Package 'LLMR'

March 13, 2025

Title Interface for Large Language Model APIs in R

Version 0.2.3

Depends R (>= 4.1.0)

Description
A unified interface to interact with various Large Language Model (LLM) APIs such as '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). Allows users to configure API parameters, send messages, and retrieve responses seamlessly within R.

License MIT + file LICENSE

Encoding UTF-8

Imports httr2, purrr, rlang

Suggests testthat (>= 3.0.0), roxygen2 (>= 7.1.2), httptest2

RoxygenNote 7.3.2

Config/testthat/edition 3

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

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

NeedsCompilation no

Author Ali Sanaei [aut, cre]

Maintainer Ali Sanaei <sanaei@uchicago.edu>

Repository CRAN

Date/Publication 2025-03-13 08:10:11 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 `call_llm()`) 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.

`call_llm` *Call LLM API*

Description

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

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

Usage

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

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

Arguments

`config`	An ‘llm_config‘ object created by ‘llm_config()‘.
`messages`	A list of message objects (for chat calls) or a character vector (for embeddings).
`verbose`	Logical. If ‘TRUE‘, prints the full API response.
`json`	Logical. If ‘TRUE‘, returns the raw JSON response as an attribute.

Value

The generated text response or embedding results with additional attributes.

The generated text response or embedding results with additional attributes.

Examples

```
## Not run:
# Make sure to set your needed API keys in environment variables
  # OpenAI Embedding Example (overwriting api_url):
  openai_embed_config <- llm_config(
    provider = "openai",
    model = "text-embedding-3-small",
    api_key = Sys.getenv("OPENAI_KEY"),
    temperature = 0.3,
    api_url = "https://api.openai.com/v1/embeddings"
  )

  text_input <- c("Political science is a useful subject",
                  "We love sociology",
                  "German elections are different",
                  "A student was always curious.")
```

```
  embed_response <- call_llm(openai_embed_config, text_input)

  # Voyage AI Example:
  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)
  embeddings |> cor() |> print()


  # Gemini Example
  gemini_config <- llm_config(
    provider = "gemini",
    model = "gemini-pro",          # Or another Gemini model
    api_key = Sys.getenv("GEMINI_API_KEY"),
    temperature = 0.9,               # Controls randomness
    max_tokens = 800,              # Maximum tokens to generate
    top_p = 0.9,                     # Nucleus sampling parameter
    top_k = 10                      # Top K sampling parameter
  )

  gemini_message <- list(
   list(role = "user", content = "Explain the theory of relativity to a curious 3-year-old!")
  )

  gemini_response <- call_llm(
    config = gemini_config,
    messages = gemini_message,
    json = TRUE # Get raw JSON for inspection if needed
  )

  # Display the generated text response
  cat("Gemini Response:", gemini_response, "\n")

  # Access and print the raw JSON response
  raw_json_gemini_response <- attr(gemini_response, "raw_json")
  print(raw_json_gemini_response)

## End(Not run)
## Not run:
# Make sure to set your needed API keys in environment variables
  # OpenAI Embedding Example (overwriting api_url):
  openai_embed_config <- llm_config(
    provider = "openai",
    model = "text-embedding-3-small",
    api_key = Sys.getenv("OPENAI_KEY"),
    temperature = 0.3,
    api_url = "https://api.openai.com/v1/embeddings"
  )
```

```
text_input <- c("Political science is a useful subject",
                "We love sociology",
                "German elections are different",
                "A student was always curious.")

embed_response <- call_llm(openai_embed_config, text_input)

# Voyage AI Example:
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)
embeddings |> cor() |> print()


# Gemini Example
gemini_config <- llm_config(
  provider = "gemini",
  model = "gemini-pro",
  api_key = Sys.getenv("GEMINI_API_KEY"),
  temperature = 0.9,
  max_tokens = 800,
  top_p = 0.9,
  top_k = 10
)

gemini_message <- list(
 list(role = "user", content = "Explain the theory of relativity to a curious 3-year-old!")
)

gemini_response <- call_llm(
  config = gemini_config,
  messages = gemini_message,
  json = TRUE
)

cat("Gemini Response:", gemini_response, "\n")
raw_json_gemini_response <- attr(gemini_response, "raw_json")
print(raw_json_gemini_response)

## 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

Creates a configuration object for interacting with a specified LLM API provider.

Creates a configuration object for interacting with a specified LLM API provider.

Usage

```
llm_config(provider, model, api_key, trouble_shooting = FALSE, ...)

llm_config(provider, model, api_key, trouble_shooting = FALSE, ...)
```

Arguments

`provider`	A string specifying the API provider. Supported providers include: "openai" for OpenAI, "anthropic" for Anthropic, "groq" for Groq, "together" for Together AI, "deepseek" for DeepSeek, "voyage" for Voyage AI, "gemini" for Google Gemini.
`model`	The model name to use. This depends on the provider.
`api_key`	Your API key for the provider.
`trouble_shooting`	Prints out all api calls. USE WITH EXTREME CAUTION as it prints your API key.
`...`	Additional model-specific parameters (e.g., ‘temperature‘, ‘max_tokens‘, etc.).

Value

An object of class ‘llm_config‘ containing API and model parameters.

An object of class ‘llm_config‘ containing API and model parameters.

Examples

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

  # OpenAI Embedding Example (overwriting api_url):
  openai_embed_config <- llm_config(
    provider = "openai",
```

```
  model = "text-embedding-3-small",
  api_key = Sys.getenv("OPENAI_KEY"),
  temperature = 0.3,
  api_url = "https://api.openai.com/v1/embeddings"
)

text_input <- c("Political science is a useful subject",
                "We love sociology",
                "German elections are different",
                "A student was always curious.")

embed_response <- call_llm(openai_embed_config, text_input)
# parse_embeddings() can then be used to convert the embedding results.

# Voyage AI Example:
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)
## Not run:
  # OpenAI Example (chat)
  openai_config <- llm_config(
    provider = "openai",
    model = "gpt-4o-mini",
    api_key = Sys.getenv("OPENAI_KEY"),
    temperature = 0.7,
    max_tokens = 500
  )

  # OpenAI Embedding Example (overwriting api_url):
  openai_embed_config <- llm_config(
    provider = "openai",
    model = "text-embedding-3-small",
    api_key = Sys.getenv("OPENAI_KEY"),
    temperature = 0.3,
    api_url = "https://api.openai.com/v1/embeddings"
  )

  text_input <- c("Political science is a useful subject",
                  "We love sociology",
                  "German elections are different",
                  "A student was always curious.")

  embed_response <- call_llm(openai_embed_config, text_input)
  # parse_embeddings() can then be used to convert the embedding results.
```

```
  # Voyage AI Example:
  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)
  embeddings |> cor() |> print()

## End(Not run)
```

`parse_embeddings` *Parse Embedding Response into a Numeric Matrix*

Description

Converts the embedding response data to a numeric matrix.

Converts the embedding response data to a numeric matrix.

Usage

```
parse_embeddings(embedding_response)
```


```
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.

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)
## 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)
  embeddings |> cor() |> print()

## End(Not run)
```

Index