Parameter, Fixed Message</i>
----------------	--

Description

Sweeps through different values of a single parameter while keeping the message constant. Perfect for hyperparameter tuning, temperature experiments, etc. This function requires setting up the parallel environment using `setup_llm_parallel`.

Usage

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

Arguments

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

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

All parallel functions require the future backend to be configured. The recommended workflow is:

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.

See Also

[setup_llm_parallel](#), [reset_llm_parallel](#)

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

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.

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. 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-large-2-instruct",
  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_api_key_env

Declare an API key sourced from an environment variable

Description

Use this when creating an LLM config to avoid placing secrets inside R objects. Store your key in your shell or in `~/ .Renvi` and reference it here by name.

Usage

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

Arguments

var	Name of the environment variable (e.g., "OPENAI_API_KEY").
required	If TRUE, missing variables cause an authentication error at call time.
default	Optional default if the environment variable is not set.

Value

An internal secret handle to be used as `api_key = llm_api_key_env("VARNAME")`.

llm_chat_session	<i>Chat Session Object and Methods</i>
------------------	--

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 <code>llm_chat_session</code> 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-4o-mini")
  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

<code>provider</code>	Character scalar. One of: "openai", "anthropic", "gemini", "groq", "together", "voyage" (embeddings only), "deepseek", "xai".
<code>model</code>	Character scalar. Model name understood by the chosen provider. (e.g., "gpt-4o-mini", "o4-mini", "claude-3.7", "gemini-2.0-flash", etc.)
<code>api_key</code>	Character scalar. Provider API key.
<code>troubleshooting</code>	Logical. If TRUE, prints the full request payloads (including your API key!) for debugging. Use with extreme caution.
<code>base_url</code>	Optional character. Back-compat alias; if supplied it is stored as <code>api_url</code> in <code>model_params</code> and overrides the default endpoint.
<code>embedding</code>	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".
<code>no_change</code>	Logical. If TRUE, LLMR never auto-renames/adjusts provider parameters. If FALSE (default), well-known compatibility shims may apply (e.g., renaming OpenAI's <code>max_tokens</code> → <code>max_completion_tokens</code> after a server hint; see <code>call_llm()</code> notes).
<code>...</code>	Additional provider-specific parameters (e.g., <code>temperature</code> , <code>top_p</code> , <code>max_tokens</code> , <code>top_k</code> , <code>repetition_penalty</code> , <code>reasoning_effort</code> , <code>api_url</code> , etc.). Values are forwarded verbatim unless documented shims apply.

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.

Endpoint overrides

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

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-4o-mini",
  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", "text-embedding-004")

# Force embeddings even if model name does not contain "embedding"
e_cfg2 <- llm_config("voyage", "voyage-large-2", embedding = TRUE)

## 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

<code>x</code>	A character vector <i>or</i> a data.frame/tibble.
<code>prompt</code>	A glue template string. With a data-frame you may reference columns (<code>{col}</code>); with a vector the placeholder is <code>{x}</code> .
<code>.config</code>	An llm_config object.
<code>.system_prompt</code>	Optional system message (character scalar).
<code>...</code>	Passed unchanged to call_llm_broadcast() (e.g. <code>tries</code> , <code>progress</code> , <code>verbose</code>).
<code>.return</code>	One of <code>c("text", "columns", "object")</code> . "columns" returns a tibble of diagnostic columns; "text" returns a character vector; "object" returns a list of <code>llmr_response</code> (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\(\)](#), [setup_llm_parallel\(\)](#), [call_llm_broadcast\(\)](#)

Examples

```
if (interactive()) {
  words <- c("excellent", "awful")
  cfg <- llm_config("openai", "gpt-4o-mini", temperature = 0)
  llm_fn(words, "Classify '{x}' as Positive/Negative.",
    cfg,
    .system_prompt="One word.",
    .return="columns")
}
```

llm_fn_structured	<i>Vectorized structured-output LLM</i>
-------------------	---

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

<code>x</code>	A character vector <i>or</i> a data.frame/tibble.
<code>prompt</code>	A glue template string. With a data-frame you may reference columns (<code>{col}</code>); with a vector the placeholder is <code>{x}</code> .
<code>.config</code>	An <code>llm_config</code> object.
<code>.system_prompt</code>	Optional system message (character scalar).
<code>...</code>	Passed unchanged to <code>call_llm_broadcast()</code> (e.g. <code>tries</code> , <code>progress</code> , <code>verbose</code>).
<code>.schema</code>	Optional JSON Schema list; if <code>NULL</code> , only JSON object is enforced.
<code>.fields</code>	Optional fields to hoist from parsed JSON (supports nested paths).
<code>.local_only</code>	If <code>TRUE</code> , do not send schema to the provider (parse/validate locally).
<code>.validate_local</code>	If <code>TRUE</code> and <code>.schema</code> provided, validate locally.

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"),
  ...
)
```

Arguments

<code>.data</code>	A <code>data.frame</code> / <code>tibble</code> .
<code>output</code>	Unquoted name that becomes the new column (generative) <i>or</i> the prefix for embedding columns.
<code>prompt</code>	Optional glue template string for a single user turn; reference any columns in <code>.data</code> (e.g. <code>"{id}. {question}\nContext: {context}"</code>). Ignored if <code>.messages</code> is supplied.
<code>.messages</code>	Optional named character vector of glue templates to build a multi-turn message, using roles in <code>c("system", "user", "assistant", "file")</code> . Values are glue templates evaluated per-row; all can reference multiple columns. For multimodal, use role <code>"file"</code> with a column containing a path template.
<code>.config</code>	An <code>llm_config</code> object (generative or embedding).
<code>.system_prompt</code>	Optional system message sent with every request when <code>.messages</code> does not include a system entry.
<code>.before</code> , <code>.after</code>	Standard <code>dplyr::relocate</code> helpers controlling where the generated column(s) are placed.
<code>.return</code>	One of <code>c("columns", "text", "object")</code> . For generative mode, controls how results are added. <code>"columns"</code> (default) adds text plus diagnostic columns; <code>"text"</code> adds a single text column; <code>"object"</code> adds a list-column of <code>llm_response</code> objects.
<code>...</code>	Passed to the underlying calls: <code>call_llm_broadcast()</code> in generative mode, <code>get_batched_embeddings()</code> 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()`. Parallel execution follows the active **future** plan; see `setup_llm_parallel()`.
- **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`, `_code`, `_param`, `_t`.

Value

.data with the new column(s) appended.

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-4o-mini",
  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
  )

## 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

<code>.data</code>	A <code>data.frame</code> / <code>tibble</code> .
<code>output</code>	Unquoted name that becomes the new column (generative) <i>or</i> the prefix for embedding columns.
<code>prompt</code>	Optional glue template string for a single user turn; reference any columns in <code>.data</code> (e.g. <code>"{id}. {question}\nContext: {context}"</code>). Ignored if <code>.messages</code> is supplied.
<code>.messages</code>	Optional named character vector of glue templates to build a multi-turn message, using roles in <code>c("system", "user", "assistant", "file")</code> . Values are glue templates evaluated per-row; all can reference multiple columns. For multimodal, use role <code>"file"</code> with a column containing a path template.
<code>.config</code>	An <code>llm_config</code> object (generative or embedding).
<code>.system_prompt</code>	Optional system message sent with every request when <code>.messages</code> does not include a system entry.
<code>.before</code> , <code>.after</code>	Standard <code>dplyr::relocate</code> helpers controlling where the generated column(s) are placed.
<code>.schema</code>	Optional JSON Schema list; if <code>NULL</code> , only JSON object is enforced.
<code>.fields</code>	Optional fields to hoist (supports nested paths).
<code>...</code>	Passed to the underlying calls: <code>call_llm_broadcast()</code> in generative mode, <code>get_batched_embeddings()</code> in embedding mode.

`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

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

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.

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.

llm_validate_structured_col	<i>Validate structured JSON objects against a JSON Schema (locally)</i>
-----------------------------	---

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".

log_llm_error	<i>Log LLMR Errors</i>
---------------	------------------------

Description

Logs an error with a timestamp for troubleshooting.

Usage

```
log_llm_error(err)
```

Arguments

- err An error object.

Value

Invisibly returns NULL.

Examples

```
## Not run:
# Example of logging an error by catching a failure:
# Use a deliberately fake API key to force an error
config_test <- llm_config(
  provider = "openai",
  model = "gpt-3.5-turbo",
  api_key = "FAKE_KEY",
  temperature = 0.5,
  top_p = 1,
  max_tokens = 30
)

tryCatch(
  call_llm(config_test, list(list(role = "user", content = "Hello world!"))),
  error = function(e) log_llm_error(e)
)

## End(Not run)
```

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-large-2",
  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	<i>Reset Parallel Environment</i>
--------------------	-----------------------------------

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	<i>Setup Parallel Environment for LLM Processing</i>
--------------------	--

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

Arguments

strategy	Character. The future strategy to use. Options: "multisession", "multicore", "sequential". If NULL (default), automatically chooses "multisession".
workers	Integer. Number of workers to use. If NULL, auto-detects optimal number (availableCores - 1, capped at 8).
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

`as.character.llmr_response`
 (`llmr_response`), 16
`as.data.frame.llm_chat_session`
 (`llm_chat_session`), 19

`bind_tools`, 2
`build_factorial_experiments`, 3, 10

`cache_llm_call`, 4, 12
`call_llm`, 4, 5, 12, 22
`call_llm()`, 18, 20
`call_llm_broadcast`, 7, 10
`call_llm_broadcast()`, 23–26, 28
`call_llm_compare`, 8, 10
`call_llm_par`, 9, 22
`call_llm_par()`, 11
`call_llm_par_structured`, 11
`call_llm_robust`, 6, 12, 15, 22
`call_llm_robust()`, 18–20
`call_llm_sweep`, 10, 13
`chat_session`, 12
`chat_session(llm_chat_session)`, 19

`disable_structured_output`, 14
`dplyr::relocate`, 25, 28

`enable_structured_output`, 14

`finish_reason`, 6
`finish_reason(llmr_response)`, 16

`get_batched_embeddings`, 15, 22
`get_batched_embeddings()`, 25, 26, 28

`head.llm_chat_session`
 (`llm_chat_session`), 19

`is_truncated(llmr_response)`, 16

`llm_api_key_env`, 18
`llm_chat_session`, 6, 19, 22

`llm_chat_session()`, 18
`llm_config`, 4–6, 12, 15, 16, 19, 21, 23–25, 28
`llm_config()`, 18, 20
`llm_fn`, 22
`llm_fn()`, 18, 20, 24
`llm_fn_structured`, 24
`llm_mutate`, 25
`llm_mutate()`, 18, 20, 23, 27
`llm_mutate_structured`, 27
`llm_parse_structured`, 28
`llm_parse_structured_col`, 29
`llm_validate_structured_col`, 30
`llmr_response`, 16, 28
`log_llm_error`, 30

`parse_embeddings`, 6, 16, 31
`parse_embeddings()`, 5
`print.llm_chat_session`
 (`llm_chat_session`), 19
`print.llmr_response(llmr_response)`, 16

`reset_llm_parallel`, 7, 9, 10, 14, 32

`setup_llm_parallel`, 7, 9, 10, 14, 33
`setup_llm_parallel()`, 23, 26
`summary.llm_chat_session`
 (`llm_chat_session`), 19

`tail.llm_chat_session`
 (`llm_chat_session`), 19
`tokens`, 6
`tokens(llmr_response)`, 16