Package 'LLMR'

February 11, 2025

Title Interface for Large Language Model APIs in R

Version 0.2.1

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/quickstart> 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 2025-02-11 13:40:02 UTC

Contents

Agent 2
AgentAction 6
call_llm 7
LLMConversation 9
llm_config 11
parse_embeddings 12

Index **14**

`Agent` *Agent Class for LLM Interactions*

Description

An R6 class representing an agent that interacts with language models. Each Agent can maintain its own memory, knowledge, and references to a model config.

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."

`knowledge` Named list for additional agent-specific info/attributes.

Methods

Public methods:

- `Agent$new()`
- `Agent$add_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, knowledge = NULL, model_config)
```

Arguments:

`id` Character. The agent's unique identifier.

`context_length` Numeric. The maximum memory length (default 5).

`knowledge` A named list of knowledge or persona details.

`model_config` An `llm_config` object specifying LLM settings.

Method `add_memory()`**:** Add a new message to the agent's memory.

Usage:

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

Arguments:

`speaker` Character. The speaker name or ID.

`text` Character. The message content.

Method `generate_prompt()`**:** Replace placeholders in a prompt template with values from 'replacements'.

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()`**:** Call the underlying LLM with a plain text 'prompt'.

Usage:

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

Arguments:

`prompt` Character. The final prompt to send.

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

Returns: A list with $text (the LLM response) plus token usage, etc.

Method `generate()`**:** Generate an LLM response using a prompt template and optional replacements.

Usage:

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

Arguments:

`prompt_template` Character. The prompt template.

`replacements` Named list for placeholder substitution in the prompt.

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

Returns: A list with $text, $tokens_sent, $tokens_received, and $full_response.

Method `think()`**:** Have the agent produce an "internal" thought about a topic, using memory, etc.

Usage:

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

Arguments:

`topic` Character. A short description or label for the thought.

`prompt_template` Character. The prompt template.

`replacements` Named list. Additional placeholders to fill.

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

Method `respond()`**:** Have the agent produce a "public" answer or response about a topic.

Usage:

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

Arguments:

`topic` Character. A short label for the question or request.

`prompt_template` Character. The prompt template.

`replacements` Named list. Placeholders to fill in the prompt.

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

Method `reset_memory()`**:** Reset the agent's memory (clear any stored conversation context).

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.

Examples

```
## Not run:
# Example: Multi-Agent Conversation on Carbon Footprint
# ------------------------------------------------------
library(LLMR)

# Set up model configuration
agentcfg <- llm_config(
  provider   = "openai",
  model      = "gpt-4o-mini",
  api_key    = Sys.getenv("OPENAI_KEY"),
  temperature = 1.0,
  max_tokens  = 1000
)

# Create three agents: liberal, conservative, mediator
liberal_agent <- Agent$new(
  id = "liberal",
  model_config = agentcfg,
  knowledge = list(
    ideology  = "liberal",
```

```
    verbosity = "brief"
  )
)

conservative_agent <- Agent$new(
  id = "conservative",
  model_config = agentcfg,
  knowledge = list(
    ideology  = "conservative",
    verbosity = "very terse"
  )
)

mediator_agent <- Agent$new(
  id = "mediator",
  model_config = agentcfg,
  knowledge = list(role = "mediator")
)

# We can inject messages to establish context
# conv$add_message(
#   "System",
#   "We are discussing the most effective ways to reduce carbon footprint."
# )
# conv$add_message(
#   "Liberal",
#   "If corporations are not held accountable, they will continue to pollute.
#    We need stricter regulations."
# )
# conv$add_message(
#   "Conservative",
#   "Regulations can stifle economic growth. We should incentivize companies
#    to reduce emissions voluntarily. Don't kill capitalism!"
# )

# Create a conversation about reducing carbon footprint
# Chain agent responses:
conv <- LLMConversation$new(topic = "Reducing Carbon Footprint Discussion") +
  AgentAction(
    liberal_agent,
    prompt_template = "{{topic}}\n From your liberal perspective, what
    strategies would you suggest to further reduce the carbon footprint?"
  )

# See how conversation is injected into the prompt:
conv <- conv + AgentAction(
  conservative_agent,
  prompt_template = "so far we heard\n {{conversation}}\n Do you agree
  with all of this?  Given your coservative background, give your comments
  about every item proposed so far and then add your own suggestions."
)

conv <- conv + AgentAction(
```

```
  mediator_agent,
  prompt_template = "Considering these views\n{{conversation}}\n identify
  common ground and propose a unified approach to reducing the carbon
  footprint."
)

# New step for the liberal agent that now includes just the last line
conv <- conv + AgentAction(
  liberal_agent,
  prompt_template = "Reviewing what the moderator summarized: {{last_output}}\n
  I understand there is disagreement, but do you agree with these common grounds?
  If not, what would you change?"
)

conv$print_history()

# Another technique for more direct control of what goes into a prompt
mediator_snippet <- paste(
  tail(conv$conversation_history, 2)[[1]]$speaker,
  tail(conv$conversation_history, 2)[[1]]$text,
  collapse = ":"
)

conv <- conv + AgentAction(
  conservative_agent,
  prompt_template = "The mediator said:\n{{frozen_text}}\nWhat is your response?",
  replacements = list(frozen_text = mediator_snippet)
)

# Print the final conversation history
conv$print_history()

## End(Not run)
```

`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.

The prompt template can include reserved placeholders that are automatically substituted:

- {{conversation}}: Inserts the current conversation history.
- {{last_output}}: Inserts the output from the last agent.
- {{topic}}: Inserts the conversation topic.
- Any key from the agent's knowledge list (e.g., {{party}} or {{role}}) is also available.

Additional custom placeholders can be provided via the replacements argument.

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.

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) to send to the API.
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.

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)
```

`LLMConversation` *LLMConversation Class for Coordinating Agents*

Description

An R6 class for managing a conversation among multiple `Agent` objects. Holds:

- `agents`: Named list of Agents, keyed by `agent$id`.
- `conversation_history`: The full log of messages so far.
- `topic`: A short label or theme for the conversation.
- `prompts`: Optional named list of prompt templates.

Agents do *not* automatically share memory. Whenever you call `converse()`, the entire conversation history is *temporarily* loaded into the targeted agent, which responds, then is pruned again.

Public fields

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

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

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

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

Methods

Public methods:

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

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

Usage:

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

Arguments:

`topic` Character. The conversation topic.

`prompts` Optional named list of prompt templates.

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

Usage:

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

Arguments:

`agent` The `Agent` to add.

Method `add_message()`**:** Add a message to the global conversation log.

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 so far is temporarily loaded into that agent's memory, the agent responds, and then we store the agent's new message in this conversation.

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 will speak.
`prompt_template` Either a single string or a named list of templates.
`replacements` Either a single list or a 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 agent memories.

Usage:

```
LLMConversation$reset_conversation()
```

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

Usage:

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

Arguments:

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

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.

Usage

```
llm_config(provider, model, api_key, ...)
```

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.
`...`	Additional model-specific parameters (e.g., ‘temperature‘, ‘max_tokens‘, etc.).

Value

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)
```

`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)
```

Index

Agent, 2
AgentAction, 6

call_llm, 7

llm_config, 11
LLMConversation, 9

parse_embeddings, 12