Package 'LLMAgentR'

April 28, 2025

Type Package

Title Language Model Agents in R for AI Workflows and Research

Version 0.2.0

Maintainer Kwadwo Daddy Nyame Owusu Boakye <kwadwo.owusuboakye@outlook.com>

Description Provides modular, graph-
based agents powered by large language models (LLMs) for intelligent task execution in R.
Supports structured workflows for tasks such as forecasting, data visualization, feature engineering, data wrangling, data cleaning, SQL, code generation, weather reporting, and research-driven question answering.
Each agent performs iterative reasoning: recommending steps, generating R code, executing, debugging, and explaining results.
Includes built-in support for packages such as 'tidymodels', 'modeltime', 'plotly', 'gg-
plot2', and 'prophet'. Designed for analysts, developers, and teams building intelligent, reproducible AI workflows in R.
Compatible with LLM providers such as 'OpenAI', 'Anthropic', 'Groq', and 'Ollama'. Inspired by the Python package 'langagent'.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

URL https://github.com/knowusuboaky/LLMAgentR

BugReports https://github.com/knowusuboaky/LLMAgentR/issues

Depends R (>= 4.1.0)

Imports plotly, stats, utils, DBI, RSQLite, dplyr, glue, httr,
officer, purrr, timetk, pdftools, parsnip, recipes, workflows,
rsample, modeltime.ensemble, modeltime, xml2

Suggests testthat (>= 3.0.0), roxygen2, jsonlite, magrittr, rlang,
tidyr, ggplot2, usethis, prophet, forcats, kernlab, xgboost,
xfun, modeltime.resample, tidymodels, tibble, lubridate,
methods, stringr, tesseract, rvest, fastDummies

NeedsCompilation no

Author Kwadwo Daddy Nyame Owusu Boakye [aut, cre]

Repository CRAN

Date/Publication 2025-04-28 18:40:15 UTC

Contents

`build_code_agent` *Build an R Code Generation Agent*

Description

Constructs an LLM-based agent for generating, debugging, explaining, or optimizing R code using structured prompts. The agent handles retries and provides comprehensive code assistance.

Arguments

`llm`	A function that accepts a character prompt and returns an LLM response.
`system_prompt`	Optional system-level instructions for the agent's behavior.
`user_input`	The user's task/query (e.g., "Write function to filter NAs").
`max_tries`	Maximum number of attempts for LLM calls (default: 3).
`backoff`	Seconds to wait between retries (default: 2).
`verbose`	Logical controlling progress messages (default: TRUE).

Value

A list containing:

- input - The user's original query
- llm_response - The processed LLM response
- system_prompt - The system instructions used
- success - Logical indicating if call succeeded
- attempts - Number of tries made

Examples

```
## Not run:
agent <- build_code_agent(
  llm = call_llm,
  user_input = "Write function to remove NA rows from dataframe",
  verbose = FALSE
)

## End(Not run)
```

---

build_data_cleaning_agent

Build a Data Cleaning Agent

Description

Constructs a multi-step agent workflow to recommend, generate, fix, execute, and explain robust R code for data cleaning tasks using LLMs and user-defined data.

Arguments

model	A function that accepts a prompt and returns a text response (e.g., OpenAI, Claude).
data_raw	A raw data.frame (or list convertible to data.frame) to be cleaned.
human_validation	Logical; whether to include a manual review step.
bypass_recommended_steps	Logical; whether to skip LLM-based cleaning step suggestions.
bypass_explain_code	Logical; whether to skip explanation of the generated code.
verbose	Logical; whether to print progress messages (default: TRUE)

Value

A compiled graph-based cleaning agent function that accepts and mutates a state list.

Examples

```
## Not run:
state <- list(
  data_raw = mtcars,
  user_instructions = "Don't remove outliers when cleaning the data."
)
agent <- build_data_cleaning_agent(model = call_llm, data_raw = mtcars)
agent(state)
str(state$data_cleaned)

## End(Not run)
```

`build_data_wrangling_agent` *Build a Data Wrangling Agent*

Description

Constructs a state graph-based agent that recommends, generates, executes, fixes, and explains data wrangling transformations based on user instructions and dataset structure. The resulting function handles list or single data frame inputs and produces a cleaned dataset.

Arguments

`model`	A function that takes a prompt string and returns LLM-generated output.
`human_validation`	Logical; whether to enable manual review step before code execution.
`bypass_recommended_steps`	Logical; skip initial recommendation of wrangling steps.
`bypass_explain_code`	Logical; skip final explanation step after wrangling.
`verbose`	Logical; whether to print progress messages (default: TRUE)

Value

A callable agent function that mutates a provided ‘state‘ list by populating: - ‘data_wrangled‘: the final cleaned data frame, - ‘data_wrangler_function‘: the code used, - ‘data_wrangler_error‘: any execution error (if occurred), - ‘wrangling_report‘: LLM-generated explanation (if ‘bypass_explain_code = FALSE‘)

Examples

```
## Not run:
state <- list(
  data_raw = iris,
  user_instructions = "Merge the data frames on the ID column."
)
```

```
agent <- build_data_wrangling_agent(model = call_llm)
agent(state)
print(state$data_wrangled)

## End(Not run)
```

`build_doc_summarizer_agent` *Build a Document Summarizer Agent*

Description

Creates an LLM-powered document summarization workflow that processes PDF, DOCX, PPTX, TXT, or plain text input and returns structured markdown summaries.

Usage

```
build_doc_summarizer_agent(
  llm,
  summary_template = NULL,
  chunk_size = 4000,
  overlap = 200,
  verbose = TRUE
)
```

Arguments

`llm`	A function that accepts a character prompt and returns an LLM response.
`summary_template`	Optional custom summary template in markdown format.
`chunk_size`	Maximum character length for document chunks (default: 4000).
`overlap`	Character overlap between chunks (default: 200).
`verbose`	Logical controlling progress messages (default: TRUE).

Value

A function that accepts file paths or text input and returns:

- summary - The generated markdown summary
- metadata - Document metadata if available
- chunks - Number of processing chunks used
- success - Logical indicating success

Examples

```
## Not run:
# With default settings
summarizer <- build_doc_summarizer_agent(llm = call_llm)

# With custom template
custom_template <- "Summarize this in 3 bullet points:\n{text}"
summarizer <- build_doc_summarizer_agent(
  llm = call_llm,
  summary_template = custom_template
)

# Process a document
result <- summarizer("report.docx")
cat(result$summary)

## End(Not run)
```

`build_feature_engineering_agent`

Build a Feature Engineering Agent

Description

Constructs a graph-based feature engineering agent that guides the process of: recommending, generating, executing, fixing, and explaining feature engineering code.

Arguments

`model`	A function that accepts a prompt and returns an LLM-generated response.
`human_validation`	Logical; include a manual review node before code execution.
`bypass_recommended_steps`	Logical; skip the LLM-based recommendation phase.
`bypass_explain_code`	Logical; skip final explanation step.
`verbose`	Logical; whether to print progress messages (default: TRUE)

Value

A callable agent function that executes feature engineering via a state graph.

Examples

```
## Not run:
state <- list(
  data_raw = iris,
  target_variable = "Species"
)
agent <- build_feature_engineering_agent(model = call_llm)
agent(state)
str(state$data_engineered)

## End(Not run)
```

`build_forecasting_agent`

Build a Time Series Forecasting Agent

Description

Constructs a state graph-based forecasting agent that: recommends forecasting steps, extracts parameters, generates code, executes the forecast using ‘modeltime‘, fixes errors if needed, and explains the result. It leverages multiple models including Prophet, XGBoost, Random Forest, SVM, and Prophet Boost, and combines them in an ensemble.

Arguments

`model`	A function that takes a prompt and returns an LLM-generated result.
`bypass_recommended_steps`	Logical; skip initial step recommendation.
`bypass_explain_code`	Logical; skip the final explanation step.
`mode`	Visualization mode for forecast plots. One of ‘"light"‘ or ‘"dark"‘.
`line_width`	Line width used in plotly forecast visualization.
`verbose`	Logical; whether to print progress messages.

Value

A callable agent function that mutates the given ‘state‘ list.

Examples

```
## Not run:
state <- list(
  data_raw = tsibble_data,
  user_instructions = "Forecast next 12 months of sales per store"
)
agent <- build_forecasting_agent(model = call_llm)
agent(state)
```

```
print(state$forecasting_result)

## End(Not run)
```

`build_interpreter_agent` *Build an Interpreter Agent*

Description

Constructs an agent that uses LLM to interpret various outputs (plots, tables, text results) and provides structured explanations suitable for both technical and non-technical audiences.

Arguments

`llm`	A function that accepts a character prompt and returns an LLM response.
`interpreter_prompt`	Optional custom prompt template (default provides structured interpretation framework).
`code_output`	The output to interpret (chart summary, table, text results etc.).
`max_tries`	Maximum number of attempts for LLM calls (default: 3).
`backoff`	Seconds to wait between retries (default: 2).
`verbose`	Logical controlling progress messages (default: TRUE).

Value

A list containing:

- prompt - The full prompt sent to LLM
- interpretation - The generated interpretation
- success - Logical indicating if interpretation succeeded
- attempts - Number of tries made

Examples

```
## Not run:
# Interpret ggplot output
plot_summary <- ggplot2::ggplot(mtcars, aes(mpg, hp)) + geom_point()
result <- build_interpreter_agent(
  llm = call_llm,
  code_output = capture.output(print(plot_summary))
)

## End(Not run)
```

build_researcher_agent *Build a Web Researcher Agent*

Description

Constructs an LLM-powered research agent that performs web searches (via Tavily API) and generates structured responses based on search results. The agent handles different question types (general knowledge, comparisons, controversial topics) with appropriate response formats.

Arguments

llm	A function that accepts a character prompt and returns an LLM response. (It must accept 'prompt' and optionally 'verbose'.)
tavily_search	Tavily API key as a string or NULL to use 'Sys.getenv("TAVILY_API_KEY")'.
system_prompt	Optional custom system prompt for the researcher agent.
max_results	Number of web search results to retrieve per query (default: 5).
max_tries	Maximum number of retry attempts for search or LLM call (default: 3).
backoff	Initial wait time in seconds between retries (default: 2).
verbose	Logical flag to control progress messages (default: TRUE).

Value

A function that accepts a user query string and returns a list with:

- query - The original research query.
- prompt - The full prompt sent to the LLM.
- response - The generated LLM response.
- search_results - Raw search results (if any were found).
- success - Logical indicating if research succeeded (both search and LLM).

Examples

```
## Not run:
agent <- create_researcher_agent(llm = my_llm_wrapper)
result <- agent("What are the top AI breakthroughs in 2024?")
cat(result$response)

## End(Not run)
```

`build_sql_agent` *Build a SQL Agent Graph*

Description

This function constructs a full SQL database agent using a graph-based workflow. It supports step recommendation, SQL code generation, error handling, optional human review, and automatic explanation of the final code.

Arguments

`model`	A function that accepts prompts and returns LLM responses.
`connection`	A DBI connection object to the target SQL database.
`n_samples`	Number of candidate SQL plans to consider (used in prompt).
`human_validation`	Whether to include a human review node.
`bypass_recommended_steps`	If TRUE, skip the step recommendation node.
`bypass_explain_code`	If TRUE, skip the final explanation step.
`verbose`	Logical indicating whether to print progress messages (default: TRUE).

Value

A compiled SQL agent function that runs via a state machine (graph execution).

Examples

```
## Not run:
agent <- build_sql_agent(
  model = call_llm,
  connection = DBI::dbConnect(RSQLite::SQLite(), ":memory:"),
  human_validation = FALSE,
  verbose = TRUE
)
state <- list(user_instructions = "Which Categories bring in the highest total revenue# Hint: ...")
agent(state)

## End(Not run)
```

build_visualization_agent *Build Visualization Agent*

Description

Creates a data visualization agent with configurable workflow steps.

Arguments

model	The AI model function to use for code generation
human_validation	Whether to include human validation step (default: FALSE)
bypass_recommended_steps	Skip recommendation step (default: FALSE)
bypass_explain_code	Skip explanation step (default: FALSE)
function_name	Name for generated visualization function (default: "data_visualization")
verbose	Whether to print progress messages (default: TRUE)

Value

A function that takes state and returns visualization results

build_weather_agent *Build a Weather Agent*

Description

Constructs an LLM-powered weather assistant that fetches data from OpenWeatherMap and generates user-friendly reports. Handles location parsing, API calls, caching, and LLM-based summarization.

Arguments

llm	A function that accepts a character prompt and returns an LLM response.
location_query	Free-text location query (e.g., "weather in Toronto").
system_prompt	Optional LLM system prompt for weather reporting.
weather_api_key	OpenWeatherMap API key (defaults to OPENWEATHERMAP_API_KEY env var).
units	Unit system ("metric" or "imperial").
n_tries	Number of retry attempts for API/LLM calls (default: 3).
backoff	Base seconds to wait between retries (default: 2).
endpoint_url	OpenWeatherMap endpoint URL.
verbose	Logical controlling progress messages (default: TRUE).

Value

A list containing:

- success - Logical indicating if operation succeeded
- location - Cleaned location string
- weather_raw - Raw API response
- weather_formatted - Formatted weather string
- llm_response - Generated weather report
- timestamp - Time of response
- cache_hit - Logical indicating cache usage
- attempts - Number of tries made

Examples

```
## Not run:
# Using environment variable
Sys.setenv(OPENWEATHERMAP_API_KEY = "your_key")
report <- build_weather_agent(
  llm = call_llm,
  location_query = "Paris, FR"
)

# With explicit API key
report <- build_weather_agent(
  llm = call_llm,
  location_query = "New York",
  weather_api_key = "your_key",
  verbose = FALSE
)

## End(Not run)
```

`check_forecasting_dependencies` *Check Forecasting Dependencies*

Description

This function ensures that all suggested packages and specific functions used in forecasting workflows are available at runtime. It assigns each function to a local variable using ‘fun <- get_suggested(pkg, fun)‘ style for runtime access.

Usage

```
check_forecasting_dependencies()
```

Value

Invisibly TRUE if all packages/functions are available or skipped (base).

`get_suggested` *Require Suggested Package or Retrieve Function at Runtime*

Description

This utility ensures a package listed in 'Suggests:' is available, and optionally returns a function from it. If no function is provided, it just checks the package presence (like a safe 'requireNamespace()').

Usage

```
get_suggested(pkg, fun = NULL)
```

Arguments

`pkg`	Character string. Name of the package.
`fun`	Optional character string. Name of the function to retrieve from the package.

Details

Base R packages are automatically considered available.

Value

If `fun` is provided, returns the function object from the package namespace. Otherwise, invisibly returns TRUE if the package is available.

`parse_and_validate_location`
Clean and Validate a Location Query

Description

Performs basic text cleaning on a free-form weather/location query, strips out common words like "weather", "forecast", "in", "for", and "please", then enforces minimal length and alphabetic-only constraints. If no country code is present, it also suggests the user include one.

Usage

```
parse_and_validate_location(query)
```

Arguments

query	Character. A free-text user request (e.g. ‘"weather in Paris"‘).

Value

A cleaned location string, suitable for passing to the OpenWeatherMap API (e.g. ‘"Paris, FR"‘). Raises an error if the cleaned location is too short or contains digits.

Examples

```
parse_and_validate_location("weather in New York")
## Not run:
parse_and_validate_location("12345")  # error

## End(Not run)
```

---

state_graph *State Graph Framework*

Description

A lightweight framework for building state-driven workflows using nodes, edges, and conditionally branching logic. Each node is a function that takes and mutates a shared ‘state‘ list.

Usage

```
make_node(func, name = NULL)

make_edge(from, to, condition = NULL, label = NULL)

StateGraph()

interrupt(value)

make_command(goto = NULL, update = list())
```

Arguments

func	A function that takes a ‘state‘ list and returns updates.
name	An optional name for the node.
from	The source node name.
to	The destination node name.
condition	An optional function that returns a label based on the ‘state‘.
label	Optional label for conditional branching.
value	A string to print before prompting the user.
goto	Name of the next node to transition to.
update	A list of key-value updates to the ‘state‘.

Value

A list containing the function and optional name.

A list representing the edge.

A list of graph methods: `add_node`, `add_edge`, `add_conditional_edges`, `set_entry_point`, `compile`, `END_NODE_NAME`

A character response entered by the user.

A list with ‘goto‘ and ‘update‘ keys.

Functions

- `make_node()`: Create a Graph Node
- `make_edge()`: Create a Graph Edge
- `StateGraph()`: Build a State Graph Execution Engine
- `interrupt()`: Pause Execution for User Input
- `make_command()`: Create a Graph Command Result

Functions

`make_node()` Wraps a function in a graph-compatible node.

`make_edge()` Creates a transition between nodes, optionally conditional.

`StateGraph()` Builds and compiles the state graph engine.

`interrupt()` Prompts the user for input during execution.

`make_command()` Signals the next node and state updates.

Examples

```
graph <- StateGraph()
graph$add_node("start", function(state) list(goto = "end"))
graph$add_node("end", function(state) list())
graph$set_entry_point("start")
agent <- graph$compile()
state <- list()
agent(state)
```

Index