

MOSAICO

D1.3: Version 2 of client and server

September, 2026

DOCUMENT STATUS: SUBMITTED (Pending review and approval from the European Commission)

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.

MOSAICO

D1.3 Version 2 of client and server

Work package	WP1 AI-agent server protocol
Task	T1.3 Support for agent-directed information retrieval
Task contributors	IMT, UDA
Due date	30/09/2026
Submission date	30/09/2026
Type of deliverable	Demonstrator
Dissemination Level	Public
Deliverable lead	University of York
Version	1.3.0
Authors	University of York
Reviewers	UDA, IMT, ECL
Keywords	Network protocols, agent-to-agent communication, multi-agent systems

Document Revision History

Version	Date	Description of change	List of contributor(s)
v1.0.0	01/09/2026	Initial version	University of York
v1.1.0	09/09/2026	Revised version based on ECL feedback	University of York
v1.2.0	15/09/2026	Restructured version based on UDA feedback	University of York
v1.3.0	22/09/2026	Document Helm chart and update MCP Inspector figure	University of York
v1.4.0	29/09/2026	Revised version based on IMT feedback	University of York

Partners

























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, 2026

This deliverable contains original unpublished work except where clearly indicated otherwise.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 report provides a high-level description of the second version of the demonstrator of the MOSAICO platform, and the reference client and server of the MOSAICO AI-agent Server Protocol (AISP). As outlined in D1.1, after considering several alternatives (creating a brand-new protocol from scratch, or basing AISP as a customisation of the increasingly popular MCP and A2A protocols), it was decided to design AISP as a profile of the A2A protocol.

AISP is defined as a set of constraints on the use of A2A by the various types of MOSAICO agents, and as a set of extensions on top of standard A2A that cover the gaps needed to achieve the requirements in MOSAICO (e.g. cross-collaboration observability, and structured collaborations among agents).

D1.3 deliverable's main contribution is the completion of the client-side tool calling extension, which allows downstream MOSAICO agents to request the execution of tools that run locally on the client (e.g. reading or editing a file in the user's workspace): the report defines its wire format, its use of the A2A *input-required* task state to pause and resume a task while a tool call is outstanding, and its implementation in the MOSAICO Base Agent library, which gives any LangGraph-based agent the full request/response loop with no additional code beyond a system prompt.

The demonstrator consists of a Visual Studio Code extension which acts as the reference client, and an orchestration of Docker container images which implement the MOSAICO repository (from WP2), and a first set of AI agents implementing AISP-based servers (from WP1). The agents include two supervision agents, a collaboration agent, and five solution agents. The reference client itself has been rewritten from scratch as a small, purpose-built extension (V2), replacing the original fork of the discontinued Continue project and adding native support for the client-side tool calling extension.

The report provides a high-level description of the architecture of the current version of the demonstrator, a discussion of the Large Language Models (LLMs) that it has been tested against (both cloud and local LLMs), and the current hardware requirements (which vary based on the use of local LLMs). The rest of the report is dedicated to providing instructions on the use of the demonstrator by three audiences:

- System administrators, who are responsible for deploying and maintaining the MOSAICO infrastructure, can use the included Docker Compose and Helm scripts to deploy the demonstrator on-premises, and experiment with its configuration.
- End-users, who only need to download and install the provided Visual Studio Code extension, and point it to the address of the reference agent. From then on, they can chat to the MOSAICO platform as any other LLM.
- Agent developers, who are provided with indications on the additional constraints and extensions introduced by AISP on top of standard A2A, and instructions about how their agents should be packaged. The report includes indications about the adjustments needed to combine the demonstrator with agents launched from local development environments, and instructions on how to experiment with the REST API and MCP server provided by the MOSAICO repository.

Table of Contents

1 Introduction . . . 10

2 Architecture . . . 11
2.1 MOSAICO components . . . 11
2.2 The MOSAICO Reference Client V2 . . . 13
2.2.1 Extension architecture . . . 13
2.2.2 Agent communication . . . 14
2.2.3 Client-side tool calling . . . 14
2.2.4 User-facing features . . . 14
2.3 The MOSAICO Base Agent library . . . 15
2.4 Supported LLMs . . . 15
2.5 Hardware requirements . . . 16

3 Instructions for administrators . . . 16
3.1 Software requirements . . . 17
3.2 Initial setup . . . 17
3.3 Pulling the embedding model . . . 18
3.4 Optional: running local LLMs . . . 18
3.4.1 Via LM Studio (recommended) . . . 18
3.4.2 Via Ollama in Docker . . . 19
3.5 Starting the demonstrator . . . 19
3.6 Container orchestration . . . 20
3.7 Automated agent registration . . . 21
3.8 Shutting down the demonstrator . . . 21
3.9 Helm script for Kubernetes . . . 22
3.9.1 Layout . . . 23
3.9.2 Adding a task agent . . . 23
3.9.3 Feature flags (`values.features`) . . . 24
3.9.4 Installation . . . 24

4 Instructions for users . . . 24
4.1 Installing the reference client . . . 24
4.2 Opening the client . . . 25
4.3 Initial configuration . . . 25
4.3.1 Setting the URL to the reference agent . . . 26
4.3.2 Tool permissions . . . 26
4.4 Using the reference client . . . 27
4.4.1 Attaching files . . . 27
4.4.2 History management . . . 27

5 Instructions for developers . . . 27
5.1 Communication protocols . . . 28
5.2 Packaging . . . 28
5.3 The MOSAICO Base Agent library . . . 29
5.3.1 Using the Library . . . 29
5.3.2 Base Agent Executor Classes . . . 30
5.3.3 Testing Support . . . 31
5.3.4 Agent Definition . . . 32

5.3.5 Example Base Agents . . . 33
5.4 Requirements per agent type . . . 34
5.4.1 Reference agent . . . 34
5.4.2 Collaboration agent . . . 35
5.4.3 Solution agent . . . 35
5.4.4 Supervision agent . . . 36
5.5 AISP-specific extension: observability metadata . . . 36
5.6 AISP-specific extension: client-side tool calling . . . 37
5.6.1 Extension declaration . . . 37
5.6.2 Wire format . . . 38
5.6.3 Wire format summary for client-side tool calling . . . 40
5.6.4 Implementation in the MOSAICO base agent . . . 40
5.7 Running agents from sources . . . 44
5.8 MOSAICO Repository interfaces . . . 44
5.8.1 REST API . . . 44
5.8.2 MCP server . . . 45

6 Conclusions . . . 45

List of Figures

1 Current architecture of the MOSAICO platform 12
2 Accessing the “Install from VSIX” option from VS Code 25
3 Screenshot of the MOSAICO Reference Client V2 main view 25
4 Screenshot of the “MOSAICO Settings” dialog configured of the reference client V2 26
5 The `docstring-agent` adding docstrings to `calculator.py` through client-side tools, showing an accepted `edit_existing_file` call (green) and a rejected one (red) 43
6 Screenshot of the Swagger UI interface for the REST API of the MOSAICO repository 44
7 Screenshot of the settings required on the Anthropic MCP Inspector to connect to the MOSAICO repository’s MCP server 45

List of Tables

1 Correspondence between Docker Compose constructs and their Helm equivalents. 22
2 Wire format summary for client-side tool use. 41

Abbreviations

A2A	Agent-to-Agent
AISP	AI-agent Server Protocol
API	Application Programming Interface
BDI	Belief-Desire-Intention
GGUF	GPT-Generated Unified Format
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LLM	Large Language Model
MCP	Model Context Protocol
RAG	Retrieval-Augmented Generation
REST	REpresentational State Transfer
SDK	Software Development Kit
TTL	Time-to-Live
UUID	Universally Unique IDentifier
VRAM	Video Random Access Memory

1 Introduction

This document is a companion report for the MOSAICO demonstrator repository[1] and the MOSAICO base agent library repository[2] hosted in the Eclipse Gitlab server, which are the main artifacts for deliverable D1.3.

The report documents the current architecture and design of the MOSAICO platform, which includes components developed in WP1, WP2, and WP3 (Section 2), and provides instructions for server administrators (Section 3), end users (Section 4), and developers (Section 5). Section 6 closes the report with a summary of the current status of the work.

The content of the report is based on the documentation maintained in the demonstrator and base agent library repositories, which will be updated beyond the submission of this deliverable, based on the evolution of the platform.

This report on D1.3 supersedes D1.2, and documents the platform as a whole rather than only the parts that have changed. For a reader familiar with D1.2, the following list summarises what is new in this version for D1.3, and where each item is described.

- **Agent-directed information retrieval, through a client-side tool calling AISP extension.** This is the objective of T1.3 and the main technical contribution of this deliverable. Agents can now request the execution of tools that run on the user's machine, which lets an agent retrieve the information it needs from the workspace instead of working only from what was attached to the prompt. The extension, its wire format, and its implementation are described in Section 5.6; the user-visible behaviour is described in Section 2.2.3.
- **A rewritten reference client (version 2 or V2 from now on).** The original client was a fork of the now-discontinued Continue project. It has been replaced by a streamlined, purpose-built VS Code extension maintained by the MOSAICO consortium, with native support for the tool calling extension. See Section 2.2.
- **The MOSAICO Base Agent library.** A Python library that implements the AISP profile, observability, and the client-side tool calling loop once, so that a new agent inherits them rather than reimplementing them. It is introduced in Section 2.3 and documented for developers in Section 5.3.
- **A `docstring-agent` demonstrating the extension end to end.** A solution agent built on the Base Agent library that reads a source file through a client-side tool, generates docstrings for it, and writes the result back through another tool. It is listed with the other agents in Section 2.1, and used as the worked example in Section 5.6.
- **A "standalone agents" deployment mode.** The Compose orchestration can now drop the agents' dependency on a locally deployed MOSAICO repository, so that a set of agents can be brought up against a repository already running on another machine. See Section 3.6.
- **RustFS in place of MinIO** for the S3-compatible blob store used by Langfuse, as MinIO was no longer maintained and MinIO had complex AGPLv3 licensing terms. See Section 2.1.

[1] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-demonstrator
[2] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-base-agents

2 Architecture

This section describes the current structure of the MOSAICO platform, and the requirements for running the platform.

2.1 MOSAICO components

Figure 1 shows a high-level outline of the current architecture of the MOSAICO platform. The *MOSAICO reference client*[3] is implemented as a *Visual Studio Code extension*.

This provider performs A2A-based communication with the *MOSAICO reference agent*[4]. Figure 1 describes the deployment in which the reference agent runs on the user's own machine, next to the reference client inside VS Code. This is not a requirement of the architecture: the reference agent may equally run on a remote host, with the client simply configured to point at its URL (Section 4.3).

The reference agent identifies whether the request needs to be handled by one of the MOSAICO agents, and if so, queries the *MOSAICO repository* for appropriate agents through its MCP API. The agents have been registered in advance, and are part of the same subnet dedicated to agents (*Agent Network*).

This second version of the demonstrator includes:

- `aisp-mini-swe-agent`[5]: An agent dedicated to generic software engineering tasks. This agent is an example of the work required to take a pre-existing agent and integrate it with the MOSAICO platform: the original agent was developed by the SWE-Bench project[6].
- `IP Solution Agent`: An agent dedicated to finding the open source licenses of a given project, contributed by ECL.
- `Emfatic metamodel collaboration agent`: A collaboration agent developed by WP3 using their SysML-based orchestration language[7], for generating Emfatic[8] metamodels according to the description of a domain, which orchestrates three lower-level agents:
  - `Emfatic solution agent`: A Smolagents[9]-based agent for producing Emfatic source code,
  - `Emfatic syntactic supervisor`: A syntactic supervision agent that checks for valid Emfatic syntax by reusing the existing Emfatic parser,
  - `Emfatic semantic supervisor`: A semantic supervision agent that uses an LLM to check the metamodel against the requirements indicated in the description provided by the user.
- `PlantUML Agent`: An agent dedicated to derive a PlantUML textual notation and/or a natural-language explanation of the UML diagram's content.

[3] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-reference-client-v2
[4] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-reference-agent/
[5] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/aisp-mini-swe-agent
[6] https://github.com/SWE-agent/mini-swe-agent
[7] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-orchestrator-agents/-/tree/main/server/metamodelman?ref_type=heads
[8] https://eclipse.dev/emfatic/
[9] https://huggingface.co/docs/smolagents/index

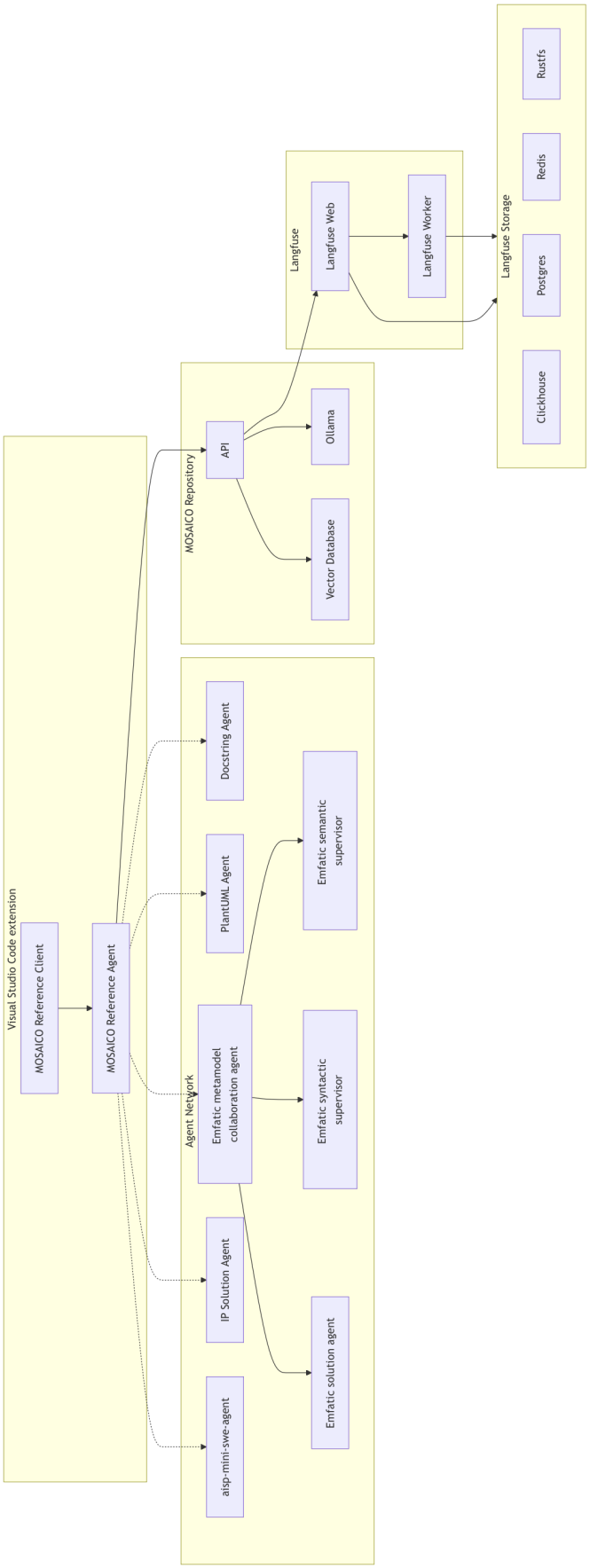


Figure 1: Current architecture of the MOSAICO platform

- `Docstring agent`: An agent dedicated to generate code documentation for existing source code.

The MOSAICO repository includes its own *Vector Database* to allow for searching agents by semantic proximity to a given task description, and uses a bundled *Ollama* server to compute embeddings from the given descriptions.

The agents send observability traces to a *Langfuse*[10] server, which is also used by the MOSAICO repository to collect performance data. The traces allow the administrators of the platform to monitor that the agents are behaving as expected. For readability, Figure 1 omits the flows from every agent to Langfuse because an arrow from each of them would obscure the rest of the architecture. Langfuse's architecture is documented in depth on their official website[11]. Regardless, it is largely made-up of:

- *Langfuse Web*: A web interface for navigating traces, datasets, and experiments.
- *Langfuse Worker*: One or more workers for processing background jobs from a queue.
- *Langfuse Storage*: A storage layer, combining transactional records (*Postgres*), analytics (*Clickhouse*), an event queue (*Redis*), and an S3-compatible blob store (*RustFS*).

Note: the first version of the demonstrator (in D1.2) used MinIO for S3-compatible storage, but this has been replaced with RustFS (with no loss of functionality) as MinIO was no longer being maintained, and also had a strong-copyleft license with complex ramifications (AGPLv3). RustFS offers the same functionality, is under active development, and is under the permissive Apache License 2.0 license.

2.2 The MOSAICO Reference Client V2

The first version of the MOSAICO Reference Client was built as a fork of Continue[12], an open-source AI coding assistant for VS Code. Development of the upstream Continue project has since been stopped as their company was acquired by Cursor[13], which left the fork carrying a large amount of inherited functionality that the MOSAICO use case did not need.

The MOSAICO Reference Client V2 is a rewrite from scratch, entirely in favour of a streamlined, purpose-built VS Code extension maintained directly by the MOSAICO consortium under the Eclipse Foundation (published as `mosaico-reference-client-v2` on Open VSX Registry[14], MIT licensed). The rewrite trades some of the breadth Continue offered for a smaller maintenance surface, tighter alignment with MOSAICO's own extension points, and independence from a stalled upstream project.

2.2.1 Extension architecture

Where v1 inherited Continue's general-purpose, multi-provider architecture, V2 is designed around a single responsibility: presenting a native sidebar chat interface that talks directly to a MOSAICO Reference Agent over A2A, with no intermediate LLM-provider abstraction layer

[10] https://langfuse.com/
[11] https://langfuse.com/handbook/product-engineering/architecture
[12] https://github.com/continuedev/continue
[13] https://web.archive.org/web/20260902034011/https://continue.dev/
[14] https://open-vsx.org/extension/MOSAICO/mosaico-reference-client-v2

of its own. The extension is written in TypeScript, and is bundled with esbuild into a single ES module (`out/extension.js`), keeping the packaged VSIX lightweight. The codebase is small and flatly organised: an `extension.ts` entry point registers a webview view in the activity bar; `chatProvider.ts` is the core orchestrator that owns the webview, the conversation state, and the streaming logic; `mosaico-a2a-client.ts` wraps the official `@a2a-js/sdk` client and translates the client's internal chat history into A2A message parts; `settings.ts` reads and writes user configuration; `markdown.ts` handles rendering; and a `tools/` module implements client-side tool calling.

2.2.2 Agent communication

The client connects to a MOSAICO Reference Agent via its A2A agent card and streams responses over `sendMessageStream`. Each turn is serialised into A2A "data" parts carrying the role, content, and a lightweight heuristic (`isSWE`) that flags whether the turn was previously classified by the reference agent's LLM as a software-engineering task, alongside any attached files as raw parts. The streaming loop in `chatProvider.ts` consumes `message`, `task`, `statusUpdate`, and `artifactUpdate` events, incrementally rendering assistant text, task-state transitions, and returned artifacts into the chat view, with a per-request timeout and a user-triggered cancellation path that can interrupt an in-flight stream at any point.

2.2.3 Client-side tool calling

A notable addition in V2 is a structured, extensible tool-calling framework absent from the original fork's simpler model. Tools are defined against a common *Tool* interface (name, description, JSON Schema parameters, and a default execution policy) and registered centrally in a *ToolRegistry*, so that adding a new capability only requires implementing one file and registering it. Two tools ship today as shown in Figure 4 (page 26): `read_file`, which returns the contents of a workspace file (capped at 256 KB) and defaults to running automatically, and `edit_existing_file` (also capped at 256 KB), which overwrites an existing file with a complete new version supplied by the agent and defaults to requiring user approval.

Each tool carries a per-tool execution policy of "automatic", "ask", or "excluded", configurable through the settings UI, and the set of non-excluded tools is advertised to the agent using OpenAI-style function-calling definitions carried under a MOSAICO-specific A2A extension URI. When a tool call requires approval, the client renders an inline card in the chat with the proposed arguments rather than a blocking modal, and the agent loop supports up to a maximum number of consecutive tool round-trips per turn (default to sixteen) as a safeguard against runaway tool use.

2.2.4 User-facing features

The webview provides offline Markdown rendering with syntax highlighting (via `marked` and `highlight.js`, bundled locally rather than fetched from a CDN), so the client works without external network access beyond the connection to the reference agent itself. Conversation history is persisted locally per workspace using VS Code's `workspaceState`, allowing users to browse, resume, retry, edit, or delete previous turns and switch between saved sessions.

Users can attach files by drag-and-drop or file picker; the extension sniffs MIME types from file contents (via `file-type`) with a filename-based fallback, and returned artifacts are rendered with inline actions to insert code at the cursor, copy it, or save it as a new file

in the workspace. All local configuration, including the reference agent's URL (defaulting to `http://localhost:4000`) and tool policies, is stored in a plain JSON settings file under `~/.mosaico-v2/settings.json`.

2.3 The MOSAICO Base Agent library

Rather than being written against the raw A2A SDK, several of the agents in the demonstrator are built on the *MOSAICO Base Agent library*, a Python package that packages the AISP profile itself into a reusable form: an agent that extends one of its executor classes obtains, without writing any additional code, the conventions that AISP requires of every MOSAICO agent.

Concretely, the library supplies:

- **The AISP profile of A2A**: the agent card, the A2A request handler, the task store, and the `/health` endpoint expected of every packaged MOSAICO agent (Section 5.2).
- **Observability**: extraction of the observability metadata from incoming requests and construction of the corresponding Langfuse trace hierarchy, so that an agent's activity appears in the right place in a multi-agent collaboration with no per-agent wiring.
- **Client-side tool calling**: a complete implementation of the request/response loop for the extension described in Section 5.6, which a LangGraph-based agent inherits by supplying only a system prompt.
- **Testing support**: base test classes that mock the A2A event bus and LLM responses, so that agent behaviour can be tested without a live LLM.

Of the agents listed in Section 2.1, the `aisp-mini-swe-agent`, the `PlantUML Agent` and the `Docstring agent` are built on this library. The `IP Solution Agent` and the four Emfatic metamodel agents implement AISP directly instead, and there are plans to migrate further agents onto the library as it matures.

For a developer, the library is a convenient way to build agents that conform to the AISP standard, and is documented in full with worked examples in Section 5.3.

2.4 Supported LLMs

The current version of the demonstrator can use both local and cloud models, through OpenAI-compatible endpoints.

The demonstrator has been tested with:

- `gemma4:26b` and `devstral-small-2` running locally on Ollama[15] or LM Studio[16]. Local models are important to test for some of the MOSAICO partners, as they may have sensitive data that must be processed on-premises. Gemma 4 includes vision support, which is required by one of the sample agents (`plantuml-agent`).
- `qwen3:32b` and `qwen3-coder:30b` running locally on Ollama or LM Studio. These were the local LLMs used for testing until the release of `devstral-small-2`.
- `glm-5.2` running on OpenRouter.
- `gpt-5-mini` and `o4-mini` running on Microsoft Azure AI Foundry[17]. This provider was

[15] https://ollama.com/
[16] https://lmstudio.ai/
[17] https://azure.microsoft.com/en-us/products/ai-foundry

selected for testing, as it can provide guarantees about the region used to run the models: both models use the Data Zone Standard (EUR) deployment type, which ensures the data processing is done from EU datacenters.

- `gpt-4o-mini` and `gpt-4o` running on OpenAI.
- `openai/gpt-oss-120b` running on Groq.
- `minimaxai/minimax-m3` provided by NVIDIA NIM.

Further testing is planned with other LLMs, particularly smaller LLMs that can run on consumer hardware, or which may be available via cloud "router" services such as Groq, Github Models, or OpenRouter. Any LLMs to test in the future should meet these requirements:

- Support structured outputs and tool calling: the reference agent uses structured outputs for input classification.
- Must be strong at coding tasks (e.g. generating Python and Bash scripts):
  - The metamodelling solution agent generates and runs Python scripts internally.
  - The mini-swe-agent generates and runs Bash commands internally.

The MOSAICO repository uses internally the `nomic-embed-text` embedding model to search for relevant agents using natural language.

2.5 Hardware requirements

When fully started, the containers in the demonstrator currently use 6GB of RAM, and the images all together occupy 22GB. For that reason, we recommend running the demonstrator in a system with at least 16GB of RAM and 50GB of available disk space.

If you are using Docker Desktop:

- On Mac, please make sure the "Disk usage limit" in your Resources settings is set accordingly, and that there is available space.
- On Windows, please check that the "WSL Settings" allow for sufficient memory and disk space.

For running the Devstral Small 2 LLM locally, we recommend having 16GiB of VRAM: it may work with less VRAM, but it will be slower.

3 Instructions for administrators

This section provides instructions aimed at server administrators that need to set up the platform in the relevant server infrastructure. The platform requires some initial setup before it can be started, and it can be later managed using some utility scripts. The section concludes with a high-level description of the various Docker Compose scripts that orchestrate the Docker containers that comprise the platform.

An alternative deployment approach using Kubernetes via a Helm script is documented in Section 3.9.



Listing 1: Commands for cloning the MOSAICO demonstrator and generating the base configuration files

```
HOST=https://gitlab.eclipse.org
REPO_PATH=/eclipse-research-labs/mosaico-project/mosaico-demonstrator.git
MOSAICO_TAG=v0.2.0
git clone "${HOST}${REPO_PATH}"
cd mosaico-demonstrator
git checkout "${MOSAICO_TAG}"
./00-copy-env.sh
```

3.1 Software requirements

To run this demonstrator, the following software is required:

- A terminal that can run Bash scripts: this is available out-of-the-box for Linux/Mac, and for Windows, Git Bash[18] can be used.
- A Docker Compose installation, whether as part of the Docker Engine[19] (only for Linux), or as part of Docker Desktop[20] (for Windows/Linux/Mac).

3.2 Initial setup

First, clone the demonstrator repository from our Gitlab project, and check out the most recent tagged release. For instance, as of the time of writing (22 September 2026), the `v0.2.0` tag is the most recent one. The demonstrator includes a number of support scripts to help automate the various steps. The next step is copying the environment configuration templates for editing. The required commands are shown on Listing 1.

The `00-copy-env.sh` script will create an `env` folder with the appropriate configuration variables:

- `langfuse.env`: settings for Langfuse. You should not need to customise this: the script will have automatically generated random Langfuse keys for you.
- `llm.env`: settings for the sample agents.
  - You will need to set `API_BASE` and `API_KEY` to point to your OpenAI-compatible LLM API, and check that the `_MODEL` variables later in the file use the correct identifier for the LLMs to be used.
  - For running local LLMs, please see the optional sections below.
  - There is a commented-out set of suggestions for models that are readily available via OpenAI / Azure AI Foundry.
- `compose.env`: settings for the Compose-based container orchestration.
  - If you would like to use your own installation of Ollama, set `WITH_OLLAMA` to 0. If `WITH_OLLAMA` is set to 1 (the default), an Ollama container will be started. Ollama is required for the embedding model used in the MOSAICO repository.

[18] https://git-scm.com/install/windows
[19] https://docs.docker.com/engine/install/
[20] https://docs.docker.com/desktop/

- If you have an NVIDIA discrete GPU on Linux/Windows and you set `WITH_OLLAMA` to 1, you can set `OLLAMA_USES_GPUS` to 1 to expose your GPU to the Ollama container, for better performance.
- If you would like to try MOSAICO from your browser, set `WITH_THEIA` to 1, which will start a distribution of Theia with the reference client pre-installed.
- As an example of the flexibility in running agents locally or remotely, there is a setting called `REGISTER_MSWEA_AS_DOCKER` that can be set to 1 so that the `mini-swe-agent` will be registered as an agent to be pulled and run locally by the reference client via the end-user's own Docker installation. If kept as 0, the agent will be managed instead by this demonstrator, and be treated as a regular remote agent by the reference client.
- If you would like to only run the included agents and not the rest of the MOSAICO platform, set `STANDALONE_AGENTS` to 1. This can be used to run the agents in a different machine from the MOSAICO repository and its dependencies, improving security and horizontal scalability.

3.3 Pulling the embedding model

Pull the `nomic-embed-text` model into your Ollama installation:

```
ollama pull nomic-embed-text
```

If you enabled `WITH_OLLAMA` in `compose.env`, the Docker Compose setup will do this automatically for you.

3.4 Optional: running local LLMs

This section explains how to run local LLMs with the demonstrator, for a testing / staging environment.

3.4.1 Via LM Studio (recommended)

For testing the demonstrator with local LLMs, we recommend using a local installation of LM Studio, as it offers convenient options to make efficient use of available hardware. The Devstral Small 2 model is 15GB in size, so it will run significantly faster in systems with at least 16GiB of VRAM, but it is possible to run with less VRAM (with lower token throughput). After installing LM Studio, please follow these steps:

- Ensure the local server is running - it can be made to start automatically on boot.
- Download the Devstral Small 2 model (tested with 'Q4_K_S' quantisation in GGUF format, and with '4bit' quantisation in MLX format).

The demonstrator is configured by default to use the locally running LM Studio server on port 1234, assuming the above steps have been followed.

3.4.2 Via Ollama in Docker

Instead of LM Studio, in a Linux or Windows environment with a discrete NVIDIA GPU, it is possible to run an Ollama server via GPU and expose the GPU to it. Note that on Linux, you will need to install the NVIDIA Container Toolkit first.

Note: we have found this option to be more resource-intensive on Windows systems, so we typically recommend the above LM Studio option instead.

To do so, first enable `WITH_OLLAMA` and `OLLAMA_USES_GPUS` in your `env/compose.env` file. Then, run these commands:

```
./01-compose.sh up ollama -d
./98-pull-models.sh
```

You will also need to adjust `API_BASE` in the `env/llm.env` file. Example configuration lines are included for this option: please comment and uncomment lines as needed.

Note that this use of Ollama via Docker has not been tested with Intel or AMD GPUs, and will require manufacturer-specific customisations to the Compose files. For AMD GPUs, we would advise consulting the AMD Container Runtime Toolkit documentation.

The context window size is 4096 by default. This can be customised via the variable called `OLLAMA_CONTEXT_LENGTH` in `llm.env`. For suggestions on values depending on available VRAM, please consult the Ollama documentation.

3.5 Starting the demonstrator

Once the settings have been saved, the demonstrator can be started by running:

```
./01-compose.sh up -d
```

The `01-compose.sh` script prepares and runs a Docker Compose stack by loading default and user-specific environment settings, detecting the host IP address when needed, and selecting the appropriate compose override files based on feature flags. Please note that if you change the configuration in Compose files, you must run the `./01-compose.sh up -d` command to have Compose apply the changes.

This will pull all necessary images from the MOSAICO Gitlab project[21]. After some time, the Compose file should have:

- Started the Langfuse observability platform at `http://localhost:3000/`: the default username and password are `admin@mosaico.local` and `mosaico2025`.
- Started the MOSAICO repository at `http://localhost:8080/`.
- Started the MOSAICO sample agents, and registered them into the MOSAICO repository. You can verify their registrations by checking the /api/agents endpoint, or querying the MCP server (Section 5.8.2 on page 45).

It is possible to quickly check if all services have correctly started, by running this command from a Bash terminal, and checking all services are healthy:

```
docker ps
```

[21] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project

3.6 Container orchestration

Except for the reference client, most of the demonstrator is structured as an orchestration of Docker containers, written as a set of Docker Compose files in the included `compose` folder. For maintainability, the Compose configuration is broken up into multiple files inside the 'compose' directory, which are layered as needed.

The layers below are mandatory (except in the "standalone agents" mode, introduced in V2 and described further below):

- `docker-compose.repository.yml` provides a local Langfuse installation, as well as an instance of the MOSAICO repository which will list the available agents for this deployment.
- `docker-compose.task_agents.yml` is the base layer for the sample agents that support various software engineering tasks.

The layers below are optional, and are enabled or disabled depending on the values set in `env/compose.env`:

- `docker-compose.mswea_docker.yml`: sets up the `mini-swe-agent` as a local agent (to be downloaded and run locally by the reference agent).
- `docker-compose.mswea_endpoint.yml`: sets up the `mini-swe-agent` as a remote agent (to be called by the reference agent over the network).
- `docker-compose.ollama.yml`: bundles an Ollama server, which should always contain at least the embedding model required for searching agents via natural language descriptions.
- `docker-compose.ollama_with_gpus.yml`: exposes the NVIDIA GPU to the Ollama container.
- `docker-compose.ollama_repository.yml`: adds a dependency from the MOSAICO repository to Ollama being healthy (ensuring it does not start until the embedding model has been pulled).
- `docker-compose.pgadmin.yml`: provides a web UI for administering the Postgres databases in the MOSAICO repository.
- `docker-compose.ref_agent.yml`: runs the reference agent from within Compose.
- `docker-compose.theia.yml`: runs the Theia web IDE with the pre-installed reference client within Compose.
- `docker-compose.with_mm_collab.yml`: runs the metamodel collaboration agent from Compose.
- `docker-compose.without_mm_collab.yml`: exposes the metamodel supervision and solution agents from ports in the host system, assuming the metamodel collaboration agent is running from the host system. Mutually exclusive with its 'with' alternative.

To improve maintainability, a number of services in the Compose files are defined by extending services in the `base-definitions.yml` file.

Since version 2, the demonstrator supports a "standalone agents" mode where it drops the dependency of the agents on the MOSAICO repository (assuming it is already running in a different machine). This is enabled by setting the `STANDALONE_AGENTS` variable in `compose.env` to 1. This is implemented by applying additional `docker-compose.standalone*.yml` layers that remove those dependencies.

3.7 Automated agent registration

Agents must be registered into the MOSAICO repository so that the reference agent may find them. This is automatically managed by the Compose scripts (Section 3.6), using a dedicated Python script in the `docker/agent-registrations` folder. The `register-agent.py` script is run with no arguments, and takes its inputs from the following mandatory environment variables:

- `REPOSITORY_API`: base URL of the MOSAICO repository. Usually will only contain protocol, host, and port: for example, `http://mosaico-app:8080`.
- `AGENT_JSON`: path to the JSON file with the information to be sent to register the agent (except for the agent's name and A2A card URL).
- `AGENT_NAME`: name of the agent to be registered. Overrides the agent name in the JSON file, and is meant to be unique across the MOSAICO repository.
- Exactly one of these two:
  - `AGENT_CARD_URL`: URL of the A2A card for the remote agent. Overrides the A2A card URL in the JSON file.
  - `AGENT_DOCKER_IMAGE`: Docker image reference for the local agent. Overrides the Docker image reference in the JSON file.

If the agent is already registered, the script will query the MOSAICO repository and automatically detect if there are any fields in the local JSON that are different from those of the agent with the same name in the MOSAICO repository. If so, it will issue a PUT request to update the details of that agent. Otherwise, it will issue a POST request to add the agent to the MOSAICO repository.

3.8 Shutting down the demonstrator

To shut down the demonstrator, you can run:

```
./01-compose.sh stop
```

If you would like to remove the containers and network from your system:

```
./01-compose.sh down
```

To remove the storage volumes as well (including any pulled models in Ollama), and delete all images:

```
./01-compose.sh down --volumes --rmi all
```



3.9 Helm script for Kubernetes

Since several of the project partners preferred Kubernetes for their on-premises deployments, a Helm chart has been developed based on the Docker Compose setup. The rest of this section describes the Helm chart: the latest version of this documentation is kept on Gitlab[22].

The `01-compose.sh` orchestration logic (conditional `-f` file inclusion based on `compose.env` flags) has been translated into feature flags under `.Values.features`. The mapping from the Compose concepts to the Helm chart is documented on Table 1.

Table 1: Correspondence between Docker Compose constructs and their Helm equivalents.

Compose construct	**Helm equivalent**
`01-compose.sh` flag → extra `-f` file	`values.features.*` gating `{{- if }}` blocks
`compose.env` booleans	`values.features`
`llm.env`, `langfuse.env`	`values.llm`, `values.langfuse` + Secrets/-ConfigMaps
`base-definitions.yml extends` / YAML anchors	named templates in `_helpers.tpl`
`depends_on: service_healthy`	readiness probes; registration Jobs poll the repository first
`*-registration` services	`post-install/post-upgrade` hook Jobs
`00-copy-env.sh` key generation	`lookup`-backed Secret (generated once, preserved on upgrade)
bind mounts (`../docker/*`)	ConfigMaps from `files/`
`DEFAULT_TASK_AGENT_HOST`, LAN IP detection	in-cluster Service DNS names
`gpus: all`	`resources.limits.nvidia.com/gpu`
stateful services + named volumes	StatefulSets + volumeClaimTemplates
`rustfs-init`, `service_completed_successfully`	initContainer on both langfuse pods

[22] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-demonstrator/-/blob/main/helm/README.md

Compose construct	Helm equivalent
host port publishing for agent cards	ClusterIP Services; cards use Service DNS

3.9.1 Layout

The layout of the chart uses one component per file. _-prefixed files are partials only, as Go templates do not allow `define` inside an `if` block, so shared `env` blocks must live there. The `templates` folder is organised as follows:

- `_helpers.tpl` — naming, labels, shared env/probe/PVC partials
- `secrets.yaml` — langfuse keys (generated once), LLM key, datastore creds
- `configmaps.yaml` — `files/` mounted into pods
- `infra/` — postgres, pgvector, clickhouse, redis, rustfs, pgadmin
- `langfuse/` — `_langfuse.tpl` (shared env) + web/worker Deployments
- `repository/` — mosaico-app
- `agents/` — `_agent.tpl` + task agents, emfatic agents, reference agent
- `registration/` — `_regjob.tpl` + post-install registration Jobs
- `ollama/` — embedding Ollama Service + StatefulSet
- `theia/` — Theia IDE Service + Deployment
- `ingress/` — single Ingress resource for the host-based routes

3.9.2 Adding a task agent

Everything a task agent needs is in `mosaico.agent`; `AGENT_CARD_HOST`, `AGENT_CARD_PORT`, `PORT`, the Service and the `/health` probes are derived from `name` and `port`:

```
{{ include "mosaico.agent" (dict
  "ctx" $ctx
  "name" "my-agent"
  "image" .Values.agents.myAgent.image
  "model" .Values.llm.models.myAgent
  "extraEnv" (list (dict "name" "FOO" "value" "bar"))
) }}
```

Registration follows the same shape via `mosaico.regJob`, which builds the A2A card URL from `agentService`.

3.9.3 Feature flags (`values.features`)

These correspond to the Docker Compose settings in the `compose.env` file.

- `withOllama`: deploy the embedding Ollama StatefulSet and point the repository at it.
- `ollamaUsesGpus`: request a GPU for Ollama
- `withMmCollabAgent`: run the Emfatic. collaboration agent (and its registration) in-cluster
- `withReferenceAgent`: deploy the reference agent + its ingress route.
- `withPgadmin`: deploy pgAdmin (dev only).
- `standaloneAgents`: agents only; skip repository + Langfuse + their dependencies.
- `withTheia`: deploy the Theia IDE + its ingress route.
- `registerMsweaAsDocker`: `REGISTER_MSWEA_AS_DOCKER`: register mini-swe-agent in DOCKER mode and do not deploy it. This is not fully supported yet: nothing in the chart gives the reference agent a container runtime to start that image with. Should be left off unless one has been arranged manually.

3.9.4 Installation

To install the chart, these commands can be used:

```
helm lint ./helm
helm install mosaico ./helm --timeout 20m \
  --set ingress.hosts.langfuse=langfuse.example.com \
  --set ingress.hosts.theia=theia.example.com \
  --set ingress.hosts.referenceClient=reference.example.com
```

`--timeout` must be raised well above the 5m default on a first install, as Helm blocks on the post-install registration jobs, which wait for the repository, which waits on pgvector, which waits on a freshly provisioned volume.

LLM endpoint/models and datastore passwords should be overridden via `--set` or a values file. For `standaloneAgents`, `repository.apiUrl` should be set to point to the external repository.

4 Instructions for users

This section is aimed at end users that wish to use the MOSAICO platform after all relevant servers have been started via the included Compose file.

4.1 Installing the reference client

The MOSAICO reference client is available as a Microsoft Visual Studio Code extension. If using VS Code, the latest `.vsix` file can be downloaded from our Gitlab package registry.

If using an OpenVSX-enabled IDE (such as VS Codium or Theia), the extension can be directly found by searching in its extensions tab, as it is listed in OpenVSX.

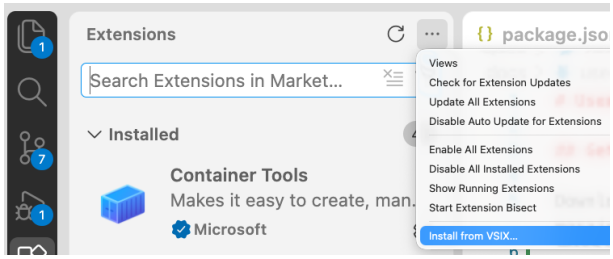


Figure 2: Accessing the “Install from VSIX” option from VS Code

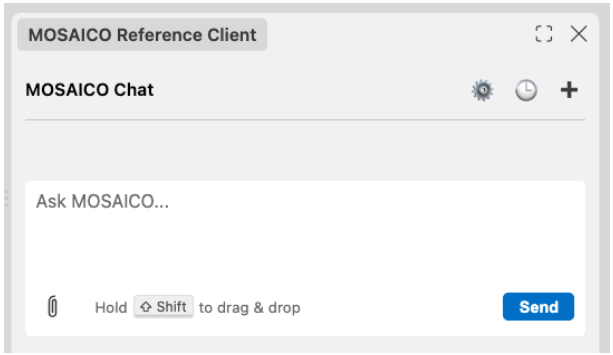


Figure 3: Screenshot of the MOSAICO Reference Client V2 main view

You can then use the “Install from VSIX” option in the VS Code Extensions tab to manually install the extension, as shown on Figure 2. After a few seconds, VS Code should report that the extension has been successfully activated.

4.2 Opening the client

You can then show the MOSAICO reference client view in one of these ways:

- Selecting the “View - Open View” menu item, and entering “MOSAICO” to find the “MOSAICO Reference Client” option to be selected.
- Entering Ctrl+Shift+P (Win/Linux) or Shift + Command + P (Mac) to access the VS Code Command Palette, and selecting “MOSAICO Reference Client: Focus on Agent Chat View”.

The MOSAICO reference client will then open, as shown in Figure 3.

4.3 Initial configuration

You can change a number of options by clicking the **Settings** (gear) icon inside the chat interface, which will show the dialog from Figure 4. Configuration is saved locally to the `.mosaico-v2` subfolder of your personal profile (`HOME` in Unix-based environments). The rest of this section explains the various available settings.

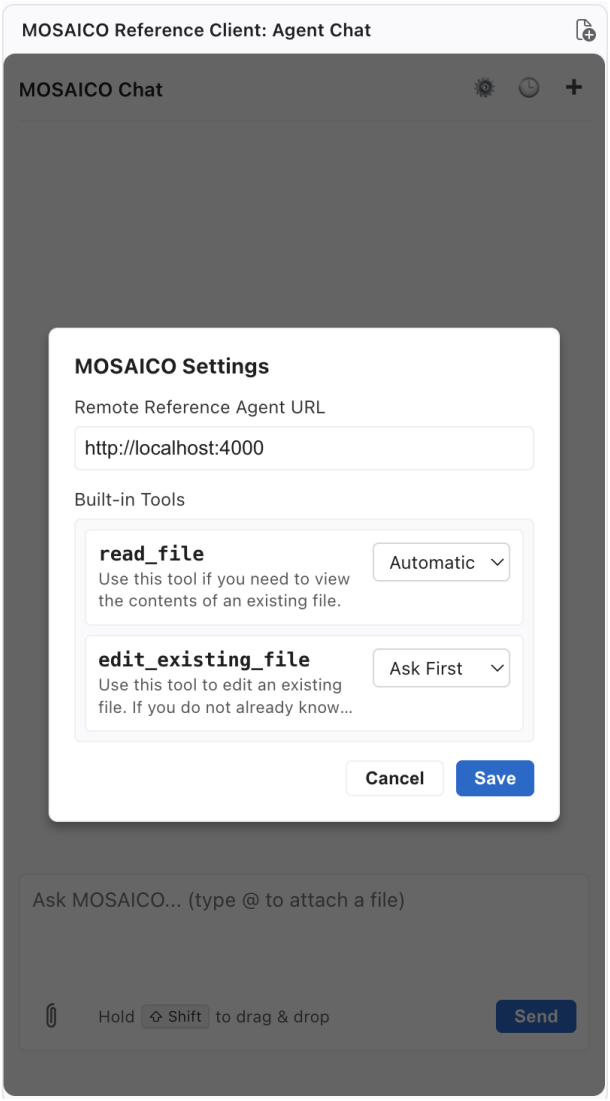


Figure 4: Screenshot of the “MOSAICO Settings” dialog configured of the reference client V2

4.3.1 Setting the URL to the reference agent

By default, the client assumes the reference agent is running locally at `http://localhost:4000`. If you are using a remote reference agent, you will have to change this setting accordingly.

4.3.2 Tool permissions

The MOSAICO client exposes certain tools to the invoked agents: currently, these tools are limited to reading and editing local files.

Each tool can be set to one of these options:

- **Automatic**: use without asking for permission.
- **Ask First**: ask the user first for permission.
- **Excluded**: do not expose to agents, and reject any requests for its use.

4.4 Using the reference client

When the reference agent receives a prompt, it will follow these steps:

1. Classify the interaction as related to software engineering tasks, or not.
2. If it is not classified as related to software engineering, it will answer with its own LLM.
3. Otherwise, it will:
   (a) Search in the repository for a suitable agent.
   (b) Delegate the task to the relevant agent.
   (c) Reply back with the result from the agent: you can expand the “Thinking” box to receive regular updates on the current status of the task.

Example conversations can be found in the `samples` folder of the Git repository for the demonstrator. Further sample conversations will be added as more agents are included in the demonstrator in the future.

You can also visit the local Langfuse server to check the traces of any agent invocations you have made. The level of detail for the traces will depend on the selected agents’ support for traceability: the MOSAICO Base Agent library in Section 5.3 provides a foundation that makes agents traceable from the start.

4.4.1 Attaching files

To attach files from outside your workspace, click on the clip icon and select them through your operating system’s file picker. Alternatively, you can hold Shift and drag-and-drop them into the chat area.

This also works for attaching files from your workspace, but since version 0.5.0, there are a few convenient shortcuts for that:

- You can attach the currently opened file in the editor by right-clicking on the editor tab and selecting “Add to MOSAICO Chat”.
- Within the prompt box in the MOSAICO chat, you can type ‘@name-of-file’ and select it among the displayed candidates.
- From anywhere in the IDE, you can press Ctrl+Alt+a (or Cmd+Alt+a in MacOS) to select files in your workspace. The command is also available as a button in the top right of the “MOSAICO Reference Client” view.

4.4.2 History management

The **History** (clock) button allows you to switch to a past conversation or delete old conversations. To start a new conversation, click on the **New Chat** (plus sign) button.

5 Instructions for developers

This section provides details that are relevant to developers wishing to create agents for the MOSAICO platform.

5.1 Communication protocols

After considering various protocol design approaches in D1.1, we decided to design the MOSAICO AISP as a profile of the A2A Protocol.

As of September 1st 2026, the latest published version of A2A is v1.0.0. We aim to maintain AISP on top of the latest stable version of A2A, once it has broad support over the selected base A2A SDKs (JavaScript, Python, and Java).

On top of base A2A, we have developed a super-task ID extension for cross-collaboration traceability (Section 5.5), and an agent communication language extension (with its own Git repository) with "tell", "achieve", and "ask" performatives. D1.3 introduces another extension to expose client-side tools to A2A agents (e.g. so they can request to create a file, or read the contents of a file from the user): further details are described in Section 5.6.

Agents are normally expected (but not required) to:

- Support observability by integrating the Langfuse SDK where possible, or supporting the export of their traces to Langfuse's OpenTelemetry endpoint otherwise.
- Use the LiteLLM abstraction library to support a variety of LLM providers, with at least support for OpenAI-compatible endpoints.

Besides the extensions, AISP sets out some general requirements on how to use A2A for the various types of MOSAICO agents: please see Section 5.4 on page 34.

The MOSAICO Base Agent library (Section 5.3) provides the reference implementation of the AISP protocols and also supplies the support for observability and client-side tool-calling.

5.2 Packaging

MOSAICO agents are distributed as Docker images, which are expected to run an A2A server on startup. They must be able to accept the following environment variables:

- `API_BASE`: base API of the API endpoint for the LLM, if required by the agent.
- `API_KEY`: token to access the above mentioned LLM endpoint.
- `LANGFUSE_HOST`: base URI of the Langfuse server (e.g. `http://langfuse:3000`).
- `LANGFUSE_PUBLIC_KEY` and `LANGFUSE_SECRET_KEY`: tokens to be used to publish traces to a Langfuse project.
- `HOST` and `PORT`: host and port that the A2A server should be bound to.
  - If `HOST` is unspecified, it should default to `0.0.0.0`.
  - If `PORT` is unspecified, agents may fall back to their own default value.
- `AGENT_CARD_HOST` and `AGENT_CARD_PORT`: host and port that the A2A server should advertise in its A2A agent card. If unspecified, these should default to `localhost` and the value of `PORT`, respectively.

The A2A server must have at least:

- At least one A2A endpoint (whether using JSON-RPC, REST, or gRPC).

- `/.well-known/agent-card.json` endpoint with the A2A agent card corresponding to their A2A endpoint.
- `/health` endpoint returning an HTTP 200 status code if the agent is healthy, or 5xx if the agent is unhealthy: for example, the agent is unable to access the LLM, or the Langfuse server.

Finally, MOSAICO agents are currently assumed to be already running and available from the network through a given URL, using a predefined LLM and its appropriate credentials. Future versions of the platform will allow users to run local agents from their machines on the fly (as Docker containers), using their configured LLM credentials.

5.3 The MOSAICO Base Agent library

The MOSAICO Base Agent library is a Python library designed to simplify the development of AISP-compatible agents for the MOSAICO platform. The library is publicly available through the Eclipse GitLab package registry. It provides support for AISP, together with reusable facilities for testing and observability. The library's main components are: base agent executor classes, agent definition abstractions, classes providing testing support, and abstractions for processing observability information.

5.3.1 Using the Library

The MOSAICO Base Agent library can be included as a dependency of a Python project by adding `mosaico-base-agent` to the project's `pyproject.toml` file:

```
dependencies = [
    "mosaico-base-agent"
]
```

The library provides optional functionality through dependency extras, which can be included according to the requirements of the agent being developed. For example, an LLM-enabled agent can include the `llm` extra:

```
dependencies = [
    "mosaico-base-agent[llm]"
]
```

As the library is hosted in the Eclipse GitLab package registry, `uv` must also be configured to obtain the dependency from this registry. This can be achieved by executing the following command from the root directory of the project containing the `pyproject.toml` file:

```
uv add mosaico-base-agent --index \
https://gitlab.eclipse.org/api/v4/projects/12942/packages/pypi/simple
```

The particular dependency extra required depends on the type of base executor used by the agent. In addition to the `llm` extra for LLM-enabled agents, the `langchain` extra is required when using the LangGraph-based executor, while the `acl` extra is required when using the BDI-based executor, as detailed in Section 5.3.2.2.

5.3.2 Base Agent Executor Classes

The library provides four base agent executor classes for implementing different types of agents: *MosaicoAgentExecutor*, *LLMAgentExecutor*, *LGReactAgentExecutor*, and *BDIAgentExecutor*. These provide progressively more specialised facilities for implementing generic MOSAICO agents, LLM-enabled agents, LangGraph ReAct agents, and BDI-based agents, respectively. All base agent executor classes inherit, either directly or indirectly, from *AgentExecutor*, and consequently conform to the *AgentExecutor* type. This abstract base class defines the `execute` method for executing the agent's logic for a given request context and the `cancel` method for requesting the cancellation of an ongoing task.

5.3.2.1 MosaicoAgentExecutor

MosaicoAgentExecutor represents the most rudimentary form of a MOSAICO agent and provides observability support by default. Its `execute` method handles the execution of an incoming request, extracts observability metadata from the request, and configures a Langfuse trace hierarchy for the agent execution.

The class also provides the `send_message` and `send_text_message` methods for sending messages to the A2A event queue. The former supports sending a message, while the latter provides corresponding functionality for sending a textual message. Concrete agents based on *MosaicoAgentExecutor* are expected to override the `execute_agent` method to define their behaviour. A typical implementation has the following form:

```
@override
async def execute_agent(
        self,
        context: RequestContext,
        event_queue: EventQueue,
        observability_metadata: Optional[MosaicoObservabilityMetadata]) -> None:
    # execute agent's business logic
```

Subclasses of *MosaicoAgentExecutor* are also expected to implement the `health` method, which performs a health check for the agent. For example, an implementation can return `HEALTH_OK` when the agent is functioning correctly. The Arithmetic Agent (see Section 5.3.5) provided with the library constitutes an example of extending *MosaicoAgentExecutor*.

5.3.2.2 BDIAgentExecutor

BDIAgentExecutor provides a base for developing agents that handle performatives in their incoming and outgoing messages. It extends *MosaicoAgentExecutor* and therefore provides observability by default.

The class provides the `execute_bdi` method, which serves a similar purpose to `execute` but additionally receives a parameter specifying the performative associated with the incoming message. The method `send_bdi_message` corresponds to `send_message` but allows a performative to be specified for the outgoing message, while `send_text_bdi_message` provides equivalent functionality for textual messages.

Projects using *BDIAgentExecutor* must include the `acl` dependency extra. The Basic BDI Agent (see Section 5.3.5) provided with the library demonstrates the use of *BDIAgentExecutor*.



5.3.2.3 LLMAgentExecutor

LLMAgentExecutor provides the base functionality required for developing LLM-enabled MOSAICO agents. It extends *MosaicoAgentExecutor* and therefore inherits its default observability support while additionally providing facilities for invoking LLMs.

To support LLM invocation, the `llm_completion` method sends a request to an LLM through LiteLLM. The `llm_first_choice_text` method provides a convenience wrapper around the method `llm_completion`, returning only the textual content of the first response choice. The class also provides a `health` method for performing a health check of the agent.

Projects using *LLMAgentExecutor* must include the `llm` dependency extra by declaring `mosaico-base-agent[llm]` as a dependency. The Basic LLM Agent (see Section 5.3.5) included with the library provides an example of a concrete subclass of *LLMAgentExecutor*.

5.3.2.4 LGReactAgentExecutor

LGReactAgentExecutor provides a base executor for developing agents based on LangGraph with the ReAct architecture. It combines observability facilities with LangGraph's ReAct agent architecture, allowing agents to dynamically reason about their next actions and invoke tools accordingly.

The executor provides an `invoke_graph` method that initialises the corresponding A2A task, when necessary, and asynchronously executes the LangGraph agent graph, including its ReAct reasoning and tool-calling process.

The `execute_agent` method implemented by *LGReactAgentExecutor* defines the agent's behaviour. It validates the incoming message, executes the LangGraph agent graph through `invoke_graph`, and adds the resulting solution, as well as an explanation when available, as artifacts of the A2A task. The executor additionally implements a `health` method for performing agent health checks by checking the configured LLM and its availability.

The *MosaicoReactMiddleware* middleware is used to stream intermediate execution updates produced by the ReAct agent through the A2A infrastructure. In particular, it propagates tool calls and their results as task status updates while the agent is executing.

Projects using *LGReactAgentExecutor* must include the `langchain` dependency extra. The LangGraph ReAct Agent (see Section 5.3.5) included with the library provides an example of a concrete subclass of *LGReactAgentExecutor*.

5.3.3 Testing Support

The library provides two base testing classes intended to be subclassed when implementing unit tests for MOSAICO agents: *AgentExecutorTest* and *LLMAgentExecutorTest*. These testing abstractions are still under development and currently provide a limited set of reusable testing facilities.

AgentExecutorTest provides facilities for unit testing generic MOSAICO agents. At present, it provides functionality for creating a mock A2A event bus that can be reused when testing agent executors. The Arithmetic Agent provides an example of a test class based on *AgentExecutorTest*.

LLMAgentExecutorTest provides corresponding facilities for unit testing LLM-enabled agents. It includes functionality for creating a mock LLM response, allowing the behaviour of an LLM-enabled agent to be tested without relying directly on an actual LLM response. The Basic LLM Agent provides an example of a test class based on *LLMAgentExecutorTest*.



5.3.4 Agent Definition

The library provides abstractions for defining and configuring MOSAICO agents. A MOSAICO agent developed using the library typically consists of four Python files: `card.py`, `executor.py`, `main.py`, and `tests.py`. For local deployment, an optional `.env` file can additionally be used to provide the agent's runtime configuration.

The `card.py` file defines a `create_card` function that returns the agent's Agent Card. The `executor.py` file defines the concrete agent executor by extending an appropriate Base Agent Executor class and, when necessary, overriding `execute_agent` to specify the agent's behaviour. The `main.py` file instantiates the MOSAICO agent using its agent card and concrete agent executor, while `tests.py` tests the agent's behaviour.

5.3.4.1 Agent Configuration

Runtime configuration is represented through *MosaicoConfig*, which centralises configuration properties used by the agent. These include network settings such as `host`, `port`, `agent_card_host`, and `agent_card_port`; optional LLM provider settings such as `api_base`, `api_key`, and `model_name`; and Langfuse observability settings such as `langfuse_host`, `langfuse_public_key`, and `langfuse_secret_key`. Configuration values can be defined programmatically or loaded from environment variables using `MosaicoConfig.from_env()`. For local deployment, these environment variables can be specified in the optional `.env` file.

5.3.4.2 Agent Instantiation

MosaicoAgentMain provides the standard bootstrap logic for a MOSAICO agent. It is responsible for wiring together the agent executor, A2A request handler, task store, and push-notification infrastructure, as well as creating the underlying A2A web application. A MOSAICO agent application instantiates *MosaicoAgentMain* with an agent card and a concrete *MosaicoAgentExecutor* implementation, allowing the library to handle the A2A communication infrastructure while the concrete executor implements the agent's behaviour.

For example, the Arithmetic Agent instantiates a MOSAICO agent as follows:

```
def main():
    config = MosaicoConfig.from_env()
    initialize_langfuse(blocked_scopes=["a2a-python-sdk"])

    agent_main = MosaicoAgentMain(
        agent_card=create_card(config),
        agent_executor=ArithmeticExecutor()
    )
    agent_main.run(config)

if __name__ == "__main__":
    main()
```

In this example, the configuration is first obtained from environment variables and Langfuse is initialised. *MosaicoAgentMain* is then instantiated using an agent card created from the configuration and an *ArithmeticExecutor*, and the agent is started by invoking `run` with the runtime configuration.

5.3.5 Example Base Agents

The library currently provides a set of example agents that demonstrate the use of its different base executor classes:

- **Arithmetic Agent** — evaluates arithmetic expressions without relying on an LLM. It supports addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`**`) on integer values. Example inputs include "`1 + 1`" and "`2 * 4 + 5`".
- **Basic BDI Agent** — shows how to handle performatives in its incoming and outgoing messages.
- **Basic LLM Agent** — processes textual requests using an LLM and returns the generated responses to the user.
- **LangGraph ReAct Agent** — a math assistant that uses an LLM to interpret natural-language questions, dynamically determines which tools to invoke, and produces answers based on the tool results. The agent provides four tools: `add`, `subtract`, `multiply`, and `divide`. Unlike the Arithmetic Agent, its input does not need to be provided as a literal arithmetic expression. For example, it can process a natural-language request such as "What is 10 plus 5, divided by 3?".
- **Context7 RAG Agent** — The Context7 RAG Agent is a RAG agent that answers documentation queries by relying on tools exposed by the remote Context7 MCP server, which provides access to software documentation. The agent is based on *LGReactAgentExecutor*, and its tools are retrieved from the Context7 MCP server at start-up rather than being defined as a fixed local toolset. For example, it can answer queries such as "`What is a protected region in Eclipse Epsilon?`". To use the agent, a Context7 API key must be provided through the `CONTEXT7_API_KEY` environment variable.

To run locally an example agent, a `.env` file should first be created in the corresponding agent directory and configured with the required parameters. For example, the network configuration can be specified as follows:

```
AGENT_CARD_HOST=host.docker.internal
AGENT_CARD_PORT=9000
AGENT_CARD_PROTOCOL=http
HOST=localhost
PORT=9000
```

For agents that use an LLM, the `MODEL_NAME`, `API_BASE`, and `API_KEY` parameters must additionally be configured. Moreover, for agents with observability enabled, the Langfuse parameters `LANGFUSE_HOST`, `LANGFUSE_PUBLIC_KEY`, and `LANGFUSE_SECRET_KEY` must be configured. Once the configuration has been provided, the agent's dependencies can be synchronised and the agent can be tested and executed as follows:

```
uv sync
uv run -m unittest
uv run main.py
```

```typescript
export type IsSWE = "true" | "false" | "unknown";

export type TextMessagePart = {
  type: "text";
  text: string;
};

export type ImageMessagePart = {
  type: "imageUrl";
  imageUrl: { url: string };
};

export type MessagePart = TextMessagePart | ImageMessagePart;

export type MessageContent = string | MessagePart[];

interface ReferenceAgentDataPart {
  role: "user" | "assistant" | "system" | "thinking" | "tool";
  content: MessageContent;
  isSWE: IsSWE;

  // Other values are allowed but may not be understood by ref agent
  // (for compatibility with existing A2A SDK types).
  [key: string]: any;
}
```

Listing 2: TypeScript declaration of A2A reference agent input data parts

5.4 Requirements per agent type

MOSAICO distinguishes four types of agent, each with its own expectations about the A2A messages it consumes and the artifacts it produces. The requirements below are stated independently of any implementation.

5.4.1 Reference agent

The reference agent is the entrypoint to the MOSAICO platform, and the only fixed agent in the workflow. The reference agent is paired up with the reference client. The reference agent is supplied by the project rather than written by developers. It is documented here because its behaviour defines what the other agent types can expect to receive, and what the client will do with what they return.

The agent is expected to receive an A2A *Message* made up of file parts and data parts. The data parts represent each message in the conversation history, and correspond to *ReferenceAgentDataPart* on Listing 2.

The `isSWE` flag is an additional piece of information produced by the reference agent (via status updates) and recorded by the reference client. It indicates whether the original message was classified as relating to software engineering tasks (to be aided by MOSAICO agents) or not. Initially, this is unknown, and the LLM of the reference agent is used to perform this classification.

The reference agent will then usually start an A2A *Task* to handle the request from the user, if it is a software engineering request. Otherwise, it will reply directly with a *Message* from its

own LLM.

The reference agent will select an agent (whether a collaboration agent or a standalone solution agent), and forward the task to it. It will then stream back the artifact updates produced by the agent, which will be rendered by the reference client them as follows:

- `data` parts will be rendered as JSON code blocks.
- `file` parts will be rendered as syntax-highlighted code blocks (for A2A *FileWithBytes*) or links (for A2A *FileWithUri*).
- `text` parts will be rendered as Markdown-formatted text.

The reference agent will also stream back any status updates from the selected agent, which will be rendered as thoughts by the reference client. Users are able to hide or show those thoughts as desired.

5.4.2 Collaboration agent

Collaboration agents are assumed to receive a *Message* and spawn a *Task*, with frequent status updates on the ongoing collaboration.

Collaboration agents may be hardwired to use specific agents, or may select them from the repository during execution. The latter type of collaboration agent is expected to support one more environment variable:

- `REPOSITORY_URL`: base URL to the repository (e.g. `http://mosaico-app:8080`).

When receiving an artifact update, the collaboration agent should discriminate on whether it should immediately forward the update to the requester, or withhold the update and use it instead to guide the collaboration and/or forward it to another member of the collaboration. This will depend on the nature of the collaboration.

Collaboration agents will typically appear to an outside requester as either providing a solution, or providing an evaluation (in the style of the Composite design pattern). Due to this reason, their expected artifacts and parts will be those of the corresponding type of agent.

5.4.3 Solution agent

Solution agents are expected to solve the task they are given, and propose a solution. Typically, they will receive a *Message*, and either immediately reply with a *Message*, or spawn a *Task*. For longer-running solution agents (in the order of minutes), streaming should be supported.

A solution agent should produce at least these artifacts:

- An artifact named `solution`, with parts dedicated to the produced solution. These should normally be `file` parts with the generated content, although in future versions they may be `data` parts requesting the use of a client-side tool (e.g. the agent suggesting the creation of a file).
- If possible, an artifact named `explanation`, with parts dedicated to providing a rationale. These should normally be `text` parts.

5.4.4 Supervision agent

Supervision agents are expected to evaluate a provided solution, and decide on its quality. This quality can be a binary pass/fail, or a score. In addition, they should provide a rationale for their evaluation.

Therefore, a supervision agent should produce at least these artifacts:

- An artifact named `evaluation`, with a `data` part with either a pass/fail score, or a score-based evaluation (e.g. tests passed).
- An artifact named `explanation`, with text parts dedicated to providing a rationale for the score.

The `data` part of the `evaluation` should match the *Evaluation* type on Listing 3.

```
type Evaluation = PassFail | ScoreBased

interface PassFail {
    accepted: boolean
}

interface ScoreBased {
    obtainedScore: number
    maximumScore?: number
    minimumScore?: number
}
```

Listing 3: TypeScript declaration of the *Evaluation* provided by supervision agents

5.5 AISP-specific extension: observability metadata

The metadata for this extension on A2A agent cards should be:

- Extension URI: `https://mosaico-project.eu/extensions/mosaico-super-task-id`
- Description: Extends MOSAICO agents with the ability to link traces of hierarchically decomposed tasks.
- Required: false.

SendMessageRequest and *Task* should include the following metadata fields:

- `mosaico-super-task-id`: UUID of the A2A task that originated this message. The invoked agent should use this super-task ID to identify the parent span of the observability traces when communicating with Langfuse (typically as the trace's `session_id`).
- `mosaico-root-task-id`: UUID originally sent by the invoking agent. Typically, the *Task* created by the reference agent handles this field.
- `mosaico-root-task-name`: name of the root A2A task that originated the message. This controls the name of the root observation in order to prevent it to be empty or set by child observations.

A2A and this extension together would map to a typical LLM observability solution (e.g. Langfuse) as follows:

- The A2A context ID will become the Langfuse session ID. This tracks an ongoing conversation between the user and the MOSAICO agents.
- `mosaico-root-task-id` will become the Langfuse trace ID, after removing the four UUIDv4 hyphens (to match the W3C Trace Context `trace-id`). This tracks a specific interaction within a conversation.
- The last 16 hex digits of the `mosaico-super-task-id` will become the Langfuse parent observation ID (to match the W3C Trace Context `parent-id`). The choice of using the last 16 hex digits is based on the use of the last 16 digits when shortening long IDs to shorter IDs in the W3C Trace Context spec. This relates the traces in a sub-agent with those of its super-agent.

5.6 AISP-specific extension: client-side tool calling

This is one of main artifacts for deliverable D1.3. It allows downstream MOSAICO agents to request the execution of tools that run locally on the client, giving an agent a way to retrieve information from, and act upon, the user's own workspace rather than working only from what was attached to the prompt. The metadata for this extension on A2A agent cards should be:

- Extension URI: `https://mosaico-project.eu/extensions/mosaico-client-side-tool-use`
- Description: Allows downstream MOSAICO agents to request the execution of tools that run locally on the client (e.g. `read_file`, `edit_existing_file`). The agent's LLM decides which tools to call; the client executes them and streams the results back.
- Required: false

The rest of this section describes the extension in the order in which a request travels through it: how an agent declares support for it, the wire format of the three payloads it defines, the behaviour of the client that executes the tools, the implementation shared by all agents built on the Base Agent library, and a worked example.

5.6.1 Extension declaration

Each agent declares client-side tool support in `AgentCard.capabilities.extensions`:

```
1 {
2   "uri": "https://mosaico-project.eu/extensions/mosaico-client-side-tool-use",
3   "description": "Supports client-side tool execution via input-required loop.",
4   "required": false
5 }
```

The tool use feature uses the A2A `input-required` task state, with a loop that repeats until the LLM produces a final answer without requesting further tool calls.



5.6.2 Wire format

All payloads are carried as strongly typed extensions inside the `metadata` property of an A2A *Message* (or a *Message* within a *TaskStatus*). The metadata dictionary uses the extension URI as its key:

```
{
  "role": "...",
  "parts": [ ... ],
  "metadata": {
    "https://mosaico-project.eu/extensions/mosaico-client-side-tool-use": {
      "type": "<tool-definitions | tool-call-request | tool-call-result>",
      "...": "..."
    }
  }
}
```

> **Note:** The A2A protocol validation requires that the `Message.parts` array **cannot be empty**. If a message is sent purely to transport tool-use metadata (such as an `input-required` status update or a follow-up tool result), you must include at least one dummy part, such as an empty text part (`{"text": {"value": ""}}`).

5.6.2.1 Tool definitions

The client collects its available tools and sends them in the `metadata` of the initial message. The reference agent then forwards them to the selected downstream agent via the message `metadata`.

```
// Metadata payload structure
interface ToolDefinitionsPayload {
  type: "tool-definitions";
  tools: ClientTool[];
}

// OpenAI function-calling format
interface ClientTool {
  type: "function";
  function: {
    name: string;
    description?: string;
    parameters?: Record<string, unknown>;  // JSON Schema
  };
}
```

Example metadata payload:

```
{
  "type": "tool-definitions",
  "tools": [
    {
      "type": "function",
      "function": {
        "name": "read_file",
```

```
        "description": "Use this tool if you need to view the contents of an existing
            file.",
        "parameters": {
          "type": "object",
          "required": ["path"],
          "properties": {
            "path": {
              "type": "string",
              "description": "Path to the file, relative to the workspace root or
                  absolute."
            }
          }
        }
      },
    /* ... rest of the tools ... */
  ]
}
```

5.6.2.2 Tool call request

When the downstream agent's LLM requests a tool call, the agent publishes an `input-required` task state. The embedded message in this status update carries the `tool-call-request` payload in its `metadata`. Multiple calls can be batched in the `requests` array if the LLM returns parallel tool calls.

```
interface ToolCallRequestPayload {
  type: "tool-call-request";
  requests: ToolCallRequest[];
}

interface ToolCallRequest {
  toolCallId: string;       // UUID matching the LLM's tool_call ID
  functionName: string;     // must be in the original tool-definitions list
  arguments: Record<string, unknown>;
}
```

Example payload:

```
{
  "type": "tool-call-request",
  "requests": [
    {
      "toolCallId": "916222791",
      "functionName": "read_file",
      "arguments": { "path": "calculator.py" }
    }
  ]
}
```

5.6.2.3 Tool call result

The client executes the requested tool locally and sends the result back as a follow-up A2A message (same `taskId` and `contextId`) containing a `tool-call-result` payload in its `metadata`.

```
interface ToolCallResultPayload {
  type: "tool-call-result";
  results: ToolCallResult[];
}

interface ToolCallResult {
  toolCallId: string;    // matches the toolCallId from the request
  toolName?: string;     // the name of the tool called
  toolCallStatus?: "accepted" | "rejected" | "errored"; // status of the tool execution
  content?: string;      // tool output on success
  error?: string;        // error message on failure
}
```

Example payload:

```
{
  "type": "tool-call-result",
  "results": [
    {
      "toolCallId": "916222791",
      "toolName": "read_file",
      "toolCallStatus": "accepted",
      "content": "class Calculator:\n def add(self, a, b):\n return a + b\n\n def
          subtract(self, a, b):\n return a - b ..."
    }
  ]
}
```

5.6.3 Wire format summary for client-side tool calling

Table 2 summarises how the tool calling payloads are wired and transported during an end-to-end tool calling session (from client, to reference agent, to downstream agents, and back to the client).

5.6.4 Implementation in the MOSAICO base agent

The client-side tool-use protocol is implemented once in the MOSAICO base agent package (`mosaico.base`) so that individual agents do not need to re-implement it. The implementation has two layers: a small, framework-agnostic protocol layer that builds and parses the metadata payloads from Section 5.6.2, and a LangGraph execution layer that gives any LangGraph ReAct agent the full input-required pause/resume loop with no extra code beyond a system prompt.

5.6.4.1 Protocol layer

The module `mosaico.base.tool_use` exposes the extension's *AgentExtension* declaration, two small data classes (*ToolCallRequest*, *ToolCallResult*), and a handful of functions for constructing and reading the `tool-definitions`, `tool-call-request`, and `tool-call-result` payloads. An agent declares tool-use capability by adding this extension to its *AgentCapabilities*. Because the A2A task's message history accumulates every `tool-call-request` and

Direction	Transport	Payload Location
Client → Reference Agent (tool definitions)	Initial message `metadata`	`metadata[extensionUri].tools`
Reference Agent → Downstream Agent (tool definitions)	Initial message `metadata`	`metadata[extensionUri].tools` (forwarded)
Downstream Agent → Reference Agent (tool call)	Status update `message.metadata`	`metadata[extensionUri].requests`
Reference Agent → Client (tool call)	Status update `message.metadata`	`metadata[extensionUri].requests` (forwarded)
Client → Reference Agent (tool result)	Follow-up message `metadata`	`metadata[extensionUri].results`
Reference Agent → Downstream Agent (tool result)	Follow-up message `metadata`	`metadata[extensionUri].results` (forwarded)

Table 2: Wire format summary for client-side tool use.

`tool-call-result` exchanged so far, the agent can recover that history on each turn, keeping the tool-use session itself stateless.

5.6.4.2 LangGraph execution layer: how the loop works

The ReAct agent architecture makes it straightforward to implement tool-use functionality, so the base package additionally provides *LGClientToolUseAgentExecutor*. A concrete agent subclasses it and supplies only a system prompt:

```
class ClientToolUseAgentExecutor(LGClientToolUseAgentExecutor):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, system_prompt="You are a helpful assistant...", **kwargs)
```

Everything else, detecting a tool request, publishing it, and resuming once the client answers, is inherited. The mechanism relies on a standard LangGraph feature: the agent's graph is built with `interrupt_before=["tools"]`, so execution pauses immediately after the model decides to call a tool, before that tool would actually run. Tools are not known until a client supplies its JSON-Schema definitions in the first request. The executor therefore builds LangChain's *StructuredTool* stubs dynamically per session, via `build_tools` to map the client's schemas onto LangGraph tools. The executor then checks whether the graph is paused with a pending tool call and, if so, packages every requested call (client-supplied tools may be called in parallel) into a single `input-required` status update via `build_tool_call_request_message`.

Because the graph is paused mid-conversation rather than finished, its state has to survive until the client's result arrives in a separate A2A message. The executor keeps this paused state, one LangGraph graph plus its own *MemorySaver* checkpoint, in an in-memory session keyed by task id rather than context id, so that a second task sharing the same conversation context cannot evict the first task's pending call. Sessions are not kept forever: they expire after a configurable `session_ttl_seconds` (one hour by default), reaped both lazily on the next

request and by a background sweep, so an abandoned mid-conversation session does not leak; a follow-up for an expired or unknown session fails cleanly rather than silently restarting. An optional `tool_call_limit` constructor argument wires in LangGraph's *ToolCallLimitMiddleware* to bound how many tool calls a single run may make.

Writing the results as the output of the interrupted `tools` node lets LangGraph resume exactly where it left off, feeding the tool output back to the model as if it had executed the call itself. This repeats, one round trip per batch of tool calls, until the model answers without requesting further tools, at which point the task completes and its session is discarded. A rejected tool call is not passed through silently: it is turned into a short synthetic system message telling the model the user refused, so it can adapt (e.g. ask the user how to proceed) rather than treating an error as valid output. An errored result is likewise turned into a system message carrying the error detail, letting the model decide whether to retry, try another approach, or give up.

This design keeps the two layers independently usable: the protocol layer alone is enough for a non-LLM, fixed-sequence agent, while the LangGraph layer is the fast path for the common case of an LLM-driven ReAct agent, requiring only a system prompt to get the full loop.

5.6.4.3 Worked example: the `docstring-agent`

To demonstrate the ease of implementation, this section presents a `docstring-agent` that uses LLM to add or update Python docstring (documentation string) in source files.

The Reference Client V2 supports a list of tools, including tools to read and edit existing files in the workspace. Given a user request with a file name, the docstring agent reads the file through a client-side tool, adds the LLM-generated docstrings, and returns the new content back to the client to write using another client-side tool. It is a realistic illustration of how little code an agent needs beyond what the base package already provides. The docstring agent inherits the *LGClientToolUseAgentExecutor* and only adds its own system prompt:

```
class DocstringAgentExecutor(LGClientToolUseAgentExecutor):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, system_prompt=SYSTEM_INSTRUCTION, **kwargs)
```

Every behaviour beyond the generic loop described above comes from the system prompt. The agent does not hardcode its tools: it expects the client to supply two JSON-Schema tool definitions (`read_file` and `edit_existing_file`). The prompt then enforces a specific two-step workflow on top of the generic mechanism: read the file first via `read_file`, then issue exactly one `edit_existing_file` call carrying the new file content. This ordering is not enforced by any code in the executor, it relies entirely on the LLM following the prompt, with the base class's pause/resume loop simply reacting to whichever tool call the model actually emits.

Figure 5 shows the agent at work in the reference client. The `read_file` call runs without interrupting the user, under its default **Automatic** policy, while `edit_existing_file` requires approval. The two outcomes of that approval are both visible: when the user accepts the edit (in green), the agent reports the docstrings it added; when the user rejects it (in red), the refusal is passed back to the model, and the agent asks the user how it should proceed rather than treating the refusal as a failure. Each of the two tool calls is a full round trip of the exchange summarised in Table 2, relayed through the reference agent in both directions.

5.7 Running agents from sources

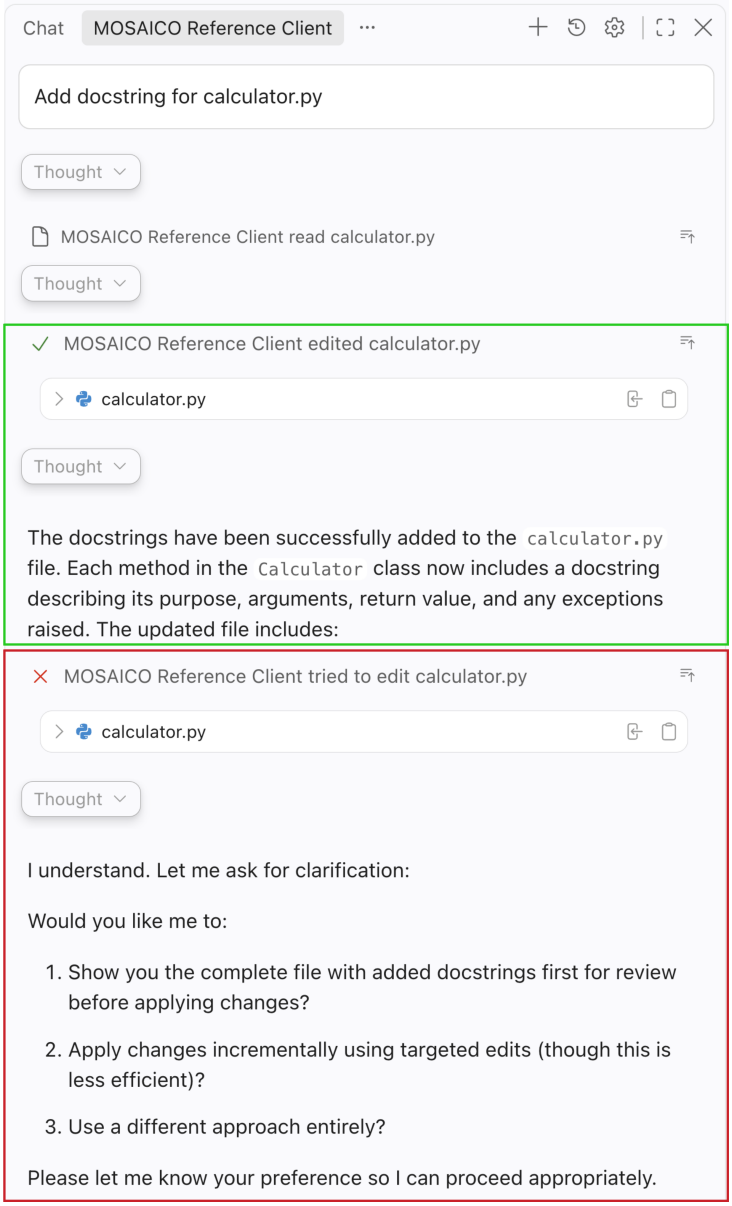


Figure 5: The `docstring-agent` adding docstrings to `calculator.py` through client-side tools, showing an accepted `edit_existing_file` call (green) and a rejected one (red)

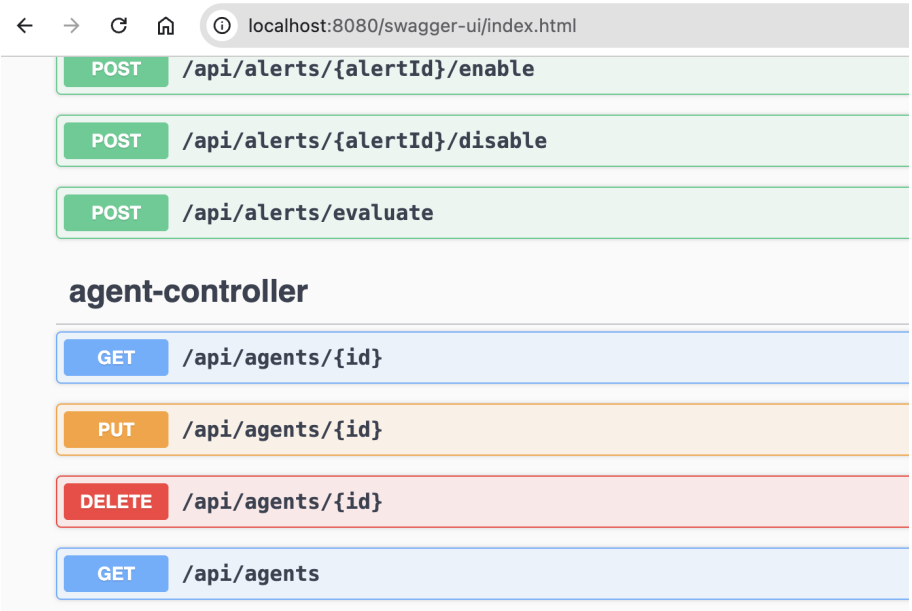


Figure 6: Screenshot of the Swagger UI interface for the REST API of the MOSAICO repository

As mentioned on Section 5.2, on a production environment, agents are expected to be distributed as Docker images. However, for local development and debugging of the integrated platform, there are cases where a developer may want to directly run an agent from their local development environment. This usually means that their agents will be reachable through the ports of the host system's network interfaces, not being part of the agent subnet. As a consequence, the registered URL to the A2A agent card and the A2A endpoint advertised on that card may require adjustments.

The Compose orchestration script (`01-compose.sh`) already supports such an option for the metamodel collaboration agent (by disabling the `WITH_MM_COLLAB_AGENT` setting).

We expect the need for such hand-written options for adjusting the Compose orchestration will reduce significantly in future releases of the reference client. It is planned to extend the reference agent to support mixing local agents started on-demand (via their Docker images) with pre-existing agents already running in the network. These pre-existing agents could include agents started by a developer from their local environment, and registered appropriately into the MOSAICO repository.

5.8 MOSAICO Repository interfaces

Besides communicating with agents over A2A, developers can query and manage the contents of the MOSAICO repository directly, either through its REST API or through its MCP server.

5.8.1 REST API

The MOSAICO repository provides a REST API to query and manage its contents, which is accessible through a Swagger UI interface (generated from its machine-readable OpenAPI specification) from `http://localhost:8080/swagger-ui.html`.

Figure 6 shows a screenshot of the Swagger UI interface that developers can use to experiment with the MOSAICO repository.

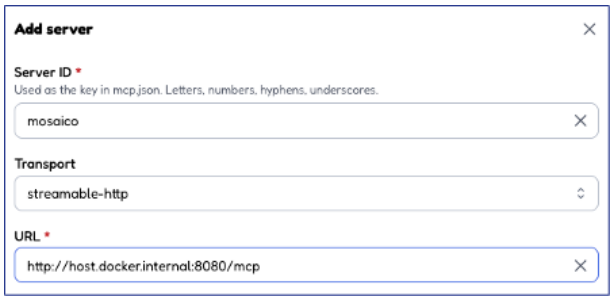


Figure 7: Screenshot of the settings required on the Anthropic MCP Inspector to connect to the MOSAICO repository's MCP server

The automated agent registration done by the Compose-based container orchestration (Section 3.7) uses the `/api/agents` endpoint in this API. Specifically, it sends GET, PUT, and POST requests to compare the registered agents against the ones in the container orchestration, and updates the records as needed.

5.8.2 MCP server

The MOSAICO repository implements an MCP server to expose its agents to LLMs (such as the one in the reference agent). To experiment with the MCP server, you can run the Anthropic MCP Inspector[23] with the following command:

```
./99-mcp-inspector.sh
```

The MCP inspector will print a URL after the following line:

```
MCP Inspector is up and running at:
```

After opening the link (which includes an authentication token), the following settings can be used to connect to the MCP server (as shown on Figure 7):

- Transport Type: Streamable HTTP
- URL: `http://host.docker.internal:8080/mcp`
- Connection Type: Via Proxy

After clicking on Connect, it is possible to access the various MCP resources and tools provided by the MOSAICO repository.

6 Conclusions

This report has documented the second version of the MOSAICO demonstrator, and with it the current state of the MOSAICO AI-agent Server Protocol.

AISP remains a profile of A2A, and now carries three extensions: observability metadata (Section 5.5), an agent communication language with the "tell", "achieve" and "ask" performatives (deliverable D3.1), and, new in this deliverable, client-side tool calling (Section 5.6). The last of these is what makes agent-directed information retrieval possible, and is the objective of

[23] https://github.com/modelcontextprotocol/inspector

D1.3. An agent is no longer restricted to the context that the user thought to attach to a prompt. It can ask the client for what it needs from the workspace, and act on that workspace in return, under a per-tool permission policy that the user controls (Section 2.2.3).

This version also consolidates the implementation behind the AISP protocol. The reference client has been rewritten as a small, purpose-built VS Code extension, ending the dependency on the discontinued project it was originally forked from (Section 2.2). The MOSAICO Base Agent library (Sections 2.3 and 5.3) implements the AISP profile, the agent requirements, observability, and the complete tool calling loop once, so that a new agent inherits them rather than reimplementing them. The `docstring-agent` extends the base agent using a minimal executor subclass and a system prompt, and demonstrates the tool calling in practice: reads a source file from the user's workspace, generates docstrings for it, and writes the result back.

Work on the demonstrator continues beyond this deliverable, with the report noting plans to migrate further agents onto the Base Agent library (Section 2.3), to extend testing to further LLMs (Section 2.4), to add further sample conversations alongside the new agents that join the demonstrator (Section 4.4), to track new stable releases of A2A as their SDK support matures (Section 5.1), and to let users run agents on demand from their own machines (Sections 5.2 and 5.7). The next task in WP1 (Task 1.4) is to implement support for fine-grained feedback, which will require further AISP extensions that will involve the reference client, reference agent, and base agent library.

