Network Management Research Group
Internet-Draft
Intended status: Informational
Expires: 8 January 2026

M. Wang
Y. Xie
Q. Wu
Huawei

7 July 2025

A Protocol-agnostic Multiple Agents Interaction Model for Autonomous Network
draft-wang-nmrg-magent-im-00

Abstract

In the push toward Level 4 Autonomous Networks, CSPs must to adopt new processes and organizational changes, along with function-centric building blocks for higher autonomy. As intelligent agents evolve, network autonomy increasingly relies on deploying these agents, enabling self-management, self-healing, and adaptation with minimal human intervention. These agents facilitate the transition toward fully autonomous network operations across multiple layers, including services and resources. However, achieving Level 4 autonomy, where networks independently handle tasks across various domains, presents significant challenges, notably secure, efficient, and accurate multi-agent interactions. This document introduces a protocol-agnostic data model that enables multiple intelligent agents to communicate effectively using the model, ensuring end-to-end task execution and closed-loop operations in autonomous networks.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 January 2026.

Table of Contents

1. Introduction

One of the main drivers for automation in communications service providers' (CSPs') networks is the urgent need to reduce complexity so that they can lower operating costs. Using artificial intelligence (AI) and machine learning, CSPs are aiming to fully automate the lifecycle of the services they deliver to end customers and of internal network services, which are chains of technical components. The idea is to abstract the network as a set of software services and then use intent, automated closed control loops, and machine learning to make networks and operations self-configuring, self-optimizing, self-organizing, self-healing, and self-evolving. TMfourm defines the development of an AN network as five levels. The fourth level requires that the system be used to process network services in four dimensions: perception, analysis, decision-making, and execution. [TMF IG1230]

With the advancement of LLM and intelligent agent technologies, the development of Autonomous Networks (AN) increasingly depends on deploying intelligent agents. However, unlike general-purpose intelligent agents, AN has distinct characteristics. Firstly, the application scenarios are relatively fixed, focusing on the automation and intelligentization of network management services. For example, TMF has defined 20 high-value scenarios in this area. Secondly, AN follows a logical framework of single-domain autonomy and cross-domain collaboration. It also encompasses multi-task collaboration and supports both proactive (self-managed) and reactive (complaint and trouble ticket handling) interaction modes, these tasks and operations still involve some ambiguity and uncertainty. Thirdly, in the field of network operation and management, widely deployed protocols such as Netconf and RestConf are in use. This requires AN agents to communicate in a way that ensures both the accuracy and efficiency of structured data for deterministic tasks, and also supports natural language interactions for understanding and handling uncertain or ambiguous tasks. Additionally, compatibility with existing network protocols is essential for seamless integration and operation.

This document presents a protocol independent data model that enables multiple intelligent agents to communicate and interact effectively. This facilitates collaboration among agents, allowing them to work together seamlessly to accomplish autonomous network (AN) tasks. The proposed data model ensures flexible, accurate, and efficient data exchange, supporting coordinated decision-making and execution within the autonomous network ecosystem.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Scenario’ s description

2.1. Use Case 1: Fault management for large-scale batch out of service of mobile network base stations.

In this scenario, the fault management process requires coordination between different domains, such as the wireless network and the transport network. The operation typically takes over one hour to complete and is prone to triggering fault escalations due to the complex, cross-domain interactions and potential network disruptions during the shutdown process. To address this issue, TMForum has defined [IG1501], a standard solution package designed to handle autonomous network fault management processes across cross-operation and maintenance management domains. This framework facilitates coordinated and efficient fault detection, diagnosis, and resolution, helping to minimize operational risks and downtime during large-scale batch out of service activities. The AN L4 fault management flow for this scenario is excerpted as follows:

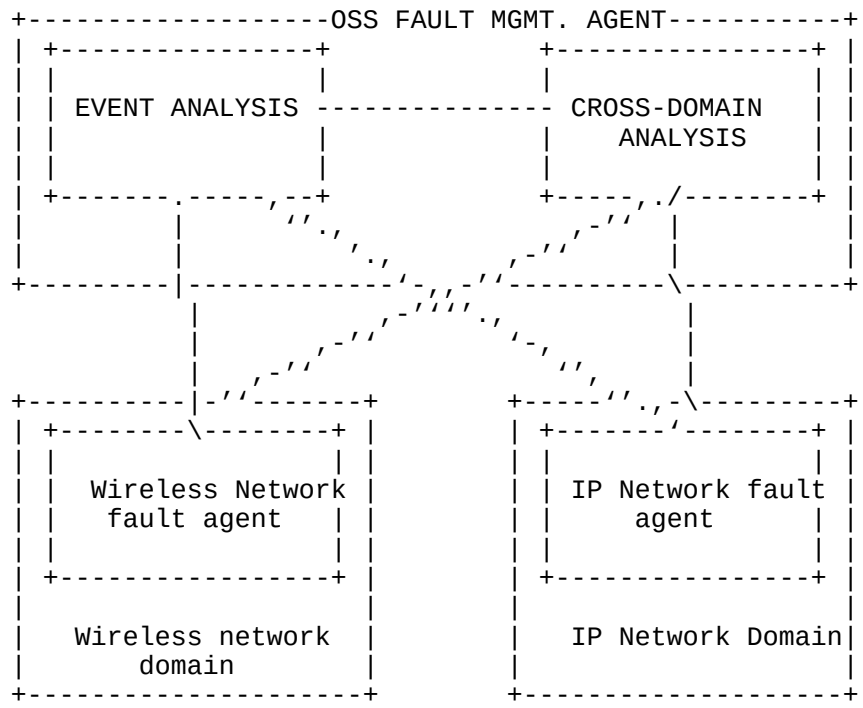


Figure 1: Work flow of AN L4 Cross Domain fault management

Step 1-2: represent the fault occurrence phase. As above example, in the IP network AN domain, a line interruption fault occurs. In the mobile network AN domain, a mass base station outage fault occurs.

Steps 3-4 represent the fault awareness phase. For example, in the above figure, the service layer fault agent receives various types of information, including (3a) incident actively sent by the mobile network agent, (3b) incident actively sent by the IP network agent, and (3c) user complaints. The service layer cross-domain agent performs cross-domain correlation analysis on this information (4).

Step 5 is the analysis and decision phase of fault management. In the above example, the service layer cross-domain identifies the fault as a cross-domain fault and determines the site where the fault occurred and the return link. It then sends diagnostic information to the corresponding resource layer fault agent (IP network fault agent). The resource layer fault agent performs self-diagnosis and returns the fault information and handling recommendations to the cross-domain agent.

Steps 6-8 represent the execution and repair phase of fault management. In the above example, the cross-domain agent receives fault information and repair suggestions sent by the resource layer agent and determines that automatic repair is not possible, prompting the initiation of a trouble ticket system to assign the repair trouble ticket to the FME (6). The FME interacts with the service layer's FME Copilot through an app to query relevant information about the faulty device (7). The service layer FME Copilot collaborates with the resource layer FME Copilot, which provides the FME with natural language-based operational guidance to assist in completing the fault repair (8).

In this scenario, the fault types and interaction processes are typically deterministic and qualitative. Utilizing structured language for communication between agents enhances reliability and efficiency.

2.2. Fault, experience, and energy-saving agents collaborate across multiple scenarios and service flows in an interconnected and coordinated manner

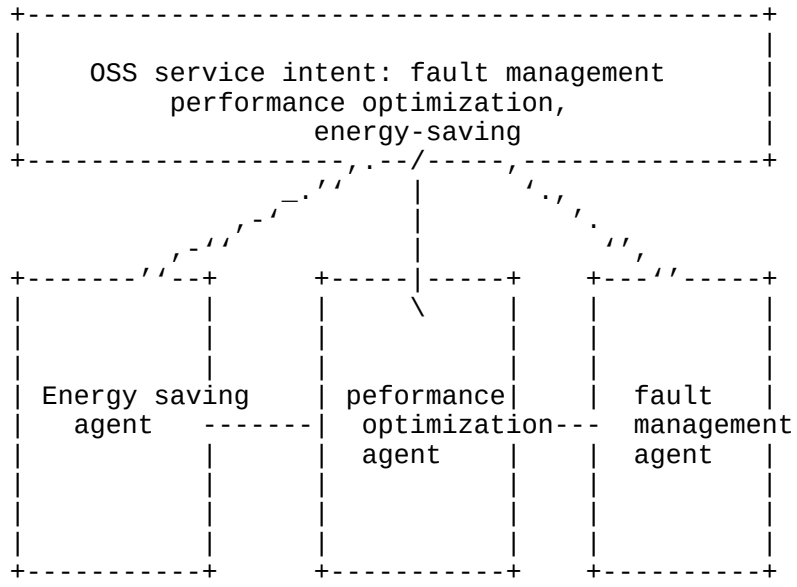


Figure 2: Multi-service agent collaboration t

Currently, fault management, performance optimization, and energy-saving workflows are operated separately, with each stream managed independently. The fault management, performance optimization, and energy-saving processes are not interconnected, leading to a lack of coordination. When a base station encounters a fault, manual intervention is required to optimize service performance for local coverage restoration and to adjust energy-saving strategies, in order to ensure a consistent user experience. This disconnected approach can cause delays and inefficiencies in resolving issues and maintaining service quality. To address this issue, in AN Level 4, the fault management, experience optimization, and energy-saving agents can autonomously collaborate and interact without human intervention. The process begins with negotiating to compensate user experience and adjust energy-saving strategies before locating the fault, thereby ensuring a seamless user experience. The workflow may proceed as follows:

The mobile network fault agent initiates collaboration with the mobile network optimization agent, prompting the optimization agent to adjust neighboring cell base station power to compensate for user experience.

The experience optimization agent collaborates with the energy-saving agent to reduce energy-saving targets or temporarily disable energy-saving functions.

The mobile network fault agent starts diagnosis by checking configurations and other parameters, then initiates a diagnostic collaboration request. The IP network fault agent conducts the analysis and returns diagnostic results. The mobile network agent performs checks and proceeds with fault repair, such as restoring the base station.

Once the issue is resolved, mobile network fault agent initiates a final collaboration, instructing the optimization agent to cancel the compensation, while the energy-saving agent resumes normal energy-saving operations.

In this scenario, multi-agent systems need to understand various tasks; however, it is impossible to exhaustively enumerate all arrangements and relationships among different operations. Understanding these tasks involves natural language processing, while executing specific tasks requires structured language, representing an intermediate state between unstructured comprehension and formal execution.

2.3. User Complaints Handling

Customer complaints are usually expressed directly in natural language, requiring intelligent agents to parse, analyze, and respond to them. A typical scenario involves complaints related to the Internet of Vehicle. The Internet of Vehicles (IoV) is an interactive wireless network built based on vehicle information such as location, speed, and routes. By utilizing devices like GPS, RFID, sensors, and camera image processing, vehicle networking enables the collection of environmental and status information of the vehicles themselves. Subsequently, through the Internet and computer technologies, this information is analyzed and processed to determine the optimal routes for different vehicles, provide real-time traffic and weather updates, and coordinate traffic signal cycles. Ultimately, it achieves organic interaction between cars, roads, and people, realizing intelligent transportation and vehicle systems. It may concern wireless communication, data transmission, cloud verification, and integration with upper-layer systems. The complaint handling interface utilizes trouble tickets information described in natural language. The workflow may proceed as follows:

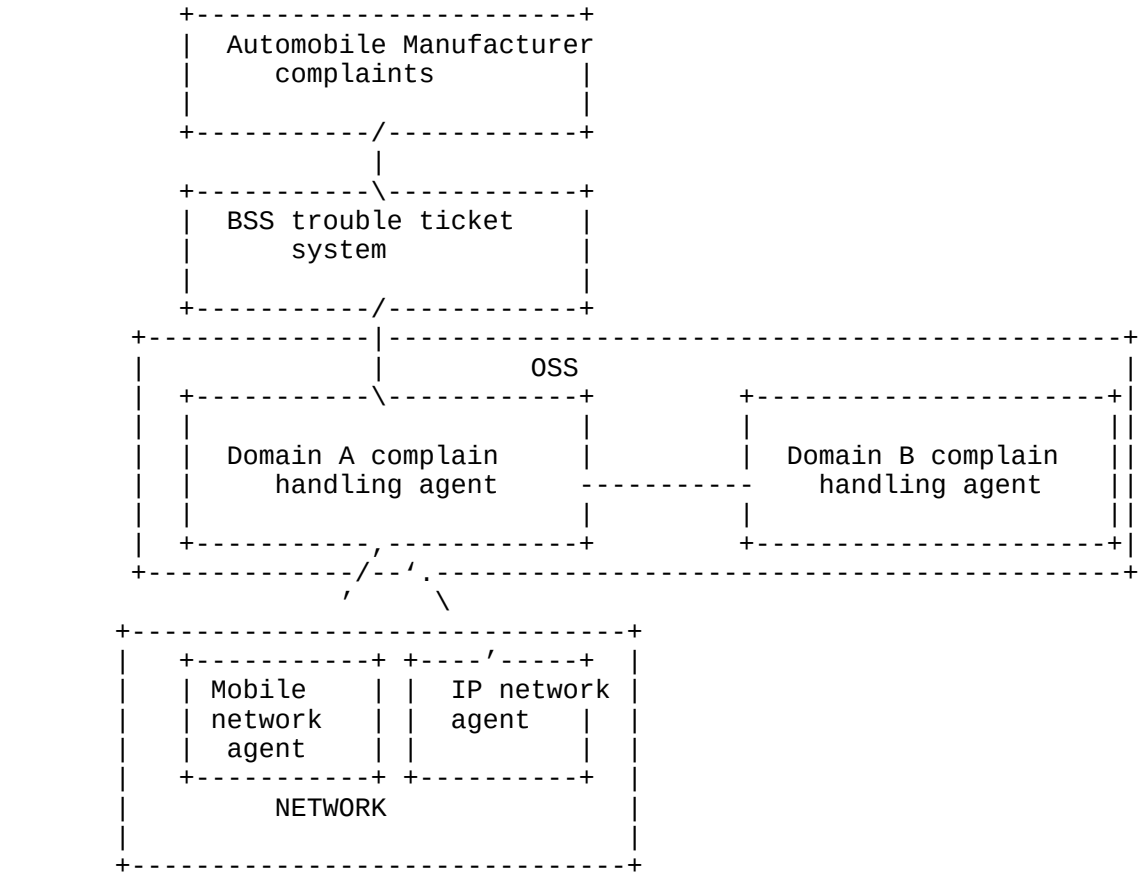


Figure 3: IOV User Complaints Handling

In this scenario, automotive companies centrally collect complaints from their customers (drivers) and use the operator' s complaint system to feedback issues to the operator. The operator's BSS trouble ticket system generates tickets from these complaints and dispatches them to the OSS. The integrated vehicle networking complaint handling agent within the OSS analyzes the trouble tickets and performs boundary localization.

Typically, boundary localization involves two situations: one within the current management and maintenance domain, such as within the same province or city; and the other across management and maintenance domains, such as when a vehicle moves to another province or city at a certain time and encounters an issue.

For the first case, the vehicle networking trouble ticket agent within the OSS will parse the ticket into multiple sub-tickets and interact with the transmission network agent and mobile network agent within its management domain to resolve the problem.

For the second case, the management and maintenance domain A’s agent analyzes the ticket and forwards the relevant information pertaining to domain B to the agent responsible for domain B. Upon receiving the trouble ticket, the domain B agent follows a process similar to that of the first situation to address the issue.

Since the content of complaints from automotive companies is usually described in natural language, the interactions of the intelligent agents in the above scenario are triggered by natural language inputs. And because of complaint content is unpredictable and the involved domains cannot be anticipated, it represents an uncertain environment. Natural language interaction is employed to effectively address and manage these ambiguous tasks.

3. Architecture

The prominent characteristic of these scenarios, which was introduced in section 2, is that the implementation of AN Level depends on the deployment of agents and the interaction among multiple agents. In response to this, TMF [IG1251C] has defined a reference architecture for AN Level 4, providing guidance on how to structure and coordinate multi-agent systems to achieve the desired levels of performance and automation. The architecture is excerpted as follows:

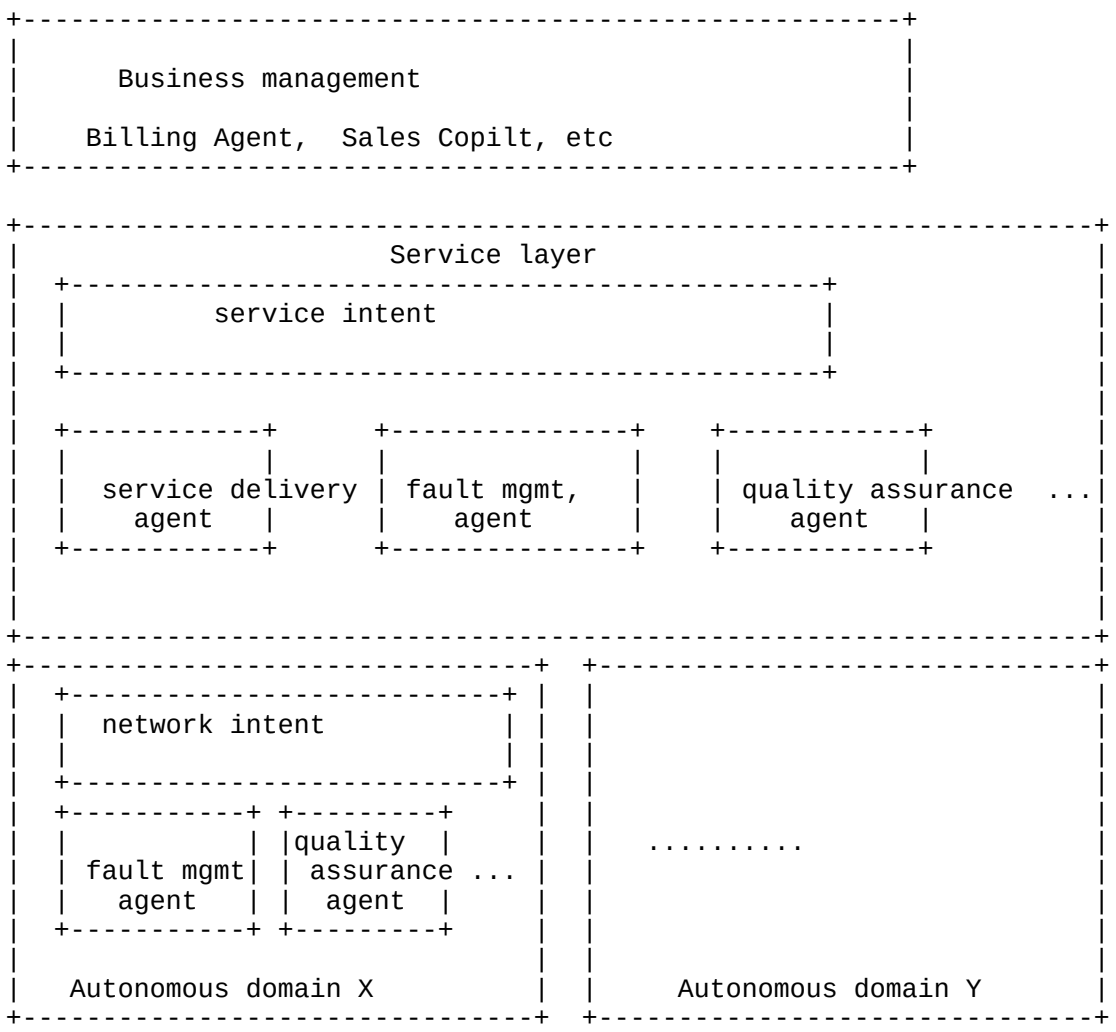


Figure 4: AN Agentic Architecture

Functional capabilities define what a network must do; agentic realization defines how it is done autonomously. Agentic realization transforms abstract functional requirements into real-time, adaptive behaviors through intelligent agents and copilots. Each agent continuously monitors its environment, makes decisions using AI-driven logic, and acts autonomously within defined operational domains. These agents operate across Business, Service, and Network layers, each guided by its own Operations Intelligence block. These blocks embed AI inference, govern agent behavior, and align decisions with domain-specific objectives. Copilots, meanwhile, assist human roles with context-aware insights and automated task execution.

In Business Operations, agents automate customer interactions, sales processes, and billing. In Service Operations, agents manage provisioning, assurance, and quality supervision. At the Resource level, agents optimize configurations, diagnose faults, and tune performance parameters dynamically.

Based on the above architecture, we can observe that the AN Level 4 network inherently involves multi-agent interactions during the autonomous closed-loop processing of tasks. For example, the interactions between agents at the business layer, service layer, and resource layer within the architecture are integral. Additionally, within the same layer, such as the resource layer, interactions occur across multiple domains or functions.

4. Functional Design

To achieve efficient interaction among agents within the AN network, this section discusses the fundamental functionalities required for AN agent interactions.

4.1. Agent register, discovery, and capability negotiation

As introduced in Chapter 2, these tasks involve interaction and cooperation among multiple agents. This raises an important issue: how do upper-layer or peer-layer agents identify which agents they should collaborate with, determine where to find these agents, and understand what capabilities these potential partners can offer to jointly complete a task? To address this, key capabilities such as agent capability description, registration and discovery, capability negotiation, and task-oriented agent team formation are essential. These functionalities facilitate efficient and effective multi-agent collaboration, ensuring that the involved agents can coordinate seamlessly to accomplish complex tasks.

4.2. Task dispatch

After completing the registration, discovery, and capability negotiation processes, a task typically involves activating one or multiple agents to carry out task allocation. Task allocation generally comprises two main parts: firstly, the description of the task and the expected outcomes; secondly, the feedback from task processing. The feedback can be categorized into two scenarios: synchronous feedback and asynchronous feedback. Synchronous feedback refers to tasks, usually with relatively short execution processes, where the agent handling the task provides feedback within the same session. In contrast, asynchronous feedback applies to tasks with longer, more complex execution processes. In this case, the task assigner subscribes to the task execution agents, awaiting updates on task status changes, such as completion or inability to complete. The executing agents then push the results and status updates back to the task publisher accordingly.

4.3. Multiple rounds of negotiation of the task

After the task is published to the agents, the task publisher should be able to engage in multi-round negotiations with the task recipients. This collaborative dialogue allows both parties to better understand the task requirements, clarify any uncertainties, and collaboratively refine the task execution plan. Such iterative negotiations help ensure that the task is comprehensively understood and effectively carried out, ultimately leading to improved task success and optimal outcomes.

4.4. Support structured and unstructured interaction of tasks

Unlike general multi-agent interactions that may serve a wide range of activities in the natural world, multi-agent interactions in autonomous network management have several distinct characteristics. Firstly, these scenarios are relatively well-defined and focused, dedicated solely to network management and operation tasks. For example, the TMF has identified twenty high-value scenarios specifically prioritized for AN Level 4 development. Under conditions where the scenarios and tasks are clearly defined, structured language enables more effective and accurate interaction between agents. Secondly, interactions among agents in autonomous network management often involve ambiguous or fuzzy tasks. This includes coordinating across multiple domains and dealing with multi-task collaboration, as well as handling user-initiated complaints that are typically expressed in natural language. In such cases, network agents need to understand user needs expressed in natural language and translate them into actionable tasks. Therefore, multi-agent interactions in autonomous networks require an “interaction

language” capable of balancing the ambiguity and expressiveness of natural language with the precision of structured languages. This enables agents to communicate efficiently and accurately, ensuring effective execution of complex, often fuzzy, tasks.

4.5. Task execution result negotiation

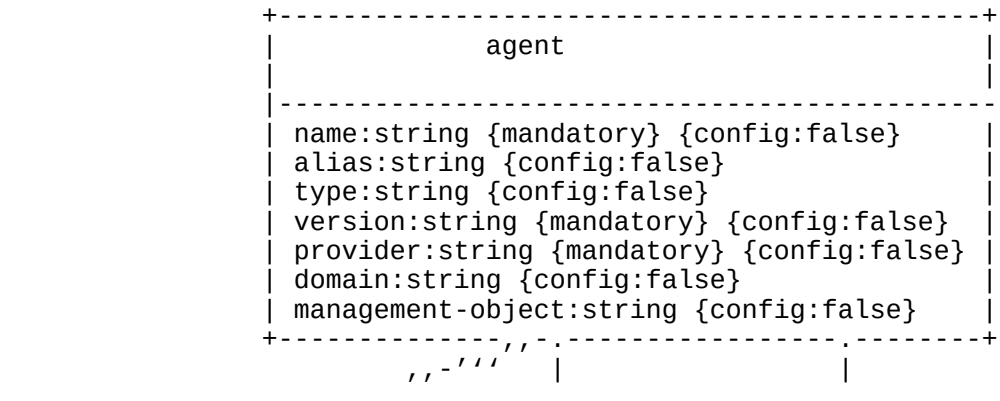
As described above, the multi-agent interactions in AN Level 4 networks are centered around tasks, which serve as the main framework for task assignment, interaction, and execution. The outcomes of these tasks require bidirectional validation, where both the task publisher and the task executor collaborate to confirm the results. Specifically, the task publisher evaluates the execution results reported by the task executor, verifying whether the outcomes meet the expected objectives. This process ensures accuracy and reliability in the network’s management and operation.

5. Model design

To achieve the aforementioned functionality, this chapter designs a protocol-independent information model for multi-agent interactions.

5.1. Agent profile

Agent profile description information used in the agent registration and capability negotiation. It should described the Agent’ s details such as agent description, manufacturer, capability set, version, management domain, and authentication information, etc. The following tree structure described the Agent profile model design information:



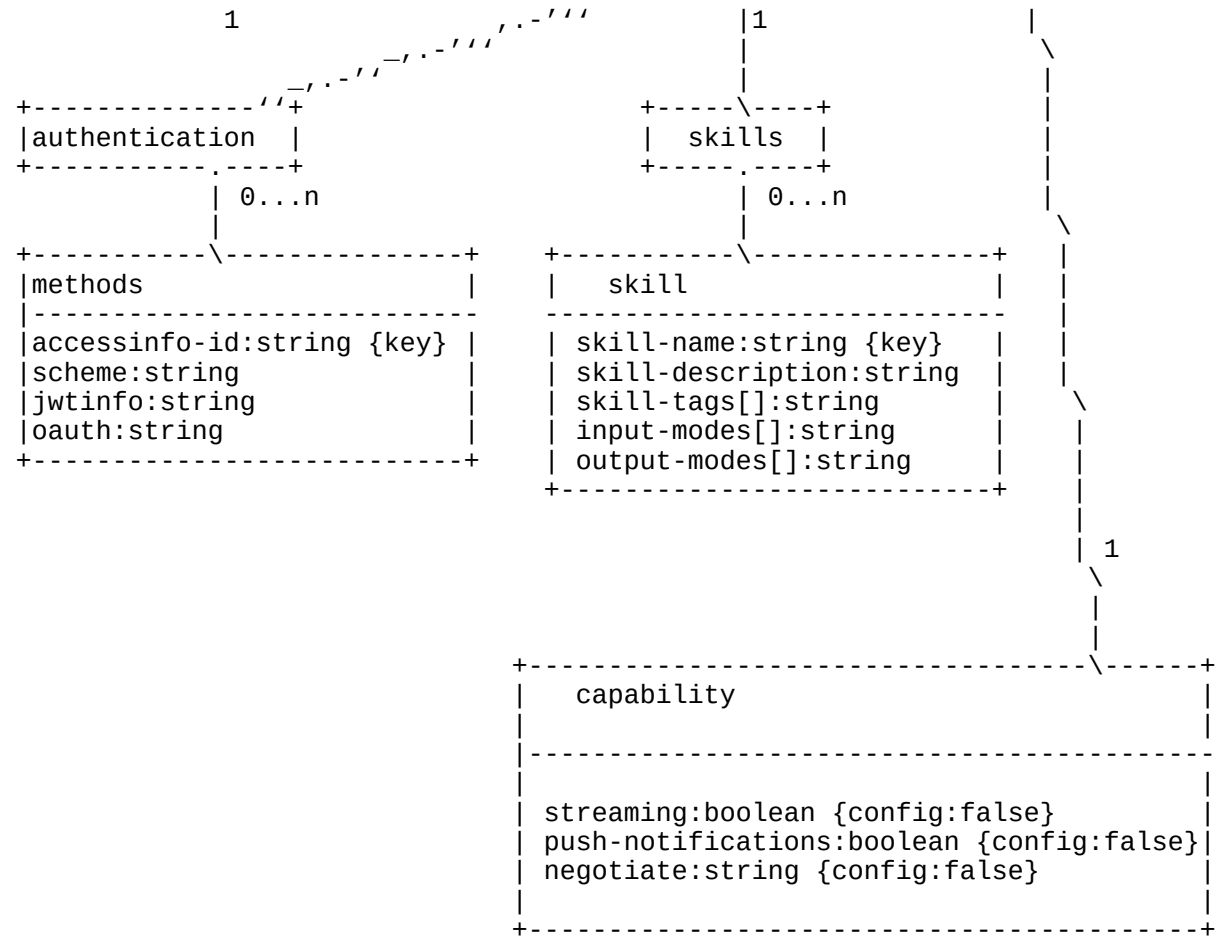


Figure 5: UML Screenshot for agent profile

Since this model is used to describe the objective capabilities of the agent, thereby supporting agent registration, discovery, capability negotiation, and capability querying, this module should be read-only. The key node definitions and their usage are as follows:

* Name: Agent name, which should remain unique within the namespace, is recommended to be defined as follows: {Standard Organization}-{Provider}-{Transmit}-{Version Number}.

* Skills: A list to describe the Agent’s business skill capabilities, that including skill name, skill description, supported input and output types, and examples.

* Input-models: a list within the skills list, it is used to describe the input types supported by the agent, such as image, text, audio, or video.

* output-models: a list within the skills list, it is used to describe the output types returned after the agent’s execution, such as image, text, audio, or video.

* Capability: The capability container is used to describe whether the agent supports specific interaction mechanisms. It includes the following leaf nodes:

** streaming, which indicates support for streaming transmission.

** pushNotifications, which indicates support for subscription reporting mechanisms.

** negotiate, which indicates support for capability negotiation mechanisms.

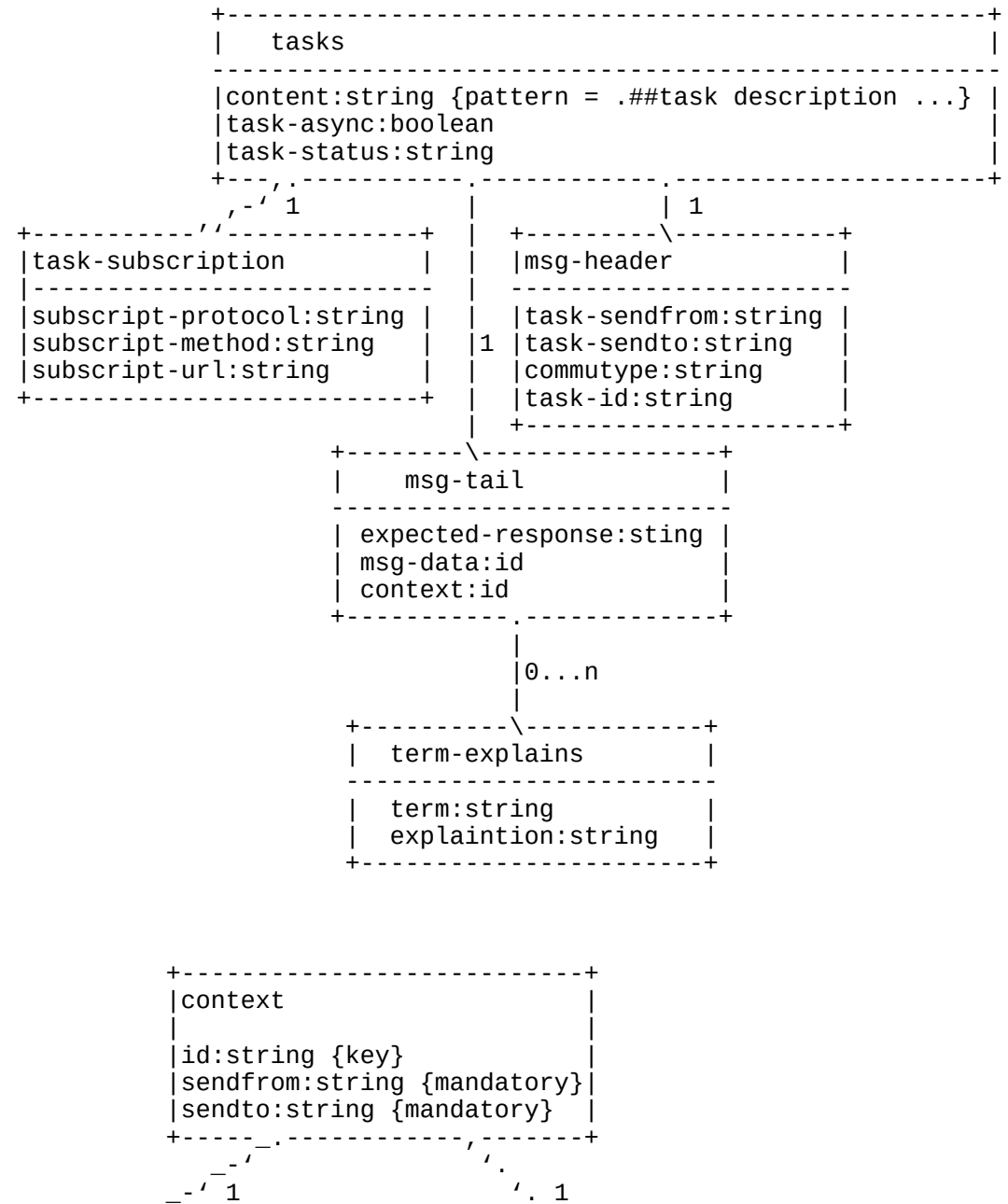
* authentication-methods: a list used to describe the authentication methods supported by the agent. In addition to the necessary ID, this list includes the following parameters: 1) scheme: the authentication scheme, which can be JWT, OAuth 2.0, API Key, etc.; 2) jwtInfo: parameters required for JWT authentication; 3) OAuth: parameters required for OAuth 2.0 authentication.

* In addition to the aforementioned modeling, the agent profile should also describe the software version of the agent, the manufacturer’s information, and the domains it can serve, such as Wireless, Transmit, Core Network domains defined in the AN. These details are provided and described by ‘version’ leaf, ‘provider’ leaf, and ‘domain’ leaf.

5.2. Task

As described above, in the AN, interactions between agents are all extended through tasks. Therefore, this chapter introduces how to model tasks. Taking a generic task as an example, a task is usually initiated by the initiating agent, which sends an instantiated task to the receiving or executing agent. The receiving or executing agent performs the task operations and returns the execution results to the initiating agent. To facilitate efficient interaction and to help both agents understand the specific task, the sending agent typically provides basic information about the task, such as the task type, description, expected outcome, and terminology explanations. Additionally, it should include prompt-like information similar to natural language interactions, which enables the initiating agent and

the receiving/executing agent to negotiate and discuss any issues that arise during the specific execution process of the task. The following UML described the Agent task model design information:



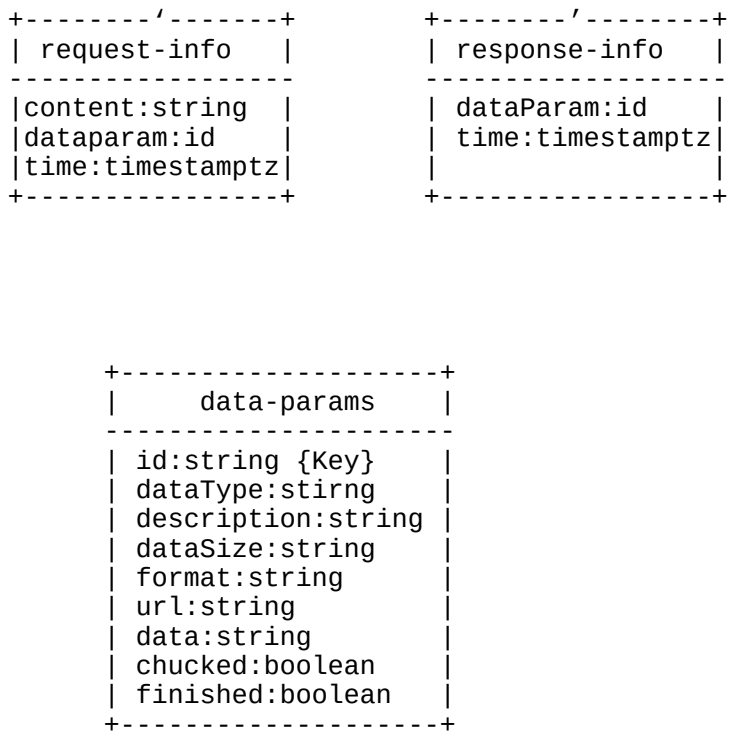


Figure 6: UML for AN task

In this model, the leaf nodes task-sendFrom and task-sendTo are used to describe the sending agent and the receiving/executing agent respectively. The commuType leaf node is used to specify the actual operation type of the task, such as task assignment, cancellation, status query, or status subscription, etc.

"The content node within the task should be able to provide prompt-like information. The current conventional approach is to use ## to indicate that the following content belongs to the prompt. A typical example is as follows:"

```
    <content namespace=\"prompt\">
##target object
<object">xxx</object">
##task description
<description">xxx</description">
##environmental infomation
<environmentInput">xxx</environmentInput">
```

Figure 7: content example

At the same time, this model supports a subscription mode for tasks, allowing users to subscribe to the status of a task and receive real-time updates on its execution. This subscription mode is particularly effective for tasks that require long-term or continuous execution.

In AN agent interactions, tasks are classified into synchronous and asynchronous tasks. For synchronous tasks, the receiving agent returns the task status in the response to the task assignment interface. For asynchronous tasks, the receiving agent pushes task status updates to the client whenever there are changes, providing real-time progress and task results. The corresponding modes are indicated in task modeling using the leaf node task-async, while the task status is stored in task-status.

In addition, the expected results required for task interaction are described using the expected-response leaf node in this model. Term explanations are provided using the term-explains list. The data exchanged between the initiating agent and the receiving agent is stored in data-param, which can include alerts, warnings, case knowledge, and other information. These data types can be files, structured data, text, and more.

Additionally, this model provides context retrieval capabilities by recording all agent communication records in list form, which can prevent agents from losing the original objectives or forgetting historical information, thereby maintaining the coherence of the agents' collaborative behavior. This capability is described in the model using the context element.

5.3. Task negotiation

As described above, agents should be allowed to engage in multi-turn interactions and dialogues regarding the same task. The involved processes may include:

1) Before one agent assigns a task to another, the former requests the latter to conduct a feasibility check to determine whether the task can be accomplished. If not, both agents negotiate to arrive at a feasible alternative task.

2) When one agent assigns a task to another, the former requires the latter to actively initiate negotiation during the execution process. The latter summarizes and generates outputs (such as proposed decisions) and requests the former to evaluate and negotiate. If the outcomes do not meet the requirements, the two agents continue negotiating to produce a result that satisfies the conditions.

In the model defined in this draft, multi-turn interactions for tasks are described using RPC, and its hierarchical structure is depicted as follows:

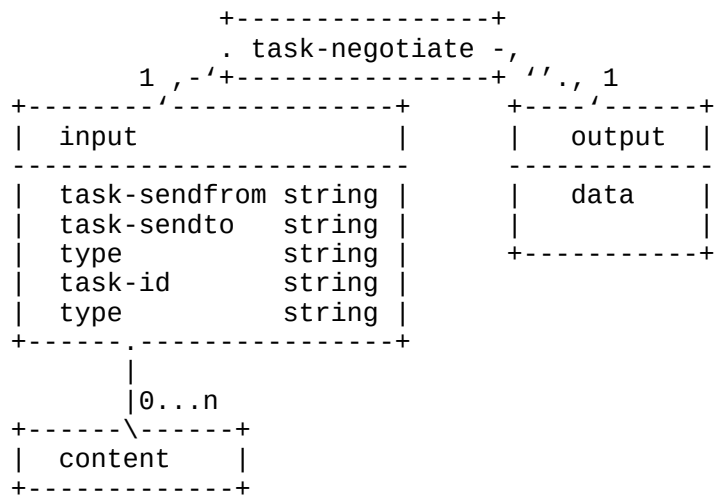


Figure 8: UML for task negotation

5.4. Task notification

As described above, agent interactions should support a task status notification mechanism, where the receiving agent pushes TaskStatus updates to the sending agent, providing real-time progress, task status, and execution results. The triggering conditions for such notifications can include any change in the task' s state, such as transitioning from create to processing, from processing to blocking, or from processing to failed. These events proactively trigger the task status notification interface. The hierarchical structure of the task notification module defined in this paper is illustrated in the diagram below.

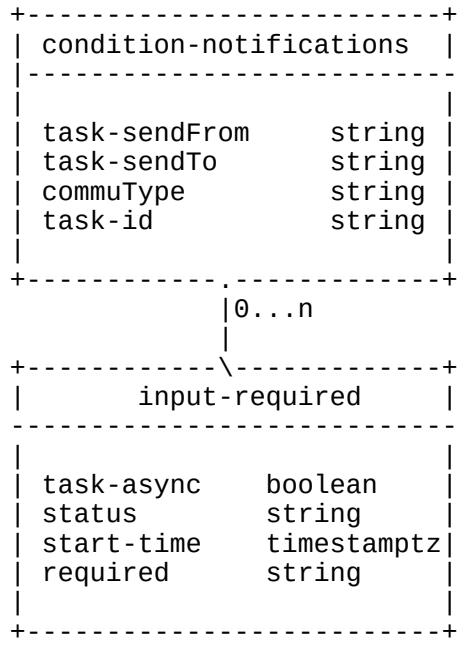


Figure 9: UML for task notification

6. IANA Considerations

This memo includes no request to IANA.

7. Security Considerations

This document should not affect the security of the Internet.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

8.2. Informative References

[IG1251C] TM Forum, "IG1251 Autonomous Networks Reference Architecture v1.0.0", 2025, <https://www.tmforum.org/resources/introductory-guide/ig1251-autonomous-networks-reference-architecture-v1-0-0/>.

[IG1501] TM Forum, "IG1501 AN L4 Fault Management Solution Package v1.0.0", 2025, <https://www.tmforum.org/resources/guidebook/ig1501-an-l4-fault-management-solution-package-v1-0-0/>.

Appendix A. appendix: YANG DATA Model

This section models the agent discussed in this document following the YANG data modeling approach adopted by the IETF. The provided code is for reference only.

<CODE BEGINS> file "irtf-agent@2025-07-01.yang"

```
module irtf-agent {
  namespace "urn:ietf:params:xml:ns:yang:irtf-agent";
  prefix agtask;

  import agent-message {
    prefix agtmsg;
  }



  organization "Example Organization";
  contact "support@example.com";
  description "header YANG Model";

  revision 2025-06-01 {
    description
      "Initial revision";
    reference
      "RFC 6241: Network Configuration Protocol";
  }


  typedef timestamptz {
    type string {
      pattern '\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(\.\d+)?'
            + '(Z|[\+\-]\d{2}:\d{2})';
    }
  }
```

```
  typedef uuid {
    type string {
      pattern '\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(\.\d+)?'
            + '(Z|[\+\-]\d{2}:\d{2})';
    }
  }

  typedef task-status {
        type enumeration {
           enum finished {
                 description
                   "The task is executed.";
           }
           enum blocked {
                 description
                   "The task is blocked.";
           }
         }
  }

  typedef datatype-enum {
        type enumeration {
           enum text {
                 description
                   "The request requires a resource that already is in use.";
           }
           enum image {
                 description
                   "The request requires a resource that already is in use.";
           }
           enum audio {
                 description
                   "The request requires a resource that already is in use.";
           }
           enum video {
                 description
                   "The request requires a resource that already is in use.";
           }
         }
  }

  container agent {
    config false;
    description "Agent configuration";

    leaf name {
      type string;
          config false;
```

```
      description "Agent name";
      mandatory true;
    }

    leaf alias {
      type string;
          config false;
      description "Agent alias";
    }

    leaf description {
      type string;
          config false;
      description "Agent description";
      mandatory true;
    }

    leaf type {
      type string;
          config false;
      description "Agent type";
    }
    container skills {
      description "abilities";
          config false;

      list skill {
            key "skill-name";
        leaf skill-name {
          type string;
          description "agent skill name";
        }
        leaf skill-description {
          type string;
          description "agent skill description";
        }
              leaf-list skill-tags {
          type string;
          description "agent skill tags";
        }
              leaf-list input-modes {
          type string;
          description "agent skill input-mode: image, text, audio, video";
        }
              leaf-list output-modes {
          type string;
          description "agent skill output-mode: image, text, audio, video";
        }
```

```
      }
    }

    leaf version {
      type string;
          config false;
      description "Agent version, comply sematics version 2.0";
      mandatory true;
    }

    leaf provider {
      type string;
          config false;
      description "vendor";
      mandatory true;
    }

    leaf domain {
      type string;
          config false;
      description "supported domain, such as: Wireless, Optical, IP, Core Network";
    }

    container capability {
      description "agent support capabilities";
          leaf streaming {
            type boolean;
                config false;
            description "if support streaming";
          }
          leaf push-notifications {
            type boolean;
                config false;
            description "if support streaming notification";
          }
          leaf negotiate {
            type string;
                config false;
            description "supported negotiate type, such as: acp-task-feasibility-negotiat
ion-v1.0,
acp-task-fulfillment-negotiation-v1.0";
          }
    }

    leaf management-object {
      type string;
          config false;
      description "Agent management domain, such as:xx subnet";
    }
```

```
    /*leaf location {
      type string;
      description "Agent belongs to";
    }*/

    /*leaf-list llms {
      type string;
      description "LLMs Agent used";
    }*/

    container authentication {
      description "Agent authentication information";

      list authentication-methods {
            key "accessinfo-id";
        leaf accessinfo-id {
          type string;
          description "unique accessinfo id";
        }

                leaf scheme {
                  type string;
                  description "scheme authentification method";
                }
                leaf jwtInfo {
                  type string;
                  description "jwt authentification method";
                }
                leaf oauth {
                  type string;
                  description "scheme authentification method";
                }
      }
    }
  }

  list context {
    description "context configuration";
        key "id";
        leaf id {
           type string;
        }

    leaf sendFrom {
      type string;
      description "Agent name of sender";
      mandatory true;
    }
```

```
    leaf sendTo {
      type string;
      description "Agent name of receiver";
      mandatory true;
    }
    container request-info {
                description "request-info configuration";
                leaf content {
                  type string;
                  description "request message";
                  mandatory true;
                }

                leaf dataParam {
                  type leafref {
                        path "/data-params/id";
                  }
                  description "data information";
                }

                leaf time {
                  type timestamptz;
                  description "UTC time: e.g., 2025-04-21 14:19:00+0";
                  mandatory true;
                }
        }
    container response-info {
          description "response-info configuration";

          leaf dataParam {
            type leafref {
                        path "/data-params/id";
                  }
            description "data information";
            mandatory true;
          }

          leaf time {
            type timestamptz;
            description "UTC time: e.g., 2025-04-21 14:19:00+0";
            mandatory true;
          }
    }

  }

   container msg-base {
    uses msg-header;
```

```
        leaf content {
                type string;
        }
        uses msg-tail;
  }

  grouping msg-header {
    description "header configuration";

        leaf task-sendFrom {
          type string;
          description "the agent which send this task";
        }
        leaf task-sendTo {
          type string;
          description "the agent which recieve this task";
        }
        leaf commuType {
      type string;
      description "communication type, which related to task type
                CapabilityRequest: request capabilities of Agent
                taskAssgin: assign task by Agent consumer
                taskCancel: cancel task by Agent consumer
                FeasibilityNegotiate/FulfillmentNegotiate: Negotiate task
                taskResultQuery: query task result
                taskResultSubscribe: subscribe task result
                taskResultNotify:notify task result";
    }

    leaf task-id {
      type string;
      description "The task id.";
    }
   }

   grouping msg-tail {
    leaf expected-response {
          description "";
          type string;
        }
        list term-explains {
            key "term";
            leaf term {
                    type string;
                }
                leaf explaination {
                    type string;
                }
```

```
        }
        leaf msg-data {
      type leafref {
             path "/data-params/id";
         }
      description "msg convey data.";
    }
    leaf context {
                description "history information";
                type leafref {
                 path "/context/id";
        }
        }
  }

  list data-params {
    description "DataParams configuration";
        key "id";

        leaf id {
          type string;
        }
    leaf dataType {
      type datatype-enum;
      description "datatype??optional value: text, image, audio, video";
      mandatory true;
    }

    leaf description {
      type string;
      description "data description";
    }

    leaf dataSize {
      type string;
      description "data size";
    }

    leaf format {
      type string;
      description "data format
                dataType = text, value:text/plain, application/json, etc.
                dataType = image, value:jpg, png, etc.
                dataType = audio, value: mp3, WAV, etc.
                dataType = video, value:mp4,avi, etc.";
    }

    leaf url {
```

```
      type string;
      description "resource url, for large data";
    }

    leaf data {
      type string;
      description "data content";
    }

    leaf chunked {
      type boolean;
      description "if data is too large, it can be chunked to several pieces";
    }

    leaf finished {
      type boolean;
      description "last piece of chunked data: true";
    }
  }

  container tasks {

          uses agtmsg:msg-header;
          leaf content {
                description
                  "You are a task executor. please execute the task based on the target o
bject, task description, and environmental information. For example.";
                type string {
                    pattern '.*##task description\s*\S+.*';
                  }
                }
          container task-subscription {
                description "";
                leaf subscript-protocol {
                        type string;
                        }
                leaf subscript-method {
                        type string;
                        }
                leaf subscript-url {
                        type string;
                        }
                }
          leaf task-async {
                  type boolean;
                  description "";
          }

          leaf task-status {
```

```
                  type string;
          }
          uses agtmsg:msg-tail;
        }

  rpc task-negotiate {
    description
      "The task-negotiate ....";

    reference "RFC xxxx, Section 7.2";

    input {
          uses agtmsg:msg-header;
      leaf-list content {
        description "You are a task executor. please execute the task based on the target
 object, task description, and environmental information. For example,
                // <anydata>
                <content namespace=\"prompt\">
                ##target object
                <object>xxx</object>
                ##task description
                <description>xxx</description>
                ##environmental infomation
                <environmentInput>xxx</environmentInput>";
        type string {
           pattern '.*## target object\s*\S+.*';
           pattern '.*## task description\s*\S+.*';
                }
      }

          leaf type {
                type enumeration {
          enum FEASIBILITY {
            value 1;
            description
              "check if task is feasibility.";
          }
          enum FULFILLMENT {
            value 2;
            description
              "check if task can be fulfillment.";
          }
                }
          }
        }
    output {
      anyxml data {
        description
          "Copy of the source datastore subset that matched
```

```
            the filter criteria (if any).  An empty data container
            indicates that the request did not produce any results.";
        }
      }
    }
   notification condition-notifications {
    description
      "When the tesk condition is met, this notification is sent.";

    uses msg-header;

    list input-required {
          key "required";

          leaf task-async {
                  type boolean;
                  description "";
          }
          leaf status {
                  type string;
                  description
                        "Indicate the status of the task, such as suspended, finished, et
c.";
          }
          leaf start-time {
            type timestamptz;
            description "UTC time: e.g., 2025-04-21 14:19:00+0";
            mandatory true;
          }
          leaf required
          {
        type string {
           pattern '.*## target object\s*\S+.*';
           pattern '.*## task description\s*\S+.*';
                   }
                 }
          }
   }
   anyxml data {
    description
    "provide another information.";
    }

}
```

Figure 10

<CODE ENDS>

Acknowledgements

The authors of this document would like to thank Yijun Yu, Fei Guo, Jinjin Chen, and others for their substantive review and comments, and proposals to stabilize and improve the document.

Contributors

The authors would like to thank Yijun Yu, Fei Guo, Jinjin Chen for their major contributions to the initial modeling and use cases.

Authors' Addresses

Michael Wang
Huawei Technologies, Co., Ltd
101 Software Avenue, Yuhua District
Nanjing
210012
China
Email: wangzitao@huawei.com

Yuan Xie
Huawei Technologies, Co., Ltd
101 Software Avenue, Yuhua District
Nanjing
210012
China
Email: yuan.xie@huawei.com

Qin Wu
Huawei Technologies, Co., Ltd
101 Software Avenue, Yuhua District
Nanjing
210012
China
Email: bill.wu@huawei.com