Package 'mLLMCelltype'

September 2, 2025

Type Package

Title Cell Type Annotation Using Large Language Models

Version 1.3.2

Description Automated cell type annotation for single-cell RNA sequencing data using consensus predictions from multiple large language models (LLMs). LLMs are artificial intelligence models trained on vast text corpora to understand and generate human-like text. This package integrates with 'Seurat' objects and provides uncertainty quantification for annotations. Supports various LLM providers including 'OpenAI', 'Anthropic', and 'Google'. The package leverages these models through their respective APIs (Application Programming Interfaces) <https://platform.openai.com/docs>, <https://docs.anthropic.com/>, and <https://ai.google.dev/gemini-api/docs>.
For details see Yang et al. (2025) <doi:10.1101/2025.04.10.647852>.

License MIT + file LICENSE

BugReports https://github.com/cafferychen777/mLLMCelltype/issues

URL https://cafferyang.com/mLLMCelltype/

Encoding UTF-8

Imports dplyr, httr (>= 1.4.0), jsonlite (>= 1.7.0), R6 (>= 2.5.0), digest (>= 0.6.25), magrittr, utils

Suggests knitr, rmarkdown, Seurat

RoxygenNote 7.3.2

Config/build/clean-inst-doc FALSE

VignetteBuilder knitr

NeedsCompilation no

Author Chen Yang [aut, cre, cph]

Maintainer Chen Yang <cafferychen777@tamu.edu>

Repository CRAN

Date/Publication 2025-09-02 20:50:12 UTC

Contents

`annotate_cell_types` *Cell Type Annotation with Multi-LLM Framework*

Description

A comprehensive function for automated cell type annotation using multiple Large Language Models (LLMs). This function supports both Seurat's differential gene expression results and custom gene lists as input. It implements a sophisticated annotation pipeline that leverages state-of-the-art LLMs to identify cell types based on marker gene expression patterns.

Usage

```
annotate_cell_types(
  input,
  tissue_name = NULL,
  model = "gpt-4o",
  api_key = NA,
  top_gene_count = 10,
```

```
  debug = FALSE,
  base_urls = NULL
)
```

Arguments

input	One of the following:
	• A data frame from Seurat's FindAllMarkers() function containing differential gene expression results (must have columns: 'cluster', 'gene', and 'avg_log2FC'). The function will select the top genes based on avg_log2FC for each cluster.
	• A list where each element has a 'genes' field containing marker genes for a cluster. This can be in one of these formats:
	– Named with cluster IDs: list("0" = list(genes = c(...)), "1" = list(genes = c(...)))
	– Named with cell type names: list(t_cells = list(genes = c(...)), b_cells = list(genes = c(...)))
	– Unnamed list: list(list(genes = c(...)), list(genes = c(...)))
	• For both input types, if cluster IDs are numeric and start from 1, they will be automatically converted to 0-based indexing (e.g., cluster 1 becomes cluster 0) for consistency.
	IMPORTANT NOTE ON CLUSTER IDs: The 'cluster' column must contain numeric values or values that can be converted to numeric. Non-numeric cluster IDs (e.g., "cluster_1", "T_cells", "7_0") may cause errors or unexpected behavior. Before using this function, it is recommended to:
	1. Ensure all cluster IDs are numeric or can be cleanly converted to numeric values
	2. If your data contains non-numeric cluster IDs, consider creating a mapping between original IDs and numeric IDs:
	`# Example of standardizing cluster IDs`
	`original_ids <- unique(markers$cluster)`
	`id_mapping <- data.frame(`
	`  original = original_ids,`
	`  numeric = seq(0, length(original_ids) - 1)`
	`)`
	`markers$cluster <- id_mapping$numeric[match(markers$cluster, id_mapping$original)`

tissue_name	Character string specifying the tissue type or cell source (e.g., 'human PBMC', 'mouse brain'). This helps provide context for more accurate annotations.
model	Character string specifying the LLM model to use. Supported models: • OpenAI: 'gpt-4o', 'gpt-4o-mini', 'gpt-4.1', 'gpt-4.1-mini', 'gpt-4.1-nano', 'gpt-4-turbo', 'gpt-3.5-turbo', 'o1', 'o1-mini', 'o1-preview', 'o1-pro' • Anthropic: 'claude-opus-4-1-20250805', 'claude-sonnet-4-20250514', 'claude-opus-4-20250514', 'claude-3-7-sonnet-20250219', 'claude-3-5-sonnet-20241022', 'claude-3-5-haiku-20241022', 'claude-3-opus-20240229' • DeepSeek: 'deepseek-chat', 'deepseek-r1', 'deepseek-r1-zero', 'deepseek-reasoner'

- Google: 'gemini-2.5-pro', 'gemini-2.5-flash', 'gemini-2.0-flash', 'gemini-2.0-flash-lite', 'gemini-1.5-pro-latest', 'gemini-1.5-flash-latest', 'gemini-1.5-flash-8b'
- Alibaba: 'qwen-max-2025-01-25', 'qwen3-72b'
- Stepfun: 'step-2-16k', 'step-2-mini', 'step-1-8k'
- Zhipu: 'glm-4-plus', 'glm-3-turbo'
- MiniMax: 'minimax-text-01'
- X.AI: 'grok-3-latest', 'grok-3', 'grok-3-fast', 'grok-3-fast-latest', 'grok-3-mini', 'grok-3-mini-latest', 'grok-3-mini-fast', 'grok-3-mini-fast-latest'
- OpenRouter: Provides access to models from multiple providers through a single API. Format: 'provider/model-name'
  - OpenAI models: 'openai/gpt-4o', 'openai/gpt-4o-mini', 'openai/gpt-4-turbo', 'openai/gpt-4', 'openai/gpt-3.5-turbo'
  - Anthropic models: 'anthropic/claude-opus-4.1', 'anthropic/claude-sonnet-4', 'anthropic/claude-opus-4', 'anthropic/claude-3.7-sonnet', 'anthropic/claude-3.5-sonnet', 'anthropic/claude-3.5-haiku', 'anthropic/claude-3-opus'
  - Meta models: 'meta-llama/llama-3-70b-instruct', 'meta-llama/llama-3-8b-instruct', 'meta-llama/llama-2-70b-chat'
  - Google models: 'google/gemini-2.5-pro', 'google/gemini-2.5-flash', 'google/gemini-2.0-flash', 'google/gemini-1.5-pro-latest', 'google/gemini-1.5-flash'
  - Mistral models: 'mistralai/mistral-large', 'mistralai/mistral-medium', 'mistralai/mistral-small'
  - Other models: 'microsoft/mai-ds-r1', 'perplexity/sonar-small-chat', 'cohere/command-r', 'deepseek/deepseek-chat', 'thudm/glm-z1-32b'

`api_key` Character string containing the API key for the selected model. Each provider requires a specific API key format and authentication method:

- OpenAI: "sk-..." (obtain from OpenAI platform)
- Anthropic: "sk-ant-..." (obtain from Anthropic console)
- Google: A Google API key for Gemini models (obtain from Google AI)
- DeepSeek: API key from DeepSeek platform
- Qwen: API key from Alibaba Cloud
- Stepfun: API key from Stepfun AI
- Zhipu: API key from Zhipu AI
- MiniMax: API key from MiniMax
- X.AI: API key for Grok models
- OpenRouter: "sk-or-..." (obtain from OpenRouter) OpenRouter provides access to multiple models through a single API key

The API key can be provided directly or stored in environment variables:

```
# Direct API key
result <- annotate_cell_types(input, tissue_name, model="gpt-4o",
                              api_key="sk-...")

# Using environment variables
Sys.setenv(OPENAI_API_KEY="sk-...")
```

```
Sys.setenv(ANTHROPIC_API_KEY="sk-ant-...")
Sys.setenv(OPENROUTER_API_KEY="sk-or-...")

# Then use with environment variables
result <- annotate_cell_types(input, tissue_name, model="claude-3-opus",
                              api_key=Sys.getenv("ANTHROPIC_API_KEY"))
```

If NA, returns the generated prompt without making an API call, which is useful for reviewing the prompt before sending it to the API.

`top_gene_count` Integer specifying the number of top marker genes to use per cluster. when input is from Seurat's FindAllMarkers(). Default: 10

`debug` Logical. If TRUE, prints additional debugging information during execution.

`base_urls` Optional custom base URLs for API endpoints. Can be:

- A single character string: Applied to all providers (e.g., "https://api.proxy.com/v1")
- A named list: Provider-specific URLs (e.g., list(openai = "https://openai-proxy.com/v1", anthropic = "https://anthropic-proxy.com/v1")). This is useful for:
  - Chinese users accessing international APIs through proxies
  - Enterprise users with internal API gateways
  - Development/testing with local or alternative endpoints If NULL (default), uses official API endpoints for each provider.

Value

A character vector containing:

- When api_key is provided: One cell type annotation per cluster, in the order of input clusters
- When api_key is NA: The generated prompt string that would be sent to the LLM

See Also

- `Seurat::FindAllMarkers()`
- `get_provider()`
- `process_openai()`

Examples

```
# Example 1: Using custom gene lists, returning prompt only (no API call)
annotate_cell_types(
  input = list(
    t_cells = list(genes = c('CD3D', 'CD3E', 'CD3G', 'CD28')),
    b_cells = list(genes = c('CD19', 'CD79A', 'CD79B', 'MS4A1')),
    monocytes = list(genes = c('CD14', 'CD68', 'CSF1R', 'FCGR3A'))
  ),
  tissue_name = 'human PBMC',
  model = 'gpt-4o',
  api_key = NA  # Returns prompt only without making API call
)
```

```
# Example 2: Using with Seurat pipeline and OpenAI model
## Not run:
library(Seurat)

# Load example data
data("pbmc_small")

# Find marker genes
all.markers <- FindAllMarkers(
  object = pbmc_small,
  only.pos = TRUE,
  min.pct = 0.25,
  logfc.threshold = 0.25
)

# Set API key in environment variable (recommended approach)
Sys.setenv(OPENAI_API_KEY = "your-openai-api-key")

# Get cell type annotations using OpenAI model
openai_annotations <- annotate_cell_types(
  input = all.markers,
  tissue_name = 'human PBMC',
  model = 'gpt-4o',
  api_key = Sys.getenv("OPENAI_API_KEY"),
  top_gene_count = 15
)

# Example 3: Using Anthropic Claude model
Sys.setenv(ANTHROPIC_API_KEY = "your-anthropic-api-key")

claude_annotations <- annotate_cell_types(
  input = all.markers,
  tissue_name = 'human PBMC',
  model = 'claude-3-opus',
  api_key = Sys.getenv("ANTHROPIC_API_KEY"),
  top_gene_count = 15
)

# Example 4: Using OpenRouter to access multiple models
Sys.setenv(OPENROUTER_API_KEY = "your-openrouter-api-key")

# Access OpenAI models through OpenRouter
openrouter_gpt4_annotations <- annotate_cell_types(
  input = all.markers,
  tissue_name = 'human PBMC',
  model = 'openai/gpt-4o',  # Note the provider/model format
  api_key = Sys.getenv("OPENROUTER_API_KEY"),
  top_gene_count = 15
)

# Access Anthropic models through OpenRouter
openrouter_claude_annotations <- annotate_cell_types(
```

```
  input = all.markers,
  tissue_name = 'human PBMC',
  model = 'anthropic/claude-3-opus',  # Note the provider/model format
  api_key = Sys.getenv("OPENROUTER_API_KEY"),
  top_gene_count = 15
)

# Example 5: Using with mouse brain data
mouse_annotations <- annotate_cell_types(
  input = mouse_markers,  # Your mouse marker genes
  tissue_name = 'mouse brain',  # Specify correct tissue for context
  model = 'gpt-4o',
  api_key = Sys.getenv("OPENAI_API_KEY"),
  top_gene_count = 20,  # Use more genes for complex tissues
  debug = TRUE  # Enable debug output
)

## End(Not run)
```

`AnthropicProcessor` *Anthropic API Processor*

Description

Anthropic API Processor

Anthropic API Processor

Details

Concrete implementation of BaseAPIProcessor for Anthropic models. Handles Anthropic-specific API calls, authentication, and response parsing.

Super class

`mLLMCelltype::BaseAPIProcessor` -> `AnthropicProcessor`

Methods

Public methods:

- `AnthropicProcessor$new()`
- `AnthropicProcessor$get_default_api_url()`
- `AnthropicProcessor$make_api_call()`
- `AnthropicProcessor$extract_response_content()`
- `AnthropicProcessor$clone()`

Method `new()`**:** Initialize Anthropic processor

Usage:

```
AnthropicProcessor$new(base_url = NULL)
```

Arguments:

`base_url` Optional custom base URL for Anthropic API

Method `get_default_api_url()`**:** Get default Anthropic API URL

Usage:

```
AnthropicProcessor$get_default_api_url()
```

Returns: Default Anthropic API endpoint URL

Method `make_api_call()`**:** Make API call to Anthropic

Usage:

```
AnthropicProcessor$make_api_call(chunk_content, model, api_key)
```

Arguments:

`chunk_content` Content for this chunk

`model` Model identifier

`api_key` API key

Returns: httr response object

Method `extract_response_content()`**:** Extract response content from Anthropic API response

Usage:

```
AnthropicProcessor$extract_response_content(response, model)
```

Arguments:

`response` httr response object

`model` Model identifier

Returns: Extracted text content

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`BaseAPIProcessor` *Base API Processor Class*

Description

Base API Processor Class

Base API Processor Class

Details

Abstract base class for API processors that provides common functionality including unified logging, error handling, input processing, and response validation. This eliminates code duplication across all provider-specific processors.

Public fields

`provider_name` Name of the API provider

`logger` Unified logger instance

`base_url` Custom base URL for API endpoints

Methods

Public methods:

- `BaseAPIProcessor$new()`
- `BaseAPIProcessor$process_request()`
- `BaseAPIProcessor$get_api_url()`
- `BaseAPIProcessor$get_default_api_url()`
- `BaseAPIProcessor$make_api_call()`
- `BaseAPIProcessor$extract_response_content()`
- `BaseAPIProcessor$clone()`

Method `new()`**:** Initialize the base API processor

Usage:

```
BaseAPIProcessor$new(provider_name, base_url = NULL)
```

Arguments:

`provider_name` Name of the API provider (e.g., "openai", "anthropic")

`base_url` Optional custom base URL for API endpoints

Method `process_request()`**:** Main entry point for processing API requests

Usage:

```
BaseAPIProcessor$process_request(prompt, model, api_key)
```

Arguments:

`prompt` Input prompt text

`model` Model identifier

`api_key` API key for authentication

Returns: Processed response as character vector

Method `get_api_url()`**:** Get the API URL to use for requests

Usage:

```
BaseAPIProcessor$get_api_url()
```

Returns: API URL string

Method `get_default_api_url()`**:** Abstract method to be implemented by subclasses for getting default API URL

Usage:

```
BaseAPIProcessor$get_default_api_url()
```

Returns: Default API URL string

Method `make_api_call()`**:** Abstract method to be implemented by subclasses for making the actual API call

Usage:

```
BaseAPIProcessor$make_api_call(chunk_content, model, api_key)
```

Arguments:

`chunk_content` Content for this chunk

`model` Model identifier

`api_key` API key

Returns: Raw API response

Method `extract_response_content()`**:** Abstract method to be implemented by subclasses for extracting content from response

Usage:

```
BaseAPIProcessor$extract_response_content(response, model)
```

Arguments:

`response` Raw API response

`model` Model identifier

Returns: Extracted text content Validate input parameters

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

CacheManager *Cache Manager Class*

Description

Manages caching of consensus analysis results

Public fields

`cache_dir` Directory to store cache files

`cache_version` Current cache version

Methods

Public methods:

- `CacheManager$new()`
- `CacheManager$generate_key()`
- `CacheManager$save_to_cache()`
- `CacheManager$load_from_cache()`
- `CacheManager$has_cache()`
- `CacheManager$get_cache_stats()`
- `CacheManager$clear_cache()`
- `CacheManager$validate_cache()`
- `CacheManager$clone()`

Method `new()`**:** Initialize cache manager

Usage:

```
CacheManager$new(cache_dir = NULL)
```

Arguments:

`cache_dir` Directory to store cache files (defaults to tempdir())

Method `generate_key()`**:** Generate cache key from input parameters (improved version)

Usage:

```
CacheManager$generate_key(input, models, cluster_id)
```

Arguments:

`input` Input data

`models` Models used

`cluster_id` Cluster ID

Returns: Cache key string

Method `save_to_cache()`**:** Save results to cache

Usage:

```
CacheManager$save_to_cache(key, data)
```

Arguments:

`key` Cache key

`data` Data to cache

Method `load_from_cache()`**:** Load results from cache

Usage:

```
CacheManager$load_from_cache(key)
```

Arguments:

`key` Cache key

Returns: Cached data if exists, NULL otherwise

Method `has_cache()`**:** Check if results exist in cache

Usage:

```
CacheManager$has_cache(key)
```

Arguments:

`key` Cache key

Returns: TRUE if cached results exist

Method `get_cache_stats()`**:** Get cache statistics

Usage:

```
CacheManager$get_cache_stats()
```

Returns: A list with cache statistics

Method `clear_cache()`**:** Clear all cache

Usage:

```
CacheManager$clear_cache(confirm = FALSE)
```

Arguments:

`confirm` Boolean, if TRUE, will clear cache without confirmation

Method `validate_cache()`**:** Validate cache content

Usage:

```
CacheManager$validate_cache(key)
```

Arguments:

`key` Cache key

Returns: TRUE if cache is valid, FALSE otherwise Extract genes from input in a standardized way

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

compare_model_predictions

Compare predictions from different models

Description

This function runs the same input through multiple models and compares their predictions. It provides both individual predictions and a consensus analysis.

Usage

```
compare_model_predictions(
  input,
  tissue_name,
 models = c("claude-sonnet-4-20250514", "claude-3-5-sonnet-20241022", "gpt-4.1-mini",
    "deepseek-r1", "gemini-2.5-flash", "qwen-max-2025-01-25", "gpt-4o", "o1",
    "grok-3-latest"),
  api_keys,
  top_gene_count = 10,
  consensus_threshold = 0.5
)
```

Arguments

input	Either the differential gene table returned by Seurat FindAllMarkers() function, or a list of genes.
tissue_name	Required. The tissue type or cell source (e.g., 'human PBMC', 'mouse brain', etc.).
models	Vector of model names to compare. Default includes one model from each provider. Supported models: • OpenAI: 'gpt-4o', 'gpt-4o-mini', 'gpt-4.1', 'gpt-4.1-mini', 'gpt-4.1-nano', 'gpt-4-turbo', 'gpt-3.5-turbo', 'o1', 'o1-mini', 'o1-preview', 'o1-pro' • Anthropic: 'claude-opus-4-1-20250805', 'claude-sonnet-4-20250514', 'claude-opus-4-20250514', 'claude-3-7-sonnet-20250219', 'claude-3-5-sonnet-20241022', 'claude-3-5-haiku-20241022', 'claude-3-opus-20240229' • DeepSeek: 'deepseek-chat', 'deepseek-r1', 'deepseek-r1-zero', 'deepseek-reasoner' • Google: 'gemini-2.5-pro', 'gemini-2.5-flash', 'gemini-2.0-flash', 'gemini-2.0-flash-lite', 'gemini-1.5-pro-latest', 'gemini-1.5-flash-latest', 'gemini-1.5-flash-8b' • Alibaba: 'qwen-max-2025-01-25', 'qwen3-72b' • Stepfun: 'step-2-16k', 'step-2-mini', 'step-1-8k' • Zhipu: 'glm-4-plus', 'glm-3-turbo' • MiniMax: 'minimax-text-01'

- X.AI: 'grok-3-latest', 'grok-3', 'grok-3-fast', 'grok-3-fast-latest', 'grok-3-mini', 'grok-3-mini-latest', 'grok-3-mini-fast', 'grok-3-mini-fast-latest'
- OpenRouter: Provides access to models from multiple providers through a single API. Format: 'provider/model-name'
  - OpenAI models: 'openai/gpt-4o', 'openai/gpt-4o-mini', 'openai/gpt-4-turbo', 'openai/gpt-4', 'openai/gpt-3.5-turbo'
  - Anthropic models: 'anthropic/claude-opus-4.1', 'anthropic/claude-sonnet-4', 'anthropic/claude-opus-4', 'anthropic/claude-3.7-sonnet', 'anthropic/claude-3.5-sonnet', 'anthropic/claude-3.5-haiku', 'anthropic/claude-3-opus'
  - Meta models: 'meta-llama/llama-3-70b-instruct', 'meta-llama/llama-3-8b-instruct', 'meta-llama/llama-2-70b-chat'
  - Google models: 'google/gemini-2.5-pro', 'google/gemini-2.5-flash', 'google/gemini-2.0-flash', 'google/gemini-1.5-pro-latest', 'google/gemini-1.5-flash'
  - Mistral models: 'mistralai/mistral-large', 'mistralai/mistral-medium', 'mistralai/mistral-small'
  - Other models: 'microsoft/mai-ds-r1', 'perplexity/sonar-small-chat', 'cohere/command-r', 'deepseek/deepseek-chat', 'thudm/glm-z1-32b'

`api_keys` Named list of API keys. Can be provided in two formats:

1. With provider names as keys: `list("openai" = "sk-...", "anthropic" = "sk-ant-...", "openrouter" = "sk-or-...")`
2. With model names as keys: `list("gpt-4o" = "sk-...", "claude-3-opus" = "sk-ant-...")`

The system first tries to find the API key using the provider name. If not found, it then tries using the model name. Example:

```
api_keys <- list(
  "openai" = Sys.getenv("OPENAI_API_KEY"),
  "anthropic" = Sys.getenv("ANTHROPIC_API_KEY"),
  "openrouter" = Sys.getenv("OPENROUTER_API_KEY"),
  "claude-3-opus" = "sk-ant-api03-specific-key-for-opus"
)
```

`top_gene_count` Number of top differential genes to be used if input is Seurat differential genes.

`consensus_threshold`
Minimum proportion of models that must agree for a consensus (default 0.5).

Value

A list containing individual predictions, consensus results, and agreement statistics.

Note

This function uses create_standardization_prompt from prompt_templates.R

Examples

```
## Not run:
# Compare predictions using different models
api_keys <- list(
  "claude-sonnet-4-20250514" = "your-anthropic-key",
  "deepseek-reasoner" = "your-deepseek-key",
  "gemini-1.5-pro" = "your-gemini-key",
  "qwen-max-2025-01-25" = "your-qwen-key"
)

results <- compare_model_predictions(
  input = list(gs1=c('CD4','CD3D'), gs2='CD14'),
  tissue_name = 'PBMC',
  api_keys = api_keys
)

## End(Not run)
```

---

configure_logger *Set global logger configuration*

Description

Set global logger configuration

Usage

```
configure_logger(level = "INFO", console_output = TRUE, json_format = TRUE)
```

Arguments

level	Logging level
console_output	Whether to output to console
json_format	Whether to use JSON format

---

create_annotation_prompt *Prompt templates for mLLMCelltype*

Description

This file contains all prompt template functions used in mLLMCelltype. These functions create various prompts for different stages of the cell type annotation process. Create prompt for cell type annotation

Usage

```
create_annotation_prompt(input, tissue_name, top_gene_count = 10)
```

Arguments

input	Either the differential gene table returned by Seurat FindAllMarkers() function, or a list of genes
tissue_name	The name of the tissue
top_gene_count	Number of top differential genes to use per cluster

Value

A list containing the prompt string and expected count of responses

---

DeepSeekProcessor *DeepSeek API Processor*

Description

DeepSeek API Processor

DeepSeek API Processor

Details

Concrete implementation of BaseAPIProcessor for DeepSeek models. Handles DeepSeek-specific API calls, authentication, and response parsing.

Super class

`mLLMCelltype::BaseAPIProcessor` -> `DeepSeekProcessor`

Methods

Public methods:

- `DeepSeekProcessor$new()`
- `DeepSeekProcessor$get_default_api_url()`
- `DeepSeekProcessor$make_api_call()`
- `DeepSeekProcessor$extract_response_content()`
- `DeepSeekProcessor$clone()`

Method `new()`**:** Initialize DeepSeek processor

Usage:

```
DeepSeekProcessor$new(base_url = NULL)
```

Arguments:

`base_url` Optional custom base URL for DeepSeek API

Method `get_default_api_url()`: Get default DeepSeek API URL

Usage:

```
DeepSeekProcessor$get_default_api_url()
```

Returns: Default DeepSeek API endpoint URL

Method `make_api_call()`: Make API call to DeepSeek

Usage:

```
DeepSeekProcessor$make_api_call(chunk_content, model, api_key)
```

Arguments:

`chunk_content` Content for this chunk

`model` Model identifier

`api_key` API key

Returns: httr response object

Method `extract_response_content()`: Extract response content from DeepSeek API response

Usage:

```
DeepSeekProcessor$extract_response_content(response, model)
```

Arguments:

`response` httr response object

`model` Model identifier

Returns: Extracted text content

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`GeminiProcessor` *Gemini API Processor*

Description

Gemini API Processor

Gemini API Processor

Details

Concrete implementation of BaseAPIProcessor for Gemini models. Handles Gemini-specific API calls, authentication, and response parsing.

Super class

`mLLMCelltype::BaseAPIProcessor` -> `GeminiProcessor`

Methods

Public methods:

- `GeminiProcessor$new()`
- `GeminiProcessor$get_default_api_url()`
- `GeminiProcessor$get_api_url_for_model()`
- `GeminiProcessor$make_api_call()`
- `GeminiProcessor$extract_response_content()`
- `GeminiProcessor$clone()`

Method `new()`**:** Initialize Gemini processor

Usage:

```
GeminiProcessor$new(base_url = NULL)
```

Arguments:

`base_url` Optional custom base URL for Gemini API

Method `get_default_api_url()`**:** Get default Gemini API URL template

Usage:

```
GeminiProcessor$get_default_api_url()
```

Returns: Default Gemini API endpoint URL template

Method `get_api_url_for_model()`**:** Get API URL for specific model

Usage:

```
GeminiProcessor$get_api_url_for_model(model)
```

Arguments:

`model` Model identifier

Returns: Complete API URL for the model

Method `make_api_call()`**:** Make API call to Gemini

Usage:

```
GeminiProcessor$make_api_call(chunk_content, model, api_key)
```

Arguments:

`chunk_content` Content for this chunk

`model` Model identifier

`api_key` API key

Returns: httr response object

Method `extract_response_content()`**:** Extract response content from Gemini API response

Usage:

```
GeminiProcessor$extract_response_content(response, model)
```

Arguments:

`response` httr response object

`model` Model identifier

Returns: Extracted text content

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`get_api_key` *Utility functions for API key management*

Description

This file contains utility functions for managing API keys and related operations. Get API key for a specific model

Usage

```
get_api_key(model, api_keys)
```

Arguments

`model`	The name of the model to get the API key for
`api_keys`	Named list of API keys

Details

This function retrieves the appropriate API key for a given model by first checking the provider name and then the model name in the provided API keys list.

Value

The API key if found, NULL otherwise

get_logger *Get the global logger instance*

Description

Get the global logger instance

Usage

```
get_logger()
```

Value

UnifiedLogger instance

GrokProcessor *Grok API Processor*

Description

Grok API Processor

Grok API Processor

Details

Concrete implementation of BaseAPIProcessor for Grok models. Handles Grok-specific API calls, authentication, and response parsing.

Super class

`mLLMCelltype::BaseAPIProcessor` -> `GrokProcessor`

Methods

Public methods:

- `GrokProcessor$new()`
- `GrokProcessor$get_default_api_url()`
- `GrokProcessor$make_api_call()`
- `GrokProcessor$extract_response_content()`
- `GrokProcessor$clone()`

Method `new()`**:** Initialize Grok processor

Usage:

```
GrokProcessor$new(base_url = NULL)
```

Arguments:

`base_url` Optional custom base URL for Grok API

Method `get_default_api_url()`**:** Get default Grok API URL

Usage:

```
GrokProcessor$get_default_api_url()
```

Returns: Default Grok API endpoint URL

Method `make_api_call()`**:** Make API call to Grok

Usage:

```
GrokProcessor$make_api_call(chunk_content, model, api_key)
```

Arguments:

`chunk_content` Content for this chunk

`model` Model identifier

`api_key` API key

Returns: httr response object

Method `extract_response_content()`**:** Extract response content from Grok API response

Usage:

```
GrokProcessor$extract_response_content(response, model)
```

Arguments:

`response` httr response object

`model` Model identifier

Returns: Extracted text content

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`interactive_consensus_annotation`

Interactive consensus building for cell type annotation

Description

This function implements an interactive voting and discussion mechanism where multiple LLMs collaborate to reach a consensus on cell type annotations, particularly focusing on clusters with low agreement. The process includes:

1. Initial voting by all LLMs
2. Identification of controversial clusters
3. Detailed discussion for controversial clusters
4. Final summary by a designated LLM (default: Claude)

This function implements an interactive voting and discussion mechanism where multiple LLMs collaborate to reach a consensus on cell type annotations, particularly focusing on clusters with low agreement. The process includes:

1. Initial voting by all LLMs
2. Identification of controversial clusters
3. Detailed discussion for controversial clusters
4. Final summary by a designated LLM (default: Claude)

Usage

```
interactive_consensus_annotation(
  input,
  tissue_name = NULL,
  models = c("claude-sonnet-4-20250514", "claude-3-7-sonnet-20250219",
   "claude-3-5-sonnet-20241022", "claude-3-5-haiku-20241022", "gemini-2.0-flash",
    "gemini-1.5-pro", "qwen-max-2025-01-25", "gpt-4o", "grok-3-latest"),
  api_keys,
  top_gene_count = 10,
  controversy_threshold = 0.7,
  entropy_threshold = 1,
  max_discussion_rounds = 3,
  consensus_check_model = NULL,
  log_dir = NULL,
  cache_dir = NULL,
  use_cache = TRUE,
  base_urls = NULL,
  clusters_to_analyze = NULL,
  force_rerun = FALSE
)
```

Arguments

`input` One of the following:

- A data frame from Seurat's FindAllMarkers() function containing differential gene expression results (must have columns: 'cluster', 'gene', and 'avg_log2FC'). The function will select the top genes based on avg_log2FC for each cluster.

- A list where each element has a 'genes' field containing marker genes for a cluster. This can be in one of these formats:
  - Named with cluster IDs: list("0" = list(genes = c(...)), "1" = list(genes = c(...)))
  - Named with cell type names: list(t_cells = list(genes = c(...)), b_cells = list(genes = c(...)))
  - Unnamed list: list(list(genes = c(...)), list(genes = c(...)))
- For both input types, if cluster IDs are numeric and start from 1, they will be automatically converted to 0-based indexing (e.g., cluster 1 becomes cluster 0) for consistency.

`tissue_name` Optional input of tissue name

`models` Vector of model names to participate in the discussion. Supported models:

- OpenAI: 'gpt-4o', 'gpt-4o-mini', 'gpt-4.1', 'gpt-4.1-mini', 'gpt-4.1-nano', 'gpt-4-turbo', 'gpt-3.5-turbo', 'o1', 'o1-mini', 'o1-preview', 'o1-pro'
- Anthropic: 'claude-opus-4-1-20250805', 'claude-sonnet-4-20250514', 'claude-opus-4-20250514', 'claude-3-7-sonnet-20250219', 'claude-3-5-sonnet-20241022', 'claude-3-5-haiku-20241022', 'claude-3-opus-20240229'
- DeepSeek: 'deepseek-chat', 'deepseek-r1', 'deepseek-r1-zero', 'deepseek-reasoner'
- Google: 'gemini-2.5-pro', 'gemini-2.5-flash', 'gemini-2.0-flash', 'gemini-2.0-flash-lite', 'gemini-1.5-pro-latest', 'gemini-1.5-flash-latest', 'gemini-1.5-flash-8b'
- Alibaba: 'qwen-max-2025-01-25', 'qwen3-72b'
- Stepfun: 'step-2-16k', 'step-2-mini', 'step-1-8k'
- Zhipu: 'glm-4-plus', 'glm-3-turbo'
- MiniMax: 'minimax-text-01'
- X.AI: 'grok-3-latest', 'grok-3', 'grok-3-fast', 'grok-3-fast-latest', 'grok-3-mini', 'grok-3-mini-latest', 'grok-3-mini-fast', 'grok-3-mini-fast-latest'
- OpenRouter: Provides access to models from multiple providers through a single API. Format: 'provider/model-name'
  - OpenAI models: 'openai/gpt-4o', 'openai/gpt-4o-mini', 'openai/gpt-4-turbo', 'openai/gpt-4', 'openai/gpt-3.5-turbo'
  - Anthropic models: 'anthropic/claude-opus-4.1', 'anthropic/claude-sonnet-4', 'anthropic/claude-opus-4', 'anthropic/claude-3.7-sonnet', 'anthropic/claude-3.5-sonnet', 'anthropic/claude-3.5-haiku', 'anthropic/claude-3-opus'
  - Meta models: 'meta-llama/llama-3-70b-instruct', 'meta-llama/llama-3-8b-instruct', 'meta-llama/llama-2-70b-chat'
  - Google models: 'google/gemini-2.5-pro', 'google/gemini-2.5-flash', 'google/gemini-2.0-flash', 'google/gemini-1.5-pro-latest', 'google/gemini-1.5-flash'
  - Mistral models: 'mistralai/mistral-large', 'mistralai/mistral-medium', 'mistralai/mistral-small'
  - Other models: 'microsoft/mai-ds-r1', 'perplexity/sonar-small-chat', 'cohere/command-r', 'deepseek/deepseek-chat', 'thudm/glm-z1-32b'

`api_keys` Named list of API keys. Can be provided in two formats:

1. With provider names as keys: `list("openai" = "sk-...", "anthropic" = "sk-ant-...", "openrouter" = "sk-or-...")`
2. With model names as keys: `list("gpt-4o" = "sk-...", "claude-3-opus" = "sk-ant-...")`

The system first tries to find the API key using the provider name. If not found, it then tries using the model name. Example:

```
api_keys <- list(
  "openai" = Sys.getenv("OPENAI_API_KEY"),
  "anthropic" = Sys.getenv("ANTHROPIC_API_KEY"),
  "openrouter" = Sys.getenv("OPENROUTER_API_KEY"),
  "claude-3-opus" = "sk-ant-api03-specific-key-for-opus"
)
```

`top_gene_count` Number of top differential genes to use

`controversy_threshold`
Consensus proportion threshold (default: 0.7). Clusters with consensus proportion below this value will be marked as controversial

`entropy_threshold`
Entropy threshold for identifying controversial clusters (default: 1.0)

`max_discussion_rounds`
Maximum number of discussion rounds for controversial clusters (default: 3)

`consensus_check_model`
Model to use for consensus checking

`log_dir` Directory for storing logs (defaults to tempdir())

`cache_dir` Directory for storing cache (defaults to tempdir())

`use_cache` Whether to use cached results

`base_urls` Optional custom base URLs for API endpoints. Can be:

- A single character string: Applied to all providers (e.g., "https://api.proxy.com/v1")
- A named list: Provider-specific URLs (e.g., list(openai = "https://openai-proxy.com/v1", anthropic = "https://anthropic-proxy.com/v1")). This is useful for:
  - Chinese users accessing international APIs through proxies
  - Enterprise users with internal API gateways
  - Development/testing with local or alternative endpoints If NULL (default), uses official API endpoints for each provider.

`clusters_to_analyze`
Optional vector of cluster IDs to analyze. If NULL (default), all clusters in the input will be analyzed. Must be character or numeric values that match the cluster IDs in your input. Examples:

- For numeric clusters: c(0, 2, 5) or c("0", "2", "5")
- This is useful when you want to focus on specific clusters without filtering the input data
- Non-existent cluster IDs will be ignored with a warning

force_rerun Logical. If TRUE, ignore cached results and force re-analysis of all specified clusters. Useful when you want to re-analyze clusters with different context or for subtype identification. Default is FALSE. Note: This parameter only affects the discussion phase for controversial clusters.

Value

A list containing consensus results, logs, and annotations

A list containing consensus results, logs, and annotations

---

list_custom_models *Get list of registered custom models*

Description

Get list of registered custom models

Usage

```
list_custom_models()
```

Value

Character vector of model names

---

list_custom_providers *Get list of registered custom providers*

Description

Get list of registered custom providers

Usage

```
list_custom_providers()
```

Value

Character vector of provider names

logging_functions *Convenience functions for logging*

Description

Convenience functions for logging

Usage

```
log_debug(message, context = NULL)

log_info(message, context = NULL)

log_warn(message, context = NULL)

log_error(message, context = NULL)
```

Arguments

message	Log message
context	Additional context (optional)

MinimaxProcessor *Minimax API Processor*

Description

Minimax API Processor

Minimax API Processor

Details

Concrete implementation of BaseAPIProcessor for Minimax models. Handles Minimax-specific API calls, authentication, and response parsing.

Super class

`mLLMCelltype::BaseAPIProcessor` -> `MinimaxProcessor`

Methods

Public methods:

- MinimaxProcessor$new()
- MinimaxProcessor$get_default_api_url()
- MinimaxProcessor$make_api_call()
- MinimaxProcessor$extract_response_content()
- MinimaxProcessor$clone()

Method `new()`**:** Initialize Minimax processor

Usage:

```
MinimaxProcessor$new(base_url = NULL)
```

Arguments:

`base_url` Optional custom base URL for Minimax API

Method `get_default_api_url()`**:** Get default Minimax API URL

Usage:

```
MinimaxProcessor$get_default_api_url()
```

Returns: Default Minimax API endpoint URL

Method `make_api_call()`**:** Make API call to Minimax

Usage:

```
MinimaxProcessor$make_api_call(chunk_content, model, api_key)
```

Arguments:

`chunk_content` Content for this chunk
`model` Model identifier
`api_key` API key

Returns: httr response object

Method `extract_response_content()`**:** Extract response content from Minimax API response

Usage:

```
MinimaxProcessor$extract_response_content(response, model)
```

Arguments:

`response` httr response object
`model` Model identifier

Returns: Extracted text content

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

OpenAIProcessor *OpenAI API Processor*

Description

OpenAI API Processor

OpenAI API Processor

Details

Concrete implementation of BaseAPIProcessor for OpenAI models. Handles OpenAI-specific API calls, authentication, and response parsing.

Super class

`mLLMCelltype::BaseAPIProcessor` -> `OpenAIProcessor`

Methods

Public methods:

- `OpenAIProcessor$new()`
- `OpenAIProcessor$get_default_api_url()`
- `OpenAIProcessor$make_api_call()`
- `OpenAIProcessor$extract_response_content()`
- `OpenAIProcessor$clone()`

Method `new()`**:** Initialize OpenAI processor

Usage:

```
OpenAIProcessor$new(base_url = NULL)
```

Arguments:

`base_url` Optional custom base URL for OpenAI API

Method `get_default_api_url()`**:** Get default OpenAI API URL

Usage:

```
OpenAIProcessor$get_default_api_url()
```

Returns: Default OpenAI API endpoint URL

Method `make_api_call()`**:** Make API call to OpenAI

Usage:

```
OpenAIProcessor$make_api_call(chunk_content, model, api_key)
```

Arguments:

`chunk_content` Content for this chunk

`model` Model identifier

`api_key` API key

Returns: httr response object

Method `extract_response_content()`**:** Extract response content from OpenAI API response

Usage:

```
OpenAIProcessor$extract_response_content(response, model)
```

Arguments:

`response` httr response object

`model` Model identifier

Returns: Extracted text content

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`OpenRouterProcessor` *OpenRouter API Processor*

Description

OpenRouter API Processor

OpenRouter API Processor

Details

Concrete implementation of BaseAPIProcessor for OpenRouter models. Handles OpenRouter-specific API calls, authentication, and response parsing.

Super class

`mLLMCelltype::BaseAPIProcessor` -> `OpenRouterProcessor`

Methods

Public methods:

- `OpenRouterProcessor$new()`
- `OpenRouterProcessor$get_default_api_url()`
- `OpenRouterProcessor$make_api_call()`
- `OpenRouterProcessor$extract_response_content()`
- `OpenRouterProcessor$clone()`

Method `new()`**:** Initialize OpenRouter processor

Usage:

```
OpenRouterProcessor$new(base_url = NULL)
```

Arguments:

`base_url` Optional custom base URL for OpenRouter API

Method `get_default_api_url()`**:** Get default OpenRouter API URL

Usage:

```
OpenRouterProcessor$get_default_api_url()
```

Returns: Default OpenRouter API endpoint URL

Method `make_api_call()`**:** Make API call to OpenRouter

Usage:

```
OpenRouterProcessor$make_api_call(chunk_content, model, api_key)
```

Arguments:

`chunk_content` Content for this chunk

`model` Model identifier

`api_key` API key

Returns: httr response object

Method `extract_response_content()`**:** Extract response content from OpenRouter API response

Usage:

```
OpenRouterProcessor$extract_response_content(response, model)
```

Arguments:

`response` httr response object

`model` Model identifier

Returns: Extracted text content

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

QwenProcessor *Qwen API Processor*

Description

Qwen API Processor

Qwen API Processor

Details

Concrete implementation of BaseAPIProcessor for Qwen models. Handles Qwen-specific API calls, authentication, and response parsing.

Qwen has two API endpoints:

- International: https://dashscope-intl.aliyuncs.com/api/v1/services/aigc/text-generation/generation (preferred)
- Domestic (China): https://dashscope.aliyuncs.com/api/v1/services/aigc/text-generation/generation (fallback) The processor automatically tries international first, then falls back to domestic if needed.

Super class

`mLLMCelltype::BaseAPIProcessor` -> `QwenProcessor`

Methods

Public methods:

- `QwenProcessor$new()`
- `QwenProcessor$get_default_api_url()`
- `QwenProcessor$get_working_api_url()`
- `QwenProcessor$make_api_call()`
- `QwenProcessor$extract_response_content()`
- `QwenProcessor$clone()`

Method `new()`**:** Test if an endpoint is accessible

Initialize Qwen processor

Usage:

```
QwenProcessor$new(base_url = NULL)
```

Arguments:

`base_url` Optional custom base URL for Qwen API

`url` The endpoint URL to test

`api_key` API key for authentication

Returns: TRUE if accessible, FALSE otherwise

Method `get_default_api_url()`**:** Get default Qwen API URL with intelligent endpoint selection

Usage:

```
QwenProcessor$get_default_api_url()
```

Returns: Default Qwen API endpoint URL

Method `get_working_api_url()`**:** Get working Qwen API URL with automatic endpoint detection

Usage:

```
QwenProcessor$get_working_api_url(api_key)
```

Arguments:

`api_key` API key for testing endpoints

Returns: Working Qwen API endpoint URL

Method `make_api_call()`**:** Make API call to Qwen

Usage:

```
QwenProcessor$make_api_call(chunk_content, model, api_key)
```

Arguments:

`chunk_content` Content for this chunk

`model` Model identifier

`api_key` API key

Returns: httr response object

Method `extract_response_content()`**:** Extract response content from Qwen API response

Usage:

```
QwenProcessor$extract_response_content(response, model)
```

Arguments:

`response` httr response object

`model` Model identifier

Returns: Extracted text content

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

register_custom_model *Register a custom model for a provider*

Description

Register a custom model for a provider

Usage

```
register_custom_model(model_name, provider_name, model_config = list())
```

Arguments

model_name	Character string, unique identifier for the model
provider_name	Character string, name of the registered provider
model_config	List of model-specific configuration parameters

Value

Invisibly returns TRUE if registration is successful

Examples

```
## Not run:
register_custom_model(
  model_name = "my_model",
  provider_name = "my_provider",
  model_config = list(
    temperature = 0.7,
    max_tokens = 2000
  )
)

## End(Not run)
```

register_custom_provider *Register a custom LLM provider*

Description

Register a custom LLM provider

Usage

```
register_custom_provider(provider_name, process_fn, description = NULL)
```

Arguments

provider_name	Character string, unique identifier for the provider
process_fn	Function that processes prompts and returns responses. Must accept parameters: prompt, model, api_key
description	Optional description of the provider

Value

Invisibly returns TRUE if registration is successful

Examples

```
## Not run:
register_custom_provider(
  provider_name = "my_provider",
  process_fn = function(prompt, model, api_key) {
    # Custom implementation
    response <- httr::POST(
      url = "your_api_endpoint",
      body = list(prompt = prompt),
      encode = "json"
    )
    return(httr::content(response)$choices[[1]]$text)
  }
)

## End(Not run)
```

StepFunProcessor *StepFun API Processor*

Description

StepFun API Processor

StepFun API Processor

Details

Concrete implementation of BaseAPIProcessor for StepFun models. Handles StepFun-specific API calls, authentication, and response parsing.

Super class

`mLLMCelltype::BaseAPIProcessor` -> `StepFunProcessor`

Methods

Public methods:

- StepFunProcessor$new()
- StepFunProcessor$get_default_api_url()
- StepFunProcessor$make_api_call()
- StepFunProcessor$extract_response_content()
- StepFunProcessor$clone()

Method `new()`**:** Initialize StepFun processor

Usage:

```
StepFunProcessor$new(base_url = NULL)
```

Arguments:

`base_url` Optional custom base URL for StepFun API

Method `get_default_api_url()`**:** Get default StepFun API URL

Usage:

```
StepFunProcessor$get_default_api_url()
```

Returns: Default StepFun API endpoint URL

Method `make_api_call()`**:** Make API call to StepFun

Usage:

```
StepFunProcessor$make_api_call(chunk_content, model, api_key)
```

Arguments:

`chunk_content` Content for this chunk
`model` Model identifier
`api_key` API key

Returns: httr response object

Method `extract_response_content()`**:** Extract response content from StepFun API response

Usage:

```
StepFunProcessor$extract_response_content(response, model)
```

Arguments:

`response` httr response object
`model` Model identifier

Returns: Extracted text content

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

UnifiedLogger *Unified Logger for mLLMCelltype Package*

Description

Unified Logger for mLLMCelltype Package

Unified Logger for mLLMCelltype Package

Details

This logger provides centralized, multi-level logging with structured output, log rotation, and performance monitoring capabilities.

Public fields

`log_dir` Directory for storing log files

`log_level` Current logging level

`session_id` Unique identifier for the current session

`max_log_size` Maximum log file size in MB (default: 10MB)

`max_log_files` Maximum number of log files to keep (default: 5)

`enable_console` Whether to output to console (default: TRUE)

`enable_json` Whether to use JSON format (default: TRUE)

`performance_stats` Performance monitoring statistics

Methods

Public methods:

- `UnifiedLogger$new()`
- `UnifiedLogger$debug()`
- `UnifiedLogger$info()`
- `UnifiedLogger$warn()`
- `UnifiedLogger$error()`
- `UnifiedLogger$log_api_call()`
- `UnifiedLogger$log_api_request_response()`
- `UnifiedLogger$log_cache_operation()`
- `UnifiedLogger$log_cluster_progress()`
- `UnifiedLogger$log_discussion()`
- `UnifiedLogger$get_performance_summary()`
- `UnifiedLogger$cleanup_logs()`
- `UnifiedLogger$set_level()`
- `UnifiedLogger$clone()`

Method `new()`**:** Initialize the unified logger

Usage:

```
UnifiedLogger$new(
  base_dir = NULL,
  level = "INFO",
  max_size = 10,
  max_files = 5,
  console_output = TRUE,
  json_format = TRUE
)
```

Arguments:

`base_dir` Base directory for logs (defaults to tempdir())

`level` Logging level: DEBUG, INFO, WARN, ERROR (default: "INFO")

`max_size` Maximum log file size in MB (default: 10)

`max_files` Maximum number of log files to keep (default: 5)

`console_output` Whether to output to console (default: TRUE)

`json_format` Whether to use JSON format (default: TRUE)

Method `debug()`**:** Log a debug message

Usage:

```
UnifiedLogger$debug(message, context = NULL)
```

Arguments:

`message` Log message

`context` Additional context (optional)

Method `info()`**:** Log an info message

Usage:

```
UnifiedLogger$info(message, context = NULL)
```

Arguments:

`message` Log message

`context` Additional context (optional)

Method `warn()`**:** Log a warning message

Usage:

```
UnifiedLogger$warn(message, context = NULL)
```

Arguments:

`message` Log message

`context` Additional context (optional)

Method `error()`**:** Log an error message

Usage:

```
UnifiedLogger$error(message, context = NULL)
```

Arguments:

`message` Log message

`context` Additional context (optional)

Method `log_api_call()`**:** Log API call performance

Usage:

```
UnifiedLogger$log_api_call(
  provider,
  model,
  duration,
  success = TRUE,
  tokens = NULL
)
```

Arguments:

`provider` API provider name

`model` Model name

`duration` Duration in seconds

`success` Whether the call was successful

`tokens` Number of tokens used (optional)

Method `log_api_request_response()`**:** Log complete API request and response for debugging and audit

Usage:

```
UnifiedLogger$log_api_request_response(
  provider,
  model,
  prompt_content,
  response_content,
  request_metadata = NULL,
  response_metadata = NULL
)
```

Arguments:

`provider` API provider name

`model` Model name

`prompt_content` The complete prompt sent to the API

`response_content` The complete response received from the API

`request_metadata` Additional request metadata (optional)

`response_metadata` Additional response metadata (optional)

Method `log_cache_operation()`**:** Log cache operations

Usage:

```
UnifiedLogger$log_cache_operation(operation, key, size = NULL)
```

Arguments:

`operation` Operation type: "hit", "miss", "store", "clear"

`key` Cache key

`size` Size of cached data (optional)

Method `log_cluster_progress()`**:** Log cluster annotation progress

Usage:

```
UnifiedLogger$log_cluster_progress(cluster_id, stage, progress = NULL)
```

Arguments:

`cluster_id` Cluster identifier

`stage` Current stage

`progress` Progress information

Method `log_discussion()`**:** Log detailed cluster discussion with complete model conversations

Usage:

```
UnifiedLogger$log_discussion(cluster_id, event_type, data = NULL)
```

Arguments:

`cluster_id` Cluster identifier

`event_type` Type of event (start, prediction, consensus, end)

`data` Event data

Method `get_performance_summary()`**:** Get performance summary

Usage:

```
UnifiedLogger$get_performance_summary()
```

Returns: List of performance statistics

Method `cleanup_logs()`**:** Clean up old log files

Usage:

```
UnifiedLogger$cleanup_logs(force = FALSE)
```

Arguments:

`force` Force cleanup even if within file limits

Method `set_level()`**:** Set logging level

Usage:

```
UnifiedLogger$set_level(level)
```

Arguments:

`level` New logging level: DEBUG, INFO, WARN, ERROR

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

ZhipuProcessor *Zhipu API Processor*

Description

Zhipu API Processor

Zhipu API Processor

Details

Concrete implementation of BaseAPIProcessor for Zhipu models. Handles Zhipu-specific API calls, authentication, and response parsing.

Super class

`mLLMCelltype::BaseAPIProcessor` -> `ZhipuProcessor`

Methods

Public methods:

- `ZhipuProcessor$new()`
- `ZhipuProcessor$get_default_api_url()`
- `ZhipuProcessor$make_api_call()`
- `ZhipuProcessor$extract_response_content()`
- `ZhipuProcessor$clone()`

Method `new()`**:** Initialize Zhipu processor

Usage:

```
ZhipuProcessor$new(base_url = NULL)
```

Arguments:

`base_url` Optional custom base URL for Zhipu API

Method `get_default_api_url()`**:** Get default Zhipu API URL

Usage:

```
ZhipuProcessor$get_default_api_url()
```

Returns: Default Zhipu API endpoint URL

Method `make_api_call()`**:** Make API call to Zhipu

Usage:

```
ZhipuProcessor$make_api_call(chunk_content, model, api_key)
```

Arguments:

`chunk_content` Content for this chunk

`model` Model identifier

`api_key` API key

Returns: httr response object

Method `extract_response_content()`: Extract response content from Zhipu API response

Usage:

```
ZhipuProcessor$extract_response_content(response, model)
```

Arguments:

`response` httr response object

`model` Model identifier

Returns: Extracted text content

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Index