Package 'LLMR'

June 26, 2025

Title Interface for Large Language Model APIs in R

Version 0.4.2

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/overview> 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), 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 2025-06-26 07:40:07 UTC

Contents

`Agent` *Agent Class for LLM Interactions*

Description

An R6 class representing an agent that interacts with language models.

At agent-level we do not automate summarization. The ‘maybe_summarize_memory()‘ function can be called manually if the user wishes to compress the agent’s memory.

Public fields

`id` Unique ID for this Agent.

`context_length` Maximum number of conversation turns stored in memory.

`model_config` The `llm_config` specifying which LLM to call.

`memory` A list of speaker/text pairs that the agent has memorized.

`persona` Named list for additional agent-specific details (e.g., role, style).

`enable_summarization` Logical. If TRUE, user *may* call ‘maybe_summarize_memory()‘.

`token_threshold` Numeric. If manually triggered, we can compare total_tokens.

`total_tokens` Numeric. Estimated total tokens in memory.

`summarization_density` Character. "low", "medium", or "high".

`summarization_prompt` Character. Optional custom prompt for summarization.

summarizer_config Optional llm_config for summarizing the agent's memory.

auto_inject_conversation Logical. If TRUE, automatically prepend conversation memory if missing.

Methods

Public methods:

- Agent$new()
- Agent$add_memory()
- Agent$maybe_summarize_memory()
- Agent$generate_prompt()
- Agent$call_llm_agent()
- Agent$generate()
- Agent$think()
- Agent$respond()
- Agent$reset_memory()
- Agent$clone()

Method new(): Create a new Agent instance.

Usage:

```
Agent$new(
  id,
  context_length = 5,
  persona = NULL,
  model_config,
  enable_summarization = TRUE,
  token_threshold = 1000,
  summarization_density = "medium",
  summarization_prompt = NULL,
  summarizer_config = NULL,
  auto_inject_conversation = TRUE
)
```

Arguments:

id Character. The agent's unique identifier.

context_length Numeric. The maximum number of messages stored (default = 5).

persona A named list of persona details.

model_config An llm_config object specifying LLM settings.

enable_summarization Logical. If TRUE, you can manually call summarization.

token_threshold Numeric. If you're calling summarization, use this threshold if desired.

summarization_density Character. "low", "medium", "high" for summary detail.

summarization_prompt Character. Optional custom prompt for summarization.

summarizer_config Optional llm_config for summarization calls.

auto_inject_conversation Logical. If TRUE, auto-append conversation memory to prompt if missing.

Returns: A new `Agent` object.

Method `add_memory()`**:** Add a new message to the agent's memory. We do NOT automatically call summarization here.

Usage:

```
Agent$add_memory(speaker, text)
```

Arguments:

`speaker` Character. The speaker name or ID.

`text` Character. The message content.

Method `maybe_summarize_memory()`**:** Manually compress the agent's memory if desired. Summarizes all memory into a single "summary" message.

Usage:

```
Agent$maybe_summarize_memory()
```

Method `generate_prompt()`**:** Internal helper to prepare final prompt by substituting placeholders.

Usage:

```
Agent$generate_prompt(template, replacements = list())
```

Arguments:

`template` Character. The prompt template.

`replacements` A named list of placeholder values.

Returns: Character. The prompt with placeholders replaced.

Method `call_llm_agent()`**:** Low-level call to the LLM (via robust call_llm_robust) with a final prompt. If persona is defined, a system message is prepended to help set the role.

Usage:

```
Agent$call_llm_agent(prompt, verbose = FALSE)
```

Arguments:

`prompt` Character. The final prompt text.

`verbose` Logical. If TRUE, prints debug info. Default FALSE.

Returns: A list with: * text * tokens_sent * tokens_received * full_response (raw list)

Method `generate()`**:** Generate a response from the LLM using a prompt template and optional replacements. Substitutes placeholders, calls the LLM, saves output to memory, returns the response.

Usage:

```
Agent$generate(prompt_template, replacements = list(), verbose = FALSE)
```

Arguments:

`prompt_template` Character. The prompt template.

`replacements` A named list of placeholder values.

`verbose` Logical. If TRUE, prints extra info. Default FALSE.

Returns: A list with fields `text`, `tokens_sent`, `tokens_received`, `full_response`.

Method `think()`**:** The agent "thinks" about a topic, possibly using the entire memory in the prompt. If auto_inject_conversation is TRUE and the template lacks {{conversation}}, we prepend the memory.

Usage:

```
Agent$think(topic, prompt_template, replacements = list(), verbose = FALSE)
```

Arguments:

`topic` Character. Label for the thought.

`prompt_template` Character. The prompt template.

`replacements` Named list for additional placeholders.

`verbose` Logical. If TRUE, prints info.

Method `respond()`**:** The agent produces a public "response" about a topic. If auto_inject_conversation is TRUE and the template lacks {{conversation}}, we prepend the memory.

Usage:

```
Agent$respond(topic, prompt_template, replacements = list(), verbose = FALSE)
```

Arguments:

`topic` Character. A short label for the question/issue.

`prompt_template` Character. The prompt template.

`replacements` Named list of placeholder substitutions.

`verbose` Logical. If TRUE, prints extra info.

Returns: A list with `text`, `tokens_sent`, `tokens_received`, `full_response`.

Method `reset_memory()`**:** Reset the agent's memory.

Usage:

```
Agent$reset_memory()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Agent$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`AgentAction` *AgentAction S3 Class*

Description

An object that bundles an Agent together with a prompt and replacements so that it can be chained onto a conversation with the '+' operator.

When 'conversation + AgentAction' is called:

1. If the agent is not yet in the conversation, it is added.
2. The agent is prompted with the provided prompt template (and replacements).
3. The conversation is updated with the agent's response.

Usage

```
AgentAction(agent, prompt_template, replacements = list(), verbose = FALSE)
```

Arguments

agent	An Agent object.
prompt_template	A character string (the prompt).
replacements	A named list for placeholder substitution (optional).
verbose	Logical. If TRUE, prints verbose LLM response info. Default FALSE.

Value

An object of class AgentAction, used in conversation chaining.

---

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,
  messages_list,
  repetitions = 1,
  config_labels = NULL,
  message_labels = NULL
)
```

Arguments

configs	List of llm_config objects to test.
messages_list	List of message lists to test (each element is a message list for one condition).
repetitions	Integer. Number of repetitions per combination. Default is 1.
config_labels	Character vector of labels for configs. If NULL, uses "provider_model".
message_labels	Character vector of labels for message sets. If NULL, uses "messages_1", etc.

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 <- list(control_messages, treatment_messages)

  experiments <- build_factorial_experiments(
    configs = configs,
    messages_list = messages_list,
    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

Sends a message to the specified LLM API and retrieves the response.

Usage

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

Arguments

`config`	An ‘llm_config‘ object created by ‘llm_config()‘.
`messages`	A list of message objects (or a character vector for embeddings). For multimodal requests, the ‘content‘ of a message can be a list of parts, e.g., ‘list(list(type="text", text="..."), list(type="file", path="..."))‘.
`verbose`	Logical. If ‘TRUE‘, prints the full API response.
`json`	Logical. If ‘TRUE‘, the returned text will have the raw JSON response and the parsed list as attributes.

Value

The generated text response or embedding results. If ‘json=TRUE‘, attributes ‘raw_json‘ and ‘full_response‘ are attached.

Examples

```
## Not run:
  # Standard text call
  config <- llm_config(provider = "openai", model = "gpt-4o-mini", api_key = "...")
  messages <- list(list(role = "user", content = "Hello!"))
  response <- call_llm(config, messages)

  # Multimodal call (for supported providers like Gemini, Claude 3, GPT-4o)
  # Make sure to use a vision-capable model in your config
  multimodal_config <- llm_config(provider = "openai", model = "gpt-4o", api_key = "...")
  multimodal_messages <- list(list(role = "user", content = list(
    list(type = "text", text = "What is in this image?"),
```

```
    list(type = "file", path = "path/to/your/image.png")
  )))
  image_response <- call_llm(multimodal_config, multimodal_messages)

## End(Not run)
```

`call_llm_broadcast` *Mode 2: Message Broadcast - 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_list, ...)
```

Arguments

`config`	Single llm_config object to use for all calls.
`messages_list`	A list of message lists, each for one API call.
`...`	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.

Examples

```
## Not run:
  # Broadcast different questions
  config <- llm_config(provider = "openai", model = "gpt-4o-mini",
                       api_key = Sys.getenv("OPENAI_API_KEY"))

  messages_list <- 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_list)
  reset_llm_parallel(verbose = TRUE)

## End(Not run)
```

`call_llm_compare` *Mode 3: Model Comparison - 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`	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.

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-3.5-turbo",
                        api_key = Sys.getenv("OPENAI_API_KEY"))

  configs_list <- list(config1, config2)
  messages <- list(list(role = "user", content = "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 = 2,
  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' (message lists). Additional columns are treated as metadata and preserved.
`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).

Examples

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

  # Build experiments with expand_grid
  experiments <- expand_grid(
    condition = c("control", "treatment"),
    model_type = c("small", "large"),
    rep = 1:10
  ) |>
    mutate(
      config = case_when(
        model_type == "small" ~ list(small_config),
        model_type == "large" ~ list(large_config)
      ),
      messages = case_when(
        condition == "control" ~ list(control_messages),
        condition == "treatment" ~ list(treatment_messages)
      )
    )

  setup_llm_parallel(workers = 4)
  results <- call_llm_par(experiments, progress = TRUE)
  reset_llm_parallel()

  # All metadata preserved for analysis
  results |>
    group_by(condition, model_type) |>
    summarise(mean_response = mean(as.numeric(response_text), na.rm = TRUE))

## 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.

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 *Mode 1: 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	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.

Examples

```
## Not run:
  # Temperature sweep
  config <- llm_config(provider = "openai", model = "gpt-4o-mini",
                       api_key = Sys.getenv("OPENAI_API_KEY"))

  messages <- list(list(role = "user", content = "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)
  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.

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.

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)
```

LLMConversation *LLMConversation Class for Coordinating Agents*

Description

An R6 class for managing a conversation among multiple Agent objects. Includes optional conversation-level summarization if ‘summarizer_config‘ is provided:

1. **summarizer_config:** A list that can contain:
   - llm_config: The llm_config used for the summarizer call (default a basic OpenAI).
   - prompt: A custom summarizer prompt (default provided).
   - threshold: Word-count threshold (default 3000 words).

- summary_length: Target length in words for the summary (default 400).

2. Once the total conversation word count exceeds 'threshold', a summarization is triggered.
3. The conversation is replaced with a single condensed message that keeps track of who said what.

Public fields

`agents` A named list of `Agent` objects.

`conversation_history` A list of speaker/text pairs for the entire conversation.

`conversation_history_full` A list of speaker/text pairs for the entire conversation that is never modified and never used directly.

`topic` A short string describing the conversation's theme.

`prompts` An optional list of prompt templates (may be ignored).

`shared_memory` Global store that is also fed into each agent's memory.

`last_response` last response received

`total_tokens_sent` total tokens sent in conversation

`total_tokens_received` total tokens received in conversation

`summarizer_config` Config list controlling optional conversation-level summarization.

Methods

Public methods:

- LLMConversation$new()
- LLMConversation$add_agent()
- LLMConversation$add_message()
- LLMConversation$converse()
- LLMConversation$run()
- LLMConversation$print_history()
- LLMConversation$reset_conversation()
- LLMConversation$|>()
- LLMConversation$maybe_summarize_conversation()
- LLMConversation$summarize_conversation()
- LLMConversation$clone()

Method `new()`**:** Create a new conversation.

Usage:

```
LLMConversation$new(topic, prompts = NULL, summarizer_config = NULL)
```

Arguments:

`topic` Character. The conversation topic.

`prompts` Optional named list of prompt templates.

`summarizer_config` Optional list controlling conversation-level summarization.

Method `add_agent()`**:** Add an `Agent` to this conversation. The agent is stored by `agent$id`.

Usage:

```
LLMConversation$add_agent(agent)
```

Arguments:

`agent` An Agent object.

Method `add_message()`**:** Add a message to the global conversation log. Also appended to shared memory. Then possibly trigger summarization if configured.

Usage:

```
LLMConversation$add_message(speaker, text)
```

Arguments:

`speaker` Character. Who is speaking?

`text` Character. What they said.

Method `converse()`**:** Have a specific agent produce a response. The entire global conversation plus shared memory is temporarily loaded into that agent. Then the new message is recorded in the conversation. The agent's memory is then reset except for its new line.

Usage:

```
LLMConversation$converse(
  agent_id,
  prompt_template,
  replacements = list(),
  verbose = FALSE
)
```

Arguments:

`agent_id` Character. The ID of the agent to converse.

`prompt_template` Character. The prompt template for the agent.

`replacements` A named list of placeholders to fill in the prompt.

`verbose` Logical. If TRUE, prints extra info.

Method `run()`**:** Run a multi-step conversation among a sequence of agents.

Usage:

```
LLMConversation$run(
  agent_sequence,
  prompt_template,
  replacements = list(),
  verbose = FALSE
)
```

Arguments:

`agent_sequence` Character vector of agent IDs in the order they speak.

`prompt_template` Single string or named list of strings keyed by agent ID.

`replacements` Single list or list-of-lists with per-agent placeholders.

`verbose` Logical. If TRUE, prints extra info.

Method `print_history()`**:** Print the conversation so far to the console.

Usage:

```
LLMConversation$print_history()
```

Method `reset_conversation()`**:** Clear the global conversation and reset all agents' memories.

Usage:

```
LLMConversation$reset_conversation()
```

Method `|>()`**:** Pipe-like operator to chain conversation steps. E.g., conv |> "Solver"(...)

Usage:

```
LLMConversation$|>(agent_id)
```

Arguments:

`agent_id` Character. The ID of the agent to call next.

Returns: A function that expects (prompt_template, replacements, verbose).

Method `maybe_summarize_conversation()`**:** Possibly summarize the conversation if summarizer_config is non-null and the word count of conversation_history exceeds summarizer_config$threshold.

Usage:

```
LLMConversation$maybe_summarize_conversation()
```

Method `summarize_conversation()`**:** Summarize the conversation so far into one condensed message. The new conversation history becomes a single message with speaker = "summary".

Usage:

```
LLMConversation$summarize_conversation()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LLMConversation$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`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()

Examples

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

## End(Not run)
```

---

llm_fn *Vectorised LLM transformer*

Description

Vectorised LLM transformer

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()‘. 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

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

llm_fn, setup_llm_parallel, reset_llm_parallel

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