Package ‘aisdk’

March 31, 2026

Title Unified Interface for AI Model Providers

Version 1.1.0

Description A production-grade AI toolkit for R featuring a layered architecture (Specification, Utilities, Providers, Core), request interception support, robust error handling with exponential retry delays, support for multiple AI model providers ('OpenAI', 'Anthropic', etc.), local small language model inference, distributed 'MCP' ecosystem, multi-agent orchestration, progressive knowledge loading through skills, and a global skill store for sharing AI capabilities.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Imports R6, httr2, jsonlite, rlang, yaml, callr, processx, memoise, digest, parallel, stats, tools, utils, base64enc, curl, openssl

Suggests testthat (>= 3.0.0), httptest2, cli, knitr, rmarkdown, skimr, evaluate, shiny, shinyjs, bslib, commonmark, ggplot2, dplyr, readr, readxl, DBI, RSQLite, torch, onnx, rstudioapi, DT, withr, httpuv, devtools, fs, dotenv, quarto, pdftools

Config/testthat/edition 3

URL `https://github.com/YuLab-SMU/aisdk`, `https://yulab-smu.top/aisdk/`

BugReports `https://github.com/YuLab-SMU/aisdk/issues`

NeedsCompilation no

Author Yonghe Xia [aut, cre]

Maintainer Yonghe Xia <`xiayh17@gmail.com`>

Depends R (>= 4.1.0)

Repository CRAN

Date/Publication 2026-03-31 15:00:07 UTC

Contents

`aisdk-package` *aisdk: AI SDK for R*

Description

A production-grade AI SDK for R featuring a layered architecture, middleware support, robust error handling, and support for multiple AI model providers.

Architecture

The SDK uses a 4-layer architecture:

- **Specification Layer**: Abstract interfaces (LanguageModelV1, EmbeddingModelV1)
- **Utilities Layer**: Shared tools (HTTP, retry, registry, middleware)
- **Provider Layer**: Concrete implementations (OpenAIProvider, etc.)
- **Core Layer**: High-level API (generate_text, stream_text, embed)

Quick Start

```
library(aisdk)

# Create an OpenAI provider
openai <- create_openai()

# Generate text
result <- generate_text(
  model = openai$language_model("gpt-4o"),
  prompt = "Explain R in one sentence."
)
print(result$text)

# Or use the registry for cleaner syntax
get_default_registry()$register("openai", openai)
result <- generate_text("openai:gpt-4o", "Hello!")
```

Author(s)

Maintainer: Yonghe Xia <xiayh17@gmail.com>

See Also

Useful links:

- https://github.com/YuLab-SMU/aisdk
- https://yulab-smu.top/aisdk/
- Report bugs at https://github.com/YuLab-SMU/aisdk/issues

Agent *Agent Class*

Description

R6 class representing an AI agent. Agents are the worker units in the multi-agent architecture. Each agent has a name, description (for semantic routing), system prompt (persona), and a set of tools it can use.

Key design principle: Agents are stateless regarding conversation history. The ChatSession holds the shared state (history, memory, environment).

Public fields

`name` Unique identifier for this agent.

`description` Description of the agent's capability. This is the "API" that the LLM Manager uses for semantic routing.

`system_prompt` The agent's persona/instructions.

`tools` List of Tool objects this agent can use.

`model` Default model ID for this agent.

Methods

Public methods:

- `Agent$new()`
- `Agent$run()`
- `Agent$stream()`
- `Agent$as_tool()`
- `Agent$create_session()`
- `Agent$print()`
- `Agent$clone()`

Method `new()`**:** Initialize a new Agent.

Usage:

```
Agent$new(
  name,
  description,
  system_prompt = NULL,
  tools = NULL,
  skills = NULL,
  model = NULL
)
```

Arguments:

`name` Unique name for this agent (e.g., "DataCleaner", "Visualizer").

`description` A clear description of what this agent does. This is used by the Manager LLM to decide which agent to delegate to.

`system_prompt` Optional system prompt defining the agent's persona.

`tools` Optional list of Tool objects the agent can use.

`skills` Optional character vector of skill paths or "auto" to discover skills. When provided, this automatically loads skills, creates tools, and updates the system prompt.

`model` Optional default model ID for this agent.

Returns: An Agent object.

Method `run()`**:** Run the agent with a given task.

Usage:

```
Agent$run(
  task,
  session = NULL,
  context = NULL,
  model = NULL,
  max_steps = 10,
  ...
)
```

Arguments:

`task` The task instruction (natural language).

`session` Optional ChatSession for shared state. If NULL, a temporary session is created.

`context` Optional additional context to inject (e.g., from parent agent).

`model` Optional model override. Uses session's model if not provided.

`max_steps` Maximum ReAct loop iterations. Default 10.

`...` Additional arguments passed to generate_text.

Returns: A GenerateResult object from generate_text.

Method `stream()`**:** Run the agent with streaming output.

Usage:

```
Agent$stream(
  task,
  callback = NULL,
  session = NULL,
  context = NULL,
  model = NULL,
  max_steps = 10,
  ...
)
```

Arguments:

`task` The task instruction (natural language).

`callback` Function to handle streaming chunks: callback(text, done).

`session` Optional ChatSession for shared state.

`context` Optional additional context to inject.

`model` Optional model override.

`max_steps` Maximum ReAct loop iterations. Default 10.

`...` Additional arguments passed to stream_text.

Returns: A GenerateResult object (accumulated).

Method `as_tool()`**:** Convert this agent to a Tool.

Usage:

```
Agent$as_tool()
```

Details: This allows the agent to be used as a delegate target by a Manager agent. The tool wraps the agent's run() method and uses the agent's description for semantic routing.

Returns: A Tool object that wraps this agent.

Method `create_session()`**:** Create a stateful ChatSession from this agent.

Usage:

```
Agent$create_session(model = NULL, ...)
```

Arguments:

`model` Optional model override.

`...` Additional arguments passed to ChatSession$new.

Returns: A ChatSession object initialized with this agent's config.

Method `print()`**:** Print method for Agent.

Usage:

```
Agent$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`AgentRegistry` *AgentRegistry Class*

Description

R6 class for managing a collection of Agent objects. Provides storage, lookup, and automatic delegation tool generation for multi-agent systems.

Methods

Public methods:

- `AgentRegistry$new()`
- `AgentRegistry$register()`
- `AgentRegistry$get()`
- `AgentRegistry$has()`
- `AgentRegistry$list_agents()`
- `AgentRegistry$get_all()`
- `AgentRegistry$unregister()`
- `AgentRegistry$generate_delegate_tools()`
- `AgentRegistry$generate_prompt_section()`
- `AgentRegistry$print()`
- `AgentRegistry$clone()`

Method `new()`**:** Initialize a new AgentRegistry.

Usage:

```
AgentRegistry$new(agents = NULL)
```

Arguments:

`agents` Optional list of Agent objects to register immediately.

Method `register()`**:** Register an agent.

Usage:

```
AgentRegistry$register(agent)
```

Arguments:

`agent` An Agent object to register.

Returns: Invisible self for chaining.

Method `get()`**:** Get an agent by name.

Usage:

```
AgentRegistry$get(name)
```

Arguments:

`name` The agent name.

Returns: The Agent object, or NULL if not found.

Method `has()`**:** Check if an agent is registered.

Usage:

```
AgentRegistry$has(name)
```

Arguments:

`name` The agent name.

Returns: TRUE if registered, FALSE otherwise.

Method `list_agents()`**:** List all registered agent names.

Usage:

```
AgentRegistry$list_agents()
```

Returns: Character vector of agent names.

Method `get_all()`**:** Get all registered agents.

Usage:

```
AgentRegistry$get_all()
```

Returns: List of Agent objects.

Method `unregister()`**:** Unregister an agent.

Usage:

```
AgentRegistry$unregister(name)
```

Arguments:

`name` The agent name to remove.

Returns: Invisible self for chaining.

Method `generate_delegate_tools()`**:** Generate delegation tools for all registered agents.

Usage:

```
AgentRegistry$generate_delegate_tools(
  flow = NULL,
  session = NULL,
  model = NULL
)
```

Arguments:

`flow` Optional Flow object for context-aware execution.

`session` Optional ChatSession for shared state.

`model` Optional model ID for agent execution.

Details: Creates a list of Tool objects that wrap each agent's run() method. These tools can be given to a Manager agent for semantic routing.

Returns: A list of Tool objects.

Method `generate_prompt_section()`**:** Generate a prompt section describing available agents.

Usage:

```
AgentRegistry$generate_prompt_section()
```

Details: Creates a formatted string listing all agents and their descriptions. Useful for injecting into a Manager's system prompt.

Returns: A character string.

Method `print()`**:** Print method for AgentRegistry.

Usage:

```
AgentRegistry$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`AgentTeam` *AgentTeam Class*

Description

R6 class representing a team of agents.

Public fields

`name` Name of the team.

`members` List of registered agents (workers).

`manager` The manager agent (created automatically).

`default_model` Default model ID for the team (optional).

`session` Optional shared ChatSession for the team.

Methods

Public methods:

- `AgentTeam$new()`
- `AgentTeam$register_agent()`
- `AgentTeam$run()`
- `AgentTeam$print()`
- `AgentTeam$clone()`

Method `new()`**:** Initialize a new AgentTeam.

Usage:

```
AgentTeam$new(name = "AgentTeam", model = NULL, session = NULL)
```

Arguments:

`name` Name of the team.

`model` Optional default model for the team.

`session` Optional shared ChatSession (or SharedSession).

Returns: A new AgentTeam object.

Method `register_agent()`**:** Register an agent to the team.

Usage:

```
AgentTeam$register_agent(
  name,
  description,
  skills = NULL,
  tools = NULL,
  system_prompt = NULL,
  model = NULL
)
```

Arguments:

`name` Name of the agent.

`description` Description of the agent's capabilities.

`skills` Character vector of skills to load for this agent.

`tools` List of explicit Tool objects.

`system_prompt` Optional system prompt override.

`model` Optional default model for this agent (overrides team default).

Returns: Self (for chaining).

Method `run()`**:** Run the team on a task.

Usage:

```
AgentTeam$run(task, model = NULL, session = NULL)
```

Arguments:

`task` The task instruction.

`model` Model ID to use for the Manager.

`session` Optional shared ChatSession (or SharedSession).

Returns: The result from the Manager agent.

Method `print()`**:** Print team info.

Usage:

```
AgentTeam$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`agent_evals` *Performance & Benchmarking: Agent Evals*

Description

Testing infrastructure for LLM-powered code. Provides testthat integration with custom expectations for evaluating AI agent performance, tool accuracy, and hallucination rates.

`agent_library` *Agent Library: Built-in Agent Specialists*

Description

Factory functions for creating standard library agents for common tasks. These agents are preconfigured with appropriate system prompts and tools for their respective specializations.

`agent_registry` *Agent Registry: Agent Storage and Lookup*

Description

AgentRegistry R6 class for storing and retrieving Agent instances. Used by the Flow system for agent delegation.

aiChatServer AI Chat Server

Description

Shiny module server for AI-powered chat, featuring non-blocking streaming via background processes and tool execution bridge.

Usage

```
aiChatServer(
  id,
  model,
  tools = NULL,
  context = NULL,
  system = NULL,
  debug = FALSE,
  on_message_complete = NULL
)
```

Arguments

`id`	The namespace ID for the module.
`model`	Either a LanguageModelV1 object, or a string ID like "openai:gpt-4o".
`tools`	Optional list of Tool objects for function calling.
`context`	Optional reactive expression that returns context data to inject into the system prompt. This is read with `isolate()` to avoid reactive loops.
`system`	Optional system prompt.
`debug`	Reactive expression or logical. If TRUE, shows raw debug output in UI.
`on_message_complete`	Optional callback function called when a message is complete. Takes one argument: the complete assistant message text.

Value

A reactive value containing the chat history.

aiChatUI *AI Chat UI*

Description

Creates a modern, streaming-ready chat interface for Shiny applications.

Usage

```
aiChatUI(id, height = "500px")
```

Arguments

`id`	The namespace ID for the module.
`height`	Height of the chat window (e.g. "400px").

Value

A Shiny UI definition.

AiHubMixProvider *AiHubMix Provider Class*

Description

Provider class for AiHubMix.

Super class

`aisdk::OpenAIProvider` -> `AiHubMixProvider`

Methods

Public methods:

- `AiHubMixProvider$new()`
- `AiHubMixProvider$language_model()`
- `AiHubMixProvider$clone()`

Method `new()`**:** Initialize the AiHubMix provider.

Usage:

```
AiHubMixProvider$new(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments:

`api_key` AiHubMix API key. Defaults to AIHUBMIX_API_KEY env var.

`base_url` Base URL. Defaults to https://aihubmix.com/v1.

`headers` Optional additional headers.

Method `language_model()`**:** Create a language model.

Usage:

```
AiHubMixProvider$language_model(model_id = NULL)
```

Arguments:

`model_id` The model ID (e.g., "claude-sonnet-3-5", "claude-opus-3", "gpt-4o").

Returns: An AiHubMixLanguageModel object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`analyze_r_package` *Analyze R Package for Skill Creation*

Description

Introspects an installed R package to understand its capabilities, exported functions, and documentation. This is used by the Skill Architect to "learn" a package.

Usage

```
analyze_r_package(package)
```

Arguments

`package`	Name of the package to analyze.

Value

A string summary of the package.

`AnthropicProvider` *Anthropic Provider Class*

Description

Provider class for Anthropic. Can create language models.

Public fields

`specification_version` Provider spec version.

Methods

Public methods:

- `AnthropicProvider$new()`
- `AnthropicProvider$enable_caching()`
- `AnthropicProvider$language_model()`
- `AnthropicProvider$clone()`

Method `new()`**:** Initialize the Anthropic provider.

Usage:

```
AnthropicProvider$new(
  api_key = NULL,
  base_url = NULL,
  api_version = NULL,
  headers = NULL,
  name = NULL
)
```

Arguments:

`api_key` Anthropic API key. Defaults to ANTHROPIC_API_KEY env var.

`base_url` Base URL for API calls. Defaults to https://api.anthropic.com/v1.

`api_version` Anthropic API version header. Defaults to "2023-06-01".

`headers` Optional additional headers.

`name` Optional provider name override.

Method `enable_caching()`**:** Enable or disable prompt caching.

Usage:

```
AnthropicProvider$enable_caching(enable = TRUE)
```

Arguments:

`enable` Logical.

Method `language_model()`**:** Create a language model.

Usage:

```
AnthropicProvider$language_model(model_id = "claude-sonnet-4-20250514")
```

Arguments:

`model_id` The model ID (e.g., "claude-sonnet-4-20250514", "claude-3-5-sonnet-20241022").

Returns: An AnthropicLanguageModel object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`apiConfigServer` *API Configuration Server*

Description

Server logic for the API Configuration UI.

Usage

```
apiConfigServer(id)
```

Arguments

`id`	The namespace ID for the module.

`apiConfigUI` *API Configuration UI*

Description

Creates a Shiny UI for configuring API providers.

Usage

```
apiConfigUI(id)
```

Arguments

`id`	The namespace ID for the module.

Value

A Shiny UI definition.

`api_diagnostics` *API Diagnostics*

Description

Provides diagnostic tools to test internet connectivity, DNS resolution, and API reachability.

`auth_hook` *Human-in-the-Loop Authorization*

Description

Provides dynamic authorization hooks to pause Agent execution and request user permission for elevated risk operations.

`auto_fix` *Autonomous Data Science Pipelines*

Description

Self-healing runtime for R code execution. Implements a "Hypothesis-Fix-Verify" loop that feeds error messages, stack traces, and context back to an LLM for automatic error correction.

Execute R code with automatic error recovery using LLM assistance. When code fails, the error is analyzed and a fix is attempted automatically.

Usage

```
auto_fix(
  expr,
  model = NULL,
  max_attempts = 3,
  context = NULL,
  verbose = TRUE,
  memory = NULL
)
```

Arguments

`expr`	The R expression to execute.
`model`	The LLM model to use for error analysis (default: from options).
`max_attempts`	Maximum number of fix attempts (default: 3).
`context`	Optional additional context about the code’s purpose.
`verbose`	Print progress messages (default: TRUE).
`memory`	Optional ProjectMemory object for learning from past fixes.

Value

The result of successful execution, or an error if all attempts fail.

Examples

```
## Not run:
# Simple usage - auto-fix a data transformation
result <- auto_fix({
  df <- read.csv("data.csv")
  df %>%
    filter(value > 100) %>%
    summarize(mean = mean(value))
})

# With context for better error understanding
result <- auto_fix(
  expr = {
    model <- lm(y ~ x, data = df)
  },
  context = "Fitting a linear regression model to predict sales"
)

## End(Not run)
```

BailianProvider *Bailian Provider Class*

Description

Provider class for Alibaba Cloud Bailian / DashScope platform.

Super class

`aisdk::OpenAIProvider` -> `BailianProvider`

Methods

Public methods:

- `BailianProvider$new()`
- `BailianProvider$language_model()`
- `BailianProvider$clone()`

Method `new()`**:** Initialize the Bailian provider.

Usage:

```
BailianProvider$new(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments:

`api_key` DashScope API key. Defaults to DASHSCOPE_API_KEY env var.

`base_url` Base URL. Defaults to https://dashscope.aliyuncs.com/compatible-mode/v1.

`headers` Optional additional headers.

Method `language_model():` Create a language model.

Usage:

```
BailianProvider$language_model(model_id = NULL)
```

Arguments:

`model_id` The model ID (e.g., "qwen-plus", "qwen-turbo", "qwq-32b").

Returns: A BailianLanguageModel object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`benchmark_agent` *Benchmark Agent*

Description

Run a benchmark suite against an agent and collect performance metrics.

Usage

```
benchmark_agent(agent, tasks, tools = NULL, verbose = TRUE)
```

Arguments

`agent`	An Agent object or model string.
`tasks`	A list of benchmark tasks (see details).
`tools`	Optional list of tools for the agent.
`verbose`	Print progress.

Details

Each task in the tasks list should have:

- prompt: The task prompt
- expected: Expected output or criteria
- category: Optional category for grouping
- ground_truth: Optional ground truth for hallucination checking

Value

A benchmark result object with metrics.

`cache` *Caching System*

Description

Utilities for caching tool execution results and other expensive operations.

`cache_tool` *Cache Tool*

Description

Wrap a tool with caching capabilities using the `memoise` package.

Usage

```
cache_tool(tool, cache = NULL)
```

Arguments

`tool`	The Tool object to cache.
`cache`	An optional memoise cache configuration (e.g., cache_memory() or cache_filesystem()). Defaults to `memoise::cache_memory()`.

Value

A new Tool object that caches its execution.

`ChannelAdapter` *Channel Adapter*

Description

Base class for transport adapters that translate external messaging events into normalized `aisdk` channel events.

Public fields

`id` Unique channel identifier.

`config` Adapter configuration.

Methods

Public methods:

- `ChannelAdapter$new()`
- `ChannelAdapter$parse_request()`
- `ChannelAdapter$resolve_session_key()`
- `ChannelAdapter$format_inbound_message()`
- `ChannelAdapter$prepare_inbound_message()`
- `ChannelAdapter$send_text()`
- `ChannelAdapter$send_status()`
- `ChannelAdapter$send_attachment()`
- `ChannelAdapter$clone()`

Method `new()`**:** Initialize a channel adapter.

Usage:

```
ChannelAdapter$new(id, config = list())
```

Arguments:

`id` Channel identifier.

`config` Adapter configuration list.

Method `parse_request()`**:** Parse a raw channel request.

Usage:

```
ChannelAdapter$parse_request(headers = NULL, body = NULL, ...)
```

Arguments:

`headers` Request headers as a named list.

`body` Raw body as JSON string or parsed list.

`...` Optional transport-specific values.

Returns: A normalized parse result list.

Method `resolve_session_key()`**:** Resolve a stable session key for an inbound message.

Usage:

```
ChannelAdapter$resolve_session_key(message, policy = list())
```

Arguments:

`message` Normalized inbound message list.

`policy` Session policy list.

Returns: Character scalar session key.

Method `format_inbound_message()`**:** Format an inbound prompt for a `ChatSession`.

Usage:

```
ChannelAdapter$format_inbound_message(message)
```

Arguments:

`message` Normalized inbound message list.

Returns: Character scalar prompt.

Method `prepare_inbound_message()`: Prepare an inbound message using session state.

Usage:

```
ChannelAdapter$prepare_inbound_message(session, message)
```

Arguments:

`session` Current `ChatSession`.

`message` Normalized inbound message list.

Returns: Possibly enriched inbound message list.

Method `send_text()`: Send a final text reply back to the channel.

Usage:

```
ChannelAdapter$send_text(message, text, ...)
```

Arguments:

`message` Original normalized inbound message.

`text` Final outbound text.

`...` Optional adapter-specific values.

Returns: Transport-specific response.

Method `send_status()`: Optionally send an intermediate status message.

Usage:

```
ChannelAdapter$send_status(
  message,
  status = c("thinking", "working", "error"),
  text = NULL,
  ...
)
```

Arguments:

`message` Original normalized inbound message.

`status` Status name such as "thinking", "working", or "error".

`text` Optional status text override.

`...` Optional adapter-specific values.

Returns: Adapter-specific status result, or NULL if unsupported.

Method `send_attachment()`: Optionally send a generated local attachment.

Usage:

```
ChannelAdapter$send_attachment(message, path, ...)
```

Arguments:

`message` Original normalized inbound message.

`path` Absolute local file path.

`...` Optional adapter-specific values.

Returns: Adapter-specific attachment result, or NULL if unsupported.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`ChannelRuntime` *Channel Runtime*

Description

Coordinates channel adapters, durable session state, and `ChatSession` execution for external messaging integrations.

Public fields

`session_store` Durable store for channel sessions.

Methods

Public methods:

- `ChannelRuntime$new()`
- `ChannelRuntime$register_adapter()`
- `ChannelRuntime$get_adapter()`
- `ChannelRuntime$handle_request()`
- `ChannelRuntime$process_message()`
- `ChannelRuntime$create_child_session()`
- `ChannelRuntime$clone()`

Method `new()`**:** Initialize a channel runtime.

Usage:

```
ChannelRuntime$new(
  session_store,
  model = NULL,
  agent = NULL,
  tools = NULL,
  hooks = NULL,
  registry = NULL,
  max_steps = 10,
  session_policy = channel_default_session_policy()
)
```

Arguments:

`session_store` File-backed session store.

`model` Optional default model id.

`agent` Optional default agent.
`tools` Optional default tools.
`hooks` Optional session hooks.
`registry` Optional provider registry.
`max_steps` Maximum tool execution steps.
`session_policy` Session routing policy list.

Method `register_adapter()`: Register a channel adapter.

Usage:

```
ChannelRuntime$register_adapter(adapter)
```

Arguments:

`adapter` Channel adapter instance.

Returns: Invisible self.

Method `get_adapter()`: Get a channel adapter.

Usage:

```
ChannelRuntime$get_adapter(channel_id)
```

Arguments:

`channel_id` Adapter identifier.

Returns: Adapter instance.

Method `handle_request()`: Handle a raw channel request.

Usage:

```
ChannelRuntime$handle_request(channel_id, headers = NULL, body = NULL, ...)
```

Arguments:

`channel_id` Adapter identifier.
`headers` Request headers.
`body` Raw or parsed body.
`...` Optional adapter-specific values.

Returns: A normalized runtime response.

Method `process_message()`: Process one normalized inbound message.

Usage:

```
ChannelRuntime$process_message(channel_id, message)
```

Arguments:

`channel_id` Adapter identifier.
`message` Normalized inbound message.

Returns: Processing result list.

Method `create_child_session()`: Create a child session linked to a parent session.

Usage:

```
ChannelRuntime$create_child_session(
  parent_session_key,
  child_session_key = NULL,
  inherit_history = TRUE,
  metadata = NULL
)
```

Arguments:

`parent_session_key`	Parent session key.
`child_session_key`	Optional child key. Generated if omitted.
`inherit_history`	Whether to copy parent state into the child.
`metadata`	Optional metadata to merge into the child session.

Returns: The child session key.

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

Usage:

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

Arguments:

`deep`	Whether to make a deep clone.

`ChannelSessionStore` *Channel Session Store*

Description

Abstract persistence seam for channel-driven sessions.

Methods

Public methods:

- `ChannelSessionStore$load_session()`
- `ChannelSessionStore$save_session()`
- `ChannelSessionStore$update_record()`
- `ChannelSessionStore$get_record()`
- `ChannelSessionStore$list_sessions()`
- `ChannelSessionStore$has_processed_event()`
- `ChannelSessionStore$mark_processed_event()`
- `ChannelSessionStore$link_child_session()`
- `ChannelSessionStore$clone()`

Method `load_session()`**:** Load a persisted `ChatSession`.

Usage:

```
ChannelSessionStore$load_session(
  session_key,
  tools = NULL,
  hooks = NULL,
  registry = NULL
)
```

Arguments:

`session_key` Session key.

`tools` Optional tools to reattach.

`hooks` Optional hooks to reattach.

`registry` Optional provider registry.

Returns: A `ChatSession` or NULL.

Method `save_session()`**:** Save a `ChatSession` and update store metadata.

Usage:

```
ChannelSessionStore$save_session(session_key, session, record = NULL)
```

Arguments:

`session_key` Session key.

`session` `ChatSession` instance.

`record` Optional record fields to merge.

Returns: Store-specific save result.

Method `update_record()`**:** Update a store record without persisting a session file.

Usage:

```
ChannelSessionStore$update_record(session_key, record)
```

Arguments:

`session_key` Session key.

`record` Record fields to merge.

Returns: Store-specific update result.

Method `get_record()`**:** Retrieve a single session record.

Usage:

```
ChannelSessionStore$get_record(session_key)
```

Arguments:

`session_key` Session key.

Returns: Store-specific record object.

Method `list_sessions()`**:** List all session records.

Usage:

```
ChannelSessionStore$list_sessions()
```

Returns: Store-specific collection of session records.

Method `has_processed_event()`**:** Check whether an event id has already been processed.

Usage:

```
ChannelSessionStore$has_processed_event(channel_id, event_id)
```

Arguments:

`channel_id` Channel identifier.

`event_id` Event identifier.

Returns: Logical scalar.

Method `mark_processed_event()`**:** Mark an event id as processed.

Usage:

```
ChannelSessionStore$mark_processed_event(channel_id, event_id, payload = NULL)
```

Arguments:

`channel_id` Channel identifier.

`event_id` Event identifier.

`payload` Optional event payload to keep in the dedupe index.

Returns: Store-specific event record.

Method `link_child_session()`**:** Register a child session relationship.

Usage:

```
ChannelSessionStore$link_child_session(parent_session_key, child_session_key)
```

Arguments:

`parent_session_key` Parent session key.

`child_session_key` Child session key.

Returns: Store-specific link result.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`channel_documents` *Channel Document Ingest*

Description

Helpers for extracting, chunking, and summarizing inbound document attachments before they are injected into chat context.

`channel_feishu` *Feishu Channel Adapter*

Description

Feishu adapter built on top of the generic channel runtime seam. Phase 1 focuses on text events and final text replies.

`channel_runtime` *Channel Runtime*

Description

Runtime orchestration layer for driving `ChatSession` objects from external messaging channels.

`channel_session_store` *Channel Session Store*

Description

Durable local storage for channel-driven chat sessions and their routing metadata.

`channel_types` *Channel Integration Types*

Description

Low-level types and seams for external messaging channels. These abstractions sit above providers and below UI surfaces.

`ChatSession` *ChatSession Class*

Description

R6 class representing a stateful chat session. Automatically manages conversation history, supports tool execution, and provides persistence.

Methods

Public methods:

- `ChatSession$new()`
- `ChatSession$send()`
- `ChatSession$send_stream()`
- `ChatSession$append_message()`
- `ChatSession$get_history()`
- `ChatSession$get_last_response()`
- `ChatSession$clear_history()`
- `ChatSession$switch_model()`
- `ChatSession$get_model_id()`
- `ChatSession$stats()`
- `ChatSession$save()`
- `ChatSession$as_list()`
- `ChatSession$restore()`
- `ChatSession$restore_from_list()`
- `ChatSession$print()`
- `ChatSession$get_memory()`
- `ChatSession$set_memory()`
- `ChatSession$list_memory()`
- `ChatSession$get_metadata()`
- `ChatSession$set_metadata()`
- `ChatSession$merge_metadata()`
- `ChatSession$list_metadata()`
- `ChatSession$clear_memory()`
- `ChatSession$get_envir()`
- `ChatSession$eval_in_session()`
- `ChatSession$list_envir()`
- `ChatSession$checkpoint()`
- `ChatSession$restore_checkpoint()`
- `ChatSession$clone()`

Method `new()`**:** Initialize a new ChatSession.

Usage:

```
ChatSession$new(
  model = NULL,
  system_prompt = NULL,
  tools = NULL,
  hooks = NULL,
  history = NULL,
  max_steps = 10,
  registry = NULL,
  memory = NULL,
  metadata = NULL,
  envir = NULL,
  agent = NULL
)
```

Arguments:

`model` A LanguageModelV1 object or model string ID (e.g., "openai:gpt-4o").

`system_prompt` Optional system prompt for the conversation.

`tools` Optional list of Tool objects for function calling.

`hooks` Optional HookHandler object for event hooks.

`history` Optional initial message history (list of message objects).

`max_steps` Maximum steps for tool execution loops. Default 10.

`registry` Optional ProviderRegistry for model resolution.

`memory` Optional initial shared memory (list). For multi-agent state sharing.

`metadata` Optional session metadata (list). Used for channel/runtime state.

`envir` Optional shared R environment. For multi-agent data sharing.

`agent` Optional Agent object. If provided, the session inherits the agent's tools and system prompt.

Method `send()`**:** Send a message and get a response.

Usage:

```
ChatSession$send(prompt, ...)
```

Arguments:

`prompt` The user message to send.

`...` Additional arguments passed to generate_text.

Returns: The GenerateResult object from the model.

Method `send_stream()`**:** Send a message with streaming output.

Usage:

```
ChatSession$send_stream(prompt, callback, ...)
```

Arguments:

`prompt` The user message to send.

`callback` Function called for each chunk: callback(text, done).

`...` Additional arguments passed to stream_text.

Returns: Invisible NULL (output is via callback).

Method `append_message()`**:** Append a message to the history.

Usage:

```
ChatSession$append_message(role, content, reasoning = NULL)
```

Arguments:

`role` Message role: "user", "assistant", "system", or "tool".

`content` Message content.

`reasoning` Optional reasoning text to attach to the message.

Method `get_history()`**:** Get the conversation history.

Usage:

```
ChatSession$get_history()
```

Returns: A list of message objects.

Method `get_last_response()`**:** Get the last response from the assistant.

Usage:

```
ChatSession$get_last_response()
```

Returns: The content of the last assistant message, or NULL.

Method `clear_history()`**:** Clear the conversation history.

Usage:

```
ChatSession$clear_history(keep_system = TRUE)
```

Arguments:

`keep_system` If TRUE, keeps the system prompt. Default TRUE.

Method `switch_model()`**:** Switch to a different model.

Usage:

```
ChatSession$switch_model(model)
```

Arguments:

`model` A LanguageModelV1 object or model string ID.

Method `get_model_id()`**:** Get current model identifier.

Usage:

```
ChatSession$get_model_id()
```

Returns: Model ID string.

Method `stats()`**:** Get token usage statistics.

Usage:

```
ChatSession$stats()
```

Returns: A list with token counts and message stats.

Method `save()`**:** Save session to a file.

Usage:

```
ChatSession$save(path, format = NULL)
```

Arguments:

`path` File path (supports .rds or .json extension).
`format` Optional format override: "rds" or "json". Auto-detected from path.

Method `as_list()`**:** Export session state as a list (for serialization).

Usage:

```
ChatSession$as_list()
```

Returns: A list containing session state.

Method `restore()`**:** Restore session from a file.

Usage:

```
ChatSession$restore(path, format = NULL)
```

Arguments:

`path` File path (supports .rds or .json extension).
`format` Optional format override: "rds" or "json". Auto-detected from path.

Method `restore_from_list()`**:** Restore session state from a list.

Usage:

```
ChatSession$restore_from_list(data)
```

Arguments:

`data` A list exported by as_list().

Method `print()`**:** Print method for ChatSession.

Usage:

```
ChatSession$print()
```

Method `get_memory()`**:** Get a value from shared memory.

Usage:

```
ChatSession$get_memory(key, default = NULL)
```

Arguments:

`key` The key to retrieve.
`default` Default value if key not found. Default NULL.

Returns: The stored value or default.

Method `set_memory()`**:** Set a value in shared memory.

Usage:

```
ChatSession$set_memory(key, value)
```

Arguments:

`key` The key to store.
`value` The value to store.

Returns: Invisible self for chaining.

Method `list_memory()`**:** List all keys in shared memory.

Usage:

```
ChatSession$list_memory()
```

Returns: Character vector of memory keys.

Method `get_metadata()`**:** Get a value from session metadata.

Usage:

```
ChatSession$get_metadata(key, default = NULL)
```

Arguments:

`key` The metadata key to retrieve.
`default` Default value if key is not present.

Returns: The stored metadata value or default.

Method `set_metadata()`**:** Set a value in session metadata.

Usage:

```
ChatSession$set_metadata(key, value)
```

Arguments:

`key` The metadata key to set.
`value` The value to store.

Returns: Invisible self for chaining.

Method `merge_metadata()`**:** Merge a named list into session metadata.

Usage:

```
ChatSession$merge_metadata(values)
```

Arguments:

`values` Named list of metadata values.

Returns: Invisible self for chaining.

Method `list_metadata()`**:** List metadata keys.

Usage:

```
ChatSession$list_metadata()
```

Returns: Character vector of metadata keys.

Method `clear_memory()`**:** Clear shared memory.

Usage:

```
ChatSession$clear_memory(keys = NULL)
```

Arguments:

`keys` Optional specific keys to clear. If NULL, clears all.

Returns: Invisible self for chaining.

Method `get_envir()`**:** Get the shared R environment.

Usage:

```
ChatSession$get_envir()
```

Details: This environment is shared across all agents using this session. Agents can store and retrieve data frames, models, and other R objects.

Returns: An environment object.

Method `eval_in_session()`**:** Evaluate an expression in the session environment.

Usage:

```
ChatSession$eval_in_session(expr)
```

Arguments:

`expr` An expression to evaluate.

Returns: The result of the evaluation.

Method `list_envir()`**:** List objects in the session environment.

Usage:

```
ChatSession$list_envir()
```

Returns: Character vector of object names.

Method `checkpoint()`**:** Save a memory snapshot to a file (checkpoint for Mission resume).

Usage:

```
ChatSession$checkpoint(path = NULL)
```

Arguments:

`path` File path (.rds). If NULL, uses a temp file and returns the path.

Returns: Invisible file path.

Method `restore_checkpoint()`**:** Restore memory and history from a checkpoint file.

Usage:

```
ChatSession$restore_checkpoint(path)
```

Arguments:

`path` File path to a checkpoint created by checkpoint().

Returns: Invisible self for chaining.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`check_api` *Connect and Diagnose API Reachability*

Description

Tests connectivity to a specific LLM, provider, or URL. This is helpful for diagnosing network issues, DNS failures, or SSL problems.

Usage

```
check_api(model = NULL, url = NULL, registry = NULL)
```

Arguments

`model`	Optional. A `LanguageModelV1` object, or a provider name string (e.g., "openai", "gemini"). If provided, the base URL for that provider will be tested.
`url`	Optional. A specific URL to test.
`registry`	Optional ProviderRegistry to use if `model` is a character string.

Value

A list containing diagnostic results (invisible).

Examples

```
if (interactive()) {
    # Test by passing a URL directly
    check_api(url = "https://api.openai.com/v1")

    # Test a model directly
    model <- create_openai()$language_model("gpt-4o")
    check_api(model)
}
```

`check_ast_safety` *Check AST Safety*

Description

Analyze R code for unsafe function calls or operations before execution.

Usage

```
check_ast_safety(code_str)
```

Arguments

`code_str`	Character string containing R code.

Value

The parsed AST if safe. Throws an error if unsafe.

`check_sdk_compatibility` *Check SDK Version Compatibility*

Description

Check if code is compatible with the current SDK version and suggest migration steps if needed.

Usage

```
check_sdk_compatibility(code_version)
```

Arguments

`code_version`	Version string the code was written for.

Value

A list with compatible (logical) and suggestions (character vector).

Examples

```
if (interactive()) {
result <- check_sdk_compatibility("0.8.0")
if (!result$compatible) {
  cat("Migration needed:\n")
  cat(paste(result$suggestions, collapse = "\n"))
}
}
```

`clear_ai_session` *Clear AI Engine Session*

Description

Clears the cached session(s) for the AI engine. Useful for resetting state between documents.

Usage

```
clear_ai_session(session_name = NULL)
```

Arguments

`session_name`	Optional name of specific session to clear. If NULL, clears all.

Value

Invisible NULL.

`compat` *Compatibility Layer: Feature Flags and Migration Support*

Description

Provides feature flags, compatibility shims, and migration utilities for controlled breaking changes in the agent SDK.

`Computer` *Computer Class*

Description

R6 class providing computer abstraction with atomic tools for file operations, bash execution, and R code execution.

Public fields

`working_dir` Current working directory

`sandbox_mode` Sandbox mode: "strict", "permissive", or "none"

`execution_log` Log of executed commands

Methods

Public methods:

- `Computer$new()`
- `Computer$bash()`
- `Computer$read_file()`
- `Computer$write_file()`
- `Computer$execute_r_code()`
- `Computer$get_log()`
- `Computer$clear_log()`
- `Computer$clone()`

Method `new()`**:** Initialize computer abstraction

Usage:

```
Computer$new(working_dir = tempdir(), sandbox_mode = "permissive")
```

Arguments:

`working_dir` Working directory. Defaults to `tempdir()`.

`sandbox_mode` Sandbox mode: "strict", "permissive", or "none"

Method `bash()`**:** Execute bash command

Usage:

```
Computer$bash(command, timeout_ms = 30000, capture_output = TRUE)
```

Arguments:

`command` Bash command to execute

`timeout_ms` Timeout in milliseconds (default: 30000)

`capture_output` Whether to capture output (default: TRUE)

Returns: List with stdout, stderr, exit_code

Method `read_file()`**:** Read file contents

Usage:

```
Computer$read_file(path, encoding = "UTF-8")
```

Arguments:

`path` File path (relative to working_dir or absolute)

`encoding` File encoding (default: "UTF-8")

Returns: File contents as character string

Method `write_file()`**:** Write file contents

Usage:

```
Computer$write_file(path, content, encoding = "UTF-8")
```

Arguments:

`path` File path (relative to working_dir or absolute)

`content` Content to write

`encoding` File encoding (default: "UTF-8")

Returns: Success status

Method **`execute_r_code()`:** Execute R code

Usage:

```
Computer$execute_r_code(code, timeout_ms = 30000, capture_output = TRUE)
```

Arguments:

`code` R code to execute

`timeout_ms` Timeout in milliseconds (default: 30000)

`capture_output` Whether to capture output (default: TRUE)

Returns: List with result, output, error

Method **`get_log()`:** Get execution log

Usage:

```
Computer$get_log()
```

Returns: List of logged executions

Method **`clear_log()`:** Clear execution log Check bash command for sandbox violations Log execution Resolve path (relative to working_dir or absolute) Check write path for sandbox violations Check R code for sandbox violations

Usage:

```
Computer$clear_log()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`configure_api` *Launch API Configuration App*

Description

Launches a Shiny application to configure API providers and environment variables.

Usage

```
configure_api()
```

Value

A Shiny app object

`console` *Console Chat: Interactive REPL*

Description

Interactive terminal chat interface for ChatSession. Provides a REPL (Read-Eval-Print Loop) for conversing with LLMs. By default, enables an intelligent terminal agent that can execute commands, manage files, and run R code through natural language.

`console_agent` *Console Agent: Intelligent Terminal Assistant*

Description

Creates a default agent for console_chat() that enables natural language interaction with the terminal. Users can ask the agent to run commands, execute R code, read/write files, and more through conversational language.

`console_chat` *Start Console Chat*

Description

Launch an interactive chat session in the R console. Supports streaming output, slash commands, and colorful display using the cli package.

The console UI has three presentation modes:

- `clean`: compact default output with a stable status bar
- `inspect`: keeps the compact transcript but adds a per-turn tool timeline and an overlay-backed inspector
- `debug`: shows detailed tool logs and thinking output for troubleshooting

In agent mode, `console_chat()` can execute shell and R tools, summarize tool progress inline, and open an inspector overlay for the latest turn or a specific tool. The current implementation uses a shared frame builder for the status bar, tool timeline, and overlay surfaces, while preserving an append-only terminal fallback.

By default, the console operates in agent mode with tools for bash execution, file operations, R code execution, and more. Set `agent = NULL` for simple chat without tools.

Usage

```
console_chat(
  session = NULL,
  system_prompt = NULL,
  tools = NULL,
  hooks = NULL,
  stream = TRUE,
  verbose = FALSE,
  agent = "auto",
  working_dir = tempdir(),
  sandbox_mode = "permissive",
  show_thinking = verbose
)
```

Arguments

`session`	A ChatSession object, a LanguageModelV1 object, or a model string ID to create a new session.
`system_prompt`	Optional system prompt (merged with agent prompt if agent is used).
`tools`	Optional list of additional Tool objects.
`hooks`	Optional HookHandler object.
`stream`	Whether to use streaming output. Default TRUE.
`verbose`	Logical. If `TRUE`, show detailed tool calls, tool results, and thinking output. Defaults to `FALSE` for a cleaner console UI.
`agent`	Agent configuration. Options: • `"auto"` (default): Use the built-in console agent with terminal tools • `NULL`: Simple chat mode without tools • An Agent object: Use the provided custom agent
`working_dir`	Working directory for the console agent. Defaults to `tempdir()`.
`sandbox_mode`	Sandbox mode for the console agent: "strict", "permissive" (default), or "none".
`show_thinking`	Logical. Whether to show model thinking blocks when the provider exposes them. Defaults to `verbose`.

Value

The ChatSession object (invisibly) when chat ends.

Examples

```
if (interactive()) {
  # Start with default agent (intelligent terminal mode)
  console_chat("openai:gpt-4o")

  # Start in debug mode with full tool logs
  console_chat("openai:gpt-4o", verbose = TRUE)
  # Simple chat mode without tools
  console_chat("openai:gpt-4o", agent = NULL)

  # Start with an existing session
  chat <- create_chat_session("anthropic:claude-3-5-sonnet-latest")
  console_chat(chat)

  # Start with a custom agent
 agent <- create_agent("MathAgent", "Does math", system_prompt = "You are a math wizard.")
  console_chat("openai:gpt-4o", agent = agent)

  # Available commands in the chat:
  # /quit or /exit - End the chat
  # /save [path]   - Save session to file
  # /load [path]   - Load session from file
  # /model         - Open the provider/model chooser
  # /model [id]    - Switch to a different model
  # /model current - Show the active model
  # /history       - Show conversation history
  # /stats         - Show token usage statistics
  # /clear         - Clear conversation history
  # /stream [on|off] - Toggle streaming mode
  # /inspect [on|off] - Toggle inspect mode
  # /inspect turn - Open overlay for the latest turn
  # /inspect tool <index> - Open overlay for a tool in the latest turn
  # /inspect next - Move inspector overlay to the next tool
  # /inspect prev - Move inspector overlay to the previous tool
  # /inspect close - Close the active inspect overlay
  # /debug [on|off]  - Toggle detailed tool/thinking output
  # /local [on|off]- Toggle local execution mode (Global Environment)
  # /help          - Show available commands
  # /agent [on|off] - Toggle agent mode
}
```

`console_confirm` *Console Confirmation Prompt*

Description

Ask a yes/no question with numbered choices. Returns `TRUE` for yes, `FALSE` for no, or `NULL` if cancelled.

Usage

```
console_confirm(question)
```

Arguments

`question`	The question to display.

Value

TRUE if user selects Yes, FALSE for No, NULL if cancelled.

Examples

```
if (interactive()) {
  if (isTRUE(console_confirm("Overwrite existing file?"))) {
    message("Overwriting...")
  }
}
```

console_input   *Console Text Input*

Description

Prompt the user for free-text input with optional default value.

Usage

```
console_input(prompt, default = NULL)
```

Arguments

prompt	The prompt message to display.
default	Optional default value shown in brackets. Returned if user presses Enter without typing.

Value

The user's input string, default if empty input and default is set, or NULL if empty input with no default.

Examples

```
if (interactive()) {
  name <- console_input("Project name", default = "my-project")
  api_key <- console_input("API key")
}
```

`console_menu` *Console Interactive Menu*

Description

Present a numbered list of choices and return the user's selection. Styled with cli to match the console chat interface. Similar to `utils::menu()` but with cli formatting.

Usage

```
console_menu(title, choices)
```

Arguments

`title`	The question or prompt to display.
`choices`	Character vector of options to present.

Value

The index of the selected choice (integer), or `NULL` if cancelled (user enters 'q' or empty input).

Examples

```
if (interactive()) {
  selection <- console_menu("Which database?", c("PostgreSQL", "SQLite", "DuckDB"))
}
```

`content_image` *Create Image Content*

Description

Creates an image content object for a multimodal message. automatically handles URLs and local files (converted to base64).

Usage

```
content_image(image_path, media_type = "auto", detail = "auto")
```

Arguments

`image_path`	Path to a local file or a URL.
`media_type`	MIME type of the image (e.g., "image/jpeg", "image/png"). If NULL, attempts to guess from the file extension.
`detail`	Image detail suitable for some models (e.g., "auto", "low", "high").

Value

A list representing the image content in OpenAI-compatible format.

`content_text` *Create Text Content*

Description

Creates a text content object for a multimodal message.

Usage

```
content_text(text)
```

Arguments

`text`	The text string.

Value

A list representing the text content.

`context` *Context Management*

Description

Utilities for capturing and summarizing R objects for LLM context.

`core_api` *Core API: High-Level Functions*

Description

User-facing high-level API functions for interacting with AI models.

core_object *Core Object API: Structured Output Generation*

Description

Functions for generating structured objects from LLMs using schemas.

Generate a structured R object (list) from a language model based on a schema. The model is instructed to output valid JSON matching the schema, which is then parsed and returned as an R list.

Usage

```
generate_object(
  model = NULL,
  prompt,
  schema,
  schema_name = "result",
  system = NULL,
  temperature = 0.3,
  max_tokens = NULL,
  mode = c("json", "tool"),
  registry = NULL,
  ...
)
```

Arguments

`model`	Either a LanguageModelV1 object, or a string ID like "openai:gpt-4o".
`prompt`	A character string prompt describing what to generate.
`schema`	A schema object created by `z_object()`, `z_array()`, etc.
`schema_name`	Optional human-readable name for the schema (default: "result").
`system`	Optional system prompt.
`temperature`	Sampling temperature (0-2). Default 0.3 (lower for structured output).
`max_tokens`	Maximum tokens to generate.
`mode`	Output mode: "json" (prompt-based) or "tool" (function calling). Currently, only "json" mode is implemented.
`registry`	Optional ProviderRegistry to use (defaults to global registry).
`...`	Additional arguments passed to the model.

Value

A GenerateObjectResult with:

- object: The parsed R object (list)

- usage: Token usage information
- raw_text: The raw text output from the LLM
- finish_reason: The reason the generation stopped

Examples

```
if (interactive()) {
# Define a schema for the expected output
schema <- z_object(
  title = z_string(description = "Title of the article"),
  keywords = z_array(z_string()),
  sentiment = z_enum(c("positive", "negative", "neutral"))
)

# Generate structured object
result <- generate_object(
  model = "openai:gpt-4o",
  prompt = "Analyze this article: 'R programming is great for data science!'",
  schema = schema
)

print(result$object$title)
print(result$object$sentiment)
}
```

`create_agent` *Create an Agent*

Description

Factory function to create a new Agent object.

Usage

```
create_agent(
  name,
  description,
  system_prompt = NULL,
  tools = NULL,
  skills = NULL,
  model = NULL
)
```

Arguments

`name`	Unique name for this agent.
`description`	A clear description of what this agent does.
`system_prompt`	Optional system prompt defining the agent's persona.
`tools`	Optional list of Tool objects the agent can use.
`skills`	Optional character vector of skill paths or "auto".
`model`	Optional default model ID for this agent.

Value

An Agent object.

Examples

```
if (interactive()) {
  # Create a simple math agent
  math_agent <- create_agent(
    name = "MathAgent",
    description = "Performs arithmetic calculations",
    system_prompt = "You are a math assistant. Return only numerical results."
  )

  # Run the agent
  result <- math_agent$run("Calculate 2 + 2", model = "openai:gpt-4o")

  # Create an agent with skills
  stock_agent <- create_agent(
    name = "StockAnalyst",
    description = "Stock analysis agent",
    skills = "auto"
  )
}
```

`create_agent_registry` *Create an Agent Registry*

Description

Factory function to create a new AgentRegistry.

Usage

```
create_agent_registry(agents = NULL)
```

Arguments

`agents`	Optional list of Agent objects to register immediately.

Value

An AgentRegistry object.

Examples

```
if (interactive()) {
# Create registry with agents
cleaner <- create_agent("Cleaner", "Cleans data")
plotter <- create_agent("Plotter", "Creates visualizations")

registry <- create_agent_registry(list(cleaner, plotter))
print(registry$list_agents())  # "Cleaner", "Plotter"
}
```

`create_aihubmix` *Create AiHubMix Provider*

Description

Factory function to create an AiHubMix provider.

AiHubMix provides a unified API for various models including Claude, OpenAI, Gemini, etc.

Usage

```
create_aihubmix(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments

`api_key`	AiHubMix API key. Defaults to AIHUBMIX_API_KEY env var.
`base_url`	Base URL for API calls. Defaults to https://aihubmix.com/v1.
`headers`	Optional additional headers.

Value

An AiHubMixProvider object.

Examples

```
if (interactive()) {
    aihubmix <- create_aihubmix()
    model <- aihubmix$language_model("claude-sonnet-3-5")
    result <- generate_text(model, "Explain quantum computing in one sentence.")
}
```

create_aihubmix_anthropic

Create AiHubMix Provider (Anthropic API Format)

Description

Factory function to create an AiHubMix provider using the Anthropic-compatible API. This allows you to use AiHubMix Claude models with the native Anthropic API format, unlocking advanced features like Prompt Caching.

Usage

```
create_aihubmix_anthropic(
  api_key = NULL,
  extended_caching = FALSE,
  headers = NULL
)
```

Arguments

`api_key`	AiHubMix API key. Defaults to AIHUBMIX_API_KEY env var.
`extended_caching`	Logical. If TRUE, enables the 1-hour beta cache for Claude.
`headers`	Optional additional headers.

Details

AiHubMix provides an Anthropic-compatible endpoint at `https://aihubmix.com/v1`. This convenience function wraps `create_anthropic()` with AiHubMix-specific defaults.

Value

An AnthropicProvider object configured for AiHubMix.

Examples

```
if (interactive()) {
    # Use AiHubMix via Anthropic API format (unlocks caching)
    aihubmix_claude <- create_aihubmix_anthropic()
    model <- aihubmix_claude$language_model("claude-3-5-sonnet-20241022")
    result <- generate_text(model, "Hello Claude!")
}
```

`create_aihubmix_gemini` *Create AiHubMix Provider (Gemini API Format)*

Description

Factory function to create an AiHubMix provider using the Gemini-compatible API. This allows you to use Gemini models with the native Gemini API structure.

Usage

```
create_aihubmix_gemini(api_key = NULL, headers = NULL)
```

Arguments

`api_key`	AiHubMix API key. Defaults to AIHUBMIX_API_KEY env var.
`headers`	Optional additional headers.

Details

AiHubMix provides a Gemini-compatible endpoint at `https://aihubmix.com/gemini/v1beta/models`. This convenience function wraps `create_gemini()` with AiHubMix-specific defaults.

Value

A GeminiProvider object configured for AiHubMix.

Examples

```
if (interactive()) {
    # Use AiHubMix via Gemini API format
    aihubmix_gemini <- create_aihubmix_gemini()
    model <- aihubmix_gemini$language_model("gemini-2.5-flash")
    result <- generate_text(model, "Hello Gemini!")
}
```

`create_anthropic` *Create Anthropic Provider*

Description

Factory function to create an Anthropic provider.

Usage

```
create_anthropic(
  api_key = NULL,
  base_url = NULL,
  api_version = NULL,
  headers = NULL,
  name = NULL
)
```

Arguments

`api_key`	Anthropic API key. Defaults to ANTHROPIC_API_KEY env var.
`base_url`	Base URL for API calls. Defaults to https://api.anthropic.com/v1.
`api_version`	Anthropic API version header. Defaults to "2023-06-01".
`headers`	Optional additional headers.
`name`	Optional provider name override.

Value

An AnthropicProvider object.

Examples

```
if (interactive()) {
  anthropic <- create_anthropic(api_key = "sk-ant-...")
  model <- anthropic$language_model("claude-sonnet-4-20250514")
}
```

create_bailian *Create Alibaba Cloud Bailian Provider*

Description

Factory function to create an Alibaba Cloud Bailian provider using the DashScope API.

Usage

```
create_bailian(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments

`api_key`	DashScope API key. Defaults to DASHSCOPE_API_KEY env var.
`base_url`	Base URL for API calls. Defaults to https://dashscope.aliyuncs.com/compatible-mode/v1.
`headers`	Optional additional headers.

Value

A BailianProvider object.

Supported Models

DashScope platform hosts Qwen series and other models:

- **qwen-plus**: Balanced performance model
- **qwen-turbo**: Fast & cost-effective model
- **qwen-max**: Most capable model
- **qwq-32b**: Reasoning model with chain-of-thought
- **qwen-vl-plus**: Vision-language model
- Other third-party models available on the platform

Examples

```
if (interactive()) {
bailian <- create_bailian()

# Standard chat model
model <- bailian$language_model("qwen-plus")
result <- generate_text(model, "Hello")

# Reasoning model (QwQ with chain-of-thought)
model <- bailian$language_model("qwq-32b")
result <- generate_text(model, "Solve: What is 15 * 23?")
print(result$reasoning) # Chain-of-thought reasoning

# Default model (qwen-plus)
model <- bailian$language_model()
}
```

`create_channel_runtime` *Create a Channel Runtime*

Description

Helper for constructing a `ChannelRuntime`.

Usage

```
create_channel_runtime(
  session_store,
  model = NULL,
  agent = NULL,
  skills = NULL,
  tools = NULL,
  hooks = NULL,
  registry = NULL,
  max_steps = 10,
  session_policy = channel_default_session_policy()
)
```

Arguments

`session_store`	File-backed session store.
`model`	Optional default model id.
`agent`	Optional default agent.
`skills`	Optional skill paths or `"auto"`. Used only when `agent` is `NULL`.
`tools`	Optional default tools.
`hooks`	Optional session hooks.
`registry`	Optional provider registry.
`max_steps`	Maximum tool execution steps.
`session_policy`	Session routing policy list.

Value

A `ChannelRuntime`.

`create_chat_session` *Create a Chat Session*

Description

Factory function to create a new ChatSession object.

Usage

```
create_chat_session(
  model = NULL,
  system_prompt = NULL,
  tools = NULL,
  hooks = NULL,
  max_steps = 10,
  metadata = NULL,
  agent = NULL
)
```

Arguments

`model`	A LanguageModelV1 object or model string ID.
`system_prompt`	Optional system prompt.
`tools`	Optional list of Tool objects.
`hooks`	Optional HookHandler object.
`max_steps`	Maximum tool execution steps. Default 10.
`metadata`	Optional session metadata (list).
`agent`	Optional Agent object to initialize from.

Value

A ChatSession object.

Examples

```
if (interactive()) {
  # Create a chat session
  chat <- create_chat_session(
    model = "openai:gpt-4o",
    system_prompt = "You are a helpful R programming assistant."
  )

  # Create from an existing agent
 agent <- create_agent("MathAgent", "Does math", system_prompt = "You are a math wizard.")
  chat <- create_chat_session(model = "openai:gpt-4o", agent = agent)

  # Send messages
  response <- chat$send("How do I read a CSV file?")
  print(response$text)

  # Continue the conversation (history is maintained)
  response <- chat$send("What about Excel files?")

  # Check stats
  print(chat$stats())

  # Save session
  chat$save("my_session.rds")
}
```

`create_coder_agent` *Create a CoderAgent*

Description

Creates an agent specialized in writing and executing R code. The agent can execute R code in the session environment, making results available to other agents. Enhanced version with better safety controls and debugging support.

Usage

```
create_coder_agent(
  name = "CoderAgent",
  safe_mode = TRUE,
  timeout_ms = 30000,
  max_output_lines = 200
)
```

Arguments

`name`	Agent name. Default "CoderAgent".
`safe_mode`	If TRUE (default), restricts file system and network access.
`timeout_ms`	Code execution timeout in milliseconds. Default 30000.
`max_output_lines`	Maximum output lines to return. Default 50.

Value

An Agent object configured for R code execution.

Examples

```
if (interactive()) {
  coder <- create_coder_agent()
  session <- create_shared_session(model = "openai:gpt-4o")
  result <- coder$run(
    "Create a data frame with 3 rows and calculate the mean",
    session = session,
    model = "openai:gpt-4o"
  )
}
```

`create_computer_tools` *Create Computer Tools*

Description

Create atomic tools for computer abstraction layer. These tools provide a small set of primitives that agents can use to perform complex actions.

Usage

```
create_computer_tools(
  computer = NULL,
  working_dir = tempdir(),
  sandbox_mode = "permissive"
)
```

Arguments

`computer`	Computer instance (default: create new)
`working_dir`	Working directory. Defaults to `tempdir()`.
`sandbox_mode`	Sandbox mode: "strict", "permissive", or "none"

Value

List of Tool objects

`create_console_agent` *Create Console Agent*

Description

Create the default intelligent terminal agent for console_chat(). This agent can execute commands, manage files, and run R code through natural language interaction.

Usage

```
create_console_agent(
  working_dir = tempdir(),
  sandbox_mode = "permissive",
  additional_tools = NULL,
  language = "auto"
)
```

Arguments

`working_dir`	Working directory. Defaults to `tempdir()`.
`sandbox_mode`	Sandbox mode: "strict", "permissive", or "none" (default: "permissive").
`additional_tools`	Optional list of additional Tool objects to include.
`language`	Language for responses: "auto", "en", or "zh" (default: "auto").

Value

An Agent object configured for console interaction.

Examples

```
if (interactive()) {
    # Create default console agent
    agent <- create_console_agent()

    # Create with custom working directory
    agent <- create_console_agent(working_dir = "~/projects/myapp")

    # Use with console_chat
    console_chat("openai:gpt-4o", agent = agent)
}
```

`create_console_tools` *Create Console Tools*

Description

Create a set of tools optimized for console/terminal interaction. Includes computer tools (bash, read_file, write_file, execute_r_code) plus additional console-specific tools.

Usage

```
create_console_tools(working_dir = tempdir(), sandbox_mode = "permissive")
```

Arguments

`working_dir`	Working directory. Defaults to `tempdir()`.
`sandbox_mode`	Sandbox mode: "strict", "permissive", or "none" (default: "permissive").

Value

A list of Tool objects.

Examples

```
if (interactive()) {
    tools <- create_console_tools()
    # Use with an agent or session
    session <- create_chat_session(model = "openai:gpt-4o", tools = tools)
}
```

`create_custom_provider` *Create a custom provider*

Description

Creates a dynamic wrapper around existing model classes (OpenAI, Anthropic) based on user-provided configuration. The returned provider can be registered in the global `ProviderRegistry`.

Usage

```
create_custom_provider(
  provider_name,
  base_url,
  api_key = NULL,
  api_format = c("chat_completions", "responses", "anthropic_messages"),
  use_max_completion_tokens = FALSE
)
```

Arguments

`provider_name`	The identifier name for this custom provider (e.g. "my_custom_openai_proxy").
`base_url`	The base URL for the API endpoint.
`api_key`	The API key for authentication. If NULL, defaults to checking environmental variables.
`api_format`	The underlying API format to use. Supports "chat_completions" (OpenAI default), "responses" (OpenAI Responses API), and "anthropic_messages" (Anthropic Messages API).
`use_max_completion_tokens`	A boolean flag. If TRUE, injects the `is_reasoning_model` capability to ensure the model uses `max_completion_tokens` instead of `max_tokens`.

Value

A custom provider object with a `language_model(model_id)` method.

`create_data_agent` *Create a DataAgent*

Description

Creates an agent specialized in data manipulation using dplyr and tidyr. The agent can filter, transform, summarize, and reshape data frames in the session environment.

Usage

```
create_data_agent(name = "DataAgent", safe_mode = TRUE)
```

Arguments

`name`	Agent name. Default "DataAgent".
`safe_mode`	If TRUE (default), restricts operations to data manipulation only.

Value

An Agent object configured for data manipulation.

Examples

```
if (interactive()) {
  data_agent <- create_data_agent()
  session <- create_shared_session(model = "openai:gpt-4o")
  session$set_var("sales", data.frame(
    product = c("A", "B", "C"),
    revenue = c(100, 200, 150)
  ))
  result <- data_agent$run(
    "Calculate total revenue and find the top product",
    session = session,
    model = "openai:gpt-4o"
  )
}
```

`create_deepseek` *Create DeepSeek Provider*

Description

Factory function to create a DeepSeek provider.

DeepSeek offers two main models:

- **deepseek-chat**: Standard chat model (DeepSeek-V3.2 non-thinking mode)
- **deepseek-reasoner**: Reasoning model with chain-of-thought (DeepSeek-V3.2 thinking mode)

Usage

```
create_deepseek(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments

`api_key`	DeepSeek API key. Defaults to DEEPSEEK_API_KEY env var.
`base_url`	Base URL. Defaults to "https://api.deepseek.com".
`headers`	Optional additional headers.

Value

A DeepSeekProvider object.

Examples

```
if (interactive()) {
# Basic usage with deepseek-chat
deepseek <- create_deepseek()
model <- deepseek$language_model("deepseek-chat")
result <- generate_text(model, "Hello!")

# Using deepseek-reasoner for chain-of-thought reasoning
model_reasoner <- deepseek$language_model("deepseek-reasoner")
result <- model_reasoner$generate(
    messages = list(list(role = "user", content = "Solve: What is 15 * 23?")),
    max_tokens = 500
)
print(result$text) # Final answer
print(result$reasoning) # Chain-of-thought reasoning

# Streaming with reasoning
stream_text(model_reasoner, "Explain quantum entanglement step by step")
}
```

`create_deepseek_anthropic`
Create DeepSeek Provider (Anthropic API Format)

Description

Factory function to create a DeepSeek provider using the Anthropic-compatible API. This allows you to use DeepSeek models with the Anthropic API format.

Usage

```
create_deepseek_anthropic(api_key = NULL, headers = NULL)
```

Arguments

`api_key`	DeepSeek API key. Defaults to DEEPSEEK_API_KEY env var.
`headers`	Optional additional headers.

Details

DeepSeek provides an Anthropic-compatible endpoint at `https://api.deepseek.com/anthropic`. This convenience function wraps `create_anthropic()` with DeepSeek-specific defaults.

Note: When using an unsupported model name, the API backend will automatically map it to `deepseek-chat`.

Value

An AnthropicProvider object configured for DeepSeek.

Examples

```
if (interactive()) {
# Use DeepSeek via Anthropic API format
deepseek <- create_deepseek_anthropic()
model <- deepseek$language_model("deepseek-chat")
result <- generate_text(model, "Hello!")

# This is useful for tools that expect Anthropic API format
# such as Claude Code integration
}
```

`create_embeddings` *Create Embeddings*

Description

Generate embeddings for text using an embedding model.

Usage

```
create_embeddings(model, value, registry = NULL)
```

Arguments

`model`	Either an EmbeddingModelV1 object, or a string ID like "openai:text-embedding-3-small".
`value`	A character string or vector to embed.
`registry`	Optional ProviderRegistry to use.

Value

A list with embeddings and usage information.

Examples

```
if (interactive()) {
  model <- create_openai()$embedding_model("text-embedding-3-small")
  result <- create_embeddings(model, "Hello, world!")
  print(length(result$embeddings[[1]]))
}
```

`create_env_agent` *Create an EnvAgent*

Description

Creates an agent specialized in R environment and package management. The agent can check, install, and manage R packages with safety controls.

Usage

```
create_env_agent(
  name = "EnvAgent",
  allow_install = FALSE,
  allowed_repos = "https://cloud.r-project.org"
)
```

Arguments

`name`	Agent name. Default "EnvAgent".
`allow_install`	Allow package installation. Default FALSE.
`allowed_repos`	CRAN mirror URLs for installation.

Value

An Agent object configured for environment management.

Examples

```
if (interactive()) {
  env_agent <- create_env_agent(allow_install = TRUE)
  result <- env_agent$run(
    "Check if tidyverse is installed and load it",
    session = session,
    model = "openai:gpt-4o"
  )
}
```

create_feishu_channel_adapter

Create a Feishu Channel Adapter

Description

Helper for creating a `FeishuChannelAdapter`.

Usage

```
create_feishu_channel_adapter(
  app_id,
  app_secret,
  base_url = "https://open.feishu.cn",
  verification_token = NULL,
  encrypt_key = NULL,
  verify_signature = TRUE,
  send_text_fn = NULL,
  send_status_fn = NULL,
  download_resource_fn = NULL
)
```

Arguments

`app_id`	Feishu app id.
`app_secret`	Feishu app secret.
`base_url`	Feishu API base URL.
`verification_token`	Optional callback verification token.
`encrypt_key`	Optional event subscription encryption key.
`verify_signature`	Whether to validate Feishu callback signatures when applicable.
`send_text_fn`	Optional custom send function for tests or overrides.
`send_status_fn`	Optional custom status function for tests or overrides.
`download_resource_fn`	Optional custom downloader for inbound message resources.

Value

A `FeishuChannelAdapter`.

create_feishu_channel_runtime

Create a Feishu Channel Runtime

Description

Construct a `ChannelRuntime` and register a Feishu adapter on it.

Usage

```
create_feishu_channel_runtime(
  session_store,
  app_id,
  app_secret,
  base_url = "https://open.feishu.cn",
  verification_token = NULL,
  encrypt_key = NULL,
  verify_signature = TRUE,
  send_text_fn = NULL,
  send_status_fn = NULL,
  download_resource_fn = NULL,
  model = NULL,
  agent = NULL,
  skills = "auto",
  tools = NULL,
  hooks = NULL,
  registry = NULL,
  max_steps = 10,
  session_policy = channel_default_session_policy()
)
```

Arguments

`session_store`	Channel session store.
`app_id`	Feishu app id.
`app_secret`	Feishu app secret.
`base_url`	Feishu API base URL.
`verification_token`	Optional callback verification token.
`encrypt_key`	Optional event subscription encryption key.
`verify_signature`	Whether to validate Feishu callback signatures when applicable.
`send_text_fn`	Optional custom send function for tests or overrides.
`send_status_fn`	Optional custom status function for tests or overrides.

`download_resource_fn`	Optional custom downloader for inbound message resources.
`model`	Optional default model id.
`agent`	Optional default agent.
`skills`	Optional skill paths or `"auto"`. Defaults to `"auto"` when `agent` is `NULL`.
`tools`	Optional default tools.
`hooks`	Optional session hooks.
`registry`	Optional provider registry.
`max_steps`	Maximum tool execution steps.
`session_policy`	Optional session policy overrides.

Value

A `ChannelRuntime` with the Feishu adapter registered.

`create_feishu_event_processor`
Create a Feishu Event Processor

Description

Create a plain event processor for Feishu events that already arrived through an authenticated ingress such as the official long-connection SDK.

Usage

```
create_feishu_event_processor(runtime)
```

Arguments

`runtime`	A `ChannelRuntime` with a Feishu adapter registered.

Value

A function `(payload)` that processes one Feishu event payload.

`create_feishu_webhook_handler` *Create a Feishu Webhook Handler*

Description

Create a transport-agnostic handler that turns a raw Feishu callback request into a JSON HTTP response payload.

Usage

```
create_feishu_webhook_handler(runtime)
```

Arguments

`runtime`	A `ChannelRuntime` with a Feishu adapter registered.

Value

A function `(headers, body)` that returns a response list.

`create_file_agent` *Create a FileAgent*

Description

Creates an agent specialized in file system operations using fs and readr. The agent can read, write, and manage files with safety guardrails.

Usage

```
create_file_agent(
  name = "FileAgent",
  allowed_dirs = ".",
 allowed_extensions = c("csv", "tsv", "txt", "json", "rds", "rda", "xlsx", "xls")
)
```

Arguments

`name`	Agent name. Default "FileAgent".
`allowed_dirs`	Character vector of allowed directories. Default current dir.
`allowed_extensions`	Character vector of allowed file extensions.

Value

An Agent object configured for file operations.

Examples

```
if (interactive()) {
  file_agent <- create_file_agent(
    allowed_dirs = c("./data", "./output"),
    allowed_extensions = c("csv", "json", "txt", "rds")
  )
  result <- file_agent$run(
    "Read the sales.csv file and store it as 'sales_data'",
    session = session,
    model = "openai:gpt-4o"
  )
}
```

`create_file_channel_session_store`
Create a File Channel Session Store

Description

Helper for creating a local file-backed channel session store.

Usage

```
create_file_channel_session_store(base_dir)
```

Arguments

`base_dir`	Base directory for channel session state.

Value

A `FileChannelSessionStore`.

create_flow *Create a Flow*

Description

Factory function to create a new Flow object for enhanced multi-agent orchestration.

Usage

```
create_flow(
  session,
  model = NULL,
  registry = NULL,
  max_depth = 5,
  max_steps_per_agent = 10,
  enable_guardrails = TRUE
)
```

Arguments

`session`	A ChatSession object.
`model`	Optional default model ID to use (e.g., "openai:gpt-4o").
`registry`	Optional AgentRegistry for agent lookup and delegation.
`max_depth`	Maximum delegation depth. Default 5.
`max_steps_per_agent`	Maximum ReAct steps per agent. Default 10.
`enable_guardrails`	Enable safety guardrails. Default TRUE.

Value

A Flow object.

Examples

```
if (interactive()) {
  # Create an enhanced multi-agent flow
  session <- create_chat_session()
  cleaner <- create_agent("Cleaner", "Cleans data")
  plotter <- create_agent("Plotter", "Creates plots")
  registry <- create_agent_registry(list(cleaner, plotter))

  manager <- create_agent("Manager", "Coordinates data analysis")

  flow <- create_flow(
    session = session,
    model = "openai:gpt-4o",
    registry = registry,
    enable_guardrails = TRUE
  )

  # Run the manager with auto-delegation (unified delegate_task tool)
  result <- flow$run(manager, "Load data and create a visualization")
}
```

`create_gemini` *Create Gemini Provider*

Description

Factory function to create a Gemini provider.

Usage

```
create_gemini(api_key = NULL, base_url = NULL, headers = NULL, name = NULL)
```

Arguments

`api_key`	Gemini API key. Defaults to GEMINI_API_KEY env var.
`base_url`	Base URL for API calls. Defaults to https://generativelanguage.googleapis.com/v1beta/models.
`headers`	Optional additional headers.
`name`	Optional provider name override.

Value

A GeminiProvider object.

Examples

```
if (interactive()) {
    gemini <- create_gemini(api_key = "AIza...")
    model <- gemini$language_model("gemini-1.5-pro")
}
```

`create_hooks` *Create Hooks*

Description

Helper to create a HookHandler from a list of functions.

Usage

```
create_hooks(...)
```

Arguments

`...`	Named arguments matching supported hook names.

Value

A HookHandler object.

`create_mcp_client` *Create an MCP Client*

Description

Convenience function to create and connect to an MCP server.

Usage

```
create_mcp_client(command, args = character(), env = NULL)
```

Arguments

`command`	The command to run the MCP server
`args`	Command arguments
`env`	Environment variables

Value

An McpClient object

Examples

```
if (interactive()) {
  # Connect to GitHub MCP server
  client <- create_mcp_client(
    "npx",
    c("-y", "@modelcontextprotocol/server-github"),
    env = c(GITHUB_PERSONAL_ACCESS_TOKEN = Sys.getenv("GITHUB_TOKEN"))
  )

  # List available tools
  tools <- client$list_tools()

  # Use tools with generate_text
  result <- generate_text(
    model = "openai:gpt-4o",
    prompt = "List my GitHub repos",
    tools = client$as_sdk_tools()
  )

  client$close()
}
```

`create_mcp_server` *Create an MCP Server*

Description

Convenience function to create an MCP server.

Usage

```
create_mcp_server(name = "r-mcp-server", version = "0.1.0")
```

Arguments

`name`	Server name
`version`	Server version

Value

An McpServer object

Examples

```
if (interactive()) {
# Create a server with a custom tool
server <- create_mcp_server("my-r-server")

# Add a tool
server$add_tool(tool(
  name = "calculate",
  description = "Perform a calculation",
  parameters = z_object(
    expression = z_string(description = "R expression to evaluate")
  ),
  execute = function(args) {
    eval(parse(text = args$expression))
  }
))

# Start listening (blocking)
server$listen()
}
```

`create_mcp_sse_client` *Create MCP SSE Client*

Description

Create MCP SSE Client

Usage

```
create_mcp_sse_client(url, headers = list())
```

Arguments

`url`	The SSE endpoint URL
`headers`	named list of headers (e.g. for auth)

`create_mission` *Create a Mission*

Description

Factory function to create a new Mission object.

Usage

```
create_mission(
  goal,
  steps = NULL,
  model = NULL,
  executor = NULL,
  stall_policy = NULL,
  hooks = NULL,
  session = NULL,
  auto_plan = TRUE
)
```

Arguments

`goal`	Natural language goal description.
`steps`	Optional list of MissionStep objects. If NULL and auto_plan=TRUE, the LLM will decompose the goal into steps automatically.
`model`	Default model ID (e.g., "anthropic:claude-opus-4-6").
`executor`	Default executor (Agent, AgentTeam, Flow, or R function). Required when auto_plan = TRUE.
`stall_policy`	Named list for failure recovery. See default_stall_policy().
`hooks`	MissionHookHandler for lifecycle events.
`session`	Optional SharedSession. Created automatically if NULL.
`auto_plan`	If TRUE (default), use LLM to create steps when none are provided.

Value

A Mission object.

Examples

```
if (interactive()) {
  # Auto-planned mission
  agent <- create_agent("Analyst", "Analyzes data", model = "openai:gpt-4o")
  mission <- create_mission(
    goal     = "Load the iris dataset and summarize each species",
    executor = agent,
    model    = "openai:gpt-4o"
  )
  mission$run()

  # Manual steps
  mission2 <- create_mission(
    goal  = "Data pipeline",
    steps = list(
      create_step("step_1", "Load CSV data", executor = agent),
      create_step("step_2", "Summarize statistics", executor = agent,
                  depends_on = "step_1")
    ),
    model = "openai:gpt-4o"
  )
  mission2$run()
}
```

`create_mission_hooks` *Create Mission Hooks*

Description

Factory function to create a MissionHookHandler from named hook functions.

Usage

```
create_mission_hooks(
  on_mission_start = NULL,
  on_mission_planned = NULL,
  on_step_start = NULL,
  on_step_done = NULL,
  on_step_failed = NULL,
  on_mission_stall = NULL,
  on_mission_done = NULL
)
```

Arguments

`on_mission_start`	Optional function(mission) called when a Mission begins.
`on_mission_planned`	Optional function(mission) called after LLM planning.
`on_step_start`	Optional function(step, attempt) called before each step attempt.
`on_step_done`	Optional function(step, result) called on step success.
`on_step_failed`	Optional function(step, error, attempt) called on step failure.
`on_mission_stall`	Optional function(mission, step) called on stall detection.
`on_mission_done`	Optional function(mission) called on Mission completion.

Value

A MissionHookHandler object.

Examples

```
hooks <- create_mission_hooks(
  on_step_done = function(step, result) {
    message("Completed: ", step$description)
  },
  on_mission_stall = function(mission, step) {
    message("STALL detected at step: ", step$id)
  }
)
```

---

create_mission_orchestrator
Create a Mission Orchestrator

Description

Factory function to create a MissionOrchestrator.

Usage

```
create_mission_orchestrator(max_concurrent = 3, model = NULL, session = NULL)
```

Arguments

`max_concurrent`	Maximum simultaneous missions. Default 3.
`model`	Optional default model for all missions.
`session`	Optional shared SharedSession.

Value

A MissionOrchestrator object.

Examples

```
if (interactive()) {
 orchestrator <- create_mission_orchestrator(max_concurrent = 5, model = "openai:gpt-4o")

  orchestrator$submit(create_mission("Task A", executor = agent_a))
  orchestrator$submit(create_mission("Task B", executor = agent_b))
  orchestrator$submit(create_mission("Task C", executor = agent_c))

  results <- orchestrator$run_all()
  print(orchestrator$status())
}
```

`create_nvidia` *Create NVIDIA Provider*

Description

Factory function to create a NVIDIA provider.

Usage

```
create_nvidia(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments

`api_key`	NVIDIA API key. Defaults to NVIDIA_API_KEY env var.
`base_url`	Base URL. Defaults to "https://integrate.api.nvidia.com/v1".
`headers`	Optional additional headers.

Value

A NvidiaProvider object.

Examples

```
if (interactive()) {
nvidia <- create_nvidia()
model <- nvidia$language_model("z-ai/glm4.7")

# Enable thinking/reasoning
result <- generate_text(model, "Who are you?",
  chat_template_kwargs = list(enable_thinking = TRUE)
)
print(result$reasoning)
}
```

`create_openai` *Create OpenAI Provider*

Description

Factory function to create an OpenAI provider.

Usage

```
create_openai(
  api_key = NULL,
  base_url = NULL,
  organization = NULL,
  project = NULL,
  headers = NULL,
  name = NULL,
  disable_stream_options = FALSE
)
```

Arguments

`api_key`	OpenAI API key. Defaults to OPENAI_API_KEY env var.
`base_url`	Base URL for API calls. Defaults to https://api.openai.com/v1.
`organization`	Optional OpenAI organization ID.
`project`	Optional OpenAI project ID.
`headers`	Optional additional headers.
`name`	Optional provider name override (for compatible APIs).
`disable_stream_options`	Disable stream_options parameter (for providers like Volcengine that don't support it).

Value

An OpenAIProvider object.

Token Limit Parameters

The SDK provides a unified `max_tokens` parameter that automatically maps to the correct API field based on the model and API type:

- **Chat API (standard models)**: `max_tokens` -> `max_tokens`
- **Chat API (o1/o3 models)**: `max_tokens` -> `max_completion_tokens`
- **Responses API**: `max_tokens` -> `max_output_tokens` (total: reasoning + answer)

For advanced users who need fine-grained control:

- `max_completion_tokens`: Explicitly set completion tokens (Chat API, o1/o3)
- `max_output_tokens`: Explicitly set total output limit (Responses API)
- `max_answer_tokens`: Limit answer only, excluding reasoning (Responses API, Volcengine-specific)

Examples

```
if (interactive()) {
  # Basic usage with Chat Completions API
  openai <- create_openai(api_key = "sk-...")
  model <- openai$language_model("gpt-4o")
  result <- generate_text(model, "Hello!")

  # Using Responses API for reasoning models
  openai <- create_openai()
  model <- openai$responses_model("o1")
  result <- generate_text(model, "Solve this math problem...")
  print(result$reasoning) # Access chain-of-thought

  # Smart model selection (auto-detects best API)
  model <- openai$smart_model("o3-mini") # Uses Responses API
  model <- openai$smart_model("gpt-4o") # Uses Chat Completions API

  # Token limits - unified interface
  # For standard models: limits generated content
  result <- model$generate(messages = msgs, max_tokens = 1000)

  # For o1/o3 models: automatically maps to max_completion_tokens
  model_o1 <- openai$language_model("o1")
  result <- model_o1$generate(messages = msgs, max_tokens = 2000)

  # For Responses API: automatically maps to max_output_tokens (total limit)
  model_resp <- openai$responses_model("o1")
  result <- model_resp$generate(messages = msgs, max_tokens = 2000)

  # Advanced: explicitly control answer-only limit (Volcengine Responses API)
  result <- model_resp$generate(messages = msgs, max_answer_tokens = 500)

  # Multi-turn conversation with Responses API
  model <- openai$responses_model("o1")
  result1 <- generate_text(model, "What is 2+2?")
  result2 <- generate_text(model, "Now multiply that by 3") # Remembers context
  model$reset() # Start fresh conversation
}
```

---

create_openrouter *Create OpenRouter Provider*

Description

Factory function to create an OpenRouter provider.

Usage

```
create_openrouter(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments

`api_key`	OpenRouter API key. Defaults to OPENROUTER_API_KEY env var.
`base_url`	Base URL for API calls. Defaults to https://openrouter.ai/api/v1.
`headers`	Optional additional headers.

Value

An OpenRouterProvider object.

Supported Models

OpenRouter provides access to hundreds of models from many providers:

- **OpenAI**: "openai/gpt-4o", "openai/o1"
- **Anthropic**: "anthropic/claude-sonnet-4-20250514"
- **Google**: "google/gemini-2.5-pro"
- **DeepSeek**: "deepseek/deepseek-r1", "deepseek/deepseek-chat-v3-0324"
- **Meta**: "meta-llama/llama-4-maverick"
- And many more at https://openrouter.ai/models

Examples

```
if (interactive()) {
openrouter <- create_openrouter()

# Access any model via a unified API
model <- openrouter$language_model("openai/gpt-4o")
result <- generate_text(model, "Hello!")

# Reasoning model
model <- openrouter$language_model("deepseek/deepseek-r1")
result <- generate_text(model, "Solve: 15 * 23")
print(result$reasoning)
}
```

`create_orchestration` *Create Orchestration Flow (Compatibility Wrapper)*

Description

Creates an orchestration flow using Flow. Provided for backward compatibility.

Usage

```
create_orchestration(
  session,
  model,
  registry = NULL,
  max_depth = 5,
  max_steps_per_agent = 10,
  ...
)
```

Arguments

`session`	A session object.
`model`	The default model ID.
`registry`	Optional AgentRegistry.
`max_depth`	Maximum delegation depth. Default 5.
`max_steps_per_agent`	Maximum ReAct steps per agent. Default 10.
`...`	Additional arguments.

Value

A Flow object.

`create_permission_hook` *Create Permission Hook*

Description

Create a hook that enforces a permission mode for tool execution.

Usage

```
create_permission_hook(
  mode = c("implicit", "explicit", "escalate"),
  allowlist = c("search_web", "read_resource", "read_file")
)
```

Arguments

`mode`	Permission mode: • "implicit" (default): Auto-approve all tools. • "explicit": Ask for confirmation in the console for every tool. • "escalate": Ask for confirmation only for tools not in the allowlist.
`allowlist`	List of tool names that are auto-approved in "escalate" mode. Default includes read-only tools like "search_web", "read_file".

Value

A HookHandler object.

`create_planner_agent` *Create a PlannerAgent*

Description

Creates an agent specialized in breaking down complex tasks into steps using chain-of-thought reasoning. The planner helps decompose problems and create action plans.

Usage

```
create_planner_agent(name = "PlannerAgent")
```

Arguments

`name`	Agent name. Default "PlannerAgent".

Value

An Agent object configured for planning and reasoning.

Examples

```
if (interactive()) {
  planner <- create_planner_agent()
  result <- planner$run(
    "How should I approach building a machine learning model for customer churn?",
    model = "openai:gpt-4o"
  )
}
```

`create_r_code_tool` *Create R Code Interpreter Tool*

Description

Creates a meta-tool (`execute_r_code`) backed by a SandboxManager. This single tool replaces all individual tools for the LLM, enabling batch execution, data filtering, and local computation.

Usage

```
create_r_code_tool(sandbox)
```

Arguments

`sandbox`	A SandboxManager object.

Value

A Tool object named `execute_r_code`.

`create_sandbox_system_prompt` *Create Sandbox System Prompt*

Description

Generates a system prompt section that instructs the LLM how to use the R code sandbox effectively.

Usage

```
create_sandbox_system_prompt(sandbox)
```

Arguments

`sandbox`	A SandboxManager object.

Value

A character string to append to the system prompt.

`create_schema_from_func` *Create Schema from Function*

Description

Inspects an R function and generates a z_object schema based on its arguments and default values.

Usage

```
create_schema_from_func(
  func,
  include_args = NULL,
  exclude_args = NULL,
  params = NULL,
  func_name = NULL,
  type_mode = c("infer", "any")
)
```

Arguments

`func`	The R function to inspect.
`include_args`	Optional character vector of argument names to include. If provided, only these arguments will be included in the schema.
`exclude_args`	Optional character vector of argument names to exclude.
`params`	Optional named list of parameter values to use as defaults. This allows overriding the function's default values (e.g., with values extracted from an existing plot layer).
`func_name`	Optional string of the function name to look up documentation. If not provided, attempts to infer from 'func' symbol.
`type_mode`	How to assign parameter types. "infer" (default) uses default values to infer types. "any" uses z_any() for all parameters.

Value

A z_object schema.

Examples

```
if (interactive()) {
my_func <- function(a = 1, b = "text", c = TRUE) {}
schema <- create_schema_from_func(my_func)
print(schema)

# Override defaults
schema_override <- create_schema_from_func(my_func, params = list(a = 99))
}
```

`create_session` *Create Session (Compatibility Wrapper)*

Description

Creates a session using either the new SharedSession or legacy ChatSession based on feature flags. This provides a migration path for existing code.

Usage

```
create_session(
  model = NULL,
  system_prompt = NULL,
  tools = NULL,
  hooks = NULL,
  max_steps = 10,
  ...
)
```

Arguments

`model`	A LanguageModelV1 object or model string ID.
`system_prompt`	Optional system prompt.
`tools`	Optional list of Tool objects.
`hooks`	Optional HookHandler object.
`max_steps`	Maximum tool execution steps. Default 10.
`...`	Additional arguments passed to session constructor.

Value

A SharedSession or ChatSession object.

Examples

```
if (interactive()) {
# Automatically uses SharedSession if feature enabled
session <- create_session(model = "openai:gpt-4o")

# Force legacy session
sdk_set_feature("use_shared_session", FALSE)
session <- create_session(model = "openai:gpt-4o")
}
```

`create_shared_session` *Create a Shared Session*

Description

Factory function to create a new SharedSession object.

Usage

```
create_shared_session(
  model = NULL,
  system_prompt = NULL,
  tools = NULL,
  hooks = NULL,
  max_steps = 10,
  sandbox_mode = "strict",
  trace_enabled = TRUE
)
```

Arguments

`model`	A LanguageModelV1 object or model string ID.
`system_prompt`	Optional system prompt.
`tools`	Optional list of Tool objects.
`hooks`	Optional HookHandler object.
`max_steps`	Maximum tool execution steps. Default 10.
`sandbox_mode`	Sandbox mode: "strict", "permissive", or "none". Default "strict".
`trace_enabled`	Enable execution tracing. Default TRUE.

Value

A SharedSession object.

Examples

```
if (interactive()) {
# Create a shared session for multi-agent use
session <- create_shared_session(
  model = "openai:gpt-4o",
  sandbox_mode = "strict",
  trace_enabled = TRUE
)

# Execute code safely
result <- session$execute_code("x <- 1:10; mean(x)")

# Check trace
print(session$trace_summary())
}
```

`create_skill` *Create Skill Scaffold*

Description

Create a new skill project with the standard structure.

Usage

```
create_skill(name, path = tempdir(), author = NULL, description = NULL)
```

Arguments

`name`	Skill name.
`path`	Directory to create the skill in.
`author`	Author name.
`description`	Brief description.

Value

Path to the created skill directory.

`create_skill_architect_agent` *Create a SkillArchitect Agent*

Description

Creates an advanced agent specialized in creating, testing, and refining new skills. It follows a rigorous "Ingest -> Design -> Implement -> Verify" workflow.

Usage

```
create_skill_architect_agent(
  name = "SkillArchitect",
  registry = NULL,
  model = NULL
)
```

Arguments

`name`	Agent name. Default "SkillArchitect".
`registry`	Optional SkillRegistry object (defaults to creating one from inst/skills).
`model`	The model object to use for verification (spawning a tester agent).

Value

An Agent object configured for skill architecture.

`create_skill_forge_tools` *Create Skill Forge Tools*

Description

Wraps the analysis and testing functions into Tools for the Skill Architect.

Usage

```
create_skill_forge_tools(registry, model)
```

Arguments

`registry`	SkillRegistry for looking up skills during testing.
`model`	Model to use for the test runner.

`create_skill_registry` *Create a Skill Registry*

Description

Convenience function to create and populate a SkillRegistry.

Usage

```
create_skill_registry(path, recursive = FALSE)
```

Arguments

`path`	Path to scan for skills.
`recursive`	Whether to scan subdirectories. Default FALSE.

Value

A populated SkillRegistry object.

Examples

```
if (interactive()) {
# Scan a skills directory
registry <- create_skill_registry(".aimd/skills")

# List available skills
registry$list_skills()

# Get a specific skill
skill <- registry$get_skill("seurat_analysis")
}
```

`create_skill_tools` *Create Skill Tools*

Description

Create the built-in tools for interacting with skills.

Usage

```
create_skill_tools(registry)
```

Arguments

`registry`	A SkillRegistry object.

Value

A list of Tool objects.

`create_standard_registry` *Create Standard Agent Registry*

Description

Creates an AgentRegistry pre-populated with standard library agents based on feature flags.

Usage

```
create_standard_registry(
  include_data = TRUE,
  include_file = TRUE,
  include_env = TRUE,
  include_coder = TRUE,
  include_visualizer = TRUE,
  include_planner = TRUE,
  file_allowed_dirs = ".",
  env_allow_install = FALSE
)
```

Arguments

`include_data`	Include DataAgent. Default TRUE.
`include_file`	Include FileAgent. Default TRUE.
`include_env`	Include EnvAgent. Default TRUE.
`include_coder`	Include CoderAgent. Default TRUE.
`include_visualizer`	Include VisualizerAgent. Default TRUE.
`include_planner`	Include PlannerAgent. Default TRUE.
`file_allowed_dirs`	Allowed directories for FileAgent.
`env_allow_install`	Allow package installation for EnvAgent.

Value

An AgentRegistry object.

Examples

```
if (interactive()) {
# Create registry with all standard agents
registry <- create_standard_registry()

# Create registry with only data and visualization agents
registry <- create_standard_registry(
  include_file = FALSE,
  include_env = FALSE,
  include_planner = FALSE
)
}
```

`create_step` *Create a MissionStep*

Description

Factory function to create a MissionStep.

Usage

```
create_step(
  id,
  description,
  executor = NULL,
  max_retries = 2,
  timeout_secs = NULL,
  parallel = FALSE,
  depends_on = NULL
)
```

Arguments

`id`	Unique step ID (e.g., "step_1").
`description`	Natural language task description.
`executor`	Agent, AgentTeam, Flow, or R function to execute the step.
`max_retries`	Maximum retry attempts before stall escalation. Default 2.
`timeout_secs`	Optional per-step timeout in seconds. Default NULL.
`parallel`	If TRUE, this step may run in parallel with other parallel steps.
`depends_on`	Character vector of prerequisite step IDs.

Value

A MissionStep object.

Examples

```
if (interactive()) {
  step <- create_step(
    id          = "load_data",
    description = "Load the CSV file and return a summary",
    executor    = my_agent,
    max_retries = 3
  )
}
```

`create_stepfun` *Create Stepfun Provider*

Description

Factory function to create a Stepfun provider.

Usage

```
create_stepfun(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments

`api_key`	Stepfun API key. Defaults to STEPFUN_API_KEY env var.
`base_url`	Base URL for API calls. Defaults to https://api.stepfun.com/v1.
`headers`	Optional additional headers.

Value

A StepfunProvider object.

Supported Models

- **step-1-32k**: Model: step-1-32k (Tools) | ctx: 32k
- **step-1v-32k**: Vision enabled model with 32k context (Vision, Tools) | ctx: 32k
- **step-1-8k**: Model: step-1-8k
- **step-1-256k**: Model: step-1-256k
- **step-1v-8k**: Model: step-1v-8k (Vision)
- **step-2-16k**: Model: step-2-16k
- **step-1x-medium**: Model: step-1x-medium
- **step-tts-mini**: Model: step-tts-mini (Audio)
- **step-2-16k-202411**: Model: step-2-16k-202411
- **step-asr**: Model: step-asr (Audio)

- **step-1o-vision-32k**: Model: step-1o-vision-32k (Vision)
- **step-2-mini**: Model: step-2-mini
- **step-2-16k-exp**: Model: step-2-16k-exp
- **step-1o-turbo-vision**: Model: step-1o-turbo-vision (Vision)
- **step-1o-audio**: Model: step-1o-audio (Audio)
- *... and 16 more models. Use* `list_models("stepfun")` *to see all.*

Examples

```
if (interactive()) {
    stepfun <- create_stepfun()
    model <- stepfun$language_model("step-1-8k")
    result <- generate_text(model, "Explain quantum computing in one sentence.")
}
```

---

create_team *Create an Agent Team*

Description

Helper to create an AgentTeam.

Usage

```
create_team(name = "AgentTeam", model = NULL, session = NULL)
```

Arguments

`name`	Team name.
`model`	Optional default model for the team.
`session`	Optional shared ChatSession for the team.

Value

An AgentTeam object.

`create_telemetry` *Create Telemetry*

Description

Create Telemetry

Usage

```
create_telemetry(trace_id = NULL)
```

Arguments

`trace_id`	Optional trace ID.

Value

A Telemetry object.

`create_visualizer_agent` *Create a VisualizerAgent*

Description

Creates an agent specialized in creating data visualizations using ggplot2. Enhanced version with plot type recommendations, theme support, and automatic data inspection.

Usage

```
create_visualizer_agent(
  name = "VisualizerAgent",
  output_dir = NULL,
  default_theme = "theme_minimal",
  default_width = 8,
  default_height = 6
)
```

Arguments

`name`	Agent name. Default "VisualizerAgent".
`output_dir`	Optional directory to save plots. If NULL, plots are stored in the session environment.
`default_theme`	Default ggplot2 theme. Default "theme_minimal".
`default_width`	Default plot width in inches. Default 8.
`default_height`	Default plot height in inches. Default 6.

Value

An Agent object configured for data visualization.

Examples

```
if (interactive()) {
  visualizer <- create_visualizer_agent()
  session <- create_shared_session(model = "openai:gpt-4o")
  session$set_var("df", data.frame(x = 1:10, y = (1:10)^2))
  result <- visualizer$run(
    "Create a scatter plot of df showing the relationship between x and y",
    session = session,
    model = "openai:gpt-4o"
  )
}
```

`create_volcengine` *Create Volcengine/Ark Provider*

Description

Factory function to create a Volcengine provider using the Ark API.

Usage

```
create_volcengine(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments

`api_key`	Volcengine API key. Defaults to ARK_API_KEY env var.
`base_url`	Base URL for API calls. Defaults to https://ark.cn-beijing.volces.com/api/v3.
`headers`	Optional additional headers.

Value

A VolcengineProvider object.

Supported Models

- **doubao-lite-128k-240428**: Model: doubao-lite-128k-240428
- **doubao-pro-128k-240515**: Model: doubao-pro-128k-240515
- **doubao-lite-4k-240328**: Model: doubao-lite-4k-240328
- **doubao-lite-32k-240428**: Model: doubao-lite-32k-240428
- **doubao-pro-4k-240515**: Model: doubao-pro-4k-240515
- **doubao-lite-4k-character-240515**: Model: doubao-lite-4k-character-240515

- **doubao-embedding-text-240515**: Model: doubao-embedding-text-240515
- **mistral-7b-instruct-v0.2**: Model: mistral-7b-instruct-v0.2 (Vision)
- **doubao-pro-4k-character-240515**: Model: doubao-pro-4k-character-240515
- **doubao-pro-4k-functioncall-240515**: Model: doubao-pro-4k-functioncall-240515
- **doubao-lite-4k-pretrain-character-240516**: Model: doubao-lite-4k-pretrain-character-240516
- **doubao-pro-32k-character-240528**: Model: doubao-pro-32k-character-240528
- **doubao-pro-4k-browsing-240524**: Model: doubao-pro-4k-browsing-240524
- **doubao-pro-32k-functioncall-240515**: Model: doubao-pro-32k-functioncall-240515
- **doubao-pro-4k-functioncall-240615**: Model: doubao-pro-4k-functioncall-240615
- *... and 123 more models. Use* `list_models("volcengine")` *to see all.*

API Formats

Volcengine supports both Chat Completions API and Responses API:

- `language_model()`: Uses Chat Completions API (standard)
- `responses_model()`: Uses Responses API (for reasoning models)
- `smart_model()`: Auto-selects based on model ID

Token Limit Parameters for Volcengine Responses API

Volcengine's Responses API has two mutually exclusive token limit parameters:

- `max_output_tokens`: Total limit including reasoning + answer (default mapping)
- `max_tokens` (API level): Answer-only limit, excluding reasoning

The SDK's unified `max_tokens` parameter maps to `max_output_tokens` by default, which is the **safe choice** to prevent runaway reasoning costs.

For advanced users who want answer-only limits:

- Use `max_answer_tokens` parameter to explicitly set answer-only limit
- Use `max_output_tokens` parameter to explicitly set total limit

Examples

```
if (interactive()) {
    volcengine <- create_volcengine()

    # Chat API (standard models)
    model <- volcengine$language_model("doubao-1-5-pro-256k-250115")
    result <- generate_text(model, "Hello")

    # Responses API (reasoning models like DeepSeek)
    model <- volcengine$responses_model("deepseek-r1-250120")

    # Default: max_tokens limits total output (reasoning + answer)
    result <- model$generate(messages = msgs, max_tokens = 2000)
    # Advanced: limit only the answer part (reasoning can be longer)
    result <- model$generate(messages = msgs, max_answer_tokens = 500)

    # Smart model selection (auto-detects best API)
    model <- volcengine$smart_model("deepseek-r1-250120")
}
```

create_xai *Create xAI Provider*

Description

Factory function to create an xAI provider.

xAI provides Grok models:

- **Grok**: "grok-3", "grok-4", "grok-beta", "grok-2-1212", etc.

Usage

```
create_xai(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments

`api_key`	xAI API key. Defaults to XAI_API_KEY env var.
`base_url`	Base URL for API calls. Defaults to https://api.x.ai/v1.
`headers`	Optional additional headers.

Value

A XAIProvider object.

Examples

```
if (interactive()) {
    xai <- create_xai()
    model <- xai$language_model("grok-beta")
    result <- generate_text(model, "Explain quantum computing in one sentence.")
}
```

`create_z_ggtree` *Create Schema for ggtree Function*

Description

Specialized wrapper around create_schema_from_func for ggtree/ggplot2 functions. Handles common mapping and data arguments specifically.

Usage

```
create_z_ggtree(func, layer = NULL)
```

Arguments

`func`	The R function (e.g., geom_tiplab).
`layer`	Optional ggplot2 Layer object. If provided, its parameters (aes_params and geom_params) will be used to override the schema defaults. This is useful for creating "Edit Mode" forms for existing plot layers.

Value

A z_object schema.

`DeepSeekProvider` *DeepSeek Provider Class*

Description

Provider class for DeepSeek.

Super class

`aisdk::OpenAIProvider` -> `DeepSeekProvider`

Methods

Public methods:

- `DeepSeekProvider$new()`
- `DeepSeekProvider$language_model()`
- `DeepSeekProvider$clone()`

Method `new()`**:** Initialize the DeepSeek provider.

Usage:

```
DeepSeekProvider$new(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments:

`api_key` DeepSeek API key. Defaults to DEEPSEEK_API_KEY env var.

`base_url` Base URL. Defaults to https://api.deepseek.com.

`headers` Optional additional headers.

Method `language_model()`**:** Create a language model.

Usage:

```
DeepSeekProvider$language_model(model_id = NULL)
```

Arguments:

`model_id` The model ID (e.g., "deepseek-chat", "deepseek-reasoner").

Returns: A DeepSeekLanguageModel object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`download_model` *Download Model from Hugging Face*

Description

Download a quantized model from Hugging Face Hub.

Usage

```
download_model(repo_id, filename, dest_dir = NULL, quiet = FALSE)
```

Arguments

`repo_id`	The Hugging Face repository ID (e.g., "TheBloke/Llama-2-7B-GGUF").
`filename`	The specific file to download.
`dest_dir`	Destination directory. Defaults to "~/.cache/aisdk/models".
`quiet`	Suppress download progress.

Value

Path to the downloaded file.

`EmbeddingModelV1` *Embedding Model V1 (Abstract Base Class)*

Description

Abstract interface for embedding models.

Public fields

`specification_version` The version of this specification.

`provider` The provider identifier.

`model_id` The model identifier.

Methods

Public methods:

- `EmbeddingModelV1$new()`
- `EmbeddingModelV1$do_embed()`
- `EmbeddingModelV1$clone()`

Method `new()`**:** Initialize the embedding model.

Usage:

```
EmbeddingModelV1$new(provider, model_id)
```

Arguments:

`provider` Provider name.

`model_id` Model ID.

Method `do_embed()`**:** Generate embeddings for a value. Abstract method.

Usage:

```
EmbeddingModelV1$do_embed(value)
```

Arguments:

`value` A character string or vector to embed.

Returns: A list with embeddings.

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

Usage:

```
EmbeddingModelV1$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`enable_api_tests` *Check if API tests should be enabled*

Description

Check if API tests should be enabled

Usage

```
enable_api_tests()
```

Value

TRUE if API tests are enabled and keys are available, FALSE otherwise

`execute_tool_calls` *Execute Tool Calls*

Description

Execute a list of tool calls returned by an LLM. This function safely executes each tool, handling errors gracefully and returning a standardized result format.

Implements multi-layer defense strategy:

1. Tool name repair (case fixing, snake_case conversion, fuzzy matching)
2. Invalid tool routing for graceful degradation
3. Argument parsing with JSON repair
4. Error capture and structured error responses

Usage

```
execute_tool_calls(
  tool_calls,
  tools,
  hooks = NULL,
  envir = NULL,
  repair_enabled = TRUE
)
```

Arguments

`tool_calls`	A list of tool call objects, each with id, name, and arguments.
`tools`	A list of Tool objects to search for matching tools.
`hooks`	Optional HookHandler object.
`envir`	Optional environment in which to execute tools. When provided, tool functions can access and modify variables in this environment, enabling cross-agent data sharing through a shared session environment.
`repair_enabled`	Whether to attempt tool call repair (default TRUE).

Value

A list of execution results, each containing:

- id: The tool call ID
- name: The tool name
- result: The execution result (or error message)
- is_error: TRUE if an error occurred during execution

`expect_llm_pass` *Expect LLM Pass*

Description

Custom testthat expectation that evaluates whether an LLM response meets specified criteria. Uses an LLM judge to assess the response.

Usage

```
expect_llm_pass(response, criteria, model = NULL, threshold = 0.7, info = NULL)
```

Arguments

`response`	The LLM response to evaluate (text or GenerateResult object).
`criteria`	Character string describing what constitutes a passing response.
`model`	Model to use for judging (default: same as response or gpt-4o).
`threshold`	Minimum score (0-1) to pass (default: 0.7).
`info`	Additional information to include in failure message.

Value

Invisibly returns the evaluation result.

Examples

```
if (interactive()) {
  test_that("agent answers math questions correctly", {
    result <- generate_text(
      model = "openai:gpt-4o",
      prompt = "What is 2 + 2?"
    )
    expect_llm_pass(result, "The response should contain the number 4")
  })
}
```

`expect_no_hallucination` *Expect No Hallucination*

Description

Test that an LLM response does not contain hallucinated information when compared against ground truth.

Usage

```
expect_no_hallucination(
  response,
  ground_truth,
  model = NULL,
  tolerance = 0.1,
  info = NULL
)
```

Arguments

`response`	The LLM response to check.
`ground_truth`	The factual information to check against.
`model`	Model to use for checking.
`tolerance`	Allowed deviation (0 = strict, 1 = lenient).
`info`	Additional information for failure message.

`expect_tool_selection` *Expect Tool Selection*

Description

Test that an agent selects the correct tool(s) for a given task.

Usage

```
expect_tool_selection(result, expected_tools, exact = FALSE, info = NULL)
```

Arguments

`result`	A GenerateResult object from generate_text with tools.
`expected_tools`	Character vector of expected tool names.
`exact`	If TRUE, require exactly these tools (no more, no less).
`info`	Additional information for failure message.

`extract_geom_params` *Extract Geom Parameters from ggproto Object*

Description

Dynamically extracts parameter information from a ggplot2 geom. This handles the "scattered definitions" problem by reading from source.

Usage

```
extract_geom_params(geom_name)
```

Arguments

`geom_name`	Name of the geom (e.g., "point", "line").

Value

List with default_aes, required_aes, optional_aes, extra_params.

`FeishuChannelAdapter` *Feishu Channel Adapter*

Description

Transport adapter for Feishu/Lark event callbacks and text replies.

Super class

`aisdk::ChannelAdapter` -> `FeishuChannelAdapter`

Methods

Public methods:

- `FeishuChannelAdapter$new()`
- `FeishuChannelAdapter$parse_request()`
- `FeishuChannelAdapter$resolve_session_key()`
- `FeishuChannelAdapter$format_inbound_message()`
- `FeishuChannelAdapter$prepare_inbound_message()`
- `FeishuChannelAdapter$send_text()`
- `FeishuChannelAdapter$send_status()`
- `FeishuChannelAdapter$send_attachment()`
- `FeishuChannelAdapter$clone()`

Method `new()`**:** Initialize the Feishu adapter.

Usage:

```
FeishuChannelAdapter$new(
  app_id,
  app_secret,
  base_url = "https://open.feishu.cn",
  verification_token = NULL,
  encrypt_key = NULL,
  verify_signature = TRUE,
  send_text_fn = NULL,
  send_status_fn = NULL,
  download_resource_fn = NULL
)
```

Arguments:

`app_id` Feishu app id.

`app_secret` Feishu app secret.

`base_url` Feishu API base URL.

`verification_token` Optional callback verification token.

`encrypt_key` Optional event subscription encryption key.

`verify_signature` Whether to validate Feishu callback signatures when applicable.

`send_text_fn` Optional custom send function for tests or overrides.
`send_status_fn` Optional custom status function for tests or overrides.
`download_resource_fn` Optional custom downloader for inbound message resources.

Method `parse_request()`**:** Parse a Feishu callback request.

Usage:

```
FeishuChannelAdapter$parse_request(headers = NULL, body = NULL, ...)
```

Arguments:

`headers` Request headers.
`body` Raw JSON string or parsed list.
`...` Unused.

Returns: Channel request result.

Method `resolve_session_key()`**:** Resolve a stable session key for a Feishu inbound message.

Usage:

```
FeishuChannelAdapter$resolve_session_key(message, policy = list())
```

Arguments:

`message` Normalized inbound message.
`policy` Session policy list.

Returns: Character scalar session key.

Method `format_inbound_message()`**:** Format a Feishu inbound message for a `ChatSession`.

Usage:

```
FeishuChannelAdapter$format_inbound_message(message)
```

Arguments:

`message` Normalized inbound message.

Returns: Character scalar prompt.

Method `prepare_inbound_message()`**:** Prepare a Feishu inbound message using stored document context.

Usage:

```
FeishuChannelAdapter$prepare_inbound_message(session, message)
```

Arguments:

`session` Current `ChatSession`.
`message` Normalized inbound message.

Returns: Enriched inbound message.

Method `send_text()`**:** Send a final text reply to Feishu.

Usage:

```
FeishuChannelAdapter$send_text(message, text, ...)
```

Arguments:

`message` Original normalized inbound message.

`text` Final outbound text.

`...` Unused.

Returns: Parsed API response.

Method `send_status()`**:** Send an intermediate status message to Feishu.

Usage:

```
FeishuChannelAdapter$send_status(
  message,
  status = c("thinking", "working", "error"),
  text = NULL,
  ...
)
```

Arguments:

`message` Original normalized inbound message.

`status` Status name.

`text` Optional status text.

`...` Unused.

Returns: Parsed API response or NULL.

Method `send_attachment()`**:** Send a generated local attachment to Feishu.

Usage:

```
FeishuChannelAdapter$send_attachment(message, path, ...)
```

Arguments:

`message` Original normalized inbound message.

`path` Absolute local file path.

`...` Unused.

Returns: Parsed API response or NULL.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`fetch_api_models` *Fetch available models from API provider*

Description

Fetch available models from API provider

Usage

```
fetch_api_models(provider, api_key = NULL, base_url = NULL)
```

Arguments

`provider`	Provider name ("openai", "nvidia", "anthropic", etc.)
`api_key`	API key
`base_url`	Base URL

Value

A data frame with 'id' column and capability flag columns

`FileChannelSessionStore` *File Channel Session Store*

Description

File-backed session store for external messaging channels.

Super class

`aisdk::ChannelSessionStore` -> `FileChannelSessionStore`

Public fields

`base_dir` Base directory for persisted channel sessions.

Methods

Public methods:

- `FileChannelSessionStore$new()`
- `FileChannelSessionStore$get_session_path()`
- `FileChannelSessionStore$get_index_path()`
- `FileChannelSessionStore$list_sessions()`
- `FileChannelSessionStore$has_processed_event()`
- `FileChannelSessionStore$mark_processed_event()`
- `FileChannelSessionStore$get_record()`
- `FileChannelSessionStore$load_session()`
- `FileChannelSessionStore$save_session()`
- `FileChannelSessionStore$update_record()`
- `FileChannelSessionStore$link_child_session()`
- `FileChannelSessionStore$clone()`

Method `new()`**:** Initialize a file-backed channel session store.

Usage:

```
FileChannelSessionStore$new(base_dir)
```

Arguments:

`base_dir` Base directory for store files.

Method `get_session_path()`**:** Get the on-disk session file path for a key.

Usage:

```
FileChannelSessionStore$get_session_path(session_key)
```

Arguments:

`session_key` Session key.

Returns: Absolute file path.

Method `get_index_path()`**:** Get the channel index path.

Usage:

```
FileChannelSessionStore$get_index_path()
```

Returns: Absolute file path.

Method `list_sessions()`**:** List all session records.

Usage:

```
FileChannelSessionStore$list_sessions()
```

Returns: Named list of session records.

Method `has_processed_event()`**:** Check whether an event id has already been processed.

Usage:

```
FileChannelSessionStore$has_processed_event(channel_id, event_id)
```

Arguments:

`channel_id` Channel identifier.

`event_id` Event identifier.

Returns: Logical scalar.

Method `mark_processed_event()`**:** Mark an event id as processed.

Usage:

```
FileChannelSessionStore$mark_processed_event(
  channel_id,
  event_id,
  payload = NULL
)
```

Arguments:

`channel_id` Channel identifier.

`event_id` Event identifier.

`payload` Optional event payload to keep in the dedupe index.

Returns: Invisible stored event record.

Method `get_record()`**:** Get a single session record.

Usage:

```
FileChannelSessionStore$get_record(session_key)
```

Arguments:

`session_key` Session key.

Returns: Session record or NULL.

Method `load_session()`**:** Load a persisted `ChatSession`.

Usage:

```
FileChannelSessionStore$load_session(
  session_key,
  tools = NULL,
  hooks = NULL,
  registry = NULL
)
```

Arguments:

`session_key` Session key.

`tools` Optional tools to reattach.

`hooks` Optional hooks to reattach.

`registry` Optional provider registry.

Returns: A `ChatSession` or NULL if no persisted state exists.

Method `save_session()`**:** Save a `ChatSession` and update the local index.

Usage:

```
FileChannelSessionStore$save_session(session_key, session, record = NULL)
```

Arguments:

`session_key` Session key.

`session` `ChatSession` instance.

`record` Optional record fields to merge into the index.

Returns: Invisible normalized record.

Method `update_record()`**:** Update a record without saving a session file.

Usage:

```
FileChannelSessionStore$update_record(session_key, record)
```

Arguments:

`session_key` Session key.

`record` Record fields to merge.

Returns: Invisible updated record.

Method `link_child_session()`**:** Register a child session relationship.

Usage:

```
FileChannelSessionStore$link_child_session(
  parent_session_key,
  child_session_key
)
```

Arguments:

`parent_session_key` Parent session key.

`child_session_key` Child session key.

Returns: Invisible updated parent record.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`Flow` *Flow Class*

Description

R6 class representing an orchestration layer for multi-agent systems. Features:

- Comprehensive delegation tracing
- Automatic delegate_task tool generation
- Depth and context limits with guardrails
- Result aggregation and summarization

Methods

Public methods:

- `Flow$new()`
- `Flow$depth()`
- `Flow$current()`
- `Flow$session()`
- `Flow$set_global_context()`
- `Flow$global_context()`
- `Flow$delegate()`
- `Flow$generate_delegate_tool()`
- `Flow$run()`
- `Flow$get_delegation_history()`
- `Flow$delegation_stats()`
- `Flow$clear_history()`
- `Flow$print()`
- `Flow$clone()`

Method `new()`**:** Initialize a new Flow.

Usage:

```
Flow$new(
  session,
  model = NULL,
  registry = NULL,
  max_depth = 5,
  max_steps_per_agent = 10,
  max_context_tokens = 4000,
  enable_guardrails = TRUE
)
```

Arguments:

`session` A ChatSession object.

`model` Optional default model ID to use. If NULL, inherits from session.

`registry` Optional AgentRegistry for agent lookup.

`max_depth` Maximum delegation depth. Default 5.

`max_steps_per_agent` Maximum ReAct steps per agent. Default 10.

`max_context_tokens` Maximum context tokens per delegation. Default 4000.

`enable_guardrails` Enable safety guardrails. Default TRUE.

Method `depth()`**:** Get the current call stack depth.

Usage:

```
Flow$depth()
```

Returns: Integer depth.

Method `current()`**:** Get the current active agent.

Usage:

```
Flow$current()
```

Returns: The currently active Agent, or NULL.

Method `session()`**:** Get the shared session.

Usage:

```
Flow$session()
```

Returns: The ChatSession object.

Method `set_global_context()`**:** Set the global context (the user's original goal).

Usage:

```
Flow$set_global_context(context)
```

Arguments:

`context` Character string describing the overall goal.

Returns: Invisible self for chaining.

Method `global_context()`**:** Get the global context.

Usage:

```
Flow$global_context()
```

Returns: The global context string.

Method `delegate()`**:** Delegate a task to another agent with enhanced tracking.

Usage:

```
Flow$delegate(agent, task, context = NULL, priority = "normal")
```

Arguments:

`agent` The Agent to delegate to.
`task` The task instruction.
`context` Optional additional context.
`priority` Task priority: "high", "normal", "low". Default "normal".

Returns: The text result from the delegate agent.

Method `generate_delegate_tool()`**:** Generate the delegate_task tool for manager agents.

Usage:

```
Flow$generate_delegate_tool()
```

Details: Creates a single unified tool that can delegate to any registered agent. This is more efficient than generating separate tools per agent.

Returns: A Tool object for delegation.

Method `run()`**:** Run a primary agent with enhanced orchestration.

Usage:

```
Flow$run(agent, task, use_unified_delegate = TRUE)
```

Arguments:

`agent` The primary/manager Agent to run.

`task` The user's task/input.

`use_unified_delegate` Use single delegate_task tool. Default TRUE.

Returns: The final result from the primary agent.

Method `get_delegation_history()`**:** Get delegation history.

Usage:

```
Flow$get_delegation_history(agent_name = NULL, limit = NULL)
```

Arguments:

`agent_name` Optional filter by agent name.

`limit` Maximum number of records to return.

Returns: A list of delegation records.

Method `delegation_stats()`**:** Get delegation statistics.

Usage:

```
Flow$delegation_stats()
```

Returns: A list with counts, timing, and success rates.

Method `clear_history()`**:** Clear delegation history.

Usage:

```
Flow$clear_history()
```

Returns: Invisible self for chaining.

Method `print()`**:** Print method for Flow.

Usage:

```
Flow$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`GeminiProvider` *Gemini Provider Class*

Description

Provider class for Google Gemini.

Public fields

`specification_version` Provider spec version.

Methods

Public methods:

- `GeminiProvider$new()`
- `GeminiProvider$language_model()`
- `GeminiProvider$clone()`

Method `new()`**:** Initialize the Gemini provider.

Usage:

```
GeminiProvider$new(
  api_key = NULL,
  base_url = NULL,
  headers = NULL,
  name = NULL
)
```

Arguments:

`api_key` Gemini API key. Defaults to GEMINI_API_KEY env var.

`base_url` Base URL for API calls. Defaults to https://generativelanguage.googleapis.com/v1beta/models.

`headers` Optional additional headers.

`name` Optional provider name override.

Method `language_model()`**:** Create a language model.

Usage:

```
GeminiProvider$language_model(
  model_id = Sys.getenv("GEMINI_MODEL", "gemini-2.5-flash")
)
```

Arguments:

`model_id` The model ID (e.g., "gemini-1.5-pro", "gemini-1.5-flash", "gemini-2.0-flash").

Returns: A GeminiLanguageModel object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`GenerateResult` *Generate Result*

Description

Result object returned by model generation.

Details

This class uses `lock_objects = FALSE` to allow dynamic field addition. This enables the ReAct loop and other components to attach additional metadata (like `steps`, `all_tool_calls`) without modifying the class.

For models that support reasoning/thinking (like OpenAI o1/o3, DeepSeek, Claude with extended thinking), the `reasoning` field contains the model's chain-of-thought content.

For Responses API models, `response_id` contains the server-side response ID which can be used for multi-turn conversations without sending full history.

Public fields

`text` The generated text content.

`usage` Token usage information (list with prompt_tokens, completion_tokens, total_tokens).

`finish_reason` Reason the model stopped generating.

`warnings` Any warnings from the model.

`raw_response` The raw response from the API.

`tool_calls` List of tool calls requested by the model. Each item contains id, name, arguments.

`steps` Number of ReAct loop steps taken (when max_steps > 1).

`all_tool_calls` Accumulated list of all tool calls made across all ReAct steps.

`reasoning` Chain-of-thought/reasoning content from models that support it (o1, o3, DeepSeek, etc.).

`response_id` Server-side response ID for Responses API (used for stateful multi-turn conversations).

Methods

Public methods:

- `GenerateResult$new()`
- `GenerateResult$print()`
- `GenerateResult$clone()`

Method `new()`**:** Initialize a GenerateResult object.

Usage:

```
GenerateResult$new(
  text = NULL,
  usage = NULL,
  finish_reason = NULL,
  warnings = NULL,
  raw_response = NULL,
  tool_calls = NULL,
  steps = NULL,
  all_tool_calls = NULL,
  reasoning = NULL,
  response_id = NULL
)
```

Arguments:

`text` Generated text.

`usage` Token usage.

`finish_reason` Reason for stopping.

`warnings` Warnings.

`raw_response` Raw API response.

`tool_calls` Tool calls requested by the model.

`steps` Number of ReAct steps taken.

`all_tool_calls` All tool calls across steps.

`reasoning` Chain-of-thought content.

`response_id` Server-side response ID for Responses API.

Method `print()`**:** Print method for GenerateResult.

Usage:

```
GenerateResult$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`generate_text` *Generate Text*

Description

Generate text using a language model. This is the primary high-level function for non-streaming text generation.

When tools are provided and max_steps > 1, the function will automatically execute tool calls and feed results back to the LLM in a ReAct-style loop until the LLM produces a final response or max_steps is reached.

Usage

```
generate_text(
  model = NULL,
  prompt,
  system = NULL,
  temperature = 0.7,
  max_tokens = NULL,
  tools = NULL,
  max_steps = 1,
  sandbox = FALSE,
  skills = NULL,
  session = NULL,
  hooks = NULL,
  registry = NULL,
  ...
)
```

Arguments

`model`	Either a LanguageModelV1 object, or a string ID like "openai:gpt-4o".
`prompt`	A character string prompt, or a list of messages.
`system`	Optional system prompt.
`temperature`	Sampling temperature (0-2). Default 0.7.
`max_tokens`	Maximum tokens to generate.
`tools`	Optional list of Tool objects for function calling.
`max_steps`	Maximum number of generation steps (tool execution loops). Default 1 (single generation, no automatic tool execution). Set to higher values (e.g., 5) to enable automatic tool execution.
`sandbox`	Logical. If TRUE, enables R-native programmatic sandbox mode. All tools are bound into an isolated R environment and replaced by a single `execute_r_code` meta-tool. The LLM writes R code to batch-invoke tools, filter data with dplyr/purrr, and return only summary results, dramatically reducing token usage and latency. Default FALSE.
`skills`	Optional path to skills directory, or a SkillRegistry object. When provided, skill tools are auto-injected and skill summaries are added to the system prompt.
`session`	Optional ChatSession object. When provided, tool executions run in the session's environment, enabling cross-agent data sharing.
`hooks`	Optional HookHandler object for intercepting events.
`registry`	Optional ProviderRegistry to use (defaults to global registry).
`...`	Additional arguments passed to the model.

Value

A GenerateResult object with text and optionally tool_calls. When max_steps > 1 and tools are used, the result includes:

- steps: Number of steps taken
- all_tool_calls: List of all tool calls made across all steps

Examples

```
if (interactive()) {
  # Using hooks
  my_hooks <- create_hooks(
    on_generation_start = function(model, prompt, tools) message("Starting..."),
    on_tool_start = function(tool, args) message("Calling tool ", tool$name)
  )
  result <- generate_text(model, "...", hooks = my_hooks)
}
```

`get_ai_session` *Get AI Engine Session*

Description

Gets the current AI engine session for inspection or manual interaction.

Usage

```
get_ai_session(session_name = "default")
```

Arguments

`session_name`	Name of the session. Default is "default".

Value

A ChatSession object or NULL if not initialized.

`get_anthropic_base_url` *Get Anthropic base URL from environment*

Description

Get Anthropic base URL from environment

Usage

```
get_anthropic_base_url()
```

Value

Base URL for Anthropic API (default: official)

`get_anthropic_model` *Get Anthropic model name from environment*

Description

Get Anthropic model name from environment

Usage

```
get_anthropic_model()
```

Value

Model name (default: claude-sonnet-4-20250514)

`get_anthropic_model_id` *Get Anthropic model ID from environment*

Description

Get Anthropic model ID from environment

Usage

```
get_anthropic_model_id()
```

Value

Model ID (default: anthropic:claude-sonnet-4-20250514)

`get_default_registry` *Get Default Registry*

Description

Returns the global default provider registry, creating it if necessary.

Usage

```
get_default_registry()
```

Value

A ProviderRegistry object.

get_memory *Get or Create Global Memory*

Description

Get the global project memory instance, creating it if necessary.

Usage

```
get_memory()
```

Value

A ProjectMemory object.

get_model *Get Default Model*

Description

Returns the current package-wide default language model. This is used by high-level helpers when `model = NULL`. If no explicit default has been set, `get_model()` falls back to `getOption("aisdk.default_model")` and then to `"openai:gpt-4o"`.

Usage

```
get_model(default = "openai:gpt-4o")
```

Arguments

`default`	Fallback model identifier when no explicit default has been set.

Value

A model identifier string or a `LanguageModelV1` object.

Examples

```
get_model()
```

`get_model_info` *Get Full Model Info*

Description

Returns the full metadata for a single model as a list. Useful for framework internals to auto-configure parameters (e.g., max_tokens, context_window).

Usage

```
get_model_info(provider, model_id)
```

Arguments

`provider`	The name of the provider.
`model_id`	The model ID string.

Value

A list containing all available metadata for the model, or NULL if not found.

`get_openai_base_url` *Get OpenAI Base URL from environment*

Description

Get OpenAI Base URL from environment

Usage

```
get_openai_base_url()
```

Value

Base URL string

`get_openai_embedding_model` *Get OpenAI Embedding Model from environment*

Description

Get OpenAI Embedding Model from environment

Usage

```
get_openai_embedding_model()
```

Value

Model name string

`get_openai_model` *Get OpenAI Model from environment*

Description

Get OpenAI Model from environment

Usage

```
get_openai_model()
```

Value

Model name string

`get_openai_model_id` *Get OpenAI Model ID from environment*

Description

Get OpenAI Model ID from environment

Usage

```
get_openai_model_id()
```

Value

Model ID string

`get_r_context` *Get R Context*

Description

Generates a text summary of R objects to be used as context for the LLM.

Usage

```
get_r_context(vars, envir = parent.frame())
```

Arguments

`vars`	Character vector of variable names to include.
`envir`	The environment to look for variables in. Default is parent.frame().

Value

A single string containing the summaries of the requested variables.

Examples

```
if (interactive()) {
df <- data.frame(x = 1:10, y = rnorm(10))
context <- get_r_context("df")
cat(context)
}
```

`get_skill_store` *Get Skill Store*

Description

Get the global skill store instance.

Usage

```
get_skill_store()
```

Value

A SkillStore object.

`ggplot_to_frontend_json` *Export ggplot as Frontend-Ready JSON*

Description

Exports a ggplot object as JSON optimized for frontend rendering. Addresses all frontend feedback:

- Strict scalar typing (no for missing values)
- Structured units with pre-calculated pixel values
- Stable IDs for React keys
- Consistent Array of Structures pattern

Usage

```
ggplot_to_frontend_json(
  plot,
  width = 800,
  height = 600,
  include_data = TRUE,
  include_built = FALSE,
  pretty = FALSE
)
```

Arguments

`plot`	A ggplot object.
`width`	Plot width in pixels.
`height`	Plot height in pixels.
`include_data`	Whether to include data.
`include_built`	Whether to include ggplot_build() output.
`pretty`	Format JSON with indentation.

Value

JSON string optimized for frontend.

Examples

```
if (interactive()) {
library(ggplot2)
p <- ggplot(mtcars, aes(wt, mpg)) + geom_point()
json <- ggplot_to_frontend_json(p, width = 800, height = 600)
}
```

`ggplot_to_z_object` *Convert ggplot Object to Schema-Compliant Structure*

Description

Converts a ggplot object to a JSON-serializable structure with precise empty value handling and render hints for frontend.

Usage

```
ggplot_to_z_object(plot, include_data = TRUE, include_render_hints = TRUE)
```

Arguments

`plot`	A ggplot object.
`include_data`	Whether to include data in output.
`include_render_hints`	Whether to include frontend render hints.

Value

A list structure matching z_ggplot schema.

`has_api_key` *Check if specific provider key is available*

Description

Check if specific provider key is available

Usage

```
has_api_key(provider)
```

Arguments

`provider`	Provider name ("openai" or "anthropic")

Value

TRUE if key is available and valid

HookHandler *Hook Handler*

Description

R6 class to manage and execute hooks.

Public fields

`hooks` List of hook functions.

Methods

Public methods:

- `HookHandler$new()`
- `HookHandler$trigger_generation_start()`
- `HookHandler$trigger_generation_end()`
- `HookHandler$trigger_tool_start()`
- `HookHandler$trigger_tool_end()`
- `HookHandler$clone()`

Method `new()`**:** Initialize HookHandler

Usage:

```
HookHandler$new(hooks_list = list())
```

Arguments:

`hooks_list` A list of hook functions. Supported hooks:
- on_generation_start(model, prompt, tools)
- on_generation_end(result)
- on_tool_start(tool, args)
- on_tool_end(tool, result)
- on_tool_approval(tool, args) - Return TRUE to approve, FALSE to deny.

Method `trigger_generation_start()`**:** Trigger on_generation_start

Usage:

```
HookHandler$trigger_generation_start(model, prompt, tools)
```

Arguments:

`model` The language model object.

`prompt` The prompt being sent.

`tools` The list of tools provided.

Method `trigger_generation_end()`**:** Trigger on_generation_end

Usage:

```
HookHandler$trigger_generation_end(result)
```

Arguments:

`result` The generation result object.

Method `trigger_tool_start():` Trigger on_tool_start

Usage:

```
HookHandler$trigger_tool_start(tool, args)
```

Arguments:

`tool` The tool object.

`args` The arguments for the tool.

Method `trigger_tool_end():` Trigger on_tool_end

Usage:

```
HookHandler$trigger_tool_end(tool, result)
```

Arguments:

`tool` The tool object.

`result` The result from the tool execution.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`hooks` *Hooks System*

Description

A system for intercepting and monitoring AI SDK events. Allows implementation of "Human-in-the-loop", logging, and validation.

`hypothesis_fix_verify` *Hypothesis-Fix-Verify Loop*

Description

Advanced self-healing execution that generates hypotheses about errors, attempts fixes, and verifies the results.

Usage

```
hypothesis_fix_verify(
  code,
  model = NULL,
  test_fn = NULL,
  max_iterations = 5,
  verbose = TRUE
)
```

Arguments

`code`	Character string of R code to execute.
`model`	LLM model for analysis.
`test_fn`	Optional function to verify the result is correct.
`max_iterations`	Maximum fix iterations.
`verbose`	Print progress.

Value

List with result, fix history, and verification status.

`init_skill` *Initialize a New Skill*

Description

Creates a new skill directory with the standard "textbook" structure: SKILL.md, scripts/, references/, and assets/.

Usage

```
init_skill(name, path = tempdir())
```

Arguments

`name`	Name of the skill.
`path`	Parent directory where the skill folder will be created.

Value

Path to the created skill directory.

`install_skill` *Install a Skill*

Description

Install a skill from the global skill store or a GitHub repository.

Usage

```
install_skill(skill_ref, version = NULL, force = FALSE)
```

Arguments

`skill_ref`	Skill reference (e.g., "username/skillname").
`version`	Optional specific version.
`force`	Force reinstallation.

Value

The installed Skill object.

Examples

```
if (interactive()) {
# Install from GitHub
install_skill("aisdk/data-analysis")

# Install specific version
install_skill("aisdk/visualization", version = "1.2.0")

# Force reinstall
install_skill("aisdk/ml-tools", force = TRUE)
}
```

`knitr_engine` *Knitr Engine for AI*

Description

Implements a custom knitr engine `{ai}` that allows using LLMs to generate and execute R code within RMarkdown/Quarto documents.

`LanguageModelV1` *Language Model V1 (Abstract Base Class)*

Description

Abstract interface for language models. All LLM providers must implement this class. Uses do_ prefix for internal methods to prevent direct usage by end-users.

Public fields

`specification_version` The version of this specification.

`provider` The provider identifier (e.g., "openai").

`model_id` The model identifier (e.g., "gpt-4o").

`capabilities` Model capability flags (e.g., is_reasoning_model).

Methods

Public methods:

- `LanguageModelV1$new()`
- `LanguageModelV1$has_capability()`
- `LanguageModelV1$generate()`
- `LanguageModelV1$stream()`
- `LanguageModelV1$do_generate()`
- `LanguageModelV1$do_stream()`
- `LanguageModelV1$format_tool_result()`
- `LanguageModelV1$get_history_format()`
- `LanguageModelV1$clone()`

Method `new()`**:** Initialize the model with provider and model ID.

Usage:

```
LanguageModelV1$new(provider, model_id, capabilities = list())
```

Arguments:

`provider` Provider name.

`model_id` Model ID.

`capabilities` Optional list of capability flags.

Method `has_capability()`**:** Check if model has a specific capability.

Usage:

```
LanguageModelV1$has_capability(cap)
```

Arguments:

`cap` Capability name (e.g., "is_reasoning_model").

Returns: Logical.

Method `generate()`**:** Public generation method (wrapper for do_generate).

Usage:

```
LanguageModelV1$generate(...)
```

Arguments:

`...` Call options passed to do_generate.

Returns: A GenerateResult object.

Method `stream()`**:** Public streaming method (wrapper for do_stream).

Usage:

```
LanguageModelV1$stream(callback, ...)
```

Arguments:

`callback` Function to call with each chunk.
`...` Call options passed to do_stream.

Returns: A GenerateResult object.

Method `do_generate()`**:** Generate text (non-streaming). Abstract method.

Usage:

```
LanguageModelV1$do_generate(params)
```

Arguments:

`params` A list of call options.

Returns: A GenerateResult object.

Method `do_stream()`**:** Generate text (streaming). Abstract method.

Usage:

```
LanguageModelV1$do_stream(params, callback)
```

Arguments:

`params` A list of call options.
`callback` A function called for each chunk (text, done).

Returns: A GenerateResult object (accumulated from the stream).

Method `format_tool_result()`**:** Format a tool execution result for the provider's API.

Usage:

```
LanguageModelV1$format_tool_result(tool_call_id, tool_name, result_content)
```

Arguments:

`tool_call_id` The ID of the tool call.
`tool_name` The name of the tool.
`result_content` The result content from executing the tool.

Returns: A list formatted as a message for this provider's API.

Method `get_history_format()`**:** Get the message format used by this model's API for history.

Usage:

`LanguageModelV1$get_history_format()`

Returns: A character string ("openai" or "anthropic").

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

Usage:

```
LanguageModelV1$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`list_local_models` *List Available Local Models*

Description

Scan common directories for available local model files.

Usage

```
list_local_models(paths = NULL)
```

Arguments

`paths`	Character vector of directories to scan. Defaults to common locations.

Value

A data frame with model information.

`list_models` *List Models for Provider*

Description

Returns a data frame of models for the specified provider based on the static configuration. Includes enriched metadata when available (context window, pricing, capabilities).

Usage

```
list_models(provider = NULL)
```

Arguments

`provider`	The name of the provider (e.g., "stepfun"). If NULL, all providers are listed.

Value

A data frame containing model details.

`list_skills` *List Installed Skills*

Description

List all installed skills.

Usage

```
list_skills()
```

Value

A data frame of installed skills.

`load_chat_session` *Load a Chat Session*

Description

Load a previously saved ChatSession from a file.

Usage

```
load_chat_session(path, tools = NULL, hooks = NULL, registry = NULL)
```

Arguments

`path`	File path to load from (.rds or .json).
`tools`	Optional list of Tool objects (tools are not saved, must be re-provided).
`hooks`	Optional HookHandler object.
`registry`	Optional ProviderRegistry.

Value

A ChatSession object with restored state.

Examples

```
if (interactive()) {
  # Load a saved session
  chat <- load_chat_session("my_session.rds", tools = my_tools)

  # Continue where you left off
  response <- chat$send("Let's continue our discussion")
}
```

McpClient *MCP Client*

Description

Connect to and communicate with an MCP server process.

Details

Manages connection to an external MCP server via stdio.

Public fields

`process` The processx process object

`server_info` Information about the connected server

`capabilities` Server capabilities

Methods

Public methods:

- `McpClient$new()`
- `McpClient$list_tools()`
- `McpClient$call_tool()`
- `McpClient$list_resources()`
- `McpClient$read_resource()`
- `McpClient$is_alive()`
- `McpClient$close()`
- `McpClient$as_sdk_tools()`
- `McpClient$clone()`

Method `new()`**:** Create a new MCP Client

Usage:

```
McpClient$new(command, args = character(), env = NULL)
```

Arguments:

`command` The command to run (e.g., "npx", "python")

`args` Command arguments (e.g., c("-y", "@modelcontextprotocol/server-github"))

`env` Environment variables as a named character vector

Returns: A new McpClient object

Method `list_tools()`**:** List available tools from the MCP server

Usage:

```
McpClient$list_tools()
```

Returns: A list of tool definitions

Method `call_tool()`**:** Call a tool on the MCP server

Usage:

```
McpClient$call_tool(name, arguments = list())
```

Arguments:

`name` The tool name

`arguments` Tool arguments as a named list

Returns: The tool result

Method `list_resources()`**:** List available resources from the MCP server

Usage:

```
McpClient$list_resources()
```

Returns: A list of resource definitions

Method `read_resource()`**:** Read a resource from the MCP server

Usage:

```
McpClient$read_resource(uri)
```

Arguments:

`uri` The resource URI

Returns: The resource contents

Method `is_alive()`**:** Check if the MCP server process is alive

Usage:

```
McpClient$is_alive()
```

Returns: TRUE if alive, FALSE otherwise

Method `close()`**:** Close the MCP client connection

Usage:

```
McpClient$close()
```

Method `as_sdk_tools()`**:** Convert MCP tools to SDK Tool objects

Usage:

```
McpClient$as_sdk_tools()
```

Returns: A list of Tool objects

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`McpDiscovery` *MCP Discovery Class*

Description

R6 class for discovering MCP servers on the local network using mDNS/DNS-SD (Bonjour) protocol.

Public fields

`discovered` List of discovered MCP endpoints.

`registry_url` URL of the remote skill registry.

Methods

Public methods:

- `McpDiscovery$new()`
- `McpDiscovery$scan_network()`
- `McpDiscovery$register()`
- `McpDiscovery$query_capabilities()`
- `McpDiscovery$list_endpoints()`
- `McpDiscovery$search_registry()`
- `McpDiscovery$print()`
- `McpDiscovery$clone()`

Method `new()`**:** Create a new MCP Discovery instance.

Usage:

```
McpDiscovery$new(registry_url = NULL)
```

Arguments:

`registry_url` Optional URL for remote skill registry.

Returns: A new McpDiscovery object.

Method `scan_network()`**:** Scan the local network for MCP servers.

Usage:

```
McpDiscovery$scan_network(timeout_seconds = 5, service_type = "_mcp._tcp")
```

Arguments:

`timeout_seconds` How long to scan for services.

`service_type` The mDNS service type to look for.

Returns: A data frame of discovered services.

Method `register()`**:** Register a known MCP endpoint manually.

Usage:

```
McpDiscovery$register(name, host, port, capabilities = NULL)
```

Arguments:

`name` Service name.

`host` Hostname or IP address.

`port` Port number.

`capabilities` Optional list of capabilities.

Returns: Self (invisibly).

Method `query_capabilities()`**:** Query a discovered server for its capabilities.

Usage:

```
McpDiscovery$query_capabilities(host, port)
```

Arguments:

`host` Hostname or IP.

`port` Port number.

Returns: A list of server capabilities.

Method `list_endpoints()`**:** List all discovered MCP endpoints.

Usage:

```
McpDiscovery$list_endpoints()
```

Returns: A data frame of endpoints.

Method `search_registry()`**:** Search the remote registry for skills.

Usage:

```
McpDiscovery$search_registry(query)
```

Arguments:

`query` Search query.

Returns: A data frame of matching skills.

Method `print()`**:** Print method for McpDiscovery.

Usage:

```
McpDiscovery$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`McpRouter` *MCP Router Class*

Description

A virtual MCP server that aggregates tools from multiple downstream MCP servers into a unified interface. Supports hot-swapping and skill negotiation.

Public fields

`clients` List of connected MCP clients.

`tool_map` Mapping of tool names to their source clients.

`capabilities` Aggregated capabilities from all clients.

Methods

Public methods:

- `McpRouter$new()`
- `McpRouter$add_client()`
- `McpRouter$connect()`
- `McpRouter$remove_client()`
- `McpRouter$list_tools()`
- `McpRouter$call_tool()`
- `McpRouter$as_sdk_tools()`
- `McpRouter$negotiate()`
- `McpRouter$status()`
- `McpRouter$close()`
- `McpRouter$print()`
- `McpRouter$clone()`

Method `new()`**:** Create a new MCP Router.

Usage:

```
McpRouter$new()
```

Returns: A new McpRouter object.

Method `add_client()`**:** Add an MCP client to the router.

Usage:

```
McpRouter$add_client(name, client)
```

Arguments:

`name` Unique name for this client.

`client` An McpClient object.

Returns: Self (invisibly).

Method `connect()`: Connect to an MCP server and add it to the router.

Usage:

```
McpRouter$connect(name, command, args = character(), env = NULL)
```

Arguments:

`name` Unique name for this connection.

`command` Command to run the MCP server.

`args` Command arguments.

`env` Environment variables.

Returns: Self (invisibly).

Method `remove_client()`: Remove an MCP client from the router (hot-swap out).

Usage:

```
McpRouter$remove_client(name)
```

Arguments:

`name` Name of the client to remove.

Returns: Self (invisibly).

Method `list_tools()`: List all available tools across all connected clients.

Usage:

```
McpRouter$list_tools()
```

Returns: A list of tool definitions.

Method `call_tool()`: Call a tool, routing to the appropriate client.

Usage:

```
McpRouter$call_tool(name, arguments = list())
```

Arguments:

`name` Tool name.

`arguments` Tool arguments.

Returns: The tool result.

Method `as_sdk_tools()`: Get all tools as SDK Tool objects for use with generate_text.

Usage:

```
McpRouter$as_sdk_tools()
```

Returns: A list of Tool objects.

Method `negotiate()`: Negotiate capabilities with a specific client.

Usage:

```
McpRouter$negotiate(client_name)
```

Arguments:

`client_name` Name of the client.

Returns: A list of negotiated capabilities.

Method `status()`**:** Get router status.

Usage:

```
McpRouter$status()
```

Returns: A list with status information.

Method `close()`**:** Close all client connections.

Usage:

```
McpRouter$close()
```

Method `print()`**:** Print method for McpRouter.

Usage:

```
McpRouter$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`McpServer` *MCP Server*

Description

Expose R functions as MCP tools to external clients.

Details

Serves R tools and resources via MCP protocol over stdio.

Public fields

`name` Server name

`version` Server version

`tools` Registered tools

`resources` Registered resources

Methods

Public methods:

- `McpServer$new()`
- `McpServer$add_tool()`
- `McpServer$add_resource()`
- `McpServer$listen()`
- `McpServer$process_message()`
- `McpServer$clone()`

Method `new()`**:** Create a new MCP Server

Usage:

```
McpServer$new(name = "r-mcp-server", version = "0.1.0")
```

Arguments:

`name` Server name

`version` Server version

Returns: A new McpServer object

Method `add_tool()`**:** Add a tool to the server

Usage:

```
McpServer$add_tool(tool)
```

Arguments:

`tool` A Tool object from the SDK

Returns: self (for chaining)

Method `add_resource()`**:** Add a resource to the server

Usage:

```
McpServer$add_resource(
  uri,
  name,
  description = "",
  mime_type = "text/plain",
  read_fn
)
```

Arguments:

`uri` Resource URI

`name` Resource name

`description` Resource description

`mime_type` MIME type

`read_fn` Function that returns the resource content

Returns: self (for chaining)

Method `listen()`**:** Start listening for MCP requests on stdin/stdout This is a blocking call.

Usage:

`McpServer$listen()`

Method `process_message()`**:** Process a single MCP message (for testing)

Usage:

`McpServer$process_message(json_str)`

Arguments:

`json_str` The JSON-RPC message

Returns: The response, or NULL for notifications

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

Usage:

`McpServer$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

`McpSseClient` *MCP SSE Client*

Description

Connect to an MCP server via Server-Sent Events (SSE).

Details

Manages connection to a remote MCP server via SSE transport.

Super class

`aisdk::McpClient` -> `McpSseClient`

Public fields

`endpoint` The POST endpoint for sending messages (received from SSE init)

`auth_headers` Authentication headers

Methods

Public methods:

- `McpSseClient$new()`
- `McpSseClient$clone()`

Method `new()`**:** Create a new MCP SSE Client

Usage:

`McpSseClient$new(url, headers = list())`

Arguments:

`url` The SSE endpoint URL

`headers` named list of headers (e.g. for auth)

Returns: A new McpSseClient object

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

Usage:

`McpSseClient$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

`mcp_discover` *Distributed MCP Ecosystem*

Description

Service discovery and dynamic composition for MCP (Model Context Protocol) servers. Implements mDNS/DNS-SD discovery, skill negotiation, and hot-swapping of tools at runtime.

Factory function to create an MCP discovery instance.

Usage

```
mcp_discover(registry_url = NULL)
```

Arguments

`registry_url`	Optional URL for remote skill registry.

Value

An McpDiscovery object.

Examples

```
if (interactive()) {
# Create discovery instance
discovery <- mcp_discover()

# Scan local network
services <- discovery$scan_network()

# Register a known endpoint
discovery$register("my-server", "localhost", 3000)

# List all discovered endpoints
discovery$list_endpoints()
}
```

mcp_router *Create MCP Router*

Description

Factory function to create an MCP router for aggregating multiple servers.

Usage

```
mcp_router()
```

Value

An McpRouter object.

Examples

```
if (interactive()) {
# Create router
router <- mcp_router()

# Connect to multiple MCP servers
router$connect("github", "npx", c("-y", "@modelcontextprotocol/server-github"))
router$connect("filesystem", "npx", c("-y", "@modelcontextprotocol/server-filesystem"))

# Use aggregated tools with generate_text
result <- generate_text(
  model = "openai:gpt-4o",
  prompt = "List my GitHub repos and save to a file",
  tools = router$as_sdk_tools()
)

# Hot-swap: remove a server
router$remove_client("github")

# Cleanup
router$close()
}
```

Middleware *Middleware (Base Class)*

Description

Defines a middleware that can intercept and modify model operations.

Public fields

`name` A descriptive name for this middleware.

Methods

Public methods:

- `Middleware$transform_params()`
- `Middleware$wrap_generate()`
- `Middleware$wrap_stream()`
- `Middleware$clone()`

Method `transform_params()`: Transform parameters before calling the model.

Usage:

```
Middleware$transform_params(params, type, model)
```

Arguments:

`params` The original call parameters.

`type` Either "generate" or "stream".

`model` The model being called.

Returns: The transformed parameters.

Method `wrap_generate()`: Wrap the generate operation.

Usage:

```
Middleware$wrap_generate(do_generate, params, model)
```

Arguments:

`do_generate` A function that calls the model's do_generate.

`params` The (potentially transformed) parameters.

`model` The model being called.

Returns: The result of the generation.

Method `wrap_stream()`: Wrap the stream operation.

Usage:

```
Middleware$wrap_stream(do_stream, params, model, callback)
```

Arguments:

`do_stream` A function that calls the model's do_stream.

`params` The (potentially transformed) parameters.

`model` The model being called.

`callback` The streaming callback function.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`migrate_pattern` *Migrate Legacy Code*

Description

Provides migration guidance for legacy code patterns.

Usage

```
migrate_pattern(pattern)
```

Arguments

`pattern`	The legacy pattern to migrate from.

Value

A list with old_pattern, new_pattern, and example.

Examples

```
if (interactive()) {
# Get migration guidance for ChatSession
guidance <- migrate_pattern("ChatSession")
cat(guidance$example)
}
```

`Mission` *Mission Class*

Description

R6 class representing a persistent, goal-oriented execution mission. A Mission is the global leadership layer above Agent/AgentTeam/Flow.

Key capabilities:

- Full state machine: pending -> planning -> running -> succeeded/failed/stalled
- LLM-driven auto-planning: converts a goal string into ordered MissionSteps
- Step-level retry with exponential backoff + error-context injection
- DAG dependency resolution (depends_on)
- Parallel step execution (parallel = TRUE groups)
- Checkpoint persistence: save() / resume()
- Full audit log for post-mortem analysis
- MissionHookHandler integration for observability

Public fields

`id` Unique mission UUID.

`goal` Natural language goal description.

`steps` List of `MissionStep` objects.

`status` Mission status string.

`session` SharedSession used across all steps.

`model` Default model ID for this mission.

`stall_policy` Named list defining failure recovery behavior.

`hooks` MissionHookHandler for lifecycle events.

`audit_log` List of event records in chronological order.

`auto_plan` If TRUE and steps is NULL, use LLM to plan before running.

`default_executor` Default executor for auto-planned steps.

Methods

Public methods:

- `Mission$new()`
- `Mission$run()`
- `Mission$save()`
- `Mission$resume()`
- `Mission$step_summary()`
- `Mission$print()`
- `Mission$clone()`

Method `new()`**:** Initialize a new Mission.

Usage:

```
Mission$new(
  goal,
  steps = NULL,
  model = NULL,
  executor = NULL,
  stall_policy = NULL,
  hooks = NULL,
  session = NULL,
  auto_plan = TRUE
)
```

Arguments:

`goal` Natural language goal description.

`steps` Optional list of `MissionStep` objects. If NULL and auto_plan=TRUE, the LLM plans them.

`model` Default model ID (e.g., "anthropic:claude-opus-4-6").

`executor` Default executor for all steps (Agent, AgentTeam, Flow, or function). Used for auto-planned steps when no per-step executor is specified.

`stall_policy` Named list with on_tool_failure, on_step_timeout, on_max_retries, escalate_fn. Defaults to default_stall_policy().

`hooks` MissionHookHandler for lifecycle events.

`session` Optional SharedSession. Created automatically if NULL.

`auto_plan` If TRUE, call LLM to decompose goal into steps when steps is NULL.

Method `run()`**:** Run the Mission synchronously until completion or stall.

Usage:

```
Mission$run(model = NULL, ...)
```

Arguments:

`model` Optional model override. Falls back to self$model.

`...` Additional arguments (reserved for future use).

Returns: Invisible self (inspect $status, $steps, $audit_log for results).

Method `save()`**:** Save mission state to a file for later resumption.

Usage:

```
Mission$save(path)
```

Arguments:

`path` File path (.rds).

Method `resume()`**:** Resume a Mission from a saved checkpoint.

Usage:

```
Mission$resume(path)
```

Arguments:

`path` File path to a previously saved mission state (.rds).

Details: Steps that are already "done" are skipped. Pending/failed/retrying steps are re-executed. The executor must be re-attached via $set_executor() or by providing a default_executor at Mission creation.

Method `step_summary()`**:** Get a summary of step statuses.

Usage:

```
Mission$step_summary()
```

Returns: Named character vector: step_id -> status.

Method `print()`**:** Print method.

Usage:

```
Mission$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`MissionHookHandler` *MissionHookHandler Class*

Description

R6 class to manage Mission-level lifecycle hooks. Supported events span the full Mission state machine: planning -> step execution -> completion / stall / escalation.

Public fields

`hooks` Named list of hook functions.

Methods

Public methods:

- `MissionHookHandler$new()`
- `MissionHookHandler$trigger_mission_start()`
- `MissionHookHandler$trigger_mission_planned()`
- `MissionHookHandler$trigger_step_start()`
- `MissionHookHandler$trigger_step_done()`
- `MissionHookHandler$trigger_step_failed()`
- `MissionHookHandler$trigger_mission_stall()`
- `MissionHookHandler$trigger_mission_done()`
- `MissionHookHandler$clone()`

Method `new()`**:** Initialize a MissionHookHandler.

Usage:

```
MissionHookHandler$new(hooks_list = list())
```

Arguments:

`hooks_list` Named list of hook functions. Supported hooks:

- on_mission_start(mission) - Called when a Mission begins running.
- on_mission_planned(mission) - Called after LLM planning produces steps.
- on_step_start(step, attempt) - Called before each step attempt.
- on_step_done(step, result) - Called when a step succeeds.
- on_step_failed(step, error, attempt) - Called on each step failure.
- on_mission_stall(mission, step) - Called when a step exceeds max_retries.
- on_mission_done(mission) - Called when the Mission completes (succeeded or failed).

Method `trigger_mission_start()`**:** Trigger on_mission_start.

Usage:

```
MissionHookHandler$trigger_mission_start(mission)
```

Arguments:

`mission` The Mission object.

Method `trigger_mission_planned()`**:** Trigger on_mission_planned.

Usage:

```
MissionHookHandler$trigger_mission_planned(mission)
```

Arguments:

`mission` The Mission object (steps are now populated).

Method `trigger_step_start()`**:** Trigger on_step_start.

Usage:

```
MissionHookHandler$trigger_step_start(step, attempt)
```

Arguments:

`step` The MissionStep object.

`attempt` Integer attempt number (1 = first try).

Method `trigger_step_done()`**:** Trigger on_step_done.

Usage:

```
MissionHookHandler$trigger_step_done(step, result)
```

Arguments:

`step` The MissionStep object.

`result` The text result from the executor.

Method `trigger_step_failed()`**:** Trigger on_step_failed.

Usage:

```
MissionHookHandler$trigger_step_failed(step, error, attempt)
```

Arguments:

`step` The MissionStep object.

`error` The error message string.

`attempt` Integer attempt number.

Method `trigger_mission_stall()`**:** Trigger on_mission_stall.

Usage:

```
MissionHookHandler$trigger_mission_stall(mission, step)
```

Arguments:

`mission` The Mission object.

`step` The step that caused the stall.

Method `trigger_mission_done()`**:** Trigger on_mission_done.

Usage:

```
MissionHookHandler$trigger_mission_done(mission)
```

Arguments:

`mission` The completed Mission object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`MissionOrchestrator` *MissionOrchestrator Class*

Description

R6 class that manages a queue of Missions and executes them within a concurrency limit. Provides full observability via status summaries and stall detection through reconcile().

Public fields

`pending_queue` List of `Mission` objects waiting to run.

`running` List of `Mission` objects currently executing.

`completed` List of `Mission` objects that have finished.

`max_concurrent` Maximum simultaneous missions. Default 3.

`global_session` Optional SharedSession shared across all missions.

`global_model` Default model ID for missions that don't specify one.

`async_handles` list of callr::r_bg handles (for run_async).

Methods

Public methods:

- `MissionOrchestrator$new()`
- `MissionOrchestrator$submit()`
- `MissionOrchestrator$run_all()`
- `MissionOrchestrator$run_async()`
- `MissionOrchestrator$poll_async()`
- `MissionOrchestrator$reconcile()`
- `MissionOrchestrator$status()`
- `MissionOrchestrator$print()`
- `MissionOrchestrator$clone()`

Method `new()`**:** Initialize a new MissionOrchestrator.

Usage:

```
MissionOrchestrator$new(max_concurrent = 3, model = NULL, session = NULL)
```

Arguments:

`max_concurrent` Maximum simultaneous missions. Default 3.

`model` Optional default model for all missions.

`session` Optional shared SharedSession.

Method `submit()`**:** Submit a Mission to the orchestrator queue.

Usage:

```
MissionOrchestrator$submit(mission)
```

Arguments:

`mission` A Mission object.

Returns: Invisible self for chaining.

Method `run_all()`**:** Run all submitted missions respecting the concurrency limit.

Usage:

```
MissionOrchestrator$run_all(model = NULL)
```

Arguments:

`model` Optional model override for all missions in this run.

Details: Missions are executed in batches of max_concurrent. Within each batch, missions run in parallel (via parallel::mclapply on Unix, sequentially on Windows). Completed missions are moved to $completed.

Returns: Invisibly returns the list of completed `Mission` objects.

Method `run_async()`**:** Run a single Mission asynchronously in a background process.

Usage:

```
MissionOrchestrator$run_async(mission, model = NULL)
```

Arguments:

`mission` A Mission object.

`model` Optional model override.

Details: Uses callr::r_bg to launch the mission in a separate R process. The mission state is serialized to a temp file, executed, and the result is written back. Call $poll_async() to check completion.

Returns: A list with $handle (callr process), $mission_id, $checkpoint_path.

Method `poll_async()`**:** Poll all async handles and collect completed missions.

Usage:

```
MissionOrchestrator$poll_async()
```

Returns: Named list with `completed` (list of `Mission` objects) and `still_running` (integer count).

Method `reconcile()`**:** Stall detection: check for missions that appear stuck.

Usage:

```
MissionOrchestrator$reconcile(stall_threshold_secs = 600)
```

Arguments:

`stall_threshold_secs` Missions running longer than this are flagged. Default 600 (10 minutes).

Returns: list of stalled mission IDs.

Method `status()`**:** Get a status summary of all missions.

Usage:

```
MissionOrchestrator$status()
```

Returns: A data.frame with id, goal, status, n_steps columns.

Method `print()`**:** Print method.

Usage:

```
MissionOrchestrator$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`MissionStep` *MissionStep Class*

Description

A single execution unit within a Mission. Each step wraps an executor (Agent, AgentTeam, Flow, or plain R function) and handles its own retry loop with error-history injection.

Public fields

`id` Unique step identifier.

`description` Natural language description of what this step does.

`executor` Agent | AgentTeam | Flow | function to perform the step.

`status` Current status: "pending"|"running"|"done"|"failed"|"retrying".

`max_retries` Maximum retry attempts before escalation. Default 2.

`retry_count` Number of retries attempted so far.

`timeout_secs` Optional per-step timeout in seconds. NULL = no timeout.

`parallel` If TRUE, this step may run concurrently with other parallel steps.

`depends_on` Character vector of step IDs that must complete before this step.

`result` The text result from the executor on success.

`error_history` List of failure records, each containing `attempt`, `error`, and `timestamp`.

Methods

Public methods:

- `MissionStep$new()`
- `MissionStep$run()`
- `MissionStep$print()`
- `MissionStep$clone()`

Method `new()`: Initialize a MissionStep.

Usage:

```
MissionStep$new(
  id,
  description,
  executor = NULL,
  max_retries = 2,
  timeout_secs = NULL,
  parallel = FALSE,
  depends_on = NULL
)
```

Arguments:

`id` Unique step ID (e.g., "step_1").

`description` Natural language task description.

`executor` Agent, AgentTeam, Flow, or R function.

`max_retries` Maximum retries before stall escalation. Default 2.

`timeout_secs` Optional per-step timeout. Default NULL.

`parallel` Can run in parallel with other parallel steps. Default FALSE.

`depends_on` Character vector of prerequisite step IDs. Default NULL.

Method `run()`: Execute this step once (no retry logic; handled by Mission).

Usage:

```
MissionStep$run(session, model, context = NULL)
```

Arguments:

`session` A ChatSession for shared state.

`model` Model ID string.

`context` Optional error-injection context string.

Returns: Character string result, or stops with an error.

Method `print()`: Print method.

Usage:

```
MissionStep$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`mission_hooks` *Mission Hook System*

Description

Mission-level event hooks for observability and intervention. Provides a higher-level hook system above the agent-level HookHandler, enabling monitoring and control of the entire Mission lifecycle.

`mission_orchestrator` *Mission Orchestrator: Concurrent Mission Scheduling*

Description

MissionOrchestrator R6 class for managing multiple concurrent Missions. Implements the Coordinator layer from Symphony's architecture: poll loop, concurrency slots, stall detection, and result aggregation.

Concurrency model:

- Synchronous batch: run_all() executes up to max_concurrent missions simultaneously using parallel::mclapply (fork-based on Unix, sequential on Windows).
- Async per-mission: run_async() launches a mission in a callr background process and returns a handle for polling.

`model` *Model Shortcut*

Description

Shortcut for default model configuration. Call with no arguments to read the current default model, or pass a model to update it. This is equivalent to calling `get_model()` and `set_model()` directly.

Usage

```
model(new)
```

Arguments

`new`	Optional model identifier string or `LanguageModelV1` object.

Value

When `new` is missing, returns the current default model. Otherwise invisibly returns the previous default model.

Examples

```
model()
model("openai:gpt-4o-mini")
model(NULL)
```

model_defaults *Default Model Configuration*

Description

Utilities for reading and updating the package-wide default language model. High-level helpers that accept `model = NULL`, including `generate_text()`, `stream_text()`, `ChatSession$new()`, `create_chat_session()`, `auto_fix()`, and the knitr `{ai}` engine, use this default when no explicit model is supplied.

multimodal *Multimodal Helpers*

Description

Helper functions for constructing multimodal messages (text and images).

NvidiaProvider *NVIDIA Provider Class*

Description

Provider class for NVIDIA.

Super class

`aisdk::OpenAIProvider` -> `NvidiaProvider`

Methods

Public methods:

- `NvidiaProvider$new()`
- `NvidiaProvider$language_model()`
- `NvidiaProvider$clone()`

Method `new()`**:** Initialize the NVIDIA provider.

Usage:

```
NvidiaProvider$new(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments:

`api_key` NVIDIA API key. Defaults to NVIDIA_API_KEY env var.

`base_url` Base URL. Defaults to https://integrate.api.nvidia.com/v1.

`headers` Optional additional headers.

Method `language_model()`**:** Create a language model.

Usage:

```
NvidiaProvider$language_model(model_id = NULL)
```

Arguments:

`model_id` The model ID (e.g., "z-ai/glm4.7").

Returns: A NvidiaLanguageModel object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`ObjectStrategy` *Object Strategy*

Description

Object Strategy

Object Strategy

Details

Strategy for generating structured objects based on a JSON Schema. This strategy instructs the LLM to produce valid JSON matching the schema, and handles parsing and validation of the output.

Super class

`aisdk::OutputStrategy` -> `ObjectStrategy`

Public fields

`schema` The schema definition (from z_object, etc.).

`schema_name` Human-readable name for the schema.

Methods

Public methods:

- `ObjectStrategy$new()`
- `ObjectStrategy$get_instruction()`
- `ObjectStrategy$validate()`
- `ObjectStrategy$clone()`

Method `new()`**:** Initialize the ObjectStrategy.

Usage:

```
ObjectStrategy$new(schema, schema_name = "json_schema")
```

Arguments:

`schema` A schema object created by z_object, z_array, etc.

`schema_name` An optional name for the schema (default: "json_schema").

Method `get_instruction()`**:** Generate the instruction for the LLM to output valid JSON.

Usage:

```
ObjectStrategy$get_instruction()
```

Returns: A character string with the prompt instruction.

Method `validate()`**:** Validate and parse the LLM output as JSON.

Usage:

```
ObjectStrategy$validate(text, is_final = FALSE)
```

Arguments:

`text` The raw text output from the LLM.

`is_final` Logical, TRUE if this is the final output.

Returns: The parsed R object (list), or NULL if parsing fails.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`OpenAIProvider` *OpenAI Provider Class*

Description

Provider class for OpenAI. Can create language and embedding models.

Public fields

`specification_version` Provider spec version.

Methods

Public methods:

- `OpenAIProvider$new()`
- `OpenAIProvider$language_model()`
- `OpenAIProvider$responses_model()`
- `OpenAIProvider$smart_model()`
- `OpenAIProvider$embedding_model()`
- `OpenAIProvider$clone()`

Method `new()`**:** Initialize the OpenAI provider.

Usage:

```
OpenAIProvider$new(
  api_key = NULL,
  base_url = NULL,
  organization = NULL,
  project = NULL,
  headers = NULL,
  name = NULL,
  disable_stream_options = FALSE
)
```

Arguments:

`api_key` OpenAI API key. Defaults to OPENAI_API_KEY env var.

`base_url` Base URL for API calls. Defaults to https://api.openai.com/v1.

`organization` Optional OpenAI organization ID.

`project` Optional OpenAI project ID.

`headers` Optional additional headers.

`name` Optional provider name override (for compatible APIs).

`disable_stream_options` Disable stream_options parameter (for providers that don't support it).

Method `language_model()`**:** Create a language model.

Usage:

```
OpenAIProvider$language_model(model_id = Sys.getenv("OPENAI_MODEL", "gpt-4o"))
```

Arguments:

`model_id` The model ID (e.g., "gpt-4o", "gpt-4o-mini").

Returns: An OpenAILanguageModel object.

Method `responses_model()`**:** Create a language model using the Responses API.

Usage:

```
OpenAIProvider$responses_model(model_id)
```

Arguments:

`model_id` The model ID (e.g., "o1", "o3-mini", "gpt-4o").

Details: The Responses API is designed for:

- Models with built-in reasoning (o1, o3 series)
- Stateful multi-turn conversations (server maintains history)
- Advanced features like structured outputs

The model maintains conversation state internally via response IDs. Call `model$reset()` to start a fresh conversation.

Returns: An OpenAIResponsesLanguageModel object.

Method `smart_model()`**:** Smart model factory that automatically selects the best API.

Usage:

```
OpenAIProvider$smart_model(
  model_id,
  api_format = c("auto", "chat", "responses")
)
```

Arguments:

`model_id` The model ID.

`api_format` API format to use: "auto" (default), "chat", or "responses".

Details: When `api_format = "auto"` (default), the method automatically selects:

- Responses API for reasoning models (o1, o3, o1-mini, o3-mini)
- Chat Completions API for all other models (gpt-4o, gpt-4, etc.)

You can override this by explicitly setting `api_format`.

Returns: A language model object (either OpenAILanguageModel or OpenAIResponsesLanguageModel).

Method `embedding_model()`**:** Create an embedding model.

Usage:

```
OpenAIProvider$embedding_model(model_id = "text-embedding-3-small")
```

Arguments:

`model_id` The model ID (e.g., "text-embedding-3-small").

Returns: An OpenAIEmbeddingModel object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`OpenRouterProvider` *OpenRouter Provider Class*

Description

Provider class for OpenRouter.

Super class

`aisdk::OpenAIProvider` -> `OpenRouterProvider`

Methods

Public methods:

- `OpenRouterProvider$new()`
- `OpenRouterProvider$language_model()`
- `OpenRouterProvider$clone()`

Method `new()`**:** Initialize the OpenRouter provider.

Usage:

```
OpenRouterProvider$new(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments:

`api_key` OpenRouter API key. Defaults to OPENROUTER_API_KEY env var.

`base_url` Base URL. Defaults to https://openrouter.ai/api/v1.

`headers` Optional additional headers.

Method `language_model()`**:** Create a language model.

Usage:

```
OpenRouterProvider$language_model(model_id = NULL)
```

Arguments:

`model_id` The model ID (e.g., "openai/gpt-4o", "anthropic/claude-sonnet-4-20250514", "deepseek/deepseek-r1", "google/gemini-2.5-pro").

Returns: An OpenRouterLanguageModel object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

OutputStrategy *Output Strategy Interface*

Description

Output Strategy Interface

Output Strategy Interface

Details

Abstract R6 class defining the interface for output strategies. Subclasses must implement `get_instruction()` and `validate()`.

Methods

Public methods:

- `OutputStrategy$new()`
- `OutputStrategy$get_instruction()`
- `OutputStrategy$validate()`
- `OutputStrategy$clone()`

Method `new()`**:** Initialize the strategy.

Usage:

```
OutputStrategy$new()
```

Method `get_instruction()`**:** Get the system prompt instruction for this strategy.

Usage:

```
OutputStrategy$get_instruction()
```

Returns: A character string with instructions for the LLM.

Method `validate()`**:** Parse and validate the output text.

Usage:

```
OutputStrategy$validate(text, is_final = FALSE)
```

Arguments:

`text` The raw text output from the LLM.

`is_final` Logical, TRUE if this is the final output (not streaming).

Returns: The parsed and validated object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`package_skill` *Package a Skill*

Description

Validates a skill and packages it into a `.skill` zip file.

Usage

```
package_skill(path, output_dir = tempdir())
```

Arguments

`path`	Path to the skill directory.
`output_dir`	Directory to save the packaged file. Defaults to `tempdir()`.

Value

Path to the created .skill file.

`print.benchmark_result` *Print Benchmark Result*

Description

Print Benchmark Result

Usage

```
## S3 method for class 'benchmark_result'
print(x, ...)
```

Arguments

`x`	A benchmark result object.
`...`	Additional arguments (not used).

`print.GenerateObjectResult` *Print GenerateObjectResult*

Description

Print GenerateObjectResult

Usage

```
## S3 method for class 'GenerateObjectResult'
print(x, ...)
```

Arguments

`x`	A GenerateObjectResult object.
`...`	Additional arguments (ignored).

`print.z_schema` *Print Method for z_schema*

Description

Pretty print a z_schema object.

Usage

```
## S3 method for class 'z_schema'
print(x, ...)
```

Arguments

`x`	A z_schema object.
`...`	Additional arguments (ignored).

`print_migration_guide` *Print Migration Guide*

Description

Print a comprehensive migration guide for upgrading to the new SDK version.

Usage

```
print_migration_guide(verbose = TRUE)
```

Arguments

`verbose`	Include detailed examples. Default TRUE.

Value

Invisible NULL (prints to console).

`ProjectMemory` *Project Memory Class*

Description

R6 class for managing persistent project memory using SQLite. Stores code snippets, error fixes, and execution graphs for resuming failed long-running jobs.

Public fields

`db_path` Path to the SQLite database file.

`project_root` Root directory of the project.

Methods

Public methods:

- `ProjectMemory$new()`
- `ProjectMemory$store_snippet()`
- `ProjectMemory$store_fix()`
- `ProjectMemory$find_similar_fix()`
- `ProjectMemory$search_snippets()`
- `ProjectMemory$store_workflow_node()`
- `ProjectMemory$update_node_status()`
- `ProjectMemory$get_workflow()`
- `ProjectMemory$get_resumable_nodes()`

- `ProjectMemory$store_conversation()`
- `ProjectMemory$get_conversation()`
- `ProjectMemory$store_review()`
- `ProjectMemory$store_review_artifact()`
- `ProjectMemory$get_review()`
- `ProjectMemory$get_review_artifact()`
- `ProjectMemory$get_review_runtime_record()`
- `ProjectMemory$get_reviews_for_file()`
- `ProjectMemory$record_review_saveback()`
- `ProjectMemory$update_execution_result()`
- `ProjectMemory$append_review_event()`
- `ProjectMemory$update_review_status()`
- `ProjectMemory$get_pending_reviews()`
- `ProjectMemory$stats()`
- `ProjectMemory$clear()`
- `ProjectMemory$print()`
- `ProjectMemory$clone()`

Method `new()`**:** Create or connect to a project memory database.

Usage:

```
ProjectMemory$new(project_root = tempdir(), db_name = "memory.sqlite")
```

Arguments:

`project_root` Project root directory. Defaults to `tempdir()`.

`db_name` Database filename. Defaults to "memory.sqlite".

Returns: A new ProjectMemory object.

Method `store_snippet()`**:** Store a successful code snippet for future reference.

Usage:

```
ProjectMemory$store_snippet(
  code,
  description = NULL,
  tags = NULL,
  context = NULL
)
```

Arguments:

`code` The R code that was executed successfully.

`description` Optional description of what the code does.

`tags` Optional character vector of tags for categorization.

`context` Optional context about when/why this code was used.

Returns: The ID of the stored snippet.

Method `store_fix()`**:** Store an error fix for learning.

Usage:

```
ProjectMemory$store_fix(
  original_code,
  error,
  fixed_code,
  fix_description = NULL
)
```

Arguments:

`original_code` The code that produced the error.

`error` The error message.

`fixed_code` The corrected code.

`fix_description` Description of what was fixed.

Returns: The ID of the stored fix.

Method `find_similar_fix()`**:** Find a similar fix from memory.

Usage:

```
ProjectMemory$find_similar_fix(error)
```

Arguments:

`error` The error message to match.

Returns: A list with the fix details, or NULL if not found.

Method `search_snippets()`**:** Search for relevant code snippets.

Usage:

```
ProjectMemory$search_snippets(query, limit = 10)
```

Arguments:

`query` Search query (matches description, tags, or code).

`limit` Maximum number of results.

Returns: A data frame of matching snippets.

Method `store_workflow_node()`**:** Store execution graph node for workflow persistence.

Usage:

```
ProjectMemory$store_workflow_node(
  workflow_id,
  node_id,
  node_type,
  code,
  status = "pending",
  result = NULL,
  dependencies = NULL
)
```

Arguments:

`workflow_id` Unique identifier for the workflow.

`node_id` Unique identifier for this node.

`node_type` Type of node (e.g., "transform", "model", "output").

`code` The code for this node.
`status` Node status ("pending", "running", "completed", "failed").
`result` Optional serialized result.
`dependencies` Character vector of node IDs this depends on.

Returns: The database row ID.

Method `update_node_status():` Update workflow node status.

Usage:

```
ProjectMemory$update_node_status(workflow_id, node_id, status, result = NULL)
```

Arguments:

`workflow_id` Workflow identifier.
`node_id` Node identifier.
`status` New status.
`result` Optional result to store.

Method `get_workflow():` Get workflow state for resuming.

Usage:

```
ProjectMemory$get_workflow(workflow_id)
```

Arguments:

`workflow_id` Workflow identifier.

Returns: A list with workflow nodes and their states.

Method `get_resumable_nodes():` Resume a failed workflow from the last successful point.

Usage:

```
ProjectMemory$get_resumable_nodes(workflow_id)
```

Arguments:

`workflow_id` Workflow identifier.

Returns: List of node IDs that need to be re-executed.

Method `store_conversation():` Store a conversation turn for context.

Usage:

```
ProjectMemory$store_conversation(session_id, role, content, metadata = NULL)
```

Arguments:

`session_id` Session identifier.
`role` Message role ("user", "assistant", "system").
`content` Message content.
`metadata` Optional metadata list.

Method `get_conversation():` Get conversation history for a session.

Usage:

```
ProjectMemory$get_conversation(session_id, limit = 100)
```

Arguments:

`session_id` Session identifier.

`limit` Maximum number of messages.

Returns: A data frame of conversation messages.

Method `store_review()`**:** Store or update a human review for an AI-generated chunk.

Usage:

```
ProjectMemory$store_review(
  chunk_id,
  file_path,
  chunk_label,
  prompt,
  response,
  status = "pending",
  ai_agent = NULL,
  uncertainty = NULL,
  session_id = NULL,
  review_mode = NULL,
  runtime_mode = NULL,
  artifact_json = NULL,
  execution_status = NULL,
  execution_output = NULL,
  final_code = NULL,
  error_message = NULL
)
```

Arguments:

`chunk_id` Unique identifier for the chunk.

`file_path` Path to the source file.

`chunk_label` Chunk label from knitr.

`prompt` The prompt sent to the AI.

`response` The AI's response.

`status` Review status ("pending", "approved", "rejected").

`ai_agent` Optional agent name.

`uncertainty` Optional uncertainty level.

`session_id` Optional session identifier for transcript/provenance.

`review_mode` Optional normalized review mode.

`runtime_mode` Optional normalized runtime mode.

`artifact_json` Optional JSON review artifact payload.

`execution_status` Optional execution state.

`execution_output` Optional execution output text.

`final_code` Optional finalized executable code.

`error_message` Optional execution or generation error.

Returns: The database row ID.

Method `store_review_artifact()`**:** Store structured review artifact metadata for a chunk.

Usage:

```
ProjectMemory$store_review_artifact(
  chunk_id,
  artifact,
  session_id = NULL,
  review_mode = NULL,
  runtime_mode = NULL
)
```

Arguments:

`chunk_id` Chunk identifier.

`artifact` A serializable list representing the review artifact.

`session_id` Optional session identifier.

`review_mode` Optional normalized review mode.

`runtime_mode` Optional normalized runtime mode.

Returns: Invisible TRUE.

Method `get_review()`**:** Get a review by chunk ID.

Usage:

```
ProjectMemory$get_review(chunk_id)
```

Arguments:

`chunk_id` Chunk identifier.

Returns: A list with review details, or NULL if not found.

Method `get_review_artifact()`**:** Get a parsed review artifact by chunk ID.

Usage:

```
ProjectMemory$get_review_artifact(chunk_id)
```

Arguments:

`chunk_id` Chunk identifier.

Returns: A list artifact, or NULL if none is stored.

Method `get_review_runtime_record()`**:** Get a review together with its parsed artifact.

Usage:

```
ProjectMemory$get_review_runtime_record(chunk_id)
```

Arguments:

`chunk_id` Chunk identifier.

Returns: A list with `review` and `artifact`, or NULL if not found.

Method `get_reviews_for_file()`**:** Get all reviews for a given source file.

Usage:

```
ProjectMemory$get_reviews_for_file(file_path)
```

Arguments:

`file_path` Source document path.

Returns: A data frame of reviews ordered by updated time.

Method `record_review_saveback()`**:** Record a saveback lifecycle event for one or more chunk reviews.

Usage:

```
ProjectMemory$record_review_saveback(
  chunk_ids,
  source_path,
  html_path = NULL,
  status = "requested",
  rerendered = FALSE,
  message = NULL
)
```

Arguments:

`chunk_ids` Character vector of chunk identifiers.

`source_path` Source document path.

`html_path` Optional rendered HTML path.

`status` Saveback status string.

`rerendered` Whether a rerender occurred.

`message` Optional message.

Returns: Invisible TRUE.

Method `update_execution_result()`**:** Update execution result fields for a chunk review.

Usage:

```
ProjectMemory$update_execution_result(
  chunk_id,
  execution_status,
  execution_output = NULL,
  final_code = NULL,
  error_message = NULL
)
```

Arguments:

`chunk_id` Chunk identifier.

`execution_status` Execution state string.

`execution_output` Optional execution output text.

`final_code` Optional finalized executable code.

`error_message` Optional execution error.

Returns: Invisible TRUE.

Method `append_review_event()`**:** Append an audit event for a reviewed chunk.

Usage:

```
ProjectMemory$append_review_event(chunk_id, event_type, payload = NULL)
```

Arguments:

`chunk_id` Chunk identifier.

`event_type` Event type string.
`payload` Optional serializable payload list.

Returns: The database row ID.

Method `update_review_status():` Update review status.

Usage:

```
ProjectMemory$update_review_status(chunk_id, status)
```

Arguments:

`chunk_id` Chunk identifier.
`status` New status ("approved" or "rejected").

Method `get_pending_reviews():` Get pending reviews, optionally filtered by file.

Usage:

```
ProjectMemory$get_pending_reviews(file_path = NULL)
```

Arguments:

`file_path` Optional file path filter.

Returns: A data frame of pending reviews.

Method `stats():` Get memory statistics.

Usage:

```
ProjectMemory$stats()
```

Returns: A list with counts and sizes.

Method `clear():` Clear all memory (use with caution).

Usage:

```
ProjectMemory$clear(confirm = FALSE)
```

Arguments:

`confirm` Must be TRUE to proceed.

Method `print():` Print method for ProjectMemory.

Usage:

```
ProjectMemory$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

project_memory *Project Memory System*

Description

Long-term memory storage for AI agents using SQLite. Stores successful code snippets, error fixes, and execution history for RAG (Retrieval Augmented Generation) and learning from past interactions.

Factory function to create or connect to a project memory database.

Usage

```
project_memory(project_root = tempdir(), db_name = "memory.sqlite")
```

Arguments

`project_root`	Project root directory.
`db_name`	Database filename.

Value

A ProjectMemory object.

Examples

```
if (interactive()) {
# Create memory for current project
memory <- project_memory()

# Store a successful code snippet
memory$store_snippet(
  code = "df %>% filter(x > 0) %>% summarize(mean = mean(y))",
  description = "Filter and summarize data",
  tags = c("dplyr", "summarize")
)

# Store an error fix
memory$store_fix(
  original_code = "mean(df$x)",
  error = "argument is not numeric or logical",
  fixed_code = "mean(as.numeric(df$x), na.rm = TRUE)",
  fix_description = "Convert to numeric and handle NAs"
)

# Search for relevant snippets
memory$search_snippets("summarize")
}
```

`ProviderRegistry` *Provider Registry*

Description

Manages registered providers and allows accessing models by ID.

Methods

Public methods:

- `ProviderRegistry$new()`
- `ProviderRegistry$register()`
- `ProviderRegistry$language_model()`
- `ProviderRegistry$embedding_model()`
- `ProviderRegistry$list_providers()`
- `ProviderRegistry$clone()`

Method `new()`**:** Initialize the registry.

Usage:

```
ProviderRegistry$new(separator = ":")
```

Arguments:

`separator` The separator between provider and model IDs (default: ":").

Method `register()`**:** Register a provider.

Usage:

```
ProviderRegistry$register(id, provider)
```

Arguments:

`id` The provider ID (e.g., "openai").

`provider` The provider object (must have `language_model` method).

Method `language_model()`**:** Get a language model by ID.

Usage:

```
ProviderRegistry$language_model(id)
```

Arguments:

`id` Model ID in the format "provider:model" (e.g., "openai:gpt-4o").

Returns: A LanguageModelV1 object.

Method `embedding_model()`**:** Get an embedding model by ID.

Usage:

```
ProviderRegistry$embedding_model(id)
```

Arguments:

`id` Model ID in the format "provider:model".

Returns: An EmbeddingModelV1 object.

Method `list_providers()`**:** List all registered provider IDs.

Usage:

```
ProviderRegistry$list_providers()
```

Returns: A character vector of provider IDs.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`provider_custom` *Custom Provider Factory*

Description

A dynamic factory for creating custom provider instances. This allows users to instantiate a model provider at runtime by configuring the endpoint (`base_url`), credentials (`api_key`), network protocol/routing (`api_format`), and specific capabilities (`use_max_completion_tokens`), without writing a new Provider class.

`reactive_tool` *Reactive Tool*

Description

Create a tool that can modify Shiny reactive values. This is a wrapper around the standard `tool()` function that provides additional documentation and conventions for Shiny integration.

The execute function receives `rv` (reactiveValues) and `session` as the first two arguments, followed by any tool-specific parameters.

Usage

```
reactive_tool(name, description, parameters, execute)
```

Arguments

`name`	The name of the tool.
`description`	A description of what the tool does.
`parameters`	A schema object defining the tool's parameters.
`execute`	A function to execute. First two args are `rv` and `session`.

Value

A Tool object ready for use with aiChatServer.

Examples

```
if (interactive()) {
# Create a tool that modifies a reactive value
update_resolution_tool <- reactive_tool(
  name = "update_resolution",
  description = "Update the plot resolution",
  parameters = z_object(
    resolution = z_number() |> z_describe("New resolution value (50-500)")
  ),
  execute = function(rv, session, resolution) {
    rv$resolution <- resolution
    paste0("Resolution updated to ", resolution)
  }
)

# Use with aiChatServer by wrapping the execute function
server <- function(input, output, session) {
  rv <- reactiveValues(resolution = 100)

  # Wrap the tool to inject rv and session
  wrapped_tools <- wrap_reactive_tools(
    list(update_resolution_tool),
    rv = rv,
    session = session
  )

  aiChatServer("chat", model = "openai:gpt-4o", tools = wrapped_tools)
}
}
```

`register_ai_engine` *Register AI Engine*

Description

Registers the `{ai}` engine with knitr. Call this function once before knitting a document that uses `{ai}` chunks.

Usage

```
register_ai_engine()
```

Value

Invisible NULL.

Examples

```
if (interactive()) {
  library(aisdk)
  register_ai_engine()
  # Now you can use ```{ai} chunks in your RMarkdown
}
```

`reload_env` *Reload project-level environment variables*

Description

Forces R to re-read the .Renviron file without restarting the session. This is useful when you've modified .Renviron and don't want to restart R.

Usage

```
reload_env(path = ".Renviron")
```

Arguments

`path`	Path to .Renviron file (default: project root)

Value

Invisible TRUE if successful

Examples

```
if (interactive()) {
# Reload environment after modifying .Renviron
reload_env()
# Now use the new keys
Sys.getenv("OPENAI_API_KEY")
}
```

render_text *Render Markdown Text*

Description

Render markdown-formatted text in the console with beautiful styling. This function uses the same rendering engine as the streaming output, supporting headers, lists, code blocks, and other markdown elements.

Usage

```
render_text(text)
```

Arguments

`text`	A character string containing markdown text, or a GenerateResult object.

Value

NULL (invisibly)

Examples

```
if (interactive()) {
# Render simple text
render_text("# Hello\n\nThis is **bold** text.")

# Render with code block
render_text("Here is some R code:\n\n```r\nx <- 1:10\nmean(x)\n```")
}
```

request_authorization *Request User Authorization (HITL)*

Description

Pauses execution and prompts the user for permission to execute a potentially risky action. Supports console environments via `readline`.

Usage

```
request_authorization(action, risk_level = "YELLOW")
```

Arguments

`action`	Character string describing the action the Agent wants to perform.
`risk_level`	Character string. One of "GREEN", "YELLOW", "RED".

Value

Logical TRUE if user permits, otherwise throws an error with the rejection reason.

`run_feishu_webhook_server` *Run a Feishu Webhook Server*

Description

Run a blocking `httpuv` loop for a Feishu callback endpoint. This helper is intended for local demos and manual integration testing.

Usage

```
run_feishu_webhook_server(
  runtime,
  host = "127.0.0.1",
  port = 8788,
  path = "/feishu/webhook",
  poll_ms = 100
)
```

Arguments

`runtime`	A `ChannelRuntime` with a Feishu adapter registered.
`host`	Bind host.
`port`	Bind port.
`path`	Callback path.
`poll_ms`	Event loop polling interval in milliseconds.

Value

Invisible server handle. Interrupt the R process to stop it.

`r_data_tasks` *Create R Data Tasks Benchmark*

Description

Create a standard benchmark suite for R data science tasks.

Usage

```
r_data_tasks(difficulty = "medium")
```

Arguments

`difficulty`	Difficulty level: "easy", "medium", "hard".

Value

A list of benchmark tasks.

`safe_eval` *Safe Eval with Timeout*

Description

Execute R code with a timeout to prevent infinite loops.

Usage

```
safe_eval(expr, timeout_seconds = 30, envir = parent.frame())
```

Arguments

`expr`	Expression to evaluate.
`timeout_seconds`	Maximum execution time in seconds.
`envir`	Environment for evaluation.

Value

The result or an error.

`safe_parse_json` *Safe JSON Parser*

Description

Parses a JSON string, attempting to repair it using `fix_json` if the initial parse fails.

Usage

```
safe_parse_json(text)
```

Arguments

`text`	A JSON string.

Value

A parsed R object (list, vector, etc.) or NULL if parsing fails even after repair.

Examples

```
safe_parse_json('{"a": 1}')
safe_parse_json('{"a": 1,')
```

`safe_to_json` *Safe Serialization to JSON*

Description

Standardized internal helper for JSON serialization with common defaults.

Usage

```
safe_to_json(x, auto_unbox = TRUE, ...)
```

Arguments

`x`	Object to serialize.
`auto_unbox`	Whether to automatically unbox single-element vectors. Default TRUE.
`...`	Additional arguments to jsonlite::toJSON.

Value

A JSON string.

sandbox *R-Native Programmatic Sandbox*

Description

SandboxManager R6 class and utilities for building an R-native programmatic tool sandbox. Inspired by Anthropic's Programmatic Tool Calling, this module enables LLMs to write R code that batch-invokes registered tools and processes data locally (using dplyr/purrr), returning only concise results to the context.

Details

The core idea: instead of the LLM making N separate tool calls (each requiring a round-trip), it writes a single R script that loops over inputs, calls tools as ordinary R functions, filters/aggregates the results with dplyr, and `print()`s only the key findings. This dramatically reduces token usage, latency, and context window pressure.

Architecture:

```
User tools -> SandboxManager -> isolated R environment
  - tool_a()
  - tool_b()
  - dplyr::*
  - purrr::*

create_r_code_tool() -> single "execute_r_code" tool
  (registered with the LLM)
```

SandboxManager *SandboxManager Class*

Description

R6 class that manages an isolated R environment for executing LLM-generated R code. Tools are bound as callable functions within this environment, enabling the LLM to batch-invoke and process data locally.

Methods

Public methods:

- `SandboxManager$new()`
- `SandboxManager$bind_tools()`
- `SandboxManager$execute()`
- `SandboxManager$get_tool_signatures()`
- `SandboxManager$get_env()`

- `SandboxManager$list_tools()`
- `SandboxManager$reset()`
- `SandboxManager$print()`
- `SandboxManager$clone()`

Method `new()`**:** Initialize a new SandboxManager.

Usage:

```
SandboxManager$new(
  tools = list(),
  preload_packages = c("dplyr", "purrr"),
  max_output_chars = 8000,
  parent_env = NULL
)
```

Arguments:

`tools` Optional list of Tool objects to bind into the sandbox.

`preload_packages` Character vector of package names to preload into the sandbox (their exports become available). Default: c("dplyr", "purrr").

`max_output_chars` Maximum characters to capture from code output. Prevents runaway `print()` from flooding the context. Default: 8000.

`parent_env` Optional parent environment for the sandbox. When a ChatSession is available, pass `session$get_envir()` here to enable cross-step variable persistence.

Method `bind_tools()`**:** Bind Tool objects into the sandbox as callable R functions.

Usage:

```
SandboxManager$bind_tools(tools)
```

Arguments:

`tools` A list of Tool objects to bind.

Returns: Invisible self (for chaining).

Method `execute()`**:** Execute R code in the sandbox environment.

Usage:

```
SandboxManager$execute(code_str)
```

Arguments:

`code_str` A character string containing R code to execute.

Returns: A character string with captured stdout, or an error message.

Method `get_tool_signatures()`**:** Get human-readable signatures for all bound tools.

Usage:

```
SandboxManager$get_tool_signatures()
```

Returns: A character string with Markdown-formatted tool documentation.

Method `get_env()`**:** Get the sandbox environment.

Usage:

`SandboxManager$get_env()`

Returns: The R environment used by the sandbox.

Method `list_tools()`**:** Get list of bound tool names.

Usage:

`SandboxManager$list_tools()`

Returns: Character vector of tool names available in the sandbox.

Method `reset()`**:** Reset the sandbox environment (clear all user variables). Tool bindings and preloaded packages are preserved.

Usage:

`SandboxManager$reset()`

Method `print()`**:** Print method for SandboxManager.

Usage:

`SandboxManager$print()`

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

Usage:

`SandboxManager$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

`scan_skills` *Scan for Skills*

Description

Convenience function to scan a directory and return a SkillRegistry. Alias for create_skill_registry().

Usage

```
scan_skills(path, recursive = FALSE)
```

Arguments

`path`	Path to scan for skills.
`recursive`	Whether to scan subdirectories. Default FALSE.

Value

A populated SkillRegistry object.

`schema` *Schema DSL: Lightweight JSON Schema Generator*

Description

A lightweight DSL (Domain Specific Language) for defining JSON Schema structures in R, inspired by Zod from TypeScript. Used for defining tool parameters.

`schema_generator` *Schema Generator*

Description

Utilities to automatically generate z_schema objects from R function signatures.

`schema_to_json` *Convert Schema to JSON*

Description

Convert a z_schema object to a JSON string suitable for API calls. Handles the R-specific auto_unbox issues properly.

Usage

```
schema_to_json(schema, pretty = FALSE)
```

Arguments

`schema`	A z_schema object created by z_* functions.
`pretty`	If TRUE, format JSON with indentation.

Value

A JSON string.

Examples

```
schema <- z_object(
  name = z_string(description = "User name")
)
cat(schema_to_json(schema, pretty = TRUE))
```

`sdk_clear_protected_vars` *Reset the Variable Registry*

Description

Clears all protected variables.

Usage

```
sdk_clear_protected_vars()
```

`sdk_feature` *Get Feature Flag*

Description

Get the current value of a feature flag.

Usage

```
sdk_feature(flag, default = NULL)
```

Arguments

`flag`	Name of the feature flag.
`default`	Default value if flag not set.

Value

The flag value.

Examples

```
if (interactive()) {
# Check if shared session is enabled
if (sdk_feature("use_shared_session")) {
  session <- create_shared_session(model = "openai:gpt-4o")
}
} else {
  session <- create_chat_session(model = "openai:gpt-4o")
}
```

`sdk_get_var_metadata` *Get Metadata for a Protected Variable*

Description

Get Metadata for a Protected Variable

Usage

```
sdk_get_var_metadata(name)
```

Arguments

`name`	Character string. The name of the variable.

Value

A list with metadata (locked, cost, etc.), or NULL if not protected.

`sdk_is_var_locked` *Check if a Variable is Locked*

Description

Check if a Variable is Locked

Usage

```
sdk_is_var_locked(name)
```

Arguments

`name`	Character string. The name of the variable.

Value

TRUE if the variable is protected and locked, FALSE otherwise.

`sdk_list_features` *List Feature Flags*

Description

List all available feature flags and their current values.

Usage

```
sdk_list_features()
```

Value

A named list of feature flags.

Examples

```
if (interactive()) {
# See all feature flags
print(sdk_list_features())
}
```

`sdk_protect_var` *Protect a Variable from Agent Modification*

Description

Marks a variable as protected so that the Agent cannot accidentally overwrite, shadow, or deeply copy it during sandbox execution.

Usage

```
sdk_protect_var(name, locked = TRUE, cost = "High")
```

Arguments

`name`	Character string. The name of the variable to protect.
`locked`	Logical. If TRUE, the variable cannot be assigned to by the Agent.
`cost`	Character string. An indicator of the variable's computation/memory cost (e.g., "High", "Medium", "Low").

Value

Invisible TRUE.

`sdk_reset_features` *Reset Feature Flags*

Description

Reset all feature flags to their default values.

Usage

```
sdk_reset_features()
```

Value

Invisible NULL.

`sdk_set_feature` *Set Feature Flag*

Description

Set a feature flag value. Use this to enable/disable SDK features.

Usage

```
sdk_set_feature(flag, value)
```

Arguments

`flag`	Name of the feature flag.
`value`	Value to set.

Value

Invisible previous value.

Examples

```
if (interactive()) {
# Disable shared session for legacy compatibility
sdk_set_feature("use_shared_session", FALSE)

# Enable legacy tool format
sdk_set_feature("legacy_tool_format", TRUE)
}
```

sdk_unprotect_var *Unprotect a Variable*

Description

Removes protection from a previously protected variable.

Usage

```
sdk_unprotect_var(name)
```

Arguments

`name`	Character string. The name of the variable to unprotect.

Value

Invisible TRUE.

search_skills *Search Skills*

Description

Search the skill registry.

Usage

```
search_skills(query = NULL, capability = NULL)
```

Arguments

`query`	Search query.
`capability`	Filter by capability.

Value

A data frame of matching skills.

session *Session Management: Stateful Chat Sessions*

Description

ChatSession R6 class for managing stateful conversations with LLMs. Provides automatic history management, persistence, and model switching.

`set_model` *Set Default Model*

Description

Sets the package-wide default language model. Pass `NULL` to restore the built-in default (`"openai:gpt-4o"` unless overridden with `options(aisdk.default_model = ...)`).

Usage

```
set_model(new = NULL)
```

Arguments

`new`	A model identifier string, a `LanguageModelV1` object, or `NULL`.

Value

Invisibly returns the previous default model.

Examples

```
old <- set_model("deepseek:deepseek-chat")
current <- get_model()
set_model(old)
set_model(NULL)
```

`SharedSession` *SharedSession Class*

Description

R6 class representing an enhanced session for multi-agent systems. Extends ChatSession with:

- Execution context tracking (call stack, delegation history)
- Sandboxed code execution with safety guardrails
- Variable scoping and access control
- Comprehensive tracing and observability

Super class

`aisdk::ChatSession` -> `SharedSession`

Methods

Public methods:

- SharedSession$new()
- SharedSession$push_context()
- SharedSession$pop_context()
- SharedSession$get_context()
- SharedSession$set_global_task()
- SharedSession$execute_code()
- SharedSession$get_var()
- SharedSession$set_var()
- SharedSession$list_vars()
- SharedSession$summarize_vars()
- SharedSession$create_scope()
- SharedSession$delete_scope()
- SharedSession$trace_event()
- SharedSession$get_trace()
- SharedSession$clear_trace()
- SharedSession$trace_summary()
- SharedSession$set_access_control()
- SharedSession$check_permission()
- SharedSession$get_sandbox_mode()
- SharedSession$set_sandbox_mode()
- SharedSession$print()
- SharedSession$clone()

Method `new()`**:** Initialize a new SharedSession.

Usage:

```
SharedSession$new(
  model = NULL,
  system_prompt = NULL,
  tools = NULL,
  hooks = NULL,
  max_steps = 10,
  registry = NULL,
  sandbox_mode = "strict",
  trace_enabled = TRUE
)
```

Arguments:

`model` A LanguageModelV1 object or model string ID.

`system_prompt` Optional system prompt for the conversation.

`tools` Optional list of Tool objects.

`hooks` Optional HookHandler object.

`max_steps` Maximum steps for tool execution loops. Default 10.

`registry` Optional ProviderRegistry for model resolution.

`sandbox_mode` Sandbox mode: "strict", "permissive", or "none". Default "strict".

`trace_enabled` Enable execution tracing. Default TRUE.

Method `push_context()`**:** Push an agent onto the execution stack.

Usage:

```
SharedSession$push_context(agent_name, task, parent_agent = NULL)
```

Arguments:

`agent_name` Name of the agent being activated.

`task` The task being delegated.

`parent_agent` Name of the delegating agent (or NULL for root).

Returns: Invisible self for chaining.

Method `pop_context()`**:** Pop the current agent from the execution stack.

Usage:

```
SharedSession$pop_context(result = NULL)
```

Arguments:

`result` Optional result from the completed agent.

Returns: The popped context, or NULL if stack was empty.

Method `get_context()`**:** Get the current execution context.

Usage:

```
SharedSession$get_context()
```

Returns: A list with current_agent, depth, and delegation_stack.

Method `set_global_task()`**:** Set the global task (user's original request).

Usage:

```
SharedSession$set_global_task(task)
```

Arguments:

`task` The global task description.

Returns: Invisible self for chaining.

Method `execute_code()`**:** Execute R code in a sandboxed environment.

Usage:

```
SharedSession$execute_code(
  code,
  scope = "global",
  timeout_ms = 30000,
  capture_output = TRUE
)
```

Arguments:

`code` R code to execute (character string).

`scope` Variable scope: "global", "agent", or a custom scope name.
`timeout_ms` Execution timeout in milliseconds. Default 30000.
`capture_output` Capture stdout/stderr. Default TRUE.

Returns: A list with result, output, error, and duration_ms.

Method `get_var()`**:** Get a variable from a specific scope.

Usage:

```
SharedSession$get_var(name, scope = "global", default = NULL)
```

Arguments:

`name` Variable name.
`scope` Scope name. Default "global".
`default` Default value if not found.

Returns: The variable value or default.

Method `set_var()`**:** Set a variable in a specific scope.

Usage:

```
SharedSession$set_var(name, value, scope = "global")
```

Arguments:

`name` Variable name.
`value` Variable value.
`scope` Scope name. Default "global".

Returns: Invisible self for chaining.

Method `list_vars()`**:** List variables in a scope.

Usage:

```
SharedSession$list_vars(scope = "global", pattern = NULL)
```

Arguments:

`scope` Scope name. Default "global".
`pattern` Optional pattern to filter names.

Returns: Character vector of variable names.

Method `summarize_vars()`**:** Get a summary of all variables in a scope.

Usage:

```
SharedSession$summarize_vars(scope = "global")
```

Arguments:

`scope` Scope name. Default "global".

Returns: A data frame with name, type, and size information.

Method `create_scope()`**:** Create a new variable scope.

Usage:

```
SharedSession$create_scope(scope_name, parent_scope = "global")
```

Arguments:

`scope_name` Name for the new scope.

`parent_scope` Parent scope name. Default "global".

Returns: Invisible self for chaining.

Method `delete_scope()`**:** Delete a variable scope.

Usage:

```
SharedSession$delete_scope(scope_name)
```

Arguments:

`scope_name` Name of the scope to delete.

Returns: Invisible self for chaining.

Method `trace_event()`**:** Record a trace event.

Usage:

```
SharedSession$trace_event(event_type, data = list())
```

Arguments:

`event_type` Type of event (e.g., "context_push", "code_execution").

`data` Event data as a list.

Returns: Invisible self for chaining.

Method `get_trace()`**:** Get the execution trace.

Usage:

```
SharedSession$get_trace(event_types = NULL, agent = NULL)
```

Arguments:

`event_types` Optional filter by event types.

`agent` Optional filter by agent name.

Returns: A list of trace events.

Method `clear_trace()`**:** Clear the execution trace.

Usage:

```
SharedSession$clear_trace()
```

Returns: Invisible self for chaining.

Method `trace_summary()`**:** Get trace summary statistics.

Usage:

```
SharedSession$trace_summary()
```

Returns: A list with event counts, agent activity, and timing info.

Method `set_access_control()`**:** Set access control for an agent.

Usage:

```
SharedSession$set_access_control(agent_name, permissions)
```

Arguments:

`agent_name` Agent name.

`permissions` List of permissions (read_scopes, write_scopes, tools).

Returns: Invisible self for chaining.

Method `check_permission()`**:** Check if an agent has permission for an action.

Usage:

```
SharedSession$check_permission(agent_name, action, target)
```

Arguments:

`agent_name` Agent name.

`action` Action type: "read", "write", or "tool".

`target` Target scope or tool name.

Returns: TRUE if permitted, FALSE otherwise.

Method `get_sandbox_mode()`**:** Get sandbox mode.

Usage:

```
SharedSession$get_sandbox_mode()
```

Returns: The current sandbox mode.

Method `set_sandbox_mode()`**:** Set sandbox mode.

Usage:

```
SharedSession$set_sandbox_mode(mode)
```

Arguments:

`mode` Sandbox mode: "strict", "permissive", or "none".

Returns: Invisible self for chaining.

Method `print()`**:** Print method for SharedSession.

Usage:

```
SharedSession$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`shared_session` *SharedSession: Enhanced Multi-Agent Session Management*

Description

SharedSession R6 class providing enhanced environment management, execution context tracking, and safety guardrails for multi-agent orchestration.

`Skill` *Skill Class*

Description

R6 class representing a skill with progressive loading capabilities. A Skill consists of:

- Level 1: YAML frontmatter (name, description) - always loaded
- Level 2: SKILL.md body (detailed instructions) - on demand
- Level 3: R scripts (executable code) - executed by agent

Public fields

`name` The unique name of the skill (from YAML frontmatter).

`description` A brief description of the skill (from YAML frontmatter).

`path` The directory path containing the skill files.

Methods

Public methods:

- `Skill$new()`
- `Skill$load()`
- `Skill$execute_script()`
- `Skill$list_scripts()`
- `Skill$list_resources()`
- `Skill$read_resource()`
- `Skill$get_asset_path()`
- `Skill$print()`
- `Skill$clone()`

Method `new()`**:** Create a new Skill object by parsing a SKILL.md file.

Usage:

```
Skill$new(path)
```

Arguments:

`path` Path to the skill directory (containing SKILL.md).

Returns: A new Skill object.

Method `load()`**:** Load the full SKILL.md body content (Level 2).

Usage:

```
Skill$load()
```

Returns: Character string containing the skill instructions.

Method `execute_script()`: Execute an R script from the skill's scripts directory (Level 3). Uses callr for safe, isolated execution.

Usage:

```
Skill$execute_script(script_name, args = list())
```

Arguments:

`script_name` Name of the script file (e.g., "normalize.R").

`args` Named list of arguments to pass to the script.

Returns: The result from the script execution.

Method `list_scripts()`: List available scripts in the skill's scripts directory.

Usage:

```
Skill$list_scripts()
```

Returns: Character vector of script file names.

Method `list_resources()`: List available reference files in the skill's references directory.

Usage:

```
Skill$list_resources()
```

Returns: Character vector of reference file names.

Method `read_resource()`: Read content of a reference file from the references directory.

Usage:

```
Skill$read_resource(resource_name)
```

Arguments:

`resource_name` Name of the reference file.

Returns: Character string containing the resource content.

Method `get_asset_path()`: Get the absolute path to an asset in the assets directory.

Usage:

```
Skill$get_asset_path(asset_name)
```

Arguments:

`asset_name` Name of the asset file or directory.

Returns: Absolute path string.

Method `print()`: Print a summary of the skill.

Usage:

```
Skill$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`SkillRegistry` *SkillRegistry Class*

Description

R6 class that manages a collection of skills. Provides methods to:

- Scan directories for SKILL.md files
- Cache skill metadata (Level 1)
- Retrieve skills by name
- Generate prompt sections for LLM context

Methods

Public methods:

- `SkillRegistry$new()`
- `SkillRegistry$scan_skills()`
- `SkillRegistry$get_skill()`
- `SkillRegistry$has_skill()`
- `SkillRegistry$list_skills()`
- `SkillRegistry$count()`
- `SkillRegistry$generate_prompt_section()`
- `SkillRegistry$print()`
- `SkillRegistry$clone()`

Method `new()`**:** Create a new SkillRegistry, optionally scanning a directory.

Usage:

```
SkillRegistry$new(path = NULL)
```

Arguments:

`path` Optional path to scan for skills on creation.

Returns: A new SkillRegistry object.

Method `scan_skills()`**:** Scan a directory for skill folders containing SKILL.md files.

Usage:

```
SkillRegistry$scan_skills(path, recursive = FALSE)
```

Arguments:

`path` Path to the directory to scan.

`recursive` Whether to scan subdirectories. Default FALSE.

Returns: The registry object (invisibly), for chaining.

Method `get_skill()`**:** Get a skill by name.

Usage:

```
SkillRegistry$get_skill(name)
```

Arguments:

`name` The name of the skill to retrieve.

Returns: The Skill object, or NULL if not found.

Method `has_skill()`**:** Check if a skill exists in the registry.

Usage:

```
SkillRegistry$has_skill(name)
```

Arguments:

`name` The name of the skill to check.

Returns: TRUE if the skill exists, FALSE otherwise.

Method `list_skills()`**:** List all registered skills with their names and descriptions.

Usage:

```
SkillRegistry$list_skills()
```

Returns: A data.frame with columns: name, description.

Method `count()`**:** Get the number of registered skills.

Usage:

```
SkillRegistry$count()
```

Returns: Integer count of skills.

Method `generate_prompt_section()`**:** Generate a prompt section listing available skills. This can be injected into the system prompt.

Usage:

```
SkillRegistry$generate_prompt_section()
```

Returns: Character string with formatted skill list.

Method `print()`**:** Print a summary of the registry.

Usage:

```
SkillRegistry$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`SkillStore` *Skill Store Class*

Description

R6 class for managing the global skill store, including installation, updates, and discovery of skills.

Public fields

`registry_url` URL of the skill registry.

`install_path` Local path for installed skills.

`installed` List of installed skills.

Methods

Public methods:

- `SkillStore$new()`
- `SkillStore$install()`
- `SkillStore$uninstall()`
- `SkillStore$get()`
- `SkillStore$list_installed()`
- `SkillStore$search()`
- `SkillStore$update_all()`
- `SkillStore$validate()`
- `SkillStore$print()`
- `SkillStore$clone()`

Method `new()`**:** Create a new SkillStore instance.

Usage:

```
SkillStore$new(registry_url = NULL, install_path = NULL)
```

Arguments:

`registry_url` URL of the skill registry.

`install_path` Local installation path.

Returns: A new SkillStore object.

Method `install()`**:** Install a skill from the registry or a GitHub repository.

Usage:

```
SkillStore$install(skill_ref, version = NULL, force = FALSE)
```

Arguments:

`skill_ref` Skill reference (e.g., "username/skillname" or registry name).

`version` Optional specific version to install.

`force` Force reinstallation even if already installed.

Returns: The installed Skill object.

Method `uninstall()`**:** Uninstall a skill.

Usage:

```
SkillStore$uninstall(name)
```

Arguments:

`name` Skill name.

Returns: Self (invisibly).

Method `get()`**:** Get an installed skill.

Usage:

```
SkillStore$get(name)
```

Arguments:

`name` Skill name.

Returns: A Skill object or NULL.

Method `list_installed()`**:** List installed skills.

Usage:

```
SkillStore$list_installed()
```

Returns: A data frame of installed skills.

Method `search()`**:** Search the registry for skills.

Usage:

```
SkillStore$search(query = NULL, capability = NULL)
```

Arguments:

`query` Search query.
`capability` Filter by capability.

Returns: A data frame of matching skills.

Method `update_all()`**:** Update all installed skills to latest versions.

Usage:

```
SkillStore$update_all()
```

Returns: Self (invisibly).

Method `validate()`**:** Validate a skill.yaml manifest.

Usage:

```
SkillStore$validate(path)
```

Arguments:

`path` Path to skill directory or skill.yaml file.

Returns: A list with validation results.

Method `print()`**:** Print method for SkillStore.

Usage:

`SkillStore$print()`

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

Usage:

`SkillStore$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

`skill_manifest` *Skill Manifest Specification*

Description

The skill.yaml specification defines the structure for distributable skills.

Specification

```
# skill.yaml - Skill Manifest Specification v1.0

# Required fields
name: my-skill                   # Unique skill identifier (lowercase, hyphens)
version: 1.0.0                    # Semantic version
description: Brief description    # One-line description

# Author information
author:
  name: Author Name
  email: author@example.com
  url: https://github.com/author

# License (SPDX identifier)
license: MIT

# R package dependencies
dependencies:
  - dplyr >= 1.0.0
  - ggplot2

# System requirements
system_requirements:
  - python >= 3.8             # Optional external requirements

# MCP server configuration (optional)
mcp:
  command: npx                  # Command to start MCP server
```

```
    args:
      - -y
      - "@my-org/my-mcp-server"
    env:
      API_KEY: "${MY_API_KEY}"   # Environment variable substitution

  # Capabilities this skill provides
  capabilities:
    - data-analysis
    - visualization
    - machine-learning

  # Prompt templates
  prompts:
    system: |
      You are a specialized assistant for...
    examples:
      - "Analyze this dataset..."
      - "Create a visualization of..."

  # Entry points
  entry:
    main: SKILL.md                # Main skill instructions
    scripts: scripts/             # Directory containing R scripts

  # Repository information
  repository:
    type: github
    url: https://github.com/author/my-skill
```

skill_registry *Skill Registry: Scan and Manage Skills*

Description

SkillRegistry class for discovering, caching, and retrieving skills. Scans directories for SKILL.md files and provides access to skill metadata.

skill_store *Global Skill Store*

Description

A CRAN-like experience for sharing AI capabilities. Skills are packaged with a skill.yaml manifest defining dependencies, MCP endpoints, and prompt templates.

SlmEngine *SLM Engine Class*

Description

R6 class for managing local Small Language Model inference. Provides a unified interface for loading model weights, running inference, and managing model lifecycle.

Public fields

`model_path` Path to the model weights file.

`model_name` Human-readable model name.

`backend` The inference backend ("onnx", "torch", "gguf").

`config` Model configuration parameters.

`loaded` Whether the model is currently loaded in memory.

Methods

Public methods:

- `SlmEngine$new()`
- `SlmEngine$load()`
- `SlmEngine$unload()`
- `SlmEngine$generate()`
- `SlmEngine$stream()`
- `SlmEngine$info()`
- `SlmEngine$print()`
- `SlmEngine$clone()`

Method `new()`**:** Create a new SLM Engine instance.

Usage:

```
SlmEngine$new(model_path, backend = "gguf", config = list())
```

Arguments:

`model_path` Path to the model weights file (GGUF, ONNX, or PT format).

`backend` Inference backend to use: "gguf" (default), "onnx", or "torch".

`config` Optional list of configuration parameters.

Returns: A new SlmEngine object.

Method `load()`**:** Load the model into memory.

Usage:

```
SlmEngine$load()
```

Returns: Self (invisibly).

Method `unload()`: Unload the model from memory.

Usage:

```
SlmEngine$unload()
```

Returns: Self (invisibly).

Method `generate()`: Generate text completion from a prompt.

Usage:

```
SlmEngine$generate(
  prompt,
  max_tokens = 256,
  temperature = 0.7,
  top_p = 0.9,
  stop = NULL
)
```

Arguments:

`prompt` The input prompt text.

`max_tokens` Maximum number of tokens to generate.

`temperature` Sampling temperature (0.0 to 2.0).

`top_p` Nucleus sampling parameter.

`stop` Optional stop sequences.

Returns: A list with generated text and metadata.

Method `stream()`: Stream text generation with a callback function.

Usage:

```
SlmEngine$stream(
  prompt,
  callback,
  max_tokens = 256,
  temperature = 0.7,
  top_p = 0.9,
  stop = NULL
)
```

Arguments:

`prompt` The input prompt text.

`callback` Function called with each generated token.

`max_tokens` Maximum number of tokens to generate.

`temperature` Sampling temperature.

`top_p` Nucleus sampling parameter.

`stop` Optional stop sequences.

Returns: A list with the complete generated text and metadata.

Method `info()`: Get model information and statistics.

Usage:

```
SlmEngine$info()
```

Returns: A list with model metadata.

Method `print()`**:** Print method for SlmEngine.

Usage:

```
SlmEngine$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`slm_engine` *Native SLM (Small Language Model) Engine*

Description

Generic interface for loading and running local language models without external API dependencies. Supports multiple backends including ONNX Runtime and LibTorch for quantized model execution.

Factory function to create a new SLM Engine for local model inference.

Usage

```
slm_engine(model_path, backend = "gguf", config = list())
```

Arguments

`model_path`	Path to the model weights file.
`backend`	Inference backend: "gguf" (default), "onnx", or "torch".
`config`	Optional configuration list.

Value

An SlmEngine object.

Examples

```
if (interactive()) {
# Load a GGUF model
engine <- slm_engine("models/llama-3-8b-q4.gguf")
engine$load()

# Generate text
result <- engine$generate("What is the capital of France?")
cat(result$text)

# Stream generation
engine$stream("Tell me a story", callback = cat)

# Cleanup
engine$unload()
}
```

---

spec_model *Specification Layer: Model Interfaces*

Description

Abstract base classes (interfaces) for AI models.

---

start_feishu_webhook_server *Start a Feishu Webhook Server*

Description

Start a minimal `httpuv` server exposing a Feishu callback endpoint.

Usage

```
start_feishu_webhook_server(
  runtime,
  host = "127.0.0.1",
  port = 8788,
  path = "/feishu/webhook"
)
```

Arguments

`runtime`	A `ChannelRuntime` with a Feishu adapter registered.
`host`	Bind host.
`port`	Bind port.
`path`	Callback path.

Value

An `httpuv` server handle.

`stdlib_agents` *Standard Agent Library: Built-in Specialist Agents*

Description

Factory functions for creating standard library agents with scoped tools and safety guardrails. These agents form the foundation of the multi-agent orchestration system.

`StepfunProvider` *Stepfun Provider Class*

Description

Provider class for Stepfun.

Super class

`aisdk::OpenAIProvider` -> `StepfunProvider`

Methods

Public methods:

- `StepfunProvider$new()`
- `StepfunProvider$language_model()`
- `StepfunProvider$clone()`

Method `new()`**:** Initialize the Stepfun provider.

Usage:

```
StepfunProvider$new(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments:

`api_key` Stepfun API key. Defaults to STEPFUN_API_KEY env var.

`base_url` Base URL. Defaults to https://api.stepfun.com/v1.

`headers` Optional additional headers.

Method `language_model()`**:** Create a language model.

Usage:

```
StepfunProvider$language_model(model_id = NULL)
```

Arguments:

`model_id` The model ID (e.g., "step-3.5-flash").

Returns: A StepfunLanguageModel object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`strategy` *Output Strategy System*

Description

Implements the Strategy pattern for handling different structured output formats from LLMs. This allows the SDK to be extended with new output types (e.g., objects, enums, dataframes) without modifying core logic.

`stream_text` *Stream Text*

Description

Generate text using a language model with streaming output. This function provides a real-time stream of tokens through a callback.

Usage

```
stream_text(
  model = NULL,
  prompt,
  callback = NULL,
  system = NULL,
  temperature = 0.7,
  max_tokens = NULL,
  tools = NULL,
  max_steps = 1,
  sandbox = FALSE,
  skills = NULL,
  session = NULL,
  hooks = NULL,
  registry = NULL,
  ...
)
```

Arguments

`model`	Either a LanguageModelV1 object, or a string ID like "openai:gpt-4o".
`prompt`	A character string prompt, or a list of messages.
`callback`	A function called for each text chunk: `callback(text, done)`.
`system`	Optional system prompt.
`temperature`	Sampling temperature (0-2). Default 0.7.
`max_tokens`	Maximum tokens to generate.
`tools`	Optional list of Tool objects for function calling.
`max_steps`	Maximum number of generation steps (tool execution loops). Default 1. Set to higher values (e.g., 5) to enable automatic tool execution.
`sandbox`	Logical. If TRUE, enables R-native programmatic sandbox mode. See `generate_text` for details. Default FALSE.
`skills`	Optional path to skills directory, or a SkillRegistry object.
`session`	Optional ChatSession object for shared state.
`hooks`	Optional HookHandler object.
`registry`	Optional ProviderRegistry to use.
`...`	Additional arguments passed to the model.

Value

A GenerateResult object (accumulated from the stream).

Examples

```
if (interactive()) {
  model <- create_openai()$language_model("gpt-4o")
  stream_text(model, "Tell me a story", callback = function(text, done) {
    if (!done) cat(text)
  })
}
```

`team` *Agent Team: Automated Multi-Agent Orchestration*

Description

AgentTeam class for managing a group of agents and automating their orchestration. It implements a "Manager-Worker" pattern where a synthesized Manager agent delegates tasks to registered worker agents based on their descriptions.

`Telemetry` *Telemetry Class*

Description

R6 class for logging events in a structured format (JSON).

Public fields

`trace_id` Current trace ID for the session.

`pricing_table` Pricing for common models (USD per 1M tokens).

Methods

Public methods:

- `Telemetry$new()`
- `Telemetry$log_event()`
- `Telemetry$as_hooks()`
- `Telemetry$calculate_cost()`
- `Telemetry$clone()`

Method `new()`**:** Initialize Telemetry

Usage:

```
Telemetry$new(trace_id = NULL)
```

Arguments:

`trace_id` Optional trace ID. If NULL, generates a random one.

Method `log_event()`**:** Log an event

Usage:

```
Telemetry$log_event(type, ...)
```

Arguments:

`type` Event type (e.g., "generation_start", "tool_call").

`...` Additional fields to log.

Method `as_hooks()`**:** Create hooks for telemetry

Usage:

```
Telemetry$as_hooks()
```

Returns: A HookHandler object pre-configured with telemetry logs.

Method `calculate_cost()`**:** Calculate estimated cost for a generation result

Usage:

```
Telemetry$calculate_cost(result, model_id = NULL)
```

Arguments:

`result` The GenerateResult object.

`model_id` Optional model ID string. if NULL, tries to guess from context (not reliable yet, passing in log_event might be better).

Returns: Estimated cost in USD, or NULL if unknown.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`test_new_skill` *Test a Newly Created Skill*

Description

Verifies a skill by running a test query against it in a sandboxed session.

Usage

```
test_new_skill(skill_name, test_query, registry, model)
```

Arguments

`skill_name`	Name of the skill to test (must be in the registry).
`test_query`	A natural language query to test the skill (e.g., "Use hello_world to say hi").
`registry`	The skill registry object.
`model`	A model object to use for the test agent.

Value

A list containing `success` (boolean) and `result` (string).

`Tool` *Tool Class*

Description

R6 class representing a callable tool for LLM function calling. A Tool connects an LLM's tool call request to an R function.

Public fields

`name` The unique name of the tool.

`description` A description of what the tool does.

`parameters` A z_object schema defining the tool's parameters.

`layer` Tool layer: "llm" (loaded into context) or "computer" (executed via bash/filesystem).

`meta` Optional metadata for the tool (e.g., caching configuration).

Methods

Public methods:

- `Tool$new()`
- `Tool$to_api_format()`
- `Tool$run()`
- `Tool$print()`
- `Tool$clone()`

Method `new()`**:** Initialize a Tool.

Usage:

```
Tool$new(name, description, parameters, execute, layer = "llm", meta = NULL)
```

Arguments:

`name` Unique tool name (used by LLM to call the tool).

`description` Description of the tool's purpose.

`parameters` A z_object schema defining expected parameters.

`execute` An R function that implements the tool logic.

`layer` Tool layer: "llm" or "computer" (default: "llm").

`meta` Optional metadata list (e.g., cache_control).

Method `to_api_format()`**:** Convert tool to API format.

Usage:

```
Tool$to_api_format(provider = "openai")
```

Arguments:

`provider` Provider name ("openai" or "anthropic"). Default "openai".

Returns: A list in the format expected by the API.

Method `run()`**:** Execute the tool with given arguments.

Usage:

```
Tool$run(args, envir = NULL)
```

Arguments:

`args` A list or named list of arguments.

`envir` Optional environment in which to evaluate the tool function. When provided, the environment is passed as `.envir` in the args list, allowing the execute function to access and modify session variables.

Returns: The result of executing the tool function.

Method `print()`**:** Print method for Tool.

Usage:

```
Tool$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`tool` *Create a Tool*

Description

Factory function to create a Tool object. This is the recommended way to define tools for LLM function calling.

Usage

```
tool(
  name,
  description,
  parameters = NULL,
  execute = NULL,
  layer = "llm",
  meta = NULL
)
```

Arguments

`name`	Unique tool name (used by LLM to call the tool).
`description`	Description of the tool's purpose. Be descriptive to help the LLM understand when to use this tool.
`parameters`	A z_schema object (z_object/z_any/etc), a named list, a character vector, or NULL. When NULL, the schema is inferred from the execute function signature (if possible) and defaults to flexible types.
`execute`	An R function that implements the tool logic. It can accept a single list argument (args), or standard named parameters. List-style functions receive a single list argument containing parameters.
`layer`	Tool layer: "llm" (loaded into context) or "computer" (executed via bash/filesystem). Default is "llm". Computer layer tools are not loaded into context but executed via bash.
`meta`	Optional metadata associated with the tool (e.g., `list(cache_control = list(type = "ephemeral"))`).

Value

A Tool object.

Examples

```
if (interactive()) {
  # Define a weather tool
  get_weather <- tool(
    name = "get_weather",
    description = "Get the current weather for a location",
    parameters = z_object(
      location = z_string(description = "The city name, e.g., 'Beijing'"),
      unit = z_enum(c("celsius", "fahrenheit"), description = "Temperature unit")
    ),
    execute = function(args) {
      # In real usage, call a weather API here
      paste("Weather in", args$location, "is 22 degrees", args$unit)
    }
  )

  # Use with generate_text
  result <- generate_text(
    model = "openai:gpt-4o",
    prompt = "What's the weather in Tokyo?",
    tools = list(get_weather)
  )
}
```

`uninstall_skill` *Uninstall a Skill*

Description

Remove an installed skill.

Usage

```
uninstall_skill(name)
```

Arguments

`name`	Skill name.

`update_renviron` *Update .Renviron with new values*

Description

Updates or appends environment variables to the .Renviron file.

Usage

```
update_renviron(updates, path = ".Renviron")
```

Arguments

`updates`	A named list of key-value pairs to update.
`path`	Path to .Renviron file (default: project root)

Value

Invisible TRUE if successful

`utils_capture` *Capture R Console Output*

Description

Internal helpers to capture printed output, messages, and warnings from evaluated R expressions so tool execution can be rendered cleanly in the console UI.

`utils_http` *Utilities: HTTP and Retry Logic*

Description

Provides standardized HTTP request handling with exponential backoff retry.

Implements multi-layer defense strategy for handling API responses:

- Empty response body handling (returns instead of parse error)
- JSON parsing with repair fallback
- SSE stream error recovery
- Graceful degradation on malformed data

`utils_json` *JSON Utilities*

Description

Provides robust utilities for parsing potentially truncated or malformed JSON strings, commonly encountered in streaming LLM outputs.

A robust utility that uses a finite state machine to close open brackets, braces, and quotes to make a truncated JSON string valid for parsing.

Usage

```
fix_json(json_str)
```

Arguments

`json_str`	A potentially truncated JSON string.

Value

A repaired JSON string.

Examples

```
fix_json('{"name": "Gene...')
fix_json('[1, 2, {"a":')
```

`utils_middleware` *Utilities: Middleware System*

Description

Provides middleware functionality to wrap and enhance language models. Middleware can transform parameters and wrap generate/stream operations.

`utils_registry` *Utilities: Provider Registry*

Description

A registry for managing AI model providers. Supports the `provider:model` syntax for accessing models.

`variable_registry` *Variable Registry*

Description

Provides a mechanism to protect specific variables from being accidentally modified or duplicated by the Agent within the sandbox environment.

`VolcengineProvider` *Volcengine Provider Class*

Description

Provider class for the Volcengine Ark platform.

Super class

`aisdk::OpenAIProvider` -> `VolcengineProvider`

Methods

Public methods:

- `VolcengineProvider$new()`
- `VolcengineProvider$language_model()`
- `VolcengineProvider$clone()`

Method `new()`: Initialize the Volcengine provider.

Usage:

```
VolcengineProvider$new(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments:

`api_key`	Volcengine API key. Defaults to ARK_API_KEY env var.
`base_url`	Base URL. Defaults to https://ark.cn-beijing.volces.com/api/v3.
`headers`	Optional additional headers.

Method `language_model()`: Create a language model.

Usage:

```
VolcengineProvider$language_model(model_id = NULL)
```

Arguments:

`model_id`	The model ID (e.g., "doubao-1-5-pro-256k-250115" or "gpt-4o").

Returns: A VolcengineLanguageModel object.

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

Usage:

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

Arguments:

`deep`	Whether to make a deep clone.

`walk_ast` *Walk an Abstract Syntax Tree*

Description

Recursively traverse an R expression and apply a visitor function to each node.

Usage

```
walk_ast(expr, visitor)
```

Arguments

`expr`	An R expression, call, or primitive type.
`visitor`	A function taking a node as argument.

`wrap_language_model` *Wrap Language Model with Middleware*

Description

Wraps a LanguageModelV1 with one or more middleware instances. Middleware is applied in order: first middleware transforms first, last middleware wraps closest to the model.

Usage

```
wrap_language_model(model, middleware, model_id = NULL, provider_id = NULL)
```

Arguments

`model`	A LanguageModelV1 object.
`middleware`	A single Middleware object or a list of Middleware objects.
`model_id`	Optional custom model ID.
`provider_id`	Optional custom provider ID.

Value

A new LanguageModelV1 object with middleware applied.

`wrap_reactive_tools` *Wrap Reactive Tools*

Description

Wraps reactive tools to inject reactiveValues and session into their execute functions. Call this in your Shiny server before passing tools to aiChatServer.

Usage

```
wrap_reactive_tools(tools, rv, session)
```

Arguments

`tools`	List of Tool objects, possibly including ReactiveTool objects.
`rv`	The reactiveValues object to inject.
`session`	The Shiny session object to inject.

Value

List of wrapped Tool objects ready for use.

`XAIProvider` *xAI Provider Class*

Description

Provider class for xAI.

Super class

`aisdk::OpenAIProvider` -> `XAIProvider`

Methods

Public methods:

- `XAIProvider$new()`
- `XAIProvider$language_model()`
- `XAIProvider$clone()`

Method `new()`**:** Initialize the xAI provider.

Usage:

```
XAIProvider$new(api_key = NULL, base_url = NULL, headers = NULL)
```

Arguments:

`api_key`	xAI API key. Defaults to XAI_API_KEY env var.
`base_url`	Base URL. Defaults to https://api.x.ai/v1.
`headers`	Optional additional headers.

Method `language_model()`**:** Create a language model.

Usage:

```
XAIProvider$language_model(model_id = NULL)
```

Arguments:

`model_id` The model ID (e.g., "grok-beta", "grok-2-1212").

Returns: A XAILanguageModel object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`z_aes_mapping` *Aesthetic Mapping Schema*

Description

Schema for ggplot2 aesthetic mappings (aes).

Usage

```
z_aes_mapping(known_only = FALSE)
```

Arguments

`known_only`	If TRUE, only include known aesthetics in schema.

Value

A z_object schema.

`z_any` *Create Any Schema*

Description

Create a JSON Schema that accepts any JSON value.

Usage

```
z_any(description = NULL, nullable = TRUE, default = NULL)
```

Arguments

`description`	Optional description of the field.
`nullable`	If TRUE, allows null values.
`default`	Optional default value.

Value

A list representing JSON Schema for any value.

Examples

```
z_any(description = "Flexible input")
```

z_array *Create Array Schema*

Description

Create a JSON Schema for array type.

Usage

```
z_array(
  items,
  description = NULL,
  nullable = FALSE,
  default = NULL,
  min_items = NULL,
  max_items = NULL
)
```

Arguments

`items`	Schema for array items (created by z_* functions).
`description`	Optional description of the field.
`nullable`	If TRUE, allows null values.
`default`	Optional default value.
`min_items`	Optional minimum number of items.
`max_items`	Optional maximum number of items.

Value

A list representing JSON Schema for array.

Examples

```
z_array(z_string(), description = "List of names")
```

z_boolean *Create Boolean Schema*

Description

Create a JSON Schema for boolean type.

Usage

```
z_boolean(description = NULL, nullable = FALSE, default = NULL)
```

Arguments

`description`	Optional description of the field.
`nullable`	If TRUE, allows null values.
`default`	Optional default value.

Value

A list representing JSON Schema for boolean.

Examples

```
z_boolean(description = "Whether to include details")
```

`z_coord` *Coordinate System Schema*

Description

Schema for coordinate system.

Usage

```
z_coord()
```

Value

A z_object schema.

`z_dataframe` *Create Dataframe Schema*

Description

Create a schema that represents a dataframe (or list of row objects). This is an R-specific convenience function that generates a JSON Schema for an array of objects. The LLM will be instructed to output data in a format that can be easily converted to an R dataframe using `dplyr::bind_rows()` or `do.call(rbind, lapply(..., as.data.frame))`.

Create a schema that represents a dataframe (or list of row objects). This is an R-specific convenience function that generates a JSON Schema for an array of objects.

Usage

```
z_dataframe(
  ...,
  .description = NULL,
  .nullable = FALSE,
  .default = NULL,
  .min_rows = NULL,
  .max_rows = NULL
)
```

Arguments

`...`	Named arguments where names are column names and values are z_schema objects representing the column types.
`.description`	Optional description of the dataframe.
`.nullable`	If TRUE, allows null values.
`.default`	Optional default value.
`.min_rows`	Optional minimum number of rows.
`.max_rows`	Optional maximum number of rows.

Value

A z_schema object representing an array of objects.

A z_schema object representing an array of objects.

Examples

```
# Define a schema for a dataframe of genes
gene_schema <- z_dataframe(
  gene_name = z_string(description = "Name of the gene"),
  expression = z_number(description = "Expression level"),
  significant = z_boolean(description = "Is statistically significant")
)

# Use with generate_object
# result <- generate_object(model, "Extract gene data...", gene_schema)
# df <- dplyr::bind_rows(result$object)
```

`z_describe` *Describe Schema*

Description

Add a description to a z_schema object (pipe-friendly).

Usage

```
z_describe(schema, description)
```

Arguments

`schema`	A z_schema object.
`description`	The description string.

Value

The modified z_schema object.

`z_empty_object` *Create Empty Object Schema*

Description

Create a JSON Schema for an empty object `{}`.

Usage

```
z_empty_object(description = NULL)
```

Arguments

`description`	Optional description.

Value

A z_schema object.

`z_enum` *Create Enum Schema*

Description

Create a JSON Schema for string enum type.

Usage

```
z_enum(values, description = NULL, nullable = FALSE, default = NULL)
```

Arguments

`values`	Character vector of allowed values.
`description`	Optional description of the field.
`nullable`	If TRUE, allows null values.
`default`	Optional default value.

Value

A list representing JSON Schema for enum.

Examples

```
z_enum(c("celsius", "fahrenheit"), description = "Temperature unit")
```

`z_facet` *Facet Schema*

Description

Schema for faceting.

Usage

```
z_facet()
```

Value

A z_object schema.

`z_geom_layer` *Build Geom-Specific Layer Schema*

Description

Creates a precise schema for a specific geom type.

Usage

```
z_geom_layer(geom_name)
```

Arguments

`geom_name`	Name of the geom.

Value

A z_object schema tailored to the geom.

`z_ggplot` *GGPlot Object Schema*

Description

Top-level schema for a ggplot object.

Usage

```
z_ggplot()
```

Value

A z_object schema.

`z_guide` *Guide Schema*

Description

Schema for guides (legend/axis).

Usage

```
z_guide()
```

Value

A z_object schema.

`z_integer` *Create Integer Schema*

Description

Create a JSON Schema for integer type.

Usage

```
z_integer(
  description = NULL,
  nullable = FALSE,
  default = NULL,
  minimum = NULL,
  maximum = NULL
)
```

Arguments

`description`	Optional description of the field.
`nullable`	If TRUE, allows null values.
`default`	Optional default value.
`minimum`	Optional minimum value.
`maximum`	Optional maximum value.

Value

A list representing JSON Schema for integer.

Examples

```
z_integer(description = "Number of items", minimum = 0)
```

`z_layer` *Layer Schema*

Description

Schema for a single ggplot2 layer.

Usage

```
z_layer()
```

Value

A z_object schema.

`z_number` *Create Number Schema*

Description

Create a JSON Schema for number (floating point) type.

Usage

```
z_number(
  description = NULL,
  nullable = FALSE,
  default = NULL,
  minimum = NULL,
  maximum = NULL
)
```

Arguments

`description`	Optional description of the field.
`nullable`	If TRUE, allows null values.
`default`	Optional default value.
`minimum`	Optional minimum value.
`maximum`	Optional maximum value.

Value

A list representing JSON Schema for number.

Examples

```
z_number(description = "Temperature value", minimum = -100, maximum = 100)
```

`z_object` *Create Object Schema*

Description

Create a JSON Schema for object type. This is the primary schema builder for defining tool parameters.

Usage

```
z_object(
  ...,
  .description = NULL,
  .required = NULL,
  .additional_properties = FALSE
)
```

Arguments

`...`	Named arguments where names are property names and values are z_schema objects created by z_* functions.
`.description`	Optional description of the object.
`.required`	Character vector of required field names. If NULL (default), all fields are considered required.
`.additional_properties`	Whether to allow additional properties. Default FALSE.

Value

A list representing JSON Schema for object.

Examples

```
z_object(
  location = z_string(description = "City name, e.g., Beijing"),
  unit = z_enum(c("celsius", "fahrenheit"), description = "Temperature unit")
)
```

`z_position` *Position Adjustment Schema*

Description

Schema for position adjustments with type-specific parameters.

Usage

```
z_position(position_type = NULL)
```

Arguments

`position_type`	Optional specific position type for strict schema.

Value

A z_object schema.

`z_scale` *Scale Schema*

Description

Schema for a scale definition.

Usage

```
z_scale()
```

Value

A z_object schema.

`z_string` *Create String Schema*

Description

Create a JSON Schema for string type.

Usage

```
z_string(description = NULL, nullable = FALSE, default = NULL)
```

Arguments

`description`	Optional description of the field.
`nullable`	If TRUE, allows null values.
`default`	Optional default value.

Value

A list representing JSON Schema for string.

Examples

```
z_string(description = "The city name")
```

`z_theme` *Theme Schema*

Description

Schema for theme settings with full hierarchy support.

Usage

```
z_theme(flat = TRUE)
```

Arguments

`flat`	If TRUE, returns flat structure. If FALSE, returns hierarchical.

Value

A z_object schema.

Index