ETSI GR ENI 056 V4.1.1 (2025-10)

GROUP REPORT

Experiential Networked Intelligence (ENI); Study on Multi-Agent Frameworks for Next-Generation Core Networks

Disclaimer

The present document has been produced and approved by the Experiential Networked Intelligence (ENI) ETSI Industry Specification Group (ISG) and represents the views of those members who participated in this ISG.
It does not necessarily represent the views of the entire ETSI membership.

Reference
DGR/ENI-0056v411_Stud_MultIA

Keywords
6G, AI, AI-native, GenAI

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI IPR online database.

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This Group Report (GR) has been produced by ETSI Industry Specification Group (ISG) Experiential Networked Intelligence (ENI).

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The present document studies Multi-Agent Systems (MASs), which comprise multiple Artificial Intelligence (AI) agents designed to solve a set of complex tasks collaboratively. It reviews popular existing open-source frameworks to implement MASs and existing multi-agent topologies, conversation patterns and key design principles. This includes interface design for collaborative inference, key performance indicators to quantify the performance of MASs, as well as various workflow options to perform closed-loop optimization between AI agents. While focusing primarily on the employment of MASs in the next generation mobile networks, the present document explicitly discusses recommended architectural changes in the design principles of core network together with potential standard impacts.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents may be useful in implementing an ETSI deliverable or add to the reader's understanding, but are not required for conformance to the present document.

[i.1] 3GPP TS 23.501 (V19.2.1): "System architecture for the 5G System (5GS)".

[i.2] 3GPP TS 29.500 (V19.2.0): "Technical Realization of Service Based Architecture".

[i.3] 3GPP TS 29.501 (V18.3.0): "Principles and Guidelines for Services Definition".

[i.4] IEEE 802.11™: "IEEE Standard for Information Technology--Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks--Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications".

[i.5] 3GPP TS 33.501 (V19.1.0): "Security architecture and procedures for 5G system".

[i.6] J. Kaplan, et al.: "Scaling Laws for Neural Language Models," arXiv preprint arXiv:2001.08361, 2020.

[i.7] Q. Wu, et al.: "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations", Conference on Language Modeling, 2024.

[i.8] T. Guo, et al.: "Large Language Model Based Multi-Agents: A survey of progress and challenges", arXiv preprint arXiv:2402.01680, 2024.

[i.9] C. Chan, et al.: "Chateval: Towards better LLM-based Evaluators Through Multi-Agent Debate", arXiv preprint arXiv:2308.07201, 2023.

[i.10] GitHub for the Agent Network Protocol.

[i.11] J. Rosenberg, C. Jennings: "Framework, Use Cases and Requirements for AI Agents", (accessed on 08 May 2025).

[i.12] ETSI GR ENI 051 (V4.1.1): "Experiential Networked Intelligence (ENI); Study on AI Agents based Next-generation Network Slicing".

[i.13] ETSI GR ENI 016 (07-2021): "Experiential Networked Intelligence (ENI); Functional Concepts for Modular System Operation".

[i.14] ETSI GS ENI 019 (V4.1.1): "Experiential Networked Intelligence (ENI); Representing, Inferring, and Proving Knowledge in ENI".

[i.15] Agent Communication Protocol.

[i.16] Anthropic: "Building Effective Agents", 2024.

[i.17] A. Ehtesham, A. Singh, G. K. Gupta, S. Kumar: "A Survey of Agent Interoperability Protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP)", arXiv preprint arXiv: 2505.02279, 2025.

[i.18] Y. Lu, J. Wang: "KARMA: Leveraging Multi-Agent LLMs for Automated Knowledge Graph Enrichment", arXiv preprint arxiv: 2502.06472, 2025.

[i.19] ETSI GS ENI 005 (V4.1.1): "Experiential Networked Intelligence (ENI); ENI System Architecture".

[i.20] Model Context Protocol (MCP).

[i.21] A2A Protocol.

[i.22] Semantic Versioning.

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the following terms apply:

5G System: 3GPP system consisting of 5G Access Network (AN), 5G Core Network and UE [i.1]

Customized Service Network (CSN): logical network that is generated on-demand by AI agents in the mobile core network according to the customers' intents and is comprised of network functions, application functions, 3rd party APIs, and the associated computing and communication resources

interface: point across which two or more components exchange information [i.19]

Multi-Agent System (MAS): collection of autonomous, interacting agents that work together (or compete) to solve problems that are too large, dynamic, or complex for any single agent to solve alone

Network Function (NF): 3GPP adopted or 3GPP defined processing function in a network, which has defined functional behaviour and 3GPP defined interfaces [i.1]

semantic similarity: metric defined over a set of documents or terms that measures the degree to which two linguistic items share similar meanings through "is-a" relationships (synonymy and hyponymy)

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

3GPP	Third Generation Partnership Project
5G	Fifth Generation
A2A	Agent2Agent

ABAC	Attribute-Based Access Control
ABI	Agent-Based Interface
ACP	Agent Communication Proxy
ADK	Agent Development Kit
ADP	Agent Description Protocol
AI	Artificial Intelligence
AMF	Access and Mobility Management Function
ANP	Agent Network Protocol
API	Application Programming Interface
ARF	Agent Repository Function
AUSF	Authentication Server Function
CN	Core Network
CRA	Conflict Resolution Agent
CRM	Customer Relationship Management
CSN	Customized Service Network
E2E	End-to-End
HTTP	Hypertext Transfer Protocol
ID	Identifier
JSON	JavaScript Object Notation
JWT	JSON Web Token
LLM	Large Language Model
LSP	Language Server Protocol
MAD	Multi-Agent Debate
MAS	Multi-Agent System
MCP	Model Context Protocol
NAS	Non-Access Stratum
NEF	Network Exposure Function
NF	Network Function
NRF	Network Repository Function
NSSF	Network Slice Selection Function
NWDAF	Network Data Analytics Function
OASF	Open Agent Schema Framework
P2P	Peer-to-Peer
PCF	Policy Control Function
PDU	Protocol Data Unit
PnP	Plug-and-Play
PLMN	Public Land Mobile Network
QoS	Quality of Service
RAN	Radio Access Network
RAG	Retrieval-Augmented Generation
RBAC	Role-Based Access Control
RFC	Request for Comments
RPC	Remote Procedure Call
RTP	Real-time Transport Protocol
SBA	Service-Based Architecture
SBI	Service-Based Interface
SCP	Service Communication Proxy
SDO	Standards Development Organization
SDK	Software Development Kit
SIP	Session Initiation Protocol
SLA	Service Level Agreement
SM	Session Management
SMF	Session Management Function
SSE	Server-Sent Events
TTL	Time to Live
UE	User Equipment
UDM	Unified Data Management
UPF	User Plane Function
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier

4 Background

4.1 Service Based Architecture

4.1.1 SBA Reference Model

The system functionality in a 5G mobile network is achieved by a set of Network Functions (NFs) providing services to other NFs. To name a few examples, Access and Mobility Management Function (AMF) and Network Slice Selection Function (NSSF) are NFs providing *Namf_Location* and *Nssf_NSSelection* services, respectively. While the former allows a service consumer to request location information for a specific UE, the latter allows the service consumer NF to request information about a network slice. As can be seen here, an NF service is a specific capability exposed by a (service producer) network function to other NFs that are authorized to consume that service. As a design principle, 3GPP requests each of the NF services to be self-contained, acted upon and managed independently from other services of the same NF.

4.1.2 Service Based Interface

Service-Based Interfaces (SBIs) are used in the 5G SBA to enable communication between NFs. Each SBI can consist of multiple services. A Service-Based Interface (SBI) represents how the set of services is provided or exposed by a given NF. This is the interface where the NF service operations are invoked [i.2]. As specified by 3GPP in [i.1], 5G system architecture contains more than twenty SBIs, including but not limited to *Namf*, *Nsmf*, and *Nssf* exhibited by the AMF, SMF and NSSF, respectively.

In the 5G SBA, each SBI provides access to multiple services via standardized Application Programming Interfaces (APIs). As defined in [i.3], each 5G CN SBI API specification is required to include the following information (among others) for each specified service, purpose of the API, URIs of resources, supported HTTP methods for a given resource (e.g. HTTP GET), supported representations (e.g. JSON), request body schema(s) (where applicable), response body schema(s) (where applicable), and supported response status codes.

By following the design principles mentioned above, 5G SBIs are built on explicit service operations, meaning that a service producer expects a structured and pre-defined input (i.e. service request) upon which it generates a pre-defined output (i.e. service response). The relationship between the input and output is specified and any deviation leads to undesired behaviour, such as the rejection of a service request, e.g. due to a syntax error, input size error, followed by the corresponding HTTP status code of 400 (i.e. bad request).

4.1.3 Network Function Service Framework

As introduced earlier, the SBA consists of multiple NF services offered by different NF instances, typically running on different machines in a distributed fashion. This mandates a set of mechanisms that allow the NF consumers to know about the existence of other NF services, as well as their location for interacting with them. To address this, the 3GPP has defined the NF service framework, which includes a list of mechanisms to enable the use of NF services in the service-based architecture. These are:

i) NF service registration and de-registration;

ii) NF service discovery;

iii) NF service authorization; and

iv) Inter-service communication [i.2].

For the registration and de-registration of NF services, the Network Repository Function (NRF) plays a central role. In addition to few other services, the NRF offers *Nnrf_NFManagement* service and *Nnrf_NFDiscovery* services that define how each service producer NF instance informs the NRF of the list of NF services that it supports and how each service consumer NF instance discovers other NF instances with the potential services they offer, respectively. The registration of an NF is done by sending a *Nnrf_NFManagement_NFRegister* request to the NRF that contains the NF profile. The NF profile contains information related to the NF instance to be registered, such as its NF instance ID and its supported NF service list. Typically, this is done when the service producer NF instance becomes operative for the first time.

Similar to the registration procedure, the standard allows an NF instance to de-register from the NRF before shutting itself down or disconnecting from the network in a controlled manner.

In the specific case of SBA security, the NF service authorization is ensured both by the NRF and the NF service producer. More specifically, the NRF is able to hide the NFs in one trust domain from entities in a different one; and ensures that the NF discovery and registration requests are authorized. At the service producer-consumer level, each NF validates the incoming messages, where invalid messages according to the protocol specification and network state are rejected/discarded [i.5]. This validation typically involves checking a JSON Web Token (JWT) based on the OAuth2.0 framework, which the consumer obtains from the NRF.

The SBA in 5G considers two options for the communication between NF service consumers and producers, namely, direct- and indirect communication. As the name suggests, while two NFs are able to send messages to directly each other in the direct communication mode, in case of indirect communication the communication takes place via an in-between entity called Service Communication Proxy (SCP).

NOTE 1: The SCP centralizes functions like load balancing, routing, and delegate discovery (where the SCP queries the NRF on behalf of the consumer), simplifying the logic required in each individual NF.

More specifically, when using direct communication, an NF service consumer sends a service discovery request to the NRF using the *Nnrf_NFDiscovery* service. After having received the necessary information about the producer of the requested service (and the NF instance producing it), the corresponding service is invoked by sending messages directly to the NF service producer. When the communication takes place indirectly, then the SCP is responsible for routing messages between the service consumer and producer. The SCP is sometimes implemented in a distributed manner (e.g. there can be multiple SCPs between the service consumer and producer NFs).

NOTE 2: An NF is configured with its serving SCP(s).

4.2 Open-Source Multi-Agent Frameworks and Communication Mechanisms

4.2.1 Introduction

AI Agents are software applications that utilize Large Language Models (LLMs) to interact with humans (or other AI Agents) for purposes of performing tasks [i.11]. While an LLM provides the core capacity for language understanding, reasoning, and generation, an AI Agent is more accurately defined as a software entity that encapsulates this cognitive core within a broader system, enabling it to perceive its environment, make autonomous decisions, and execute actions to achieve specified goals.

The transition from a simple LLM-powered application to a true AI Agent is marked by the introduction of several key architectural components and capabilities. Foremost among these are planning and reasoning faculties dedicated to the strategic use of available resources. These resources are not limited to internal knowledge but extend to a diverse array of external utilities, including software tools, Application Programming Interfaces (APIs) for accessing web services and databases, and extensive document corpuses for retrieval-augmented generation. The agent's defining characteristic is its ability to formulate a plan, select the appropriate tool for a given sub-task, execute that tool, observe the outcome, and reason about the next step in a cyclical process. This iterative loop of thought and action moves the system from a passive, input-output mechanism to a proactive, goal-oriented problem solver.

While single-agent systems represent a powerful paradigm, their capabilities are inherently bounded. As the complexity of tasks increases, a single agent sometimes encounters significant bottlenecks. These limitations include a finite context window, which restricts the amount of information the agent holds in its working memory, and the challenge of effectively selecting from an ever-expanding suite of tools, which leads to poor or inefficient decision-making. Multi-Agent Systems (MASs) have emerged as a direct architectural response to these challenges, designed to solve problems that are beyond the capacity of any individual agent.

The following clauses provide a brief overview of existing open-source frameworks to implement Multi-Agent Systems (MASs) and discuss different topologies and patterns for agent-to-agent communication within a MAS.

4.2.2 Overview of Existing Multi-Agent Frameworks

Multi-agent frameworks have emerged as powerful tools for developing complex AI systems, enabling multiple agents to collaborate and solve tasks beyond the capabilities of individual models. There are various open-source frameworks available to implement multi-agent systems. Some prominent examples are AutoGen, CrewAI, and LangGraph. Each framework offers unique strengths, catering to different needs in multi-agent system development.

AutoGen focuses on a multi-agent conversation framework where agents communicate with each other to accomplish tasks. Its core philosophy is that complex tasks are orchestrated and automated by framing them as structured dialogues between a collection of capable, customizable, and "conversable" agents. This approach simplifies the development of complex LLM workflows by abstracting them into automated chats. These agents are conversable, customizable, and are able to integrate LLMs, humans, and tools. It supports both static and dynamic conversations, allowing for flexible interaction patterns based on the workflow. AutoGen emphasizes human involvement, enabling human feedback and judgment in the decision-making process. This is particularly useful for tasks requiring human oversight.

CrewAI is another example of open-source multi-agent frameworks. CrewAI is an open-source Python framework designed to facilitate the creation of autonomous AI agent teams that collaborate on complex tasks. Its core philosophy is built around the intuitive and powerful metaphor of a "crew" of agents working together, much like a human project team or a company with specialized departments. This approach emphasizes role-based agent design, where each agent is defined by a specific role, a clear goal, and a detailed backstory. These narrative elements are not merely descriptive; they serve as a form of meta-prompting that guides the agent's behaviour, decision-making, and collaborative style. CrewAI organizes agents into roles with specific goals and tools. It uses event-driven workflows to manage agent interactions dynamically, while supporting conditional logic and state management. In contrast to AutoGen, which integrates human feedback more prominently, CrewAI focuses more on autonomous agent collaboration.

LangGraph is an open-source AI agent framework built on top of LangChain, designed to create, deploy, and manage complex generative AI workflows. LangGraph is an open-source library, built by the creators of LangChain, designed to orchestrate agentic and multi-agent workflows by modelling them as cyclical graphs. This graph-based paradigm is a direct and powerful response to the inherent limitations of the linear, sequential execution models (or "chains") that characterized early LLM applications. Many complex reasoning processes, such as chain-of-thought or ReAct (Reason and Act) patterns, are naturally cyclical. LangGraph provides the primitives to build these cycles explicitly, enabling more flexible, robust, and sophisticated agent behaviours. It utilizes a graph-based architecture, where workflows are structured as directed graphs consisting of nodes and edges. In LangGraph, workflows are modelled as graphs, enabling branching, looping, and conditional logic. This makes it ideal for handling complex interactions and dynamic decision-making. LangGraph also supports running multiple tasks simultaneously for a more efficient workflow execution.

Agent Development Kit (ADK) is an open-source, flexible, and modular framework designed to simplify the creation, orchestration, evaluation, and deployment of AI agents. It is model-agnostic and deployment-agnostic, allowing integration with various frameworks and environments. It is a code-first toolkit designed to bring the discipline and practices of traditional software development to the world of AI agents. It provides a flexible and modular framework for defining agent logic, tool integrations, and orchestration directly in code (primarily Python, with Java support as well), emphasizing crucial software engineering principles like testability, versioning, and modularity.

Each of these frameworks supports a large variety of existing LLMs, such as GPT-4, Llama3, and Mistral. Table 4.2.2-1 compares these four multi-agent frameworks.

Table 4.2.2-1: Comparison of AutoGen, CrewAI, LangGraph, and ADK

Feature	AutoGen	CrewAI	LangGraph	ADK
Philosophy	Multi-Agent Conversation	Autonomous Agent Teams	Stateful Graph-Based Workflows	Code-First Software Engineering
Primary Abstraction	Conversable agents engaging in automated "chats"	A "crew" of agents with defined roles, goals, and tasks	A cyclical graph of nodes (agents or functions) and edges (control flow)	Agents as modular, testable software components defined directly in code
Control vs. Autonomy	Balances both; supports dynamic, LLM-driven conversations but allows for human-in-the-loop control	Offers a dual model: *Crews* for high autonomy and *Flows* for granular, event-driven control	High developer control; the graph structure and conditional edges need to be explicitly defined by the developer.	High developer control; promotes deterministic logic and orchestration, treating agents as versioned code assets.
Communication Model	Internal Orchestration: Focuses on patterns like group chat and nested chat within a single system	Internal Orchestration: Manages collaboration within a crew via sequential or parallel processes	Internal Orchestration: Models agent interactions as state transitions within a self-contained graph	External Interoperability: Natively supports the A2A protocol for communication between disparate, remote agent systems
Key Differentiator	Seamless human-in-the-loop integration and dynamic conversational patterns	Intuitive role-playing metaphor and the duality of autonomous Crews vs. controlled Flows	Explicit, cyclical, and stateful graph architecture enabling durable and observable execution	The A2A protocol for building open, interoperable, and distributed agent services
Ideal Use Cases	Interactive problem-solving, tasks requiring human oversight, rapid prototyping of conversational agents	Business process automation, modelling real-world teams, applications requiring a balance of creativity and structure	Complex, long-running tasks, workflows requiring robust error handling and looping, applications needing deep observability	Building enterprise-grade, standalone agentic microservices intended to collaborate with external, multi-vendor systems

4.2.3 Multi-Agent Communication Mechanisms

Multi-Agent Communication Patterns

AutoGen mentions different multi-agent conversation patterns (depicted in Figure 4.2.3-1):

- *Two-agent chat*: considers two agents that exchange messages one-after-another, i.e. first Agent A then Agent B. An example of the two-agent chat pattern is a customer support scenario where a client and a specialist communicate.
- *Sequential chat*: involves a sequence of chats between two or more agents. This pattern extends the two-agent chat by chaining multiple conversations together. It is designed for tasks with interdependent steps. This approach is particularly useful for tasks requiring a sequence of inter-dependent multi-agent conversations (i.e. when they have to run one after the other). An example of a sequential chat is when Agent A tells Agent B "Retrieve pointcloud data of Central Park" followed by Agent A requesting Agent C to "Plot a 3D map based on the pointcloud data". Note that the second task depends on the result of the first one.
- *Nested chat*: According to this pattern, one agent holds the current conversation while invoking conversations with other agents depending on the content of the current message and context. This is useful for creating a hierarchical structure of agents, as the nested agents are not allowed to communicate directly with other agents outside the same group.

- *Group chat*: Involves more than two agents in a single conversation thread, sharing the same context. Each agent participating in the group chat is typically specialized for a particular task, such as a 'code generator', 'code reviewer', and a 'code executor'. In a group chat pattern, the agents do not talk directly with each other. Instead, everything is managed by the 'Group Chat Manager', which is a special agent that uses an LLM to orchestrate the conversation flow. The group chat manager selects an agent from the group as the "speaker", and then the speaker agent talks to the manager. The manager then broadcasts the message to all agents and chooses the next speaker. Therefore, group chat follows a publish-subscribe communication model, where each participating agent is a publisher and a subscriber of the same topic with the manager serving as the message broker.

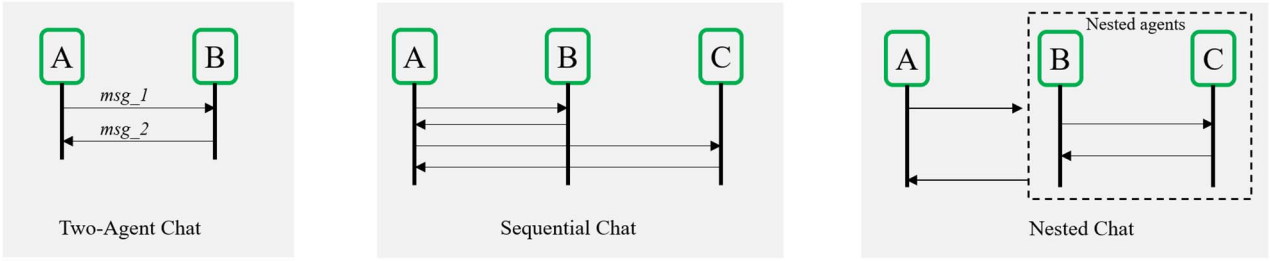


Figure 4.2.3-1: Visualization of two-agent chat, sequential chat and nested chat patterns

The communication pattern plays an important role in determining the performance of a MAS. For a given task, the MAS with different conversation patterns may output different results. Therefore, it is important to design the customized conversation patterns for various network services.

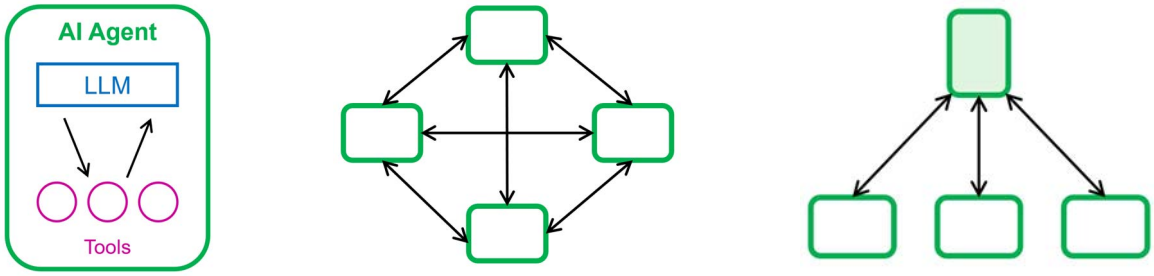
Multi-Agent Topologies

LangGraph mentions different multi-agent topologies that are used to design and implement multi-agent systems. In the following, each AI agent is represented as a graph node, where the agent-to-agent messages are passed from one graph node to another. The following content does not necessarily imply a distributed implementation where nodes (i.e. AI Agents) are physically decoupled from each other, meaning, they are running on different machines. On the contrary, it studies different multi-agent topologies from a conceptual point-of-view, without diving deep into technical implications, which are discussed later in clause 6, when the agent-to-agent communication implies actual data transmission between different network nodes.

Single Agent: For completeness, the simplest topology possible is considered when there is only a single agent. Figure 4.2.3-2a) shows that the AI agent has an LLM inside, which is the sub-component of an AI agent making it an intelligent system that solves various complex tasks. Moreover, an AI agent has multiple tools that are available to be called to assist task execution and output generation (e.g. web search tool). No agent-to-agent communication takes place in this topology.

Network: As depicted in Figure 4.2.3-2b), when each AI agent can directly talk to all other AI agents in the multi-agent system, then this is called the "network" topology. This is similar to "complete graph" from graph theory, where each pair of distinct vertices is connected by a unique edge.

Supervisor: The supervisor topology is illustrated in Figure 4.2.3-2c) which contains a supervisor agent deciding which agent needs to be called next. This means the supervisor agent is operating as a task scheduler and handoffs to other agents. The agents that are not in the supervisor role are allowed to only communicate with the supervisor agent, which corresponds to the "star topology" from graph theory. This pattern is highly effective for task delegation where a central intelligence is needed to orchestrate the workflow. It is analogous to AutoGen's *GroupChatManager* and is implemented in LangGraph by having a supervisor node that uses conditional edges to route to the appropriate worker node. The decision mechanism is implementation-specific, ranging from a turn-based policy like round robin or a more advanced policy that utilizes the LLM of the supervisor agent. A special case of this is using a tool-calling LLM as the supervisor, where each worker agent is exposed as a "tool" that the supervisor chooses to call when needed.

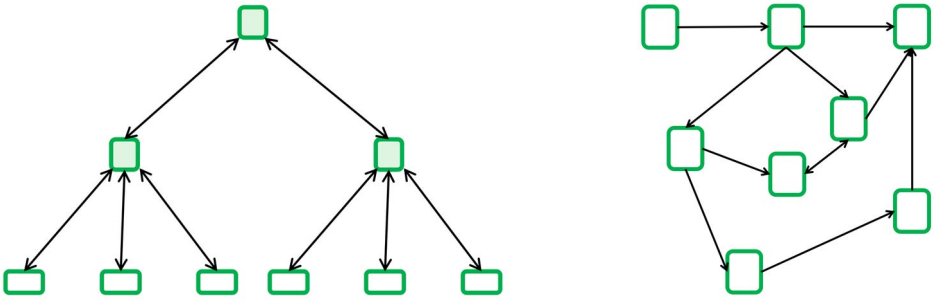


a) Single Agent b) Network c) Supervisor

Figure 4.2.3-2

Hierarchical: As the name suggests, this topology allows the design of a multi-agent system as a hierarchical (or tree) structure and is used to implement more complex workflow control. The "hierarchical" topology is considered as a supervisors of supervisors. Hence, it is considered as a generalization of the "supervisor" architecture. The main motivation behind this topology over the supervisor is the same behind multi-agent systems over single-agent systems, which is the increased scalability. More specifically, a single supervisor is likely to start making poor handoff decisions due to a growing context. This renders the decision-making of the supervisor very complex to keep track of and calls for a division of responsibilities among multiple supervisors, while each supervisor is managing a smaller team of specialized agents. Figure 4.2.2-3a) shows an example multi-agent system following the hierarchical topology.

Custom: The topology of the multi-agent system typically uses a custom architecture, where each agent communicates only with a subset of agents. This topology represents a hybrid approach that provides a balance between rigid structure and complete dynamism. The communication paths are explicitly defined, but they do not conform to a simple star or tree structure. The multi-agent collaboration allows a hybrid combination of explicit and dynamic workflow control. In other words, while some agents deterministically handoff to a specific agent, some agents are allowed to decide which agent to call next. Figure 4.2.2-3b) depicts a multi-agent system with a custom topology.



a) Hierarchical b) Custom

Figure 4.2.3-3

4.2.4 Existing Multi-Agent Protocols

4.2.4.1 Introduction

The need for multi-agent collaboration created a new scenario for communication between different software components over the Internet. This calls for the standardization of further protocol development to facilitate the deployment of multi-agent systems. This has been rightfully recognized by the industry leading to a rapid stream of open-source efforts towards agent-to-agent communication. Some prominent examples are A2A [i.21], Model Context Protocol (MCP) [i.20], Agent Connect Protocol (by AGNTCY™), Agent Network Protocol (ANP), and Agent Communication Protocol. [i.17] contains further information about agent interoperability and provides details on a set of selected multi-agent communication protocols. In [i.11], the multi-agent communication protocols are located on top of the application layer protocols such as HTTP, SIP, and RTP.

Modern agent protocols are broadly organized along a key architectural distinction:

1) context-oriented protocols, which standardize how a single agent interacts with external tools and data, and

2) inter-agent protocols, which enable multiple autonomous agents to collaborate as peers.

NOTE: AGNTCY™ is an open-source infrastructure stack for building and managing the "Internet of Agents." While it is not a communications protocol, it is still of interest in the present document. Hence, it is described in Annex B.

4.2.4.2 Context-Oriented Protocols

4.2.4.2.1 Introduction

LLMs have been designed to predict the next word given an input sequence; therefore, they are not capable of performing an action on behalf of the user, as by definition LLMs do not possess the tools to do so. Software developers overcame this shortcoming of LLMs by combining them with external tools so that they can do more than responding to a text input with a text output. For instance, when combined with appropriate tools, LLMs gain the ability to fetch information from the internet and present it to the human user. However, this approach comes with the significant challenge that as the number and variety of tools increase, it becomes very difficult for the developer to support tool invocation by LLMs, since each tool comes with its own custom API. This means, the LLM has to be able to send and receive messages to invoke the tool based on a format that is tool-specific. In addition to the implementation difficulty mentioned above, this also introduces an additional point of failure as the LLM may end up attempting erroneous utilization of the tool APIs and lead to hallucinations.

Context-oriented protocols address the fundamental challenge of how a single LLM-based agent acquires the external context (e.g. data from databases, information from APIs, or the ability to execute functions) that are necessary to complete its tasks. They provide a standardized bridge between the agent's reasoning capabilities and the outside world of tools and resources.

4.2.4.2.2 Model Context Protocol (MCP)

Tackling these challenges, MCP aims to simplify how LLMs access external tools, databases, and services by serving as an intermediate layer between an LLM and the tools. It is often described as a "USB-C port for AI" providing a single, standardized way to plug models into the vast ecosystem of external context they need to perform complex tasks. The protocol was created to address the significant challenge of fragmentation, where every AI model and external tool required a custom, brittle integration, hindering scalability and creating isolated "digital silos" [i.20].

The MCP has been designed around a client-server architecture inspired by the Language Server Protocol (LSP), which standardized how development tools integrate programming language support. The protocol uses JSON-RPC 2.0 for its messaging format. The MCP protocol implements a bidirectional connection between an MCP client and a MCP server. The MCP server exposes to the MCP client three types of capabilities, namely, *tools*, *prompts*, and *resources*. Resources are contextual data such as files or database records, that are used by the AI model or presented to the user. For instance, an MCP resource can be text, binary, and image file. Tools are executable functions that the AI model calls to perform actions, such as sending a message, running a database query, or interacting with an API. Prompts are templated messages or that guide the user or the AI model through a specific process.

An MCP client can query the resources of an MCP server by sending a '*resources/list*' request and after receiving the necessary information, such as the *name* of the resource, its *description*, and *URI*, it sends a 'resources/read' request to fetch the resource content by specifying its URI. Similarly, MCP servers expose tools in a similar and standardized manner to an MCP client. A tool has a *name* that serves as a unique identifier for the tool, a *description* for the client to understand its main purpose, and an *inputSchema* (JSON) defining the expected parameters for the tool. Similarly, a client can discover the available tools by sending a '*tools/list*' request to the MCP server and can call them using the '*tools/call*' endpoint. MCP supports dynamic tool life-cycle management, where new tools can be added or removed during runtime. Moreover, MCP allows servers to notify clients about changes in its tools or resources (e.g. '*notifications/resources/list_changed*').

The MCP protocol defines two transport mechanisms for communication between an MCP client and an MCP server: standard input and output (*stdio*) and *streamable HTTP*. The *stdio* transport can be used when the client and server are located on the same host. In such a setting, the client launches the MCP as a subprocess. The server reads from its standard input to receive MCP messages from the client and responds by writing to its standard output. The streamable HTTP transport is useful for the typical deployment scenario when the client and server are not co-located, i.e. when the MCP server is running remotely. It is also possible to use streamable HTTP transport when the client and server are running on the same host.

The streamable HTTP transport supports stateless connections, which is particularly useful for microservices architecture and the server offers simple tools to be called by the clients. When used, the MCP server provides a single HTTP endpoint path, e.g. "https://<server-ip-addr>/mcp", to which the clients send their HTTP POST and HTTP GET messages in JSON-RPC format.

An example client-to-server JSON-RPC message body to invoke a server tool that adds two integers 'a'=22 and 'b'=11 looks as follows:

```
{"jsonrpc": "2.0", "id":1, "method": "tools/call", "params":{"name": "add", "arguments":{"a": 22, "b": 11}}}
```

It is important to mention that MCP mandates the client to include certain information in the HTTP header, more specifically, in the *Accept* and *Content-Type* fields, which is not shown in the example above for presentation purposes. The reader can refer to the official MCP website for more information. It is important to mention that in contrast to Server-Sent Events (SSE), streamable HTTP utilizes a single HTTP endpoint for both requests and responses, i.e. bi-directional communication, thus, eliminating the necessity to manage separate endpoints for requests and responses.

The protocol's open nature and its direct solution to a widespread industry problem led to rapid and broad adoption. This industry backing, combined with a rich open-source ecosystem, has solidified MCP's position as the de facto standard for agent-to-tool communication. This widespread support has enabled a variety of powerful use cases, from internal enterprise assistants that access proprietary Customer Relationship Management (CRM) data to academic tools that perform semantic searches across research libraries.

4.2.4.3 Inter-Agent Protocols

4.2.4.3.1 Introduction

While MCP excels at connecting a single agent to its tools, inter-agent protocols address the next level of complexity: enabling multiple, independent agents to communicate, coordinate, and collaborate to solve problems that are beyond the scope of any single agent.

4.2.4.3.2 The Agent-to-Agent (A2A) Protocol

The Agent-to-Agent (A2A) protocol is an open standard designed to enable seamless communication and collaboration between AI agents, regardless of the platform, framework, or company that created them. The project is currently hosted by the Linux Foundation to ensure neutral, community-driven governance. The protocol's core mission is to break down the "digital silos" that isolate agentic systems, allowing them to collaborate as peers rather than just interacting as tools [i.10]. Put another way, A2A is a protocol developed that aims to standardize how multiple AI agents collaborate, regardless of the underlying multi-agent framework or vendor.

A2A is strategically positioned to complement, not replace, other key standards like the Model Context Protocol (MCP). While MCP excels at connecting a single agent to its external tools and data sources, A2A focuses on the next layer of interaction: enabling multiple, independent agents to delegate tasks and work together on complex problems. The recommended architectural pattern is to use MCP for an agent's internal connections to resources and A2A for its external communications with other agents [i.10].

The A2A protocol builds on established and widely adopted web standards to ensure easy integration into existing IT stacks. It uses JSON-RPC 2.0 over HTTP(S) for standardized communication, with support for real-time updates and streaming via Server-Sent Events (SSE).

A defining principle of A2A is opacity. Agents are allowed to collaborate without needing to expose their internal memory, proprietary logic, or the specific tools they use. This enhances security, protects intellectual property, and simplifies interactions, as agents do not need to understand each other's internal workings to collaborate effectively. The protocol is also modality-agnostic, designed to support rich data exchange including text, audio, video, and structured JSON data.A2A considers three actors of a multi-agent system:

i) a *user*, which is either a human or a service that is utilizing a multi-agent system to accomplish certain tasks;

ii) a *client*, which is a service, an agent, or an application that is requesting an action from a remote agent on behalf of the user;

iii) a *remote agent* that is acting as the A2A server.

A2A protocol supports both pull-based and push-based communication. This means, A2A not only supports the standard request-response patterns or polling for updates, but also streaming updates through SSE and push notifications are also supported. An example of pull-based communication is when a client polls a remote agent (server) to retrieve the latest status of a long-running task or to initiate a new task for the remote agent to perform. Alternatively, in the example of long-running tasks, a remote agent can push notifications to the client, e.g. HTTP-based SSE.

For authentication and authorization, A2A does not transmit identity information as a part of the A2A payload (i.e. in-band), but it obtains materials (such as tokens) out-of-band and transmits materials in HTTP headers. The only information for authentication that is transmitted in the A2A payload is the authentication requirements, which can be sent by an A2A server. Every client request is authenticated by a remote agent to prevent unauthorized access and is responded to by standard HTTP response codes.

A2A introduces the concept of an *AgentCard* for agent discovery purposes. An *AgentCard* is published by remote agents and contains information about the AI agents such as their capabilities, skills, and authentication requirements. An example format for the *AgentCard* is JSON. The key fields of *AgentCard* are listed in Table 4.2.4.3.2-1.

Table 4.2.4.3.2-1: Key fields of an *AgentCard*

Field Name	Type	Required	Description
name	string	Yes	Human-readable name of the agent.
description	string	Yes	A human-readable description of the agent. Used to assist users and other agents in understanding what the agent is able to do.
url	string	Yes	A URL where the agent is hosted at.
provider	*AgentProvider*	No	Information about the agent's provider.
iconUrl	string	No	A URL to an icon for the agent.
version	string	Yes	The version string for the agent or its A2A implementation.
documentationUrl	string	No	URL to human-readable documentation for the agent.
capabilities	*AgentCapabilities*	Yes	Specifies optional A2A protocol features supported (e.g. streaming).
securitySchemes	{ [scheme: string]: *SecurityScheme* }	No	Security scheme details used for authenticating with this agent.
security	{ [scheme: string]: string[]; }[]	No	Security requirements for contacting the agent.
defaultInputModes	string[]	Yes	Input Media Types accepted by the agent.
defaultOutputModes	string[]	Yes	Output Media Types produced by the agent.
skills	*AgentSkill*[]	Yes	Array of skills. Mandatory to have at least one if the agent performs actions.
supportsAuthenticated ExtendedCard	boolean	No	Indicates support for retrieving a more detailed Agent Card via an authenticated endpoint.

Among these, the *provider* field of type *AgentProvider* includes the name of organization and the URL for the provider's website. The skills field, which is an array of *AgentSkill*s, describes a specific capability, function, or area of expertise the agent performs or addresses. An *AgentSkill* object further includes the following fields listed in Table 4.2.4.3.2-2.

Table 4.2.4.3.2-2: Key fields of *AgentSkill*

Field Name	Type	Required	Description
name	string	Yes	Human-readable name of the agent.
id	string	Yes	Unique skill identifier within this agent.
name	string	Yes	Human-readable name of the skill.
description	string	Yes	Detailed skill description.
tags	string[]	Yes	Keywords and/or categories for discoverability.
examples	string[]	No	Example prompts or use cases demonstrating skill usage.
inputModes	string[]	No	Media types accepted for this specific skill, overriding defaults.
outputModes	string[]	No	Media types produced by this specific skill, overriding defaults.

More details on A2A protocol specification and key protocol data objects are summarized in Annex A.

4.2.4.3.3 Agent Communication Protocol

The Agent Communication Protocol (ACP) is an open standard for inter-agent communication and governed by the Linux Foundation to ensure vendor-neutral, community-driven development [i.15]. Its primary goal is to solve the challenge of a fragmented AI ecosystem, where agents are often built in isolated "silos" using incompatible frameworks. ACP provides a universal language that allows these disparate agents to discover, understand, and collaborate with one another, aiming to become a foundational "HTTP for AI agents.

ACP's design is intentionally simple, flexible, and aligned with modern web development practices to lower the barrier to adoption. Its architecture is built on three core principles:

1) REST-based Communication: Unlike protocols that use JSON-RPC, ACP is explicitly RESTful. It uses standard HTTP conventions, verbs, and status codes for communication. This design choice makes it easy to integrate into existing production environments and allows developers to interact with agents using common tools like curl or Postman without needing specialized libraries or SDKs.

2) Async-First Design: The protocol is optimized for asynchronous communication, which is ideal for the long-running, complex tasks common in agentic workflows. While async is the default, synchronous requests are also fully supported for simpler, immediate interactions.

3) Developer Ergonomics: ACP prioritizes a straightforward developer experience. While official SDKs for Python and TypeScript are available for convenience, they are not required for interaction, reflecting the protocol's lightweight and accessible nature.

ACP uses standard MIME types to identify content, making the protocol inherently extensible. This allows agents to exchange any form of data - including text, images, audio, video, or custom binary formats - without requiring any changes to the protocol itself. The protocol is designed to support both stateless interactions and stateful, long-running sessions. This allows agents to maintain context and history across multiple interactions when needed, which is critical for complex collaborative tasks.

A key differentiator for ACP is its support for both online and offline agent discovery. Agents are able to be discovered online via a standard REST endpoint. Uniquely, they also support offline discovery, where an agent's metadata and capabilities are embedded directly into its distribution package. This allows other agents to discover it even when it is inactive or running in a scale-to-zero cloud environment.

Table 4.2.4.3.3-1 describes its core protocol objects:

Table 4.2.4.3.3-1: Key Objects of ACP

Object Name	Description
Agent	A model describing an agent's capabilities, including its name and description. This is used for discovery without exposing internal implementation detail.
Run	Represents a single execution of an agent with specific inputs. A run is either synchronous or asynchronous and is able to stream intermediate and final outputs.
Message	The fundamental structure for communication. A message consists of an ordered sequence of MessagePart objects to form a complete, multi-modal exchange.
MessagePart	An individual unit of content within a Message, such as text, an image, or structured JSON data.
Await	A mechanism that allows an agent to pause its execution to request additional information or input from the client before resuming its task.
Session	Enables agents to maintain state and conversation history across multiple interactions using a unique session identifier.

4.2.4.3.4 Agent Network Protocol (ANP)

The Agent Network Protocol (ANP) is an open-source communication protocol with the ambitious vision of becoming the "HTTP of the agent internet era." Developed by an open-source team from China, ANP aims to create a truly open, secure, and efficient collaboration network for billions of intelligent agents. It seeks to fundamentally reshape the digital landscape from a "platform-centric" model, where data is locked in silos, to a "protocol-centric" one where every agent discovers, connects, and interacts with any other node without barriers.

ANP's design is a radical departure from protocols that build upon existing client-server web infrastructure [i.10]. It argues that for a true "Internet of Agents" to exist, the foundational layers of identity and negotiation needs to be re-examined. To achieve this, ANP is built on a distinctive three-layer architecture that systematically addresses these core challenges:

1) Identity and Secure Communication Layer. This is the foundational layer of ANP and its most critical differentiator. Instead of relying on centralized or federated identity systems like OAuth, this layer uses the W3C® Decentralized Identifiers (DIDs) specification. This allows any two agents, regardless of their platform or creator, to establish a secure, end-to-end encrypted communication channel and verify each other's identity without needing a central authority. This decentralized, peer-to-peer trust model is the cornerstone of ANP's vision for an open and secure agent web.

2) Meta-Protocol Layer. This innovative layer addresses the challenge of protocol negotiation. It defines a standard for how agents dynamically and automatically negotiate which application-layer protocols they will use for a specific interaction. This allows the agent network to be self-organizing and adaptable; new, more efficient communication methods need to be adopted by agents over time without requiring a rigid, top-down update to the entire protocol standard.

3) Application Protocol Layer. This top layer provides the mechanisms for agents to describe and discover capabilities. It is built on semantic web specifications and includes two key components:

   a) Agent Description Protocol (ADP): A standardized way for agents to document their own capabilities and interfaces in a structured, machine-readable format.

   b) Agent Discovery Protocol: Specifies how agents find each other and establish connections, effectively acting as a "search engine protocol" for the agent internet.

ANP's unique architecture provides four key features that distinguish it from other agent communication protocols:

1) Decentralized Peer-to-Peer (P2P) Model: ANP is fundamentally agent-centric and P2P, allowing any agent to directly connect with any other agent as an equal.

2) Identity-First Security: By building on a foundation of decentralized identity, ANP prioritizes verifiable trust and security from the ground up.

3) AI-Native and Efficient: The protocol is designed for AI-native interaction, allowing agents to communicate via APIs and protocols rather than inefficiently simulating human interaction with web interfaces.

4) Extensibility and Interoperability: The layered structure is modular, allowing each layer to be upgraded independently. Through its meta-protocol, it theoretically bridges to existing protocols, providing the services to implement the necessary identity standards.

4.2.4.4 Comparison of Inter-Agent Protocols

Table 4.2.4.4-1 compares A2A, ACP, and ANP.

Table 4.2.4.4-1: Comparison of A2A, ACP, and ANP

Feature	A2A	ACP	ANP
Primary Goal	Standardize inter-agent collaboration and task delegation, especially for enterprise use.	Provide a simple, universal, and developer-friendly protocol for inter-agent communication.	Build a foundational, decentralized, and secure P2P network for a future "Internet of Agents."
Architectural Style	JSON-RPC 2.0 over HTTP(S).	RESTful HTTP.	3-layer decentralized P2P.
Discovery Mechanism	AgentCard JSON file at a well-known endpoint.	Online (REST endpoint) and Offline (embedded metadata).	Agent Discovery Protocol based on semantic descriptions.
Security Model	OpenAPI authentication schemes (out-of-band tokens).	Standard HTTP authentication methods.	W3C® Decentralized Identifiers (DIDs).
Key Differentiator	Task-oriented lifecycle; agent opacity; enterprise-grade security and features.	Lightweight REST-based simplicity; no SDK required; unique offline discovery feature.	Decentralized identity-first model; meta-protocol for dynamic negotiation; ambitious P2P vision.
Session	Linux Foundation	Linux Foundation	Chinese Open Source team

The ENI initiative aims to define a cognitive network management architecture that uses AI to automate and optimize network operations. This system is based on a hierarchical, closed-loop "Observe-Orient-Decide-Act" (OODA) model composed of modular functional blocks. Given these requirements, the Agent-to-Agent (A2A) protocol is the best fit for the ETSI ENI architecture. Brief rationale is:

1) ENI is fundamentally an enterprise-grade system for telecom operators. A2A is explicitly designed for such environments, with a focus on enterprise-grade security and long-running, stateful tasks. Its use of established standards like JSON-RPC and OpenAPI security schemes provides a stable and trusted foundation.

2) The ENI System is responsible for executing complex, state-of-the-art network management operations. A2A's Task object, with its well-defined lifecycle, is perfectly suited for managing and tracking these operations (e.g. reconfiguring a network slice, predicting a fault, or optimizing resources). This provides a more structured and robust model for the ENI control loop than ACP's more general-purpose approach.

3) The telecom landscape is inherently multi-vendor. A2A's principle of opacity is a critical advantage here. It allows functional blocks from different vendors to collaborate and delegate tasks without having to expose their proprietary internal logic or tools. This directly supports the modular, functional block design of the ENI architecture and aligns with key software design principles like loose coupling.

4) The AgentCard provides a clear and effective mechanism for the various functional blocks within the ENI system to discover each other and understand their specific capabilities, which is essential for a complex, modular architecture.

4.3 The Role of MCP and A2A in an ENI System

MCP and A2A are designed to be complementary, not competitive.

MCP serves as the agent-to-tool communication layer. Its primary function is to provide a standardized way for a single functional block (acting as an agent) within the ENI architecture, defined in [i.19], to connect to and use its necessary external resources, data, and APIs. For example:

- A Data Ingestion block would use MCP to connect to various network data sources, such as telemetry streams, fault logs, and performance metric databases.
- A Cognition Framework block would use MCP to access the pre-processed data stores and to call specific machine learning models or analytics functions as tools.
- An Output Generation block would use MCP to connect to the APIs of network management and orchestration systems to execute a configuration change.

In essence, MCP handles the "vertical" integration, connecting an agent to the specific tools it needs to perform its individual function.

A2A serves as the agent-to-agent communication layer. Its role is to enable the different, independent functional blocks within the ENI architecture to collaborate, delegate tasks, and communicate with each other as peers. A2A provides the "horizontal" integration that connects the modular components of the ENI system into a cohesive whole. For example:

- When the Cognition Framework block predicts an impending network fault, it would use A2A to send a Task to the Core Business Logic block to decide on a course of action.
- The Core Business Logic block would then use A2A to delegate sub-tasks to other specialized blocks, such as instructing a Network Slice Management block to reallocate resources.
- This communication is managed through a structured, stateful process, which is ideal for the complex, multi-step workflows of the ENI control loop.

By using both protocols, the ETSI ENI architecture benefits from a clear and robust separation of concerns:

1) MCP standardizes how each individual agent accesses its tools.

2) A2A standardizes how all the agents collaborate with each other.

5 Multi-Agents System in Core Network

5.1 Benefits of Multi-Agent Systems

5.1.1 Energy Consumption and Inference Time

A single agent responsible for all ENI tasks - from real-time fault detection to long-term capacity planning and service orchestration - would need to be a massive, highly generalized model. According to the scaling laws of AI [i.6], such a model would require an enormous number of parameters to perform adequately across this diverse range of tasks. This results in consistently high inference latency, massive computational resource requirements, and significant energy consumption for every decision, regardless of its complexity. For instance, in the context of an AI agent-based mobile core network, these tasks include service orchestration, resource scheduling, and tool invocation.

In contrast, a MAS employs multiple agents with distinct roles that collaborate to achieve common goals. Often, this takes the form of a federation of smaller, highly specialized agents. While there is communication overhead between agents, this is often outweighed by the immense efficiency gains of using optimized models for specific tasks. The ENI system uses a small, rapid agent for real-time anomaly detection at the network edge and a larger, more comprehensive model for non-real-time trend analysis in a central cloud. This targeted allocation of computational resources is far more efficient than running a single, oversized model for every task. This also reduces inference latency, resource usage, and energy consumption.

5.1.2 Modularity and Extensibility

Single-agent systems, which employ a single generalized AI agent for all tasks, have certain limitations. For instance, a single-agent system sometimes performs poorly when it comes to deciding which tool to use to perform the task at hand if there are too many tools at the agent's disposal. Another prominent example is the rapid growth of context that the agent needs to keep track of. In addition, since the AI agent is utilized for all tasks that it is expected to solve, it lacks the specialization for a variety of tasks from different domains leading to deterioration of inference accuracy and task completion rate. A user interacting with one agent might require the agent to perform a task that is beyond its capabilities, requiring the agent to interact with a different agent that has the appropriate capabilities.

A MAS architecture embodies modern software design principles like loose coupling and high cohesion [i.13]. Each agent is a self-contained functional block, analogous to a microservice. This makes the entire ENI system easier to develop, test, and maintain. More importantly, it allows for true plug-and-play extensibility. If a new network technology is introduced, a new specialized agent is able to be developed and integrated without destabilizing the entire system. This is crucial for the long-term evolution of a network management platform.

MASs allow conversations between different specialized agents(i.e. a multi-agent chat, where each agent is focused on one or a set of specific sub-task(s)). By designing agents for specific tasks, multi-agent systems leverage the unique strengths of each agent, improving overall decision quality and system performance. This approach has been shown to improve the overall performance of the system, explained through the fact that LLMs have demonstrated the ability to solve complex tasks, when these are broken into simpler sub-tasks [i.7]. Furthermore, such a highly modular architecture renders the life-cycle management of AI agents easier, as it simplifies the process of development, testing and maintenance of such systems. Moreover, it is straightforward to introduce new agents into the system in a Plug-and-Play (PnP) manner, when the system is expected to tackle novel complex problems [i.8]. Similarly, those agents whose service is not needed anymore is typically removed from the system during runtime. These render MASs a significantly flexible and extensible system compared to single-agent systems.

5.1.3 Robustness

Single-agent systems face significant decision-making risks due to potential knowledge gaps and cognitive biases, which sometimes leads to incorrect decisions, especially in situations, when the AI agent is requested to solve an unknown and novel task. Furthermore, these systems have poor fault tolerance; if the agent is compromised or fails, service continuity is severely impacted.

In contrast, MASs effectively address these challenges. On one hand, decision errors made by an agent are sometimes mitigated or corrected through collaboration mechanisms among multiple agents [i.9]. These mechanisms include multi-agent debate, closed-loop optimization, and reflection, which are effective mechanisms to allow agents to cross-validate and refine their decisions. These render the system more adaptive and improve the overall performance at task completion by reducing the likelihood of persistent errors.

On the other hand, MASs offer enhanced fault tolerance. In case of failures of an agent, the system is able to continue to operate by dynamically reassigning roles so that the remaining agents take over the tasks that the failed agent is expected to do. This redundancy and adaptability make MASs more robust in decision-making and more resilient to failures compared to single-agent systems.

The robustness of a MAS goes beyond simple redundancy. It enables cognitive resilience. In a single-agent system, a failure or a cognitive bias in the model leads to a total system failure. In a MAS, the failure of one agent is not catastrophic. For example, if a primary "Fault Prediction Agent" fails, the system dynamically reroutes its data to a secondary instance or even a less-specialized "Anomaly Detection Agent" to provide degraded but continuous service. Furthermore, a MAS is able to implement cognitive cross-validation. An agent's decision is optionally debated or verified by other agents, mitigating the risk of a single point of cognitive failure and reducing the impact of hallucinations or biases in any one model.

5.1.4 Distributed Decision-making

Single-agent systems are inherently centralized, with all decisions made by a single entity. This centralized approach makes it challenging to handle hierarchical decisions across different tasks, as it often requires a broad understanding of the entire system and its complexities.

In contrast, multi-agent systems offer a flexible and distributed decision-making framework. They deploy agents with diverse roles in different locations to support both local, real-time decision-making and global, non-real-time decision-making. For instance, agents responsible for scheduling network resources are typically strategically placed near the functions running relevant services. This proximity allows them to make prompt decisions in response to changes in function status or network conditions, ensuring adaptability and efficiency. As the system grows or evolves, multi-agent architectures are able to easily scale by adding new agents or reconfiguring the existing ones, thereby enhancing distributed decision-making capabilities.

However, the true power of MASs is *enabling distributed cognition.* For instance, the ENI System, specified in [i.19], is composed of various functional blocks that need to work in concert. A MAS provides the natural framework for this. For example, a "Performance Monitoring Agent" that detects a Service-Level Agreement (SLA) violation does not act in isolation. It initiates a collaborative workflow:

1) It communicates its findings to a "Fault Correlation Agent" to identify the root cause.
2) Together, they consult a "Network Knowledge Graph Agent" to understand dependencies.
3) Once a hypothesis is formed, they delegate the remediation task to a "Network Slice Management Agent" to execute the necessary changes.

This collaborative process, where specialized agents share evidence and delegate tasks, is the essence of a cognitive system in action.

MASs also prevent cognitive overload. For example, a single agent would be overwhelmed by the data torrent during a major network event (e.g. a fibre cut or a traffic surge for a live sporting event). A MAS architecture allows a "Manager Agent" to dynamically spawn multiple instances of a "Log Analysis Agent" to process the data in parallel, ensuring that insights are generated in a timely manner.

A final, but very important, point is that MASs reflect the cognitive architecture of an ENI System. A MAS is the only architectural paradigm that truly enables the system to be cognitive, rather than just automated. For example, an ENI System is based on an enhanced OODA (Observe-Orient-Decide-Act) closed control loop. A single-agent system is a cognitive black box. It observes and acts, but the "orient" and "decide" phases are opaque processes within a single model. In contrast, a MAS externalizes this cognitive process. Different agents represent different stages of cognition: some agents observe (data ingestion), others orient (data processing, correlation, prediction), and others decide and act (policy management, output generation). The communication and collaboration between these agents is the cognitive workflow. This makes the system more transparent, auditable, and aligned with the foundational principles of ENI.

5.2 Design Principles

The adoption of a Multi-Agent System (MAS) architecture is fundamental to realizing the vision of ENI. While a single, monolithic AI system presents a seemingly simpler approach, a MAS is uniquely suited to deliver the cognitive, resilient, and scalable capabilities that network automation demands. The design of a MAS for a mobile core network that is aligned with the ENI System architecture [i.19] is guided by a set of core principles to ensure that it is autonomous, resilient, scalable, transparent, and trustworthy. These principles include:

- **Hierarchical Autonomy and Federated Control Loops**. A mobile core network requires both high-level stability and rapid, localized agility. A flat agent architecture very likely results in chaotic and difficult behaviour to govern, while a single, monolithic control system tends to be unresponsive and brittle. The ENI architecture's model of nested inner and outer control loops provides the solution. This directly translates into a principle of hierarchical autonomy, where high-level "Orchestrator Agents" manage long-term goals and delegate tasks to teams of specialized agents that operate in faster, more localized control loops.

- **Cognition and Continuous Learning**. The system is required to be truly cognitive, not merely automated. This requires agents to possess self-reflection and self-optimization capabilities, enabling them to monitor their performance, learn from interactions, and continuously improve their behaviour over time. This principle mandates the inclusion of explicit feedback loops and integration with MLOps pipelines to ensure the system adapts to the complex, evolving network environment.

- **Comprehensive Observability and Explainability**. For operators to trust an autonomous system, they need to be able to understand its behaviour and the reasoning behind its decisions. This requires:

  - Observability: Implementing end-to-end tracing to monitor the flow of tasks across multiple agents.

  - Explainability [i.19]: Cognition Agents are required to support decisions that have associated explanations for how the decision was arrived at (e.g. identifying the key factors in a prediction). This is critical for debugging and building operator confidence, as well as regulatory compliance.

- **High Cohesion and Loose Coupling [**i.13**]**. In a system responsible for a nation's communication infrastructure, failures are required to be isolated, and components need to be independently scalable and upgradeable. High cohesion dictates that each agent has a single, well-defined responsibility that is logically self-contained (e.g. a "Fault Prediction Agent"). Loose Coupling ensures that agents interact only through standardized, asynchronous interfaces, preventing fragile dependencies and allowing components to be replaced or updated without causing a system-wide ripple effect.

- **Structured and Stateful Collaboration**. The core function of the MAS is to execute complex network processes. This necessitates more than simple communication; it demands structured, goal-oriented collaboration, including:

  - Workflow-Driven Collaboration: The architecture is required to support explicit and dynamic workflows, allowing tasks to be allocated, negotiated, and coordinated in a way that adapts to changing goals and contexts.

  - Stateful, Long-Running Task Management: Core network operations (e.g. provisioning a service, healing a fault) are inherently long-running and stateful. The communication backbone is required to natively support the management of these complex tasks through their entire lifecycle, a key feature of protocols like A2A.

  - Standardized Interfaces: To enable this collaboration, the system is required to implement flexible and reliable interfaces for communication between agents, using standardized protocols to facilitate efficient information exchange. In particular, a MAS is required to be accessible only using ENI External Interfaces [i.19].

- **Agent Opacity and Explicit Discovery**. An agent from one vendor needs to be able to collaborate with an agent from another without revealing its proprietary logic, internal memory structure, or specific tool implementations. This concept of opacity is crucial for both intellectual property protection and system modularity. It corresponds to information hiding, which is a foundational aspect of encapsulation [i.13]. To enable collaboration between these "black box" agents, their capabilities need to be explicitly advertised through a standardized discovery mechanism, such as the A2A's AgentCard.

- **Protocol Agnosticism using a Semantic Abstraction Layer**. While telecommunications standards evolve over years and decades, AI and agent standards evolve over months. Tightly coupling the core logic of a network management system to a specific, volatile protocol is a significant architectural risk. Adhering to the Dependency Inversion Principle [i.13], the MAS is required to isolate its core logic from the underlying communication technologies. This is achieved by building agents against a stable, internal semantic model and using Protocol Adapters to translate to and from on-the-wire formats.

- **Data-Centric Architecture**. The intelligence of the MAS is dependent on the quality and accessibility of its data. The data pipeline needs to be treated as a first-class architectural component. This involves implementing a canonical data model based on ENI standards [i.14], robust data governance, and efficient data processing mechanisms to ensure that agents are operating on high-quality, reliable information.

5.3 Example AI Agents for the Core Network

In addition to the four AI agents presented in [i.12] (i.e. namely, planning agent, assemble agent, connection agent, and execution agent), the following contains some other example AI agents to be utilized in a next-generation core network. These include:

- *Charging Agent*: This AI agent is responsible for calculating the costs associated with various services provided by the AI core. It ensures accurate and transparent billing based on service usage.

- *Trustworthiness Agent*: This agent monitors the actions and outputs of other core network agents to verify their compliance with relevant regulations, such as the EU AI Act. Its primary goal is to prevent potential violations, which in turn could threaten the integrity or operation of the overall system or compromise subscriber privacy or security.

- *Critic Agent*: The Critic Agent evaluates the joint policies and contributions of each agent involved in a collaborative task. It provides rewards to individual agents based on their performance throughout task execution, thereby facilitating the continuous improvement and evolution of the multi-agent system.

6 Inter-Agent Communication

6.1 Agent-Based Interface

6.1.1 Motivation

As described in clause 4.1.2, service-based interfaces are imperative interfaces that expose a set of standardized services provided by network functions. They require structured and pre-defined input and generate pre-defined output. However, AI Agents do not mandate such a rigid interface. The interaction between multiple AI agents is much more flexible and is typically in the form of intents. These sometimes are provided in different modalities, such as text, image, and audio.

NOTE: Text will provide the most robust intent. This is because it is validated using a traditional (i.e. non-AI) parser if the input is structured as described in clause 6.9 of [i.19]. However, there is no equivalent mechanism for other modalities, such as image, audio, and video.

Thanks to the strong inference and reasoning capability of AI Agents, AI agents are able to understand the task, divide it into sub-tasks if necessary, and through the available tools, they are able to perform the necessary actions to complete the task autonomously. As an example, they are able to retrieve the required knowledge through a web search tool, or interact with other agents to let the other agents solve a sub-task and use the returned result to complete the remaining sub-tasks. As a result, these aspects call for new interfaces with features different than in the (imperative) service-based architecture to enable inter-agent communication in an agent-based mobile core network.

However, it is important to mention that the output generated by AI agents that are based on LLMs is non-deterministic. In particular, when two inputs have the same semantics but vary in the exact expression or they are identical but the context is different, the output of an AI agent is typically different due to the inherent statistical prediction mechanism of the LLM.

6.1.2 Definition and Characteristics

As it has been showcased in [i.12], a multi-agent-based core network architecture accommodates multiple AI agents that flexibly customize network services. To enable a multi-agent core network architecture that is able to flexibly customize network services, as shown in [i.12], agents require a well-defined way to communicate. Therefore, an 'Agent-Based Interface' (ABI) is defined as the shared boundary across which agents exchange information. An ABI specifies the allowed interactions by exposing methods, each with a corresponding input and output schema, without revealing internal implementations.

NOTE 1: This is standard practice in modern API frameworks. The input schema serves as a request contract, and the output schema serves as a response contract.

As depicted in Figure 6.1.2-1, there are mainly five types of interfaces for an agent-based core network architecture. These ABIs are characterized by their purpose and endpoints:

- **Agent-to-User**: the interface between the user and a network agent (e.g. it is used for users to initiate a service request or update an existing service).

NOTE 2: As this interface indicates an interaction with domain external to the mobile network, security requirements are expected to be different and more stringent than for a purely internal one. The present document does not discuss security aspects in detail, which are to be studied in a further document.

- **Agent-to-ARF** the interface between a network agent and an agent repository ("Agent Repository Function" (ARF) in [i.12]), which is responsible for maintaining the information of agents, this interface can be used for network agents' registration, update, de-registration and discovery.
- **Agent-to-Agent**: the interface among network agents, which is used for interacting task information when they work collaboratively.
- **Agent-to-Resource**: the interface between a network agent and resources, such as tools and data sources. An interface of this type is used for a network agent to invoke tools or retrieve data, e.g. network data, knowledge base.
- **Agent-to-Infrastructure**: the interface between a network agent and the underlying infrastructure, including RAN, computing platform, etc. It can be used for a network agent to configure the infrastructure.

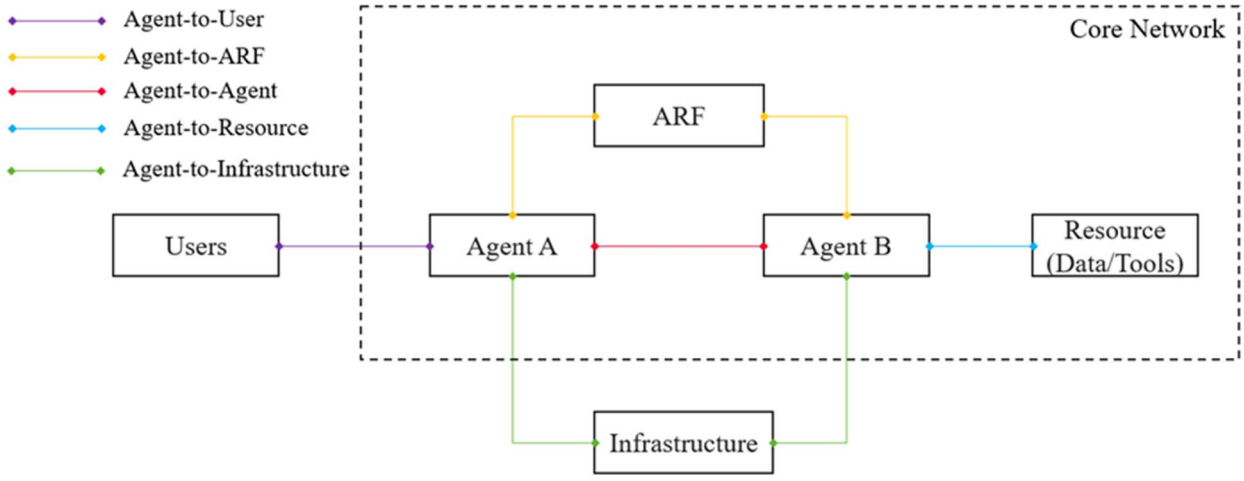


Figure 6.1.2-1: Five interface types for multi-agent-based mobile core network

NOTE 3: The interface types listed above do not imply any implementation choices. For example, the agent-to-agent interface is not necessarily a point-to-point interface. For example, it is possible to use the semantic bus from [i.19], which supports indirect communication with routing capabilities.

6.1.3 Interface Design Principles

The future core network faces more and diverse usage scenarios. This requires interfaces of the multi-agent- based core network to be flexible enough to adapt to each new scenario without having to be re-design. It also enables consumers to learn how to use the interface without needing to be programmed explicitly. In addition, operations in the core network usually have different security or delay requirements. For example, the access to the network operator data has higher security requirements than that to general knowledge acquisition. Hence, they are usually required to be designed independently to avoid coupling. Network agents are served as opaque entities without needing to expose their internal models, tools, memory, and other critical resources.

NOTE: In the following, a consumer is any component that uses or calls that interface to get something done. A provider is any component that offers a service or capability through an interface.

Thus, the design of interfaces for a multi-agent-based core network follows the below principles:

- **Adaptability**: Interfaces need to be designed to accommodate new and unforeseen requirements without necessitating a redesign. This in turn requires:
  - Declarative Interaction: Interfaces are required to be declarative, allowing a consumer to specify the desired outcome (what) rather than the explicit sequence of steps (how). This abstracts away implementation complexity and permits the provider agent to evolve its internal logic to meet the goal, enhancing flexibility.
  - Semi-structured Schemas: Interfaces should use semi-structured data formats (e.g. JSON, XML) for message payloads. This allows for the evolution of the data model, such as the addition of new optional fields, without breaking compatibility for existing consumers.
- **Learnability**: Interfaces are required to support programmatic discovery to enable dynamic and autonomous collaboration. A consumer agent is able to programmatically query a provider agent (or a central agent repository) to determine its capabilities, available methods, and the required input/output schemas for each method. This principle is fundamental to creating a flexible multi-agent system where agents are able to find and utilize new tools and services at runtime.
- **Encapsulation and Transparency**: This principle establishes a balance between abstraction and accountability through the following sub-principles:
  - Encapsulation: An agent is required to hide its internal implementation details, such as its specific AI models, internal memory structure, and the logic of its tools. This creates a stable, well-defined interface that separates the public contract from the private implementation, allowing internal components to evolve without affecting consumers.
  - Transparency: While internal logic is encapsulated, the agent is required to not be a completely opaque "black box." It needs to provide mechanisms for explainability and auditability to ensure trust, accountability, and regulatory compliance. For any significant action, the system is required to provide a rationale, allowing operators to understand why a decision was made. This is critical for upcoming regulatory compliance (e.g. the EU's AI Act), and is also critical for debugging, governance, and building operator confidence.
- **Simplicity**: Interfaces should be built upon existing, well-understood standards (e.g. HTTP, JSON-RPC 2.0, Server-Sent Events) wherever possible. This promotes interoperability, reduces integration complexity for developers, and allows the system to leverage the mature ecosystem of tools, libraries, and security practices associated with those standards.

6.2 Key Methods

6.2.1 Agent-to-User

By leveraging advanced agentic AI technology, the next-generation core network is expected to be a service innovation platform that can flexibly combine various network services, including AI, sensing, computing, communication to provide new services for users (e.g. UEs, the 3rd party application and vertical industry tenants). Thus, the *Agent-to-User* interface needs to support the users to request a new service, query the status of an existing service, update or delete it. Accordingly, the network agent returns the service status or results to users.

For different types of users, the interface may take various forms. When a user is a UE, it will communicate with the agents in the CN through a Non-Access Stratum (NAS) interface. This renders the *Agent-to-User* interface a type of new NAS interface. However, when a user is a 3rd party application or a vertical industry customer, it will request a service through an API exposed by the CN. In this case, the *Agent-to-User* interface is a type of new API.

6.2.2 Agent-to-ARF

6.2.2.1 Agent Registration, De-Registration and Update

As described in [i.12], one of the promising directions for future mobile core network architecture is to design agent functionality as a multi-agent system, where the mobile network comprises multiple AI agents with heterogeneous knowledge and skills. For instance, while one of the core network agents is responsible for planning tasks, another one is responsible for deploying end-to-end slices to deliver the requested service. This calls for a framework to allow different AI agent instances to register and de-register themselves and their capabilities with a centralized entity such that other AI agents in the network are able to detect their existence and identify the right agent to interact.

The NRF serves this purpose for network function registration and de-registration in a service-based architecture. The present document recommends the creation of an Agent Repository Function (ARF) with the functionalities described for maintaining agent information in an AI agent-based mobile network. When a new AI agent instance wants to register its profile containing the essential information about the agent, the ARF is the entity that it communicates with. Similarly, when an already registered agent needs to de-register from the mobile network (e.g. for operational reasons, such as energy saving), the ARF is again used to perform a de-registration procedure so that other agents will not discover or will be notified to stop using it. This is illustrated in Figure 6.2.2.1-1

For an agent registration procedure, some important fields that are to be included in an agent profile are agent ID/name, agent role/description, agent skill(s), and agent tool information. An example agent profile is provided in Table 6.2.2.1-1.

Table 6.2.2.1-1: Example information contained in an agent profile used when registering an AI Agent

Agent Profile	
Agent Name	Planning Agent
ID	Unique identifier
Agent Role / Description	Understand the user input in natural language, decompose the intent into multiple executable sub-tasks
Agent Skill	Task graph generation
Agent Tool Information	Web search, weather API, maps API
Endpoint URI(s)	Endpoint address(es) of the agent
API Version	API Version (e.g. semantic versioning can be used as in [i.22])
Auth scheme	Supported authentication scheme
Provider	Information about the agent's provider
PLMN ID	Identifier of Public Land Mobile Network (PLMN) that the agent resides in
Location	Location information of the agent

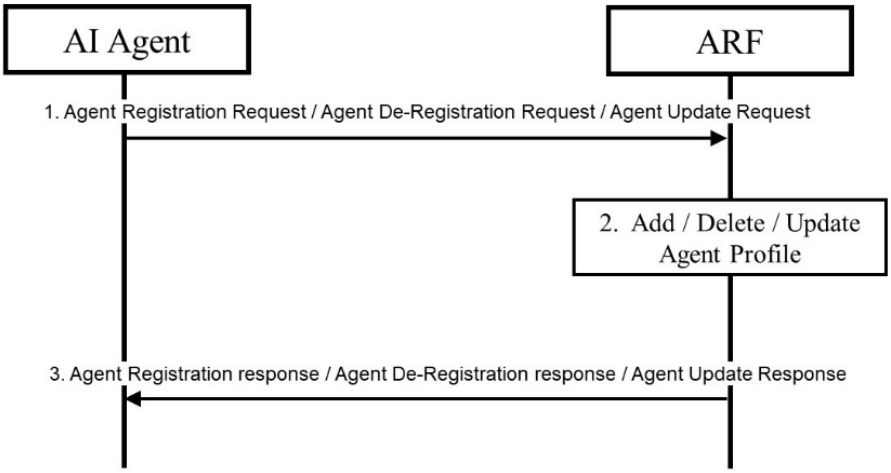


Figure 6.2.2.1-1: Example procedure between an AI agent and the agent repository function for agent registration

6.2.2.2 Agent Discovery and Selection

When an agent wants to interact with other agents to perform tasks collaboratively, it needs to discover other available AI agents that it can interact with. As described in clause 6.1.2, a logically centralized registry (i.e. the ARF stores and maintains the agent profiles of all registered agents). Therefore, this makes it a suitable candidate for providing the necessary information to the requester agent about other available agents upon request.

As it is shown in Figure 6.2.2.2-1, the AI Agent interacts with the ARF by providing information about the target to the ARF. This information contains a description of the task to be performed. Upon receiving the request, the ARF performs hybrid discovery, which is a combination of filter-based (as in the NRF) and semantic matching between the provided description and the description of the agents that have already been registered.

NOTE 1: This differs from NRF discoverability, which is filter-based. Semantic discoverability enables better matching to agent functionality.

The ARF supports exact/Boolean filters over profile metadata (e.g. agent type/role, capabilities, API version, tenant/slice, locality/latency domain, security posture, availability state) to produce a candidate set of agents. The ARF also supports semantic similarity between the task and each candidate's unstructured profile fields (e.g. role/summary, skills, tool descriptions).

An example of the semantic matching is the identification and the provision of the top-K most matching agent profiles (e.g. based on agent roles, agent capabilities, etc.) to the requester. There are two proposed modes of operation:

1) computing semantic similarity or semantic relatedness; and

2) computing a weighted average of filters and semantic matching.

NOTE 2: Semantic similarity is a metric that measures the degree to which two linguistic items share similar meanings through "is-a" relationships (synonymy and hyponymy). It estimates the strength of the semantic relationship between units of language, concepts, or instances through numerical description obtained by comparing information supporting their meaning. In contrast, semantic relatedness is a broader concept that encompasses any linguistic, functional, and/or thematic relation between two terms. Semantic relatedness reflects the degree of semantic feature overlap between words and encompasses the full spectrum of conceptual relationships. Formally, semantic relatedness includes all types of lexical and functional associations between concepts, such as synonymy (similarity using is-a relationships), antonymy (oppositional relationships), meronymy (part-whole relationships), hyponymy (a "kind of" relationship where one term denotes a subtype of another), hypernym (the reverse of hyponymy), co-hyponymy (terms that share the same hypernym but are not synonymous with each other), polysemy (a single word form has multiple related senses or meanings), functional associations (co-occurrence in similar contexts), and thematic relationships (belonging to the same semantic field).

For example, the concepts router and bandwidth are not semantically similar, because router is a type of equipment, while bandwidth is a measurement. However, they have a high semantic relatedness score because there is a strong functional association (routers manage bandwidth) and they co-occur frequently in telecom documentation and troubleshooting.

The operator configures an Agent Selection Policy that determines whether the ARF returns a candidate set for the caller to choose from (Mode A: Recommend) or performs the final selection (Mode B: Delegate). In Mode A, the ARF returns the top-K tuples: {agent_id, score, last_seen, ttl_seconds, health/load_hints}, and the caller selects and is responsible for retries/failover. In Mode B, the ARF returns selected_agent_id plus decision_reason, optional considered_candidates, and route_cache_ttl_seconds. In both, responses are required to include policy_id, request_id, and model_version (if semantic ranking used).

If Mode B fails (SELECTION_FAILED), the caller is required to retry with Mode A unless the policy forbids it from doing so. For both, if there are no candidates, the ARF returns NO_MATCH with missing_requirements.

It is recommended to use semantic relatedness, as it is more precise.

It is further recommended that the score be computed as a weighted average. This is because lexical relationships and thematic relationships operate through different cognitive mechanisms. This takes the form:

$$S_{rel} = w_{lex} + w_{func} + w_{theme}$$

where S_{rel} is the final computed score and w_{lex}, w_{func} and w_{theme} are the weighted averages of all lexical, functional, and thematic relationships used. Functional relationships are often underestimated but can be particularly powerful. Examples include equipment-function relations (e.g. router performs routing and switching), Service Provider relationships (e.g. Mobile Service provided by Mobile Network Operator), and technology-standard relations (e.g. Wi-Fi® uses IEEE 802.11 [i.4]).

It is further recommended that the optimal set of weights is provided from training data.

Figure 6.2.2.2-1 illustrates an optional interaction between the ARF and an AI model to perform the semantic matching jointly. It is important to note that in the depicted example, the ARF is only responsible for discovering a list of candidate agents matching the task description, while the requester agent is the entity selecting the final target agent out of this list. An alternative procedure, which is not illustrated in the present document, is when the ARF selects the target agent based on the description provided in the first query message before sending the agent profile of the selected agent to the requester agent.

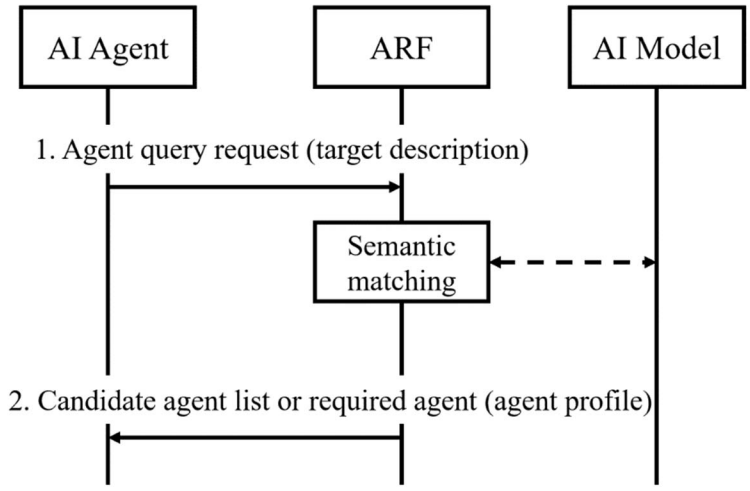


Figure 6.2.2.2-1: Procedure between an AI agent and the ARF for the discovery and selection of a target agent

6.2.3 Agent-to-Agent

The multiple AI agents in core network need to interact with each other to achieve a common goal collaboratively. Similar to A2A, a network agent may require another agent to perform a task. While executing a task, the network agent may query the state of an ongoing task, request to update, and cancel it. These can be performed via the methods of a multi-agent protocol such as the A2A protocol.

Agents interact via a task-centric interface that supports initiating a new task, updating or cancelling an existing task, querying the status of an ongoing task, and an optional event subscription. Each task is addressable by *task_id*, traceable via *correlation_id*, and follows a well-defined lifecycle (such as *submitted*, *working*, *completed*, *cancelled*, *failed* as defined in A2A and shown in Annex A). The Planning Agent decomposes user intent into subtasks and invokes the Assemble Agent using *message/send* with an explicit subtask list, constraints, and expected outputs.

[i.12] considers *Planning Agent* and *Assemble Agent* in the reference architecture. The *Planning Agent* is responsible for decomposing the user intent into multiple sub-tasks and the *Assemble Agent* is responsible for selecting proper functions for each sub-task. After task decomposition step, the *Planning Agent* requests the *Assemble Agent* to perform the function selection task by using the *A2A protocol*. Figure 6.2.3-1 shows the example request sent by the *Planning Agent*.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "message/send",
  "params": {
    "message":{
        "role": "agent",
        "kind": "message",
        "parts":[
            {
                "kind": "text",
                "text": "Selecting the proper functions from the tool repository to execute the
attached sub-tasks."
            },
            {
                "kind": "data",
                "data":[
                    {
                        "sub-task id": 1,
                        "description": "generate a 3D map from location A and location B"
                    },
                    {
                        "sub-task id": 2,
                        "description": "Plan the optimal route from location A to location B"
                    }
                ]
            }
        ]
        "messageId": "acccde2346-9802-bef32-ceff9365"
    }
  },
  "metadata": {}
}
```

Figure 6.2.3-1: Example agent-to-agent message compliant with the A2A protocol

NOTE: The names of method and fields in parameters can be different in the future.

Besides, it has been demonstrated in [i.12] that the multi-agent system in Core Network (CN) can be used to autonomously generate network slices that integrate various network functions and application functions. In this scenario, in addition to *Task* exchanged among AI agents in CN, the information about network slices also needs to be exchanged. Thus, it is required to extend more methods of *agent-to-agent* interface for the operations associated to new types of network slices.

6.2.4 Agent-to-Resource

In an agentic core network, agents are required to access the external data services and tools to improve decision-making. The CN exposes service endpoints for operator data (e.g. UDM/UDR for subscription, AUSF for authentication vectors, PCF for policy, NWDAF for analytics, NEF/EES for event exposure). Agents often need to:

1) Discover available data resources (capabilities, schemas, access policies).

2) Subscribe to change notifications on specific resources (e.g. "UE policy updated", "QoS flow changed") and receive asynchronous events.

3) Read/write resource contents via the owning service subject to authorization and audit.

For example, during UE Registration, the responsible agent obtains authentication vectors by invoking AUSF/UDM services; it does not read authentication data from a shared cache.

Since data classes differ in security, privacy, and latency characteristics (e.g. UE subscription/policy data vs. shared knowledge bases), the interface is required to be modular, with per-class policies for authorization, retention, and QoS.

Common tools, such as functions and APIs, are published in a Tool Registry (metadata, version, auth scope). Tools are executed via a Tool Execution Service or delegated to the owning agent. Agents need to query and subscribe to tool availability and versions. In addition, agents need to be able to remotely execute a tool with explicit inputs, timeouts, idempotency keys, and billing/quotas.

To enhance reliability, a verification environment (e.g. Sandbox in [i.12]) is introduced in CN to test and verify the decision results of agents. The Sandbox is required to support shadow/canary execution and offline replay. Agents are allowed to submit proposed actions for validation and receive test results. Sandbox endpoints are required to be isolated from production state.

The Model Context Protocol (MCP) specifies how an AI agent (or LLM) interacts with the external tools and data sources. Therefore, it is able to be used as the agent-to-resource facade for tool/data access.

6.2.5 Agent-to-Infrastructure

The core network relies on RAN and compute/transport infrastructure. Agents interact with these domains to perform:

- RAN control via the Agent-gNB (N2) procedures.
- User plane path management.
- NF/compute lifecycle management.
- Computing and communication resource control

For example, a UE issues a PDU Session Establishment request (NAS to an Agent in CN). The responsible agent coordinates and invokes the NFs, such as the AMF and the SMF which selects UPF(s) and returns session rules; the agent sends N2 SM messages to the gNB to create the RAN PDU Session context.

If a request requires reconfiguration of NF instances or compute, the responsible agent issues intent/policy changes and generates the actions (e.g. scale SMF, adjust UPF placement), which then enforces them.

NOTE: An adaptor is needed to translate the instructions generated by agents to the standard signalling that can be understood by the infrastructure, the adaptor can be deployed in agents or infrastructure.

Designing the agent-to-infrastructure interface therefore means:

1) control the infrastructure intelligently and autonomously based on the decisions of agents;

2) preserving observability and rollback (idempotent operations, audit logs, and failure handling).

6.3 Agent Communication Model

An AI agent possesses the autonomy to explore optimal solutions for achieving its goals. Moreover, the capabilities of AI agents evolve, resulting in non-deterministic outputs. Consequently, interactions among AI agents are inherently flexible and can vary depending on the agents' outputs.

When the conversation flow between agents is not fixed or pre-defined, the messages generated by the agents do not specify the receivers. In such a setting, to facilitate efficient multi-agent conversation among a group of agents, an intermediary entity is required to route the messages based on their content. In the following, this entity is referred as the Agent Communication Proxy (ACP).

The ACP is responsible for parsing the capability requirements from the message envelope (optionally augmented by LLM-based hints).

NOTE: The routing mechanisms of ACP can be further studied in the future.

This communication model supports scalability by allowing easy expansion or modification of agent roles and dynamic adjustment of communication sequences. Additionally, deploying multiple ACPs across different regions enables parallel message routing, preventing any single ACP from becoming a communication bottleneck.

Taking the agents demonstrated in [i.12] as an example, a "planning agent", which is responsible for decomposing the complex service request into multiple executable sub-tasks, would transmit the output to the ACP. This message comprises key fields as depicted in Figure 6.3-1 for improved understanding. After having received the message, the ACP parses the task description and generates a capability requirement for selecting a target agent. Subsequently, it sends a request to the ARF with the requirement provided, and receives the profile of the selected target agent as a response. Finally, it forwards the message originating from the planning agent to the target agent, i.e. assemble agent, which is another agent from [i.12]. Note that in this example procedure, the ARF performs the selection of the target agent, instead of sending a list of candidate targets to the ACP.

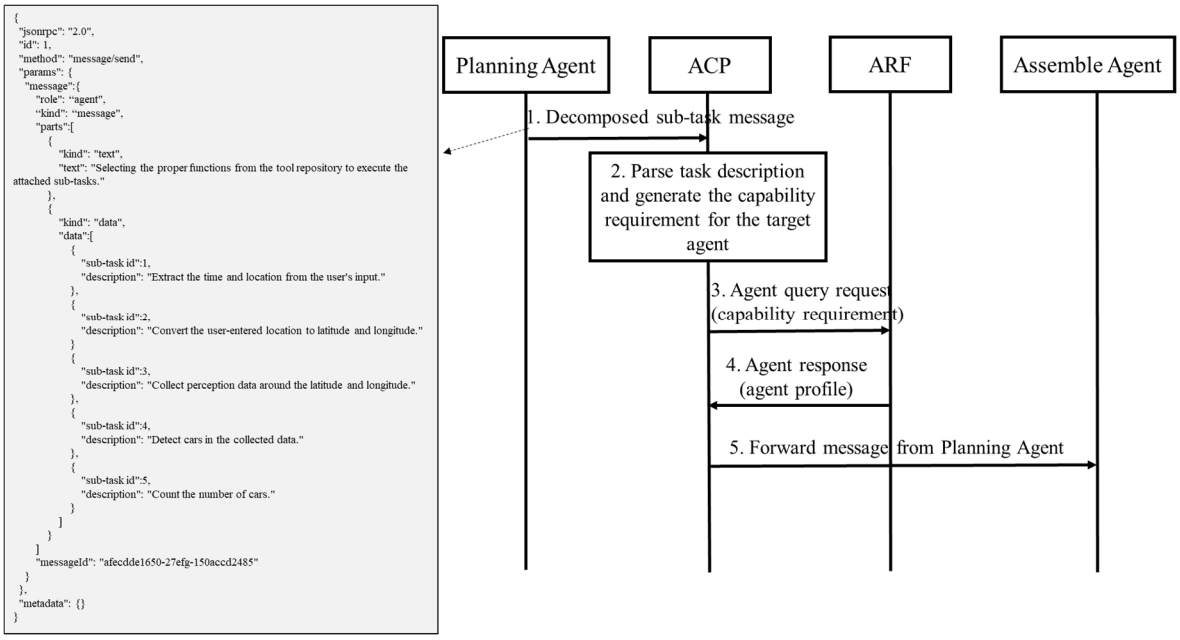


Figure 6.3-1: Example procedure for indirect agent-to-agent communication where the agent communication proxy routes messages between two communicating partners

Alternatively, direct agent-to-agent communication corresponds to the case when there is no intermediary between two communicating AI agents. In this case, an AI agent that is about to handoff a task to a second agent is required to determine the most suitable target agent for the task at hand, before sending the message to the destination. Using the same example from Figure 6.3-1, suppose that the "planning agent" decomposes the complex request and generates the output message. In order to identify the most suitable target, it first uses an Agent Selection Policy as defined in clause 6.2.2.2. All A2A interactions (direct or via ACP) are required to apply the Agent Selection Policy. If indirect (via ACP), then for Mode A, the ARF returns the Top-K tuple, from which the ACP selects under operator policy (e.g. latency/load/trust), forwards the message, and logs the *routing_decision*. If Mode B is used, then the ARF returns *selected_agent_id*, and the ACP forwards without altering selection. If direct (e.g. agent to agent) communication takes place, then for Mode A, the requester agent selects from Top-K and handles retries/failover. If Mode B is used, then the requester uses *selected_agent_id* and proceeds; on delivery failure it attempts a re-query (policy permitting).

7 Multi-Agent Collaboration Mechanism

7.1 Workflow Orchestration

A workflow in the context of agentic AI refers to the orchestration and coordination of tasks carried out by multiple AI agents. As defined in [i.16], "workflows are systems where LLMs and tools are orchestrated through pre-defined code paths". There are a few workflow patterns that are commonly found in existing multi-agent frameworks.

One of the most prominent workflow patterns is when the AI agents in the multi-agent system are activated in a pre-defined sequence. In their documentation, LangGraph and Agent Development Kit (ADK) refer to this approach as *prompt chaining* and ***sequential pipeline pattern***, respectively. It is worth mentioning that while LangGraph achieves this by defining nodes (agents) and edges (task handoffs) to strictly control the sequence, ADK defines a special type of agent, i.e. *SequentialAgent*, which comprises multiple sub-agents (workers) and calls them in the order they are provided during instantiation. AutoGen implements the Further, CrewAI utilizes *sequential processes* for the same purpose where they assign a *Task* characterized by a task description and expected output to an *Agent*, where each task has an AI agent assigned to it. The workflow is initiated in the order the tasks are registered to it. An example usage of this pattern is a multi-agent system that is comprised of a writer, formatter agent and user agent. These agents perform an essay writing task given an input prompt, where the agents are triggered in the given order, i.e. writer agent generates the text based on the description, formatter agent polishes the draft by refining the grammar and user agent presents the final refined output to the human user, completing the workflow.

The second workflow pattern is the ***iterative refinement pattern*** as it has been named in ADK. It refers to the type of orchestration pattern, where one or more agents work on a task over multiple iteration. To implement the iterative refinement pattern, ADK defines a special built-in workflow agent type, namely the *LoopAgent*, which runs one or more sub-agents repeatedly until a termination condition is met. This is similar to the iterative writing process of a human writer, which undergoes multiple iterations until the polished document is published. A common example of this pattern in multi-agent systems is when the workflow involves a code generation and refinement task, which is typically performed via multiple task handoffs between code generator and code reviewer agents. In LangGraph and AutoGen, this shows up as the *evaluator-optimizer* pattern in LangGraph and the *reflection pattern* by Autogen. In both, one LLM (or agent) generates, another evaluates and feeds back, iterating until a stop condition is met.

Another common multi-agent pattern is ***parallel fan-out-gather*** pattern in ADK, which is referred as the *parallelization* mechanism for multiple LLMs in LangGraph. As the name suggests, agents run concurrently (i.e. simultaneously) to perform the same task, often followed by a later agent serving as the aggregator of the results. This approach is particularly beneficial for tasks where tasks in a workflow which:

- can be performed independently without any direct or indirect information exchange between the concurrent agents, e.g. via messages or a shared state, respectively, and
- are resource-intensive potentially leading to long task completion duration when not parallelized.

ADK has a built-in *workflow agent* type to implement parallel fan-out-gather pattern, i.e. the *ParallelAgent*. It executes multiple sub-agents concurrently. The *ParallelAgent* aggregates the results after each sub-agent completed its execution. ADK also supports communication between agents for more flexible implementation options. For instance, a shared *InvocationContext* object or an external database or queue can be used, while the concurrent access needs to be managed carefully to avoid race conditions, e.g. using locks. An example task where the parallelization is applicable is a parallelized web research, where each sub-agent performs search based on a different query. In LangGraph, the parallelization is supported through the concept of *supersteps*. More specifically, LangGraph models agent workflows as graphs, the behaviour of which is characterized by three key components, *state*, *nodes*, and *edges*. By composing *nodes* and *edges*, one can create complex workflows that evolve the *state* over time. As stated in their documentation, "while nodes do the work, edges define what to do next". LangGraph's underlying graph algorithm is based on message passing. When a node (e.g. an agent) completes its operation, it sends messages along one or more edges. Thus, the program is executed in discrete "super-steps", where each super-step is a single iteration over the graph nodes. The parallelization lies in the fact that the nodes that are part of the same super-step are run concurrently. Nodes in the same super-step run in parallel (LangGraph uses a message-passing model). For instance, if a node (i.e. an agent) has multiple outgoing edges according to the workflow, all of the destination nodes are executed parallelly as they are part of the subsequent superstep.

A very important multi-agent pattern is the ***human-in-the-loop*** that integrates the human intervention points within an agent workflow. This is useful for tasks that require human oversight, approval, or correction that AI cannot perform. Although the name and the purpose of the human-in-the-loop pattern is consistent among LangGraph, ADK, and AutoGen, its implementation slightly differs. AutoGen provides a built- *UserProxyAgent* for human participation. LangGraph provides a built-in function, *interrupt*, that pauses the graph at a specific node and resumes it after the corresponding information is presented to the user and the user input is provided. ADK does not have a built-in agent or function, and instead relies on custom agents and/or tools to integrate human interaction.

All of the previous patterns that have been introduced so far have a deterministic nature with respect to how the workflow orchestration is performed. In particular, they are characterized by a fixed sequence of task execution and agent activation, where each step follows a deterministic and pre-defined order. In the present document, this category of multi-agent patterns is called "explicit workflow". Explicit workflow is useful when one needs strict control over how a series of tasks or agents are executed, which in turn improves the predictability and reliability of the system, by ensuring that tasks are performed in the required order or pattern.

Contrarily, the "dynamic workflow" follows a non-deterministic flow, offering a greater flexibility in task orchestration and execution. In the case of dynamic workflow, the sequence of tasks and selection of agents are determined at runtime based on the condition(s), context, and data. In contrast to explicit workflow, dynamic workflows offer flexibility in routing, agent selection and task execution patterns. A typical scenario for dynamic workflow orchestration is when the multi-agent system is operating in complex and unpredictable environments, where identifying a deterministic order of task handoffs beforehand is impractical.

The dynamic workflow calls for runtime decision-making to determine the control flow of processes either in a centralized or decentralized way. This is best reflected in AutoGen's ***selector group chat*** pattern where a group of agents exchange messages to perform a collaborative task and the decision for the activation order is determined centrally by a *GroupChatManager*. The group chat manager selects the next agent to speak using an LLM depending on the task at hand and the roles of agents participating the conversation. This is achieved by providing a custom prompt to the manager that provides information about the roles of the participating agents in the group chat, where each agent is typically specialized for a particular task or domain. This renders the workflow non-deterministic, as the sequence of agents is determined during runtime. Possible extensions and variations of selector group chat pattern are possible such as selecting multiple agents at once for a round of conversation, instead of selecting the next speaker one-by-one.

7.2 Closed-loop Optimization Mechanism

7.2.1 Multi-Agent Coordination and Optimization

Figure 7.2.1-1 illustrates the procedure of closed-loop optimization mechanism, which uses the Semantic Bus of ENI to achieve functions such as task allocation and scheduling, AI agent management, AI agent recommendations and convergence, AI agent coordination and collaboration, and multi-agent service provision and feedback.

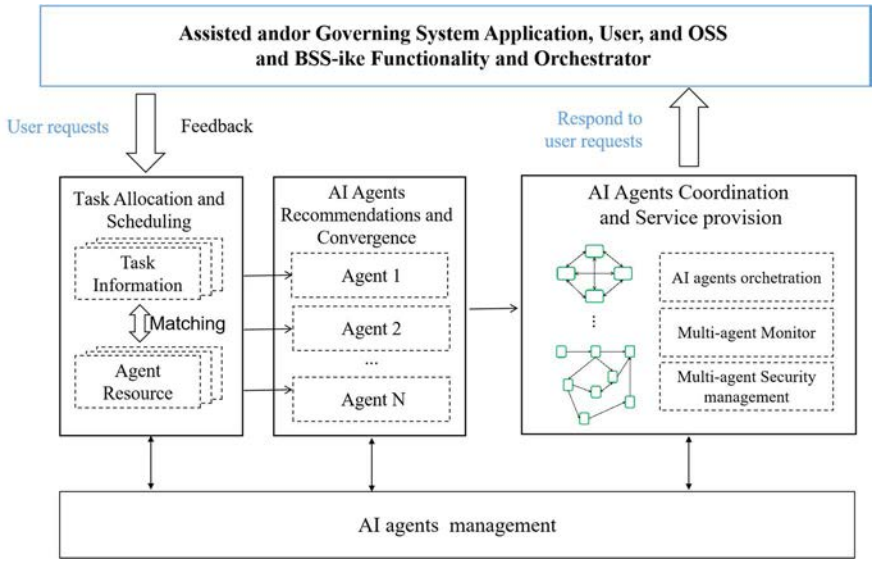


Figure 7.2.1-1: Closed-loop optimization mechanism procedure

Task allocation and scheduling

The present document defines a strategy that uses a MAS to divide complex AI tasks (user requests) into smaller, manageable sub-tasks that can be assigned to individual AI agents, ensuring that each sub-task is well-defined and that dependencies between tasks are clearly understood and properly managed.

AI agents' recommendations and convergence

Task allocation, scheduling, and agent assignment are required to run inside an ENI-compliant closed loop with explicit observe/decide/act boundaries, policy gates, and rollback. Agent selection is required to follow the Selection Policy defined in clause 6.2.2.2.

AI agents' coordination and collaboration

The present document recommends the development of coordination mechanisms to manage the interactions and dependencies between AI agents, using centralized or decentralized approaches based on the complexity and scale of the system. This promotes information sharing and integration among multiple recommended agents, enabling them to respond to complex tasks and requests as a unified whole.

Multi-agent service provision and feedback

The independent execution and interaction with external systems and users provides functional capabilities for external applications, responds to user requests, performs tasks, and provides results. In conjunction with continuously monitoring the performance of MASs enables feedback to be provided to improve their performance.

7.2.2 Network Feedback Reinforcement Learning

Figure 7.2.2-1 shows an architecture where multiple agents receive rewards, for instance considered to fine-tune their AI models. The public memory records the interactions and output decisions of the agents, as well as the performance of Customized Service Networks (CSNs) and user feedback. Therefore, the public memory (i.e. shared memory component between agents) is filled with the experience of multi-agent system.

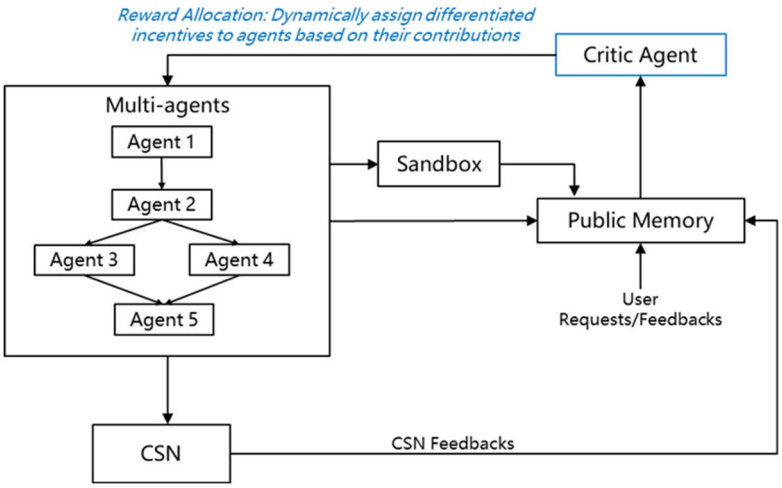


Figure 7.2.2-1: Network feedback reinforcement learning procedure

The critic agent evaluates the joint policies of all agents participating in the creation of CSN from a global perspective.

Performance-based rewards:

During multi-agent collaboration, different agents typically contribute differently to the CSN creation. However, feedback from CSNs and users is typically sparse and based on the joint actions of all agents. Therefore, the critic agent is required to allocate reward incentives, such as counterfactual baselines (e.g. COMA), difference rewards, or Shapley-value approximations to attribute joint rewards to individual agents based on their specific contributions.

Contribution-aware optimization:

Reward decomposition is required to use a credit-assignment method (e.g. counterfactual advantage, difference rewards) under Centralized Training, Decentralized Execution Training. This trains with a global critic and deploys agents that act on local observations. Such training is required to be offline/sandboxed, with shadow validation and canary rollouts before production use. Different reward signals help agents recognize their role-specific impacts. Each agent can use their assigned rewards to fine-tune their local models and guide targeted self-improvement.

7.2.3 Multi-Agent Self-Reflection

Apart from model fine-tuning, each agent can also achieve self-reflection through prompt engineering, which is a more lightweight process. Figure 7.2.3-1 illustrates the procedure of closed-loop agent optimization via self-reflection.

The short-term memory stores various feedback signals, including user feedback, CSN feedback, and verification results from the sandbox. By analysing the agents' interaction and feedback, the critic agent can send error or performance-based rewards to the agents. The agents can use these error or reward signals to self-reflect and generate reflection text, to be stored in the long-term memory. This reflection text captures high-level knowledge derived from the agents' self-reflection.

In this way, in-context information and feedback from the network can be transformed into long-term memory through reflection. When making decisions, agents retrieve knowledge from the long-term memory to augment their prompts and generate decisions. By leveraging this reflected knowledge, agents can improve the quality of their decisions, thus achieving optimization. It is recommended that reflection only augments prompts. It is recommended not to let reflection override hard policy or security constraints.

Error signals facilitate the correction of agents' decisions, while reward signals enhance the quality of their decisions. This dual approach ensures continuous improvement and optimization of the agents' performance.

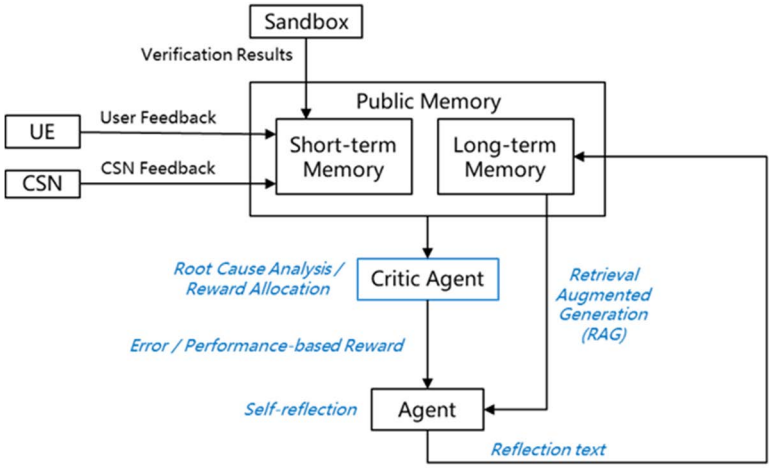


Figure 7.2.3-1: Procedure of closed-loop optimization by self-reflection

Self-correction by reflection:

When an error occurs in the network, the critic agent can intelligently identify the agent responsible for the error, analyse the cause, and send the error analysis to the targeted agent. Based on this error feedback, the agent reflects on its previous decision, generates reflection text that includes strategies to avoid repeating the error, and stores this text in the long-term memory. By leveraging Retrieval-Augmented Generation (RAG) techniques from the long-term memory, the agent can use this reflective knowledge to avoid similar errors in the future, thus achieving self-correction.

Self-enhancement by reflection:

Different decisions are associated with varying rewards. By comparing different trajectories and their corresponding rewards, the agent can summarize performance records related to similar tasks and generate reflection text that outlines strategies for achieving higher rewards. By storing this knowledge in the long-term memory, agents can use RAG to improve the quality of their decisions and enhance their performance.

Safety ensured by reflection:

It is recommended that reflections include provenance, confidence, and TTL. It is further recommended that only trusted reflections (post-validation) are used to influence production prompts. For instance, threshold-based policies can be considered to filter out reflections with low confidence, which in turn helps to improve safety. It is further recommended that reflections are not allowed to bypass operator policy or security controls.

7.2.4 Multi-Agent Conflict Resolution

In a multi-agent AI-Core system, multiple agents operate in a distributed fashion to fulfil different parts of an E2E service request. However, without coordination, conflicts can occur in various forms. Additionally, dynamic changes in resource availability, component failures, or delayed execution is likely to render static planning and assembling strategies ineffective, resulting in conflicts and system instability, if unresolved, can severely disrupt system performance. Therefore, it is recommended to define conflict classes and priorities:

1) Resource contention;
2) topology/order inconsistencies;
3) concurrent state mutation; and
4) policy violations.

This is an exemplary list; refinement and priority assignment (e.g. safety > availability > cost) is for further study.

Take the AI-agent-based core network architecture described in [i.12] as an example. The *planning agent* is responsible for decomposing high-level intents or policies into executable tasks and the *assemble agent* maps the resulting sub-tasks to the available tools. These occasionally reach a conflicting state because of limited knowledge about runtime information from other network elements. In such a setting, it is likely that neither agent has the global perspective to arbitrate between conflicting decisions, handle dynamic runtime failures, or coordinate retries and fallbacks.

To address these challenges, Multi-Agent Debate (MAD) introduces a collaborative and autonomous approach to resolve conflicts, offering benefits such as decentralized decision-making, flexibility, and avoiding single point of failure. However, in certain scenarios, particularly in complex and dynamically changing environments, MAD faces limitations including high communication- and computational overhead, as well as difficulty in reaching convergence. In such cases, the introduction of a *Conflict Resolution Agent (CRA)* to detect, arbitrate, and resolve multi-agent conflicts can be a practical and effective complement to MAD, as demonstrated in [i.18]. Rather than replacing MAD, it enhances system stability and coordination efficiency, making the two approaches mutually supportive.

It is recommended to keep MAD within time and communication budgets. It is further recommended to fall back to CRA decision with explainable rationale when non-convergent.

The CRA serves as both the system's conflict detector and resolver. It continuously monitors outputs such as agent feedback, CSN resource utilization, task dependencies, and compares them to user intent to identify configuration inconsistencies or resource contention, enabling early detection of potential or actual conflicts. When a conflict is detected, the agent formulates resolution strategies based on factors like policy priorities, current system load, and task urgency, while also leveraging agent models to reference historical conflict resolution strategies to inform its decisions. These strategies often involve halting certain actions, rescheduling tasks, or modifying execution parameters. Once a strategy is selected, the agent dispatches coordination instructions, which typically includes adjusting execution sequences, reallocating or modifying resources, and supporting inter-agent negotiation to align objectives and resolve discrepancies.

Moreover, when the CRA detects a conflict that cannot be resolved autonomously, such as insufficient resources to satisfy all requested functions, it can request human involvement for the final decision-making. Escalation should produce options with trade-offs (partial fulfil, degrade, reject) and simulate them in the sandbox first. In other words, the unresolved issue is forwarded to human operators or intent managers, who can reassess priorities, make trade-offs, or override automated decisions. Trade-offs include:

- Partial Fulfilment: Provisioning only a subset of the requested functions or serving a reduced number of customers.
- Service Degradation: Adjusting service performance levels to align with current resource availability or system constraints.
- Request Rejection or Deferral: Temporarily denying or postponing service requests based on system capacity, policy constraints, or operational priorities.

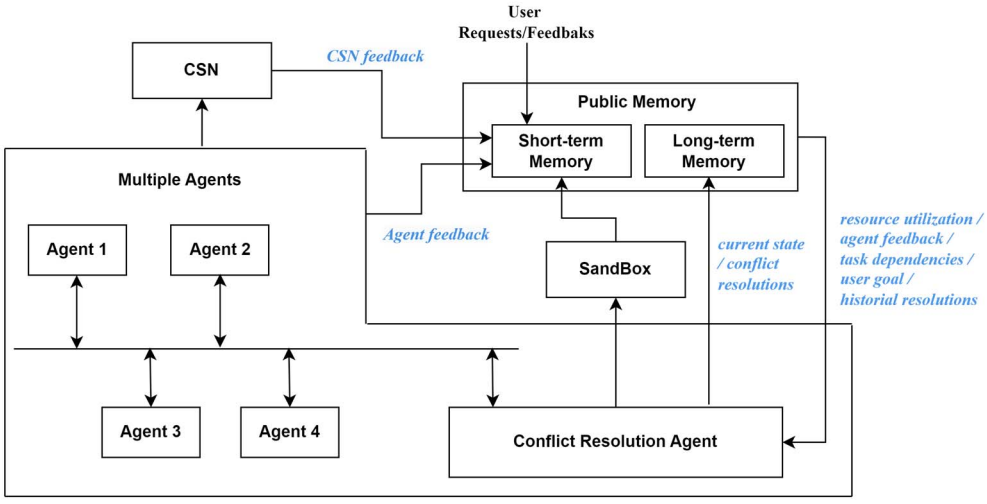


Figure 7.2.4-1: Procedure of Multi-Agent Conflict Resolution

Figure 7.2.4-1 illustrates an example procedure of multi-agent conflict resolution in the AI-Core architecture from [i.12]. The CRA interacts closely with other components of the AI core to form a coherent and adaptive decision-making system. These can include:

- the communication between the *conflict resolution agent* and other AI core agents, such as the planning or assemble agent by monitoring their outputs, thus, identifying potential inconsistencies or conflicts, and coordinating the synchronization and interactions among agents;

- running simulations by interacting with the sandbox component to evaluate the feasibility or performance impact of different conflict-resolution strategies before deployment;
- storing the result of conflict resolutions in public memory, which stores both real-time and historical knowledge;
- retrieving the historical records of previous conflicts and their resolutions, while also logging new conflict cases to support continuous learning and future optimization.

The CRA is required to enforce conflict-resolution invariants and use saga-style compensations for partial execution. When MAD exceeds latency or message budgets or fails to converge, the CRA is required to decide or escalate with sandbox-simulated alternatives.

8 Standard Impact Analysis

To enable multi-vendor interoperability among AI Agents, in the future core network, various roles of core network AI Agents and their communication interfaces as well as the new multi-agent communication protocols need to be standardized.

As described in clause 5.3 and [i.12], different core network AI Agents with distinct roles, skills, and responsibilities are required for an agent-based next-generation core network. Their logical operational principles are significantly different from those of the conventional Network Functions (NFs) in the service-based architecture. These AI Agents and their agent profiles need to be standardized so that they can discover each other. In addition to agents, other shared components to facilitate agent collaboration and operation, such as shared memory, tool repository and agent repository, also need to be standardized to support usage by multiple AI Agents.

The present document recommends the prioritization of the following functions for standardization:

1) An Agent Repository Function (ARF) service and Agent Profile schema (ID, capabilities, endpoints, versions, auth, lifecycle, health, policy tags):
    a) The interfaces exposed by AI Agents, including agent-to-user, agent-to-ARF, agent-to-agent, agent-to-resource, and agent-to-infrastructure interfaces described in clause 6, are required to be standardized to enable the interactions of various agents, users and infrastructures.
    b) This includes the standardization of methods, input and output parameters of each interface.
2) Agent-to-agent task-lifecycle semantics (create/get/update/cancel, idempotency, retries).
3) Agent-to-resource to invoke existing CN NFs (UDM/AUSF/PCF/NWDAF/NEF).
4) Observability/security (correlation IDs, audit, RBAC/ABAC, tenancy).

In addition, new application protocols, such as A2A and MCP, are recommended to be standardized. Transports (HTTP/3 over QUIC) are recommended to follow existing IETF RFCs; SDOs should define telecom profiles on top (security, streaming, payloads).

9 Conclusion and Recommendations

Agentic AI technology is rapidly evolving and is expected to have a significant impact on the next-generation telecommunication systems, particularly the mobile core. Architectures based on multiple agents offer clear advantages over single-agent systems, including improved modularity, enhanced robustness, distributed decision-making, and more efficient energy consumption during training and inference *when specialization reduces compute per task and orchestration overhead is controlled*.

In this context, inter-agent communication emerges as a critical topic for next-generation mobile network architectures. To that end, the present document studies existing multi-agent topologies, workflow orchestration methods and a selected subset of existing open-source multi-agent frameworks and multi-agent communication protocols. Further, it defines five types of interfaces for the interaction of core network AI agents with other entities, including those facilitating communication between agents themselves as well as interfaces connecting users, resources, and infrastructure to agents. In addition, the mechanisms to improve collaboration between multiple agents are studied in the context of core network agents. These include the selection of the most suitable workflow orchestration mechanisms, applying closed-loop optimization methods, such as reinforcement learning based on runtime network feedback, self-reflection, and conflict resolution among agents.

It is recommended that the adoption of AI agents in mobile core networks is evaluated in a practical study (e.g. through a proof-of-concept implementation showcasing their key features and capabilities in the context of telecommunication systems). Since the existing frameworks and multi-agent communication protocols have been designed primarily without the telecommunication use case in mind, it is of the utmost importance to evaluate whether these sufficiently capture the needs of a mobile core network. To that end, it is recommended to study and specify (if necessary) a multi-agent interface and determine if a profile and/or extension of MCP/A2A with telecom semantics is required. If gaps remain after this effort is completed, then one or more new wire protocols need to be examined.

Annex A:
More details about the A2A Protocol

A2A defines several RPC methods invoked by the A2A client by sending an HTTP POST request to the A2A server's URL, including:

- *message/send*: Sends a message to an agent to initiate a new interaction or to continue an existing one.
- *message/stream*: Sends a message to an agent to initiate/continue a task and subscribes the client to real-time updates for that task via SSE.
- *tasks/get*: Retrieves the current state (including status, artifacts, and optionally history) of a previously initiated task. This is typically used for polling the status of a task or for fetching the final state of a task.
- *tasks/cancel*: Requests the cancellation of an ongoing task.
- *tasks/pushNotificationConfig/set*: Sets or updates the push notification configuration for a specified task. This allows the client to tell the server where and how to send asynchronous updates for the task.
- *tasks/pushNotificationConfig/get*: Retrieves the current push notification configuration for a specified task.
- *tasks/resubscribe*: Allows a client to reconnect to an SSE stream for an ongoing task after a previous connection was interrupted.

Further, A2A defines several protocol data objects that define the structure of data within these JSON-RPC methods. The most important of these are *Task*, *TaskStatus*, *Message*, *Artifact*, and *Part*.

Task represents the stateful unit of work being processed by the A2A server for an A2A client. A task encapsulates the entire interaction related to a specific goal or request. Table A-1 lists its key fields.

Table A-1: Key fields of a *Task*

Field Name	Type	Required	Description
id	string	Yes	Server generated unique task identifier (e.g. UUID).
contextId	string	Yes	Server generated ID for context across interactions.
status	*TaskStatus*	Yes	Current status of the task (state, message, timestamp).
artifacts	*Artifact*[]	No	Array of outputs generated by the agent for this task.
history	*Message*[]	No	Optional array of recent messages exchanged.
metadata	Record<string, any>	No	Arbitrary key-value metadata associated with the task.

TaskStatus represents the current state and associated context of a Task. The structure of a TaskStatus is demonstrated in Table A-2.

Table A-2: Key fields of a *TaskStatus*

Field Name	Type	Required	Description
state	enum	Yes	Current lifecycle state of the task.
message	*Message*	No	Optional message providing context for the current status.
timestamp	string	No	Timestamp when this status was recorded.

The possible *state*s of a task include:

- **submitted**: Task received by the server and acknowledged, but processing has not yet actively started.
- **working**: Task is actively being processed by the agent. In certain instances, the client expects further updates or a terminal state.
- **input-required**: The agent is paused, awaiting additional input from the client.
- **completed**: Task finished successfully. Results are typically available in *Task.artifacts* or *TaskStatus.message*.

- **cancelled**: Task was cancelled (e.g. by a *tasks/cancel* request or server-side policy).
- **failed**: Task terminated due to an error during processing. *TaskStatus.message* usually contains error details.
- **rejected**: Task terminated due to rejection by remote agent. TaskStatus.message typically contains error details.
- **auth-required**: Agent is paused, awaiting additional authentication from the client/user to proceed.
- **unknown**: The state of the task cannot be determined (e.g. task ID is invalid, unknown, or has expired).

The actual content of the communication is carried by three main data objects:

- Message: Represents a single turn in a conversation, such as an instruction, prompt, or reply. It has a role ("user" for the client, "agent" for the server) and contains one or more Part objects.
- Artifact: Represents a tangible output or deliverable generated by the agent during a task, such as a document, image, or structured data. It also contains one or more Part objects.
- Part: The fundamental content unit. A Part can be simple text (TextPart), a file reference (FilePart), or structured JSON data (DataPart), enabling rich, multi-modal exchanges.

Message represents a single communication turn or a piece of contextual information between a client and an agent. Messages are used for instructions, prompts, replies, and status updates, they include the key information listed in Table A-3.

Table A-3: Key fields of a *Message*

Field Name	Type	Required	Description
role	"user" | "agent"	Yes	Indicates the sender: "user" (from A2A Client) or "agent" (from A2A Server).
parts	*Part*[]	Yes	Array of content parts. It contains at least one part.
metadata	Record<string, any>	No	Arbitrary key-value metadata associated with this message.
extensions	string[]	No	A list of extension URIs that contributed to this message.
referenceTaskIds	string[]	No	List of tasks referenced as contextual hint by this message.
messageId	string	Yes	Message identifier generated by the message sender.
taskId	string	No	Task identifier the current message is related to.
contextId	string	No	Context identifier the message is associated with.
kind	"message"	Yes	Type discriminator, literal value.

Artifact represents a tangible output generated by the agent during a task and includes the fields listed in Table A-4.

Table A-4: Key fields of an *Artifact*

Field Name	Type	Required	Description
artifactId	string	Yes	Identifier for the artifact generated by the agent.
name	string	No	Descriptive name for the artifact.
description	string	No	Human-readable description of the artifact.
parts	*Part*[]	Yes	Content of the artifact, as one or more Part objects.
metadata	Record<string, any>	No	Arbitrary key-value metadata associated with the artifact.
extensions	string[]	No	A list of extension URIs that contributed to this artifact.

A **Part** is a union type representing exportable content as either TextPart, FilePart, or DataPart, the structure of them are shown in the following tables.

Table A-5: Key fields of a *TextPart*

Field Name	Type	Required	Description
kind	"text" (literal)	Yes	Identifies this part as textual content.
text	string	Yes	The textual content of the part.
metadata	Record<string, any>	No	Optional metadata specific to this text part.

Table A-6: Key fields of a *FilePart*

Field Name	Type	Required	Description
kind	"file" (literal)	Yes	Identifies this part as file content.
file	*FileWithBytes* or *FileWithUri*	Yes	Contains the file details and data/reference.
metadata	Record<string, any>	No	Optional metadata specific to this file part.

Table A-7: Key fields of a *DataPart*

Field Name	Type	Required	Description
kind	"data" (literal)	Yes	Identifies this part as structured data.
data	Record<string, any>	Yes	The structured JSON data payload (an object or an array).
metadata	Record<string, any>	No	Optional metadata specific to this data part (e.g. reference to a schema).

Annex B:
AGNTCY™ and Agent Connect Protocol

AGNTCY™ is an open-source "collective" that aims to create an open, interoperable internet for agent-to-agent communication. It aims to standardize how agents declare their skills and discover each other based on searches on those skills. It introduces an extensible data model, the *Open Agent Schema Framework (OASF)*, describing agent attributes, capabilities, and metrics to allow standardized discovery and evaluation (e.g. for workflow orchestration). The OASF defines a taxonomy of agent skills, where each agent skill has a unique identifier, a description and belongs to a skill category. For instance, under the skill category natural language processing with the category identifier 1, there are multiple agent skills including but not limited to natural language understanding (with identifier 101), natural language generation (with identifier 102). Similarly, the multi-modal skill category with category identifier 7, there are image processing (701), audio processing (702), and any-to-any transformation (703). The OASF defines further distinguishes between specific agent skills such as image to text (70101), text to image (70102), text to video (70103), and so on, by relating them to the image processing skill through the way they select their identifiers. However, according to the current documentation they are all defined as agent skills and they do not introduce a term to distinguish between image to text skill (70101) and image processing (701).

In addition to the OASF, AGNTCY™ considers the *agent connect protocol* in order to enable agent-to-agent communication in a standardized fashion. The agent connect protocol addresses authentication (how a caller authenticates with an agent), configuration (how to configure a remote agent), invocation (how to invoke a remote agent providing input for its execution), output retrieval (how to retrieve the result of an agent invocation), interrupt handling (how agents notify the caller about execution suspension), and error handling for multi-agent communication. The agent connect protocol specifies the interface that an agent exposes to allow for its invocation and configuration. Although it specifies methods (i.e. endpoints) for each of these mechanisms (i.e. configuration, invocation, etc.), it does not specify the format of the data structures that an agent receives or produces.

One of the most important concepts proposed by the is the AGNTCY™ *agent manifest*. The agent manifest is a standard format to describe agents, their capabilities, how to deploy and consume them. It is designed to be used by the agent connect protocol, stored in an *agent directory* with corresponding OASF extensions. The agent manifest contains a unique name and a version to identify an agent within the namespace it is part of. In addition, an agent manifest contains an agent description that describes what the agent is capable of, which in turn helps other AI agents to select the best agent for multi-agent collaboration. The agent manifest contains a section dedicated to *deployments*. This specifies the deployment options that the respective AI agent supports. To elaborate, AGNTCY™ considers three deployment options for AI agents:

1) Remote service deployment: In this case, the agent does not come as a deployable artifact, but it is already deployed and available as a service. If the manifest indicates that the remote service deployment option is supported, then it also provides where the agent is accessible (i.e. the network endpoint), and the authentication information to be used by the agent connect protocol.

2) Docker deployment: In this case, the agent can be deployed starting from a docker image. If the docker deployment option is supported, then the manifest also contains the image of the agent container, and the authentication to be used by the agent connect protocol for this agent.

3) Source code deployment: In this case, the agent can be deployed starting from its code. If the manifest indicates that the source code deployment of the agent is supported, then it also provides the location of the source code (e.g. URL), the framework that the source code has been developed with (e.g. LangGraph), and further framework-specific configuration information necessary to run the agent. See Figure B-1 for an example agent manifest for an AI agent that supports source code deployment.

An example part out of an agent manifest is given as follows:

```
{
    …
        “deployments”: [
            “type”: “source_code”,
            “name”: “src”,
            “url”: “git@github.com:agentcy/mailcomposer.git”,
            “framework_config”: {
                “framework_type”: “langgraph”,
                “graph”: “mailcomposer”
            }
        ]
    …
}
```

Figure B-1: Example deployment options section in an agent manifest that belongs to a mail composer agent

History

Document history		
V4.1.1	October 2025	Publication