Package ‘llmflow’

October 17, 2025

Type Package

Title Streamlined LLM-R Integration with Intelligent Workflow Automation

Version 2.0.0

Maintainer The package maintainer `<yourself@somewhere.net>`

Description Provides a comprehensive framework for integrating Large Language Models (LLMs) with R programming through intelligent workflow automation. Built on the ReAct (Reasoning and Acting) architecture, 'llmflow' enables seamless bi-directional communication between LLMs and R environments. Key features include automated code generation and execution, intelligent error handling with retry mechanisms, persistent session management, structured JSON output validation, and context-aware conversation management. The package transforms complex LLM interactions into streamlined, reproducible workflows for data analysis, statistical computing, and research automation. Designed for data scientists, researchers, and analysts who want to leverage AI assistance while maintaining full control over their R computational environment.

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.2

Imports ellmer

Contents

`AutoFlow` *AutoFlow - Automated R Analysis Workflow with LLM*

Description

AutoFlow - Automated R Analysis Workflow with LLM

Usage

```
AutoFlow(
  react_llm,
  task_prompt,
  rag_llm = NULL,
  max_turns = 15,
  pkgs_to_use = c(),
  objects_to_use = list(),
  existing_session = NULL,
  verbose = TRUE,
  r_session_options = list(),
  context_window_size = 3000,
  max_observation_length = 800,
  error_escalation_threshold = 3
)
```

Arguments

`react_llm`	Chat object for ReAct task execution (required)
`task_prompt`	Task description (required)
`rag_llm`	Chat object for RAG documentation retrieval (default: NULL, uses react_llm)
`max_turns`	Maximum ReAct turns (default: 15)
`pkgs_to_use`	Packages to load in R session
`objects_to_use`	Named list of objects to load
`existing_session`	Existing callr R session
`verbose`	Verbose output (default: TRUE)
`r_session_options`	Options for callr R session

context_window_size
Context window size for history

max_observation_length
Maximum observation length

error_escalation_threshold
Error count threshold

Details

Dual-LLM Architecture:

AutoFlow supports using different models for different purposes: - ‘rag_llm‘: Retrieval-Augmented Generation - retrieves relevant function documentation - ‘react_llm‘: ReAct execution - performs reasoning and action loops

Why separate models? - RAG tasks are simple (extract function names) -> use fast/cheap models - ReAct tasks are complex (coding, reasoning) -> use powerful models - Cost savings: ~70

If ‘rag_llm‘ is NULL, both operations use ‘react_llm‘.

Value

ReAct result object

Examples

```
## Not run:
# Simple: same model for both
llm <- llm_openai(model = "gpt-4o")
result <- AutoFlow(llm, "Load mtcars and plot mpg vs hp")

# Optimized: lightweight RAG, powerful ReAct
rag <- llm_openai(model = "gpt-3.5-turbo") # Fast & cheap
react <- llm_openai(model = "gpt-4o") # Powerful
result <- AutoFlow(
  react_llm = react,
  task_prompt = "Perform PCA on iris dataset",
  rag_llm = rag
)

# Cross-provider: DeepSeek RAG + Claude ReAct
rag <- chat_deepseek(model = "deepseek-chat")
react <- chat_anthropic(model = "claude-sonnet-4-20250514")
result <- AutoFlow(react, "Complex analysis", rag_llm = rag)

# Batch evaluation with shared RAG
rag <- chat_deepseek(model = "deepseek-chat")
react <- chat_openai(model = "gpt-4o")

for (task in tasks) {
  result <- AutoFlow(react, task, rag_llm = rag, verbose = FALSE)
}

## End(Not run)
```

extract_bash_code *Extract Bash/Shell code from a string*

Description

This function extracts Bash/Shell code from a string by matching all content between '“‘bash', '“‘sh', '“‘shell' and '“‘'.

Usage

```
extract_bash_code(input_string)
```

Arguments

`input_string`	A string containing Bash/Shell code blocks, typically a response from an LLM

Value

A character vector containing the extracted Bash/Shell code

Examples

~~~
# Simple bash example
text <- "Run this:\n```bash\necho 'Hello'\n```"
extract_bash_code(text)

# Using 'sh' tag
text <- "```sh\nls -la\npwd\n```"
extract_bash_code(text)

# Using 'shell' tag
text <- "```shell\nfor i in {1..5}; do echo $i; done\n```"
extract_bash_code(text)

# Multiple blocks with different tags
response <- "
Setup script:
```bash
#!/bin/bash
mkdir -p /tmp/test
cd /tmp/test
~~~

Installation:
```sh
apt-get update
apt-get install -y git
```

Configuration:
```shell
export PATH=$PATH:/usr/local/bin
source ~/.bashrc
```
```
```

~~~
"
codes <- extract_bash_code(response)
length(codes) # Returns 3

# Complex script example
script_response <- "
Here's a backup script:
```bash
#!/bin/bash

# Set variables
BACKUP_DIR='/backup'
DATE=$(date +%Y%m%d)

# Create backup
tar -czf ${BACKUP_DIR}/backup_${DATE}.tar.gz /home/user/

# Check if successful
if [ $? -eq 0 ]; then
    echo 'Backup completed successfully'
else
    echo 'Backup failed'
    exit 1
fi
```
"
extract_bash_code(script_response)
~~~

`extract_chat_history` *Extract chat history from ellmer chat object*

Description

Extract chat history from ellmer chat object

Usage

```
extract_chat_history(
  chat_obj,
  include_tokens = TRUE,
  include_time = TRUE,
  tz = "Asia/Shanghai"
)
```

Arguments

`chat_obj`	An ellmer chat object
`include_tokens`	Whether to include token information
`include_time`	Whether to include timestamp information
`tz`	Time zone for timestamps (default "Asia/Shanghai" for CST)

Value

A data frame with chat history

---

extract_code *Generic function to extract code of any specified language*

Description

This function provides a flexible way to extract code blocks of any language from a string by specifying the language identifier(s).

Usage

```
extract_code(input_string, language, case_sensitive = FALSE)
```

Arguments

input_string	A string containing code blocks
language	Language identifier(s) to extract (e.g., "r", "python", c("bash", "sh"))
case_sensitive	Whether the language matching should be case-sensitive (default: FALSE)

Value

A character vector containing the extracted code

Examples

~~~
# Extract R code
text <- "```r\nx <- 1:10\n```"
extract_code(text, "r")

# Extract multiple language variants
text <- "```bash\necho 'test'\n```\n```sh\nls -la\n```"
extract_code(text, c("bash", "sh"))

# Case-sensitive extraction
text <- "```R\nplot(1:10)\n```\n```r\nprint('hello')\n```"
extract_code(text, "r", case_sensitive = TRUE) # Only matches lowercase 'r'
extract_code(text, "r", case_sensitive = FALSE) # Matches both 'R' and 'r'

# Extract custom language
text <- "```julia\nprintln(\"Julia code\")\n```"
extract_code(text, "julia")

# Extract YAML configuration
config_text <- "
Here's the configuration:
```yaml
database:
  host: localhost
  port: 5432
  name: mydb
```
~~~

~~~
"
extract_code(config_text, "yaml")

# Extract multiple TypeScript and JavaScript blocks
mixed_text <- "
TypeScript:
```typescript
interface User {
    name: string;
    age: number;
}
```

JavaScript:
```js
const user = {name: 'John', age: 30};
```
"
# Extract TypeScript
extract_code(mixed_text, "typescript")
# Extract both TypeScript and JavaScript
extract_code(mixed_text, c("typescript", "js"))
~~~

---

extract_docs_from_tarball

Extract Documentation from Package Tarball

Description

This function extracts comprehensive documentation from an uninstalled R package in tar.gz format. It processes the .Rd files directly from the tarball without requiring package installation. The return structure is identical to `extract_package_docs`, allowing you to use `generate_qa_from_docs` directly on the results.

Usage

```
extract_docs_from_tarball(tar_path)
```

Arguments

`tar_path`	A character string specifying the full path to the package tar.gz file

Details

This is particularly useful for:

- Batch processing large collections of packages (e.g., Bioconductor)
- Extracting documentation from packages that cannot be installed
- Processing packages in environments without installation privileges

Value

A named list where each element corresponds to a function in the package. Each function's documentation includes:

- `package`: Package name
- `function_name`: Function name
- `title`: Function title
- `description`: Detailed description
- `usage`: Usage syntax
- `parameters`: Data frame of parameters with defaults
- `return_value`: Description of return value
- `examples`: Example code
- `formatted_arguments`: Formatted parameter descriptions
- `simple_arguments`: Required parameters only

Returns `NULL` if extraction fails or no documentation is found.

Examples

```
## Not run:
# Extract documentation from a tarball
docs <- extract_docs_from_tarball("path/to/dplyr_1.0.0.tar.gz")

# Use with existing QA generation function
qa_pairs <- generate_qa_from_docs(docs)

# View documentation for a specific function
docs$filter

# List all extracted functions
names(docs)

## End(Not run)
```

`extract_function_docs` *Extract Documentation for a Single Function*

Description

This function extracts comprehensive documentation for a specific function in an R package. It retrieves the function's title, description, usage pattern, parameters with defaults, return value, and examples. Unlike `extract_package_docs`, this function targets a single function and returns its documentation directly rather than in a list.

Usage

```
extract_function_docs(package_name, function_name)
```

Arguments

`package_name`	A character string specifying the name of the package
`function_name`	A character string specifying the name of the function

Value

A list containing the function's documentation with the following elements:

- `package`: Package name
- `function_name`: Function name
- `title`: Function title
- `description`: Detailed description
- `usage`: Usage syntax
- `parameters`: Data frame of parameters with defaults
- `return_value`: Description of return value
- `examples`: Example code
- `formatted_arguments`: Formatted parameter descriptions
- `simple_arguments`: Required parameters only

Returns `NULL` if the function is not found or an error occurs.

Examples

```
## Not run:
# Extract documentation for dplyr's filter function
filter_doc <- extract_function_docs("dplyr", "filter")

# View the description
cat(filter_doc$description)

# View the parameters
print(filter_doc$parameters)

## End(Not run)
```

`extract_function_examples`
Extract Examples from a Package Function

Description

This function extracts and cleans the examples section from a specific function's documentation in an R package. It uses the 'tools' package to access the Rd database and extracts examples using 'tools::Rd2ex()'. The output is cleaned to remove metadata headers and formatting artifacts.

Usage

```
extract_function_examples(package_name, function_name)
```

Arguments

`package_name`	A character string specifying the name of the package
`function_name`	A character string specifying the name of the function

Value

A character string containing the cleaned examples code, or ‘NA‘ if no examples are found or an error occurs

Examples

```
## Not run:
# Extract examples from ggplot2's geom_point function
examples <- extract_function_examples("ggplot2", "geom_point")
cat(examples)

## End(Not run)
```

`extract_javascript_code` *Extract JavaScript code from a string*

Description

This function extracts JavaScript code from a string by matching all content between ’“‘javascript’, ’“‘js’, ’“‘jsx’ and ’“‘.

Usage

```
extract_javascript_code(input_string)
```

Arguments

`input_string`	A string containing JavaScript code blocks, typically a response from an LLM

Value

A character vector containing the extracted JavaScript code

Examples

```
# Simple JavaScript example
text <- "Code:\n```javascript\nconsole.log('Hello');\n```"
extract_javascript_code(text)

# Using 'js' tag
text <- "```js\nconst x = 42;\n```"
extract_javascript_code(text)

# Using 'jsx' tag for React
text <- "```jsx\n<div>Hello World</div>\n```"
extract_javascript_code(text)
```

~~~~
# Multiple blocks with different tags
response <- "
Frontend code:
```javascript
function fetchData() {
    return fetch('/api/data')
        .then(response => response.json());
}
```

React component:
```jsx
const MyComponent = () => {
    const [data, setData] = useState([]);

    useEffect(() => {
        fetchData().then(setData);
    }, []);

    return (
        <div>
            {data.map(item => <p key={item.id}>{item.name}</p>)}
        </div>
    );
};
```

Node.js backend:
```js
const express = require('express');
const app = express();

app.get('/api/data', (req, res) => {
    res.json([{id: 1, name: 'Item 1'}]);
});

app.listen(3000);
```
"
codes <- extract_javascript_code(response)
length(codes) # Returns 3
~~~~

`extract_json` *Extract and parse JSONs from a string (LLM response)*

Description

This function extracts JSON blocks from a string and parses them using ‘jsonlite::fromJSON()‘. This can be used to extract all JSONs from LLM responses, immediately converting them to R objects.

Usage

```
extract_json(llm_response)
```

Arguments

`llm_response`	A character string

Value

A list of parsed JSON objects

`extract_package_docs` *Extract Documentation for All Functions in a Package*

Description

This function extracts comprehensive documentation for all exported functions in an R package. It retrieves function names, titles, descriptions, usage patterns, parameters with defaults, return values, and examples. The function processes the package's Rd database and returns a structured list containing all documentation elements.

Usage

```
extract_package_docs(package_name)
```

Arguments

`package_name`	A character string specifying the name of the package to extract documentation from

Value

A named list where each element corresponds to a function in the package. Each function's documentation includes:

- `package`: Package name
- `function_name`: Function name
- `title`: Function title
- `description`: Detailed description
- `usage`: Usage syntax
- `parameters`: Data frame of parameters with defaults
- `return_value`: Description of return value
- `examples`: Example code
- `formatted_arguments`: Formatted parameter descriptions
- `simple_arguments`: Required parameters only

Examples

```
## Not run:
# Extract documentation from dplyr
docs <- extract_package_docs("dplyr")

# View documentation for a specific function
docs$filter

# List all extracted functions
names(docs)

## End(Not run)
```

`extract_python_code` *Extract Python code from a string*

Description

This function extracts Python code from a string by matching all content between '"'python', '"'py' and '"''.

Usage

```
extract_python_code(input_string)
```

Arguments

`input_string`	A string containing Python code blocks, typically a response from an LLM

Value

A character vector containing the extracted Python code

Examples

~~~
# Simple example
text <- "Python code:\n```python\nprint('Hello World')\n```"
extract_python_code(text)

# Using 'py' tag
text <- "```py\nimport numpy as np\n```"
extract_python_code(text)

# Multiple blocks with different tags
response <- "
Data processing:
```python
import pandas as pd
df = pd.read_csv('data.csv')
df.head()
```
~~~

~~~
Visualization:
```py
import matplotlib.pyplot as plt
plt.plot([1, 2, 3], [4, 5, 6])
plt.show()
~~~
"
codes <- extract_python_code(response)
length(codes) # Returns 2

# Complex example with classes and functions
llm_response <- "
Here's a complete Python solution:
```python
class DataProcessor:
    def __init__(self, data):
        self.data = data

    def process(self):
        return [x * 2 for x in self.data]

processor = DataProcessor([1, 2, 3])
result = processor.process()
print(result)
```
"
extract_python_code(llm_response)
```

---

## `extract_r_code` *Extract R code from a string*

---

### Description

This function extracts R code from a string by matching all content between '“‘r' or '“‘R' and '“‘'.

### Usage

```
extract_r_code(input_string)
```

Arguments

`input_string`	A string containing R code blocks, typically a response from an LLM

Value

A character vector containing the extracted R code

Examples

```
# Simple example
text <- "Here is some R code:\n```r\nprint('Hello')\n```"
extract_r_code(text)
```
```

~~~
# Multiple code blocks
response <- "
First block:
```r
x <- 1:10
mean(x)
```

Second block:
```R
library(ggplot2)
ggplot(mtcars, aes(mpg, hp)) + geom_point()
```
"
codes <- extract_r_code(response)
length(codes) # Returns 2

# With surrounding text
llm_response <- "
To calculate the mean, use this code:
```r
data <- c(1, 2, 3, 4, 5)
result <- mean(data)
print(result)
```
The result will be 3.
"
extract_r_code(llm_response)
~~~

`extract_sql_code` *Extract SQL code from a string*

Description

This function extracts SQL code from a string by matching all content between ’“‘sql’ and ’“‘’ (case-insensitive).

Usage

```
extract_sql_code(input_string)
```

Arguments

`input_string`	A string containing SQL code blocks, typically a response from an LLM

Value

A character vector containing the extracted SQL code

Examples

~~~
# Simple SQL query
text <- "Query:\n```sql\nSELECT * FROM users;\n```"
extract_sql_code(text)

# Case-insensitive matching
text <- "```SQL\nSELECT COUNT(*) FROM orders;\n```"
extract_sql_code(text)

# Multiple SQL blocks
response <- "
Create table:
```sql
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

Insert data:
```sql
INSERT INTO employees (id, name, department, salary)
VALUES
    (1, 'John Doe', 'IT', 75000),
    (2, 'Jane Smith', 'HR', 65000);
```

Query data:
```sql
SELECT name, salary
FROM employees
WHERE department = 'IT'
ORDER BY salary DESC;
```
"
codes <- extract_sql_code(response)
length(codes) # Returns 3

# Complex query with joins
complex_query <- "
Here's the analysis query:
```sql
WITH monthly_sales AS (
    SELECT
        DATE_TRUNC('month', order_date) as month,
        SUM(total_amount) as total_sales,
        COUNT(DISTINCT customer_id) as unique_customers
    FROM orders
    WHERE order_date >= '2024-01-01'
    GROUP BY DATE_TRUNC('month', order_date)
)
SELECT
    month,
    total_sales,
~~~

```
    unique_customers,
    total_sales / unique_customers as avg_per_customer
FROM monthly_sales
ORDER BY month;
```
"
extract_sql_code(complex_query)
```

`generate_qa_from_docs` *Generate Question-Answer Pairs from Package Documentation*

Description

This function generates comprehensive question-answer (QA) pairs optimized for R code generation and learning from package documentation. It processes the documentation structure returned by `extract_package_docs` and creates multiple types of QA pairs for each function, including package context, purpose, syntax, parameters, examples, and return values.

Usage

```
generate_qa_from_docs(docs)
```

Arguments

`docs`	A named list of function documentation, typically returned by `extract_package_docs`. Each element should contain function metadata including package name, function name, description, usage, parameters, examples, and return value.

Value

A list of QA pairs where each element contains:

- `question`: A natural language question about the function
- `answer`: The corresponding answer, often containing R code
- `package`: The source package name
- `function_name`: The function name
- `qa_type`: The type of QA pair (e.g., "package_context", "purpose", "syntax", "examples", etc.)

The function generates up to 8 different types of QA pairs per function:

1. **package_context**: Which package to load
2. **purpose**: When and why to use the function
3. **syntax**: Correct function call syntax
4. **essential_parameters**: Required arguments
5. **all_parameters**: All available parameters
6. **examples**: Working code examples
7. **code_completion**: Parameter completion patterns
8. **return_value**: Expected output
```

## Examples

```
## Not run:
# Extract documentation from a package
docs <- extract_package_docs("dplyr")

# Generate QA pairs
qa_pairs <- generate_qa_from_docs(docs)

# View a specific QA pair
qa_pairs[[1]]

# Filter by QA type
examples_qa <- Filter(function(x) x$qa_type == "examples", qa_pairs)

## End(Not run)
```

`hello` *Hello, World!*

Description

Prints 'Hello, world!'.

Usage

```
hello()
```

Examples

```
hello()
```

`package_extraction_prompt` *Generate Function Extraction Prompt for LLM Analysis*

Description

Creates a highly refined prompt that guides LLMs to identify ONLY the most documentation-critical, domain-specific R functions from a task description. The prompt uses sophisticated filtering criteria to exclude common, well-known functions (like read.csv, mean, order) that any LLM can use correctly without explicit documentation, focusing instead on specialized functions where examples truly add value.

Usage

```
package_extraction_prompt(
  task_description,
  include_criteria = NULL,
  exclude_criteria = NULL,
  prioritization_factors = NULL,
  emphasis = NULL
)
```

Arguments

`task_description`	Character string. Detailed description of the R task or analysis workflow that needs to be performed. Should include: - Data types and sources involved - Analytical objectives and methods - Expected outputs or deliverables - Domain-specific context (e.g., bioinformatics, spatial analysis) The more domain-specific the description, the better the function selection.
`include_criteria`	Character vector. Additional inclusion criteria beyond the defaults. Specify domain-specific requirements or function characteristics that should be documented. Default is NULL (use standard criteria).
`exclude_criteria`	Character vector. Additional exclusion criteria beyond the defaults. Specify function types or patterns that should be skipped (e.g., "Basic ggplot2 themes", "Standard dplyr verbs"). Default is NULL.
`prioritization_factors`	Character vector. Additional factors for prioritizing functions beyond the defaults. Specify what makes certain functions more important to document. Default is NULL (use standard priorities).
`emphasis`	Character string. Additional emphasis or context to guide the extraction process. Use this to highlight specific aspects of the task or to emphasize certain types of functions. Default is NULL.

Details

This function applies a "documentation necessity test": only include functions where a proficient LLM would struggle without explicit documentation and examples. This dramatically improves output quality and reduces token waste.

The enhanced prompt applies a rigorous "documentation necessity test" with four key questions:

1. Would a proficient LLM struggle without documentation? 2. Is this function domain-specific or universally known? 3. Does it use specialized terminology or workflows? 4. Would examples significantly improve usage accuracy?

Automatic exclusions (common functions that waste tokens): - Data I/O: read.csv, write.csv, readLines - Basic operations: order, sort, subset, head, tail - Simple statistics: mean, median, sd, sum - Core structures: c, list, data.frame - Well-known tidyverse: simple dplyr::filter, dplyr::mutate - Basic control flow: if, for, while - Common utilities: paste, grep, unique

What gets included (documentation-critical functions): - Domain-specific methods (clusterProfiler::enrichGO for GO analysis) - Complex statistical procedures (DESeq2::DESeq) - Specialized transformations (sf::st_transform for spatial data) - Functions with many non-obvious parameters - Methods where wrong usage produces plausible but incorrect results

This approach ensures that "GO enrichment analysis" returns clusterProfiler functions, NOT read.csv or order.

Value

Character string containing the complete extraction prompt with:

- Clear documentation necessity principle
- Strict inclusion criteria for domain-specific functions
- Comprehensive exclusion rules with concrete examples

- Four-question decision heuristic for each function
- Concrete good/bad examples from multiple domains
- Prioritization by domain specialization and complexity
- Quality-over-quantity guidance

See Also

retrieve_docs for using this prompt in documentation extraction

Examples

```
# Basic usage with improved prompt engineering
# Now automatically excludes common functions like read.csv, order, mean
prompt <- package_extraction_prompt(
  "Perform GO enrichment analysis on differentially expressed genes"
)
# Will focus on: clusterProfiler::enrichGO, org.Hs.eg.db::org.Hs.eg.db
# Will exclude: read.csv, dplyr::filter, order, mean

# Compare: general data analysis (less domain-specific)
prompt1 <- package_extraction_prompt(
  "Clean and analyze time series sales data using tidyverse"
)
# Will likely return fewer functions since tidyverse basics are well-known

# Compare: highly specialized domain
prompt2 <- package_extraction_prompt(
  "Spatial analysis of seismic events using terra and sf packages"
)
# Will focus on: terra::extract, sf::st_transform, sf::st_join
# Will exclude: basic ggplot2, dplyr operations

# Add domain-specific guidance
prompt <- package_extraction_prompt(
  task_description = "Single-cell RNA-seq analysis with Seurat",
  include_criteria = c(
    "Seurat-specific normalization and scaling methods",
    "Dimensionality reduction functions unique to scRNA-seq"
  ),
  exclude_criteria = c(
    "Standard dplyr data manipulation",
    "Basic ggplot2 visualization"
  ),
  emphasis = "Focus on Seurat functions that require understanding of scRNA-seq
              methodology, not general R programming constructs."
)

# Use with retrieve_docs - complete workflow
docs <- retrieve_docs(
  chat_obj = llm,
  prompt = package_extraction_prompt(
    task_description = "Perform differential expression and pathway enrichment",
    emphasis = "Exclude basic data manipulation, focus on statistical methods"
  )
)
# Result: Only domain-critical functions with examples
```

`package_function_schema` *Create JSON Schema for Package Function Validation*

Description

Create JSON Schema for Package Function Validation

Usage

```
package_function_schema(
  min_functions = 0,
  max_functions = 10,
  description =
   "Array of ONLY the most critical, domain-specific functions that truly require documentation (exc
)
```

Arguments

`min_functions`	Integer. Minimum number of functions (default: 0 to allow empty)
`max_functions`	Integer. Maximum number of functions (default: 10)
`description`	Character string. Custom description

Value

List containing JSON schema

`prompt_from_history` *Build prompt from chat history*

Description

Build prompt from chat history

Usage

```
prompt_from_history(
  chat_obj,
  add_text = NULL,
  add_role = "user",
  start_turn_index = 1
)
```

Arguments

`chat_obj`	Chat object
`add_text`	Additional text to append
`add_role`	Role for add_text ("user" or "assistant")
`start_turn_index`	Starting turn index (default 1)

Value

Formatted prompt string

`react_r` *Simplified interface - Enhanced react_r*

Description

Simplified interface - Enhanced react_r

Usage

```
react_r(chat_obj, task, ...)
```

Arguments

`chat_obj`	Chat object
`task`	Task description
`...`	Additional arguments passed to react_using_r

Value

Formatted result display

`react_using_r` *ReAct (Reasoning and Acting) using R code execution - Optimized Version*

Description

ReAct (Reasoning and Acting) using R code execution - Optimized Version

Usage

```
react_using_r(
  chat_obj,
  task,
  max_turns = 15,
  pkgs_to_use = c(),
  objects_to_use = list(),
  existing_session = NULL,
  verbose = TRUE,
  r_session_options = list(),
  context_window_size = 3000,
  max_observation_length = 800,
  error_escalation_threshold = 3
)
```

Arguments

`chat_obj`	Chat object from ellmer
`task`	Character string. The task description to be solved
`max_turns`	Integer. Maximum number of ReAct turns (default: 15)
`pkgs_to_use`	Character vector. R packages to load in session
`objects_to_use`	Named list. Objects to load in R session
`existing_session`	Existing callr session to continue from (optional)
`verbose`	Logical. Whether to print progress information
`r_session_options`	List. Options for callr R session
`context_window_size`	Integer. Maximum characters before history summary (default: 3000)
`max_observation_length`	Integer. Maximum observation length (default: 800)
`error_escalation_threshold`	Integer. Error count threshold for escalation (default: 3)

Value

List with complete ReAct results

`response_as_json` *Get JSON response from LLM with validation and retry*

Description

Get JSON response from LLM with validation and retry

Usage

```
response_as_json(
  chat_obj,
  prompt,
  schema = NULL,
  schema_strict = FALSE,
  max_iterations = 3
)
```

Arguments

`chat_obj`	Chat object (LLM client). Must be a properly initialized LLM client instance.
`prompt`	Character string. The user prompt to send to the LLM requesting JSON output.
`schema`	List or NULL. Optional JSON schema for response validation (as R list structure). When provided, validates the LLM response against this schema. Default is NULL.
`schema_strict`	Logical. Whether to use strict schema validation (no additional properties allowed). Only applies when schema is provided. Default is FALSE.
`max_iterations`	Integer. Maximum number of retry attempts for invalid JSON or schema validation failures. Must be positive. Default is 3.

Value

List. Parsed JSON response from the LLM.

Examples

```
## Not run:
# Basic usage without schema
result <- response_as_json(
  chat_obj = llm_client,
  prompt = "List three colors"
)

# With schema validation
schema <- list(
  type = "object",
  properties = list(
    equation = list(type = "string"),
    solution = list(type = "number")
  ),
  required = c("equation", "solution")
)
result <- response_as_json(
  chat_obj = llm_client,
  prompt = "How can I solve 8x + 7 = -23?",
  schema = schema,
  schema_strict = TRUE,
  max_iterations = 3
)

## End(Not run)
```

`response_to_r` *Response to R code generation and execution with session continuity*

Description

Response to R code generation and execution with session continuity

Usage

```
response_to_r(
  chat_obj,
  prompt,
  add_text = NULL,
  pkgs_to_use = c(),
  objects_to_use = list(),
  existing_session = NULL,
  list_packages = TRUE,
  list_objects = TRUE,
  return_session_info = TRUE,
  evaluate_code = TRUE,
  r_session_options = list(),
  return_mode = c("full", "code", "console", "object", "formatted_output", "llm_answer",
    "session"),
  max_iterations = 3
)
```

Arguments

`chat_obj`	Chat object from ellmer
`prompt`	User prompt for R code generation
`add_text`	Additional instruction text
`pkgs_to_use`	Packages to load in R session
`objects_to_use`	Named list of objects to load in R session
`existing_session`	Existing callr session to continue from (optional)
`list_packages`	Whether to list available packages in prompt
`list_objects`	Whether to list available objects in prompt
`return_session_info`	Whether to return session state information
`evaluate_code`	Whether to evaluate the generated code
`r_session_options`	Options for callr R session
`return_mode`	Return mode specification
`max_iterations`	Maximum retry attempts

Value

Result based on return_mode

retrieve_docs *Retrieve and Format R Function Documentation for LLM Consumption*

Description

Retrieve and Format R Function Documentation for LLM Consumption

Usage

```
retrieve_docs(
  chat_obj,
  prompt,
  schema = package_function_schema(),
  schema_strict = TRUE,
  skip_undocumented = TRUE,
  use_llm_fallback = FALSE,
  example_count = 2,
  warn_skipped = TRUE
)
```

Arguments

`chat_obj`	LLM chat client object
`prompt`	Task description
`schema`	JSON schema (default: package_function_schema())
`schema_strict`	Strict schema validation
`skip_undocumented`	Skip functions without docs
`use_llm_fallback`	Use LLM to generate examples
`example_count`	Number of examples per function
`warn_skipped`	Show warning for skipped functions

Value

Formatted documentation string

save_code_to_file *Save extracted code to file*

Description

This function saves extracted code to a file with appropriate extension based on the programming language.

Usage

```
save_code_to_file(code_string, filename = NULL, language = "r")
```

Arguments

`code_string`	String or character vector containing the code to save
`filename`	Output filename. If NULL, generates a timestamped filename
`language`	Programming language for determining file extension (default: "r")

Value

The path to the saved file

Examples

```
## Not run:
# Extract and save R code
llm_response <- "```r\nplot(1:10)\n```"
code <- extract_r_code(llm_response)
save_code_to_file(code) # Saves as "code_20240101_120000.R"

# Save with custom filename
save_code_to_file(code, "my_plot.R")

# Save Python code with auto extension
py_code <- "import pandas as pd\ndf = pd.DataFrame()"
save_code_to_file(py_code, language = "python") # Creates .py file

# Save multiple code blocks
response <- "```r\nx <- 1\n```\n```r\ny <- 2\n```"
codes <- extract_r_code(response)
save_code_to_file(codes, "combined_code.R")

## End(Not run)
```

Index