



# MOSAICO

**D1.1: Comparison of protocol design approaches**  
September, 2025



Funded by  
the European Union

Funded by the European Union under the Grant Agreement No 101189664. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HADEA). Neither the European Union nor the granting authority can be held responsible for them.

## D1.1 Comparison of protocol design approaches

Work package	WP1 AI-agent server protocol
Task	T1.1 API design and evaluation
Due date	30/09/2025
Submission date	26/09/2025
Type of deliverable	Report
Dissemination Level	Public
Deliverable lead	University of York
Version	1.0
Authors	University of York
Reviewers	UDA, IMT, LIST, QODO
Keywords	Network protocols, agent-to-agent communication, multi-agent systems

### Document Revision History

Version	Date	Description of change	List of contributor(s)
v0.1.0	07/07/2025	Initial population of template	University of York
v0.2.0	01/08/2025	Interim version before internal review	University of York
v0.3.0	31/08/2025	First version for internal review	University of York
v0.4.0	09/09/2025	Revision from LIST feedback	University of York
v0.5.0	15/09/2025	Revision from UDA feedback	University of York
v1.0.0	19/09/2025	Expanded MCP content	University of York

## Partners



Netcompany



## Funding



Funded by  
the European Union

Call	Digital and emerging technologies for competitiveness and fit for the Green Deal (HORIZON-CL4-2024-DIGITAL-EMERGING-01)
Topic	HORIZON-CL4-2024-DIGITAL-EMERGING-01-22: Fundamentals of Software Engineering
Type of action	Research and Innovation Action (RIA)

## Disclaimer

Funded by the European Union under the Grant Agreement No 101189664. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HADEA). Neither the European Union nor the granting authority can be held responsible for them.

## Copyright Notice

© MOSAICO Consortium, 2025

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both. Reproduction is authorised provided the source is acknowledged.

## Executive Summary

This deliverable surveys the current state-of-the-art in communication protocols for agent collaborations, and evaluates their fit-for-purpose against the MOSAICO goals of using AI agents to solve software engineering tasks in a reliable and transparent way, promoting the development of high-quality software systems. The aim of this report is to decide on a technical direction to follow for implementing the Minimal Viable Product version of the reference MOSAICO client and server in Task 1.2.

The report specifies a set of requirements for MOSAICO AISP, including its expected functionality, and its architectural characteristics. An initial investigation of currently available options identified a number of protocols for agent collaboration, out of which the two protocols with the largest market penetration were included in the comparison: the Agent-to-Agent (A2A) protocol (currently governed by the Linux Foundation) and the Model Context Protocol (an open-source project from Anthropic). In parallel with this investigation, a new protocol was created from scratch within MOSAICO, formalised as a machine-readable OpenAPI specification: the aim was to investigate the minimal set of features needed for a protocol to suit MOSAICO's needs.

The protocols were compared in two ways: by reviewing their official specifications, and by using them to implement a multi-agent system that solved a software engineering task comprising a collaboration agent, a solution agent, and two supervision agents. The chosen task was to design an object-oriented data model of a given domain from a natural language description, and formalise it in a machine-readable language (Eclipse Emfatic). The solution agent was responsible for generating the Emfatic source code, and the supervision agents were responsible to check the syntactic and semantic validity of the generated code.

The results of the comparison showed that our new protocol met the most requirements (only having one partial and one unmet requirement). A2A was a very close second by only missing one more requirement (cross-collaboration observability) that could be solved through its support for third-party extensions. MCP was a distant third, requiring significant adaptations to meet MOSAICO's needs.

Neither A2A nor MCP met the requirement for enabling agent transparency, with A2A explicitly mentioning opaqueness as one of their design goals, to protect privacy and intellectual property. For this reason, we postponed the design of transparency-related features in our new protocol until the desired tradeoffs between transparency, privacy, and intellectual property protection were clarified through the development of the industrial use cases.

Similarly, we found that the agent descriptions offered by MCP and A2A were missing some of the desired information. This additional information could be included in A2A through its protocol extensibility mechanisms. As a prototype, our new protocol's agent descriptions are also missing some aspects (e.g., description of skills according to a taxonomy) that would be fleshed out throughout the development of the MOSAICO Agent Repository in WP2.

In conclusion, MOSAICO can either attempt to start a new community from scratch with our new protocol (competing against A2A), or instead collaborate with the A2A community, developing the extensions needed to meet MOSAICO's goals. In the interest of maximising the impact of the MOSAICO Platform, our current direction for further work in WP1 is to build the MOSAICO AISP as an extension of A2A which would inter-operate with third-party A2A agents with minimal adaptations and extensions. We anticipate combining A2A with MCP as well, as MCP is better suited to be used within specific agents that need to interact with existing information systems, whereas A2A can handle intercommunications with other agents.

## Table of Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
<b>2</b>	<b>Protocol requirements</b>	<b>10</b>
2.1	R1: Agent discoverability	11
2.2	R2: Multi-part and multi-modal messaging	12
2.3	R3: Multi-turn conversations	12
2.4	R4: Asynchronous and streaming communication	13
2.5	R5: Agent observability	13
2.6	R6: Agent transparency	13
2.7	R7: Interoperability	14
2.8	R8: Access control	14
2.9	R9: Bandwidth-efficient serialisation	15
<b>3</b>	<b>Protocol design and selection</b>	<b>15</b>
3.1	Design of a custom inter-agent protocol	15
3.1.1	Agent discovery	16
3.1.2	Thread management	17
3.2	Review of the state-of-the-art	19
3.2.1	Agent-to-Agent protocol (A2A)	20
3.2.2	Model Context Protocol (MCP)	22
<b>4</b>	<b>Comparison methodology</b>	<b>22</b>
4.1	Software engineering task to solve	23
4.2	Requirements evaluation	24
<b>5</b>	<b>Results</b>	<b>26</b>
5.1	Using the new protocol as a base	26
5.1.1	Implementation	26
5.1.2	Demonstration	27
5.1.3	Requirements evaluation	29
5.2	Using A2A as a base	31
5.2.1	Implementation	31
5.2.2	Demonstration	32
5.2.3	Requirements evaluation	35
5.3	Using MCP as a base	37
5.3.1	Implementation	37
5.3.2	Demonstration	38
5.3.3	Requirements evaluation	40
<b>6</b>	<b>Conclusion and next steps</b>	<b>42</b>
6.1	New protocol	42
6.2	A2A	42
6.3	MCP	43
6.4	Overall conclusion	44

**List of Figures**

1 Overview of the MOSAICO framework . . . . . 9

2 Screenshot of the Swagger-UI client operating on the OpenAPI specification of the custom protocol . . . . . 16

3 A2A task lifecycle . . . . . 21

4 MCP interaction between Host, Client, Server, and Resource . . . . . 22

5 Envisioned agent collaboration for the chosen software engineering task (object-oriented model generation) . . . . . 23

6 Screenshot of the Phoenix traces for the production of the second and final version of the metamodel by the new protocol solution agent . . . . . 29

7 Screenshot of the Phoenix traces for the semantic supervision of the first version of the generated metamodel (which was missing the title attribute in the Book class) . . . . . 30

8 Multi agent system with A2A implementation . . . . . 33

9 Screenshot of Langfuse observability of the A2A implementation . . . . . 34

10 Multi agent system with MCP implementation . . . . . 37

11 MCP inspector calling the Collaboration Agent . . . . . 39

12 Screenshot of Langfuse observability of the MCP implementation . . . . . 40

**List of Tables**

- 1 Requirements for the MOSAICO AI-agent Server Protocol (AISP) . . . . . 10
- 2 Recently proposed protocols for agent collaboration, ordered by date of first release. Items marked with \* are the protocols chosen for comparison against creating a new protocol. . . . . 19
- 3 Agents in the new protocol implementation . . . . . 27
- 4 Agents in the A2A implementation . . . . . 32
- 5 Agents in the MCP implementation . . . . . 38
- 6 Requirement evaluation summary: ✓ is met, × is not met, ~ is partially met. For R9, the amount of bandwidth used by each multi-agent collaboration is displayed. 44

## Abbreviations

A2A	Agent-to-Agent
AISP	AI-agent Server Protocol
BDI	Belief-Desire-Intention
CORS	Cross-Origin Resource Sharing
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LLM	Large Language Model
LSP	Language Server Protocol
MCP	Model Context Protocol
MIME	Multipurpose Internet Mail Extensions
RPC	Remote Procedure Call
SDK	Software Development Kit
SE	Software Engineering
SSE	Server-Sent Events
TLS	Transport Layer Security
UUID	Universally Unique Identifier

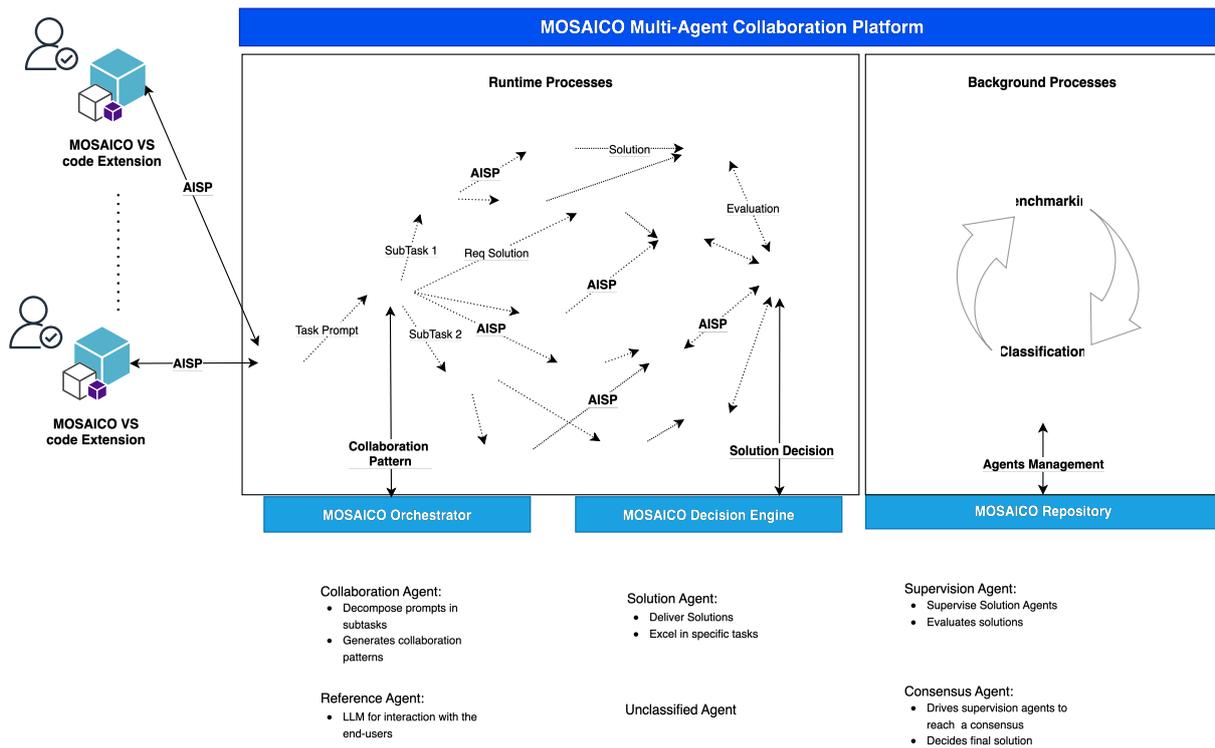


Figure 1: Overview of the MOSAICO framework

## 1 Introduction

The MOSAICO vision is to use AI agents to solve software engineering tasks in a reliable and transparent way, which promotes the development of high-quality software systems. It has been recognised in the literature that multi-agent collaborations where each agent is specialised on a given task can perform better than a single generalist agent (He et al., 2024). Likewise, it is a known problem that many Large Language Models (LLMs) are vulnerable to *task switches*, where the user changes the task to be performed in the middle of a long conversation (Gupta et al., 2024): it is more reliable to keep a conversation focused on a single task (as they would when separating a large task into subtasks to be given to dedicated agents).

Figure 1 shows an overview of the integrated solution that is to be provided by MOSAICO. The user communicates their task to a *reference agent* (in red) through their client (in this case, a Microsoft Visual Studio Code extension). The reference agent then discovers the most appropriate task by querying the *MOSAICO repository*: for complex tasks, this may be a *collaboration agent* (in green) which will orchestrate other MOSAICO agents to solve the task in a reliable way. These may be *solution agents* which propose solutions to the given task (in blue), *supervision agents* which evaluate the proposed solutions (in yellow), *consensus agents* (which decide the final solution among several), or even other collaboration agents.

This vision requires having a common mechanism to intercommunicate these specialised agents: a shared network protocol that the agents understand — the MOSAICO AI Agent Server Protocol (shown as “AISP” in Figure 1, and as all the interconnections between the agents and the VS Code extension). This deliverable aims to evaluate the available technical approaches to design and implement AISP: starting from a clean slate, or building it as an extension of industry-led protocols (such as Google’s A2A protocol). The evaluation will be done against a set of requirements elicited from reviewing the literature and the requirements

Identifier	Name	Priority
Agent functionality		
R1	Agent discoverability	SHALL
R2	Multi-part and multi-modal messaging	SHALL
R3	Multi-turn conversations	SHALL
R4	Asynchronous and streaming communication	SHALL
Architectural characteristics		
R5	Agent observability	SHALL
R6	Agent transparency	SHOULD
R7	Interoperability	SHALL
R8	Access control	SHALL
R9	Bandwidth-efficient serialisation	SHOULD

Table 1: Requirements for the MOSAICO AI-agent Server Protocol (AISP)

from the use case providers.

The rest of the document is structured as follows. Section 2 presents and justifies the various requirements for the protocol. Section 3 describes the new protocol designed during Task 1.1, and presents several state-of-the-art protocols to be used for comparison. Section 4 explains how the various technical approaches will be compared to each other. Section 5 describes the experimental results obtained from building a prototype multi-agent collaboration using each of the approaches. Finally, Section 6 will summarise our findings and set out the approach to be followed for developing AISP and its reference implementations throughout MOSAICO.

## 2 Protocol requirements

In order to enable the MOSAICO vision of enabling collaborations of specialised agents for reliable AI in software engineering tasks, we have identified a number of requirements that the MOSAICO AISP would need to meet. Table 1 lists the identified requirements: these cover both the desired functionality that agents must be able to provide, as well as any architectural characteristics required by the platform that the protocol should enable. Requirements with a SHALL priority are required, whereas requirements with a SHOULD priority are optional but desirable. The following sections will identify and describe in further detail each of these requirements.

## 2.1 R1: Agent discoverability

In order to join the MOSAICO platform, an agent will need to identify itself and describe its provenance, its functionality, and its correct form of use. In order to support transparency (R6) it may also need to reveal some of its internals. This implies that the protocol should have an endpoint that produces a machine-readable description which can be consumed by the MOSAICO Agent Repository. The description would include:

- **Identification:** name, description, and version (ideally based on Semantic Versioning<sup>1</sup>) for the agent.
- **Provenance:** developing organisation, website, and developer contact details.
- **Skills:** description of the software engineering tasks supported by the agent, ideally in relation to the taxonomy of SE tasks to be developed in WP2. These tasks may be related to producing a solution, supervising the solution proposed by another agent, orchestrating a collaboration of multiple agents, or reaching a consensus among solution agents.

*Note:* a MOSAICO collaboration agent focused on a specific SE tasks may be exposed to the rest of the MOSAICO platform as a solution agent for that task, in order to enable hierarchical decomposition of a complex SE task.

- **Message formats:** description of the expected format for the input and output messages for this agent. If the agent is designed to use multi-turn conversations, it should also describe the expected format of any interrupts. Ideally, this description should be machine-readable and provide enough detail to support code generation for the agent.
- **Protocol support:** description of optional protocol features that are supported by the agent. This is to support the creation of a spectrum of agents, from simpler agents specialised on a single task, to more advanced ones dedicated to complex tasks.  
*Note:* this is inspired by protocols such as the Microsoft Language Server Protocol (LSP), which allows both client and server to declare their support for various protocol features<sup>2</sup>.
- **Security aspects:** for cloud-hosted agents, which authentication schemes are required, and which authentication authority is involved.
- **Transparency support:** the description may include additional internal details about the agents. These may include references to the other MOSAICO agents it relies upon, the tools that it may require, or the underlying LLM model that is being used. The description may also list configuration options that may be used to customise the behaviour of the agent (e.g., enabling or disabling the use of certain tools).

Note that this protocol will not include facilities for finding agents that meet certain criteria: search is to be implemented by the MOSAICO Agent Repository in WP2, who will consume the above agent descriptions.

<sup>1</sup><https://semver.org/>

<sup>2</sup><https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#capabilities>

## 2.2 R2: Multi-part and multi-modal messaging

In order to enable automated agent-to-agent communication, the messages to be exchanged will need to be structured in a way that allows for separating the various relevant parts of the inputs to a task (e.g., a general description of the task, and any accompanying artifacts) and/or their outputs (e.g., a set of possible solutions to propose to the user, together with explanatory texts for each one). In short, it should be possible to exchange *multi-part* messages.

These different parts may be of several kinds, including natural language, code, data, or images. This implies that the protocol should be able to support non-textual parts if needed, e.g., by accompanying each part with a reference to the content type it includes (e.g., its MIME type<sup>3</sup>). We note this support of a mix of different content types as having *multi-modal* messages.

*Note:* beyond the protocol's base support for multi-part and multi-modal messaging, which will be evaluated in this document, there may be a need to standardise the specific parts used for each type of MOSAICO agent (solution, supervision, consensus, and collaboration). Particularly, there will need to be a standardised way for a solution agent to represent multiple alternative solutions, and to represent the failure to handle a given request. This aspect will be elaborated in future WP1 deliverables and is outside the scope of this document, which focuses on the core capabilities of the protocols.

## 2.3 R3: Multi-turn conversations

We anticipate that agents in the MOSAICO platform will exhibit different levels of agency, with some of them being able to detect when more information is needed before the request can be fulfilled. Specifically, this must be supported in order to tackle Task 1.3 (“Support for agent-directed information retrieval”). This means that rather than just providing a single turn of request-response communication between two agents, the protocol will need to allow for a *multi-turn* interaction that is made of an arbitrary sequence of exchanges.

In other cases, MOSAICO agents may need to relate feedback from other agents or from a human user to a previous task they attempted. In the case of a human user, this is considered to support a *human-in-the-loop* approach where a user can provide feedback on a given solution, or grant permission before a high-impact action is taken. This touches upon the needs of Task 1.4 (“Support for fine-grained feedback on task performance”) later in the project.

Both scenarios will require protocol messages to include a *task ID* that can be used by agents to identify subsequent turns from an initial communication. The protocol may also require a *context ID* that further places the conversation within a larger context (e.g., the software development project within which the software engineering task would take place). Conversations, task IDs and context IDs would typically be tracked from the client side (e.g., the MOSAICO reference client invoking the main agent to solve the task, or a collaboration agent invoking a solution agent).

The request for additional information may be targeted at the user (e.g., to clarify a specific requirement), or it may be targeted at the agent that sent the request (e.g., to run an LLM query from the client side). This second scenario (where the server agent asks the client agent to run an LLM query) is known as *sampling* in some protocols, such as MCP<sup>4</sup>.

<sup>3</sup><https://datatracker.ietf.org/doc/html/rfc6838>

<sup>4</sup><https://modelcontextprotocol.io/specification/2025-06-18/client/sampling>

## 2.4 R4: Asynchronous and streaming communication

Compared to simple information retrieval (typically done within a few seconds at most), the task performed by an agent may take longer (e.g., in the order of minutes) if it requires significant processing or collating of multiple information sources. While a *synchronous* (blocking) interaction which waits for a single response is easier to implement, it would block valuable resources for a significant time when used for those long-running tasks.

Instead, the protocol should support one or more forms of *asynchronous* communication where the client can send the initial request and later check on the progress of the task. This can be done through *polling* (the client periodically asks the agent on the current status of the task), or *callbacks* (the client tells the agent to send a message to a specific location for every update on the task).

As a middle ground, the protocol should support a *streaming* type of interaction where the client receives regular updates over an open connection, providing more timely feedback over the progress of a long-running task. Within the Web ecosystem, a number of technologies have emerged to do this, such as WebSockets<sup>5</sup> for full-duplex communication between client and server, or HTTP Server-Sent Events (SSE) for half-duplex communication. HTTP SSE, for example, is used by the OpenAI API to send regular updates on the tokens that are being produced<sup>6</sup>. More recently, MCP has proposed Streamable HTTP as a more flexible option than SSE<sup>7</sup>.

## 2.5 R5: Agent observability

The LLM community has already recognised the importance of being able to observe the various prompts being sent by agentic frameworks to the LLMs and their tool calls, with the release of *observability platforms* such as LangSmith<sup>8</sup> or Langfuse<sup>9</sup> that can collect these interactions and present them for analysis. These platforms provide libraries to instrument programs that use well-known LLM client libraries and agentic frameworks: the instrumentation typically sends traces following the OpenTelemetry<sup>10</sup> protocol to their servers.

MOSAICO has an additional layer above the interactions between manually-written code, LLMs, and tools: the collaborations between the various types of agents in the platform. In order to provide a unified view of the entire collaboration, messages will need to be related i) to the specific task they are solving, and ii) to their immediate super-task that they are a part of. For i), Section 2.3 already mentioned the use of task IDs, which could be mentioned in the OpenTelemetry traces submitted by the agent. For ii), AISP will require messages to include a *super-task ID* so that any traces sent from the agent are also related to the higher-level task that represents the overarching collaboration.

## 2.6 R6: Agent transparency

There are scenarios that may require deeper access to the internals of an agent. Section 2.1 already considered that agent descriptions could have additional information about the design

<sup>5</sup><https://websockets.spec.whatwg.org>

<sup>6</sup><https://platform.openai.com/docs/guides/streaming-responses?api-mode=responses>

<sup>7</sup><https://modelcontextprotocol.io/specification/2025-03-26/basic/transport#streamable-http>

<sup>8</sup><https://www.langchain.com/langsmith>

<sup>9</sup><https://langfuse.com/docs>

<sup>10</sup><https://opentelemetry.io>

of an agent, but this is not enough on its own. Transparency would also require to consider the actual behaviour of the agent, e.g., being able to access the memories it may hold on a specific topic, the reasons why it made a particular decision, or the selection of tools to be used for a given task.

In this regard, the protocol could require additional endpoints in some scenarios (e.g., for *accessing memories*), include additional *outputs with justifications* for its decisions (e.g., in a Belief-Desire-Intention or BDI format), or accept *configuration options* in its inputs that control the tools that it can use.

In any case, it is worth noting that transparency needs to be balanced against privacy concerns (e.g., being careful that one user cannot access the agent's memories about another user), and intellectual property concerns (as accessing internals may divulge protected IP). For this reason, this requirement has been marked as SHOULD for now as it will need to be revisited and evaluated by the various MOSAICO partners as their use cases further develop.

## 2.7 R7: Interoperability

Agents in the MOSAICO platform may be written in a variety of programming languages and may use one of the many available agentic frameworks. For example, there are cases where the ideal option for implementing a given solution or supervision agent may be written in a specific language. In other cases, the nature of the problem to be solved may lend itself especially well to a particular agentic framework.

For that reason, the protocol should not arbitrarily limit options for developers. For *programming language interoperability*, one common approach is to describe the various endpoints and messages in the protocol using a machine-readable format (such as Google Protobuf<sup>11</sup> or OpenAPI<sup>12</sup>), from which it is possible to generate base client/server code for multiple programming languages.

*Interoperability with agentic frameworks* is largely a software design concern rather than a protocol concern, so it will not be covered in this document. It is envisioned that example agents will be implemented on top of the chosen protocol and a selection of the most relevant agentic frameworks, for reuse by the use case providers and other adopters of the MOSAICO platform.

Another aspect is *network interoperability*: the chosen protocol should ideally build on top of broadly-accepted standards that will work well with existing network infrastructure (e.g., enterprise firewalls). HTTP would be the ideal base for AISP, given its broad adoption, the existence of mature specifications to augment it (e.g., TLS for encryption and SSE for streaming responses), and how it is well understood by both developers and systems administrators. It is already used by the most important commercial LLM APIs, such as OpenAI<sup>13</sup> or Anthropic<sup>14</sup>.

## 2.8 R8: Access control

There may be scenarios where the availability of an agent may be limited to a specific subset of users, or where the actions that an agent can perform may be limited depending on the user performing them. In this sense, the protocol needs to allow for *access control* to take place,

<sup>11</sup><https://protobuf.dev/programming-guides/editions/>

<sup>12</sup><https://swagger.io/specification/>

<sup>13</sup><https://platform.openai.com/docs/api-reference/introduction>

<sup>14</sup><https://docs.anthropic.com/en/api/overview>

which is further divided into *authentication* (ensuring clients are who they say they are) and *authorisation* (ensuring clients have permission to take a certain action).

In this area, it is best to rely on existing mature specifications whose existing implementations can offer the necessary level of security. Since Section 2.7 already highlighted HTTP as a strong base, it would be possible to use the authentication approaches supported by it: these include client/server verification via TLS certificates, or providing HTTP headers with authentication tokens (e.g., an API key). This would mean that authentication and authorisation would be mostly done at the HTTP layer and therefore *off-band* from AISP. One concern to be considered is that a collaboration agent that acts on behalf of a user may need to forward those credentials to any solution agents that it invokes on their behalf.

In regard to authorisation, this would be mostly implemented within the agent itself and not as part of the protocol. One possible option is to have the agent limit the tools it can use based on the user identity. Another option is to have the agent use tools that can prompt for additional permissions when required, and if so prompt the user for that additional authentication by taking advantage of the multi-turn support in Section 2.3.

## 2.9 R9: Bandwidth-efficient serialisation

In order to save bandwidth, the messages exchanged as part of AISP should be compact in size. One option is to use efficient serialisation methods from frameworks such as Google Protobuf<sup>15</sup> or Apache Thrift<sup>16</sup>.

This requirement has been marked as optional (SHOULD), as it may be possible to obtain similar bandwidth savings by leveraging the existing compression support in HTTP. Most HTTP clients support *gzip* compression for responses, by using the *Content-Encoding* header<sup>17</sup>). More efficient serialisation may still be worth pursuing, however, as it would reduce the need to spend processing time on compression and decompression.

## 3 Protocol design and selection

This section will discuss the protocol developed within Task 1.1 to meet the requirements in Section 2.1, as well as the current state-of-the-art in agent intercommunication protocols. A subset of the current state-of-the-art protocols will be selected for comparison with the new protocol, in relation to the needs of MOSAICO.

### 3.1 Design of a custom inter-agent protocol

This subsection presents the design and implementation of the new inter-agent protocol developed for MOSAICO: the code and materials used for this section are available from the *aisp-prototyping* Gitlab project<sup>18</sup>.

The protocol was first defined in the machine-readable OpenAPI format, to allow for automated generation of client and server code. Figure 2 shows a screenshot of the Swagger-UI

<sup>15</sup><https://protobuf.dev/programming-guides/encoding/>

<sup>16</sup><https://thrift.apache.org/>

<sup>17</sup><https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Encoding>

<sup>18</sup><https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/aisp-prototyping/-/tree/main/new-protocol>

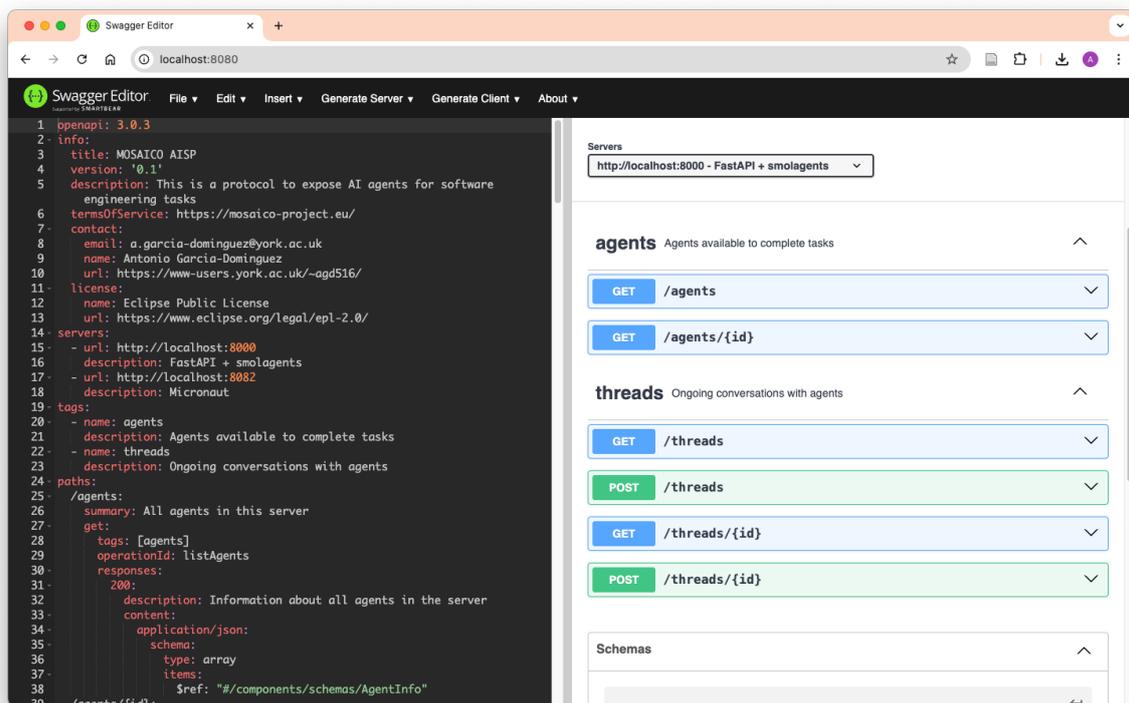


Figure 2: Screenshot of the Swagger-UI client operating on the OpenAPI specification of the custom protocol

web interface interpreting the protocol specification to produce a client to test and refine the protocol.

The protocol is based on open standards, using HTTP to exchange JSON messages (to meet R7), and has been designed to be minimalistic to reduce the effort required to support it from an agent. The protocol is based on two main concepts: that an *agent server* hosts one or more *agents* that can complete certain tasks, and that every conversation with an agent involves the exchange of a number of messages grouped in a *thread*.

### 3.1.1 Agent discovery

In order to support R1, the protocol includes two HTTP endpoints for listing the agents provided by an agent server: **GET /agents** describes all the agents provided by the server, and **GET /agents/{id}** describes a specific agent.

Listing 1 shows an example response from **GET /agents/{id}** that describes an implementation of the syntactic supervision agent from Section 4.1. The description currently includes the universally unique identifier (UUID) for the agent, a description in natural language (for end-users or for LLMs), an indication of the type of MOSAICO agent it is (solution, supervision, collaboration, or consensus), and then a description of the various pieces of content that can be provided as input, produced as output, or asked as part of a question back to the original requester (not shown in this example).

Each piece of content is further described with their own ID (in this case, a simple string that only needs to be different from the other pieces of content), a MIME type identifying the kind of content that is expected, and a description in natural language of the expected content

Listing 1: Description of the syntactic supervision agent in the new protocol

```

1 {
2   "id": "f1d7f8e9-1d8d-4c9d-809f-4f3a160066fd",
3   "description": "Evaluates a metamodel expressed in the Emfatic format [...]",
4   "agentType": "supervision",
5   "inputContent": [
6     {
7       "id": "emfatic",
8       "mimeType": "application/x-emfatic",
9       "description": "Emfatic code to be evaluated"
10    }
11  ],
12  "outputContent": [
13    {
14      "id": "evaluation",
15      "mimeType": "text/plain",
16      "description": "valid/invalid"
17    },
18    {
19      "id": "feedback",
20      "mimeType": "text/plain",
21      "description": "Feedback in natural language (for user or LLM)"
22    }
23  ]
24 }

```

Listing 2: Example response from **GET /threads** in the new protocol

```

1 [
2   {
3     "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
4     "agentId": "3fa85f64-5717-4562-b3fc-2c963f66afa6"
5   }
6 ]

```

for that part. In this case, we can see that it is a supervision agent that takes in Emfatic code as input, and provides two outputs: a machine-readable pass/fail evaluation (indicated as “valid” or “invalid”), and a piece of feedback in natural language (which is aimed at the end-user or at the LLM used by a solution agent).

### 3.1.2 Thread management

The **/threads** and **/threads/id** endpoints relate to managing the conversations between the clients and the agents hosted at the server.

First, the **GET /threads** endpoint lists all the threads being managed by the server: a minimal response is shown in Listing 2. In a production environment, these would be limited to the threads accessible by the user (usually, the threads they started).

To start a new conversation, a client would send a request as the one in Listing 3. The request indicates the agent that should be addressed, as well as an optional *super-thread ID* that encodes the *super-task ID* concept needed for cross-collaboration observability (R5).

Listing 3: Example request to **POST /threads** in the new protocol

```

1 {
2   "agentId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
3   "superThreadId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
4   "firstMessage": {
5     "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
6     "messageType": "solution-request",
7     "source": "http://uri/of/other/thread",
8     "content": [
9       {
10        "id": "instructions",
11        "mimeType": "text/plain",
12        "content": "Provided instructions for the agent."
13      }
14    ]
15  }
16 }

```

This is followed by the message used to start the thread. Every message has its own UUID, a message type (in this case, a “solution-request”, which alternatively could be a “supervision-request”), an optional source (to identify the user/agent that may have sent the request), and one or more pieces of content (in this case, a set of instructions that the agent should follow).

The agent server will then respond with a thread ID, which can be used with the **GET /threads/{id}** and **POST /threads/{id}** endpoints. **GET /threads/{id}** can be used in two ways, depending on the content type requested by the client through the HTTP Accept header:

- If the client asks for `application/json` content (i.e. JSON), the server will respond with the full history of that thread (all messages exchanged) up to that point, then close the connection. These messages may also include interim updates from the agent (represented as “agent-step” messages), system prompts used by LLMs (represented as “system” messages), tool calls to be performed by the client (represented as “tool” messages), questions to be asked to the user for multi-turn support (“question” messages), the result from a supervision agent (“supervision-result”), or the result from a solution agent (“solution-agent”).

This type of request fulfils the support for polling required by R4.

- If the client asks for `text/event-stream` content, then the server will use SSE to provide a stream of all the messages so far in the thread, and any future messages (until the server considers the thread to be completed, e.g., because a result has been produced). This is crucial for agent-to-agent communication, as it allows a collaboration agent to receive updates about the progress of a task that they have given to another agent, without incurring the overhead of repeated polling that would be required without streaming.

This type of request fulfils the support for streaming required by R4.

Finally, the **POST /threads/{id}** endpoint allows clients to respond to a question raised in a thread, by appending a message to the end of the thread. This is intended to support multi-turn communication (R3).

Name	Description	Date	Reference
ANP	Agent Network Protocol	Oct 2024	(ANP Team, 2025)
MCP *	Model Context Protocol	Nov 2024	(Spahr-Summers et al., 2025)
LMOS	Language Model Operating System	Dec 2024	(Eclipse Foundation, 2025)
AGNTCY ACP	Agent Connect Protocol	Feb 2025	(Duminuco et al., 2025)
IBM ACP	Agent Communication Protocol	Mar 2025	(BeeAI, 2025)
A2A *	Agent-to-Agent	Apr 2025	(Linux Foundation, 2025)

Table 2: Recently proposed protocols for agent collaboration, ordered by date of first release. Items marked with \* are the protocols chosen for comparison against creating a new protocol.

## 3.2 Review of the state-of-the-art

In the original proposal, we suggested the Microsoft Language Server Protocol (LSP) as a potential base from which AISP could be developed. This was because of its use within GenAI-based code completion extensions for IDEs such as CoPilot. However, LSP on its own is dedicated to adding language support to an IDE, and does not have any specific functionalities for collaborations between AI agents.

From the second half of 2024, there was a significant push from industry on agent-to-agent collaborations, and a number of protocols closer to the goals of MOSAICO have been proposed, with recent surveys comparing their relative strengths and weaknesses (Ehtesham et al., 2025; Yang et al., 2025). Table 2 lists the protocols that were first selected for a cursory examination, based on the above surveys and our own literature review.

From these protocols, we selected MCP and A2A for a comparison against creating a minimalist protocol from scratch. This was based on the following considerations:

- The MCP and A2A protocols have the most penetration within the developer community. At the time of writing (July 2025), the MCP Python SDK Github project had 16k stars, and the main A2A Github project had 18.4k stars. In comparison, AGNTCY ACP has 145, ANP has 904, IBM ACP has 688, and LMOS has 123.
- MCP and A2A have significant amounts of documentation and SDKs that could be reused later by the use case partners. Most of the other protocols also had documentation, although not always to the same extent: for example, a significant portion of the ANP documentation was only available in Chinese. We also noticed inconsistencies between various AGNTCY documents: for instance, ACP did not explicitly allow for describing agent skills according to AGNTCY's own skills taxonomy.
- Inspecting the documentation of MCP and A2A indicated that both met many of the requirements in Section 2, at least partially. In contrast, we found a number of limitations in AGNTCY ACP and IBM ACP: for example, the former lacked explicit support for multi-part messages (leaving it up to the agent developer), and the latter did not allow for indicating

more than one level of context in multi-turn communications (i.e. it only supported a task ID, accordingly to the terminology in Section 2.3).

- A2A showed significant support for multi-turn conversations in comparison to AGNTCY ACP and IBM ACP, with the ability not only to request user input, but also to request additional authorisation.
- MCP pioneered the use of Streamable HTTP over SSE for more flexible deployment and improved scalability (by supporting ephemeral servers and reducing the required number of HTTP connections), and a number of MCP servers had already been implemented to allow LLMs to access various popular information systems<sup>19</sup>.
- Both protocols are open-source and have well-defined governance structures. MCP is under the MIT License, and A2A is under the Apache 2.0 license.

The following subsections present the key characteristics for each protocol. Where necessary, footnotes with links to specific sections of their specifications will be provided throughout the rest of the document.

### 3.2.1 Agent-to-Agent protocol (A2A)

The Agent-to-Agent (A2A) protocol was originally announced by Google in April 2025<sup>20</sup>, and later migrated to the stewardship of the Linux Foundation in June 2025<sup>21</sup>. A2A was designed under five key design principles:

- **Simplicity:** Reusing existing standards like HTTP, JSON-RPC, and Server-Sent Events (SSE).
- **Enterprise Readiness:** Aligning with standard web practices for authentication, authorisation, security, privacy, tracing, and monitoring.
- **Asynchronous:** Natively supporting long-running tasks, where agents or users may not remain continuously connected.
- **Modality Independent:** Allowing agents to communicate using a variety of content types.
- **Opaque Execution:** Allowing agents to collaborate without exposing internal logic, memory, or proprietary tools.

The key principles are largely aligned with the MOSAICO requirements, with simplicity related to R7 (interoperability), enterprise readiness related to R5 and R8, asynchronous related to R4, and modality independence related to R2. The only mismatch is in its opaqueness, which is at odds with R6 (transparency).

The official specification of A2A<sup>22</sup> is expressed as a set of JSON-RPC messages to be exchanged between an A2A client and an A2A server. An A2A server hosts one or more *agents*, which are described using *agent cards* served from the same server, or from a registry or catalog. Agent cards are JSON documents served over HTTP that provide structured metadata advertising each agent's capabilities and providing the necessary communication endpoints.

<sup>19</sup><https://github.com/modelcontextprotocol/servers>

<sup>20</sup><https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/>

<sup>21</sup><https://developers.googleblog.com/en/google-cloud-donates-a2a-to-linux-foundation/>

<sup>22</sup><https://a2a-protocol.org/latest/specification/>

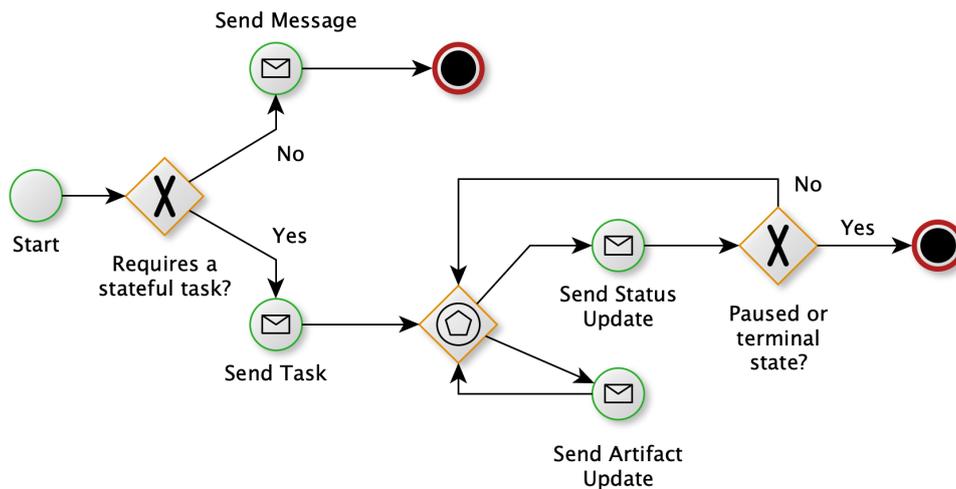


Figure 3: A2A task lifecycle

A2A supports a combination of synchronous (blocking) request-response interactions, and two forms of asynchronous communication: push notifications (i.e. having the agent initiate a connection to provide an update), and streaming (i.e. having the user maintain a long-running connection to the agent, who can share updates when appropriate).

A2A defines a specific structure for a message, which can be divided into multiple parts. Within a specific turn of communication between a client and an agent, the client will send a message with its request to the agent, who can reply directly with another message (in case a response can be immediately given), or with a *task*. A task represents a stateful and long-running piece of work whose progress needs to be tracked by the client and the server (e.g., the use of an LLM in combination with various tools to propose or evaluate a solution).

Tasks have a well-defined lifecycle within A2A, which is outlined in Figure 3:

1. A task is created in response to a message from the client when the agent considers that producing the response will take some time. This task is immediately returned to the client, and includes a unique ID for later reference by the client.
2. During the execution of the task, the agent can send interim updates on any changes in the *artifacts* produced by the task (e.g., source code or documents), or in the *state* of the task.
3. At some point, the task will reach a terminal state (e.g., *completed*, *cancelled*, *failed*, or *rejected*), or a paused state (*input-required* or *auth-required*). Paused states allow the client to provide needed additional information, allowing the agent to resume the execution of the task.

### 3.2.2 Model Context Protocol (MCP)

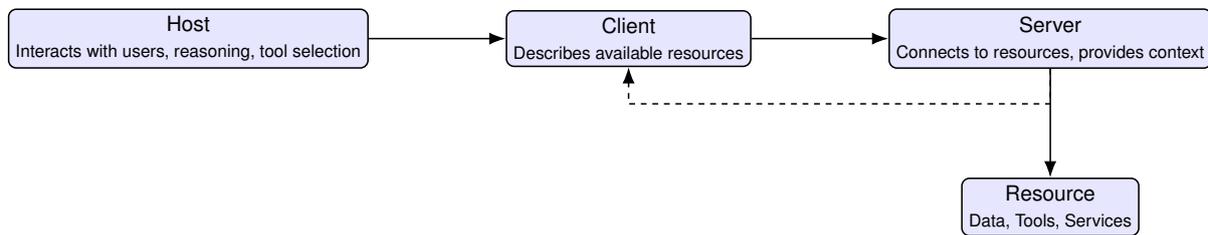


Figure 4: MCP interaction between Host, Client, Server, and Resource

The Model Context Protocol (MCP) is a JSON-RPC client–server interface for secure context ingestion and structured tool invocation which streamlines the integration of LLMs with external data sources and tools. To this end, it supports flexible plug-and-play tools, safe infrastructure integration, and compatibility across LLM vendors (Ehtesham et al., 2025). According to the AI agent protocol taxonomy described in (Yang et al., 2025), MCP is a general-purpose protocol designed to accommodate a wide range of entities through a unified protocol paradigm that facilitates diverse communication scenarios. The usage of the MCP protocol is characterized by the presence of four components as depicted in Figure 4:

- **Host** – refers to agents responsible for interacting with users, understanding and reasoning through user queries and selecting tools.
- **Client** – is connected to a host and is responsible for providing descriptions of available resources.
- **Server** – is connected to the resource(s) and establishes a one-to-one connection with the client, providing the required context from the resource(s) to the client.
- **Resource** – refers to *data* (e.g., local file systems), *tools* (e.g., Git), or *services* (e.g., search engines) provided locally or remotely.

When MCP is used for its primary intended purpose (i.e., the decoupling and integration of external data sources and tools), MCP can be used in conjunction with other AI agent protocols. In this case, MCP can complement other agent communication protocols, such as A2A, as presented in (Li et al., 2025). Therefore, MCP and A2A are not necessarily competing protocols, as they can also be used complementarily. However, this report will explore the usage of the MCP protocol for agent-to-agent communication as a competing (i.e., standalone) protocol. Note that while MCP is not primarily designed for agent-to-agent communication, it can also be used for that purpose<sup>23</sup>: given its significant market share, we considered it appropriate to evaluate this use case.

## 4 Comparison methodology

This section will describe the approach used to compare the various options to be evaluated in Section 5, by using each protocol to implement a multi-agent system that solves a software engineering task, and contrasting our experiences.

<sup>23</sup><https://aws.amazon.com/blogs/opensource/open-protocols-for-agent-interopability-part-1-inter-agent-communication-on-mcp/>

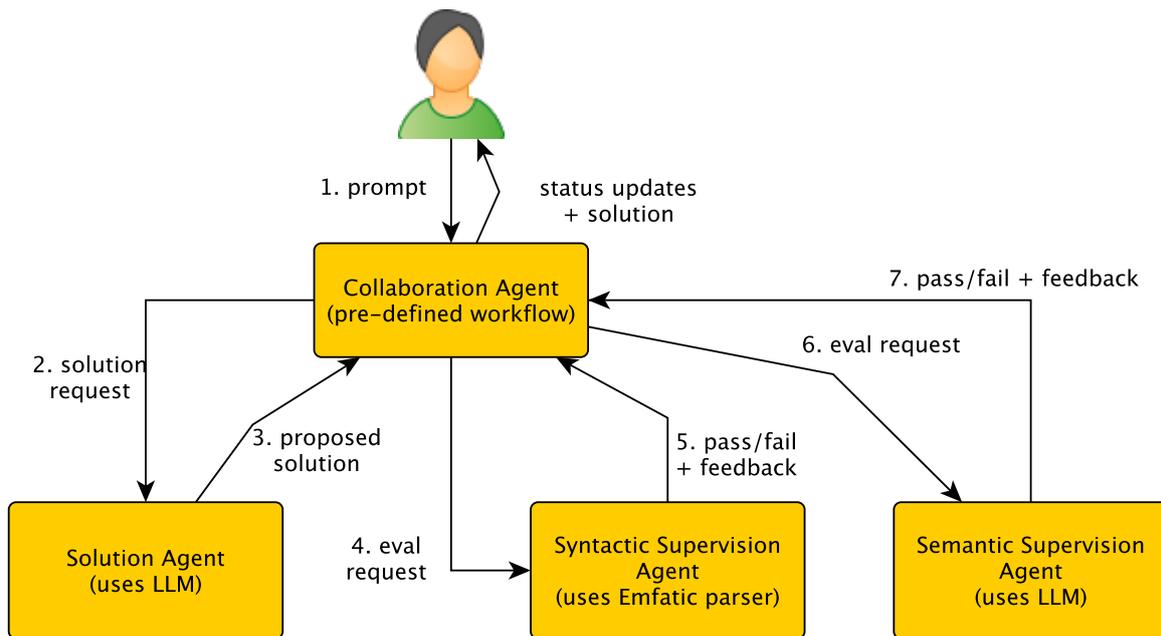


Figure 5: Envisioned agent collaboration for the chosen software engineering task (object-oriented model generation)

## 4.1 Software engineering task to solve

In order to evaluate the protocols in a realistic setting, we will implement a collaboration of multiple agents that solves a common software engineering task: given a prompt in natural language, produce an object-oriented model of the domain.

The model will need to be expressed in a machine-readable format, so that it can be used by tools to produce code and documentation automatically. While there are formats like PlantUML<sup>24</sup> and MermaidJS<sup>25</sup> that may be sufficiently popular to be directly generated by existing foundational models (e.g., GPT-4), these formats are only intended for drawing and not for automated model-to-text transformations (e.g., generating Java code or HTML documentation of a domain). Instead, the Eclipse Emfatic<sup>26</sup> format will be produced, which has an automated transformation to the Ecore language implemented by the Eclipse Modelling Framework (EMF) (Steinberg et al., 2008), from which it is possible to generate Java code and perform other automated transformations: for example, using the model management languages from the Eclipse Epsilon<sup>27</sup> project.

From the description above, we can envisage the following multi-agent collaboration to solve this domain model generation task (the *happy path* where all evaluations indicate a valid solution is shown in Figure 5):

1. A *collaboration agent* receives the description of the domain from the user (step 1 in Figure 5), and forwards it as a new request to the *solution agent* (step 2). The collaboration agent may use an LLM as a frontend to the user (e.g., to filter out irrelevant queries), or

<sup>24</sup><https://plantuml.com/class-diagram>

<sup>25</sup><https://mermaid.js.org/syntax/classDiagram.html>

<sup>26</sup><https://eclipse.dev/emfatic/>

<sup>27</sup><https://eclipse.dev/epsilon/>

it may just be written as a traditional imperative program: as the focus of this deliverable is to compare protocols, using LLMs to guide collaborations is not a requirement. Either way, the workflow is fixed for this deliverable's experiments: computing collaborations on the fly is outside the scope of this deliverable.

2. A *solution agent* uses an LLM to drive the production of the domain model, which it returns to the collaboration agent (step 3). This LLM may directly generate the domain model in the Emfatic language, or it may use tools to build up a model and then generate Emfatic source code, by following the ReAct architecture (Yao et al., 2023). The choice will depend on the capabilities and training data used by the selected LLM: larger models may be able to directly generate Emfatic, whereas smaller models may require the assistance of purpose-specific tools and the ReAct architecture. It is important to note that solution agents may fail to produce a solution altogether: this failure should be signalled to the collaboration server, which would ask the solution agent (up to a certain number of attempts).
3. There is the risk that the solution agent may produce output that may not conform to the Emfatic grammar, or that may not represent a valid use of Emfatic's features. For that reason, the collaboration agent will take the proposed solution and forward it to the *syntactic supervision agent* (step 4), which will parse the solution using the Emfatic parser (written in Java).
  - If any errors are reported, the syntactic supervision agent will inform the collaboration agent that the solution is invalid, and include feedback with the specific errors. The collaboration agent will then pass on the feedback to the solution agent, who will be given another attempt to produce a valid solution (up to a certain number of attempts).
  - If no errors are reported, the agent will only indicate that the solution is valid (step 5), and the collaboration agent will proceed to the next step.
4. There is also the risk that the output, while syntactically valid, may not meet all the requirements from the original description. For that reason, the collaboration agent will pass on the original description of the domain and the generated Emfatic source to a *semantic supervision agent* (step 6). This agent will use an LLM that will have been prepared in some way to read Emfatic code: a low-cost way to do this is through *few-shot prompting* (giving a few examples of Emfatic syntax within the prompt).
  - If any issues are detected, the semantic supervision agent informs the collaboration agent, who forwards the feedback to the solution agent and asks for an updated solution (up to a certain number of attempts).
  - If the semantic supervision agent reports that there are no issues in the solution (step 7), the collaboration agent passes on the solution to the user.

Throughout every step of the collaboration, the collaboration agent is expected to provide regular status updates on the various agents that are being involved and the various outputs being produced.

## 4.2 Requirements evaluation

The intended approach to evaluate each of the requirements in Section 2 is as follows:

- R1: Agent discoverability** The protocol should have a way to list the collaboration / solution / supervision agents implemented by a given *agent server*, and describe their most important features (listed in Section 2.1). It is envisioned that two agent servers may be needed for the task in Section 4.1: one written in Java (as the Emfatic parser is written in Java), and one written in Python (as most ReAct implementations are Python-based).
- R2: Multi-part and multi-modal messaging** The protocol should separate the natural language parts of their responses from their structured parts (e.g., valid/invalid evaluations, or the generated Emfatic source code).
- R3: Multi-turn conversations** The protocol should allow the solution agent to request further clarification on the domain description if it considered necessary. Note that for this experiment, it is not needed to have the implemented agents actually ask for clarification: it would be enough to point to the specific part of the protocol that would support this in a future version of the agent.
- R4: Asynchronous and streaming communication** The protocol should allow a user to either ask regularly for updates on the generation of the solution (polling), or to maintain an ongoing connection and receive live updates (streaming). It is anticipated that the collaboration agent will use the same part of the protocol to stay up-to-date on the progress of the solution and supervision agents.
- R5: Agent observability** For this experiment, we will evaluate integrating the agents with existing observability platforms for AI which can be run on-premises, such as Phoenix and Langfuse. We will also examine whether the protocols support (whether directly or through extensions) specifying the super-task ID mentioned in Section 2.5.
- R6: Agent transparency** The support for transparency from the protocol will be evaluated by inspecting its specification, rather than by implementing it in the agents themselves. As mentioned in Section 2.6, this will include whether there are ways to access memories, allow agents to include outputs with justifications, or specify configuration inputs that allow for controlling which tools it can use.
- R7: Interoperability** As mentioned above, we anticipate needing at least two agent servers: one implemented in Java, and one in Python. This will stress the programming language interoperability of the protocols. We may consider agentic framework interoperability by using different agentic frameworks as well (e.g., smolagents<sup>28</sup> and LangGraph<sup>29</sup>), although this is not a high priority (as mentioned in Section 2.7, this is largely a software design concern and not a protocol design concern). For network interoperability, both A2A and MCP are based on HTTP so they already meet that aspect: the new minimalistic protocol will also be HTTP-based for this reason.
- R8: Access control** Access control will be evaluated through examination of the protocol itself, rather than through the implemented agents. It is envisioned that HTTP-based protocols can largely treat authentication in an *off-band* manner, through existing HTTP mechanisms (additional headers, TLS encryption, etc.). Authorisation will require having some way to signal the refusal to comply with a specific request, or to provide access to a specific agent.

---

<sup>28</sup><https://huggingface.co/docs/smolagents/index>

<sup>29</sup><https://www.langchain.com/langgraph>

**R9: Bandwidth-efficient serialisation** To evaluate bandwidth usage, a network monitoring tool (such as those bundled by operating systems or containerisation tools, or standalone protocol analysers such as Wireshark<sup>30</sup>) will be used to measure the total size in bytes of all messages sent as part of a given collaboration. In regard to latency: whereas the exact amount is dependent on the implementation details of each agent rather than the protocol itself, we can measure the number of messages that were exchanged to estimate how many round-trip communications would be needed, which is a matter of the design of the protocol.

## 5 Results

In this section we will present the results of evaluating three approaches for implementing the MOSAICO AISP, according to the approach in Section 4: using our new protocol, using A2A as a base, and using MCP as a base. For A2A and MCP, we will provide footnotes with links to the relevant sections of their specifications when needed. This section will finish with a comparison of the bandwidth usage of the various protocols.

### 5.1 Using the new protocol as a base

This section presents the results obtained by implementing the multi-agent system on top of the custom protocol described in Section 3.1.

#### 5.1.1 Implementation

The protocol has been implemented in two agent servers: a Python-based agent server which provides the solution agent, and a Java-based agent server which provides the collaboration and supervision agents. Table 3 summarises the main features of the agents provided by the agent servers.

The Python agent server has been implemented from scratch as a FastAPI<sup>31</sup> application, where the various messages in the protocol have been manually specified as Pydantic<sup>32</sup> models for automated validation. This agent server only provides the solution agent, which uses the Smolagents agentic framework to allow a local LLM to perform Pythonic tool calling to build up the Emfatic metamodel that meets the user's demands. The various tools validate every step the LLM takes, and a final model-to-text transformation via a Jinja<sup>33</sup> template produces an Emfatic code fragment (avoiding situations where the LLM will hallucinate an invalid syntax).

The Java agent server was implemented from scratch as a Micronaut application. Micronaut includes an automated code generator from OpenAPI specifications, so in this case it was possible to automatically generate a significant part of the client and server code, only requiring minor customisations via Java subclassing. The Java server provides both supervision agents: the syntactic supervision agent uses the Emfatic parser library to validate the syntax of an Emfatic file, and the semantic supervision agent uses a local LLM via Langchain4j<sup>34</sup> to judge whether the metamodel meets the requirements provided in the user's original prompt. The

<sup>30</sup><https://www.wireshark.org/>

<sup>31</sup><https://fastapi.tiangolo.com/>

<sup>32</sup><https://docs.pydantic.dev/latest/>

<sup>33</sup><https://jinja.palletsprojects.com/en/stable/>

<sup>34</sup><https://docs.langchain4j.dev/>

Name	Responsibilities	Programming language	LLM	Agentic framework
Collaboration agent	Interact with user, orchestrates agent interactions via streaming	Java	None	None
Solution agent	Analyse metamodel requirements and generate Emfatic code	Python	qwen2.5-coder:32b	Smolagents
Syntactic supervision agent	Validate syntax of Emfatic code using a parser	Java	None	None
Semantic supervision agent	Check whether all user requirements are met	Java	qwen2.5-coder:32b	Langchain4j

Table 3: Agents in the new protocol implementation

collaboration agent reuses the client code generated from the OpenAPI spec to integrate the solution and supervision agents into a fixed workflow. If a supervision agent rejects a given solution, the collaboration agent provides the feedback to the solution agent so it can provide an updated solution.

Observability (R5) has been achieved at the individual agent level, by integrating both agent servers with the Phoenix Arize observability server<sup>35</sup>, which implements the OpenTelemetry protocol. Every use of the Python-based solution agent is recorded, including every LLM interaction and tool call. Likewise, every LLM use from the Java-based semantic supervision agent is recorded. The integration of these per-agent traces into a unified trace across a collaboration of agents (using super-task IDs) is outside the scope of this deliverable: this may require customizing the instrumentation that is performed by these libraries, which may require contributions to several related open source projects.

## 5.1.2 Demonstration

To test the implementation, an example prompt was designed to describe a metamodel in detail, to avoid ambiguity and get a consistent result from the LLM for evaluation. Every class was listed with their attributes and corresponding data types, and their containment and non-containment references. The following prompt was given to the collaboration agent:

Please generate a metamodel of a simple library management system with three main concepts: Library, Book, and Author.

Each Library has a name (String) and contains a collection of Books and a collection of Authors (both containment references). Each Book has a title (String) and a page

<sup>35</sup><https://phoenix.arize.com/>

```

1 @namespace(uri="http://librarymanagement.com/1.0", prefix="p")
2 package LibraryManagementSystem;
3
4 class NamedElement {
5     attr String name;
6 }
7 class Library extends NamedElement {
8     val Book[*] books;
9     val Author[*] authors;
10 }
11 class Book {
12     attr String title;
13     attr int pageCount;
14     ref Author mainAuthor;
15 }
16 class Author extends NamedElement {
17     attr int age;
18     ref Book[*] booksWritten;
19 }
20 class Model {
21     val Library[*] libraries;
22 }

```

Listing 4: The Emfatic metamodel generated by the solution agent based on the new protocol

count(int) and has a non-containment reference to its main Author. Each Author has a name (String) and age (int) and knows which Books they have written (non-containment reference to Books).

The system should clearly distinguish between containment and non-containment relationships: A Library owns its Books and its Authors (deleting the Library deletes its Books and its Authors). Authors and Books reference each other but do not manage each other's lifecycles.

The collaboration agent forwards this request to the solution agent, which uses Smolagents to build and generate the Emfatic metamodel in Listing 4. The metamodel meets the above requirements: within a "LibraryManagementSystem" package, it has defined classes for Library, Book, and Author. Inheritance has been used to place the common "name" attribute in a common NamedElement superclass, and the Library contains both the Book objects and the Author objects. Books have a title, page count, and a main author. Authors have their age and references to the books they have written. Finally, there is a Model class that acts as the root container for the entire model, which is a common pattern in metamodels that the semantic supervision agent was instructed to check.

A sample trace collected by the Phoenix observability server from the solution agent is shown in Figure 6. The collected traces include every interaction with an LLM, as well as all the Python tool calls that the LLM performed to build up the metamodel in its short-term memories and generate its source code.

In addition to the above traces, the solution agent uses "agent-step" messages to share every step it takes: these steps are streamed by the collaboration agent back to the user. The collaboration agent also streams back to the user the intermediate requests and responses exchanged with the supervisor agents to check the syntactic validity of the provided solution, as well as its semantic correctness (in terms of meeting the original requirements). Since

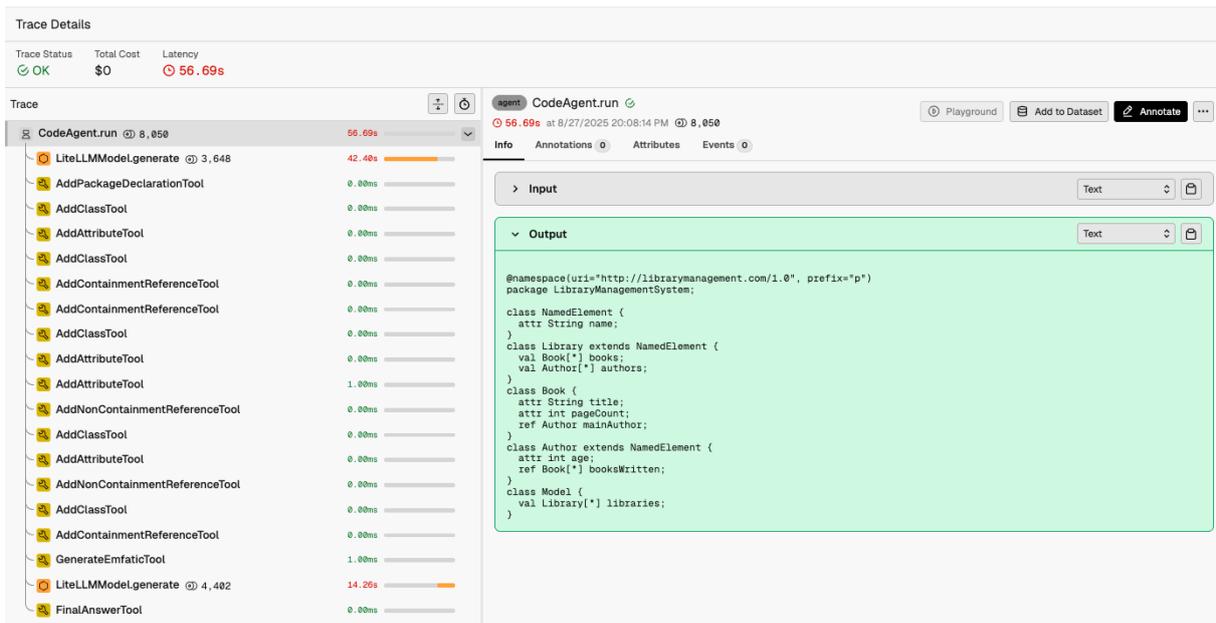


Figure 6: Screenshot of the Phoenix traces for the production of the second and final version of the metamodel by the new protocol solution agent

the semantic supervision agent uses an LLM as well, its interactions are also recorded by the Phoenix server: Figure 7 shows how the semantic supervision agent rejected a previous version of the solution, which was missing the title attribute in the Book class. The traces include the system prompt, the user prompt that tells the agent about the original prompt from the user and the proposed solution, and the final output from the LLM (which has both positive and negative feedback).

### 5.1.3 Requirements evaluation

Based on the above experiment, the protocol can be evaluated as fully meeting 6 requirements, and partially meeting 1 requirement, with bandwidth efficiency to be later compared against A2A and MCP in Sections 5.2.3 and 5.3.3 respectively. The remaining requirement (transparency) will need further clarification between the partners on the balance to strike between transparency and the protection of intellectual property and personal data:

**R1: Agent discoverability** Partially: the protocol includes `/agent` and `/agent/{id}` endpoints to retrieve descriptions of the agents, including their expected content for their inputs, outputs, and questions. The descriptions do not yet include provenance information, explicit description of skills according to a taxonomy, security aspects, descriptions of optional protocol features (as no such optional features exist at the moment), or transparency support information.

**R2: Multi-part and multi-modal messaging** Yes: the protocol allows for messages to include an arbitrary number of parts, with different content types and arbitrary content.

**R3: Multi-turn conversations** Yes: the protocol includes a `POST /threads/{id}` endpoint for adding further messages to a thread, and a thread may include “question” messages from an agent back to the original sender of the request.

Trace Details

Trace Status: OK Total Cost: \$0 Latency: 9.09s

Trace: generate 919 9.09s

Info Annotations 0 Attributes Events 0

```

class NamedElement {
  attr String name;
}
class Library extends NamedElement {
  val Book[*] books;
  val Author[*] authors;
}
class Book {
  attr int pageCount;
  ref Author mainAuthor;
}
class Author extends NamedElement {
  attr int age;
  ref Book[*] booksWritten;
}
class Model {
  val Library[*] libraries;
}

```

You are asked to evaluate the provided metamodel in terms of its semantic quality. Do not judge the syntax itself, or any other related artifacts (e.g. validation rules or constraints). Do not ask to define constraints or rules to manage references, as they cannot be expressed in Erratic. Instead, consider the following topics:

- Does the metamodel include the classes, attributes, and references needed to meet the instructions? You must only suggest adding new elements if they are needed to meet the instructions.
- Has inheritance been used effectively to classify instances and avoid duplication of fields?
- Is there a class that acts as the root container?

If you do not see any issues, you must reply "VALID" and nothing else. If you see any issues, you must reply with "INVALID" followed by a list of recommendations for improvement. Please write all your recommendations for improvement in English, without using code."}]}

Output Messages Output

assistant Markdown

INVALID

- The Book class does not include the title attribute as specified in the instructions.
- Inheritance from NamedElement is used effectively to avoid duplication of the name attribute in both Library and Author.
- The Model class acts as a root container for Library instances, which meets the requirement of having a root container.

Recommendations:  
Add a title attribute to the Book class to meet the specified requirements.

Figure 7: Screenshot of the Phoenix traces for the semantic supervision of the first version of the generated metamodel (which was missing the title attribute in the Book class)

- R4: Asynchronous and streaming communication** Yes: the **GET /threads/{id}** allows for both polling a thread for new messages, as well as streaming its messages as they arrive. These streams have been used by the collaboration agent to stay up-to-date with every step taken by the solution agent, and to receive the responses from the supervision agents.
- R5: Agent observability** Yes: the **POST /threads** endpoint allows for specifying a “super-thread ID” which encodes the concept of super-task ID in Section 2.5. This should allow agents to correctly identify the parent span to associate their traces while communicating with OpenTelemetry servers such as Phoenix Arize (as demonstrated above).
- R6: Agent transparency** No: the current version of the protocol does not include any specific mechanisms for transparency (R6), as the specific needs have yet to be discussed among the MOSAICO partners. However, given the support for multi-part responses, it would be possible to separate a justification of the agent’s output into its own part. Agent descriptions could be extended with support for configuration options, which would be provided when starting a new thread.
- R7: Interoperability** Yes: the protocol has been implemented on two different programming languages, and these agents can communicate with each other. Agentic framework interoperability has not been evaluated in this scenario, but we do not anticipate any major issues given the ease of the integration between Smolagents and this protocol. Regarding network interoperability, the protocol is HTTP based so it should be able to interoperate well with existing network infrastructure.
- R8: Access control** Yes: while the protocol does not have any specific mechanisms for authorisation or authentication, it is based on HTTP which allows for addressing these concerns through *off-band* mechanisms such as `Authorisation` HTTP headers and standard HTTP 401/403 status codes, in an orthogonal way to the provided endpoints. Our aim was to integrate with existing mature encryption and access control mechanisms for HTTP whenever possible.
- R9: Bandwidth-efficient serialisation** Using Wireshark to capture all HTTP traffic to and from the Python and Java servers while the agent collaboration was taking place, a total of 28 HTTP requests and responses were exchanged, adding up to 15,711 bytes. These consisted of 7 **POST /threads** requests and their responses, and 7 **GET /threads/{id}** requests and their responses. These results will act as a comparison baseline for the other protocols.

## 5.2 Using A2A as a base

This section explains the implementation of the system described in Section 4.1 using the A2A communication protocol, and evaluates A2A against the requirements in Section 4.2.

### 5.2.1 Implementation

To implement the task in Section 4.1, four agents were designed and implemented with distinct well-defined responsibilities, which communicated with each other to perform a series of action

Name	Responsibilities	Programming language	LLM	Agentic framework
Collaboration agent	Interact with user, orchestrate agent interactions, and manage task polling	Python	None	A2A SDK
Solution agent	Analyse metamodel requirements, and generate Emfatic code	Python	qwen2.5-coder:32b	Smolagents, A2A SDK
Syntactic supervision agent	Validate syntax of Emfatic code using a parser	Java	None	A2A SDK
Semantic supervision agent	Check whether all user requirements are met	Python	qwen2.5-coder:32b	Langchain, A2A SDK

Table 4: Agents in the A2A implementation

to solve the task. The source code is available on MOSAICO repository <sup>36</sup>. Between agents, there are variabilities in programming languages (Python, Java), LLMs (local or no LLM), agentic framework (A2A SDK, Smolagents, LangChain). Please refer to the Table 4 for agent implementation details. The user interaction and orchestration of agents are demonstrated in Figure 8 including A2A implementation details.

To support monitoring and debugging, observability is enabled through Langfuse, which provides detailed logs and traces of agent interactions.

For reliability, the system was tested with a predefined detailed use case executed via a Python script, which enforces a fixed workflow to validate consistency and correctness across different agent interactions.

## 5.2.2 Demonstration

To allow for a direct comparison between the various approaches, the same prompt about a library metamodel from Section 5.1.2 was given to the A2A-based collaboration agent. The collaboration agent forwarded the request to the solution agent, which generated the metamodel in Listing 5.

For observability, Langfuse was set up to trace activity of the solution agent and the collaboration agent. Figure 9 shows the screenshot of Langfuse traces across all agents. There are four sub-figures. The top-left sub-figure shows the the collaboration agent making three function calls to communicate with the solution agent, the syntactic supervisor agent, and the semantic supervisor agent. The expanded section of the `_send_to_solution_agent` function is in the top-right sub-figure, showing the tool calling performed by the solution agent. This also

<sup>36</sup><https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/aisp-prototyping/-/tree/main/a2a>

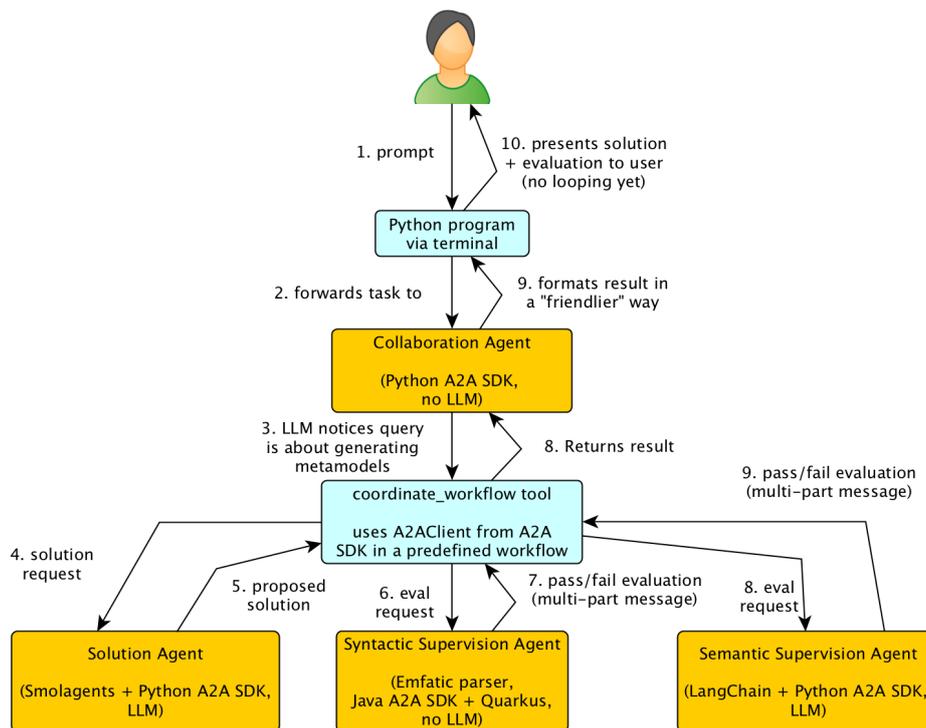


Figure 8: Multi agent system with A2A implementation

shows a full screenshot of a trace, where the left section is a tree of events and the right section displays in detail the LLM input and output messages, along with all the arguments, Parts, and Artifacts.

The implementation leverages the task lifecycle management approach and streaming capabilities in A2A to orchestrate communication and manage task execution between agents. When a user sends a requirement or instruction, the collaboration agent creates a task with a task ID and context ID, which serve as persistent identifiers throughout the entire workflow. Each agent can update the task by generating a *TaskStatusUpdateEvent* object containing state information with TaskStatus object and human-readable message with A2A Message object. The artifact management system utilizes *TaskArtifactUpdateEvent* object to send results between agents with each artifact containing metadata and actual content in multi-part objects. The solution agent produces artifacts containing the generated Emfatic metamodel, and the supervisor agents generate artifacts with validity status and feedback or explanation.

Our streaming leverages the A2A's *SendStreamingMessageRequest* mechanism, enabling asynchronous delivery where the client receives chunks of *SendStreamingMessageResponse* objects in real-time. The collaboration agent acts as an orchestrator, and sends status updates to the user at key milestones: agent discovery, agent invocations, result update, retry attempts, and final completion or failure. In this version of the implementation, there are only streaming updates from collaboration agent to user. However, it is possible to set up other agents to send further updates of their own task progress to the collaboration, in case users are interested in further internal details of agent's handling of task or it takes too long.

The collaboration workflow implements retry logic with configurable maximum attempts, where validation failures trigger regeneration cycles with enriched context from supervisor feedback. Throughout this process, the task ID and context ID remain constant, ensuring correlation of messages and artifacts, and enabling the reconstruction of the complete interaction history.

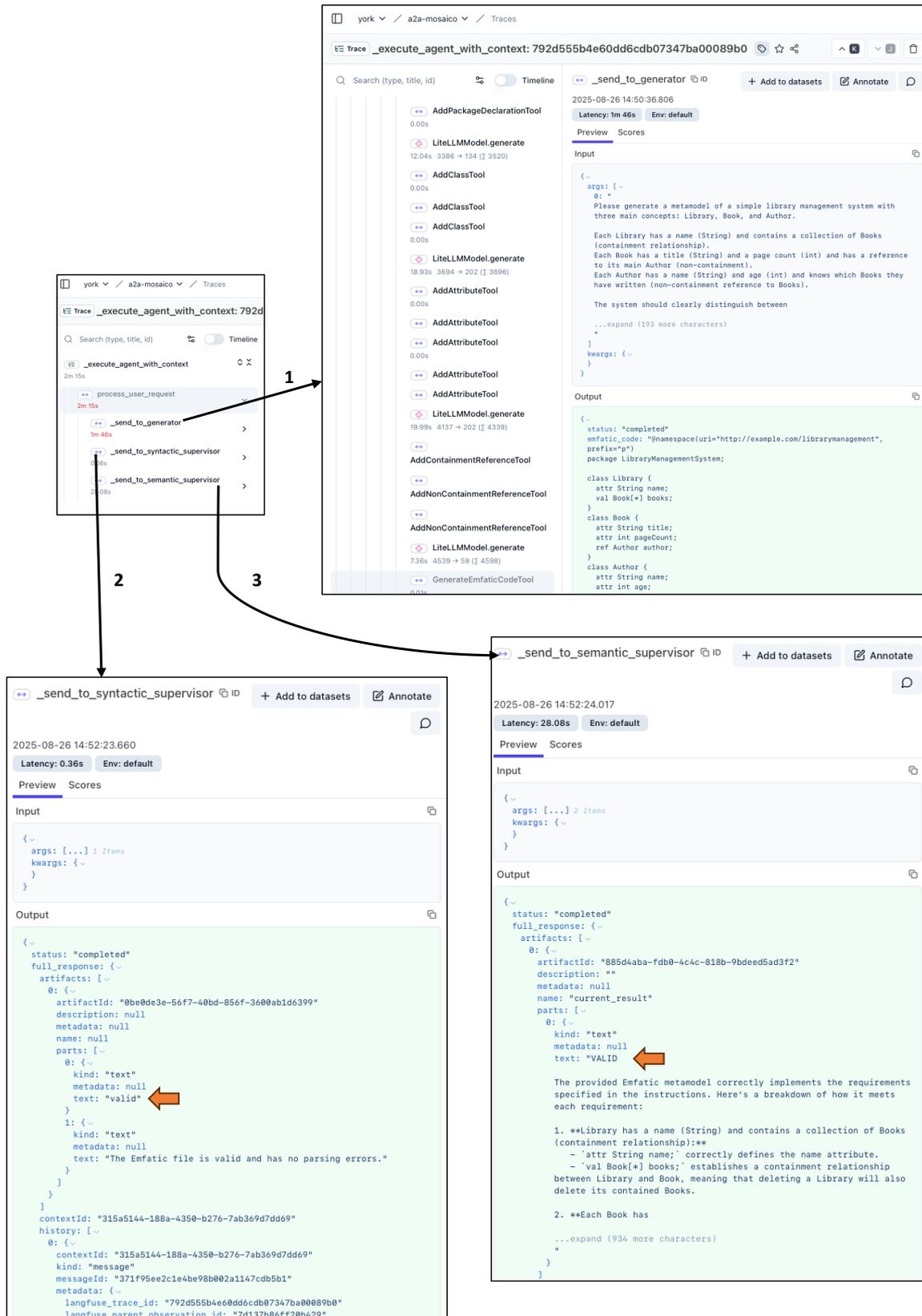


Figure 9: Screenshot of Langfuse observability of the A2A implementation

```

1 @namespace(uri="http://www.example.com/librarymanagement", prefix="p")
2 package LibraryManagement;
3
4 class Library {
5     attr String name;
6     val Book[*] books;
7     val Author[*] authors;
8 }
9 class Book {
10    attr String title;
11    attr int pageCount;
12    ref Author author;
13 }
14 class Author {
15    attr String name;
16    attr int age;
17    ref Book[*] books;
18 }

```

Listing 5: The Emfatic metamodel generated by the A2A solution agent

The final successful completion is signalled through a *TaskStatusUpdateEvent* with state set to completed and final set to true, accompanied by consolidated artifacts containing the validated Emfatic code and comprehensive validation and feedback from all participating agents. This event-driven setup allows real-time visibility in the multi-step process where each agent operates independently while contributing to the shared task context.

### 5.2.3 Requirements evaluation

In this section, the A2A implementation will be evaluated against the nine requirements in Section 2, following the Section 4.2 guidelines. In summary, the A2A protocol is evaluated as meeting 5 requirements (R2, R3, R4, R7, R8), partially meeting 1 requirements (R1), not meeting 2 requirements (R5, R6), and with 1 requirement based on a comparison against our new protocol (R9).

**R1: Agent discoverability** Partially. A2A describe agents using Agent Cards and Agent Skills<sup>37</sup>.

Compared against the discoverability descriptions in Section 2.1, A2A includes identification, skills, message formats, security aspects; and did not include provenance, protocol support, transparency support information. However, A2A does offer extension<sup>38</sup> capability for additional structure and custom logic to be layered on top of the core protocol. In our implementation, four agent servers with relevant agent cards and agent skills were implemented for four agents: a collaboration agent, a solution agent, a syntactic supervisor agent, and a semantic supervisor agent.

**R2: Multi-part and multi-modal messaging** Yes. A2A support multi-part and multi-model messaging because its message support different types of Part<sup>39</sup> objects (text, file, data). In our implementation, text Parts were used to store different relevant information of the task: task status, user instructions, generated metamodel, and validation results.

<sup>37</sup><https://a2a-protocol.org/latest/specification/#55-agentcard-object-structure>

<sup>38</sup><https://a2a-protocol.org/latest/topics/extensions/>

<sup>39</sup><https://a2a-protocol.org/latest/specification/#65-part-union-type>

**R3: Multi-turn conversations** Yes. A2A allows the solution agent to request further clarification on the domain description. An agent can ask for more information for a task by sending a message with input-required<sup>40</sup> state. In our use case, the solution agent can ask for more information if the instructions are unclear. In addition, when the supervisor agents found an issue during validation, the collaboration agent will read the validity status and let the solution agent fix the metamodel by providing the issue found by supervisor.

**R4: Asynchronous and streaming communication** Yes. A2A supports streaming and asynchronous operations<sup>41</sup>: (1) streaming with Server-Sent Events (SSE) and (2) polling with push notifications. In our implementation, the collaboration agent can receive information from other agents and inform user of the progress of the task.

**R5: Agent observability** No. While A2A encourages tracing and observability<sup>42</sup> and supports OpenTelemetry tracing in their SDK, there is no explicit support for a specific platform. Instead, the observability needs to be implemented via callback functions or third-party libraries and frameworks. A2A did not explicitly support super-task IDs for observability of an entire collaboration of agents, either.

Our implementation uses Langfuse to track general progress of the task in the collaboration agent and detailed tool calls made by the solution agent. However, the integration of Langfuse for multi agent collaboration was not simple. Langfuse supports many agentic framework (specifically Smolagents in our case) so it was straightforward to trace specific LLM call. However, when we wanted to trace Python function calls with Langfuse's Observer Decorator, there were many unrelated traces generated by the A2A Python SDK since it also supports OpenTelemetry. Our solution for more control is to use Langfuse's Context Manager. To track traces across agents in a nested structure like showing in Figure 9, we used a `trace_id` and a `parent_span_id` to ensure observations from different agents are recorded in the same trace and under the right level of the nested structure.

**R6: Agent transparency** No. A2A's Opaque Execution principle is to enable collaboration without requiring agents to expose their internal logic, memory, or proprietary tools.

**R7: Interoperability** Yes. A2A is based on HTTP thus supported interoperability<sup>43</sup>. In our case, the syntactic agent was implemented in Java, while the other three are in Python. We also use variety of frameworks: pure Python, smolagents, A2A SDK, and LangChain.

**R8: Access control** Yes. A2A protocol supports authentication<sup>44</sup> and authorisation<sup>45</sup>. Authentication is registered in Agent Card and delegated to standard web mechanisms and established standards (OAuth2 or OpenID Connect). Once a client is authenticated, the authorization logic is handled by the A2A Server.

**R9: Bandwidth-efficient serialisation** Using Wireshark to capture all HTTP traffic to and from the Python and Java servers while the agent collaboration was taking place, it was found that a total of 22 HTTP requests and responses were exchanged (lower than the 28 from our new protocol), adding up to 20,542 bytes (30.7% higher than our new protocol, but still acceptable in absolute terms).

<sup>40</sup><https://a2a-protocol.org/latest/specification/#94-multi-turn-interaction-input-required>

<sup>41</sup><https://a2a-protocol.org/latest/topics/streaming-and-async/>

<sup>42</sup><https://a2a-protocol.org/latest/topics/enterprise-ready/#5-tracing-observability-and-monitoring>

<sup>43</sup><https://a2a-protocol.org/latest/specification/#34-transport-compliance-and-interoperability>

<sup>44</sup><https://a2a-protocol.org/latest/topics/enterprise-ready/#2-authentication>

<sup>45</sup><https://a2a-protocol.org/latest/topics/enterprise-ready/#3-authorization>

## 5.3 Using MCP as a base

In the following, this section explains the implementation of the system described in Section 4.1 using MCP, and evaluates MCP against the requirements in Section 4.2.

### 5.3.1 Implementation

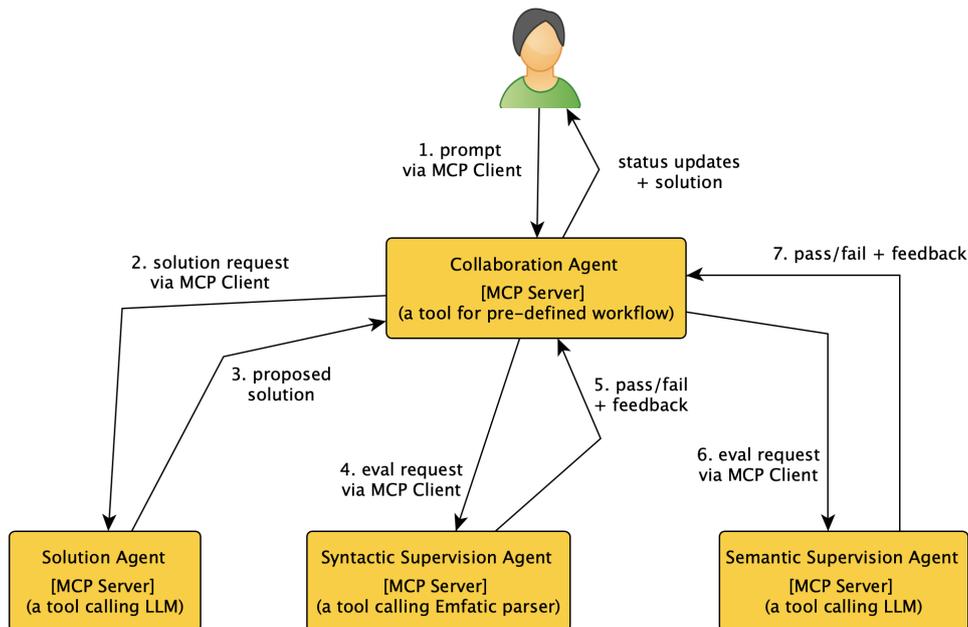


Figure 10: Multi agent system with MCP implementation

The proposed multi-agent system workflow based on MCP is depicted in Figure 10. Each agent is exposed as an MCP server that contains one tool. The tool of an agent carries out the actions of the agent, e.g., the solution agent exposes a tool that calls an LLM for proposing a solution. Therefore, a tool and an agent are semantically equivalent in this case, as a tool is a wrapper of an agent's execution. The user uses an MCP client (e.g., the MCP inspector (Anthropic, 2025)) to connect to the MCP server associated with the collaboration agent, and then calls its tool with a prompt as a parameter for initiating a pre-defined workflow. Then, the collaboration agent uses an MCP client to connect to the MCP server of the solution agent, and calls the tool of the solution agent by passing a solution request as a parameter. The solution agent computes the solution by using an LLM, and then returns the proposed solution to the collaboration agent. The remainder of the workflow is the same as in Section 4.1, with the difference that the agent-to-agent communication entails an MCP client connecting to an MCP server for calling a tool wrapping the agent's behaviour.

The source code of the implemented system based on MCP is available in the MOSAICO repository <sup>46</sup>. The implementation details of the agents are outlined in Table 5. The tools representing the solution agent, the syntactic supervision agent and the semantic supervision agent send only the final solution to the collaboration agent. Intermediate results can be sent via progress and logging notifications through the MCP SDKs (MCP, 2025). However,

<sup>46</sup><https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/aisp-prototyping/-/tree/main/mcp>

Name	Responsibilities	Programming language	LLM	Agentic framework
Collaboration agent	Interact with user, orchestrates agent interactions via MCP	Java	None	Spring AI, MCP Java SDK
Solution agent	Analyse metamodel requirements, and generate Emfatic code	Python	qwen2.5-coder:32b	Smolagents, MCP Python SDK
Syntactic supervision agent	Validate syntax of Emfatic code using a parser	Java	None	Spring AI, MCP Java SDK
Semantic supervision agent	Check whether all user requirements are met	Python	qwen2.5-coder:32b	Smolagents, MCP Python SDK

Table 5: Agents in the MCP implementation

for sending notifications, currently there is a technical incompatibility between the *smolagents* agentic framework and the MCP SDKs. The *smolagents* agent APIs require synchronous callback methods, while the progress and logging notifications sent through the MCP SDKs rely on asynchronous calls. Because of this limitation, and to simplify our protocol evaluation, we chose to have the agents send only the final results of their associated tool, omitting intermediate results, with the exception of the collaboration agent. As such, the collaboration agent sends to the calling MCP client (e.g., the MCP inspector), logging notifications regarding agent invocations, including input and output messages.

### 5.3.2 Demonstration

The orchestration workflow is initiated by the MCP inspector, which represents the MCP client calling the MCP server associated with the collaboration agent, i.e., the tool of the collaboration agent. Figure 11 shows the interface of the MCP inspector, which connects to the MCP server of the collaboration agent via the SSE mode of transport, and then calls the tool named `orchestrate_metamodel_generation`, which executes the behaviour of the collaboration agent. For allowing a fair comparison between the various protocols, the same prompt related to a library metamodel from Section 5.1.2 was passed as input to the called tool.

Langfuse was configured to enable observability, to trace the activity of the collaboration agent and the solution agent. Figure 12 outlines the traces across both agents, highlighting the HTTP requests in the collaboration agent and the LLM calls in the solution agent. The right side of the figure shows the expanded section of the `CodeAgent.run` tool invocation, including the input and a part of the output of the called tool. Note that the HTTP requests traced by Langfuse are not currently linked to specific agents, making it unclear which requests correspond to which MCP server. However, further extending this behaviour is feasible to implement.

Asynchronous intermediate messages are sent through logging. The collaboration agent

```

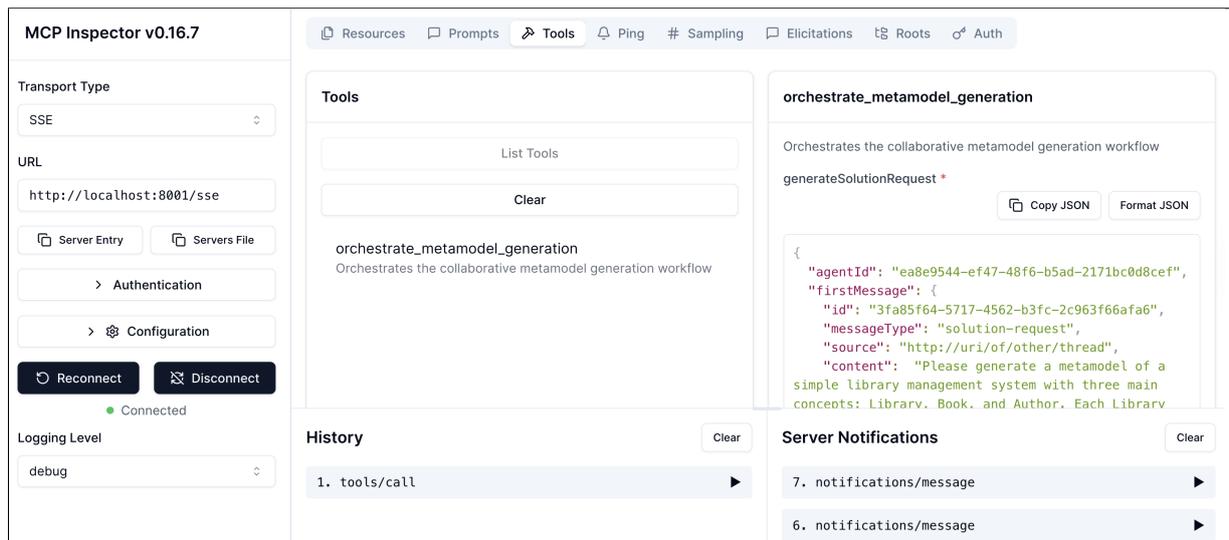
1 @namespace(uri="http://librarymanagement.com", prefix="p")
2 package LibraryManagement;
3
4 class Model {
5     val Library[*] libraries;
6 }
7 class Library {
8     attr String name;
9     val Book[*] books;
10    val Author[*] authors;
11 }
12 class Book {
13     attr String title;
14     attr int pageCount;
15     ref Author author;
16 }
17 class Author {
18     attr String name;
19     attr int age;
20     ref Book[*] books;
21 }

```

Listing 6: The Emfatic metamodel generated by the MCP solution agent

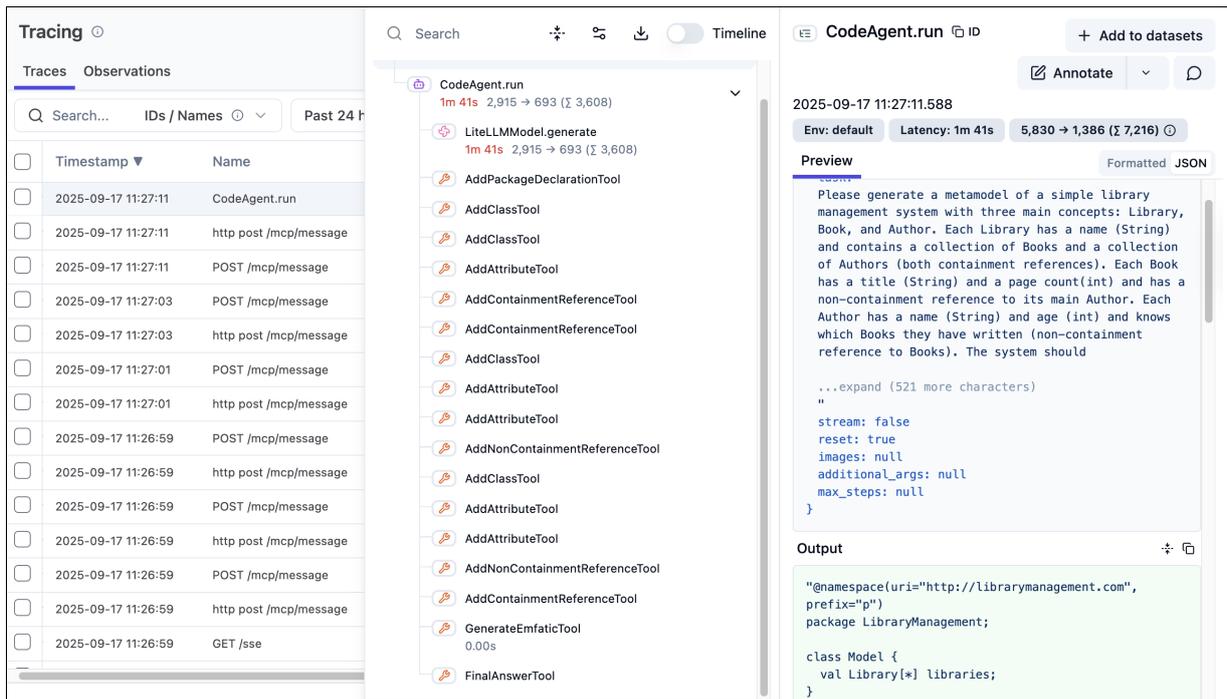
sends logging notifications to the MCP client (i.e., the MCP inspector) regarding agent invocations, including input and output messages. Such notification messages can be seen in the bottom-right side of Figure 11.

Listing 6 represents the final solution outputted by the solution agent, which has been syntactically and semantically validated successfully. Note that the solution in Listing 6 is semantically similar to the solutions of the custom protocol and the A2A protocol.



The screenshot displays the MCP Inspector v0.16.7 interface. On the left, there are settings for Transport Type (SSE), URL (http://localhost:8001/sse), and a 'Connected' status. The main area is divided into 'Tools' and 'History'. The 'Tools' section shows a tool named 'orchestrate\_metamodel\_generation' with a description: 'Orchestrates the collaborative metamodel generation workflow'. Below this, a JSON message is displayed for the 'generateSolutionRequest' field, containing details about the agent and the request content. The 'Server Notifications' section at the bottom right shows a list of notifications, with the most recent being '7. notifications/message' and '6. notifications/message'.

Figure 11: MCP inspector calling the Collaboration Agent



The screenshot displays the Langfuse observability interface for the MCP implementation. It is divided into three main sections:

- Tracing (Left):** A table listing traces with columns for 'Timestamp' and 'Name'. The table shows a series of messages (POST /mcp/message) and a final GET /sse message.
- CodeAgent.run (Center):** A detailed view of a specific trace. It shows a tree of operations:
  - CodeAgent.run (1m 41s, 2,915 → 693 (Σ 3,608))
  - LiteLLMModel.generate (1m 41s, 2,915 → 693 (Σ 3,608))
  - Multiple instances of tools: AddPackageDeclarationTool, AddClassTool, AddAttributeTool, AddContainmentReferenceTool, AddNonContainmentReferenceTool, and GenerateEmfaticTool.
  - FinalAnswerTool (0.00s)
- CodeAgent.run ID (Right):** A detailed view of a specific trace instance (ID: 2025-09-17 11:27:11.588). It shows the environment (default), latency (1m 41s), and token usage (5,830 → 1,386 (Σ 7,216)). The 'Preview' section shows the prompt text, and the 'Output' section shows the generated JSON response:
 

```

"@namespace(uri="http://librarymanagement.com",
prefix="p")
package LibraryManagement;

class Model {
    val Library[*] libraries;
}

```

Figure 12: Screenshot of Langfuse observability of the MCP implementation

### 5.3.3 Requirements evaluation

In this section, the MCP implementation will be evaluated against the nine requirements in Section 2, following the Section 4.2 guidelines. In summary, the MCP protocol is evaluated as meeting 3 requirements (R4, R7, R8), partially meeting 1 requirement (R1), not meeting 4 requirements (R2, R3, R5, R6), and with 1 requirement based on a comparison against our new protocol (R9).

**R1: Agent discoverability** Partially. MCP servers can declare their capabilities, i.e., their resources (e.g., data, tools, prompts), and a client can query servers for their resources. As in our evaluation, we are concerned with agent discoverability, this would specifically refer to tool discoverability, as we consider an agent to be a tool. A client can retrieve the list of tools an MCP server exposes, and then make calls to the tools of interest. Compared against the discoverability descriptions in Section 2.1, MCP includes identification (excluding version), skills (in the case an MCP server contains multiple tools rather than one, while each tool represents a different skill), message formats and security aspects; and it does not include provenance, protocol support and transparency support information.

**R2: Multi-part and multi-modal messaging** No. MCP does not provide built-in multi-part messaging as a formal feature, but it can be implemented using MCP's existing mechanisms (e.g., notifications). For example, large messages can be split into several parts and then sent asynchronously to clients via notifications. Additionally, MCP does not natively support multi-modal messaging. MCP exchanges structured JSON messages between clients and servers, however, one could embed references or encoded content in JSON (e.g., base64-encoded image/audio). In our implementation, only text is exchanged between agents.

- R3: Multi-turn conversations** No. MCP does not support multi-turn conversations by default. Additionally, MCP does not natively provide a *task ID* that can be used by agents, however, a workaround could be implemented by defining tools with a *task ID* parameter. Nevertheless, MCP provides mechanisms (e.g., partial results notifications) that can enable multi-turn conversations between agents. True multi-turn functionality depends on how each MCP server tracks context and orchestrates sequential interactions. Therefore, MCP itself does not impose how memory or context is stored, and the multi-turn conversations capability relies on servers implementing context tracking (e.g., in memory, or an external resource).
- R4: Asynchronous and streaming communication** Yes. MCP supports asynchronous request/response handling and event notifications by design, letting clients and servers communicate concurrently without blocking each other. This is enabled by the SSE and Streamable HTTP modes of transport between MCP servers and clients. When an MCP server's tools are added, removed, or updated, it can notify connected clients regarding those changes. These updates are delivered as JSON-RPC 2.0 notifications, allowing MCP servers to keep clients updated in real time. Streaming communication is handled through partial results that are sent by tools as progress updates or logging notifications.
- R5: Agent observability** No. The MCP SDKs do not provide built-in support for tracing, observability or OpenTelemetry. However, observability can be implemented via callback functions or third-party support. Examples of providing MCP tracing include using integrations with Phoenix (Arize AI, 2025), MCPcat (MCPcat, 2025) and Dynatrace (Dynatrace, 2025). In addition, MCP does not natively provide a *super-task ID*, however, a workaround could be implemented by defining tools with a *super-task ID* parameter.
- R6: Agent transparency** No. MCP is opaque by default, as it does not expose internal details, logic, and resources, without them being explicitly made available in the form of resources. If a client wishes to retrieve certain internal data, the server must implement a tool or resource that returns that information. For example, to allow clients to fetch memory data, a tool called *getRecentMemory()* should be implemented on the MCP server to return recent memory entries.
- R7: Interoperability** Yes. MCP is designed to provide interoperability between agents (i.e., tools and clients). MCP provides a transport-agnostic design, as it defines a protocol for messages, tools, resources, notifications, and streaming, but it does not mandate a specific transport layer (e.g., STDIO, SSE, Streamable HTTP). There are nine official MCP SDKs (MCP, 2025), corresponding to nine programming languages, thus heterogeneous systems can communicate via MCP. In our case, the collaboration agent was implemented in Java, whereas the solution agent was implemented in Python, using the MCP SDKs.
- R8: Access control** Yes. MCP supports access control via server-side mechanisms<sup>47</sup>, but the protocol itself does not enforce it. Access control must be explicitly implemented by the MCP server. The MCP SDKs provide several APIs<sup>48</sup> for implementing authentication and authorization between MCP servers and clients.
- R9: Bandwidth-efficient serialisation** Using Wireshark, all HTTP traffic was captured, to and from the Python and Java servers, while the agent collaboration was taking place. It was

<sup>47</sup><https://modelcontextprotocol.io/specification/draft/basic/authorization>

<sup>48</sup><https://github.com/modelcontextprotocol/python-sdk?tab=readme-ov-file#authentication>

found that a total of 23 HTTP requests and responses were exchanged (higher than the 22 from the A2A protocol), adding up to 8,189 bytes (60.1% lower than the A2A protocol). It is worth noting that the MCP-based system does not send intermediate messages for LLM callbacks, therefore, the smaller total request size was expected.

## 6 Conclusion and next steps

In this section, we will summarise the strengths and weaknesses of each option, and conclude the report with a general conclusion on the direction that further development of the MOSAICO AISP should take.

### 6.1 New protocol

**Strengths:** The new protocol designed within MOSAICO is concisely described in a single OpenAPI specification, making it simple to generate the base client and server code by reusing existing code generators. The new protocol fully meets 7 of the 9 requirements in Section 2, including multi-part, multi-turn, and asynchronous communication. It is the only of the three options which explicitly supports having a super-task ID to enable cross-collaboration observability (R5). Meanwhile, implementing a server for this protocol does not entail significant effort, making it simpler to support new programming languages in the future should it be necessary: existing frameworks such as FastAPI (for Python) and Micronaut (for Java) were used in our experiment, without needing significant changes.

**Weaknesses:** R1 (“Agent discoverability”) would need to support additional metadata (e.g., taxonomy classification, pending the completion of the taxonomy in WP2), but this information is simple to add as MOSAICO controls the OpenAPI specification of the protocol. R6 (“Agent transparency”) has been postponed for further discussion within the consortium over the course of the project, as the tradeoff between transparency, privacy, and intellectual property protection is defined during the further development of the use cases. Besides these two requirements, the main weakness in this new protocol is that it would require significant effort to gain market adoption compared to existing options (MCP and A2A) which have already built up a community.

**Overall:** The new protocol is concise, interoperable, and already meets most of the needs of MOSAICO. The protocol could continue to be customised throughout the project to meet future needs. However, it does not have the community adoption that A2A and MCP already benefit from, and their resulting wealth of resources in terms of documentation and software development kits. MOSAICO adopters who already have working A2A-based agents would need to develop adapters or re-engineer their agents if the protocol were to be adopted.

### 6.2 A2A

**Strengths:** The A2A protocol stands out for several notable strengths in enabling secure and flexible multi-agent collaboration. Firstly, A2A provides extensive support for agent discoverability (R1) through Agent Card and Agent Skill, as well as extension capabilities for additional structure or custom logic. One of its prominent advantage is interoperability (R7): allowing seamless communication between agents built using different frameworks or by different organizations. This capacity is critical for enterprises seeking modular AI systems. Furthermore, A2A facilitates task hand-offs between agents while preserving the *opacity* of each agent’s internal state (R6). Sensitive workflow logic, intellectual property, and proprietary data remain

protected, making the protocol attractive for regulated industries and privacy-focused businesses. Another technical merit is its robust support for message and rich data modalities (R2). A2A has Messages and Artifacts to distinguish between LLM-generated content and everything else. Within Artifacts, there can be Parts that can handle not just textual data, but also structured forms, media files, audio, and video to accommodating a wide range of real-world interaction needs. The protocol also includes features for managing asynchronous, long-running workflows, offering real-time streaming of events and state changes (R3, R4). Its open-source ecosystem, supported by many technology partners and major corporations, signals growing industry endorsement and confidence in the protocol's direction.

**Weaknesses:** Despite these strengths, the current limitations of the A2A protocol are notable. Its ecosystem remains relatively early-stage, with breaking changes across early versions and limited production-grade large-scale adoption. In addition, A2A offer limited observability support (R5) with minimal implementation for OpenTelemetry and no super-task ID for observability management across multi-agent collaborations. Lastly, there are also security limitations: A2A leaves security as an orthogonal concern to be dealt via HTTP headers, and requires developers to implement proper security measures, such as authorization, validation and sanitization of external data in LLM messages (R8).

**Overall:** While A2A is aligned with MOSAICO's goals and technically advanced, broader impact will depend on further improvements in reliability, user experience, and the maturing of the surrounding toolchain and community support. A2A will require several extensions to fully meet the MOSAICO requirements: fortunately, A2A has extensibility mechanisms built into its design, with the ability to attach additional information to messages, and describe these extensions in its Agent Cards.

## 6.3 MCP

**Strengths:** The MCP protocol demonstrates several strengths as a general-purpose protocol designed for structured and secure communication between agents and external resources. Its plug-and-play architecture makes it flexible, enabling seamless integration of diverse tools and data sources. This adaptability also extends to interoperability, as MCP can operate across programming languages, agentic frameworks, and LLM vendors, allowing heterogeneous systems to communicate effectively. Another advantage of MCP is its built-in support for asynchronous communication through event notifications and streaming, which makes it appropriate for long-running or complex tasks. In addition, MCP offers mechanisms for access control and authentication, providing reliable security measures when resources need to be shared across different environments.

**Weaknesses:** However, MCP also has weaknesses in areas that are critical for more advanced multi-agent interactions. Its discovery capabilities do not natively provide information regarding provenance and semantic versioning. Transparency is constrained, as MCP does not, by default, expose details about decision-making processes or internal states, leaving this responsibility to custom implementations. Support for multi-part and multi-modal communication is also incomplete: the protocol primarily handles structured JSON messages, and while non-textual data can be embedded or referenced, this relies on workarounds. Multi-turn conversations have a partial gap, as MCP lacks native mechanisms for managing context across multiple exchanges; instead, MCP servers must implement their own methods for context tracking and memory. In addition, MCP does not include native support for observability and monitoring, however, tracing and logging can be added externally through SDKs or third-party integrations.

**Overall:** MCP provides a strong and adaptable foundation that excels in interoperability, asynchronous communication and access control. Its limited discoverability, transparency, and

Requirement	New protocol	A2A	MCP
R1: Agent discoverability	~	~	~
R2: Multi-part and multi-modal messaging	✓	✓	×
R3: Multi-turn conversations	✓	✓	×
R4: Asynchronous and streaming communication	✓	✓	✓
R5: Agent observability	✓	×	×
R6: Agent transparency	×	×	×
R7: Interoperability	✓	✓	✓
R8: Access control	✓	✓	✓
R9: Bandwidth-efficient serialisation	15kB	20kB	8kB

Table 6: Requirement evaluation summary: ✓ is met, × is not met, ~ is partially met. For R9, the amount of bandwidth used by each multi-agent collaboration is displayed.

observability, with incomplete support for multi-turn conversations, multi-part and multi-modal messaging, indicate that additional layers or complementary mechanisms are needed to enable richer forms of collaboration between agents.

## 6.4 Overall conclusion

Table 6 summarises the general findings from evaluating each protocol over the same multi-agent collaboration. It can be seen that the new protocol and A2A cover most of the requirements. All protocols would need further extension to meet R1, and only our new protocol explicitly covers R5 (cross-collaboration observability) with its support for specifying a super-task ID. As mentioned before, R6 will be further fleshed out as the industrial use cases develop.

On the other hand, MCP does not meet many of the needs from MOSAICO: while it will be useful to provide agents with the tools they need to complete their tasks, it would need significant extensions to be usable as a way to communicate between agents.

Two possible directions lie ahead: either building a community from scratch around the new protocol, or proposing a number of MOSAICO-led extensions on the A2A protocol and leveraging its existing community, documentation, software development kits, and agents. To maximise the potential reach of the MOSAICO Platform, our current direction for Task 1.2 (the creation of the Minimal Viable Product versions of the reference client and server) will be to extend A2A to intercommunicate agents (while contributing to their existing community), and to reuse MCP within specific agents to provide them with the ability to integrate with enterprise information systems, the MOSAICO Agent Repository, and IDEs. Our long-term goal is to engage with the A2A specification developers and contribute any MOSAICO extensions to the main specification whenever possible.

## References

- ANP Team (2025), *Agent Network Protocol website*, URL: <https://www.agent-network-protocol.com/> (visited on 07/16/2025). [Cited on page 19]
- Anthropic (2025), *MCP inspector*, URL: <https://modelcontextprotocol.io/legacy/tools/inspector>. [Cited on page 37]
- Arize AI (2025), *Phoenix MCP Tracing*, URL: [https://arize.com/docs/phoenix/integrations/model-context-protocol/mcp-tracing?utm\\_source=chatgpt.com](https://arize.com/docs/phoenix/integrations/model-context-protocol/mcp-tracing?utm_source=chatgpt.com) (visited on 07/16/2025). [Cited on page 41]
- BeeAI (2025), *Agent Communication Protocol documentation*, URL: <https://agentcommunicationprotocol.dev/introduction/welcome> (visited on 07/16/2025). [Cited on page 19]
- Duminuco, Alessandro, Giovanni Conte, Jeffrey Napper, and Marco Trinelli (2025), *Agent Connect Protocol specification*, URL: <https://github.com/agntcy/acp-spec> (visited on 07/16/2025). [Cited on page 19]
- Dynatrace (2025), *OpenTelemetry Instrumentation for MCP*, URL: <https://github.com/theharithsa/opentelemetry-instrumentation-mcp> (visited on 07/16/2025). [Cited on page 41]
- Eclipse Foundation (2025), *Eclipse LMOS protocol website*, URL: [https://eclipse.dev/lmos/docs/lmos\\_protocol/introduction/](https://eclipse.dev/lmos/docs/lmos_protocol/introduction/) (visited on 07/16/2025). [Cited on page 19]
- Ehtesham, Abul, Aditi Singh, Gaurav Kumar Gupta, and Saket Kumar (May 2025), *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:2505.02279 [cs], DOI: 10.48550/arXiv.2505.02279, URL: <http://arxiv.org/abs/2505.02279> (visited on 07/16/2025). [Cited on pages 19, 22]
- Gupta, Akash, Ivaxi Sheth, Vyas Raina, Mark Gales, and Mario Fritz (Oct. 2024), *LLM Task Interference: An Initial Study on the Impact of Task-Switch in Conversational History*, arXiv:2402.18216 [cs], DOI: 10.48550/arXiv.2402.18216. [Cited on page 9]
- He, Junda, Christoph Treude, and David Lo (Dec. 2024), *LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead*, arXiv:2404.04834 [cs], DOI: 10.48550/arXiv.2404.04834. [Cited on page 9]
- Li, Qiaomu and Ying Xie (2025), "From Glue-Code to Protocols: A Critical Analysis of A2A and MCP Integration for Scalable Agent Systems", in: *arXiv preprint arXiv:2505.03864*. [Cited on page 22]
- Linux Foundation (2025), *Agent2Agent (A2A) Protocol Github project*, URL: <https://github.com/a2aproject/A2A> (visited on 07/16/2025). [Cited on page 19]
- MCP (2025), *Official SDKs*, URL: <https://modelcontextprotocol.io/docs/sdk> (visited on 07/16/2025). [Cited on pages 37, 41]
- MCPcat (2025), *MCP OpenTelemetry*, URL: [https://mcpcat.io/guides/monitor-mcp-performance-opentelemetry/?utm\\_source=chatgpt.com](https://mcpcat.io/guides/monitor-mcp-performance-opentelemetry/?utm_source=chatgpt.com) (visited on 07/16/2025). [Cited on page 41]
- Spahr-Summers, Justin and David Soria Parra (2025), *Model Context Protocol website*, URL: <https://modelcontextprotocol.io/> (visited on 07/16/2025). [Cited on page 19]
- Steinberg, Dave, Frank Budinsky, Marcelo Paternostro, and Ed Merks (Dec. 2008), *EMF: Eclipse Modeling Framework*, 2nd ed., Eclipse Series, Addison-Wesley Professional, ISBN: 978-0-321-33188-5. [Cited on page 23]
- Yang, Yingxuan, Huacan Chai, Yuanyi Song, Siyuan Qi, Muning Wen, Ning Li, Junwei Liao, Haoyi Hu, Jianghao Lin, Gaowei Chang, Weiwen Liu, Ying Wen, Yong Yu, and Weinan Zhang (June 2025), *A Survey of AI Agent Protocols*, arXiv:2504.16736 [cs], DOI: 10.48550/arXiv.2504.16736, URL: <http://arxiv.org/abs/2504.16736> (visited on 07/16/2025). [Cited on pages 19, 22]

Yao, Shunyu, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao (May 2023), “ReAct: Synergizing Reasoning and Acting in Language Models”, en, *in: Proceedings of ICLR 2023*, Kigali, Rwanda, ISBN: 978-1-71389-925-9, URL: <https://par.nsf.gov/biblio/10451467-react-synergizing-reasoning-acting-language-models> (visited on 07/17/2025). [Cited on page 24]