# Open Agent Specification (Agent Spec) Technical Report

**Oracle Corporation**

## Abstract

**Open Agent Specification (Agent Spec)** is a declarative language that allows AI agents and their workflows to be defined in a way that is compatible across different AI frameworks, promoting portability and interoperability within AI Agent frameworks.

Agent Spec aims to resolve the challenges of fragmented agent development by providing a common unified representation that allows AI agents to be designed once and deployed across various frameworks, improving interoperability, reusability and reducing redundant development efforts. Additionally, Agent Spec facilitates development tools and portability, allowing AI agents to be created independently of their execution environment and enabling teams to exchange solutions without implementation-specific limitations.

Agent Spec benefits four key groups: (i) Agent developers, who gain access to a superset of reusable components and design patterns, enabling them to leverage a broader range of functionalities; (ii) Agent framework and tool developers, who can use Agent Spec as an interchange format and therefore benefit from the support of other frameworks as well as other tools; (iii) Researchers, who can achieve reproducible results and comparability, facilitating more reliable and consistent outcomes; (iv) Enterprises, which benefit from faster prototype-to-deployment as well as increased productivity, scalability, and maintainability for their AI agent solutions. This technical report provides an overview of the technical foundations of Agent Spec, including motivation, benefits, and future developments.

## 1 Introduction

This technical report introduces Agent Spec, a declarative, framework-agnostic configuration language designed to define AI agents and their workflows with high fidelity. By providing a unified representation, Agent Spec enables seamless portability and interoperability of agents across diverse frameworks, ensuring consistent behavior and execution across various AI Agent frameworks. Agent Spec serves as an abstraction layer above framework-specific specifications, acting as a unifying layer that encapsulates agent functionality beyond individual framework constraints.

### 1.1 Background & Motivation

Existing agentic frameworks have their own strengths and limitations. While flexibility is crucial to accommodating diverse agentic design patterns, currently available frameworks do not fully support all of them. Table 1 outlines key objectives a comprehensive framework should cover and describes how Agent Spec addresses these.

Moreover, different frameworks are usually parameterized and configured by different means. This makes porting solutions between frameworks a tedious and error-prone process, hindering collaboration and knowledge sharing among teams developing agentic applications. Consequently, a declarative, portable, and framework-agnostic representation of agentic designs is needed.

Table 1: Challenges and objectives in agentic frameworks and solutions that we define with Agent Spec.

| Topic | Objective | Agent Spec Solution |
|---|---|---|
| Agents | Author and edit conversational solutions such as ReAct [12] style agents. | Ability to define components that represent "resources" (with no impact on conversation flow routing) such as LLMs, memory, prompt templates, tools, other Agents which can be attached to other components' properties. |
| Flows | Author and edit graph-style workflows such as multi-step business processes. | Ability to define components that have a well-defined execution path such as a directed graph which may contain conditional routes or loops. |
| Control flow routing | Ability to define the flow of execution between components in the system, including conditional or looping flows. | Explicit "relationships" (edges) for control flow supporting 1-many and many-1 transitions. The specific transition to be taken is determined at runtime for a given execution. |
| I/O routing | Ability to define which outputs are consumed by which inputs for fixed flow components. This includes conditional or looping flows, and outputs that are consumed in multiple places. | Explicit "relationships" (edges) for data flow. Support for 1-many and many-1 transitions where the specific transition to be taken is determined at runtime for a given execution. |
| Nesting/Composition | Reuse components or solutions at any level of granularity in other solutions.<br><br>Includes multi-agent composition. | Definition of components that can be encapsulated and reused elsewhere. Established required properties for such components. Varying levels of visibility of exchanged messages between components. Orchestration of execution order and dependency either statically or dynamically. |
| Referencing | Ability to define a component instance once and refer to it in multiple places. | "Reference" syntax to specify property values by referencing and reuse. Provision of components for defining an element and for using them.<br>For example, an Agent component to define agent and an AgentNode in a Flow to use it in a certain place, or a Tool component to define a tool and a ToolNode in a Flow to use the tool in a specific place. |

For instance, ONNX [8] has revolutionized deep learning by providing a consistent way to port ML models across different frameworks (e.g., PyTorch [9], TensorFlow [11]). Similarly, Agent Spec aims to establish a unified representation for AI agents, enabling seamless interoperability and execution across diverse agentic frameworks. Just as ONNX allows models to be trained in one framework and executed in another, Agent Spec allows AI agents to be designed once and deployed across multiple platforms without reimplementation.

### 1.2 Objectives of Agent Spec

Agent Spec aims to define a common representation and act as a superset of capabilities for AI agents and workflows. Agent Spec graphs can be executed in different frameworks by means of runtimes that map Agent Spec components to the underlying framework. Agent Spec, along with importers and exporters for various underlying frameworks, enables transparent portability of agent workflows. Agent Spec also provides methods to validate agent workflows before execution, ensuring they are syntactically and logically correct and executable.

### 1.3 Key Benefits of Agent Spec

#### 1.3.1 Portability & Reusability

- **Framework-Agnostic:** Agent Spec abstracts agent definitions from specific implementations by providing a unified declarative representation, allowing agents to operate uniformly across multiple platforms (e.g., AutoGen, LangGraph, OCI Agents) without any need for reimplementation.
- **Modular Design:** Agent Spec's component-based structure fosters reusability and extensibility, enabling enterprises to build complex agentic systems by assembling and configuring standardized components.

- **Support for Complex Agentic Flows:** Agent Spec facilitates the creation of intricate agentic workflows by supporting various modular components and reusable Agent Spec-defined patterns, enabling streamlined development of advanced AI agent systems.
- **Portable Execution Graphs:** Agent Spec enables execution graphs to be described in a way that is independent of the underlying runtime implementation, ensuring seamless portability across frameworks. Converters between the representations of other frameworks to Agent Spec would simplify the porting of previously developed agents to Agent Spec compatible frameworks.

#### 1.3.2 Interoperability & Compatibility

- **SDK Support:** Agent Spec provides SDKs in various programming languages (starting with Python), supporting serialization and deserialization of agents into Agent Spec configurations. This simplifies development, deployment, and debugging across different environments.
- **Robustness & Consistency:** Agents defined in Agent Spec can be built on top of components that have been defined and validated within the Agent Spec specification, leading to higher reliability and predictability.
- **Define once, run on multiple frameworks:** Agentic systems leveraging Agent Spec can run on various frameworks through runtime adapters. Agent Spec's open specification allows developers to implement these adapters to support any framework, enabling agents to run in optimized environments.

#### 1.3.3 Unification across Frameworks

- **Common Format:** Agent Spec's common format simplifies integration of agents and enables easier agent sharing across platforms.
- **Knowledge Exchange:** The unified declarative format that Agent Spec provides helps developers as well as maintainers of agents to efficiently exchange solutions and ideas.
- **Conformance Test Suite:** The Agent Spec conformance test suite offers a comprehensive set of tests that cover the various functionalities and components available within the specification, such as Agent, Flow, and ToolNode. It includes illustrative validation examples of Agent Spec serialized configurations and verifies that these configurations produce consistent execution outcomes and behaviors across multiple runtime environments. This approach ensures interoperability and correctness of implementations supporting the Agent Spec specification.

## 2 Agent Spec Design

This section provides an overview of Agent Spec's declarative representation model. It details how Agent Spec functions across different frameworks and outlines implementation specifics, including the SDK structure, supported programming languages, agent definition using the SDK, and the serialization/deserialization of agents and workflows.

Agent Spec is intended to be a portable, platform-agnostic configuration language that allows Agents and Agentic Systems to be described with sufficient fidelity. It defines the conceptual objects, called components, that compose Agents in typical Agent systems, including the properties that determine the components' configuration, and their respective semantics.

Runtimes implement the Agent Spec components for execution with agentic frameworks or libraries. Agent Spec would be supported by SDKs in various languages (e.g., Python), enabling serialization and deserialization of Agents to different formats (e.g., JSON or YAML), or creating them from object representations with assurance of conformance to the specification.

An outline of the components involved in the Agent Spec ecosystem is depicted in Fig. 1.

Table 2: Agent Spec aims to do for AI agents what ONNX did for ML models

| Feature | ONNX (ML Models) | Agent Spec (AI Agents) |
|---|---|---|
| Scope | Unified representation of ML models, allowing portability across different deep learning frameworks. | Unified representation of AI agents and workflows, enabling them to run across diverse agentic frameworks. |
| Portability | ONNX allows ML models to be trained in one framework (e.g., PyTorch, TensorFlow) and executed in another. | Agent Spec enables AI agents to be built in one framework (e.g., LangGraph [10], AutoGen [6], OCI Agents [4]) and run in another without modification. |
| Unification | Provides a common declarative format which is versioned as an Opset. | Provides a common agent configuration format. |
| Extensibility | Supports various ML operations and optimizations | Defines modular components for scalable agent systems. |

Figure 1: Agent Spec's Design


### 2.1 Overview of Agent Spec's specification

The Agent Spec language specification defines structure and behavior of the conceptual building blocks, called components, that make up agents in typical agent-based systems. This structure can be serialized into a serialization language, like JSON or YAML. An overview of the language specification of Agent Spec follows, while the full specification can be accessed at `https://oracle.github.io/agent-spec/agentspec_language_spec_25_4_0.html`.

### 2.2 Components

In Agent Spec, components are the building blocks of the intermediate representation and are elements that are commonly used in agentic systems, such as (but not limited to) LLMs and tools.

Describing these components with specific types enables:

- Type safety + Usage hints for GUIs
  - *Example: if a component's property is expecting an LLM, only allow LLMs to be connected to it.*
  - *Example: highlight valid connections from one component to others in a graphical programming interface for agents.*
- Static analysis + validation
  - *Examples: if a component is a `Flow`, it must have a start node. If a component is an `Agent`, it must have an LLM.*
- Ease of programmatic use
  - `Component` families each have corresponding class definitions in the Agent Spec SDK. This allows consumers (agent execution environments, editing GUIs) to utilize concrete classes, rather than having to implicitly understand which type has which properties.

### 2.2.1 Base Component

The basic block in Agent Spec is a `Component`, which by itself can be used to describe any instance of any type, guaranteeing flexibility to adapt to future changes.

Note that Agent Spec does not need to encapsulate the implementation code it describes; it simply needs to be able to express enough information to instantiate a uniquely identifiable object of a specific type with a specific set of property values.

A `metadata` field contains all additional information that could be useful in different usages of the intermediate representation. For example, GUIs will require including some information about the components (e.g., position coordinates, size, color, notes, ...), so that they will be able to visualize them properly even after an export-import process. Metadata is optional and set to null by default.

**Symbolic references (configuration components)** When a component references another component entity (for example as a property value), there is a simple symbolic syntax to accomplish this, "`$component_ref:`" followed by the id of the other component.

This type of relationship is applicable to both Agent components as well as Flow components (a fixed flow component may also use an LLM or memory component).

We do not need a separate object for this type of reference, as the reference is explicitly assigned to some property or parameter.

```
"$component_ref:{COMPONENT_ID}"
```

**Input/output schemas** Components might need some input data to perform their task and expose some output as result. These inputs and outputs must be declared and described in Agent Spec, so that users, developers and other components of the agent are aware of what is exposed as inputs and outputs respectively. Note that we do not add input/output schemas directly to the base `Component` class, as there are a few cases where they do not really apply (for example `LLMConfig` and edges, please see the next sections).

**Input and output properties** Input and output schemas are used to define which values the different components accept as input or provide as output. We call these values "properties".

The term "properties" originates from JSON schema [5]. In the Agent Spec specification, we rely on this widely accepted and consolidated format. For more information about JSON schema definition, please check the official website at `https://json-schema.org/understanding-json-schema`.

In Agent Spec, a valid schema requires specific attributes:

- `title`: the name of the property
- `type`: the type of the property, following the type system described below

Additionally, users can specify:

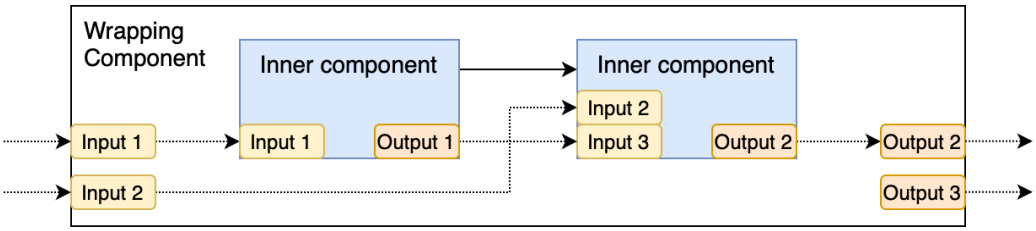
- `default`: the default value of this property
- `description`: a textual description for the property

These are the minimal set of attributes (or annotations, as per JSON schema terminology) that we require to be supported for Agent Spec compatibility.

**Type system** We rely on the typing system defined by the JSON schema format (see `https://json-schema.org/understanding-json-schema/reference/type`). At its core, JSON Schema defines the basic types available in most programming languages, though they may go by different names.

***Inputs and outputs of nested components*** In case of nested components (for example using an Agent component inside a Flow, or a Flow inside another Flow, or just a step in a Flow), the wrapping component is supposed to expose a (sub)set of the inputs/outputs provided by the inner components together with additional inputs/outputs it might generate as shown in Fig. 2. We require to replicate

Figure 2: Inputs and outputs of nested components and the component they are part of




the JSON schema of each value in the inputs/outputs lists of every component that exposes it, hence, if a wrapper component exposes some inputs and outputs of some of its internal components, it will have to include them in its own list of inputs/outputs as well. This makes the representation of the components a bit more verbose, but clearer and more readable. Parsers of the representation are required to ensure the consistency of the input/output schemas defined in it.

In the example above the wrapping Flow component (see sections below for more details about Flows) requires two inputs that coincide with two of the inputs exposed by the nested components and exposes one output among those exposed by the inner components in addition to one it computes itself.

***Validation of specified inputs/outputs*** Some input and output schemas are automatically generated by components based on their configuration. However, for the sake of clarity and readability of Agent Spec, we require that the descriptions of inputs and outputs are always reported in the intermediate representation. Consequently, when a serialized intermediate representation is read and imported by a runtime or an SDK, the inputs and outputs configuration of each component must be validated against the one generated by its configuration (if any).

**Specifying inputs through placeholders** Some components might infer inputs from special parts of their attributes. For example, some nodes (e.g., the `LLMNode`) extract inputs from their configuration attributes, so that their values can be adapted based on previous computations We let users define placeholders in some attributes, by simply wrapping the name of the property among a double set of curly brackets. The node will automatically infer the inputs by interpreting those configurations and the actual value of the input/output will be replaced at execution time. Whether an attribute accepts placeholders depends on the definition of the component itself. For example, setting the `prompt_template` configuration of an `LLMNode` to "You are a great assistant. Please help the user with this request: {{request}}" will generate an input called "request" of type string.

### 2.2.2 Agent

The `Agent` is the top-level component that holds shared resources such as conversation memory and tools. It also represents the entry point for interactions with the agentic system. Its purpose is to fill out the values for all the properties defined in the outputs attribute, with or without any interaction with the user.

It is important to have a separate definition for Agents as components, so that the same Agent component can be reused several times in a Flow or as part of a more complex agent design, without replicating its definition multiple times.

### 2.2.3 LLM

Large Language Models (LLM) are used in agents as generative models. To embed them into the agent, they need to be configurable flexibly with configuration parameters like the connection details, model id, the generation parameters, and more.

By default, we require only to specify the generation parameters that should be used when prompting the LLM. Specific extensions of `LLMConfig` for the most common model providers can be derived.

Of course, this is not limited to locally hosted models but in principle also enables the configuration of API endpoints like provided by OpenAI, Anthropic, Mistral etc.

#### 2.2.4 Tool

A tool is a procedural function or a Flow that can be made available to an Agent component to execute and perform tasks. The Agent component can decide to call a tool based on its signature and description. In a fixed Flow context, a node would need to be configured to call a specific tool.

In Agent Spec, we differentiate tools according to where the actual functionality is executed or run and specify three corresponding types:

- **ServerTools** are executed in the same runtime environment where the agent is being executed. The definitions of these tools therefore must be available to the agent's environment. It is expected that this type of tool will be limited in number, and generally in functionality.
- **ClientTools** are not executed by the runtime. The client environment must execute the tool and provide the results back to the runtime (like OpenAI's function calling model).
- **RemoteTools** are run in an external environment and are triggered by an RPC or REST call from the agent runtime.

Agent Spec is not supposed to provide an implementation of how these types of tools should work but provides their representation, so that they can be correctly interpreted and ported across platforms and languages. In principle a `Tool` is a function that is called providing some parameters (i.e., inputs), performs some transformation, and returns the result (i.e., outputs). The function has a name, and a description that can be used by an LLM for context when it is relevant to perform a task. Therefore, we define `Tool` as an extension of `ComponentWithIO`.

We let tools specify multiple outputs. In this case, the expected return value of the tool is a dictionary where each entry corresponds to an element specified in the output. The key of the dictionary entry must match the name of the property specified in the output. The runtime should parse the dictionary to extract the different outputs and bind them correctly. While `ServerTool` and `ClientTool` do not require specific additional parameters, the
`RemoteTool` requires including also the details needed to perform the remote call to the tool.

An important security aspect of Agent Spec is that we do not want to store arbitrary code in the agent's representation. The intermediate representation of an agent contains only the full description of the tool (including its attributes, inputs, outputs, etc.), but code is not included.

#### 2.2.5 Flow

Flows (or graphs) are directed, potentially cyclic workflows. They can be thought of as "subroutines" that encapsulate consistently repeatable processes. As such they provide more determinism and reliability compared to puristic Agent components at the cost of flexibility.

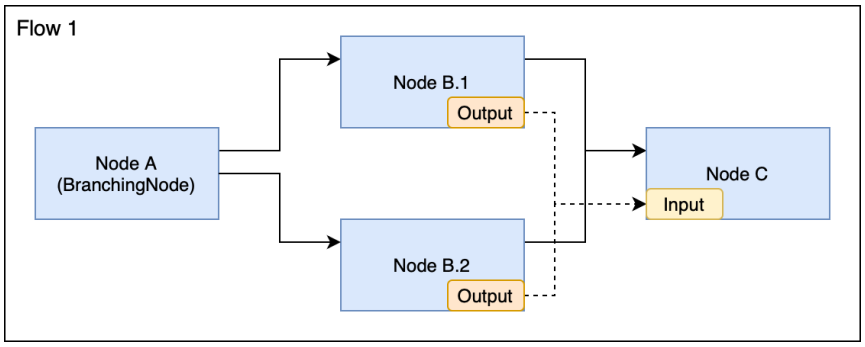
Flows are executable as tools by Agent components. Each Flow requires 0 or more inputs (any input edges expressed on the starting node of the graph) and may produce 0 or more outputs (any output edges expressed by the terminal node of the graph - note that a graph can have more than one terminal node, in the case of branching logic). As such, the I/O surface of a Flow is directly mutually exchangeable with that of a tool.

`Flow` objects specify the entry point to the directed graph, the nodes, and edges for the graph. Components inherent in the Flow are described in subsequent sections.

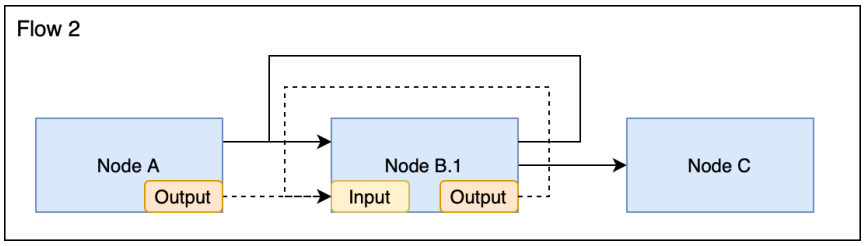
A Flow exposes the set of outputs that are available in all `EndNode` objects, and all inputs that are defined in its `StartNode`.

**Relationships / Edges** For Flows, Agent Spec defines separate relationships (edges) for both properties and control flow. This allows Agent Spec to unambiguously handle patterns such as conditional branches and loops, which are challenging to support with data-only flow control.

Relations are only applicable for Flows. All relations express a transition from some source node to some target node.




(a) Control flow with two branches that converge into the same node with a property filled based on the branch taken




(b) Control flow with a self-loop and a property updated by consequent execution of the same node

Figure 3: Possible connections formed by ControlFlowEdges and DataFlowEdges. ControlFlowEdges are shown as solid lines, DataFlowEdges with dotted lines.

### 2.2.6 ControlFlowEdge

A control relationship defines a potential transition from one branch of a node to another. The actual transition to be taken during a given execution is defined by the implementation of a specific node.

The `from_branch` attribute of a `ControlFlowEdge` is set to null by default, which means that it will connect the default branch (i.e., "*next*" ) of the source node.

### 2.2.7 DataFlowEdge

Considering an I/O system, a component id alone is not sufficient to identify a data relationship. To this purpose, a data relationship is needed to define what output property value should be used to fill an input property.

A component may take multiple input parameters and/or produce multiple output values. We must be able to determine what value or reference is mapped to each input and where each output is used. Destinations (Inputs of the destination node) can be connected to outputs of another node or to static values. Sources (Outputs of the source node) can be connected to inputs of another node or left unconnected.

**Connecting multiple data edges** In the definition of a Flow, it is allowed to connect two or more different data outputs to the same input as shown in Fig. 3. The last node that was executed and that has an output connected to the input will have the priority. In other words, the behavior is like having the input exposed as a public variable and every node that has an output connected to that input updates its value.

For the control flow, instead, edges just define which are the allowed "next step" transitions: multiple connections from the same outgoing control flow branch are not allowed, and they do not involve any ambiguity (note that we do not enable parallelism through edges).

**Optionality of data relationships** Some frameworks do not require specifying the data flow explicitly as they assume that all the properties are publicly exposed in a shared "variable space" that

any component can access, and the read/write strategy is simply name based. In practice, this means that:

- When a component has an input with a specific name, it will access the public variable space to read the value of the variable with that name.
- When a component has an output with a specific name, it will write (or overwrite if it already exists) into the public variable space the variable with that name.

We let users adopt this type of I/O system by setting the `data_flow_connections` parameter of the Flow to `None`. In this case, the name-based approach explained above will be adopted, by using the shared variable space described above as the public space where values are published.

Note that this name-based approach can be expressed defining explicitly data flow connections (but the opposite is not possible) by connecting all the outputs to the inputs with the same name, following the control flow connections in the Flow to account for overwrites.

**Flow state** We define as "flow state" the set of all values that describe at what point of execution the flow is. We let users define the set of values that are publicly exposed in the flow state to all nodes.

These values are treated differently compared to values that are exposed through inputs and outputs of nodes:

- They are always available in every node and do not need to be connected by a `DataFlowEdge`
- If a component writes an output which name matches a flow state value, the respective value in the flow state is overwritten

**Conversation** At the core of the execution of a conversational agent, there's the conversation itself. The conversation is implicitly passed to all the components throughout the flow. It contains the messages being produced by the different nodes: each node in a flow and each agent can append messages to the conversation if needed. Messages have, at least, a type (e.g., system, agent, user), content, the sender, and the recipient.

### 2.2.8 Node

A Node is a vertex in a Flow and is derived from `ComponentWithIO`. Agent Spec provides a library of nodes designed to streamline workflow orchestration, enabling rapid development and seamless integration for a wide range of applications:

- `LLMNode`: uses a LLM to generate some text from a prompt.
- `APINode`: performs an API call with the given configuration and inputs.
- `AgentNode`: runs a multi-round conversation with an Agent component, used to better structure agents and easily reuse Agent components.
- `FlowNode`: runs a Flow, used to better structure agents and easily reuse Flows.
- `MapNode`: performs a map-reduce operation on a given input collection, applying a specified Flow to each element in the collection.
- `StartNode`: entry point of a Flow.
- `EndNode`: exit point of a Flow.
- `BranchingNode`: allows conditional branching based on the value of an input.
- `ToolNode`: executes a tool.

A more detailed description of each node type is given at `https://oracle.github.io/agent-spec/agentspec_language_spec_25_4_0.html#standard-library-of-nodes-to-use-in-flows`.

### 2.3 Agent Spec Representation Model

The objective of Agent Spec is to be agnostic with respect to any agentic framework and the programming language used to build and execute an agent. For this reason, an agent represented with Agent Spec should be exportable to, and importable from an intermediate representation that is specified in a common serialization language. To this extent, Agent Spec adopts JSON as the designated language for the serialization of its components.

All building blocks of Agent Spec are designed with the goal of being trivially serializable. It follows that the Agent Spec representation of a component can be obtained by serializing its definition according to the JSON specification. In particular, the serialization of a component includes its attributes, plus an additional field called `component_type` that will be used at deserialization time to infer the correct component type to deserialize.

Note that, as defined in the Agent Spec language specification, components can be declared only once in the representation and then referenced according to the rules described in the Symbolic references section of this document.

A few examples of component JSON serializations are given at `https://oracle.github.io/agent-spec/agentspec_language_spec_25_4_0.html#language-examples`.

### 2.4 How Agent Spec Works Across Frameworks

Agent Spec serves as an abstraction layer above framework-specific implementations, acting as a unifying layer that encapsulates agent functionality beyond individual framework constraints.

Hence, the goal of Agent Spec is to define behavioral patterns for the different components that should be implemented according to the specificities of the different agentic frameworks. An Agent Spec runtime for a specific framework should be able to read the Agent Spec representation of a component (e.g., a Flow or an Agent), and then execute its logic, adhering to the behavior described in the Agent Spec specification.

As an alternative, if the agentic framework - used to implement the runtime - offers a serialization mechanism, it is also possible to build a translation script between the Agent Spec representation of an agent and the equivalent framework specific representation, then execute the runtime on the latter.

### 2.5 Implementation Details

#### 2.5.1 Agent Spec SDKs

To facilitate the process of building programmatically framework-agnostic agents, Agent Spec SDK packages can be conveniently implemented. These SDKs should provide two functionalities:

- Building Agent Spec component abstractions through the implementation of the relevant interfaces, in conformity with the Agent Spec specification.
- Import and export these abstractions from and to their respective serialized version in JSON format.

As part of the Agent Spec effort, we provide Agent Spec's Python SDK that we call PyAgentSpec. PyAgentSpec allows building Agent Spec compliant agents in python. Users can create their own assistants using component classes that replicate the interface and the behavior defined by Agent Spec, and they can finally export them in JSON format. Here is an example of how the definition of a simple agentic Flow prompting an LLM with a custom user input and returning the LLM's output would look like in PyAgentSpec.

```
from PyAgentSpec.property import Property
from PyAgentSpec.flows.flow import Flow
from PyAgentSpec.flows.edges import ControlFlowEdge, DataFlowEdge
from PyAgentSpec.flows.nodes import LlmNode, StartNode, EndNode
from PyAgentSpec.llms import VllmConfig
llm_config = VllmConfig(
    name="<Your Model Name Here>",
    url="<url.of.your.llm.deployment:port>",
```

```
    model_id="<provider/model-identifier>",
)
prompt_property = Property(
    json_schema={"title": "prompt", "type": "string"}
)
llm_output_property = Property(
    json_schema={"title": "llm_output", "type": "string"}
)
start_node = StartNode(name="start", inputs=[prompt_property])
end_node = EndNode(name="end", outputs=[llm_output_property])
llm_node = LlmNode(
    name="simple llm node",
    llm_config=llm_config,
    prompt_template="{{prompt}}",
    inputs=[prompt_property],
    outputs=[llm_output_property],
)
flow = Flow(
    name="Simple prompting flow",
    start_node=start_node,
    nodes=[start_node, llm_node, end_node],
    control_flow_connections=[
        ControlFlowEdge(name="start_to_llm", from_node=start_node,
    to_node=llm_node),
        ControlFlowEdge(name="llm_to_end", from_node=llm_node, to_node=
    end_node),
    ],
    data_flow_connections=[
        DataFlowEdge(
            name="prompt_edge",
            source_node=start_node,
            source_output="prompt",
            destination_node=llm_node,
            destination_input="prompt",
        ),
        DataFlowEdge(
            name="llm_output_edge",
            source_node=llm_node,
            source_output="llm_output",
            destination_node=end_node,
            destination_input="llm_output"
        ),
    ],
)
```

To export the Flow that was just built in JSON format, the code is the following.

```
serializer = AgentSpecSerializer()
serialized_flow = serializer.to_json(flow)
```

The full documentation of PyAgentSpec, including installation instructions, APIs documentation and examples, is available at oracle.github.io.

### 2.5.2 Agent Spec SDK Plugins

The PyAgentSpec SDK also supports the concept of plugins, because we expect that teams using Agent Spec may want to create new component types to be included in their configurations. To this end, a plugin can enable newer components in addition to the standard components of Agent Spec, simply by specifying the list of component types that the plugin includes and by providing the serialization and deserialization logic for the plugin. Such plugins can be implemented for components not yet included in the first version of Agent Spec, such as multi-agent patterns, or planning and memory components. See below the abstract interface to be implemented for a serialization plugin (the deserialization interface follows the same logic).

```python
from abc import ABC, abstractmethod
from typing import Any, Dict, List

from PyAgentSpec.component import Component
from PyAgentSpec.serialization.serializationcontext import SerializationContext

class ComponentSerializationPlugin(ABC):
    """Base class for Component serialization plugins."""

    @property
    @abstractmethod
    def plugin_name(self) -> str:
        """Return the plugin name."""
        pass

    @property
    @abstractmethod
    def plugin_version(self) -> str:
        """Return the plugin version."""
        pass

    @abstractmethod
    def supported_component_types(self) -> List[str]:
        """Indicate what component types the plugin supports."""
        pass

    @abstractmethod
    def serialize(
        self, component: Component, serialization_context: SerializationContext
    ) -> Dict[str, Any]:
        """Serialize a component that the plugin should support."""
        pass
```

### 2.5.3 Agent Spec Runtime Adapters

A runtime implementation makes Agent Spec components abstractions runnable by implementing the behavioral logic described in the Agent Spec language specification.

As previously discussed, there are several ways to implement runtimes for Agent Spec. Ideally, a runtime implementation is an agentic framework that natively supports Agent Spec: it should read an Agent Spec JSON specification and transform it directly into an equivalent component that can be executed.

As an example of an Agent Spec runtime implementation, we provide a package called `WayFlow`, which is a powerful, intuitive Python library for building sophisticated AI-powered assistants. It includes a library of plan steps to streamline the creation of AI-powered assistants, supports reusability, and it is ideal for rapid development. The `WayFlow` framework provides a serialization and deserialization interface to import and export Agent components according to the Agent Spec specification. It is a reference framework for Agent Spec and our runtime of reference, with native support for all Agent Spec Agents and Flows.

When an agentic framework does not allow reading Agent Spec representations directly, a translation layer that transforms the Agent Spec representation into a framework-specific representation could be adopted. This layer, that we can call Agent Spec Runtime adapter, loads an Agent Spec representation and transforms it into an equivalent component of the specific agentic framework.

Runtime adapters enable existing agentic frameworks that do not support Agent Spec natively executing Agent Spec representations. For example, we also provide runtime adapters for some of the most popular agentic frameworks, including, but not limited to, LangGraph and Microsoft AutoGen. These adapters load Agent Spec representations taking advantage of the Agent Spec Python SDK, and they transform the Agent Spec components into the respective versions defined in the agentic framework of interest.

### 2.6 Performance Evaluation & Benchmarks

Agent Spec enables key aspects of productizing agent-driven solutions, including comparison and evaluation of agent task performance across different executing frameworks and models, as well as optimizations (including costs) of execution for underlying hardware and frameworks. By covering commonly accepted concepts in agentic systems, such as workflows and agents, Agent Spec allows developers to map these concepts to various execution frameworks, facilitating comparison of framework specific task performance. Compute optimization is also possible, as Agent Spec enables hardware-optimized and framework-optimized implementation of agentic design patterns.

#### 2.6.1 Portability Analysis

Agentic AI is rapidly evolving, and a unified benchmark for assessing agent reliability, particularly in enterprise settings, is currently lacking. Future updates for Agent Spec report will include empirical results demonstrating the reliability of Agent Spec-defined agents across different frameworks.

`https://oracle.github.io/agent-spec/howtoguides/howto_execute_agentspec_across_frameworks.html` provides an example of how an agent defined with Agent Spec can be run in different frameworks. In this Retrieval Augmented Generation case study, an agent utilizing the same tools is instantiated in AutoGen, LangGraph, and WayFlow. In a practical setting, a specific mapping from Agent Spec to each framework would be implemented.

#### 2.6.2 Use Cases & Real-World Applications

A common use case of Agent Spec is the porting of agent-based assistants from one framework to another. Let's take as an example a team that implemented agent-based workflows using AutoGen. As business and process needs to change, evaluations are conducted to consider alternative frameworks. Due to AutoGen's specific architecture, migrating to another agentic framework proves to be complex with significant rewriting overhead. In this example, Agent Spec enables the definition of business unit-specific agentic systems using a declarative representation. This allows for conversion to specific framework formats or flexible execution across different agentic frameworks with appropriate adapters, enabling straightforward comparison of task performance and runtime cost depending on the execution framework. Consequently, utilizing Agent Spec throughout the agent design and deployment process reduces complexity, costs, implementation effort, and potential inconsistencies.

Another use case involves developing and deploying Agents across varied compute environments, such as cloud platforms and on-premise datacenters. In the cloud, Agents may run virtualized or serverless as part of a scalable system design. In an on-premise scenario, they may run within an application server. Agent Spec's framework and runtime agnosticism enables development of the Agent in a local development environment, followed by flexible deployment across either of these scenarios.

A different compelling real-world application involves advanced multi-agent systems in the domain of financial crime compliance. By integrating generative AI agents, it enables to automate key investigative processes - supplementing human investigators with narrative-driven analyses, uncovering critical insights, and recommending evidence-based actions. In this context, a comprehensive Anti-Money Laundering platform may comprise specialized agents tasked with sanctions screening, transaction monitoring for suspicious activity, and the generation of enhanced due diligence reports. These sub-agents are often developed by independent teams using a variety of frameworks and technologies, and may need to be deployed across heterogeneous environments, including cloud, on-premises, or even within database systems. Leveraging Agent Spec's framework-agnostic and declarative approach, development teams can seamlessly integrate and orchestrate these diverse agent workflows, reducing integration overhead and enabling consistent, scalable deployment. This holistic and automated strategy minimizes manual effort, decreases operational risk, and empowers financial institutions to respond more rapidly and consistently to evolving financial crime patterns, regardless of the underlying deployment environment.

### 2.7 Positioning in the Ecosystem

Agent Spec aims to streamline the architecture and design of agentic assistants and workflows, serving as an intermediate representation that abstracts away implementation details of specific

agentic frameworks. It is a portable configuration language that describes agentic design patterns, components and expected behavior. However, Agent Spec is not the only effort aimed at unifying the different parts that compose a common agentic ecosystem.

- The Model Context Protocol (MCP) [7], introduced by Anthropic, standardizes resource and data provision to agents. Resources are made available with a client/server based Restful API.
- Google's Agent2Agent Protocol [2] and BeeAi's Agent Communication Protocol (ACP) [3] propose standardized APIs for distributed agent communication.
- Nvidia recently announced NeMo Agent toolkit which is a framework independent library that treats "every agent, tool, and agentic workflow as a function call - enabling composability between these agents, tools, and workflows" [1].The idea is to compose existing agentic solutions and resources, potentially written using different frameworks, into new systems.

While protocols like MCP and A2A standardize tool or resource provisioning as well as inter-agent communication, Agent Spec complements these efforts by enabling standardized configuration of components related to agentic system design and execution in general, as shown in Fig. 4. Hence, Agent Spec provides strong synergies with other standardization efforts through its abstract definition of agentic system design. In the context of ongoing unification efforts within the agentic system landscape, Agent Spec aims to serve as the "common foundation" that connects these initiatives, enhancing their effectiveness and fostering a more cohesive ecosystem. Support for the aforementioned protocols may be added in future versions of Agent Spec as new components.

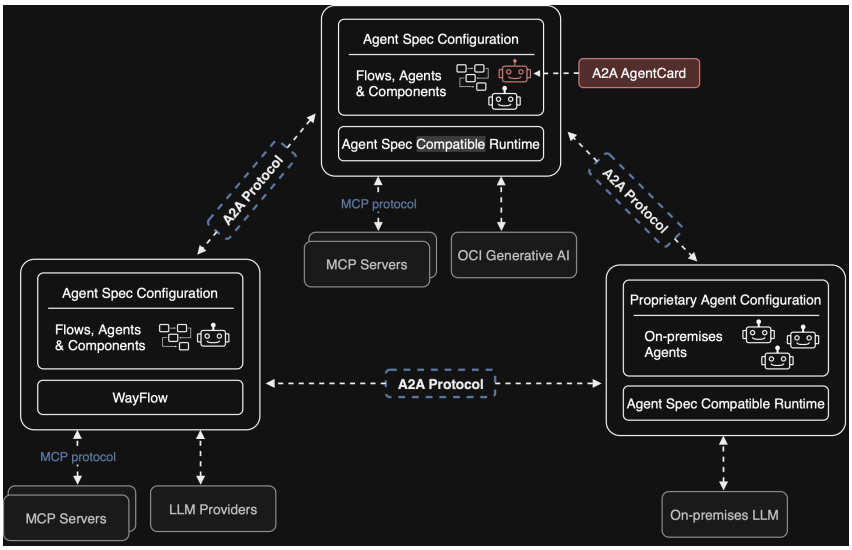



Figure 4: Agent Spec complements other standardizations, such as MCP or A2A

### 2.8 Future Roadmap & Enhancements

#### 2.8.1 Upcoming Features

**Expanding the Agent Spec language specification** We are looking forward to extending Agent Spec with new concepts and components in the upcoming versions of Agent Spec. A few notable ones include:

- Memory
- Planning
- Datastores
- Support for remote agents (e.g. A2A agents)
- Support for new types of tools (e.g. MCP tools)

**Extending runtimes support**

We want to foster the implementation of Agent Spec runtime adapters for the popular agentic frameworks in the vibrant AI community. At the same time, we aim to offer a conformance test suite for runtimes to ensure consistent behavior in accordance with the Agent Spec specification.

**Improving user experience**

Even though Agent Spec SDKs simplify the creation and validation of Agent Spec representations, creating agents by writing code might be complex and error-prone. We are working on a Drag&Drop UI that will allow users to build their own agents in a visual interface and then export the Agent Spec representation to be used in the framework of their choice via a runtime adapter.

### 2.8.2 Community & Ecosystem Growth

We strongly believe Agent Spec will contribute to streamlining agent development and open-sourcing its specification to encourage community contributions. This will foster a rich ecosystem of interchangeable and reusable designs and tools for agentic AI development. Integration of Agent Spec by other third-party open-source projects as an agent design language and representation will benefit the open-source community, which is currently evolving and lacks established common unified representation. In summary, the key aspects of impact on community growth are:

- Design patterns that can be easily shared & reused to collaboratively build increasingly complex solutions.
- Enable open exchange of building blocks and agentic patterns.
- Encourage contributions by the open-source community.

To help guide the evolution of the Agent Spec specification in alignment with its core design principles, we will establish a steering committee. Similar to ONNX principles, this committee will work closely with the community to review proposed contributions, foster open discussion, and publish versioned updates.

## 3 Contributors and Acknowledgements

Please cite this work as "Agent Spec (2025)".

Yassine Benajiba, Cesare Bernardis, Vladislav Blinov, Paul Cayet, Hassan Chafi, Abderrahim Fathan, Louis Faucon, Damien Hilloulin, Sungpack Hong, Ingo Kossyk, Rhicheek Patra, Sujith Ravi, Jonas Schweizer, Jyotika Singh, Shailender Singh, Xuelin Situ, Weiyi Sun, Jerry Xu, Ying Xu.[1]

We are also thankful to our Oracle colleagues who engaged in active discussions during the design of Agent Spec and provided valuable feedback that helped us refine its specifications.

## References

[1] NVidia. AgentIQ. [Online]. Available: https://developer.nvidia.com/nemo-agent toolkit.

[2] Google. Agent2Agent Protocol. [Online]. Available: https://developers.googleblog.com/en/a2a-a-new-era-of-agent interoperability/.

[3] BeeAI. Agent Communication Protocol. [Online]. Available: https://docs.beeai.dev/acp/alpha/introduction.

[4] OCI. Available: https://docs.oracle.com/en-us/iaas/Content/generative-ai agents/overview.htm.

[5] JSON-schema. [Online]. Available: https://json-schema.org/.

[6] Microsoft. AutoGen. [Online]. Available: https://microsoft.github.io/autogen/stable//index.html.

[7] Anthropic. Model Context Protocol. [Online]. Available: https://modelcontextprotocol.io/introduction.

[8] ONNX. [Online]. Available: https://onnx.ai/.

[1] Author list is sorted alphabetically.

[9] PyTorch. [Online]. Available: https://pytorch.org/.

[10] LangGraph. [Online]. Available: https://www.langchain.com/langgraph.

[11] TensorFlow. [Online]. Available: https://www.tensorflow.org/.

[12] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In International Conference on Learning Representations (ICLR), 2023.