KNEO AGENT SDK

KNEO AGENT SDK

API Reference Manual

Complete reference for the public Kneo Agent SDK surface, including runtimes, workflows, providers, middleware, utilities, and embedded practical guides.

VERSION

v1.2.0

RUNTIME

Python 3.12+

LICENSE

MIT

Prepared as a printable reference edition with API details, usage guidance, and architecture-facing concepts consolidated into one manual.

asyncio | fully typed | agents + workflows

Table of Contents

Use this section as the printable navigation guide for the manual. The linked entries mirror the document structure that follows after the introduction.

KNEO AGENT SDK

API Reference Manual

Complete reference for all public classes, methods, protocols, and utilities.

v1.2.0 Python 3.12+ MIT License asyncio fully typed

What's new in v1.2.0

Released **2026-05-07**. Fully backwards-compatible with 1.1.x — every public name from 1.1.x still works the same way. This callout is a quick index to the new public surface; the authoritative spec for each item lives in its dedicated section below.

New public surface ADDED

Local-LLM support. New `base_url=` and `api_key=` keyword arguments on `OpenAIAgentsImpl`, `NativeRuntimeFactory.for_openai`, `BridgeAgentFactory.for_openai`, `kneo_agent.simple.build_agent`, and `build_sync_agent`. Point at any OpenAI-compatible HTTP endpoint (Ollama, vLLM, llama.cpp, LocalAI) to run on-prem.

Middleware bundle — new `kneo_agent.middleware` subpackage with `RetryMiddleware`, `RateLimitMiddleware`, `TokenBudgetMiddleware` (+ `TokenBudgetExceeded`), `RedactionMiddleware` (+ `COMMON_PATTERNS`). Built on the existing `AgentMiddleware` framework; framework types re-exported from the same module.

Secret plumbing — new `SecretProvider` Protocol in `kneo_agent.utils`, plus three reference implementations: `EnvSecretProvider`, `FileSecretProvider`, `MappingSecretProvider`. Pair with `SecretNotFound`.

Workflow retry — new `kneo_agent.workflows.RetryStep` wraps any `WorkflowComponent` with retry-on-failure semantics, mirroring the middleware-side knobs.

MCP TLS / mTLS — new `verify=`, `ca_bundle=`, `client_cert=`, `client_key=` parameters on `MCPServerConfig.http()` and `MCPServerConfig.sse()`. New `MCPServerConfig.build_ssl_context()` method.

Formal `RunResult.metadata["usage"]` schema — keys `input_tokens`, `output_tokens`, `total_tokens` (with `prompt_tokens` / `completion_tokens` accepted as aliases). Populated automatically by the OpenAI Agents native runtime; spec on `RunResult`.

Python 3.13 classifier added; CI matrix already exercised 3.13 in 1.1.x.

New documentation GUIDES

`docs/user/api_stability.md` — public-surface definition and deprecation policy.
`docs/user/upgrading_to_1.2.md` — practical "should I upgrade?" walkthrough with code examples for each user-visible change.
`docs/user/self_hosted_observability.md` — end-to-end recipes wiring `OpenTelemetryMiddleware` to a user-run OTLP collector / Jaeger / Grafana Tempo / SigNoz.
`docs/user/offline_install.md` — air-gapped install via `pip download + --no-index --find-links` ; documented no-phone-home audit.

New cookbook recipes EXAMPLES

`examples/cookbook/local_ollama.py`
`examples/cookbook/workflow_branching_and_retry.py`
`examples/cookbook/mcp_server_catalog.py`
`examples/cookbook/sql_query_tool.py`
`examples/cookbook/rest_api_tool.py`
`examples/cookbook/object_store_tool.py`
`examples/cookbook/search_index_tool.py`

> **Population status for `RunResult.metadata["usage"]` :** the OpenAI Agents native runtime populates the documented keys today. LangChain, Google ADK, and Bridge runtimes do not yet populate them; apps that need usage on those paths can populate via custom middleware. Per-runtime rollout is tracked for a follow-up release.

Introduction

Kneo Agent is a framework-agnostic Python Agent SDK that lets you build LLM-powered agents and composite workflows running on Google ADK, OpenAI Agents SDK, or LangChain using four runtime categories: **Bridge**, **Native**, **Adapter**, and **Workflow**.

The entire public API is exported from the top-level `kneo_agent` package. Most application code only needs three imports:

```
from kneo_agent import Agent, AgentBuilder, RunConfig, ToolDefinition
from kneo_agent.patterns import NativeRuntimeFactory
from kneo_agent.utils import ToolRegistry
```

Design Patterns CONCEPT

Bridge Pattern

Decouples the agentic loop *strategy* (Simple, ReAct, Plan-Act) from the platform *implementation* (Google ADK, OpenAI, LangChain). Both sides vary independently. Use when designing from scratch.

Adapter Pattern

Wraps an existing, fixed-interface platform object and translates its API to `AgentRuntime`. The platform owns its own loop. Use when integrating an existing agent executor.

Native Runtime

Hands loop ownership to the platform SDK/runtime itself while still exposing `AgentRuntime`. Use for platform-owned runtimes such as the OpenAI Agents SDK and native Google ADK execution.

Workflow Runtime

Composes agents, nested workflows, and custom function steps using the composite pattern. A workflow itself implements `AgentRuntime`, so it can be used as an agent runtime directly or wrapped with `workflow.as_agent()`.

Quick Start

Two convenience builders cover the common cases. `build_sync_agent` is the friction-free starting point — a blocking wrapper that hides `asyncio` entirely, ideal for scripts and prototypes. `build_agent` is the async equivalent for code already inside an event loop; it accepts the same parameters and returns a regular `Agent` you `await`. The parameter table below applies to both.

```
build_sync_agent(provider: "openai" | "langchain" | "google-adk", **kwargs) → SyncAgent
```

Blocking one-liner from `kneo_agent.simple`, re-exported as `kneo_agent.build_sync_agent`. Picks a runtime factory based on `provider`, applies sensible defaults, attaches optional tools and middlewares, and returns a `SyncAgent` whose `run`, `chat`, and `stream` methods block until the underlying coroutine completes — no `asyncio` at the call site.

```python
from kneo_agent import build_sync_agent

agent = build_sync_agent(
    "openai",
    model="gpt-4o-mini",
    system_prompt="You are a helpful assistant.",
)

print(agent.chat("What is 2 + 2?"))    # blocks, returns str
```

> **Heads up:** Each call spins up a fresh event loop via `asyncio.run`. `SyncAgent` raises `RuntimeError` if invoked from inside an already-running event loop (Jupyter, FastAPI handlers, other async code) rather than silently deadlocking; reach for the async `build_agent` instead in those contexts.

PARAMETER	TYPE	DESCRIPTION
`provider`	`"openai" | "langchain" | "google-adk" | None`	Provider shortcut. Mutually exclusive with `runtime`.
`runtime`	`AgentRuntime | None`	Pre-built runtime. Use this when you already have a custom runtime, a workflow, or a third-party adapter.
`model`	`str`	OpenAI model id. Default `"gpt-4o-mini"`.
`openai_client`	`Any | None`	Optional `openai.AsyncOpenAI` client.
`chat_model`	`Any`	Required for `"langchain"`: any `langchain_core.language_models.BaseChatModel`.

`adk_runner`	`Any`	Required for `"google-adk"`: an ADK runner with `run_async`.
`adk_app_name / adk_user_id / adk_session_id`	`str`	ADK identifiers. All default to `"default"`.
`name / description / system_prompt`	`str / str / str | None`	Agent metadata.
`tools`	`ToolRegistry | None`	Tool registry to attach. For LangChain Bridge runtimes the registry's handlers are also wired into the runtime for dispatch.
`middlewares`	`list[AgentMiddleware] | None`	Middlewares to attach (e.g. `OpenTelemetryMiddleware()`).
`strategy`	`"simple" | "react" | "plan-act"`	Bridge agent loop strategy. Default `"react"`.

```
build_agent(*args, **kwargs) → Agent
```

Async variant of `build_sync_agent`. Accepts the same parameters (see the table above) but returns a regular async `Agent` directly, with no `SyncAgent` wrapper. Use this when you are already writing async code — e.g. inside a Jupyter cell, a FastAPI handler, or another `asyncio` task.

```python
import asyncio
from kneo_agent import build_agent

agent = build_agent(
    "openai",
    model="gpt-4o-mini",
    system_prompt="You are a helpful assistant.",
)

async def main():
    print(await agent.chat("What is 2 + 2?"))

asyncio.run(main())
```

class SyncAgent wrapper

Composition wrapper returned by `build_sync_agent`. Properties (`config`, `agent_name`, `runtime_name`, `history`) and history-management methods (`clear_history`, `inject_history`) pass through unchanged. The async surface (`run`, `chat`, `stream`) is replaced with blocking equivalents; `stream` returns the full list of chunks rather than yielding incrementally.

Explicit builder

Use the explicit chain when you need behavior `build_agent` does not expose: workflow composition, custom `RuntimeImpl` implementations, fine-grained per-step skill loading, etc.

```python
import asyncio
from kneo_agent import AgentBuilder, ToolDefinition
from kneo_agent.patterns import NativeRuntimeFactory
from kneo_agent.workflows import WorkflowBuilder
from kneo_agent.utils import ToolRegistry

# 1. Define tools
registry = ToolRegistry()

@registry.tool(
    name="get_weather",
    description="Get current weather for a city.",
    parameters={"type": "object", "properties": {"city": {"type": "string"}},
"required": ["city"]},
)
def get_weather(args: dict) -> str:
    return f"22 °C and sunny in {args['city']}"

# 2. Create a platform-owned runtime
runtime = NativeRuntimeFactory.for_openai(model="gpt-4o", strategy="react")

# 3. Build the agent
agent = (
    AgentBuilder()
    .with_name("Weather Assistant")
    .with_system_prompt("You are a helpful weather assistant.")
    .with_tools(registry.definitions)
    .use_runtime(runtime)
    .build()
)

# 4. Run
async def main():
    result = await agent.run("What is the weather in Tokyo?")
    print(result.final_message)
```

```
asyncio.run(main())
```

> **Workflow note:** Workflows are first-class runtimes. You can compose multiple agents into a `SequentialWorkflow` and expose that workflow as a normal `Agent` with `workflow.as_agent(...)`.

Agent

The central class of the SDK. Stateful, framework-agnostic, and built via `AgentBuilder`. Never instantiated directly.

class Agent CLASS

A stateful conversational agent. Holds conversation history and delegates all LLM work to an injected `AgentRuntime`.

Properties

PROPERTY	TYPE	DESCRIPTION
`agent_name`	`str`	Alias for `config.name`.
`runtime_name`	`str`	The `name` of the injected runtime (e.g. `"react@openai-agents"`, `"react@langchain"`, or `"google-adk-native"`).
`config`	`AgentConfig`	Static agent configuration (read-only).
`history`	`list[Message]`	Shallow copy of the current conversation. Safe to read; cannot be mutated.

```
async def run(user_message: str, *, run_config: RunConfig | None = None, **extra) → RunResult
```

Send a user message and execute a full agent run. Appends both the user turn and the final assistant reply to `history`.

PARAMETER	TYPE	DEFAULT	DESCRIPTION
`user_message`	`str`	—	The text sent as the user turn.
`run_config`	`RunConfig | None`	`None`	Per-call overrides merged on top of the agent's defaults.
`**extra`	`Any`	—	Forwarded into `RunConfig.extra`.

RETURNS

`RunResult` — full run output including `final_message`, `iterations`, tool calls, and timing.

```python
# Simple run
result = await agent.run("Who invented Python?")
print(result.final_message)
print(f"Took {result.duration_ms:.0f} ms, {result.iterations} iteration(s)")

# With per-call temperature override
result = await agent.run(
    "Write a haiku about Python.",
    run_config=RunConfig(temperature=0.9),
)
```

```
async def stream(user_message: str, *, run_config: RunConfig | None = None) → AsyncGenerator[StreamChunk, None]
```

Stream the agent response. Returns an async generator of `StreamChunk` objects. The final chunk always has `type == "done"`.

RAISES

`StreamingNotSupportedError` if the runtime does not support streaming.

```
async for chunk in await agent.stream("Explain async/await."):
    match chunk.type:
        case "text":
            print(chunk.content, end="", flush=True)
        case "tool_call":
            print(f"\n→ calling {chunk.tool_call.name}")
        case "tool_result":
            print(f"\n← {chunk.tool_result.name}: {chunk.tool_result.result}")
        case "done":
            print()  # newline after stream
```

```
async def chat(user_message: str, **kwargs) → str
```

Convenience wrapper around `run()`. Returns only the final text string. All keyword arguments are forwarded to `run()`.

```
reply = await agent.chat("What is 2 + 2?")
# → "4"
```

```
def clear_history() → None
```

Reset the conversation to a clean slate. Subsequent runs start with no prior context.

```
def inject_history(messages: list[Message]) → None
```

Seed the conversation with pre-existing messages, e.g. loaded from a database. Replaces any current history.

```
# Restore a conversation from storage
saved = load_conversation_from_db(session_id)
agent.inject_history(saved)

# Continue the conversation
reply = await agent.chat("Where were we?")
```

```
def add_tool(tool: ToolDefinition) → None
```

Dynamically register a tool after the agent has been built. The tool is added to `config.tools` and included in all subsequent runs.

```
def as_tool(* , name: str | None = None, description: str | None = None, parameters: dict | None = None, arg_name: str = "input", run_config: RunConfig | None = None, skills: list[Skill] | None = None, include_history: bool = False) → AgentTool
```

Expose the agent as a first-class tool without mutating the wrapped agent's history. By default the generated tool accepts a single string argument and runs against an isolated message list.

> **Agent tool note:** Set `include_history=True` when the delegated tool call should be seeded with the wrapped agent's current conversation history.

AgentBuilder

class AgentBuilder BUILDER

Fluent builder for `Agent` instances. Every method returns `self` for chaining. Call `.build()` last.

> **Note:** You must call one of `use_bridge()`, `use_adapter()`, or `use_runtime()` before `build()`, or a `RuntimeNotConfiguredError` is raised.

Identity methods

METHOD	PARAMETER	DESCRIPTION
`.with_name(name)`	`str`	Set the agent's display name. Default: `"unnamed-agent"`.
`.with_description(desc)`	`str`	Human-readable description for documentation/registries.
`.with_version(ver)`	`str`	Semantic version string. Default: `"1.0.0"`.

`.with_tags(*tags)`	`str...`	Arbitrary labels for grouping agents (e.g. `"prod"`, `"v2"`).

Behaviour methods

METHOD	PARAMETER	DESCRIPTION
`.with_system_prompt(p)`	`str`	System prompt prepended to every run.
`.with_tools(tools)`	`list[ToolDefinition]`	Replace the tool list. Passed to every `RunConfig`.
`.add_tool(tool)`	`ToolDefinition`	Append a single tool to the list.
`.with_tool_registry(registry, *, skill_name="tool-registry", description="...")`	`ToolRegistry`	Attach registry definitions and package registry handlers into an implicit skill. Useful for MCP-backed tool sets.
`.with_middlewares(middlewares)`	`list[AgentMiddleware]`	Replace the agent's static middleware chain.
`.add_middleware(middleware)`	`AgentMiddleware`	Append one middleware to the agent's static middleware chain.
`.with_defaults(**kw)`	`Any`	Override default `RunConfig` fields by name, e.g. `max_iterations=5`.

Runtime wiring methods

METHOD	PARAMETER	DESCRIPTION
`.use_bridge(runtime)`	`AgentRuntime`	Attach a Bridge-pattern runtime (any `AgentExecutor` subclass).
`.use_adapter(runtime)`	`AgentRuntime`	Attach an Adapter-pattern runtime.
`.use_runtime(runtime)`	`AgentRuntime`	Attach any object satisfying the `AgentRuntime` protocol.

```
def build() → Agent
```

Construct and return an `Agent`. Raises `RuntimeNotConfiguredError` if no runtime has been attached.

```python
from kneo_agent import AgentBuilder, ToolDefinition, RunConfig
from kneo_agent.patterns import BridgeAgentFactory

search_tool = ToolDefinition(
    name="web_search",
    description="Search the web.",
    parameters={"type": "object", "properties": {"query": {"type": "string"}}},
)

agent = (
    AgentBuilder()
    .with_name("Research Agent")
    .with_description("Searches the web and summarises findings.")
    .with_version("2.1.0")
    .with_tags("prod", "research")
    .with_system_prompt("You are a thorough research assistant.")
    .with_tools([search_tool])
    .with_defaults(max_iterations=8, temperature=0.3)
    .use_bridge(BridgeAgentFactory.for_openai(client, strategy="react"))
    .build()
)
```

Middleware APIs

Middleware is the SDK's cross-cutting interception layer. It combines shared mutable context objects with ordered `next(...)` delegation across runs, streams, model calls, and tool calls.

class BaseAgentMiddleware MIDDLEWARE BASE

Convenience base class with pass-through defaults for all middleware hooks. Override only the hooks you need.

```
async def wrap_run(context: AgentRunContext, handler) → RunResult
```

Wrap a full `Agent.run(...)` execution. Applies to Bridge, Adapter, Native, and workflow-backed runtimes.

```
async def wrap_stream(context: StreamContext, handler) →
AsyncGenerator[StreamChunk, None]
```

Wrap a full `Agent.stream(...)` execution. Middleware may short-circuit by returning its own async generator.

```
async def wrap_model_call(context: ModelCallContext, handler) → ModelResponse
```

Wrap one Bridge executor model invocation. Available only when Kneo owns the inner loop through Bridge runtimes.

```
async def wrap_tool_call(context: ToolCallContext, handler) → ToolResult
```

Wrap one Bridge executor tool dispatch. Useful for logging, guardrails, retries, or result rewriting.

Context Dataclasses MIDDLEWARE

TYPE	KEY FIELDS	DESCRIPTION
AgentRunContext	agent_name , runtime_name , user_message , messages , run_config , metadata	Shared mutable state for one run() call.
StreamContext	agent_name , runtime_name , user_message , messages , run_config , metadata	Shared mutable state for one stream() call.
ModelCallContext	executor_name , runtime_name , iteration , messages , run_config , metadata	State for one Bridge model call.
ToolCallContext	executor_name , runtime_name , iteration ,	State for one Bridge tool invocation.

	`messages` , `run_config` , `tool_call` , `metadata`	
`ModelResponse`	`text` , `tool_calls`	Normalized Bridge model-call result returned from `wrap_model_call(...)` .

> **Ordering:** Static middleware from `AgentBuilder` is applied first. Per-run middleware from `RunConfig(middlewares=[...])` is appended after it, so it executes deeper in the chain.

Observability

The `kneo_agent.observability` subpackage provides first-party OpenTelemetry support. It is gated behind the optional `[telemetry]` extra (`pip install "kneo-agent[telemetry]"`) so the core package keeps no telemetry dependencies.

> **Why this is a separate subpackage, not part of `kneo_agent.utils` :**
> 1. **Optional dependency boundary.** Every name re-exported from `kneo_agent.utils` is import-safe with just the core install. Hosting OTel-backed code there would either force `opentelemetry-api` into the core install or require conditional `try: import opentelemetry` guards in `utils/__init__.py` . A dedicated subpackage makes the `[telemetry]` boundary visually obvious — `from kneo_agent.observability import ...` clearly signals "this requires the extra".
> 2. **It is a middleware, not a helper.** `OpenTelemetryMiddleware` subclasses `BaseAgentMiddleware` and participates in run / stream / model-call / tool-call dispatch. Architecturally it belongs alongside `core/middleware.py` , not in the utility bin where `ToolRegistry` and message constructors live.
> 3. **Room to grow.** Observability tends to accrete: OTel metrics, log correlation, custom exporters, structured-logging adapters. Starting as its own subpackage gives it space without bloating `utils` . `configure_logging` and `get_logger` remain in `kneo_agent.utils` because they are thin stdlib wrappers with no optional deps and don't participate in the middleware contract. If logging ever grows OTel-aware features (log/trace correlation, OTLP log export), the cleaner move is to *also* extract logging here — possibly renaming this subpackage to `instrumentation` — rather than merge observability back into `utils` .

class OpenTelemetryMiddleware MIDDLEWARE

A `BaseAgentMiddleware` subclass that emits spans for all four hook points (run, stream, model call, tool call) following the OpenTelemetry GenAI semantic conventions.

```
def __init__(tracer: Tracer | None = None, *, record_arguments: bool = True, record_results: bool = False)
```

PARAMETER	TYPE	DESCRIPTION
`tracer`	`opentelemetry.trace.Tracer | None`	Pre-configured tracer. If `None`, a tracer named `"kneo_agent"` is acquired from the global tracer provider.
`record_arguments`	`bool`	Serialize tool-call arguments into `gen_ai.tool.call.arguments`. Disable when arguments may contain PII. Default: `True`.
`record_results`	`bool`	Attach tool results to spans. Off by default since results can be large or sensitive.

Span hierarchy

```
chat {agent_name}            (wrap_run)
├── chat iter=1              (wrap_model_call)
├── execute_tool {tool}      (wrap_tool_call)
└── chat iter=2              (wrap_model_call)
```

Span attributes

ATTRIBUTE	WHEN EMITTED	SOURCE
`gen_ai.system`	all spans	`runtime.name`
`gen_ai.operation.name`	all spans	`"chat"` for runs and model calls; `"execute_tool"` for tool spans
`gen_ai.agent.name`	run + stream spans	`Agent.name`

`gen_ai.tool.name`	tool spans	`tool_call.name`
`gen_ai.tool.call.id`	tool spans	`tool_call.id`
`gen_ai.tool.call.arguments`	tool spans (when `record_arguments=True`)	JSON-serialised `tool_call.arguments`, truncated at 4096 chars
`gen_ai.request.model`	any span	`RunConfig.extra["model"]` when set
`gen_ai.request.temperature`	any span	`RunConfig.temperature`
`gen_ai.usage.input_tokens`	run spans	`RunResult.metadata["usage"]["input_tokens"]` or `"prompt_tokens"` when surfaced by the runtime
`gen_ai.usage.output_tokens`	run spans	`RunResult.metadata["usage"]["output_tokens"]` or `"completion_tokens"` when surfaced by the runtime

Exceptions raised by downstream handlers are recorded on the active span via `record_exception` and the span status is set to `ERROR` before re-raising.

> **Install:** `pip install "kneo-agent[telemetry]"`. Importing the module without the extra installed raises `ImportError` at `OpenTelemetryMiddleware()` construction.

Workflows

Composite workflows provide a workflow-as-agent model. Each participant receives the full conversation history, appends its own contribution, and passes the updated message list to the next participant.

The workflow package is split by runtime type: `workflow.py` contains the graph runtime, `sequential_workflow.py` contains the default orchestration runtime, `concurrent_workflow.py`, `handoff_workflow.py`, and `group_chat_workflow.py` contain the specialized orchestration runtimes, and `builders.py` contains the workflow and orchestration builders.

Workflow GRAPH RUNTIME

General graph workflow with explicit executors and edges. Execution proceeds in supersteps: run the current executor set, merge results into shared history, then route along matching edges. Like the other workflow runtimes, it also satisfies `AgentRuntime` and can be wrapped as a regular agent.

CONSTRUCTOR PARAMETER	TYPE	DESCRIPTION
executors	dict[str, WorkflowComponent]	Registered workflow executors by name.
start_executor_ids	list[str]	Executor names scheduled in the first superstep.
edges	list[WorkflowEdge]	Directed routing edges between executors.

```
def as_agent(* , name: str | None = None, description: str = "",
system_prompt: str | None = None) → Agent
```

Wrap the graph workflow in the regular `Agent` facade so callers can use `run()`, `stream()`, and `chat()` through the existing architecture.

WorkflowEdge DATACLASS

Represents a directed edge between two workflow executors.

FIELD	TYPE	DESCRIPTION
source	str	Source executor name.
target	str	Target executor name.
condition	Callable | None	Optional route predicate evaluated after the source executor runs.
label	str	Optional human-readable edge label.

WorkflowComponent PROTOCOL

Protocol implemented by workflow participants. A participant must expose `name`, support `run(messages, config)`, and report whether it supports streaming.

MEMBER	TYPE	DESCRIPTION
`name`	`str`	Human-readable participant identifier.
`run(messages, config)`	`RunResult`	Execute the participant against the full conversation history.
`supports_streaming()`	`bool`	Whether the participant can stream.

Built-In Orchestration Runtimes FAMILY

Kneo Agent ships four participant-oriented orchestration runtimes with the same overall shape. Internally they share the same `OrchestrationBase`, and all of them satisfy both `WorkflowComponent` and `AgentRuntime`, can be nested, and can be wrapped as regular agents with `as_agent(...)`.

RUNTIME	PURPOSE	DEFAULT SHAPE
`SequentialWorkflow`	`Default orchestration runtime`	Runs participants in order against the evolving shared conversation.
`ConcurrentWorkflow`	`Fan-out orchestration`	Runs all participants against the same input snapshot in parallel.
`HandoffWorkflow`	`Selector-driven orchestration`	Chooses the next participant dynamically at runtime.
`GroupChatWorkflow`	`Round-robin orchestration`	Cycles through participants in a shared conversation for a configured number of rounds.

SequentialWorkflow ORCHESTRATION

The default orchestration runtime. It runs participants in order and shares the same workflow/runtime surface as the other built-in orchestration runtimes.

CONSTRUCTOR PARAMETER	TYPE	DESCRIPTION

`participants`	`list[Agent | AgentRuntime | WorkflowComponent | FunctionStep]`	Ordered workflow participants. The list must not be empty.
`name`	`str`	Workflow runtime name. Exposed as `runtime_name` when wrapped as an agent.
`description`	`str`	Human-readable description for docs or registries.

```
async def run(messages: list[Message], config: RunConfig) → RunResult
```

Execute all workflow participants in order. The final result aggregates the full conversation, tool results, total iterations, and workflow metadata such as `workflow_steps`.

```
def as_agent(* , name: str | None = None, description: str = "",
system_prompt: str | None = None) → Agent
```

Wrap the workflow in the regular `Agent` facade so callers can use `run()`, `stream()`, and `chat()` exactly as they would for any other agent.

ConcurrentWorkflow ORCHESTRATION

Fan-out orchestration. Runs all participants against the same input snapshot in parallel and merges their output back into shared history.

Uses the same participant-oriented runtime surface as `SequentialWorkflow`, but changes the execution policy from ordered execution to parallel fan-out.

HandoffWorkflow ORCHESTRATION

Dynamic specialist routing. A selector callback chooses which participant handles the next turn.

Uses the same participant-oriented runtime surface as `SequentialWorkflow`, but changes the execution policy to selector-driven routing.

GroupChatWorkflow ORCHESTRATION

Round-robin shared conversation. Participants speak in order for a configured number of rounds.

Uses the same participant-oriented runtime surface as `SequentialWorkflow`, but changes the execution policy to round-robin conversation turns.

FunctionStep WORKFLOW LEAF

Custom workflow participant backed by a Python callable. The handler receives the full conversation history and current `RunConfig`.

FIELD	TYPE	DESCRIPTION
`step_name`	`str`	Workflow step name.
`handler`	`Callable[[list[Message], RunConfig], WorkflowReturn]`	Function or coroutine used to generate the step output.
`description`	`str`	Optional human-readable description.

A handler may return `RunResult`, `Message`, `list[Message]`, `str`, or `None`.

`FunctionExecutor` is an exported alias for `FunctionStep` so workflow leaves can be named as executors when that terminology is clearer.

HumanInTheLoopStep WORKFLOW LEAF

Workflow participant that requests explicit human review or approval. It can resolve inline through a callback or pause execution by raising `HumanInterventionRequiredError`.

FIELD	TYPE	DESCRIPTION
`step_name`	`str`	Workflow step name.
`prompt`	`str | callable`	Prompt shown to the human reviewer. May be computed from the current messages and run config.
`resolver`	`callable | None`	Optional callback that returns the human response inline.

`response_role`	`str`	Message role used when the human response is appended. Default: `"user"` .

HumanInterventionRequest DATACLASS

Structured request object passed to human-step resolvers and surfaced in pause exceptions.

WorkflowBuilder FACTORY

Builder for graph workflows plus convenience helpers for workflow leaves and common orchestration patterns.

METHOD	RETURN	DESCRIPTION
`WorkflowBuilder(start_executor, *, name="workflow", description="")`	`WorkflowBuilder`	Create a graph workflow builder with an initial executor.
`.add_executor(executor)`	`WorkflowBuilder`	Register another executor in the graph.
`.add_edge(source, target, *, condition=None, label="")`	`WorkflowBuilder`	Add a routing edge between executors.
`.build()`	`Workflow`	Build the graph workflow.
`.step(name, handler, *, description="")`	`FunctionStep`	Create a custom function-based workflow participant.
`.human_step(name, prompt, *, resolver=None, description="", response_role="user")`	`HumanInTheLoopStep`	Create a workflow participant that requests explicit human input.
`.sequential(participants, ...)`	`SequentialWorkflow`	Convenience helper for the default linear orchestration runtime.
`.concurrent(participants, ...)`	`ConcurrentWorkflow`	Convenience helper for fan-out composition.

Orchestration Builders FACTORIES

BUILDER	BUILDS	DESCRIPTION
SequentialBuilder	SequentialWorkflow	Default linear orchestration runtime.
ConcurrentBuilder	ConcurrentWorkflow	Parallel fan-out orchestration.
HandoffBuilder	HandoffWorkflow	Selector-driven specialist routing.
GroupChatBuilder	GroupChatWorkflow	Round-robin shared conversation orchestration.

```
from kneo_agent import AgentBuilder
from kneo_agent.workflows import WorkflowBuilder

writer = AgentBuilder().with_name("writer").use_runtime(writer_runtime).build()
reviewer =
AgentBuilder().with_name("reviewer").use_runtime(reviewer_runtime).build()

editorial = WorkflowBuilder.sequential([writer, reviewer], name="editorial")
publish = WorkflowBuilder.step("publish", lambda messages, config: "Published.")

release = WorkflowBuilder.sequential([editorial, publish], name="release")
workflow_agent = release.as_agent(name="Release Workflow")
```

Core Data Classes

AgentConfig DATACLASS

Static metadata that describes *what the agent is*. Constructed by `AgentBuilder.build()`. Exposed via `agent.config`.

FIELD	TYPE	DEFAULT	DESCRIPTION

`name`	`str`	—	Agent display name.
`description`	`str`	`""`	Human-readable description.
`version`	`str`	`"1.0.0"`	Semantic version.
`system_prompt`	`str | None`	`None`	Default system prompt for all runs.
`tools`	`list[ToolDefinition]`	`[]`	Tool definitions passed to every run.
`middlewares`	`list[AgentMiddleware]`	`[]`	Static middleware chain attached to the agent.
`default_run_config`	`dict[str, Any]`	`{}`	RunConfig field overrides applied to every run.
`tags`	`list[str]`	`[]`	Arbitrary labels for grouping/filtering.

RunConfig DATACLASS

Per-run configuration describing *how a particular run behaves*. Merged: `AgentConfig.default_run_config` → `run_config` kwarg → `**extra` kwargs.

FIELD	TYPE	DEFAULT	DESCRIPTION
`max_iterations`	`int`	`10`	Maximum agentic loop iterations before stopping.
`temperature`	`float`	`0.7`	Sampling temperature forwarded to the model.
`system_prompt`	`str | None`	`None`	Overrides `AgentConfig.system_prompt` for this run.

`tools`	`list[ToolDefinition]`	`[]`	Overrides the agent's tool list for this run.
`middlewares`	`list[AgentMiddleware]`	`[]`	Additional middleware appended for this single invocation.
`extra`	`dict[str, Any]`	`{}`	Provider-specific key/value pairs forwarded verbatim (e.g. `tool_handlers` for OpenAI provider).

```
# Override temperature and add a per-run middleware/tool handler
result = await agent.run(
    "Summarise the latest AI news.",
    run_config=RunConfig(
        temperature=0.2,
        max_iterations=5,
        middlewares=[MyLoggingMiddleware()],
        extra={"tool_handlers": {"web_search": my_search_fn}},
    ),
)
```

RunResult DATACLASS

Everything produced by a completed `agent.run()` call.

FIELD	TYPE	DESCRIPTION
`final_message`	`str`	The agent's final text response.
`messages`	`list[Message]`	Full conversation as it stood at run end, including tool messages.
`iterations`	`int`	Number of LLM calls made in the loop.

`tool_calls_performed`	`list[ToolResult]`	All tool calls executed during the run.
`duration_ms`	`float`	Wall-clock time in milliseconds.
`metadata`	`dict[str, Any]`	Strategy-specific extras, e.g. `{"plan": "..."}` from Plan-Act, `{"stopped_by_max_iterations": True}`.

```
result = await agent.run("Research quantum computing.")

print(result.final_message)
print(f"Iterations: {result.iterations}")
print(f"Duration:   {result.duration_ms:.1f} ms")
print(f"Tools used: {[r.name for r in result.tool_calls_performed]}")

if result.metadata.get("plan"):
    print(f"Plan used:\n{result.metadata['plan']}")

if result.metadata.get("stopped_by_max_iterations"):
    print("Warning: loop was cut short by max_iterations.")
```

StreamChunk DATACLASS

A single unit yielded by a streaming agent run. Always check `chunk.type` before accessing optional fields.

FIELD	TYPE	DESCRIPTION
`type`	`"text" | "tool_call" | "tool_result" | "done"`	Discriminator. Always present.
`content`	`str | None`	Text token. Present only when `type == "text"`.

`tool_call`	`ToolCall | None`	Tool invocation details. Present only when `type == "tool_call"`.
`tool_result`	`ToolResult | None`	Tool execution result. Present only when `type == "tool_result"`.

Message DATACLASS

A single turn in an agent conversation.

FIELD	TYPE	DESCRIPTION
`role`	`"user" | "assistant" | "system" | "tool"`	Message role.
`content`	`str`	Message text.
`name`	`str | None`	Tool name. Populated for `role="tool"` messages.
`tool_call_id`	`str | None`	Links a tool result to its originating call. Populated for `role="tool"` messages.

ToolDefinition DATACLASS

FIELD	TYPE	DESCRIPTION
`name`	`str`	Tool name used by the model to call it.
`description`	`str`	Natural-language description shown to the model.
`parameters`	`dict[str, Any]`	JSON Schema object describing arguments.

ToolCall DATACLASS

FIELD	TYPE	DESCRIPTION

`id`	`str`	Unique call identifier (from the model).
`name`	`str`	Tool name to invoke.
`arguments`	`dict[str, Any]`	Parsed JSON arguments from the model.

ToolResult DATACLASS

FIELD	TYPE	DESCRIPTION
`tool_call_id`	`str`	Links back to the originating `ToolCall.id`.
`name`	`str`	Tool name.
`result`	`str`	Serialised result string returned to the model.

AgentRuntime PROTOCOL

The structural protocol that every runtime must satisfy. `AgentExecutor` subclasses (Bridge), platform-owned native runtimes, and `*Adapter` classes all implement this. The `Agent` class only ever depends on this protocol.

MEMBER	SIGNATURE	DESCRIPTION
`name`	`str (property)`	Human-readable identifier (e.g. `"react@openai-agents"`, `"react@langchain"`, or `"google-adk-native"`).
`run()`	`async → RunResult`	Execute a full agent run.
`stream()`	`def → AsyncGenerator[StreamChunk, None]`	Async generator yielding run chunks. Declared as `def` (not `async def`) so implementations can be async generator functions and call sites iterate with `async for`

		`chunk` in `runtime.stream(...)` directly.
`supports_streaming()`	`bool`	Whether `stream()` is supported.
`supports_tools()`	`bool`	Whether the runtime dispatches tool calls.

BridgeAgentFactory

class BridgeAgentFactory FACTORY

Static factory that constructs Kneo-owned Bridge runtimes. Strategy and platform are chosen independently when the underlying provider satisfies the Bridge `RuntimeImpl` contract.

> **Scope:** LangChain Bridge remains a true Bridge. Google ADK Bridge is a compatibility translation layer. OpenAI is kept as a compatibility alias and now returns a native runtime with a warning.

```
@staticmethod for_google_adk(adk_runner, session_id: str, strategy: BridgeStrategy = "react") → AgentRuntime
```

PARAMETER	TYPE	DESCRIPTION
`adk_runner`	`Any`	A `google.adk.runners.InMemoryRunner` or compatible instance.
`session_id`	`str`	ADK session identifier for this conversation.
`strategy`	`BridgeStrategy`	`"simple"`, `"react"`, or `"plan-act"`. Default: `"react"`.

Warning: this is a compatibility bridge over ADK-shaped payloads. For native Google ADK loop ownership, prefer `NativeRuntimeFactory.for_google_adk(...)` or

`AdapterAgentFactory.for_google_adk(...)` .

```
@staticmethod for_openai(openai_client = None, model: str = "gpt-4o",
strategy: BridgeStrategy = "react", runner = None) → AgentRuntime
```

PARAMETER	TYPE	DESCRIPTION
`openai_client`	`Any | None`	Optional `AsyncOpenAI` client used by the OpenAI Agents SDK chat-completions model wrapper.
`model`	`str`	Model identifier. Default: `"gpt-4o"` .
`strategy`	`BridgeStrategy`	Strategy label preserved in the runtime name.
`runner`	`Any | None`	Optional OpenAI Agents SDK runner override, mainly useful for tests.

Compatibility alias. This now returns the platform-owned OpenAI Agents SDK runtime and emits a warning. Prefer `NativeRuntimeFactory.for_openai(...)` .

```
@staticmethod for_langchain(chat_model, tool_registry: dict | None = None,
strategy: BridgeStrategy = "react") → AgentRuntime
```

PARAMETER	TYPE	DESCRIPTION
`chat_model`	`Any`	Any `langchain_core.language_models.BaseChatModel` subclass.
`tool_registry`	`dict[str, Callable] | None`	Tool name → callable mapping. Can also be registered post-construction via `LangChainImpl.register_tool()` .

`strategy`	`BridgeStrategy`	Loop strategy. Default: `"react"`.

```
@staticmethod custom(impl: RuntimeImpl, strategy: BridgeStrategy = "react") → AgentRuntime
```

Bridge a custom `RuntimeImpl` with any strategy. Use this to plug in a provider not natively supported by Kneo Agent.

```python
from kneo_agent.patterns import BridgeAgentFactory, NativeRuntimeFactory
from langchain_openai import ChatOpenAI

# True Bridge: Kneo loop over a LangChain chat model
lc_react_rt = BridgeAgentFactory.for_langchain(ChatOpenAI(), strategy="react")

# Native runtime: platform owns the loop
oai_native_rt = NativeRuntimeFactory.for_openai(model="gpt-4o", strategy="react")

print(oai_native_rt.name) # "react@openai-agents"
print(lc_react_rt.name) # "react@langchain"
```

NativeRuntimeFactory

class NativeRuntimeFactory FACTORY

Constructs platform-owned runtimes. Use this when the underlying SDK/runtime should own the loop instead of Kneo.

```
@staticmethod for_openai(openai_client = None, model: str = "gpt-4o", strategy: BridgeStrategy = "react", runner = None) → OpenAIAgentsRuntime
```

Returns the OpenAI Agents SDK-backed runtime. The SDK owns the loop; the strategy is preserved in the runtime name and for Plan-Act behavior.

```
@staticmethod for_google_adk(adk_runner, app_name: str, user_id: str,
session_id: str) → GoogleADKRuntime
```

Returns the native Google ADK runtime backed by `runner.run_async(...)`. Google ADK owns the loop.

```
from kneo_agent.patterns import NativeRuntimeFactory

oai_runtime = NativeRuntimeFactory.for_openai(model="gpt-4o")
adk_runtime = NativeRuntimeFactory.for_google_adk(runner, "app", "user-1",
"session-1")
```

AdapterAgentFactory

class AdapterAgentFactory FACTORY

Constructs Adapter-pattern runtimes that wrap existing platform objects. The platform manages its own loop; no strategy choice is needed or possible.

```
@staticmethod for_google_adk(adk_runner, app_name: str, user_id: str,
session_id: str) → GoogleADKAdapter
```

Wraps an ADK runner that exposes `run_async(app_name, user_id, session_id, new_message)` as an async generator.

```
@staticmethod for_openai(runner, agent_definition: dict) → OpenAIAgentsAdapter
```

Wraps an `@openai/agents` Runner. `agent_definition` requires at minimum `{"name": ..., "instructions": ...}`.

```
@staticmethod for_langchain(agent_executor) → LangChainAdapter
```

Wraps a LangChain `AgentExecutor` (or `CompiledGraph`) with `ainvoke` and optionally `astream`.

```python
from kneo_agent.patterns import AdapterAgentFactory

# Wrap an already-running LangChain AgentExecutor
lc_executor = build_langchain_agent()   # your existing code, untouched
runtime = AdapterAgentFactory.for_langchain(lc_executor)

agent = AgentBuilder().use_adapter(runtime).build()
reply = await agent.chat("What can you do?")
```

Bridge Executors

The Bridge abstraction side. These are Kneo-owned loop strategies. Inject a `RuntimeImpl` at construction time. Use Native runtimes when the platform should own the loop.

SimpleAgentExecutor EXECUTOR

One-shot completion. No loop, no tool calls. `name` → `"simple@<platform>"`.

```python
from kneo_agent import SimpleAgentExecutor
from kneo_agent.providers import LangChainImpl

impl = LangChainImpl(chat_model)
runtime = SimpleAgentExecutor(impl)
# runtime.name == "simple@langchain"
```

ReActAgentExecutor EXECUTOR

Reason → Act → Observe loop. Tool calls are executed concurrently with `asyncio.gather`. Terminates when the model returns no tool calls or `max_iterations` is reached. `name` → `"react@<platform>"`.

Loop invariants

- All tool calls within a single model response are dispatched in parallel.
- Tool results are appended as `role="tool"` messages before the next completion.
- If `max_iterations` is reached, `RunResult.metadata["stopped_by_max_iterations"] == True`.

```python
from kneo_agent import ReActAgentExecutor
from kneo_agent.providers import LangChainImpl

impl = LangChainImpl(chat_model)
runtime = ReActAgentExecutor(impl)
agent = AgentBuilder().with_tools([search_tool]).use_bridge(runtime).build()

result = await agent.run("What is the latest news on AI?")
if result.metadata.get("stopped_by_max_iterations"):
    print("Warning: loop hit max_iterations")
```

PlanActAgentExecutor EXECUTOR

Two-phase strategy: generate a numbered plan (tools disabled), then execute it with the full ReAct loop. The plan is stored in `RunResult.metadata["plan"]`. `name` → `"plan-act@<platform>"`.

When to use Plan-Act

- Complex tasks with 3+ sequential dependent steps.
- Tasks where implicit ReAct loops tend to lose track of the goal.
- When you want a human-readable trace of intent before execution.

```python
from kneo_agent import PlanActAgentExecutor
from kneo_agent.providers import LangChainImpl

runtime = PlanActAgentExecutor(LangChainImpl(chat_model))
agent = AgentBuilder().with_tools(tools).use_bridge(runtime).build()

result = await agent.run("Research and write a report on quantum computing.")
print("Plan:\n", result.metadata["plan"])
print("\nReport:\n", result.final_message)
```

Platform Adapters

The Adapter pattern side. Each class wraps a fixed-interface platform object and implements `AgentRuntime`. The platform owns the agentic loop.

GoogleADKAdapter ADAPTER

Wraps an ADK runner whose `run_async()` method yields typed events. ADK handles tool dispatch, retries, and multi-turn internally. `name` → `"google-adk-adapter"`.

```
from google.adk.runners import InMemoryRunner
from kneo_agent.runtime.adapters import GoogleADKAdapter

runner = InMemoryRunner(agent=my_adk_agent, app_name="my-app")
adapter = GoogleADKAdapter(runner, app_name="my-app", user_id="u-1",
session_id="s-1")
agent = AgentBuilder().use_adapter(adapter).build()
```

OpenAIAgentsAdapter ADAPTER

Wraps an `@openai/agents` Runner. Translates the SDK's `list[Message]` into the runner's single `input: str` and maps `new_items` back to `ToolResult[]`. `name` → `"openai-agents-adapter"`.

```
from agents import Agent, Runner    # @openai/agents
from kneo_agent.runtime.adapters import OpenAIAgentsAdapter

oai_agent = Agent(name="helper", instructions="Be helpful.", tools=[...])
runner = Runner()
adapter = OpenAIAgentsAdapter(
    runner,
    agent_definition={"name": "helper", "instructions": "Be helpful."},
)
agent = AgentBuilder().use_adapter(adapter).build()
```

LangChainAdapter ADAPTER

Wraps a LangChain `AgentExecutor`. Translates `list[Message]` → `(input, chat_history)` tuple format. `intermediate_steps` becomes `ToolResult[]`. `name` → `"langchain-adapter"`.

> **Streaming:** Only available if the executor exposes `astream()`.

```
from langchain.agents import AgentExecutor, create_openai_functions_agent
from kneo_agent.runtime.adapters import LangChainAdapter

lc_executor = AgentExecutor(agent=..., tools=[...])
adapter = LangChainAdapter(lc_executor)
kneo_agent = AgentBuilder().use_adapter(adapter).build()

# Multi-turn: the adapter correctly extracts chat_history from history
await kneo_agent.chat("Hello!")
await kneo_agent.chat("What did I just say?")   # history is carried forward
```

RuntimeImpl Protocol

RuntimeImpl PROTOCOL

The Implementor interface in the Bridge pattern. Concrete Bridge providers such as `GoogleADKImpl` (compatibility layer) and `LangChainImpl` implement this. Native runtimes such as `OpenAIAgentsRuntime` do not implement `RuntimeImpl` because the platform owns the loop directly.

METHOD	SIGNATURE	DESCRIPTION
`platform_name`	`str (property)`	Short platform ID (e.g. `"langchain"` or `"google-adk"`).
`complete()`	`async → (str, list[ToolCall])`	Send one completion request; return `(text, tool_calls)`.
`execute_tool()`	`async → str`	Dispatch a tool call and return its result.
`stream_tokens()`	`async generator → str`	Yield raw text tokens.
`supports_tools()`	`bool`	Whether tool calling is supported.

`supports_streaming()`	`bool`	Whether `stream_tokens()` is supported.

Providers

GoogleADKImpl PROVIDER

`kneo_agent.providers.google_adk` — Compatibility `RuntimeImpl` for Google ADK. Translates `list[Message]` / `RunConfig` into ADK-shaped payloads so Kneo Bridge executors can drive the loop.

Install: `pip install kneo-agent[google-adk]`

> **Important:** This is not the native Google ADK loop-owned runtime. For that, use `NativeRuntimeFactory.for_google_adk(...)` or `GoogleADKRuntime`.

```
GoogleADKImpl(adk_runner: Any, session_id: str)
```

```
from google.adk.runners import InMemoryRunner
from kneo_agent.providers import GoogleADKImpl
from kneo_agent import ReActAgentExecutor

runner = InMemoryRunner(agent=my_adk_agent, app_name="app")
impl    = GoogleADKImpl(runner, session_id="session-1")
runtime = ReActAgentExecutor(impl)       # or SimpleAgentExecutor,
PlanActAgentExecutor
```

OpenAIAgentsImpl NATIVE RUNTIME

`kneo_agent.providers.openai_agents` — OpenAI Agents SDK-backed native runtime. Uses the SDK's `Agent`, `Runner.run()`, and `Runner.run_streamed()` APIs directly.

Install: `pip install kneo-agent[openai]`

```
OpenAIAgentsImpl(openai_client: Any | None = None, model: str = "gpt-4o",
strategy: str = "react", runner: Any | None = None)
```

> **Tool execution:** Pass tool handlers in `RunConfig.extra["tool_handlers"]` as a `dict[str, Callable]`. They are converted into OpenAI Agents SDK `FunctionTool` instances.

```
from kneo_agent.patterns import NativeRuntimeFactory
from kneo_agent import RunConfig

runtime = NativeRuntimeFactory.for_openai(model="gpt-4o", strategy="react")

result = await agent.run(
    "Search for Python tutorials.",
    run_config=RunConfig(extra={"tool_handlers": {"web_search": my_search_fn}}),
)
```

LangChainImpl PROVIDER

`kneo_agent.providers.langchain` — Wraps any `BaseChatModel` subclass. Tool handlers are registered in a plain Python dict — no `StructuredTool` wrapping required.

Install: `pip install kneo-agent[langchain]`

```
LangChainImpl(chat_model: Any, tool_registry: dict | None = None)
```

```
def register_tool(name: str, handler: Callable) → None
```

```
from langchain_openai import ChatOpenAI
from kneo_agent.providers import LangChainImpl
from kneo_agent import ReActAgentExecutor

impl = LangChainImpl(ChatOpenAI(model="gpt-4o"))
impl.register_tool("get_weather", lambda args: f"22 °C in {args['city']}")

runtime = ReActAgentExecutor(impl)
```

Skill APIs

Skills are reusable capability bundles that compile into `RunConfig`. Kneo Agent supports in-memory skills, Agent Skills-compatible `SKILL.md` directories, skill discovery, and on-demand loading of progressive-disclosure resources such as `references/`, `scripts/`, and `assets/`.

Skill DATACLASS

`kneo_agent.core.skills` — reusable skill definition plus loader and resource helpers.

FIELD	TYPE	DESCRIPTION
`name`	`str`	Agent Skills-compatible skill name. Must match the parent directory name when loaded from disk.
`description`	`str`	Required short description from `SKILL.md` frontmatter.
`system_prompt`	`str | None`	Markdown body of the skill. This is the instruction payload compiled into agent runs.
`tools`	`list[ToolDefinition]`	Tool definitions declared by the skill.
`defaults / extra / tags`	`dict | list`	Default run settings, provider extras such as `tool_handlers`, and arbitrary routing labels.
`license / compatibility / metadata / allowed_tools`	`str | dict`	Agent Skills frontmatter metadata preserved by the loader.
`source_path`	`str | None`	Absolute path to the loaded `SKILL.md` file when sourced from disk.

```
classmethod from_path(path: str | Path) → Skill
```

Load a skill from either a skill directory or a direct `SKILL.md` file.

```
def with_tool_handlers(handlers: dict[str, Any]) → Skill
```

Return a copy of the skill with `extra["tool_handlers"]` merged in for runtime execution.

```
def resource_paths() → list[str]
```

List bundled skill resources under `references/`, `scripts/`, and `assets/`, relative to the skill root.

```
def read_resource(relative_path: str) → str
```

Read a bundled resource file safely relative to the skill directory. Path escapes are rejected.

```
def activation_prompt() → str
```

Render a structured activation payload containing the skill body and discoverable bundled resources. Useful when integrating with prompt-based skill activation flows.

```
from kneo_agent import load_skill

skill = load_skill("examples/skills/weather")
skill = skill.with_tool_handlers({"get_weather": get_weather})

print(skill.resource_paths())
print(skill.read_resource("references/REFERENCE.md"))
print(skill.activation_prompt())
```

Skill Discovery FUNCTIONS

`discover_skills`, `discover_default_skills`, and `load_skill` provide startup-time discovery and runtime activation for Agent Skills directories.

```
def load_skill(path: str | Path) → Skill
```

Load a skill from a skill directory or direct `SKILL.md` path.

```
def discover_skills(roots: list[str | Path], max_depth: int = 4) →
list[SkillCatalogEntry]
```

Scan one or more roots for `SKILL.md` files and return lightweight catalog metadata only.

```
def discover_default_skills(project_root: str | Path | None = None, user_home: str | Path | None = None, max_depth: int = 4) → list[SkillCatalogEntry]
```

Scan conventional project and user locations such as `.agents/skills` and `.claude/skills`.

```
SkillCatalogEntry(name: str, description: str, path: str) → dataclass
```

Lightweight startup-time metadata used for skill menus and registries without loading full resource trees.

> **Standard:** The loader expects an Agent Skills-style `SKILL.md` with YAML frontmatter and a Markdown body. Kneo Agent preserves progressive disclosure by listing bundled files but leaving resource loading explicit and on demand. See Agent Skills Guide below for the full integration model.

ToolRegistry

class ToolRegistry UTILITY

`kneo_agent.utils.tools` — Maps tool names to `(ToolDefinition, handler)` pairs. Thread-safe for reads. Supports sync and async handlers.

```
def register(definition: ToolDefinition, handler: Callable) → None
```

Explicitly register a tool definition and handler. Overwrites if the name already exists (logs a warning).

```
def register_agent_tool(tool: AgentTool) → None
```

Register an `AgentTool` returned by `Agent.as_tool()`.

```
def add_agent(agent: Agent, **kwargs) → AgentTool
```

Create an `AgentTool` from an existing agent, register it immediately, and return it.

```
@registry.tool(name, description, parameters) → decorator
```

Decorator that registers the decorated function as a tool. Returns the original function unmodified.

```
async def call(tool_call: ToolCall) → str
```

Dispatch a `ToolCall` to its handler. Supports both sync and async handlers. Returns the result coerced to a JSON string.

RAISES

`KeyError` if no handler is registered for `tool_call.name`.

```
def call_sync(tool_call: ToolCall) → str
```

Synchronous variant. Raises `RuntimeError` if the handler is async.

```
async def register_mcp_server(server: MCPServerConfig, *, prefix: str | None = None) → list[ToolDefinition]
```

Connect to an MCP server, discover its tools, and register proxy handlers for them in the registry.

```
async def aclose() → None
```

Close background MCP sessions and subprocesses owned by the registry.

Properties

PROPERTY	TYPE	DESCRIPTION
`definitions`	`list[ToolDefinition]`	All registered `ToolDefinition` objects (pass to `AgentBuilder.with_tools()`).
`names`	`list[str]`	All registered tool names.
`len(registry)`	`int`	Number of registered tools.
`"name" in registry`	`bool`	Check if a tool is registered.

```
def to_skill(name: str, description: str = "", system_prompt: str | None = None, defaults: dict | None = None, extra: dict | None = None, tags: list[str] | None = None) → Skill
```

Package the registry as a `Skill` with both tool definitions and `tool_handlers` merged into `extra`.

```python
import json
from kneo_agent.utils import ToolRegistry
from kneo_agent import ToolCall

registry = ToolRegistry()

# Synchronous handler
@registry.tool(
    name="get_weather",
    description="Return current weather for a city.",
    parameters={
        "type": "object",
        "properties": {"city": {"type": "string"}},
        "required": ["city"],
    },
)
def get_weather(args: dict) -> str:
    return json.dumps({"city": args["city"], "temp_c": 22})

# Async handler — both are supported
@registry.tool(
    name="get_time",
    description="Return the current UTC time.",
    parameters={"type": "object", "properties": {}},
)
async def get_time(args: dict) -> str:
    import datetime
    return datetime.datetime.utcnow().isoformat() + "Z"

# Agent-as-tool registration
weather_tool = weather_agent.as_tool(name="get_weather", arg_name="city")
registry.register_agent_tool(weather_tool)

# Use in agent
agent =
AgentBuilder().with_tools(registry.definitions).use_bridge(runtime).build()
```

```
# Dispatch directly (e.g. for testing)
tc = ToolCall(id="1", name="get_weather", arguments={"city": "Tokyo"})
result = await registry.call(tc)   # → '{"city": "Tokyo", "temp_c": 22}'
```

MCP

`kneo_agent.mcp` exposes lightweight MCP helpers for importing remote tools into the existing tool and skill pipeline.

MCPServerConfig DATACLASS

Connection description for one MCP server. Supported transports are `stdio`, `http`, and `sse`.

```
classmethod stdio(*, name, command, args = (), env = {}, cwd = None) → MCPServerConfig
```

```
classmethod http(*, name, url, headers = {}, timeout = 30.0, sse_url = None) → MCPServerConfig
```

```
classmethod sse(*, name, sse_url, message_url = None, headers = {}, timeout = 30.0) → MCPServerConfig
```

MCPClientSession CLIENT

Async MCP client session used by `ToolRegistry.register_mcp_server(...)`. Supports initialization, `tools/list`, and `tools/call`.

MCPTool DATACLASS

Lightweight MCP tool description with `name`, `description`, and `input_schema`, plus conversion to `ToolDefinition`.

> **Guide:** See MCP Guide below and the runnable examples `examples/12_mcp_stdio_filesystem.py` and `examples/13_mcp_http_sse.py`.

messages utilities

`kneo_agent.utils.messages` — Convenience constructors and inspection helpers for `Message` lists.

Constructors

`user(content)` `assistant(content)` `system(content)`

`tool_result(content, *, tool_call_id, name=None)`

Inspection helpers

`last_user_message(messages) → Message | None`

`last_assistant_message(messages) → Message | None`

`filter_by_role(messages, role) → list[Message]`

Serialisation

`to_dict_list(messages) → list[dict]` `from_dict_list(data) → list[Message]`

```python
from kneo_agent.utils.messages import (
    user, assistant, system, tool_result,
    last_user_message, to_dict_list, from_dict_list,
)
import json

# Build a history from scratch
history = [
    system("You are a helpful assistant."),
    user("What is the capital of France?"),
    assistant("Paris."),
]

# Serialise to JSON for storage
serialised = json.dumps(to_dict_list(history))

# Restore from JSON
loaded = from_dict_list(json.loads(serialised))
agent.inject_history(loaded)

# Inspect
last = last_user_message(history)
print(last.content)   # "What is the capital of France?"
```

logging utilities

`kneo_agent.utils.logging` — Helpers for the `kneo_agent.*` logger hierarchy. Kneo Agent never adds handlers automatically; you opt in by calling `configure_logging()`.

```python
configure_logging(level: str | int = "INFO", stream = sys.stderr) → None
```

Add a `StreamHandler` to the root `kneo_agent` logger. Idempotent — calling twice does not add duplicate handlers.

```
get_logger(name: str) → logging.Logger
```

Return a child logger named `"kneo_agent.<name>"`.

```python
from kneo_agent.utils import configure_logging, get_logger

# Enable debug output for all kneo_agent internals
configure_logging("DEBUG")

# Get a namespaced logger for your own code
log = get_logger("my_app")
log.info("Starting agent session.")

# Or control via standard logging
import logging
logging.getLogger("kneo_agent").setLevel(logging.WARNING)
```

Exceptions

```
KneoAgentError (Exception)
  ├─ ConfigurationError
  │    └─ RuntimeNotConfiguredError
  ├─ StreamingNotSupportedError
  ├─ ToolNotFoundError
  ├─ MaxIterationsReachedError
  └─ ProviderError
```

EXCEPTION	WHEN RAISED	NOTABLE ATTRIBUTES
KneoAgentError	Base. Catch this to handle all SDK errors.	—
ConfigurationError	Agent or runtime is misconfigured.	—

`RuntimeNotConfiguredError`	`AgentBuilder.build()` called without a runtime.	—
`StreamingNotSupportedError`	`agent.stream()` called on a non-streaming runtime.	—
`ToolNotFoundError`	Model requested an unregistered tool.	`.tool_name: str`
`MaxIterationsReachedError`	Raised optionally; executors also signal via metadata.	—
`ProviderError`	Wraps errors from an underlying LLM provider.	`.provider: str`, `.cause: Exception`

```python
from kneo_agent import KneoAgentError, StreamingNotSupportedError, ProviderError

try:
    result = await agent.run("Hello")
except StreamingNotSupportedError as e:
    print(f"Runtime does not stream: {e}")
except ProviderError as e:
    print(f"Provider '{e.provider}' failed: {e}")
    if e.cause:
        raise e.cause
except KneoAgentError as e:
    print(f"SDK error: {e}")
```

Writing a Custom Provider

Implement the `RuntimeImpl` protocol to integrate any LLM platform not natively supported. Your class only needs to satisfy the structural protocol — no inheritance required.

```python
from typing import AsyncGenerator
from kneo_agent import Message, RunConfig, ToolCall
from kneo_agent.patterns import BridgeAgentFactory

class MyCustomImpl:
    platform_name = "my-platform"

    def __init__(self, my_client):
        self._client = my_client

    async def complete(
        self, messages: list[Message], config: RunConfig
    ) -> tuple[str, list[ToolCall]]:
        # Translate messages → your platform's format
        response = await self._client.generate(
            prompt=messages[-1].content,
            max_tokens=1024,
        )
        return response.text, []   # (text, tool_calls)

    async def execute_tool(self, call: ToolCall, config: RunConfig) -> str:
        # Route tool calls to registered handlers
        handler = config.extra.get("tool_handlers", {}).get(call.name)
        if handler:
            return handler(call.arguments)
        return f'{{"error": "tool {call.name!r} not registered"}}'

    async def stream_tokens(
        self, messages: list[Message], config: RunConfig
    ) -> AsyncGenerator[str, None]:
        async for token in self._client.stream(messages[-1].content):
            yield token

    def supports_tools(self) -> bool: return True
    def supports_streaming(self) -> bool: return True

# Wire it up via BridgeAgentFactory.custom()
```

```
runtime = BridgeAgentFactory.custom(MyCustomImpl(my_client), strategy="react")
agent = AgentBuilder().use_bridge(runtime).build()
```

Streaming Guide

All streaming runtimes yield `StreamChunk` objects with four possible types. The stream always ends with `type == "done"`.

```python
import sys

async def stream_with_tool_trace(agent, message: str):
    collected_text = []
    tool_calls = []

    async for chunk in await agent.stream(message):
        match chunk.type:
            case "text":
                print(chunk.content, end="", flush=True)
                collected_text.append(chunk.content)

            case "tool_call":
                tc = chunk.tool_call
                print(f"\n[→ {tc.name}({tc.arguments})]")
                tool_calls.append(tc)

            case "tool_result":
                tr = chunk.tool_result
                print(f"[← {tr.name}: {tr.result[:60]}]")

            case "done":
                print()  # final newline

    return "".join(collected_text), tool_calls

text, tools = await stream_with_tool_trace(agent, "Research quantum computing.")
```

> **Note:** `agent.stream()` is itself async (because it may check streaming support before yielding), so always use `await` on the call and `async for` on the result.

Bridge vs Adapter — Decision Reference

Dimension	Bridge	Native	Adapter
Intent	Decouple strategy from platform so both can vary independently	Let the platform SDK/runtime own the loop directly	Convert an existing, fixed interface into the target protocol
When to use	Designing from scratch; need Kneo-owned strategies and a Bridge-compatible provider	Platform has a strong native runtime you want to preserve	Existing platform object already running; cannot change its interface
Strategy control	You own the loop (Simple / ReAct / Plan-Act)	Platform owns the loop; strategy is metadata or platform-specific behavior	Platform owns the loop; no strategy choice
Class count	M + N (not M×N): 3 + 3 = 6 classes for 9 variants	One runtime per native platform family	Exactly N classes (one per platform)
Runtime name	`"react@langchain"`	`"react@openai-agents"` or `"google-adk-native"`	`"openai-agents-adapter"`
Extensibility	New strategy → one class; works on all platforms automatically	New native runtime → one platform-owned implementation	New platform → one new adapter; no impact on others
Migration path	Use for Kneo-managed strategies	Use for OpenAI Agents SDK and native Google ADK execution	Use when migrating existing agents incrementally
Factory	`BridgeAgentFactory.for_*()`	`NativeRuntimeFactory.for_*()`	`AdapterAgentFactory.for_*()`
Builder method	`.use_bridge(runtime)`	`.use_runtime(runtime)`	`.use_adapter(runtime)`

Guides

These guide sections are embedded directly in the API reference so the reference manual reads as a single document instead of sending you to separate files.

Agent Middleware Guide GUIDE

Middleware is the SDK's cross-cutting interception layer. It is best for logging, guardrails, request mutation, tool-result rewriting, and short-circuiting.

`wrap_run(...)` and `wrap_stream(...)` apply to every runtime style.
`wrap_model_call(...)` and `wrap_tool_call(...)` apply only to Bridge runtimes, where Kneo owns the inner loop.
Static middleware comes from `AgentBuilder.add_middleware(...)` or `with_middlewares(...)`.
Per-run middleware comes from `RunConfig(middlewares=[...])`.

```
from kneo_agent import AgentBuilder, BaseAgentMiddleware, RunConfig

class LoggingMiddleware(BaseAgentMiddleware):
    async def wrap_run(self, context, handler):
        print(f"running {context.agent_name}")
        return await handler(context)

agent = (
    AgentBuilder()
    .add_middleware(LoggingMiddleware())
    .use_runtime(runtime)
    .build()
)

result = await agent.run(
    "hello",
    run_config=RunConfig(middlewares=[LoggingMiddleware()]),
)
```

> **Examples:** See `examples/16_agent_middleware_logging.py` and

> `examples/17_agent_middleware_short_circuit.py` .

Agent Skills Guide GUIDE

Kneo Agent supports both in-memory skills and disk-backed Agent Skills-compatible `SKILL.md` directories.

The Markdown body becomes `Skill.system_prompt` .
Frontmatter `tools` becomes `ToolDefinition[]` .
`defaults` contributes to `RunConfig` .
`extra` merges into `RunConfig.extra` .
Middleware is intentionally separate: skills compile into data, middleware stays executable policy.

```
from kneo_agent import load_skill

skill = load_skill("examples/skills/weather")
skill = skill.with_tool_handlers({"get_weather": get_weather})

print(skill.resource_paths())
print(skill.read_resource("references/REFERENCE.md"))
```

> **Progressive disclosure:** bundled `references/` , `scripts/` , and `assets/` stay discoverable but are not injected automatically into every run.

MCP Guide GUIDE

MCP is treated as an external tool source, not a separate runtime family.

Connect with `MCPServerConfig` .
Discover tools with `ToolRegistry.register_mcp_server(...)` .
Convert them into ordinary `ToolDefinition` plus proxy handlers.
Attach the registry with `AgentBuilder.with_tool_registry(...)` .
Bridge, Native, and Adapter runtimes can all use MCP tools through the existing `RunConfig` and `tool_handlers` path.

```python
from kneo_agent import MCPServerConfig
from kneo_agent.utils import ToolRegistry

registry = ToolRegistry()
await registry.register_mcp_server(
    MCPServerConfig.stdio(
        name="filesystem",
        command="npx",
        args=["-y", "@modelcontextprotocol/server-filesystem", "/tmp"],
    ),
    prefix="fs_",
)
```

> **Middleware pairing:** use run middleware for tracing or guardrails, and Bridge tool-call middleware when you want to inspect or rewrite MCP tool results before the next model turn.

Human In The Loop Guide GUIDE

Human review is modeled as a normal workflow participant rather than a separate runtime family.

Use middleware for automated policy, logging, and short-circuiting.
Use `WorkflowBuilder.human_step(...)` when a real person must approve, edit, or pause execution.
Human steps work in sequential workflows, graph workflows, nested workflows, and orchestration runtimes.

```python
from kneo_agent.workflows import WorkflowBuilder

review = WorkflowBuilder.human_step(
    "approval",
    "Approve this draft?",
    resolver=lambda request: "Approved by Alice",
)
```

> **Examples:** See `examples/14_workflow_human_in_loop_resolver.py` and `examples/15_workflow_human_in_loop_pause.py`.

Kneo Agent v1.2.0 · MIT License · Python 3.12+ · `pip install kneo-agent`

Run the test suite: `pytest -q tests` · Or: `make test`