

MOSAICO

D1.2: MVP reference implementations of client and server

March, 2026

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

D1.2 MVP reference implementations of client and server

Work package	WP1 AI-agent server protocol
Task	T1.2 Minimal viable product (MVP) implementation
Task contributors	IMT, UDA, ECL
Due date	31/03/2026
Submission date	26/03/2026
Type of deliverable	Demonstrator
Dissemination Level	Public
Deliverable lead	University of York
Version	1.0
Authors	University of York
Reviewers	UDA, IMT, LIST
Keywords	Network protocols, agent-to-agent communication, multi-agent systems

Document Revision History

Version	Date	Description of change	List of contributor(s)
v0.1.0	05/03/2026	Initial version	University of York
v0.2.0	13/03/2026	Updated architecture	University of York
v0.3.0	16/03/2026	Added GPU flag	University of York
v0.4.0	18/03/2026	Updated configuration options	University of York
v0.5.0	25/03/2026	Added IP solution agent	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This deliverable contains original unpublished work except where clearly indicated otherwise.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 demonstrator of the minimal viable product (MVP) version of 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 will be 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 stuctured collaborations among agents).

The demonstrator is comprised of a Visual 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 three solution agents.

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 can use the included Docker Compose container orchestration 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 repository.

Table of Contents

1 Introduction . . . 8

2 Architecture . . . 8
2.1 MOSAICO components . . . 8
2.2 Supported LLMs . . . 10
2.3 Hardware requirements . . . 11

3 Instructions for administrators . . . 11
3.1 Software requirements . . . 11
3.2 Initial setup . . . 12
3.3 Pulling the embedding model . . . 13
3.4 Optional: running local LLMs . . . 13
3.4.1 Via LM Studio (recommended) . . . 13
3.4.2 Via Ollama in Docker . . . 13
3.5 Starting the demonstrator . . . 14
3.6 Container orchestration . . . 14
3.7 Automated agent registration . . . 15
3.8 Shutting down the demonstrator . . . 16

4 Instructions for users . . . 16
4.1 Installing the reference client . . . 16
4.2 Initial configuration . . . 16
4.3 Using the reference client . . . 17

5 Instructions for developers . . . 19
5.1 Communication protocols . . . 19
5.2 Packaging . . . 19
5.3 Requirements per agent type . . . 20
5.3.1 Reference agent . . . 20
5.3.2 Collaboration agent . . . 21
5.3.3 Solution agent . . . 22
5.3.4 Supervision agent . . . 22
5.4 AISP-specific extension: super-task IDs . . . 22
5.5 Running agents from sources . . . 23
5.6 Repository REST API . . . 23
5.7 Repository MCP server . . . 23

List of Figures

1 Current architecture of the MOSAICO platform 9
2 Accessing the “Install from VSIX” option from VS Code 16
3 Screenshot of the user interface of the MOSAICO reference client 17
4 Screenshot of the “Add chat model” dialog configured for the reference agent . . 18
5 Screenshot of the Swagger UI interface for the REST API of the MOSAICO repository 24
6 Screenshot of the settings required on the Anthropic MCP Inspector to connect to the repository’s MCP server 24

Abbreviations

A2A	Agent-to-Agent
AISP	AI-agent Server Protocol
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
SDK	Software Development Kit
SE	Software Engineering

1 Introduction

This document is a companion report for the MOSAICO demonstrator repository hosted in the Eclipse Gitlab server[1], which is the main artifact for deliverable D1.2.

The report documents the current architecture and design of the WP1 and WP2 components of the MOSAICO platform (Section 2), and provides instructions for server administrators (Section 3), end users (Section 4), and developers (Section 5).

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

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*[2] is implemented as a Visual Studio Code extension, which extends the open-source Continue[3] project with a MOSAICO-specific *LLM provider*.

This provider replaces the direct-to-LLM communication with A2A-based communication to the *MOSAICO reference agent*[4]. 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.

This first version of the demonstrator includes:

- An agent dedicated to generic software engineering tasks (`mini-swe-agent`[5]). This agent is an example of the work required to take an pre-existing agent and integrate it with the MOSAICO platform: the original agent was developed by the SWE-Bench project[6].
- A collaboration agent for generating Emfatic[7] metamodels according to the description of a domain, which orchestrates three lower-level agents:
  - a Smolagents[8]-based agent for producting Emfatic source code,
  - a syntactic supervision agent that checks for valid Emfatic syntax by reusing the existing Emfatic parser,
  - and a semantic supervision agent that uses an LLM to check the metamodel against the requirements indicated in the description provided by the user.

[1] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-demonstrator
[2] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-ai-coding-extension
[3] https://github.com/continuedev/continue
[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://eclipse.dev/emfatic/
[8] https://huggingface.co/docs/smolagents/index

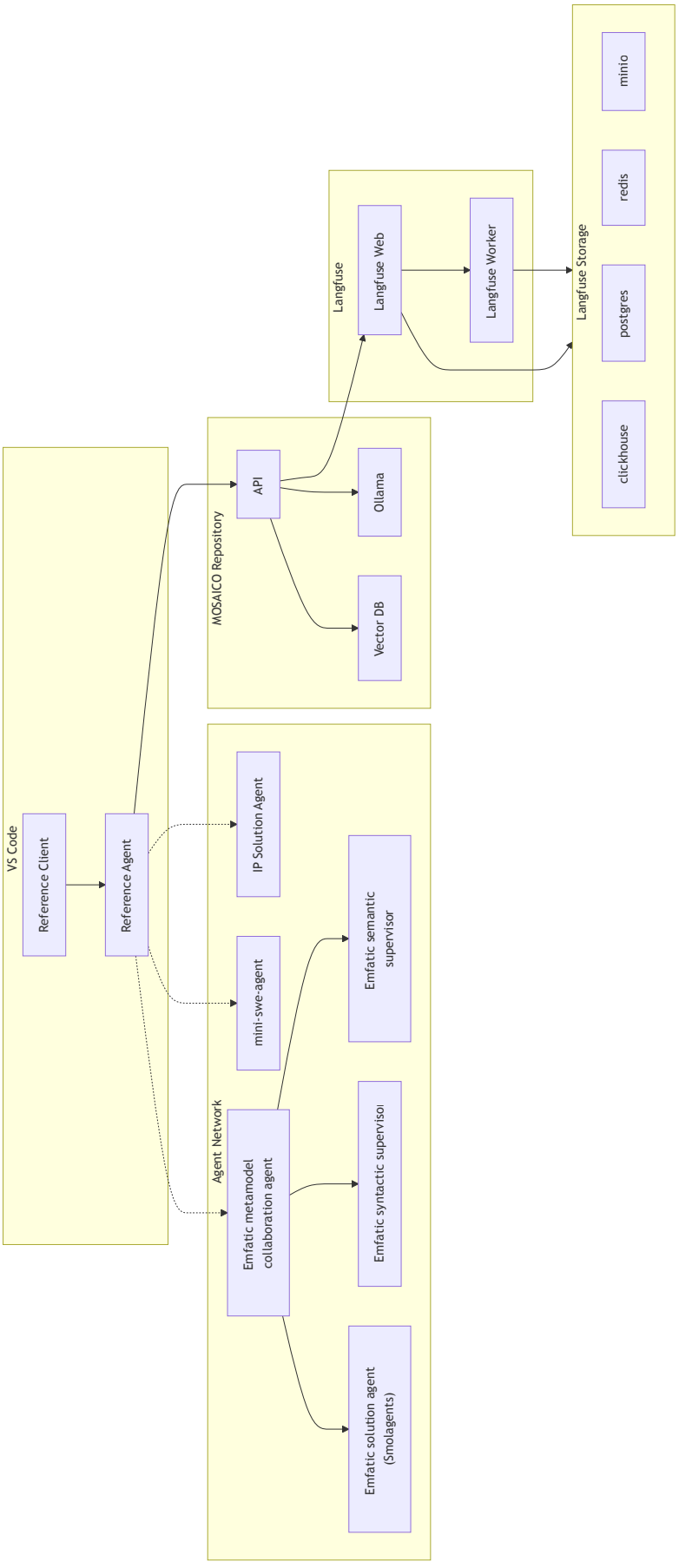


Figure 1: Current architecture of the MOSAICO platform

– An agent dedicated to finding the open source licenses of a given project (IP Solution Agent), contributed by ECL.

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[9] 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. Langfuse's architecture is documented in depth on their official website[10]. Regardless, it is largely made-up of:

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

2.2 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:

- `devstral-small-2` running locally on Ollama[11] or LM Studio[12]. Local models are important to test for some of the MOSAICO partners, as they may have sensitive data that must be processed on-premises.
- `qwen3:32b` and `qwen3-coder:30b` running locally on Ollama[13] or LM Studio[14]. These were the local LLMs used for testing until the release of `devstral-small-2`.
- `gpt-5-mini` and `o4-mini` running on Microsoft Azure AI Foundry[15]. This provider was 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.
- `meta-llama/llama-4-scout-17b-16e-instruct` running on Groq.
- `moonshotai/kimi-k2-instruct` 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[16], Github Models[17], or OpenRouter[18]. Any LLMs chosen for testing in the future should meet these requirements:

[9] https://langfuse.com/
[10] https://langfuse.com/handbook/product-engineering/architecture
[11] https://ollama.com/
[12] https://lmstudio.ai/
[13] https://ollama.com/
[14] https://lmstudio.ai/
[15] https://azure.microsoft.com/en-us/products/ai-foundry
[16] https://groq.com/
[17] https://docs.github.com/en/github-models
[18] https://openrouter.ai/

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

2.3 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.

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[19] can be used.
- A Docker Compose installation, whether as part of the Docker Engine[20] (only for Linux), or as part of Docker Desktop[21] (for Windows/Linux/Mac).

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

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

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 (25 March 2026), the `v0.1.2` tag is the most recent one:

```
git clone https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-demonstrator.git
cd mosaico-demonstrator
git checkout v0.1.2
```

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 script will create a `.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 identified 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 repository.
  - 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.
  - 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.

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

3.3 Pulling the embedding model

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

If you enabled `WITH_OLLAMA` in `compose.env`, you can do so with these commands:

```
./01-compose.sh up ollama -d
./01-compose.sh exec ollama ollama pull nomic-embed-text
```

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
```

This will pull all necessary images from the MOSAICO Gitlab project[22]. 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 repository. You can verify their registrations by checking the /api/agents endpoint, or querying the MCP server (Section 5.7 on page 23).

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
```

Please note that if you change the above configuration, you must run the following command to have Compose apply the changes:

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

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:

- `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_endpoint.yml` sets up the 'mini-swe-agent' as a remote agent (to be called by the reference agent over the network).
- `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).

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

- `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.pgadmin.yml` provides a web UI for administering the Postgres databases in the repository.
- `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.

3.7 Automated agent registration

Agents must be registered into the 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 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 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 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 repository.



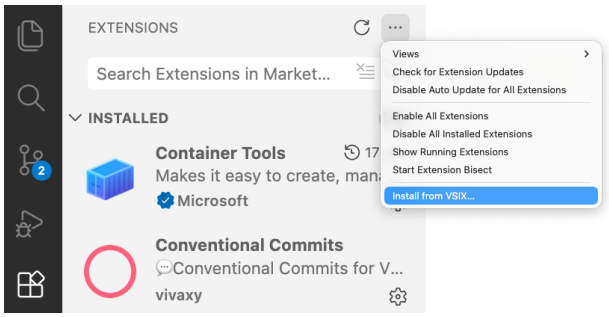


Figure 2: Accessing the “Install from VSIX” option from VS Code

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
```

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. The latest `.vsix` file for the relevant operating system can be downloaded from our Gitlab package registry.

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 Initial configuration

The next step is to show the MOSAICO reference client view, which can be done 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.

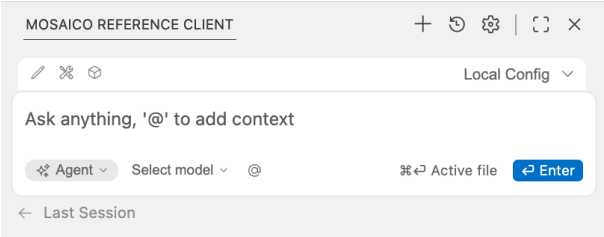


Figure 3: Screenshot of the user interface of the MOSAICO reference client

- Entering CTRL+SHIFT+P (Win/Linux) or SHIFT + COMMAND + P (Mac) to access the VS Code Command Palette, and selecting “MOSAICO: Focus Continue Chat”.

The MOSAICO reference client will open on one side of the IDE, as shown on Figure 3. In order to use the reference agent with the client, you will need to register it as a provider. Click on “Select model” and then “Add Chat model”, so it looks as on Figure 4:

- Select the “Mosaico” provider and the “Mosaico” model.
- Copy the value of `LANGFUSE_PUBLIC_KEY` in `env/langfuse.env` to the “Langfuse Public Key” field.
- Copy the value of `LANGFUSE_SECRET_KEY` in `env/langfuse.env` to the “Langfuse Secret Key” field.
- Customise the model name, API base and API key, if not using the suggested LM Studio configuration.

  For example, when using the Devstral Small 2 LLM from an Ollama installation, the model name should be changed to `devstral-small-2`, and the Ollama API base URL should be used instead.

You can now click on “Connect”, and use that as your chat model for conversations from now on.

4.3 Using the reference client

You can chat to the reference agent as you would normally do with any other LLM, with some limitations that we will address in future releases:

- File uploads are not currently supported, but you can insert the contents of a file from your workspace into the prompt by using the syntax `@filename`.
- The Chat/Plan/Agent mode switch is not currently taken into account, as the agents cannot invoke client-side tools at the moment.

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:

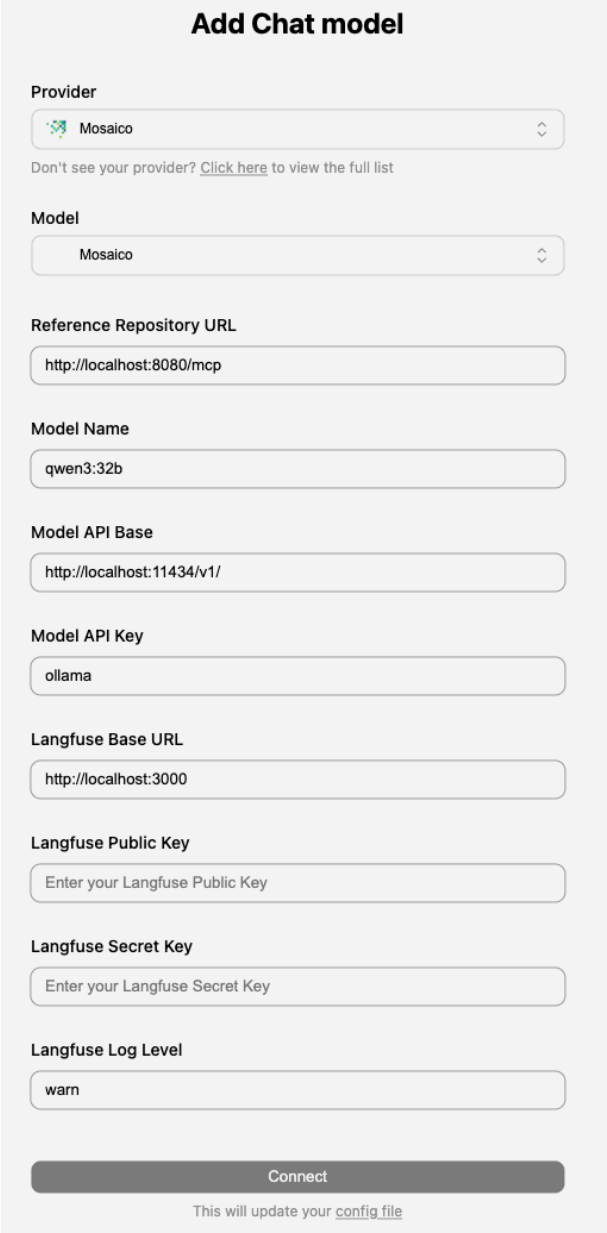


Figure 4: Screenshot of the “Add chat model” dialog configured for the reference agent

(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 Github 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: we are working on a base agent that will provide a foundation for agents that are traceable from the start.

5 Instructions for developers

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

5.1 Communication protocols

After considering various protocol design approaches in D1.1, we decided to design the MOSAICO AI Agent Server Protocol (AISP) as a profile of the Agent2Agent Protocol.

As of March 1st 2026, the latest published version of A2A is v0.3.0. 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, work is ongoing on adding several AISP-specific extensions: a super-task ID extension for cross-collaboration traceability (Section 5.4), and an agent communication language extension (with its own Github repository) with “tell”, “achieve”, and “ask” performatives. In the future, we plan to add an 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).

Agents are also expected 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 5.3 on page 20.

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 Requirements per agent type

5.3.1 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 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:



```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

- `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.

5.3.2 Collaboration agent

Collaboration agent images should support one more environment variable:

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

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

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.

5.3.3 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.

5.3.4 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.

```
1  type Evaluation = PassFail | ScoreBased
2
3  interface PassFail {
4      accepted: boolean
5  }
6
7  interface ScoreBased {
8      obtainedScore: number
9      maximumScore?: number
10     minimumScore?: number
11 }
```

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

5.4 AISP-specific extension: super-task IDs

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

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

- Required to be understood.

Messages sent from a MOSAICO collaboration agent to one of its selected agents with must 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`).

Tasks started by the invoked agent should include the following metadata fields:

- `mosaico-super-task-id`: UUID originally sent by the invoking agent.

5.5 Running agents from sources

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 enviroment. 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 repository.

5.6 Repository 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 5 shows a screenshot of the Swagger UI interface that developers can use to experiment with the repository.

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.

5.7 Repository 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:

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

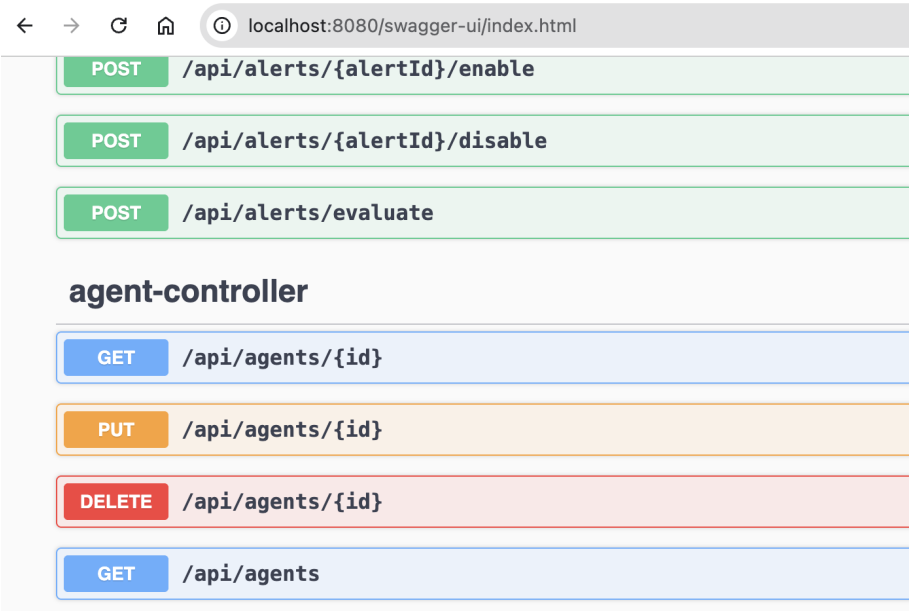


Figure 5: Screenshot of the Swagger UI interface for the REST API of the MOSAICO repository



Figure 6: Screenshot of the settings required on the Anthropic MCP Inspector to connect to the repository's MCP server

```
./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 6):

- 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.