Package 'LLMR'

July 17, 2025

Title Interface for Large Language Model APIs in R

Version 0.5.0

Depends R (>= 4.1.0)

Description Provides a unified interface to interact with multiple Large Language Model (LLM) APIs. The package supports text generation, embeddings, parallelization, as well as tidyverse integration. Users can switch between different LLM providers seamlessly within R workflows, or call multiple models in parallel. The package enables creation of LLM agents for automated tasks and provides consistent error handling across all supported APIs. APIs include 'OpenAI' (see <https://platform.openai.com/docs> for details), 'Anthropic' (see <https://docs.anthropic.com/en/api/getting-started> for details), 'Groq' (see <https://console.groq.com/docs/api-reference> for details), 'Together AI' (see <https://docs.together.ai/docs/quickstart> for details), 'DeepSeek' (see <https://api-docs.deepseek.com> for details), 'Gemini' (see <https://aistudio.google.com> for details), 'xAI' (see <https://docs.x.ai/> for details), and 'Voyage AI' (see <https://docs.voyageai.com/docs/introduction> for details).

License MIT + file LICENSE

Encoding UTF-8

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

Suggests testthat (>= 3.0.0), roxygen2 (>= 7.1.2), httptest2, progressr, knitr, rmarkdown, ggplot2, R.rsp (>= 1.19.0)

RoxygenNote 7.3.2

Config/testthat/edition 3

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

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

VignetteBuilder R.rsp

NeedsCompilation no

Author Ali Sanaei [aut, cre]

Maintainer Ali Sanaei <sanaei@uchicago.edu>

Repository CRAN

Date/Publication 2025-07-17 00:10:02 UTC

Contents

`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).
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, json = 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).
json	Logical. If TRUE, returns raw JSON (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 *Call LLM API*

Description

Send text or multimodal messages to a supported Large-Language-Model (LLM) service and retrieve either a **chat/completion** response or a set of **embeddings**.

Usage

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

Arguments

config	An llm_config object created by llm_config().
messages	Character vector, named vector, or list of message objects as described above.
verbose	Logical; if TRUE, prints the full API response.
json	Logical; if TRUE, returns the raw JSON with attributes.

Details

Generative vs. embedding mode:

- **Generative calls** (the default) go to the provider's chat/completions endpoint. Any extra model-specific parameters supplied through `...` in `llm_config()` (for example `temperature`, `top_p`, `max_tokens`) are forwarded verbatim to the request body. For reasoning, some provider have shared model names and ask for a config argument that indicates reasoning should be enabled, others have dedicated model names for reasoning-enabled models, and may or may not allow for another argument that indicates the effort level.
- **Embedding calls** are triggered when `config$embedding` is `TRUE` or the model name contains the string "embedding". These calls are routed to the provider's embedding endpoint and return raw embedding data. At present, extra parameters are *not* passed through to embedding endpoints.

Messages argument:

`messages` can be

- a plain character vector (each element becomes a **user** message),
- a **named** character vector whose names are interpreted as roles, or
- a list of explicit message objects (`list(role = ..., content = ...)`).

For multimodal requests you may either use the classic list-of-parts **or** (simpler) pass a named character vector where any element whose name is `"file"` is treated as a local file path and uploaded with the request (see Example 3 below).

Value

- **Generative mode:** a character string (assistant reply). When `json = TRUE`, the string has attributes `raw_json` (JSON text) and `full_response` (parsed list). call_llm supports reasning models as well, but whether the output of these models include the text of the reasning or not depends on the provider.
- **Embedding mode:** a list with element `data`, compatible with `parse_embeddings()`.

See Also

`llm_config` to create the configuration object. `call_llm_robust` for a version with automatic retries for rate limits. `call_llm_par` helper that loops over texts and stitches embedding results into a matrix.

Examples

```
## Not run:
## 1. Generative call ------------------------------------------------------
cfg <- llm_config("openai", "gpt-4o-mini", Sys.getenv("OPENAI_API_KEY"))
call_llm(cfg, "Hello!")

## 2. Embedding call -------------------------------------------------------
embed_cfg <- llm_config(
  provider  = "gemini",
  model     = "gemini-embedding-001",
  api_key   = Sys.getenv("GEMINI_KEY"),
```

```
  embedding = TRUE
)
emb <- call_llm(embed_cfg, "This is a test sentence.")
parse_embeddings(emb)

## 3. Multimodal call (named-vector shortcut) -----------------------------
cfg <- llm_config(
  provider = "openai",
  model    = "gpt-4.1-mini",
  api_key  = Sys.getenv("OPENAI_API_KEY")
)

msg <- c(
  system = "you answer in rhymes",
  user   = "interpret this. Is there a joke here?",
  file   = "path/to/local_image.png")


call_llm(cfg, msg)

## 4. Reasoning example
cfg_reason <- llm_config("openai",
                         "o4-mini",
                         Sys.getenv("OPENAI_API_KEY"),
                         reasoning_effort = 'low')
call_llm(cfg_reason,
            c(system='Only give an integer number. Nothing else',
            user='Count "s" letters in Mississippi'))

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

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

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",
                       api_key = Sys.getenv("OPENAI_API_KEY"))

  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

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

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",
                        api_key = Sys.getenv("OPENAI_API_KEY"))
  config2 <- llm_config(provider = "openai", model = "gpt-4.1-nano",
                        api_key = Sys.getenv("OPENAI_API_KEY"))

  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 from experiments (metadata, config, messages), plus new columns: response_text, raw_response_json (the raw JSON string from the API), success, error_message, duration (in seconds).

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", Sys.getenv("OPENAI_API_KEY"))
cfg2 <- llm_config("groq", "llama-3.3-70b-versatile", Sys.getenv("GROQ_API_KEY"))

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_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,
  json = 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.
json	Logical. If TRUE, returns the raw JSON as an attribute.
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:
  # Basic usage:
  robust_resp <- call_llm_robust(
    config = my_llm_config,
    messages = list(list(role = "user", content = "Hello, LLM!")),
    tries = 5,
    wait_seconds = 10,
    memoize = FALSE
  )
  cat("Response:", robust_resp, "\n")

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

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

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",
                       api_key = Sys.getenv("OPENAI_API_KEY"))

  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)
```

---

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)
```

---

llm_chat_session *Stateful chat session constructor*

Description

Stateful chat session constructor

Coerce a chat session to a data frame

Summary statistics for a chat session

Display the first part of a chat session

Display the last part of a chat session

Print a chat session object

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** (i.e. embedding = FALSE).
system	Optional system prompt inserted once at the beginning.
...	Arguments passed to other methods. For chat_session, these are default arguments forwarded to every call_llm_robust call (e.g. verbose = TRUE, json = 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. For other methods, the return value is described by their respective titles.

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 (fields are detected by name, so new providers continue to work).

Public methods

$send(text, ..., role = "user") Append a message (default role "user"), query the model, print the assistant's reply, and invisibly return it.

$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 to create the configuration object. call_llm_robust for single, stateless API calls. llm_fn for applying a prompt to many items in a vector or data frame.

Examples

```
## Not run:
cfg  <- llm_config("openai", "gpt-4o-mini", Sys.getenv("OPENAI_API_KEY"))
chat <- chat_session(cfg, system = "Be concise.")
chat$send("Who invented the moon?")
chat$send("Explain why in one short sentence.")

# Using S3 methods
chat           # print() shows a summary and first 10 turns
summary(chat)  # Get session statistics
tail(chat, 2)  # See the last 2 turns of the conversation
df <- as.data.frame(chat) # Convert the full history to a data frame

## End(Not run)
```

---

llm_config *Create LLM Configuration*

Description

Create LLM Configuration

Usage

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

Arguments

provider	Provider name (openai, anthropic, groq, together, voyage, gemini, deepseek)
model	Model name to use
api_key	API key for authentication
troubleshooting	Prints out all api calls. USE WITH EXTREME CAUTION as it prints your API key.
base_url	Optional base URL override
embedding	Logical indicating embedding mode: NULL (default, uses prior defaults), TRUE (force embeddings), FALSE (force generative)
...	Additional provider-specific parameters

Value

Configuration object for use with call_llm()

See Also

The main ways to use a config object:

- call_llm for a basic, single API call.
- call_llm_robust for a more reliable single call with retries.
- chat_session for creating an interactive, stateful conversation.
- llm_fn for applying a prompt to a vector or data frame.
- call_llm_par for running large-scale, parallel experiments.
- get_batched_embeddings for generating text embeddings.

Examples

```
## Not run:
  cfg <- llm_config(
    provider   = "openai",
    model      = "gpt-4o-mini",
    api_key    = Sys.getenv("OPENAI_API_KEY"),
    temperature = 0.7,
    max_tokens  = 500)

  call_llm(cfg, "Hello!")  # one-shot, bare string

## End(Not run)
```

---

llm_fn *Applies an LLM prompt to every element of a vector*

Description

Applies an LLM prompt to every element of a vector

Usage

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

Arguments

x	A character vector **or** a data.frame/tibble.
prompt	A glue template string. *If* x is a data frame, use {col} placeholders; *if* x is a vector, refer to the element as {x}.
.config	An llm_config object.
.system_prompt	Optional system message (character scalar).
...	Passed unchanged to call_llm_broadcast (e.g.\ tries, progress, verbose).

Details

Runs each prompt through `call_llm_broadcast()`, which forwards the requests to `call_llm_par()`. Internally each prompt is passed as a **plain character vector** (or a named character vector when `.system_prompt` is supplied). That core engine executes them *in parallel* according to the current *future* plan. For instant multi-core use, call `setup_llm_parallel(workers = 4)` (or whatever number you prefer) once per session; revert with `reset_llm_parallel()`.

Value

A character vector the same length as `x`. Failed calls yield `NA`.

See Also

`setup_llm_parallel`, `reset_llm_parallel`, `call_llm_par`, and `llm_mutate` which is a tidy-friendly wrapper around `llm_fn()`.

Examples

```
## --- Vector input -------------------------------------------------------
## Not run:
cfg <- llm_config(
  provider = "openai",
  model    = "gpt-4.1-nano",
  api_key  =  Sys.getenv("OPENAI_API_KEY"),
  temperature = 0
)

words <- c("excellent", "awful", "average")

llm_fn(
  words,
  prompt   = "Classify sentiment of '{x}' as Positive, Negative, or Neutral.",
  .config  = cfg,
  .system_prompt = "Respond with ONE word only."
)

## --- Data-frame input inside a tidyverse pipeline ----------------------
library(dplyr)

reviews <- tibble::tibble(
  id     = 1:3,
  review = c("Great toaster!", "Burns bread.", "It's okay.")
)

reviews |>
  llm_mutate(
    sentiment,
    prompt  = "Classify the sentiment of this review: {review}",
    .config = cfg,
    .system_prompt = "Respond with Positive, Negative, or Neutral."
  )
```

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

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

Description

A convenience wrapper around llm_fn that inserts the result as a new column via mutate.

Usage

```
llm_mutate(
  .data,
  output,
  prompt,
  .config,
  .system_prompt = NULL,
  .before = NULL,
  .after = NULL,
  ...
)
```

Arguments

`.data`	A data frame / tibble.
`output`	Unquoted name of the new column you want to add.
`prompt`	A glue template string. *If* `x` is a data frame, use `{col}` placeholders; *if* `x` is a vector, refer to the element as `{x}`.
`.config`	An llm_config object.
`.system_prompt`	Optional system message (character scalar).
`.before, .after`	Standard mutate column-placement helpers.
`...`	Passed unchanged to call_llm_broadcast (e.g.\ `tries`, `progress`, `verbose`).

Details

Internally calls `llm_fn()`, so the API requests inherit the same parallel behaviour. Activate parallelism with `setup_llm_parallel()` and shut it off with `reset_llm_parallel()`.

See Also

`setup_llm_parallel`, `reset_llm_parallel`, `call_llm_par`, `llm_fn`

Examples

```
## See examples under \link{llm_fn}.
```

log_llm_error *Log LLMR Errors*

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 *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(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
  old_plan <- 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
  future::plan(old_plan)

## End(Not run)
```

Index