MOSAICO

D2.1: A taxonomy of AI agents for SE

September, 2025

Funded by the European Union

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

D2.1 A taxonomy of AI agents for SE

Work package	WP2
Task	T2.1 Definition of a taxonomy of AI agents for SE
Due date	30/09/2025
Submission date	03/10/2025
Type of deliverable	Report
Dissemination Level	Public
Deliverable lead	UDA
Version	1.0.0
Authors	University of L'Aquila (UDA)
Reviewers	University of York (UY), Institut Mines Telecom (IMT)
Keywords	AI agents, software engineering, multi-agent systems, repository, taxonomy

Document Revision History

Version	Date	Description of change	List of contributor(s)
v0.1.0	15/07/2025	Initial structure and first content draft	University of L'Aquila
v0.3.0	05/08/2025	Extended background and taxonomy, added framework and repository analysis	University of L'Aquila
v0.9.0	27/08/2025	First complete draft delivered for review	University of L'Aquila
v0.9.5	10/09/2025	Internal review and feedback integration	University of York, Institut Mines Telecom
v1.0.0	01/10/2025	Final version after polishing and reviewer suggestions	University of L'Aquila, University of York, Institut Mines Telecom

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

Executive Summary

This deliverable provides the foundational taxonomy for AI agents in the MOSAICO project. The taxonomy is a preparatory step for the implementation of the MOSAICO Repository, whose purpose is to enable discovery, evaluation, and governed reuse of AI agents in Software Engineering (SE) contexts.

The document first reviews the state of the art in LLM-based Multi-Agent Systems (MAS), examining both academic taxonomies and industrial initiatives. An analysis of existing MAS frameworks is performed to elicit a set of foundational concepts (FC1–FC5) that characterize agent-based solutions with respect to capabilities, governance, evaluation, interaction, and consumption. These dimensions form the basis for a conceptual taxonomy of AI agents. Next, we analyze model repositories and agent marketplaces to identify reusability features and current gaps in metadata, monitoring, and governance. This analysis highlights the limitations of existing platforms and emphasizes the need for a richer metadata schema tailored to SE.

Building upon these foundations, the deliverable introduces a taxonomy of AI agents formalized through a metamodel that extends the conceptual dimensions (FC1–FC5) with repository-specific entities, including benchmarks, telemetry data, governance policies, and usage records. The metamodel ensures alignment with industrial standards (e.g., the A2A protocol) and integrates feedback mechanisms to support performance-based agent recommendation.

Finally, the report provides an initial preview of the repository architecture, showing how the proposed taxonomy and metadata model will be operationalized to support advanced management, search, and reuse of AI agents for software engineering.

Table of Contents

List of Figures

List of Tables

Abbreviations

A2A	Agent-to-Agent
AI	Artificial Intelligence
API	Application Programming Interface
CD	Continuous Deployment
CI	Continuous Integration
DL	Deep Learning
GUI	Graphical User Interface
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
IDE	Integrated Development Environment
LLM	Large Language Model
MAS	Multi-Agent System
ML	Machine Learning
NLP	Natural Language Processing
QA	Quality Assurance
REST	Representational State Transfer
RL	Reinforcement Learning
SE	Software Engineering
UML	Unified Modeling Language
PTM	Pre-trained model

1 Introduction

In recent years, the application of large language models (LLMs) has revolutionized the field of automated software engineering by conceiving solutions that span the entire development lifecycle [1, 2]. These models are now integral to various stages, such as requirements engineering, where they assist developers in gathering, analyzing, and refining user needs with greater accuracy and efficiency [3]. In the code development phase, LLMs such as `GitHub` Copilot[1] or DeepCode[2] leverage natural language understanding and code synthesis capabilities to generate code snippets, suggest optimizations, and even debug errors in real time [4]. These advances have accelerated the coding process, allowing developers to focus on higher-level design and problem-solving. This shift has contributed to the rise of the AIware paradigm [5, 6], where AI agents are embedded as active collaborators across all phases of the software lifecycle, assisting in tasks ranging from documentation writing to automated testing [7]. Agents are systems that can plan, reason, and take action to accomplish tasks with minimal human intervention [1].

Despite these advances, LLM-based solutions still face critical challenges. Issues such as catastrophic forgetting [8], hallucinations [9], and privacy or trustworthiness concerns [10, 11] undermine their reliability and limit their adoption in production environments. These limitations have motivated the exploration of *multi-agent systems* (MAS) as a promising approach to orchestrate, manage, and evaluate multiple AI agents collaboratively [12]. By assigning specialized roles to individual agents, such as code generation, validation, or monitoring, MAS mitigate the shortcomings of single models, improving performance, reliability, and trustworthiness [13]. This has led to the emergence of the *AgentWare* paradigm, where communities of agents work together to solve complex software engineering tasks, as demonstrated by systems such as AutoGen [14].

While MAS are gaining traction, the field still lacks a comprehensive understanding of the foundational concepts and features that enable robust and reusable agent-based solutions. Key questions remain open, such as how to define agent roles, ensure seamless inter-agent communication [15, 16], or evaluate collective performance [17]. Moreover, the lack of standardized taxonomies and metadata models hinders the discoverability, comparison, and reuse of agents across frameworks and domains.

The MOSAICO project addresses these challenges by creating an integrated framework for the management, orchestration, and benchmarking of AI agent communities for software engineering. As shown in Fig. 1, the envisioned MOSAICO platform is composed of three main components: the *MOSAICO Orchestrator*, the *MOSAICO Decision Engine*, and the *MOSAICO Repository*.

The user interacts with the platform through the used IDE by submitting a prompt, which is then interpreted by a *Reference Agent* (red). The Reference Agent forwards the task to a *Collaboration Agent* (green), which decomposes the task into subtasks and defines the collaboration pattern to be executed. In the central area (Runtime Processes), various *Solution Agents* (blue) are assigned specific subtasks and work collaboratively to deliver partial or complete solutions. Their outputs are evaluated by *Supervision Agents* (yellow), which are responsible for quality assessment and supervision of the task execution. A *Consensus Agent* (depicted as a human-like head) aggregates the results and evaluations to select the most suitable solution. This solution is then returned to the user as a response to the initial prompt.

In parallel, the right-hand section (Background Processes) represents the continuous main-

[1] `https://github.com/features/copilot`
[2] `https://github.com/DeepCodeAI`

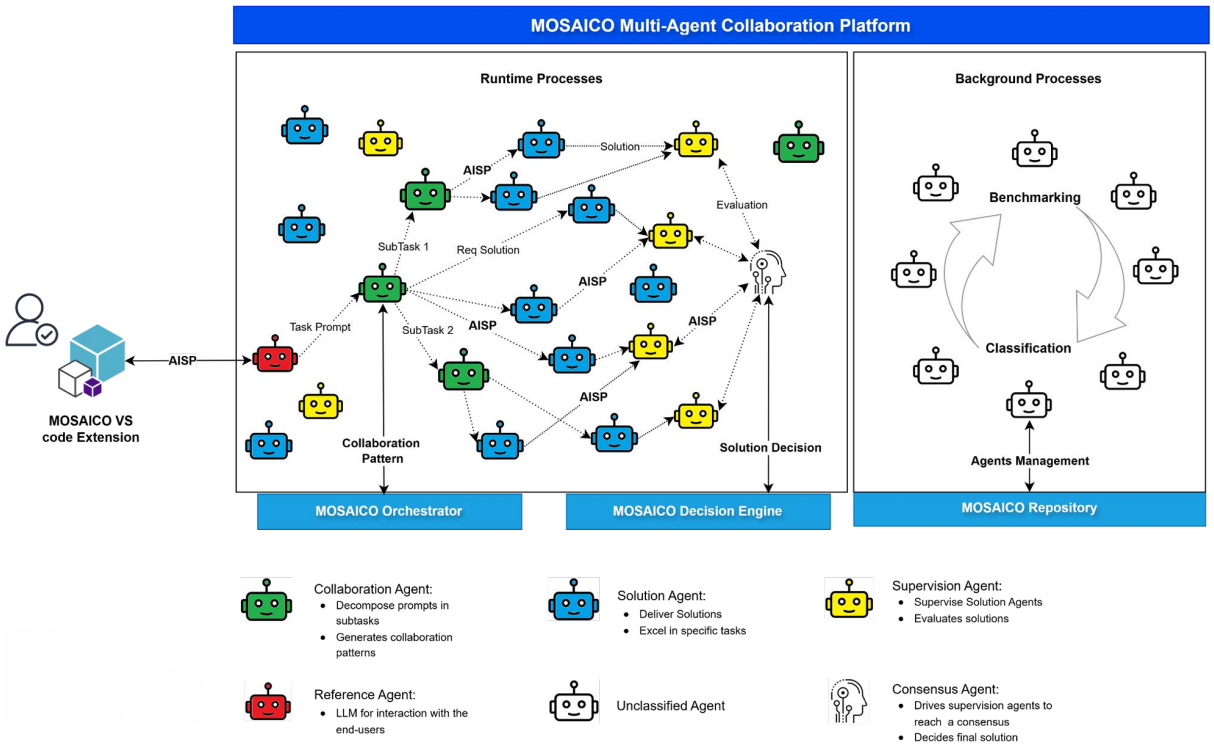


Figure 1: Overview of the MOSAICO framework.

tenance of the MOSAICO Repository, where agents are benchmarked, classified, and managed to ensure optimal performance. This repository supports dynamic agent discovery, selection, and deployment during runtime.

Work Package 2 (WP2) is responsible for developing the MOSAICO *Repository of AI agents for SE*, which will serve as a central platform for storing, categorizing, and evaluating agents based on rich metadata and Key Performance Indicators (KPIs). The repository is designed to enable advanced search, recommendation, and interoperability with orchestration protocols developed in other work packages.

This deliverable, represents the first step of WP2 by defining a *taxonomy of AI agents for software engineering*. The taxonomy is metamodel-driven and integrates insights from:

- **Scientific literature** on LLM-based MAS and software engineering agents;
- **Existing taxonomies and interoperability protocols**, including the Agent2Agent (A2A) specification and the taxonomy of agent skills proposed by Agntcy[3];
- **Marketplaces and agent repositories**, such as BeeAI[4], AgentLocker[5], AI Agents List[6], and Microsoft's AI Toolkit for Visual Studio Code[7];
- **State-of-the-art frameworks** supporting MAS development, including LangChain [18], LlamaIndex [19], Semantic Kernel [20], Smolagents [21], and Dify [22].

[3] https://docs.agntcy.org/oasf/taxonomy/
[4] https://beeai.dev/agents
[5] https://www.agentlocker.ai/
[6] https://aiagentslist.com/
[7] https://code.visualstudio.com/docs/intelligentapps/models

By consolidating these diverse sources, this deliverable provides a unified conceptual basis for characterizing and managing AI agents for SE. In particular, this document adopts a two-step approach: first, it builds a *conceptual taxonomy* derived from MAS frameworks, capturing foundational concepts (FC1–FC5) that describe agent architectures, behaviors, and reusability features. Second, it builds upon these concepts to define an **agent taxonomy**, formalized through a metamodel that defines the entities and relationships required to store, evaluate, and govern agents in the MOSAICO Repository.

The proposed taxonomy facilitates *agent discovery, reuse, and comparison* and serves as a foundation for subsequent MOSAICO deliverables on KPIs (D2.2), benchmarking (D2.3), and repository implementation (D2.4). It is also intended to guide the development of the other MOSAICO components under the responsibility of the other work packages, such as the AI-agent server protocol (WP1), orchestration protocols (WP3), and the governance framework (WP4). The goal is ensuring that the repository supports advanced search, recommendation, and evaluation capabilities tailored to the needs of software engineering practitioners and researchers.

1.1 Key contributions of this deliverable:

- A comprehensive analysis of MAS frameworks and industrial taxonomies, leading to the identification of foundational concepts (FC1–FC5).
- A set of reusability features derived from the evaluation of model repositories and agent marketplaces.
- A *metamodel-driven taxonomy* that integrates conceptual features with operational metadata for the MOSAICO repository.
- We establish the basis for the MOSAICO Repository, providing an initial preview of the repository architecture, showing how the proposed taxonomy and metadata model will be operationalized to support advanced search, governance enforcement, and orchestration capabilities.

1.2 Structure of the document

The remainder of this document is organized as follows:

- Section 2 introduces the background on AI agents in Software Engineering, including the evolution from the *Codeware* to the *AgentWare* paradigm, and reviews existing academic and industrial taxonomies.
- Section 3 analyzes current MAS frameworks to elicit the foundational dimensions (FC1–FC5) and presents the conceptual taxonomy derived from literature and framework analysis.
- Section 4 provides an experience report on using MAS frameworks, highlighting practical insights and limitations observed during their evaluation.
- Section 5 analyzes existing model repositories and marketplaces, identifying key reusability features (R1–R10) and evaluating current platforms against these features. This section also reviews industrial marketplaces and interoperability protocols relevant to MOSAICO.

- Section 6 introduces the MOSAICO agent taxonomy. This taxonomy, formalized through a metamodel, describes how agents, benchmarks, policies, and telemetry are structured to enable discovery, governance, and monitoring.
- Section 7 outlines the initial architecture of the MOSAICO Repository, showing how the presented taxonomy will be implemented in practice to support advanced repository functionalities.
- Section 8 concludes the document by summarizing the contributions and discussing future work, including the next deliverables in WP2 and their integration with the broader MOSAICO framework.

2 Background: AI Agents in Software Engineering

In this section, we briefly introduce the *AgentWare* paradigm [7] and discuss existing taxonomies for multi-agent systems (MAS) in the context of software engineering. We also highlight external taxonomies and repositories, such as those offered by industrial frameworks and marketplaces, that inform the development of the MOSAICO taxonomy.

2.1 LLM-based Multi-Agent Systems for Software Engineering

Before the advent of AI models, the *Codeware* paradigm was widely adopted, involving professional programmers whose lifecycle was driven by traditional phases, i.e., requirements, design, implementation, testing, deployment, and maintenance, carried out mostly manually [23]. With the rise of generative AI models, the paradigm shifted from *Codeware* to multiple instances of *AIware* (see Figure 2). This evolution is characterized by three main stages: *NeuralWare*, *PromptWare*, and *AgentWare*.

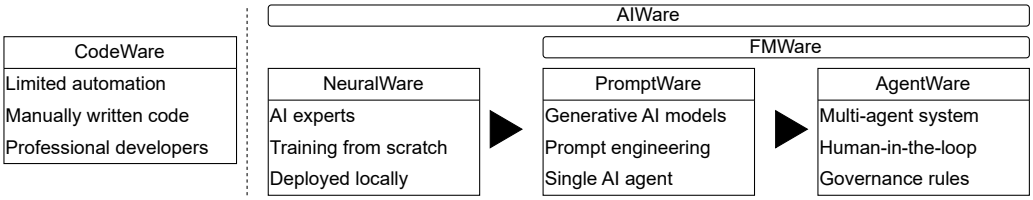


Figure 2: The evolution of software development paradigms defined by Cogo et al. [7].

NeuralWare development involves AI experts started to drive the development of a new generation of software [24] by constructing a new set of assets such as datasets and models in a significantly more iterative and experiment-driven lifecycle. However, this new paradigm has brought new challenges to the software engineering community, e.g., versioning of AI models [25], AI-related code smells [26], or additional requirements [27].

With the adoption and the further enhancements to the transformer architecture [28] and the launch of ChatGPT in November 2022, the *PromptWare* paradigm has emerged, as those models relies on natural language queries, known as *prompts*, allowing the development by programmers without deep programming and AI skills, ultimately democratizing software creation. In particular, researchers have started to improve the prompting process by introducing new techniques, moving from zero-shot [29] to more advanced paradigms, i.e., few-shot prompting [30] and chain-of-thought prompting [31]. Although promising, the PromptWare paradigm has also brought new challenges to the SE community, e.g., hallucination [32, 33], energy consumption [34, 35] and ethical concerns [36].

The most recent evolution is *AgentWare*, where multiple foundation model (FM) agents collaborate to solve tasks by reasoning, decision-making, and interacting with other agents or humans. MAS approaches, such as AutoGen [14] and MetaGPT [37], showcase the synergy of agents working together to achieve goals with improved performance and reliability.

2.2 Existing Taxonomies for Multi-Agent Systems

A variety of taxonomies have been proposed to classify MAS components and behaviors, but few directly address the challenges of software engineering. Hepworth et al. [38] introduced Onto4MAT, an ontology for human–MAS communication, focusing on intent sharing and semantic understanding. Similarly, Naja et al. [39] extended the RAInS ontology to audit accountability

in AI systems using knowledge graphs. While relevant, these works focus on narrow aspects (e.g., communication, accountability) rather than comprehensive MAS characterization.

Biancofiore et al. [40] compared conversational agents (CAs), classifying them by domain, architecture (modular vs. end-to-end), and goals (task-oriented vs. open-goal). While insightful, their analysis is limited to dialogue-driven agents rather than the broader spectrum of MAS in SE.

The most relevant work to ours is the taxonomy proposed by Händler et al. [41] that examined the interplay between autonomy and alignment with human users' goals. In particular, this work introduces a comprehensive multi-dimensional taxonomy that adheres to the 4+1 view model [42] of software architecture tailored to agent composition and orchestration. On top of the proposed architectural model, a set of feature models has been depicted and discussed. To evaluate the expressiveness, seven different frameworks have been discussed using the proposed taxonomy. Their work emphasizes agent autonomy and orchestration but does not integrate practical reusability aspects such as agent discovery, benchmarking, or metadata standardization, which are essential in SE.

Beyond academic works, several industrial initiatives have emerged to provide practical techniques and tools for managing AI agents, each offering different levels of integration, categorization, and operational utility. These initiatives are particularly relevant in the context of the *AgentWare* paradigm, as they attempt to standardize how agents are described, discovered, and reused across diverse environments. Notable examples include:

- The **VS Code AI Toolkit**[8] provides a structured interface to explore AI models directly from the development environment. It supports classification by task (e.g., summarization, classification, generation), model type, and underlying framework (e.g., PyTorch, TensorFlow). The toolkit aims to lower the barrier for integrating AI into software development by offering in-editor previews, and deployment instructions. Its taxonomy is implicitly defined through tags and filters, which align with software engineering tasks and user needs.

- The **Agent2Agent (A2A) Protocol**[9] introduces a formal specification for AI agent interaction and capabilities. Central to this protocol is the `AgentSkill` object, which represents functional competencies of agents and supports skill discovery, exchange, and composition across platforms. The protocol promotes modularity and interoperability among agents, making it possible to compose multi-agent systems using components from different vendors or communities. The A2A specification effectively serves as a skill-based taxonomy, mapping capabilities to operational semantics.

- Several online marketplaces, such as **BeeAI**[10], **AgentLocker**[11], and **AI Agents List**[12], host a growing ecosystem of deployable AI agents. These platforms categorize agents by domain (e.g., finance, marketing, legal), task (e.g., data extraction, writing, summarization), and deployment model (cloud-hosted, downloadable, API-accessible). Although their classification schemes vary in granularity and completeness, they provide useful metadata such as input/output format, licensing, provider, and performance indicators. However, many still lack standardized benchmarking or detailed evaluation features, limiting the ability to compare agent quality across platforms.

[8] https://code.visualstudio.com/docs/intelligentapps/models
[9] https://a2a-protocol.org/latest/specification/
[10] https://beeai.dev/agents
[11] https://www.agentlocker.ai/
[12] https://aiagentslist.com/

- The taxonomy proposed by **Agntcy**[13] offers an ontology-inspired view of agent capabilities. Their framework, part of the Open Agent System Framework (OASF), organizes agent skills into hierarchical categories, including core capabilities (e.g., language understanding, planning), interface types (e.g., chat, CLI, API), and operational scope (e.g., one-shot, multi-turn, continuous agents). This taxonomy is explicitly designed to support interoperability, lifecycle management, and agent discovery across evolving ecosystems. It serves as a reference for designing classification modules in MAS repositories such as MOSAICO.

These industrial efforts demonstrate a growing interest in building AI agents and in establishing shared conventions for their management and reuse. They offer complementary perspectives to academic taxonomies, emphasizing developer usability, platform interoperability, and marketplace readiness. As such, they provide valuable inspiration for defining the metadata, categorization dimensions, and discovery mechanisms needed in the design of repositories like the one envisioned in the MOSAICO platform.

2.3 Conclusion

The analysed sources demonstrate a growing ecosystem of classification schemes; however, they remain fragmented and often lack support for quality attributes, governance policies, or SE-specific tasks.

The MOSAICO taxonomy builds upon these foundations by unifying academic and industrial perspectives. It integrates agent capabilities, orchestration patterns, governance rules, monitoring features, and reusability aspects, aligning with the repository metadata (see Section 6). This approach ensures that the taxonomy supports both *research needs* (e.g., comparison and analysis) and *practical adoption* (e.g., recommendation, interoperability) in the context of software engineering.

[13] https://docs.agntcy.org/oasf/taxonomy/

3 Analysis of MAS Frameworks

This section presents the analysis of existing frameworks for developing multi-agent systems (MAS), focusing on their foundational concepts and features. The goal is to identify the key dimensions that characterize MAS, which will serve as a basis for the subsequent development of the MOSAICO Repository. By analyzing these frameworks, we aim to understand how they define and implement core concepts such as agents, roles, communication mechanisms, and governance rules.

The analysis process involved a systematic literature review to identify relevant papers and frameworks. Concerning the scientific papers, we employed a combination of keyword searches, citation tracking, and expert recommendations to compile a comprehensive list of sources. Each selected paper was then analyzed to extract key information related to the foundational concepts of MAS. Concerning the selection of MAS frameworks and tools, we searched the `GitHub` platform for open-source projects related to MAS. We focused on high-quality repositories that are well-maintained and have a high number of stars and forks [43]. Table 1 provides an overview of various multi-agent system (MAS) frameworks, highlighting their paradigms, release dates, licenses, and popularity metrics, i.e., stars, forks, contributors, and dependent projects.

Table 1: Overview of MAS frameworks.

ID	Name	First release (Total)	License	# Stars	# Forks	# Contrib.	# Dep.
MF1	AutoGPT	Apr 12, 2023 (53)	MIT	174,000	45,400	781	1
MF2	LangChain[a]	Jan 16, 2023 (879)	MIT	105,000	17,000	3,784	270,534
MF3	Dify	May 15, 2023 (130)	Apache 2.0	85,700	12,700	993	0
MF4	MetaGPT	Jul 15, 2023 (22)	MIT	53,500	6,300	113	118
MF5	AutoGen	Sep 19, 2023 (87)	CC-BY-4.0, MIT	42,100	6,300	558	3,822
MF6	Llama Index	Jan 29, 2023 (440)	MIT	40,400	5,700	1,679	22,801
MF7	Flowise	May 30, 2023 (67)	Apache 2.0	36,500	19,100	256	0
MF8	CrewAi	Nov 14, 2023 (82)	MIT	29,000	3,900	273	17,032
MF9	Semantic Kernel	Apr 25, 2023 (203)	MIT	23,700	3,600	411	2,177
MF10	Agno	Jan 30, 2025 (54)	MPL 2.0	21,900	2,900	297	0
MF11	Haystack	Nov 28, 2019 (165)	Apache 2.0	20,000	2,100	309	1,181
MF12	OpenAI SDK[b]	Mar 11, 2025 (13)	MIT	19,400	2,100	15	0
MF13	Smolagents	Dec 27, 2024 (20)	Apache 2.0	15,700	1,400	183	0
MF14	Letta[c]	Oct 27, 2023 (163)	Apache 2.0	18,482	1,922	142	0

[a] LangChain is the backend of LangGraph `https://www.langchain.com/langgraph`
[b] OpenAI SDK has replaced SwarmAI
[c] Letta (formerly MemGPT)[44]

Table 2: Foundational Concepts (FC) of MAS Frameworks.

Alias	Aspect	Description
FC1	Core Architecture	The pivotal elements of a MAS, i.e., agents, tools, memory, and the application domain. A MAS can be homogeneous or heterogeneous depending on whether the agents share the same architecture.
FC2	Governance Rules	Policies or orchestration rules integrated into the platform, including interfaces for agent-environment interaction.
FC3	Agent Monitoring	Evaluation of agents' performance, feedback mechanisms, and continual evolution capabilities.
FC4	Behavioral Model	It describes agents' roles, objectives, and interactions with the environment and other agents.
FC5	Reusability	Capabilities provided by frameworks, such as agent deployment, discovery features, and tool integration.

3.1 Conceptual Taxonomy

The analysis of the elicited research papers and selected frameworks revealed five foundational concepts (FC1-FC5) that are essential for understanding and implementing MAS frameworks. These concepts encompass *core architecture*, *governance rules*, *agent monitoring*, *behavioral models*, and *reusability features*. The five dimensions consist of 21 sub-concepts that can be used to characterize the MAS frameworks as detailed in Table 3. The findings will inform the design of the MOSAICO Repository, ensuring that it supports the necessary features for managing and orchestrating AI agents in software engineering contexts. In the following, we provide a detailed analysis of the taxonomy dimensions and their corresponding sub-concepts.

Core architecture concepts (FC1.1-FC1.5)

The majority of the analyzed studies agree on the essential components of MAS systems, i.e., agents, tools, and memory. With respect to the application domain (FC1.1), this concept represents the high-level goal of the whole MAS that may be defined as a composition of the single agent's objectives (FC4.1), defined in *Behavioral model*. Therefore, none of the frameworks allows for specifying the application domain directly, but it is possible to specify the agents' objectives (FC4.1), external tools to rely on (FC1.5), and roles (FC4.4) that can be used to accomplish the task.

Agents (FC1.4) are the core components of any MAS system, and they can be homogeneous or heterogeneous depending on whether they share the same architecture. In particular, *Homogeneous vs. Heterogeneous (FC1.2)* refers to the possibility of defining a MAS with agents that share the same architecture or not. Notably, only three research papers, i.e., P3, P7, and P13 explicitly define the difference between homogeneous and heterogeneous MAS. Meanwhile, all the frameworks allow the definition of heterogeneous MAS, i.e., the possibility to deploy agents with different providers and architectures, e.g., GPT, Llama or Claude.

The *Tools (FC1.5)* concept represents the external resources that agents can use to accomplish their tasks, whereas *Memory (FC1.3)* is used to store information about the agents' state and interactions with the environment.

Table 3: Elicited taxonomy from literature and MAS frameworks analysis.

Dim.		Description	Proposed by the analyzed Papers (P) and MAS Frameworks (MF)
FC1	FC1.1 - Application domain	The main task that the MAS is designed to solve	P1, P2, P3, P4, P5, P6, P7, P9, P11, P13, P14, P15, P18
	FC1.2 Type	It identifies if the system is homogeneous or heterogeneous	P1, P7, P13, MF1, MF2, MF3, MF4, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
	FC1.3 Memory	Knowledge store of the MAS, shared or private	P1, P2, P3, P4, P6, P7, P8, P9, P11, P12, P15, P17, P18, MF1, MF2, MF3, MF4, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF13
	FC1.4 Agent	Generic AI agent with objective, role and environment interface	P1, P2, P3, P4, P5, P6, P7, P8, P9, P11, P12, P13, P14, P15, P16, P17, P18, MF1, MF2, MF3, MF4, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
	FC1.5 Tool	External APIs or services that agents can exploit	P1, P2, P3, P4, P7, P8, P11, P15, P18, MF1, MF2, MF3, MF4, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
FC2	FC2.1 - Coordination Patterns	It defines how agents interact with other agents	P1, P2, P4, P6, P7, P9, P11, P12, P13, P15, P14, P16, P17, P18, MF1, MF2, MF3, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
	FC2.2 - Communication Mechanism	It defines the type of communication at the MAS level	P1, P2, P4, P6, P7, P9, P11, P12, P13, P14, P15, P16, P17, P18, MF1, MF2, MF3, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
	FC2.3 - Agents–Environment Interface	The specific contexts in which the MAS systems are deployed, e.g., Sandbox, Physical, or Text-based.	P2, P4, P7, P8, P11, P12, P14, P18, MF1, MF2, MF3, MF5, MF6, MF7, MF9, MF10, MF11, MF12, MF13
	FC2.4 - Competency mapping	Mapping between agent roles and external tools	P1, P2, P3, P4, P7, P8, P11, P15, P18, MF1, MF2, MF3, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
FC3	FC3.1 - Value-addition modeling	KPI-oriented assessment of agents (performance, cost, scalability)	P1, P2, P4, P5, P6, P15, P18, MF2, MF6, MF7, MF13
	FC3.2 - Feedback	Specific functionalities to embody human feedback	P1, P2, P4, P5, P6, P7, P11, P12, P18, MF2, MF3, MF4, MF5, MF6, MF8, MF9
	FC3.3 - Continual evolution	It involves the agents' ability to evolve continuously, adapting their parameters, memory, and objectives.	P2, P4, P6, P7, P8, P15, MF14
FC4	FC4.1 - Objective	The goal that a single agent aims to achieve	P1, P2, P3, P4, P5, P6, P7, P8, P9, P11, P12, P13, P14, P15, P16, P17, P18, MF1, MF2, MF3, MF4, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
	FC4.2 - Perception	Ability to sense and interpret the environment	P1, P2, P3, P4, P7, P9, P11, P12, P15, P16, P18, MF1, MF3, MF4, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
	FC4.3 - Action	Atomic executions (tool usage, communication)	P1, P2, P3, P4, P5, P6, P7, P8, P9, P11, P12, P13, P14, P15, P16, P17, P18, MF1, MF2, MF3, MF4, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
	FC4.4 - Role	Behaviour specification (objectives, actions, final goal)	P1, P2, P3, P4, P5, P6, P7, P8, P9, P11, P12, P13, P14, P15, P16, P17, P18, MF1, MF2, MF3, MF4, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
	FC4.5 - Message Content	Pre-defined outputs an agent sends to others	P2, P3, P6, P7, P12, P14, P15, MF1, MF2, MF3, MF4, MF5, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
FC5	FC5.1 - Remote access	Access to agents/tools via services	MF1, MF2, MF3, MF6, MF7, MF8, MF9, MF10, MF11, MF12, MF13
	FC5.2 - Agent comparison	Functionalities to compare agents (offline or at runtime)	MF2, MF3, MF6, MF7, MF12, MF13
	FC5.3 - Benchmarking	Re-using benchmark datasets for evaluation	MF2, MF3, MF5, MF11, MF13
	FC5.4 - Discovery	Functionalities to elicit agents/tools from an available pool	MF2, MF3, MF7

Governance rules (FC2.1-FC2.4)

The governance rules are the core of any MAS system. *Coordination Patterns (FC2.1)* are rules that define how agents interact with each other and with the environment. This concept is well covered by the literature, and supported by some of the analyzed frameworks. For instance, LlamaIndex [19] allows developers to define a set of coordination patterns in terms of rules, as shown in Figure 3, which shows an example of a cooperative pattern involving multiple agents to support the summarization process described in Section 4.1. In particular, the round nodes represent events, the square ones the different agents, and the edges the transition between them. Agents decide when to invoke tools, when to stop, and whether to hand off control to another agent. While this enables more flexible and modular behaviors, it also introduces a higher dependency on the LLM's ability to follow instructions and maintain internal consistency.

Communication Mechanism (FC2.2) refers the protocol for agent communication, e.g., centralized, decentralized, or shared message pool. This concept is also well covered by the literature, and by the considered frameworks. These two features are heavily connected, since any coordination pattern needs a proper communication mechanism to be deployed.

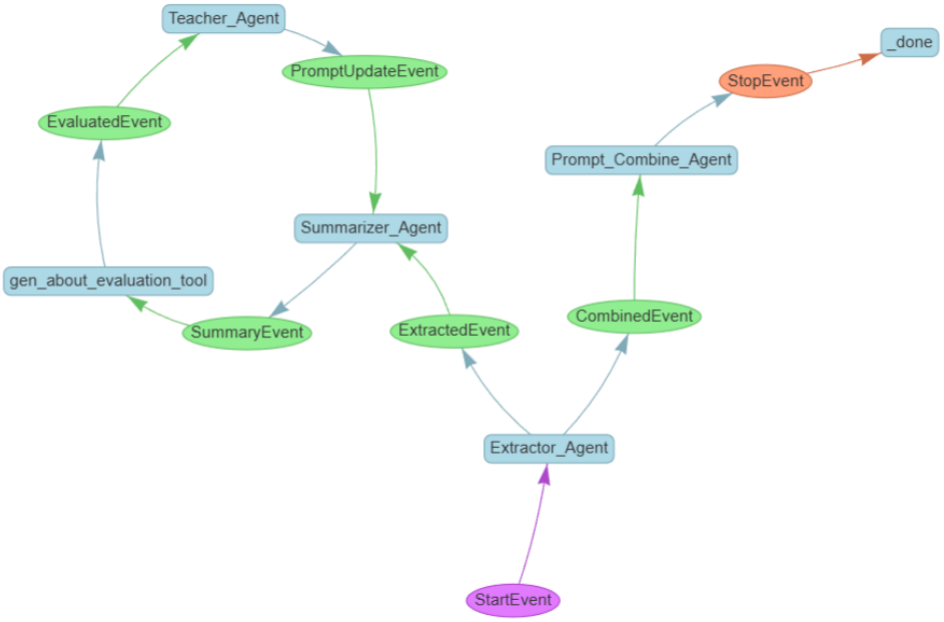


Figure 3: Example of Coordination Patterns in the LlamaIndex framework. [MF6]

Agent-Environment Interface (FC2.3) plays a crucial role in defining the specific contexts in which multi-agent systems are deployed and interact in the operational environments. In particular, a specific environment impacts sensory inputs, action space, and interactive potential of agents. Consequently, agents influence the state of the environment through their behaviors and decisions. Such interface can be of three types, i.e., *Text-based*, *Virtual sandbox*, or *Physical*, depending on the application domain and the requirements of the system. *Text-based* refers to the interaction with the environment through natural language descriptions without direct involvement of other modalities, representing the most natural platform for agents to operate in. In particular, entities and resources can be presented in plain natural language or

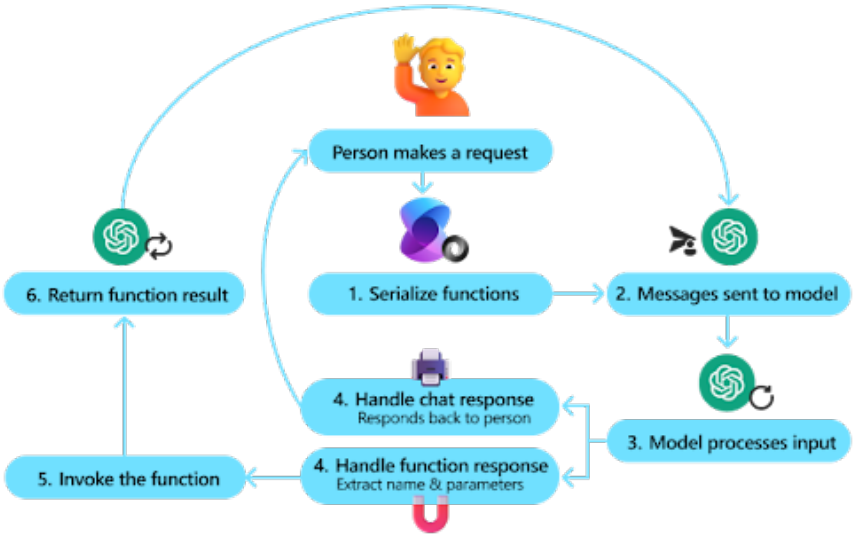


Figure 4: Example of Competency Mapping in the Semantic Kernel framework. [MF9]

structured formats, e.g., JSON or YAML. On the contrary, *Virtual sandbox* provides a visualized and extensible platform for agent society, bridging the gap between simulation and reality. In particular, this kind of environment provides a visual representation of the agents, spanning from simple 2D interfaces to complex and immersive 3D environments. For instance, avatars can be used to represent agents in those type of environments, enabling real-time tracking of movement and interactions. This kind of environment are tailored for specific application, such as gaming or robotics. Finally, the *Physical* environment refers to the interaction with the real world, where agents can interact with physical objects and entities. However, designing this kind of interface is challenging as developers needs to process a huge number of sensory inputs and physical constraints, e.g., moving a robotic arm to grasp objects.

While eight papers explicitly mention the *Agent-Environment Interface* concept, all frameworks allow only text-based interfaces. Nevertheless, we believe that those systems can be adapted to support other types of interfaces, such as physical or sandboxed ones, by implementing the necessary components and protocols.

Competency Mapping (FC2.4) refers to the set of rules that define which agents can employ a set of external tools. This concept is not fully covered by the literature, while the frameworks provide and implement this concept in practice. In particular, the competency mapping is used to define which agents can use which tools, thus enabling a more flexible and modular approach to agent design. For instance, in the Semantic Kernel framework [20], agents can be configured to use specific tools based on their roles and objectives, as shown in Figure 4.

Agent monitoring (FC3.1-FC3.3)

According to the analyzed literature, the monitoring of agents is a crucial aspect of MAS systems. *Value-addition modeling (FC3.1)* represents a generic KPI-oriented assessment of agents, spanning from the accuracy metrics to advanced qualitative aspects, e.g., energy consumption, cost, and execution time. However, most of the considered frameworks neglect this aspect. The support for human-in-the-loop functionalities *(FC3.2)* (e.g., explicit user feedback

or well-established human-agent interaction protocols) is also limited. Interestingly, we report that none of the tools has a specific component for *Continual evolution (FC3.3)* of the agents, apart from Letta MF14, even though this concept is mentioned in the literature. Our intuition is that those qualitative aspects are not yet fully implemented in the frameworks, triggering the need for further improvements.

Behavioral model (FC4.1-FC4.5)

The Behavioral model and the related sub-features (FC4.1-FC4.5) are well covered by the literature and the considered frameworks. It encompasses all the behavioral features related to agents, spanning from the definition of roles to interaction with the environment and other agents. The reason behind this finding is that all those features have been used already in single-agent systems.

The definition of the agent's *Objective (FC4.1)* represents the minimal concepts of any MAS whereas *Perception (FC4.2)* is bounded by the context in which the agent operates. *Action (FC4.3)* is also well-defined in the literature, providing a clear understanding of how agents interact with each other and how they communicate. A pivotal feature that impacts the agents' behavior is the role (FC4.4), which specifies all relevant information needed to perform the objective.

Listing 1: Example of agent roles in CrewAI tool. [MF8]

```
researcher:
  role: >
    {topic} Senior Data Researcher
  goal: >
    Uncover cutting-edge developments in {topic}
  backstory: >
    You're a seasoned researcher with a knack for uncovering the latest developments in {topic}.
    Known for your ability to find the most relevant information and present it in a
    clear and concise manner.

reporting_analyst:
  role: >
    {topic} Reporting Analyst
  goal: >
    Create detailed reports based on {topic} data analysis and research findings
  backstory: >
    You're a meticulous analyst with a keen eye for detail. You're known for your
     ability to turn complex data into clear and concise reports, making it easy for
    others to understand and act on the information you provide.
```

Listing 1 illustrates an example of role specification using the CrewAI framework MF8 for performing a basic research-oriented task. In this case, the configuration is expressed as a YAML file,[14] which serves as a human-readable format for defining the structure, attributes, and behavior of agents. The listing defines two distinct agents, each associated with a specific role: *researcher* and *reporting_analyst*. Each role encapsulates three essential components:

- *role*: this field provides a high-level functional title for the agent, expressed as a prompt string that can be parameterized with a domain-specific topic (e.g., AI ethics, software

[14] https://yaml.org/

```
GroupChat: Summarizer - Evaluator - Teacher

# Summarizer: React Test-Driven Development Guide

# Evaluator: The ROUGE-L score for the recent summary "React Test-Driven Development Guide" com
pared to the ground truth is approximately 0.14. Since this score is higher than the previous b
est score of 0.0, the prompt has been updated to the most recent one provided by the Teacher.

# Teacher: Summarize the following extracted text from a Github repository README into a short
term/phrase introducing the repository: If the text contains an existing tagline, functional de
scription, purpose statement, or overview, prioritize using it. The output should include only
a short term/phrase introducing the repository.

# Summarizer: A series of examples on how to TDD React

# Evaluator: The recent summary "A series of examples on how to TDD React" achieved a perfect R
OUGE-L score of 1.0 when compared to the ground truth. This score is higher than the previous b
est score of approximately 0.14, so the prompt has been updated to the most recent one provided
 by the Teacher.
```

Figure 5: Example of Message Content in the Semantic Kernel Framework. [MF9]

architecture, etc.). This abstraction facilitates the reuse of role definitions across different application domains.

- *goal*: it specifies the overarching objective of the agent in natural language. In this example, the *researcher* is tasked with uncovering cutting-edge developments in a given topic, while the *reporting_analyst* is responsible for transforming the research outcomes into detailed, accessible reports.
- *backstory*: it describes the agent's persona, expertise, and working style. This narrative context is leveraged by the underlying LLM to inform its responses and decision-making behavior, enriching the overall coherence and alignment of the agent with its role.

This approach exemplifies a key feature of the *AgentWare* paradigm, where agents are instantiated from declarative and prompt-based templates that guide their decision logic and interaction strategies. The modular and readable YAML syntax simplifies the definition and orchestration of multiple agents in a MAS, supporting a "plug-and-play architecture" in which roles and tasks can be easily extended or modified.

Message Content (FC4.5) refers to the content of the messages exchanged between agents and the environment. This concept is not explicitly covered by the literature, but it is a crucial aspect of any MAS system, since it defines how agents communicate with each other and with the environment. All the frameworks provide this specific feature, allowing for the ingesting of different types of well-known file output formats, e.g., JSON or YAML. For instance, Figure 5 shows an example of the message content in the Semantic Kernel framework, where the message is defined as a structured textual file. In particular, the output highlights the different agents involved in the reasoning process to perform a summarization task.

Reusability (FC5.1-FC5.4)

Reusability of agents is a crucial aspect of MAS frameworks, enabling developers to leverage existing components and functionalities. The analysed research papers highlight the importance of reusability features, but there is a lack of systematic studies addressing this issue. Instead, most frameworks provide capabilities that can be used to enhance agent reusability.

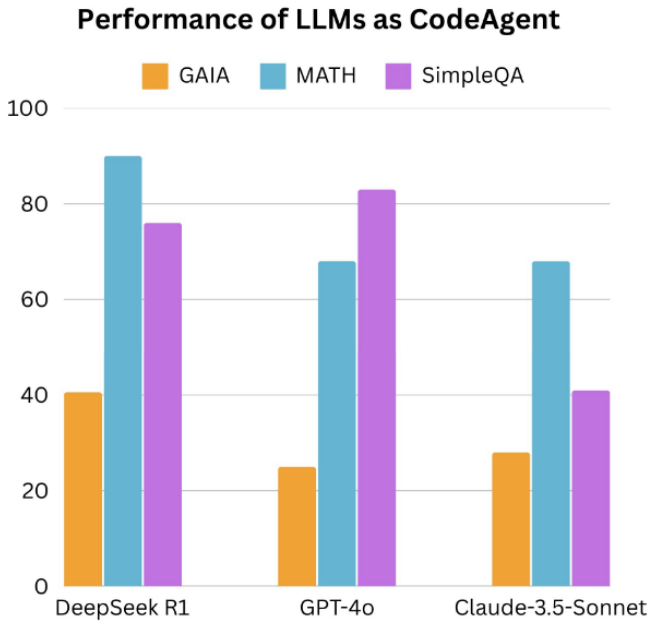


Figure 6: Example of Benchmarking in the Smolagents framework. [MF13]

Remote access (FC5.1) and *Agent comparison (FC5.2)* are covered by the majority of the frameworks. On the contrary, *Benchmarking (FC5.3)*, i.e., dedicated functions to import and manipulate datasets and measuring agents' performance, and *Discovery (FC5.4)*, i.e., capabilities to select and categorize agents, are still neglected by most of the frameworks.

The benchmarking aspect is crucial for the evaluation of the system, since it allows reusing well-known datasets and to compare the performance of different agents. For instance, Smolagents [21] provides a direct integration with the Hugging Face platform and manipulate datasets from different sources, e.g., the agents medium benchmark.[15] Such datasets can be used to compare different models as shown in Figure 6. The bar chart reports the performance of three different LLMs, `DeepSeek R1`, `GPT-4o`, and `Claude-3.5-Sonnet`, acting as CodeAgents, evaluated across three datasets: `GAIA`[16], `MATH`[17], and `SimpleQA`[18]. The benchmark results show considerable performance variability depending on both the task and the model. This highlights the need for a structured benchmarking pipeline to evaluate agent capabilities in a reproducible and comparable way. In Smolagents, such benchmarking is integrated directly into the agent orchestration pipeline, enabling developers to assess model suitability before deployment.

Concerning discovery capabilities, they encompass the possibility to elicit new agents/tools from a pool of available ones according to different filters, e.g., category, hyperparameters, or hardware and software requirements. For instance, in Figure 7 we show an example of the discovery process in the Dify framework [22], where the system is able to elicit new agents/tools from a pool of available resources, namely the Dify Marketplace.

Although these aspects are still neglected by the majority of the analyzed frameworks, we believe that advanced discovery and reusability features are crucial for the future development of MAS systems, since they provide a clear understanding of the performance of the system and the possibility to elicit new agents/tools from existing marketplaces.

[15] `https://huggingface.co/datasets/m-ric/agents_medium_benchmark_2`
[16] `https://huggingface.co/datasets/gaia-benchmark/GAIA`
[17] `https://huggingface.co/datasets/HuggingFaceTB/MATH/tree/main`
[18] `https://openai.com/index/introducing-simpleqa/`

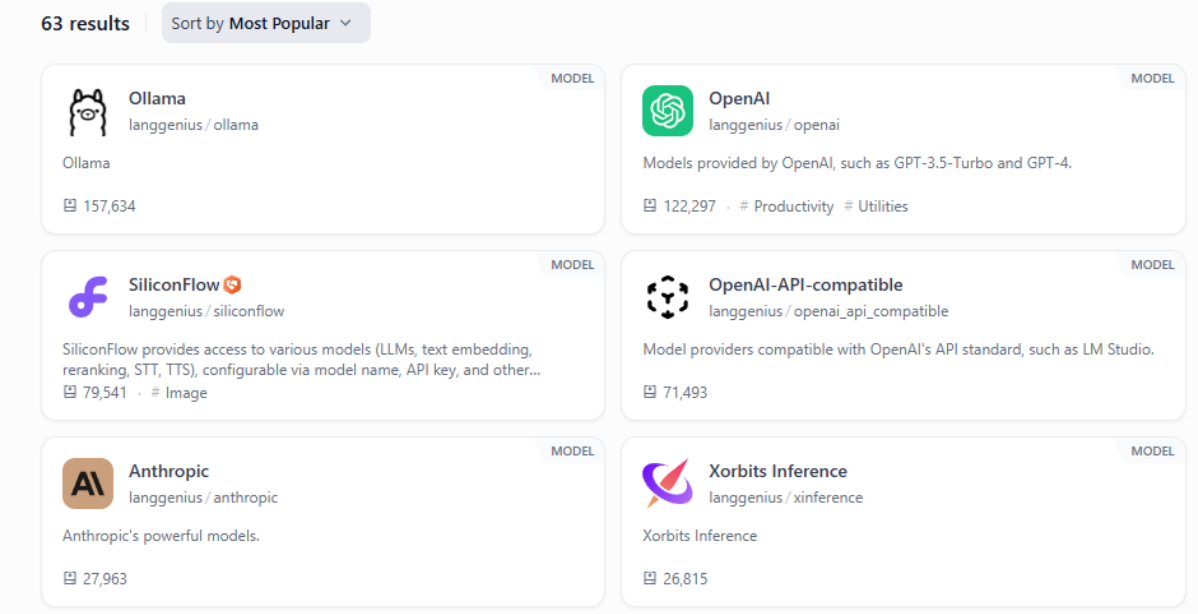


Figure 7: Example of Discovery process in the Dify framework. [MF3]

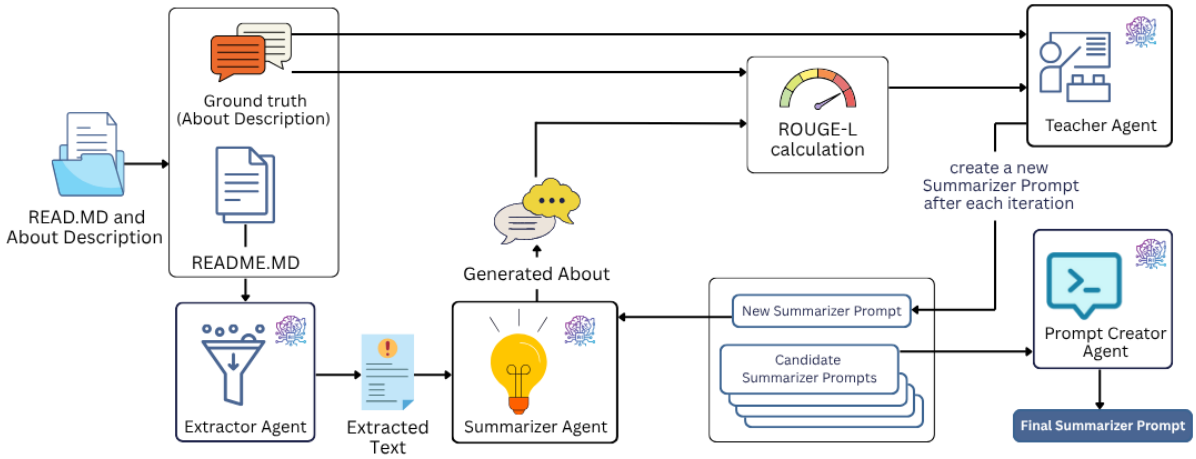
3.2 Conclusion

Our analysis reveals a strong alignment between scientific surveys and practical frameworks in terms of the core architectural components of multi-agent systems, notably agents, tools, and memory. However, discrepancies emerge when examining advanced features: while conceptual elements such as continual evolution procedures and application domain specification are acknowledged in the literature, they are seldom supported in current technical implementations. Conversely, practical aspects related to reusability, such as agent deployment, comparison, and tool integration, are more widely implemented in existing frameworks. Nevertheless, critical capabilities like benchmarking and agent/tool discovery remain underdeveloped or inconsistently supported. This gap highlights the need for further integration of advanced and evaluative functionalities to fully realize the AgentWare paradigm in software engineering.

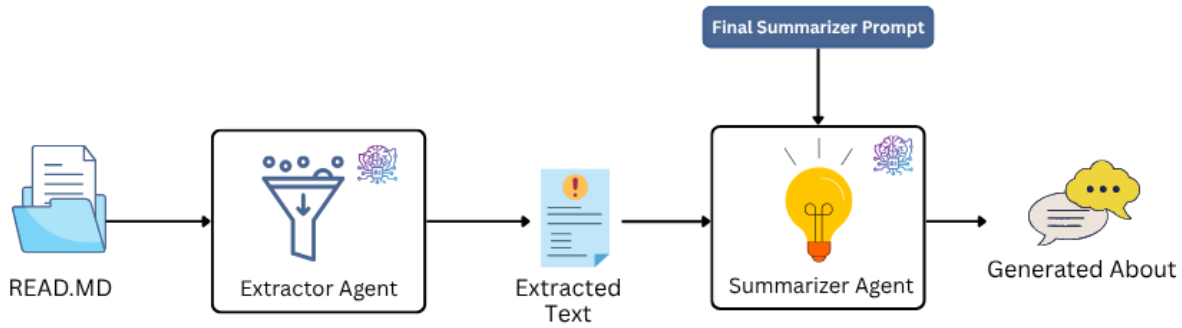
4 Experience Report on using MAS Frameworks

To further analyze the capabilities of existing MAS frameworks, we conduct both qualitative and quantitative evaluations by selecting the most representative frameworks from the list shown in Table 1. Both analyses have been performed by considering an illustrative multi-agent system presented in the next section.

4.1 METAGENTE: An illustrative multi-agent system



(a) Optimization Pipeline



(b) Evaluation Pipeline

Figure 8: Pipelines used in the multi-agent system.

To conduct the performed evaluations, we first introduce the METAGENTE multi-agent system that implements a concrete scenario, i.e., the summarization of `GitHub` README.MD files that have been recently explored by the SE community [45, 46]. While this task can be accomplished by a single agent, it can be enhanced by the collaboration of multiple agents as recently demonstrated by Nguyen et al. [47]. Four different agents have been used to accomplish the task, i.e., the *Extractor*, *Summarizer*, *Teacher*, and *Prompt Creator* agents.

As shown in Figure 8, in the original work [47] the four agents are involved in two different pipelines, i.e., the *Optimization Pipeline* and the *Evaluation Pipeline* that are implemented by leveraging the LangChain framework [18] and the GPT-4o-mini model as detailed below.

Optimization Pipeline

Figure 8a illustrates the Optimization Pipeline with four agents: `Extractor Agent`, `Summarizer Agent`, `Teacher Agent`, and `Prompt Creator Agent`. Throughout the pipeline, each agent receives a task-specific prompt that guides its behavior. These prompts were adapted from the ones used in the baseline experiment referenced earlier, with minor modifications introduced to accommodate the specific requirements and constraints of each framework. For the sake of references, the complete set of prompts used for each framework is presented in the online appendix [48].

▷ `Extractor Agent` is responsible for identifying and filtering relevant information from a raw README.MD file by removing noisy or irrelevant content. Its primary goal is to extract a concise description of the repository by omitting sections such as installation instructions and configuration details, which do not add much information to the summarization. This preprocessing step helps reduce computational cost and execution time according to the base study. The agent receives a full README.MD file as input and outputs a shortened, descriptive version of the repository content.

▷ `Summarizer Agent` generates a summary based on the extracted repository description. Unlike traditional summarization agents with a fixed prompt, this agent uses a dynamic prompt that is updated at each iteration by `Teacher Agent`. The only fixed prompt is provided by the user during the first iteration, as no feedback from `Teacher Agent` is available yet. In subsequent iterations, `Summarizer Agent` receives a revised prompt intended to improve its performance. The agent takes two inputs: the summarization prompt and the extracted repository description. Its output is a generated summary of the README.MD file.

▷ `Teacher Agent` is tasked with improving the prompt used by `Summarizer Agent` at each iteration. To do so, it receives the following inputs: the summary generated by `Summarizer Agent`, the prompt used in that iteration by the Summarizer, the ROUGE-L score [49] comparing the generated summary with the ground truth, the ground truth summary itself, and the original extracted text provided by `Extractor Agent`. Based on this information, the Teacher analyzes the inputs and produces a refined prompt to be used in the next iteration of the optimization process by `Summarizer Agent`.

▷ `Prompt Creator Agent` is responsible for the creation of the final prompt. Throughout the optimization task, several README.MD files are processed to generate candidate prompts. However, only the prompts that produce summaries with a ROUGE-L score equal to or greater than a predefined threshold during the iterative process are selected. `Prompt Creator Agent` collects these high-performing prompts and combines them into a single, representative prompt that captures the most effective features across examples. The output of `Prompt Creator Agent` is the final output of the pipeline, and it is the outcome of the optimization phase. This prompt will be once again utilized in the Evaluation phase.

As illustrated in Figure 8a, the workflow of the agents unfolds as follows:

1. A README.MD file is provided to `Extractor Agent`, which generates a concise version of the text to be used as input for `Summarizer Agent`.
2. Optimization Loop:
   (a) `Summarizer Agent` receives the extracted text and produces a summary based on its current prompt.
   (b) A ROUGE-L score is then calculated by comparing the generated summary with the ground truth summary. If the score is equal to or exceeds a predefined threshold, the

optimization loop is interrupted, and the current prompt used by `Summarizer Agent` is saved as a candidate prompt.

(c) If the ROUGE-L score is below the threshold, `Teacher Agent` receives the inputs from `Extractor Agent` and `Summarizer Agent` along with the ROUGE-L score. It uses this information to generate a new prompt, which is then passed to `Summarizer Agent` for the next iteration.

(d) The loop continues until either the threshold is met or the maximum number of predefined iterations is reached.

3. Once the loop ends, a new README.MD file is passed through the same pipeline to generate another candidate prompt.

4. After all README.MD files have been processed, `Prompt Creator Agent` collects all candidate prompts that achieved a ROUGE-L score above the defined threshold and aggregates them into a single, final prompt.

Evaluation Pipeline

Once the optimized prompt has been obtained from the Optimization Pipeline, the evaluation phase can be initiated. As illustrated in Figure 8b, the Evaluation Pipeline is comparatively simpler and consists of only two agents: `Extractor Agent` and `Summarizer Agent`. The former uses the same prompt and performs the same function as in the optimization phase, processing the README.MD file to extract a concise description. The latter, however, uses the final optimized prompt produced in the previous phase for all summarization. After the summarization, ROUGE-1, ROUGE-2, and ROUGE-L were collected to evaluate the quality of the generated summaries against their respective ground truth references. These metrics are collected from the evaluation of several README.MD files so that the optimization task could be assessed.

In the following we report on the experience we have performed by reimplementing the pipelines shown in Figure 8 by means of different MAS frameworks. The main objective of the performed experience is not to evaluate the involved LLMs themselves or the pipeline design per se, but rather to assess the capabilities of each framework in developing and executing the same pipelines. This includes measuring performance using ROUGE scores, collecting model usage data, and reporting the developer experience during the implementation and usage phases.

4.2 Qualitative Analysis

We report the main advantages and disadvantages of each tool from a developer's perspective, with respect to the technical dimensions shown in Table 4. The frameworks selected for analysis have been chosen from the initial list in Table 1 by considering only those that *(i)* have a *permissive license*, e.g., developers can reuse the tool without any subscription or commercial license; *(ii)* have a *large number of stars and forks*, i.e., they are well-maintained and have a large community of developers according to the number of stars and forks; and *(iii)* provide a *comprehensive documentation and examples* to support developers in using the tool.

By applying these criteria, we ended up with 8 frameworks (i.e., AutoGPT, AutoGen, Dify, Flowise, Heystack, LlamaIndex, OpenAI SDK, and SemanticKernel) that have been evaluated by reimplementing the METAGENTE scenario presented in Section 4.1.

Table 4: The Analysed Technical Dimensions.

Alias	Dimension	Description
D1	Installation	It describes how to install the tool, including any dependencies or prerequisites that need to be met. There are also instructions for setting up the tool in a local environment, as well as any necessary configurations.
D2	Developer Interface	This concerns the ease of use and intuitiveness of the supported interface. It is also related to the quality of the documentation provided, including its clarity, completeness, and organization. This includes evaluating the availability of tutorials, examples, and API references.
D3	Model and Tools Integration	Support for integrating with different AI models, including LLMs and other types of models, as well as external services. We assess the flexibility of the tool in terms of model selection and integration, as well as its ability to work with different model architectures. This includes evaluating the availability of pre-built integrations and the flexibility of the tool in terms of custom integration.
D4	Agent Creation	Support for creating and managing agents, i.e., the ease of creating new agents, including the availability of templates and examples. In addition, this concerns how role and external knowledge are defined in the tool.
D5	Agent Orchestration	Support for orchestrating agents and managing their interactions, i.e., the ease of defining workflows and coordinating multiple agents, including the availability of pre-built orchestration patterns.
D6	Monitoring	Support for monitoring and debugging agents, i.e., the availability of monitoring tools, including logging and telemetry features. This includes evaluating the ease of tracking agent performance and identifying issues during development and deployment.

4.2.1 D1 - Installation

Table 5 summarizes main installation aspects for each framework, highlighting the setup method and the quickstart support in terms of documentation, tutorial, or similar materials. Overall, all the frameworks provide well-structured documentation, even though at different levels of detail. In terms of installation, most of the frameworks can be installed using common package managers, e.g., `pip` or `npm`, while others require Docker to be run, especially the one classified as low-code. In addition, some frameworks provide a web-based dashboard to facilitate the development and management of MAS applications.

▷ **AutoGPT**. The tool can be used in two ways, i.e., locally or in the cloud. However, the cloud service is still not available for common users since the tool is still in beta version, and thus not open to all users. As a result, the tested version was run locally using Docker for the backend and `npm` (Node Package Manager) for the front end. Concerning the quickstart support, a brief tutorial introduces them to the basic functions of the tool. However, the documentation, which provides additional examples of the tool's potential uses, is quite limited, making it challenging to grasp how the system operates and what it can achieve.

▷ **AutoGen**. The setup of the tool is straightforward, using common `pip` commands which will install the AgentChat and Extension packages in their Python environment. The documentation provides simple examples of developing agents with the tool, such as agents counting numbers or providing weather information. Additionally, tutorials can be found online, even though a new version was released in early 2025, rendering them outdated.

▷ **Dify**. Similar to AutoGPT, Dify can be run locally using Docker, or it can be deployed in the cloud with either a free plan or a paid option. On the one hand, the cloud tool requires minimal setup, and it has some limitations on the number of apps, requests, and memory according to

Table 5: Installation.

Framework	Local	Cloud	Docker	Quickstart support
AutoGPT	✔	✖	✔	Documentation, Tutorials
AutoGen	✔	✖	✖	Documentation, Tutorials
Dify	✔	✔	✔	Documentation, Templates
Flowise	✔	✖	✔	Tutorials
Haystack	✔	✖	✖	Documentation
LlamaIndex	✔	✖	✖	Visualization
OpenAI SDK	✔	✖	✖	Documentation
SemanticKernel	✔	✖	✖	Jupyter notebooks, Examples

the chosen plan. On the other hand, the self-hosted version is free, but it requires more effort in the setup. In this respect, the documentation is well-structured and drives the users through the whole process properly.

▷ **Flowise**. The framework requires familiarity with JavaScript development, particularly the use of `npm` using the standard installation process. This method involves setting up the local environment manually, which may pose a barrier for beginners or people with no experience with JS technologies. Alternatively, Flowise also provides Docker images that can be run with minimal configuration. Concerning the documentation, Flowise provides some quick-start material, although it is not very comprehensive compared to other frameworks. In particular, a detailed rationale behind some components and their intended usage is sometimes missing. This makes it difficult for beginners to fully grasp how to build effective LLM workflows using Flowise alone.

▷ **Haystack**. Developers can choose to install Haystack using *pip* commands or relying on the deepSet studio,[19] a dedicated IDE, even though it needs a dedicated account. Besides the technical documentation that offers a good overview of basic building blocks, a set of tutorials is available on a dedicated `GitHub` repository.[20] However, it lacks examples for more complex or edge-case scenarios, which would be more useful for expert developers.

▷ **LlamaIndex**. The installation is simple and streamlined, with support for dependency management systems such as Conda and Poetry. For most use cases, a single command (i.e., `pip install llama-index`) is sufficient for enabling core functionalities, including the definition of structured workflows and agent-based architectures. The documentation offers both a conceptual overview, including use cases and practical examples, even though some resources may be outdated with the newest releases.

▷ **OpenAI SDK**. The installation process is straightforward, requiring only the installation of the SDK package via `pip`. Additional setup includes configuring environment variables used by agents and tools. In addition, the SDK offers a quick-start guide that effectively walks users through installation and basic usage. It covers how to define agents, add guardrails, and orchestrate workflows, making it easy for new users to get started. Documentation is comprehensive, covering everything from installation and basic usage to advanced capabilities. It includes example code, best practices, and implementation tips.

[19] `https://www.deepset.ai/deepset-studio`
[20] `https://github.com/deepset-ai/haystack-tutorials`

▷ **Semantic Kernel**. The platform supports three different languages, i.e., C#, Python, and Java, with instructions for the corresponding package managers. Alternatively, developers can directly pass parameters to the instantiated function responsible for setting up models. Nonetheless, the initial process can introduce some confusion due to the existence of multiple setup methodologies, which may leave the user uncertain about the most appropriate approach to take. Concerning the quickstart support, Semantic Kernel offers several Jupyter notebooks that showcase various use cases for the tool.

4.2.2 D2 - Developer Interface

Table 6 summarizes the characteristics related to the developer interface for each framework, highlighting the CLI, IDE/SDK, and web-based dashboard support for each tool. The CLI column indicates whether the framework provides a command-line interface for developers to interact with the system, while the IDE/SDK column signifies the presence of an integrated development environment or software development kit. Finally, the web-based dashboard column indicates if the framework provides a web-based interface for managing and monitoring the MAS applications.

Table 6: Developer Interface.

Framework	CLI	IDE/SDK	Web-based dashboard
AutoGPT	✔	✖	✔
AutoGen	✔	✔	✖
Dify	✖	✔	✖
Flowise	✖	✖	✔
Haystack	✔	✖	✖
LlamaIndex	✔	✖	✖
OpenAISDK	✔	✔	✖
Semantic Kernel	✔	✖	✖

▷ **AutoGPT.** Users can interact with the system through a dashboard accessed via a web application, which provides three different views, i.e., *Build*, *Library*, and *Marketplace*. In the Build section, users can create workflows using a system of visual blocks to perform specific tasks with LLM agents, while the Library view stores and displays the created workflows. Finally, users can visualize workflows in the Marketplace created by others if the cloud version is used. Otherwise, the marketplace is empty, limiting its usage while using local deployments.

▷ **AutoGen**. All the implementation is carried out by writing code using Python and users can opt for using Python scripts or Jupyter notebooks. The documentation is easy to follow and covers different aspects of the tool. Additionally, it provides simple examples to be reimplemented by the user.

▷ **Dify**. It offers a user interface, which is designed as an intuitive dashboard, enabling users to seamlessly create and manage various applications, e.g., chatbots, agents, or workflows. The dashboard also allows for the modification of the system's configuration and models' setup.

▷ **Flowise**. The platform provides a browser-based interface characterized by its drag-and-drop interaction model. Users can visually construct workflows by selecting components from

a categorized library and positioning them on a design canvas. Each component is configured to suit specific functional needs.

▷ **Haystack**. The open-source version does not include a native user interface, and it is primarily code-driven, making it ideal for developers familiar with Python. Meanwhile, the Studio version provides a drag-and-drop editor to build and manage pipelines visually. It is worth mentioning that we evaluated the open-source version of the interface, i.e., without relying on the IDE.

▷ **LlamaIndex**. It relies on Python classes, decorators, and agent-based patterns, with type-safe constructs often defined via Pydantic[21]. It does not provide a graphical or declarative interface for composing workflows, as all components must be implemented directly in code. Despite this, LlamaIndex offers a wide set of auxiliary libraries that simplify the development of LLM-based applications, including components for data ingestion, tool definition, and workflow orchestration.

▷ **OpenAI SDK**. The framework offers both integration with native Python and a dedicated SDK. The former consists of interfaces and classes for orchestrating and chaining agents, including decorators and inheritance capabilities. The latter is compatible with popular IDEs such as PyCharm and VSCode and includes rich code assistance features e.g., type hints and docstrings.

▷ **Semantic Kernel**. Similar to LlamaIndex, Semantic Kernel does not provide graphical features but the official version has comprehensive documentation to outline its functionalities and examples to be used as Python scripts. Nevertheless, the same outcome can be achieved through various functions that can be somewhat confusing for newcomers.

4.2.3 D3 - Agents and Tools Integration

Table 7 depicts an overview of the integration strategies for agents and external tools. In particular, the function calling column indicates whether the framework supports invoking third-party APIs, while the *hosted tools* column dictates the presence of pre-defined tools that can be used within the MAS application (e.g., Wikipedia, Reddit, and Web searches). The *agent-as-a-tool* column signals the availability of utilities to incorporate agents as reusable components within workflows. The last column shows the presence of a *marketplace* for sharing and discovering agents and tools.

▷ **AutoGPT.** The system provides the possibility of using different agents from different providers by specifying the corresponding API keys, if requested by the provider. However, the chosen agent cannot be changed at runtime as it is linked to the execution node. The system includes built-in blocks that provide integration with external services, such as Reddit, Medium, and Wikipedia. Additionally, there is a block called *Send Web Request* that allows users to make HTTP requests to simple external APIs. Furthermore, users can create custom blocks using Python code to integrate external services into the code. However, block creation seems to be possible only for local users, as it is necessary to add the Python script to one of the backend folders and then restart the application. Alternatively, users can submit a pull request to the AutoGPT repository to have their block included in the official version of the tool.

▷ **AutoGen**. Agents can be utilized by installing the extension specific to the used provider even though some of them are still in the experimental phase, e.g., Anthropic and Ollama. During the configuration phase, users can choose which model the agent will utilize. Afterwards, during runtime, it is possible to switch the model as long as the user has written the code for doing this at execution time. Furthermore, each agent can use external tools using two types of

[21] https://docs.pydantic.dev/latest/

Table 7: Agents and tools integration.

Platform	Function calling	Hosted tools	Agent-as-Tool	Marketplace
AutoGPT	✔	✔	✖	✖
AutoGen	✔	✔	✖	✖
Dify	✖	✔	✖	✔
Flowise	✔	✔	✖	✖
Haystack	✔	✔	✖	✖
LlamaIndex	✔	✔	✖	✔
OpenAISDK	✔	✔	✔	✖
Semantic Kernel	✔	✖	✖	✔

dedicated functions, i.e., *Tool Calling* or *Function Calling*. The first function enables the usage of pre-defined external tools or other frameworks, e.g., GraphRAG or LangChain. Instead, users have the freedom to create their own tool in Python using the *FunctionTool* class even though the call mechanism is similar compared to built-in tools.

▷ **Dify**. It supports various models within the workflow application type. To access different types of agents in the application, users must install the corresponding model plugin, available in the application marketplace. Once installed, they can be accessed within the app. Additionally, users can easily set up and switch agents in the application via the dashboard while configuring the workflow under definition.

Concerning external tools, plugins can be provided by the Dify team or the community, which also submits its own plugins to be incorporated into the marketplace after approval. Users develop custom plugins using the *Dify Plugin Daemon*, a dedicated tool that is not included in the Dify main platform. Using the Plugin Daemon requires programming knowledge and the understanding of YAML configuration files. For simple functions, the integration with external API is still possible, as shown in Listing 2, where the ROUGE score function is called in the context of the summarization task using Pydantic and FastAPI libraries.

Listing 2: Custom ROUGE service defined in Dify.

```
from fastapi import FastAPI, Body, HTTPException, Header
from pydantic import BaseModel
from rouge_score import rouge_scorer

app = FastAPI()

# Define request model
class InputData(BaseModel):
    point: str
    params: dict

@app.post("/rouge")
async def dify_receive(data: InputData = Body(...), authorization: str = Header(None)):
    """
    Receive API query data from Dify.
    """
    auth_scheme, _, api_key = authorization.partition(" ")

```

```python
    if auth_scheme.lower() != "bearer" or api_key != EXPECTED_API_KEY:
        raise HTTPException(status_code=401, detail="Unauthorized")

    point = data.point

    # Debugging logs
    print(f"Received point: {point}")
    print(f"Params: {data.params}")

    # Ping-Pong Test
    if point == "ping":
        return {"result": "pong"}

    # Call ROUGE computation inside the handler function
    if point == "app.external_data_tool.query":
        return handle_app_external_data_tool_query(params=data.params)

    raise HTTPException(status_code=400, detail="Not implemented")

def handle_app_external_data_tool_query(params: dict):
    """
    Handles external data tool query and computes ROUGE score.
    """
    # Extract expected parameters
    ground_truth = params.get("ground_truth")
    generated = params.get("generated")

    # Debugging logs
    print(f"Ground Truth: {ground_truth}")
    print(f"Generated: {generated}")

    # Validate input
    if not ground_truth or not generated:
        return {"error": "Both 'ground_truth' and 'generated' text are required"}

    # Compute ROUGE scores
    scorer = rouge_scorer.RougeScorer(["rouge1", "rouge2", "rougeL"], use_stemmer=True)
    scores = scorer.score(ground_truth, generated)

    return {
        "result": {
            "rouge1": scores["rouge1"].fmeasure,
            "rouge2": scores["rouge2"].fmeasure,
            "rougeL": scores["rougeL"].fmeasure
        }
    }
```

▷ **Flowise**. The platform supports the integration with a wide range of agent providers, offering developers flexibility in choosing and switching between different LLMs. Changing from one model to another is relatively straightforward, as it only requires replacing or reconfiguring the appropriate model component within a given flow. In addition, Flowise supports different built-in components that allow users to perform actions such as web searches, API calls, and information retrieval from external sources, while custom components can be achieved using JavaScript. This extensibility opens possibilities for developers to build tailored logic, although the framework prevents the use of Python-based tools. Such a limitation may hamper the overall flexibility, as developers are forced to build and expose such functionality through external APIs to incorporate Python logic or use specific libraries.

▷ **Haystack**. It provides a dedicated functionality, called “Generators,” to support agents’ in-

tegration, supporting both notable providers and local models. Similarly, it integrates external tools like retrievers, fetchers, and other components developed by the Haystack team or by the community. To enable tool integrations, Haystack relies on the concept of a *SuperComponent*, which allows multiple pipeline components to be grouped into a single logical unit, characterized by a unified input and output interface.

▷ **LlamaIndex**. The integration of the model into LlamaIndex is designed to be highly adaptable, with native support for both remote LLMs such as OpenAI and local deployments through Ollama, OpenLLM, or vLLM. Due to its decoupled architecture, switching between agents typically requires only minor changes to the configuration or import paths, leaving the core logic of workflows untouched. This abstraction simplifies experimentation and portability across different LLM backends.

Concerning tool integration, built-in toolsets include financial queries, web search, Wikipedia access, file readers, and structured data extraction via Pydantic schemas. These tools can be plugged into agent workflows using the `FunctionAgent` abstraction, enabling a mix of retrieval, computation, and transformation capabilities. In addition to predefined tool specs, developers can easily define custom tools as regular Python functions, which are automatically wrapped and exposed to the language model through function-calling mechanisms. A collection of official and community tools is also available via the **LlamaIndex Hub**, a centralized repository for discovering, sharing and loading tools and agents. The tools of the hub can be integrated programmatically to simplify experimentation and reuse across different projects. However, as a general problem occurring also with the other analyzed frameworks, we report some limitations i.e., only models with function-calling capabilities can be executed and some agents do not support tool invocation, which may lead to runtime errors unless properly handled.

▷ **OpenAI SDK**. While the framework supports OpenAI models natively, custom model classes must be implemented for external providers according to the required interface. This has an impact also on agents' selection, i.e., third-party providers need a corresponding class to be integrated in the workflows. The OpenAI SDK provides built-in support for several tools, including web search, file search, and a computing tool for automating tasks. Tools fall into three categories, *(i) Hosted Tools*, predefined tools hosted by OpenAI, such as web search, knowledge retrieval, and computing tools; *(ii) Function Calling Tools*, i.e., tools defined using standard Python functions with input/output schemas; or *(iii) Agent-as-a-Tool*, tools implemented as reusable agents embedded within workflows.

▷ **Semantic Kernel**. Using and selecting different agents is straightforward even though the process requires some effort for developers, as utilizing different model providers involves invoking a distinct function from the SDK. Therefore, changing models entails rewriting portions of the code. Additionally, users can incorporate multiple models into the kernel and can switch between them at runtime, as long as the code clearly specifies the desired method for doing so. Similar to other frameworks, built-in tools and custom ones can be integrated into the pipeline. For custom tools, a registration process is needed, allowing the kernel to analyze and interpret the function descriptions automatically, triggering the necessary function on its own. Nevertheless, plugins can also be invoked manually by developers. To be fully integrated, custom plugins must be *(i)* written in Python and *(ii)* equipped with annotations and detailed descriptions of their purpose and functionality.

4.2.4 D4 - Agent Creation

Table 8 summarizes agent creation aspects for each framework, highlighting the role specification, memory management, and template support. The role specification indicates how the agent's behavior is defined, while memory refers to the type of memory used by the agent, i.e., short-term (ST) or long-term (LT). Finally, the template column indicates whether the framework provides templates for creating agents.

Table 8: Agent Creation.

Framework	Role specification	Memory	Template
AutoGPT	Prompt-based	ST, LT	✖
AutoGen	Class-based	ST, LT	✖
Dify	Node-based	ST	✖
Flowise	Prompt-based	ST	✖
Haystack	Prompt-based	ST, LT	✖
LlamaIndex	Prompt-based	ST, LT	✔
OpenAISDK	Class-based	ST, LT	✖
Semantic Kernel	File-based	ST, LT	✔

▷ **AutoGPT**. Agents can be defined using any block under the *AI category* of the provided graphical palette but no predefined template is provided. Once the type of block is chosen, the user can manually define the prompt to specify the behaviors and roles. Another option is to utilize predefined AI blocks, which implements agents specialized in well-known tasks, e.g., summarization or image generation. Concering the memory, AutoGPT supports both ST and LT memory. To handle the former, the framework provides a dedicated block called *Store Value* where the user can store values to be utilized during workflow execution. Meanwhile, users rely on the Pinecone vector database[22] to handle LT memory. Nevertheless, AutoGPT allows for additional vector databases but the implementation is left to the users.

▷ **AutoGen**. The creation process involves instantiating the class associated with the desired agent. For instance, the creation of the *Extractor Agent* of the summarization example can be implemented in AutoGen as shown in Listing 3 while the corresponding prompt is shown in Listing 4. The agent is designed to extract the introduction and description of a GitHub repository from its README file. The agent uses the OpenAI GPT-4o model, which is specified during the instantiation of the model client.

Listing 3: Agent creation in AutoGen

```
    import os
from string import Template
from autogen_agentchat.agents import AssistantAgent
from autogen_agentchat.messages import StructuredMessage, TextMessage
from autogen_agentchat.ui import Console
from autogen_core import CancellationToken
from autogen_ext.models.openai import OpenAIChatCompletionClient

```

[22] https://www.pinecone.io/

```python

class ExtractorAgent():
    """Agent for extracting information based on a prompt template."""

    def __init__(self, name):
        self.name = name
        # Create an agent that uses the OpenAI GPT-4o model.
        model_client = OpenAIChatCompletionClient(
            model= "gpt-4o-mini",
            api_key= os.getenv('OPENAI_API_KEY'),
            temperature=0
        )

        self.agent = AssistantAgent(
            name=name,
            model_client=model_client,
            system_message="Use tools to solve tasks.",
        )

    def _build_prompt(self, prompt: str, readme_text: str) -> str:
        prompt = Template(prompt)
        prompt = prompt.substitute(readme_text=readme_text)
        return prompt

    async def run_agent(self, prompt: str, readme_text) -> str:
        """Creates a ChatCompletionAgent for extraction."""
        # Create instruction  prompt
        prompt = self._build_prompt(prompt, readme_text)
        # Send prompt to agent
        response = await self.agent.on_messages(
        [TextMessage(content=prompt, source="user")],
        cancellation_token=CancellationToken(),
        )
        response_text = response.chat_message.content
        print(f"[{self.name} sent]: {response_text}\n")
        return response_text
```

Listing 4: Example of prompt for the agent in AutoGen

```

EXTRACTOR_PROMPT = """
Your task is to shorten and extract only the introduction and description information from
    the README of a Github repository. You are given the following README text from a GitHub
    repository:
<README>
$readme_text
</README>

# Steps
- **Identify the structure of the repository**: The README file is a structure text file that
    might contains many sections such as introduction, description, installation,
    contributing, license,...
- **Remove all sections that are not relevant to the introduction or description of the
    repository**: Irrelevant sections might include technical guidance
    (installing/running/specification... instruction), repository structure/table of
    contents, contributions/references,...
- **Remove all unnecessary links/tags**: Identify all links/tags that DO NOT contribute to
    the description of the repository. You must remove all of these reference links and tags.
- **Return only text that is relevant to the description of the repository**: The output
    should only contains the text that is relevant to the introduction/description of the
```

```
    repository, including the project name/title, project tagline/functional
    description/purpose statement/overview. DO NOT include any output identifications such
    as: "Here's the ..." or "Extracted README:"
"""
```

The usage of ST memory is enabled by utilizing code variables and storing the agents' last conversation with built-in functions. Additionally, it is possible to save the agent's state to disk for later retrieval. Instead, LT memory can be achieved by using third-party vector databases that are accessible during code execution.

▷ **Dify**. The agent creation process is essentially done by adding the LLM or Agent node to an existing workflow, even though roles and behaviours can be customized using prompts. In the workflow, the agents' ST memory essentially consists of its output, which can be accessed by subsequent nodes. Furthermore, LT memory can be built by using miscellaneous data sources e.g., text files, Notion DB,[23] or external websites. It is important to note that the system enables the creation of prompts where variables can be defined, offering greater flexibility for the prompts.

▷ **Flowise**. The system does not provide templates for agent creation, but it allows the user to define the agent's role and behavior using prompts. However, the documentation does not offer any examples on how to implement agents, which may make it challenging for new users to understand how to create them effectively. Flowise supports the use of traditional and vector databases for implementing LT memory while there is no evidence for ST memory support, at least with dedicated features.

▷ **Haystack**. Developers can define intelligent LLM-powered agents that interact with tools and manage internal state across multiple steps by exploiting the built-in `Agent` component. Each agent is configured with a chat generator, a set of tools, and specific exit conditions to control when the agent should stop. A system prompt can be added to define the agent's role or behavior, while a state schema enables ST memory management.

▷ **LlamaIndex**. LlamaIndex provides abstractions for building LLM-based agents using two primary templates i.e., `FunctionAgent` and `ReActAgent`. The `FunctionAgent` is designed for scenarios where LLM supports function calling, i.e., invoking tools asynchronously and maintaining a history of interactions. The agent processes tool results, which are returned as messages with the role "tool," and updates its context based on the outcomes. Meanwhile, the `ReActAgent` focuses on structured reasoning through iterative steps of thought, action, and observation. In addition to these structured agents, LlamaIndex also allows defining simpler agents where an LLM with an injected prompt and optional document or memory access handles tasks like question answering or summarization. Finally, the `Context` and `Memory` components enable the handling of LT and ST memory.

▷ **OpenAI SDK**. In OpenAI SDK, creating an agent involves instantiating the `Agent` class with the chosen model and instruction prompt. The base agent class can be extended to define custom behaviors and capabilities. Agent roles and behaviors are defined using the `instruction` parameter during initialization, which acts as the system prompt guiding the agent's operation. Agents maintain implicit access to conversation history, thus supporting ST memory. The LT memory can be provided in different ways, i.e., as part agent's instruction, as part of the input during agent execution, exposing it through function tools, or using hosted retrieval or web search tools.

▷ **Semantic Kernel**. While the system does not provide predefined agents, Semantic Kernel drives the developers using YAML templates to facilitate the specification of agent instructions.

[23] `https://www.notion.com/help/intro-to-databases`

```
GroupChat: Summarizer - Evaluator - Teacher

# Summarizer: React Test-Driven Development Guide

# Evaluator: The ROUGE-L score for the recent summary "React Test-Driven Development Guide" com
pared to the ground truth is approximately 0.14. Since this score is higher than the previous b
est score of 0.0, the prompt has been updated to the most recent one provided by the Teacher.

# Teacher: Summarize the following extracted text from a Github repository README into a short
term/phrase introducing the repository: If the text contains an existing tagline, functional de
scription, purpose statement, or overview, prioritize using it. The output should include only
a short term/phrase introducing the repository.

# Summarizer: A series of examples on how to TDD React

# Evaluator: The recent summary "A series of examples on how to TDD React" achieved a perfect R
OUGE-L score of 1.0 when compared to the ground truth. This score is higher than the previous b
est score of approximately 0.14, so the prompt has been updated to the most recent one provided
 by the Teacher.
```

Figure 9: Semantic Kernel output of the summarization scenario.

Alternatively, the user can write the prompts manually where input variables and function calls can also be utilized. However, there is no distinction between these; the user can include whatever they wish in the instructions. To enable the LT memory feature, Semantic Kernel provides an internal vector database. Additionally, the developer can leverage chat history, where prompts and responses can serve as ST memory. An explanatory output produced by the Semantic Kernel implementation of the summarization scenario is shown in Figure 9.

4.2.5 D5 - Agent Orchestration

Table 9 summarizes the agent orchestration capabilities in terms of execution, message passing, and handoff mechanisms.

The *Execution flow* column indicates whether the framework supports *sequential* or *hierarchical* execution of agents. Sequential orchestration means agents act one after another in a linear pipeline. Hierarchical orchestration introduces an explicit control structure: some agents can act as supervisors or coordinators, delegating subtasks to other agents and integrating their results. Dify, Flowise, LlamaIndex, OpenAISDK, and Semantic Kernel all support sequential and hierarchical execution. This suggests these frameworks allow not just chained execution, but also a task decomposition and coordination layer. AutoGPT, AutoGen, and Haystack only support sequential execution, indicating a more linear flow of control without built-in mechanisms for agent supervision or delegation.

Finally, the *Message passing* column indicates whether it supports message passing between agents. We characterized the message passing mechanism as *Stateful* if the information is preserved or *Stateless* otherwise. In this respect, a manual *handoff* means that developers have to write code to handle the message passing. On the contrary, some frameworks also support the automated handoff, i.e., agents can communicate autonomously.

▷ **AutoGPT**. The orchestration workflow is represented through a visual, block-based dashboard, enabling users to add, connect, and configure blocks to create flows leveraging the low-code paradigm. Nevertheless, users seeking more specialized functionality may eventually need to write Python code to create custom blocks. The AutoGPT platform incorporates various logic and data blocks intended to replicate the functionality of code-based workflows.

Table 9: Agent Orchestration.

Framework	Execution flow	Message passing	Handoff
AutoGPT	Sequential	Stateless	Automated
AutoGen	Sequential	Stateless	Automated
Dify	Sequential, Hierarchical	Stateless	Automated
Flowise	Sequential, Hierarchical	Stateful	Automated
Haystack	Sequential	Stateless	Manual
LlamaIndex	Sequential, Hierarchical	Stateful	Manual, Automated
OpenAISDK	Sequential, Hierarchical	Stateful	Manual, Automated
Semantic Kernel	Sequential, Hierarchical	Stateful	Manual, Automated

However, users may encounter limitations in the level of control provided by these blocks, particularly when handling tasks that involve iterative processes such as loops, which are essential for optimization. For instance, in the `GitHub` README summarization task, it was necessary to execute a loop 15 times. The system did not offer a straightforward method to specify the number of iterations directly. Instead, it required the use of a block to increment the iteration count during each cycle, resulting in an increased number of blocks in the workflow. As a result, while the platform provides a broad range of blocks for representing logical flows, reliance on these blocks to express complex logic flows can introduce usability challenges. These challenges may undermine the usability in terms of orchestration patterns. Similarly, the communication mechanisms is implemented by connecting blocks and both sequential and parallel execution are allowed.

▷ **AutoGen**. The orchestration is mainly done programmatically, i.e., developers have to manually specify all aspects of execution, including the order of loops and agents. However, a multi-agent team template can be used to facilitate the task as it provides high-level definitions of the workflow, requiring the developer to only specify the involved agents, the turn style of the agents, and the stop criteria. With this configuration in place, the framework manages the orchestration. Two paradigms, i.e., *high-code* or *low-code*, can be reused for setting up the communication mechanism. On the one hand, the multi-agent team approach automates the delivery of messages among agents by handling both text and multimodal messages. On the other hand, developers can implement the communication logic themselves, allowing for greater control over the whole process.

▷ **Dify**. In this framework, workflows must be created to orchestrate the agents as shown in Figure 10. Workflows are defined by a visual block-based dashboard where users can add nodes. Each node is responsible for performing a small task to achieve a larger goal, including the run of code snippets written in Python or NodeJS. Nevertheless, the usage of a visual block-based dashboard limits the expressiveness of the flow as only nodes available in the system can be used. For example, implementing a classical LOOP is not possible in the flow since this feature was recently added and does not retain the memory of variables from previous iterations. Such a paradigm is also used to manage the communication between agents using a dedicated types of nodes. For instance, the START node is responsible for starting the process for sequential workflows. Similarly, branching execution and later joining the output are supported by using the ITERATION node, which acts as a large box that contains a sequence

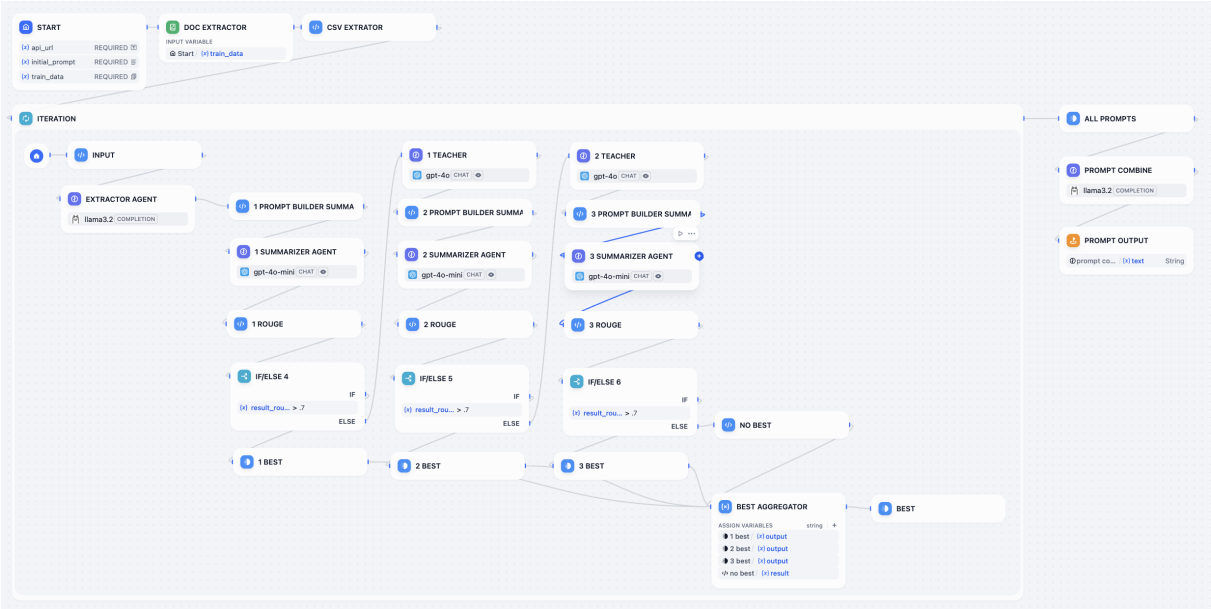
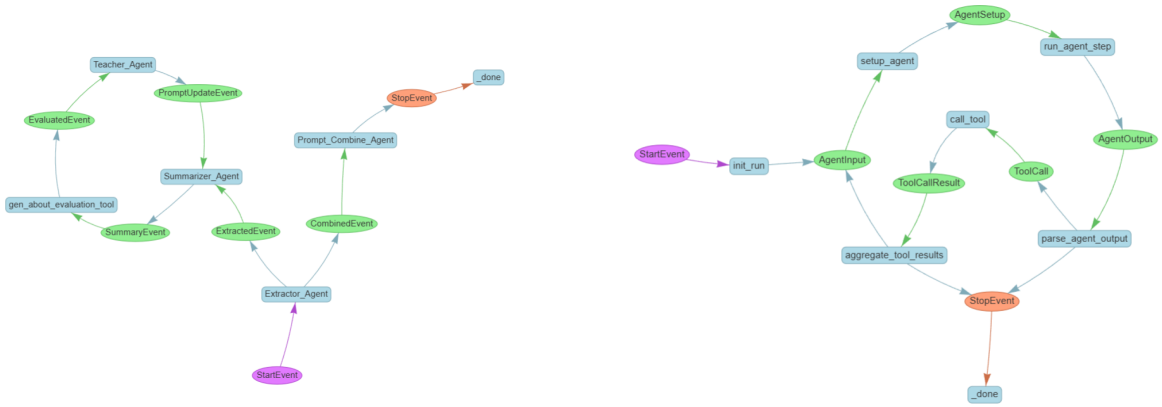


Figure 10: A Dify workflow.

of nodes that can perform the same tasks in parallel.

▷ **Flowise**. The expressiveness of control flow in Flowise is achieved through the connection of agent outputs to the sequential inputs of other agents. This output-input linkage defines the logic chain and facilitates the development of dynamic workflows. Components such as *condition nodes* and *loop handlers* allow developers to simulate decision-making and repetition, which are crucial for adapting agent behavior to real-time data or user input. These elements enable the platform to mimic the flow control available in programming languages like Python or JavaScript, making it easier to implement logic such as branching paths or iterative tasks within the visual editor. In addition, agents can leverage the same mechanism for implementing the message passing, ensuring that agents have access to the latest context or generated content when making decisions or generating responses. Additionally, Flowise supports a more flexible communication model through shared state variables, which can be read and written by each agent. It also makes it possible to propagate changes or data across various parts of the workflow without requiring explicit connections between every agent.

▷ **Haystack**. While agent orchestration is not natively provided in Haystack as a high-level abstraction, Haystack relies on a pipeline-based architecture where each step may be treated as a stand-alone component with defined inputs and outputs. This modular system allows developers to build complex workflows by chaining components, including multiple agents, in sequences or parallel branches. Each agent operates with its own internal state and toolset, and decisions about control flow must be implemented manually within the pipeline. Developers can implement conditional branching, loop constructs, and step-by-step decision-making manually. Message passing between agents is based on the *ChatMessage* object, which keeps track of the conversation context and can be reused or modified across steps. However, Haystack does not provide a built-in way to handle direct inter-agent communication—this must be managed externally through custom logic or wrappers. A notable feature that partially addresses orchestration is the *SuperComponent* class. This allows developers to encapsulate an entire pipeline (including multiple agents and components) into a single abstract block with one input and one output. While this is useful for managing complexity and reusability, it does not solve the challenge of communication between multiple agents, which still requires ad hoc mechanisms.



(a) Workflow pattern. (b) AgentWorkflow pattern.

Figure 11: Comparison of orchestration patterns in LlamaIndex.

▷ **LlamaIndex**. There are two distinct paradigms for agent orchestration: *Workflow* and *AgentWorkflow*. The *Workflow* module offers a strictly event-driven design, where the control logic is encoded in a sequence of explicitly defined `@step` functions. Figure 11 illustrates the two orchestration patterns supported in LlamaIndex. The adopted notation in the diagrams distinguishes between different elements of the orchestration: rectangular blue nodes denote agents, which are active components responsible for executing tasks; green ovals denote events, i.e., data units or signals exchanged between agents; purple ovals indicate start events, which trigger the execution of a workflow; and orange ovals represent stop events, marking the termination of the orchestration. Directed edges indicate the control and data flow between agents and events. The Workflow pattern (Figure 11a) enables the definition of domain-specific pipelines in which multiple specialized agents interact by producing and consuming intermediate events. For example, events such as *SummaryEvent*, *CombinedEvent*, or *EvaluatedEvent* trigger the execution of corresponding agents (e.g., *Summarizer*, *Extractor*, and *Teacher*), thus allowing a flexible composition of tasks tailored to the target application. The AgentWorkflow pattern shown in Figure 11b provides a more generic orchestration loop for agent-based reasoning and tool usage. Here, the flow is standardized, starting from an initialization step (*StartEvent*), followed by agent setup, input handling, tool invocation, and output parsing, before reaching the termination state (*StopEvent*). While the *Workflow* pattern offers greater expressiveness for domain-driven compositions, the *AgentWorkflow* pattern promotes reuse and modularity by encapsulating common interaction cycles between agents and tools. *AgentWorkflow* may be more prone to failures or unexpected behavior if the model misinterprets prompts or generates incorrect tool calls. Those patterns integrate also the message passing mechanisms with automated handoff.

▷ **OpenAI SDK**. Two orchestration techniques are supported, i.e., *LLM-Orchestrated Workflows* and the *Code-Orchestrated Workflows*. The former involves a “triage agent” planning the workflow by invoking tools and delegating tasks to sub-agents. In the latter, users explicitly define workflow logic using Python code for more deterministic control. Control flow expressiveness depends on the orchestration method. In LLM-orchestrated workflows, the triage agent uses “handoffs” to delegate tasks. Handoffs can refer to agent names or functions, with optional prompt prefixes for improved clarity. In code-based orchestration, developers can use Python control structures (e.g., `if`, `for`, `while`) to control execution flow. Communication between agents is handled through handoffs in LLM-based orchestration or explicitly via function argu-

ments and return values in code-based orchestration.

▷ **Semantic Kernel**. There are two different ways of handling the workflow between agents, i.e., managing the agents manually or using the *Agent Framework* capability. Concerning the former, the user must write all interactions between the agents. As a result, the entire orchestration relies on the user writing the code for it. Meanwhile, the *Agent Framework* can support *(i)* the writing of the orchestration code manually, similar to how it would be done without a framework or *(ii)* creating a group chat namely *AgentGroupChat*, that is based on top of a sequence of interactions, known as the selection function, as well as how these interactions between agents will be concluded, referred to as the termination function. Both functions can be articulated in natural language, and a model will manage their execution. One can write anything in the orchestration that the chosen programming language permits. However, given the option to make plugin/function calls and to write the selection and termination functions in natural language, there is a heavy reliance on the expressiveness of natural language, which can be ambiguous, as well as on the ability of the model to interpret it. Concerning the communication mechanism, developers can orchestrate agent communication through explicit coding, directly managing message passing, or leveraging the *Agent Framework*'s *AgentGroupChat* for automated interaction based on a defined selection function.

4.2.6 D6 - Monitoring

Table 10 summarizes the monitoring capabilities of each framework, focusing on the availability of real-time logs, debugging tools, and telemetry features. Real-time logs indicate whether the framework provides live feedback during execution, while debugging tools refer to the ability to inspect and troubleshoot workflows. Telemetry features encompass the collection of metrics such as token usage, cost, and latency.

Table 10: Monitoring.

Framework	Real-time Logs	Debug	Telemetry
AutoGPT	✔	✖	✖
AutoGen	✔	✔	✖
Dify	✔	✔	✔
Flowise	✖	✔	✖
Haystack	✔	✔	✔
LlamaIndex	✔	✔	✔
OpenAISDK	✔	✔	✖
Semantic Kernel	✔	✔	✔

▷ **AutoGPT**. After the execution, the user can verify the input and output of each block and workflow. In the case of the LLM agent, the used prompt is also displayed as input. No debugging tool was found and only workflow information is available, describing the duration, cost, and the starting time.

▷ **AutoGen**. It is possible to store the sequence of messages exchanged between agents in a variable or use prints to observe the conversation between agents. Moreover, the platform enables developers to debug the execution of the agents by utilizing logging, print statements

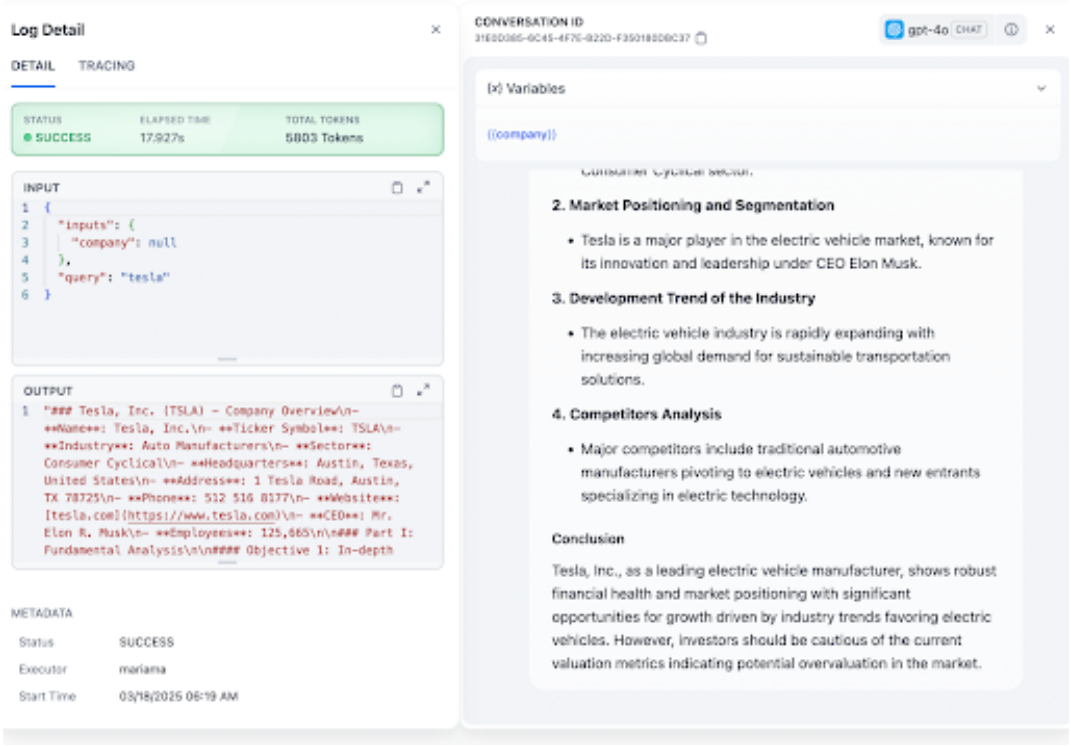


Figure 12: Telemetry features supported by Dify.

in the code, and the debugging tools offered in the developer's IDE. The framework monitors token usage and the total duration of the conversation and errors can be caught using Python code.

▷ **Dify**. After executing a workflow in Dify, it is possible to check the logs of each executed node, which describe the input and output of each node, as well as metrics such as token consumption, cost, and latency in the case of nodes that deal with models. Analyzing these logs facilitates the identification of potential errors related to parameter passing as well as runtime errors. Debugging the node can be accomplished by using the functionality to run the node individually, which may act as a debugging step. The user can also analyze all execution run history, which displays previous execution logs. However, unlike code debugging, it is not possible to execute the entire flow step by step while monitoring the variables and their changes. After the execution of each node, it is possible to collect useful telemetry information, e.g., token usage, model pricing, cost, and latency as shown in Figure 12.

▷ **Flowise**. It provides a basic level of logging through its test chatbot interface, which displays some information about the output generated by each agent during a workflow run. While this feature offers a quick way to preview the behavior of the system, it is not sufficient for in-depth analysis or diagnostics. The limited information makes it difficult to understand the internal state of the system or trace the full execution path of a complex agent workflow. To address this limitation, Flowise supports integration with external observability tools e.g., LangSmith[24] or Arize.[25] The primary debugging tool within Flowise is the same test chatbot used for logging. Developers can use it to run the workflow and observe outputs in real time. However, when an error occurs, the information provided is often vague and lacks specificity. Errors are generally displayed without identifying the exact component or node responsible for the failure, which

[24] https://www.langchain.com/langsmith
[25] https://arize.com/

significantly complicates troubleshooting efforts. In many cases, developers are forced to manually inspect individual components or connect to external platforms to obtain more detailed debugging information. Additionally, Flowise does not offer a native feature for examining the precise inputs and outputs of each agent within the interface, making it harder to isolate faulty logic or misconfigured components during development.

▷ **Haystack**. Versatile tools are provided for monitoring and debugging pipelines, both during development and in production. Basic logging relies on Python's standard library, but can be extended with structured formats and advanced rendering for more complex environments. During prototyping, real-time logging can be enabled to trace each pipeline step, highlighting the inputs and outputs of individual components. For more advanced use cases, Haystack also supports full execution tracing through external backends such as OpenTelemetry or Datadog. This provides deep visibility into data flow and pipeline behavior, which is particularly useful in production environments where performance and bottlenecks must be monitored. In addition to runtime monitoring, Haystack includes built-in capabilities for pipeline evaluation. For instance, the LLMEvaluator component uses language models to assess pipeline outputs based on user-defined instructions. It supports few-shot examples and allows developers to define custom input/output schemas and scoring criteria.

▷ **LlamaIndex**. There are basic supports for monitoring agent workflows through logging, event streaming, and internal instrumentation. Logs and execution traces are primarily exposed via the *Context* object, which maintains a streaming queue of events such as tool invocations, agent inputs/outputs, and intermediate reasoning steps. Developers can consume this stream using the stream event method to observe runtime behavior step by step. For more advanced use cases, LlamaIndex integrates with external telemetry, enabling model evaluation, trace visualization, and performance monitoring. While there is no centralized dashboard, these integrations allow developers to capture fine-grained runtime information, facilitating debugging, profiling, and system analysis in both development and production environments.

▷ **OpenAI SDK**. Tracing is built in and enabled by default, leveraging the OpenAI dashboard that groups them using workflow names or trace IDs. It can be disabled via the *OPENAI_AGENTS_DISABLE_TRACING* environment variable or programmatically via `RunConfig.tracing.disable = True`. Sensitive agents/tools can also be excluded from tracing. Debugging follows standard Python debugging practices, leveraging IDE features, print statements, or logging modules. The SDK does not offer built-in telemetry, and users may implement custom telemetry by integrating third-party libraries or custom logging solutions.

▷ **Semantic Kernel**. It is possible to include prints in the execution of the code to visualize the agents' responses. Furthermore, developers can include logging functions to check all the kernel invocations. In addition, the framework allows users to debug the agent's execution by using logging in the code, as well as debugging tools available in the developer's IDE.

4.2.7 Summary of the qualitative analysis

The analysed frameworks exhibit a range of capabilities in terms of agent creation, orchestration, and monitoring at different levels of abstraction. As expected, low-code MAS frameworks offer more built-in functions, even though limitations in terms of agent selection and sharing represent a drawback. In contrast, high-code frameworks provide more flexibility and control over agent behavior, but require more development effort. The choice of framework depends on the specific requirements of the application, the level of expertise of the developers, and the desired balance between ease of use and customization. It is worth noting that monitoring

capabilities often requires external telemetry tools or services, which can add complexity to the development process.

4.3 Quantitative Analysis

Based on the results of the qualitative evaluation, we select four frameworks, i.e., AutoGen, AutoGPT, Dify, and Semantic Kernel, which give an adequate coverage of supported features, programming paradigms, and popularity considering the GitHub platform. For Semantic Kernel, we also assess the *Chat* version, where the agents exploit a chat-based interaction instead of traditional mechanisms. The four frameworks have been compared with METAGENTE [47], the explanatory LLM-based Multi-Agent System developed using LangChain and presented in Section 4.1. It is worth mentioning that our analysis does not aim to find the best framework to implement the summarization task. Instead, we compare the frameworks to extensively discuss their ability to support the design, development, and evaluation of a multi-agent system.

We evaluated the frameworks with respect to two main performance traits, i.e., *Effectiveness* and *Efficiency* by considering two README.MD datasets described in Section 4.3.1. For the former, we measure the ability to provide meaningful summarization, being relevant to the ground-truth data, using popular metrics including the ROUGE-1, ROUGE-2, and ROUGE-L scores. Meanwhile, with the latter, we evaluate whether the frameworks are efficient with respect to the following metrics: *Token usage*, *Number of requests*, *Usage time*, and *Usage by model type*. Details on the used metrics are given in Section 4.3.2. Hardware and software configurations that have been used to conduct the experiments are given in Section 4.3.3.

The experimental results in terms of effectiveness and efficiency of the analyzed MAS frameworks are discussed in Section 4.3.4.

4.3.1 Datasets

We made use of the datasets curated from our previous work [47]. The datasets were initially adopted from an existing dataset for README.MD-related tasks [46], and extended by incorporating a diverse range of data sources to enhance its comprehensiveness and applicability. First, the initial dataset was augmented with `GitHub` repositories categorized under the awesome-lists and documentation-related topics[26] that align with the document repositories category [50]. Then, it was enriched with curated repositories containing popular Python projects [51], and Jupyter notebooks for data analysis [52, 53]. This is motivated by the fact that popular repositories often feature well-maintained and detailed README.MD files [54]. Eventually, a set of 6,933 unique repositories containing at least a README.MD file was curated. Through a manual inspection, we noticed that in several repositories, the About descriptions do not match what was written in the README.MD files. This happens because developers changed the README.MD files, but then forgot to update the corresponding About. Thus, we filtered out those repositories and eventually obtained 925 samples, from which 2 *training sets* were randomly selected with 10 and 50 samples to yield TS_{10} and TS_{50}, and a *testing set* of 865 samples named as *ES*.

Table 11 provides descriptive statistics for the resulting dataset. It includes the average length and standard deviation (St. Dev.) for both the About description and README.MD across *ES*, TS_{10}, and TS_{50}. These statistics highlight the variability and characteristics of the dataset used in the study.

[26] `https://github.com/topics`

Table 11: Datasets statistics.

Dateset	# Samples	Description		README.MD	
		Avg. length	St. Dev.	Avg. length	St. Dev.
ES	865	76.09	53.81	6,591.44	8,976.48
TS_{10}	10	85.70	50.43	4,021.50	5,334.84
TS_{50}	50	61.76	31.29	5,111.20	6,094.71

4.3.2 Metrics

Based on the METAGENTE scenario, we compute the ROUGE score [49] to evaluate the quality of the generated summary. In particular, we compute the ROUGE-1, ROUGE-2, and ROUGE-L scores, thus replicating the original experiment. While we acknowledge that more advanced metrics have been proposed in the literature, we consider ROUGE to avoid a biased comparison with the original experiment.

$$\text{ROUGE-N} = \frac{\sum_{S \in \{\text{Reference Summaries}\}} \sum_{\text{gram}_n \in S} \min\left(\text{Count}_{\text{gen}}(\text{gram}_n), \text{Count}_{\text{ref}}(\text{gram}_n)\right)}{\sum_{S \in \{\text{Reference Summaries}\}} \sum_{\text{gram}_n \in S} \text{Count}_{\text{ref}}(\text{gram}_n)} \tag{1}$$

where gram_n denotes an n-gram, $\text{Count}_{\text{gen}}$ is the count in the generated summary, and $\text{Count}_{\text{ref}}$ is the count in the reference summary.

The ROUGE-L score is based on the Longest Common Subsequence (LCS):

$$\text{ROUGE-L} = \frac{\text{LCS}(X, Y)}{\text{Length}(Y)} \tag{2}$$

where $\text{LCS}(X, Y)$ is the length of the longest common subsequence between the generated summary X and the reference summary Y.

ROUGE scores were used to evaluate the effectiveness of different frameworks in optimizing prompts within a multi-agent setting. While basic statistics (mean and standard deviation) offer a preliminary view, the similarity of scores necessitated a statistical significance test to identify meaningful differences between frameworks.

To assess the statistical significance of the results, we first compute the Shapiro-Wilk test [55] to check for normality of the distribution. Since none of the distributions were normal, we perform a Wilcoxon signed-rank test [56]. The null hypothesis states that there is no significant difference between the two samples, while the alternative hypothesis states that there is a significant difference. A p-value less than 0.05 indicates that we can reject the null hypothesis and conclude that there is a significant difference between the two samples. In our case, we use the Wilcoxon signed-rank test to compare the performance of each MAS framework with the METAGENTE framework.

4.3.3 Hardware and Software Configurations

All experiments were conducted under the same hardware environment and operating system, the specifications of which are provided in Table 12. In contrast, software configurations varied slightly depending on the framework in use. Framework-specific software details, including versions and dependencies, are summarized in Table 13.

Table 12: Hardware configuration.

Component	Specification
Processor	Apple M2 (8-core CPU)
RAM Memory	16 GB
Operating System	macOS 15.5 (24F74)

Table 13: Environment configurations.

Framework	Python	Version	Docker
AutoGen	Python 3.12.2	0.5.4	No
AutoGPT	Python 3.12.2	0.6.4 (beta)	Yes (only databases)
Dify	–	1.0.1	Yes
Semantic Kernel	Python 3.12.2	1.29.0	No

To ensure fair comparisons among the evaluated frameworks, we standardized the evaluation metrics and configurations across all implementations. This included using the same LLM model versions, temperature settings, and optimization parameters. As shown in Table 14 the temperature parameter was set to 0.7 for the `Extractor Agent` and `Summarizer Agent` agents, which are responsible for generating the initial and optimized prompts, respectively. The `Teacher Agent`, which is responsible for generating the final summary, was set to 0.0 to ensure deterministic output. The `Prompt Creator Agent`, which is responsible for creating the prompt for the `Teacher Agent`, was set to 0.2 to allow some variability in the generated prompt. The `Extractor Agent` was set to 0.0 to ensure deterministic evaluation of the generated summaries. As for optimization pipeline-level variables, all frameworks were configured with a maximum of 15 iterations per optimization loop and a ROUGE-L threshold of 0.7. These values were adopted from the reference study [47], which is the basis for this optimization task.

4.3.4 Results of the quantitative analysis

This section reports the results obtained by comparing the selected MAS framework in terms of quantitative metrics. In particular, we analyze i) the effectiveness of the generated summaries in terms of ROUGE scores and ii) the efficiency of the whole process in terms of consumed tokens and number of requests.

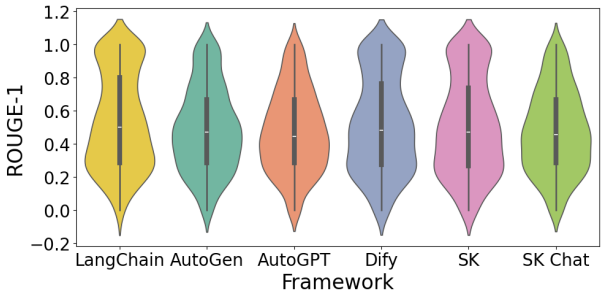
Effectiveness

For the first series of experiments, we used the TS_{10} dataset for fine-tuning METAGENTE on the four frameworks, and the results are shown in the boxplots in Figure 13. The violin plots for both frameworks display a dense concentration of scores between 0.2 and 0.7. There are also thicker tails near 1.0, especially for ROUGE-1. This indicates that the optimized prompts were able to generate not only average summaries but also summaries that are identical or almost identical to the ground truth.

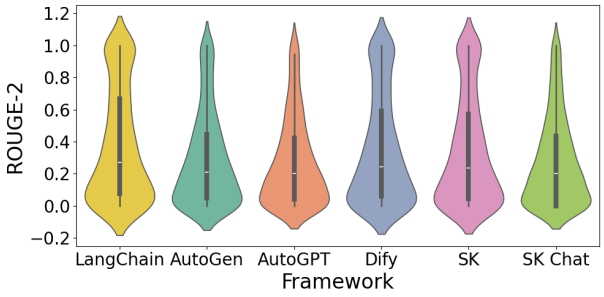
Table 14: Configuration of agents used in the optimization and evaluation pipelines.

Agent	Model Version	Temperature
`Extractor Agent`	gpt-4o-mini (2024-07-18)	0.7
`Summarizer Agent`	gpt-4o-mini (2024-07-18)	0.7
`Teacher Agent`	gpt-4o (2024-08-06)	0.0
`Prompt Creator Agent`	gpt-4o (2024-08-06)	0.2
Evaluator*	gpt-4o-mini (2024-07-18)	0.0

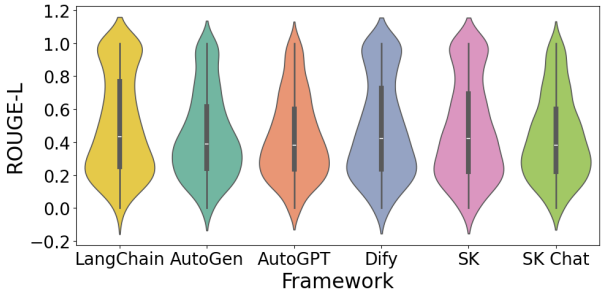
** The Evaluator agent was only used in the Semantic Kernel Chat implementation to facilitate autonomous orchestration during optimization.*



(a) ROUGE-1



(b) ROUGE-2



(c) ROUGE-L

Figure 13: ROUGE scores for prompt optimized with TS_{10}.

To aim for a clearer view of the results, we computed and depicted in Table 15 the mean and standard deviation of the scores represented in Figure 13. As can be seen in the table, Dify achieves the highest average ROUGE scores (ROUGE-1: 0.515, ROUGE-2: 0.352, ROUGE-L:

Table 15: Mean and standard deviation of ROUGE scores using TS_{10}.

Framework	ROUGE-1	ROUGE-2	ROUGE-L
AutoGen	0.488 (0.257)	0.294 (0.292)	0.441 (0.264)
AutoGPT	0.473 (0.247)	0.275 (0.274)	0.428 (0.253)
Dify	**0.515** (0.291)	0.352 (0.339)	0.479 (0.301)
Semantic Kernel	0.505 (0.293)	0.344 (0.336)	0.472 (0.300)
Semantic Kernel Chat	0.472 (0.251)	0.278 (0.276)	0.427 (0.256)

Table 16: Mean and standard deviation of ROUGE scores using TS_{50}.

Tool	ROUGE-1	ROUGE-2	ROUGE-L
AutoGen	0.486 (0.256)	0.293 (0.291)	0.441 (0.263)
AutoGPT	0.462 (0.239)	0.254 (0.261)	0.408 (0.243)
Dify	0.503 (0.270)	0.320 (0.314)	0.459 (0.279)
Semantic Kernel	0.487 (0.264)	0.300 (0.301)	0.444 (0.272)
Semantic Kernel Chat	0.483 (0.253)	0.285 (0.279)	0.435 (0.257)

0.479), followed by Semantic Kernel (ROUGE-1: 0.505, ROUGE-2: 0.344, ROUGE-L: 0.472).

The third-best results are obtained by AutoGen (ROUGE-1: 0.488, ROUGE-2: 0.294, ROUGE-L: 0.441), with mean scores being approximately 5.24%, 16.48%, 7.93% respectively for each metric lower than Dify's. Pairwise analysis show that AutoGen's results are not comparable to those of any framework. The violin plots demonstrate that AutoGen clusters most samples in the lower range, aligning more closely with the bottom performers.

Among others, Semantic Kernel Chat (ROUGE-1: 0.472, ROUGE-2: 0.278, ROUGE-L: 0.427) and AutoGPT (ROUGE-1: 0.473, ROUGE-2: 0.275, ROUGE-L: 0.428) are the two worst performing frameworks. According to the pairwise test, they share similar distributions across all ROUGE scores. Compared to Dify, Semantic Kernel Chat's scores were approximately 8.35%, 21.02%, and 10.85% lower, while AutoGPT's were 8.16%, 21.88%, and 10.65% lower across ROUGE-1, ROUGE-2, and ROUGE-L respectively.

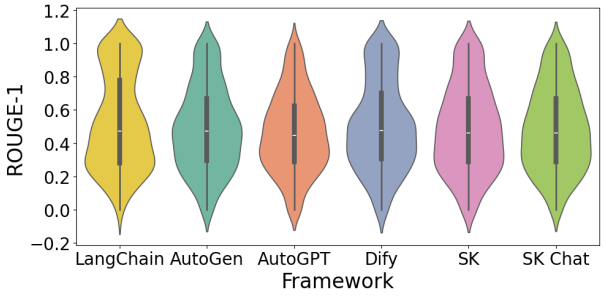
Figure 14 displays ROUGE score distributions, and Table 16 reports the corresponding means, standard deviations for the results obtained with the TS_{50} dataset.

With the exception of Semantic Kernel Chat, all frameworks perform worse when using the TS_{50} dataset for prompt optimization. Among other, Dify once again achieves the highest scores (ROUGE-1: 0.503, ROUGE-2: 0.320, ROUGE-L: 0.459). By comparing Figures 13 and 14, we see that Dify's shape on the top of TS_{50} resembles TS_{10}. This demonstrates that Dify still manages to generate results close to the ground truth, a feat not accomplished by the other frameworks.

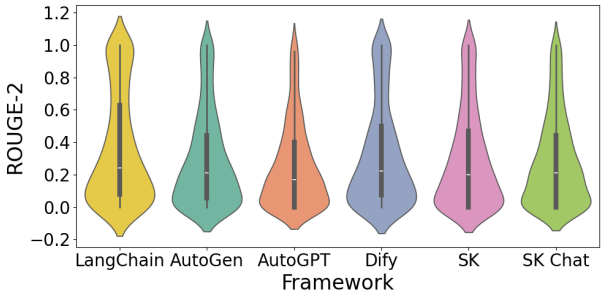
Semantic Kernel, AutoGen, and Semantic Kernel Chat earn comparable results according to the Wilcoxon test, and their mean scores were percentually lower than Dify's by approximately 3.18%, 6.25%, and 3.27% for Semantic Kernel; 3.38%, 8.44%, and 3.92% for AutoGen; and 3.98%, 10.94%, and 5.23% for Semantic Kernel Chat across ROUGE-1, ROUGE-2, and ROUGE-L, respectively. Finally, AutoGPT ranks lowest once again, with mean scores approximately 8.14%, 20.63%, and 11.10% lower than Dify's across the same metrics.

To further analyze the obtained results, we compute the pairwise Wilcoxon test depicted in Tables 17–19. In particular, the ✔ sign indicates statistical significance ($p < 0.05$), while ✖ means the opposite. The p-values indicate significant differences between most framework pairs, meaning that some frameworks are better for implementing the `GitHub` README summarization task. In particular, Dify achieves the best ROUGE scores compared to the other frameworks, apart from Semantic Kernel. Notably, the chat version of Semantic Kernel performs worse than the original version, which is likely due to the fact that it uses a chat-based model for orchestration and function calling, leading to less effective agent communication.

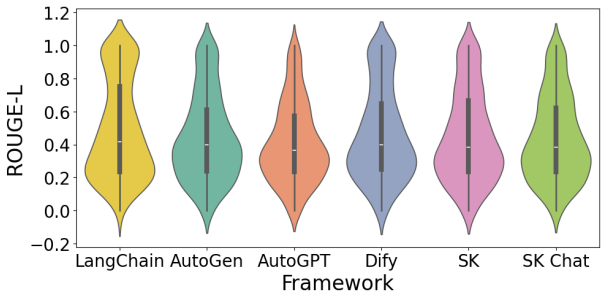
While the Wilcoxon signed-rank test highlights statistically significant differences across frameworks, two important aspects deserve further discussion. First of all, the non-determinism of LLM-based agents (due to sampling temperature, random seeds, or stochastic decoding strategies) can introduce variability in results. To mitigate this, our evaluation relied on multiple runs per configuration. Nonetheless, we acknowledge that residual randomness may still contribute to observed performance fluctuations, and future evaluations could incorporate larger sampling or deterministic decoding to further control for this factor. Additionally, beyond statistical significance, it is important to consider *effect sizes*. Although our tests often yielded p-values below 0.05, the magnitude of the observed differences is sometimes limited, as also



(a) ROUGE-1

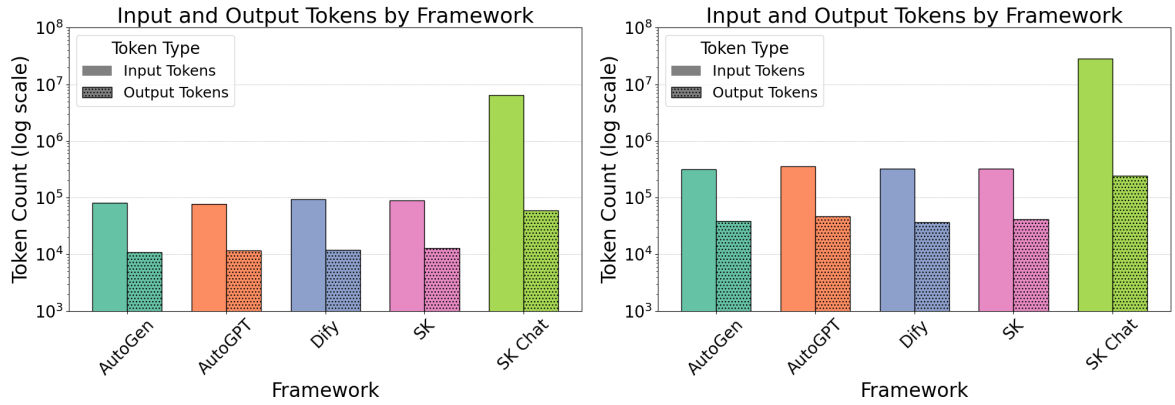


(b) ROUGE-2



(c) ROUGE-L

Figure 14: ROUGE scores for prompts optimized with TS_{50}.



(a) TS_{10} (b) TS_{50}

Figure 15: Token usage on the optimization pipeline.

suggested by the relatively small deltas in ROUGE scores between frameworks. This indicates that while frameworks differ in measurable ways, the practical impact of these differences may be modest in real-world applications. To address this, future work should complement significance testing with standardized effect size measures providing a more comprehensive view of both statistical and practical significance.

Efficiency

For this performance trait, the following metrics are considered: *Token usage*, *Number of Requests*, and *Usage time*.

Before the used model processes an input, the prompt is first segmented into tokens, which represent units such as words or subwords. The cost of using the model API depends on both the number of tokens in the input and output, as well as on the specific model employed. Token usage plays a crucial role in the context of fine-tuning LLMs, as it directly influences

Table 17: Wilcoxon signed-rank test results for ROUGE-1 across framework pairs for TS_{10}.

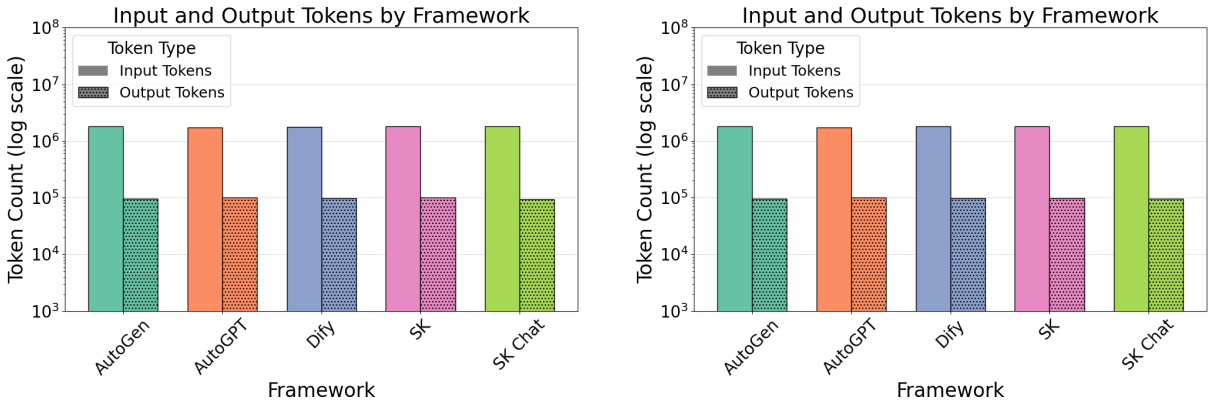
Framework 1	Framework 2	p-value	Significant
AutoGen	AutoGPT	<0.001	✔
AutoGen	Dify	<0.001	✔
AutoGen	Semantic Kernel	0.014	✔
AutoGen	Semantic Kernel Chat	<0.001	✔
AutoGPT	Dify	<0.001	✔
AutoGPT	Semantic Kernel	<0.001	✔
AutoGPT	Semantic Kernel Chat	0.995	✖
Dify	Semantic Kernel	0.053	✖
Dify	Semantic Kernel Chat	<0.001	✔
Semantic Kernel	Semantic Kernel Chat	<0.001	✔

Table 18: Wilcoxon signed-rank test results for ROUGE-2 across framework pairs for TS_{10}.

Framework 1	Framework 2	p-value	Significant
AutoGen	AutoGPT	<0.001	✔
AutoGen	Dify	<0.001	✔
AutoGen	Semantic Kernel	<0.001	✔
AutoGen	Semantic Kernel Chat	0.106	✖
AutoGPT	Dify	<0.001	✔
AutoGPT	Semantic Kernel	<0.001	✔
AutoGPT	Semantic Kernel Chat	0.045	✔
Dify	Semantic Kernel	0.488	✖
Dify	Semantic Kernel Chat	<0.001	✔
Semantic Kernel	Semantic Kernel Chat	<0.001	✔

computational cost, efficiency, and the overall feasibility of the fine-tuning process. We counted the number of tokens used as a metric to compare the frameworks.

Figure 15a and 15b show the token usage during the optimization pipeline using TS_{10} and TS_{50}, respectively. As expected, token usage is lower for the smaller sample size and increases with the number of samples in the optimization pipeline. Nevertheless, it remains in a similar range across all frameworks during the evaluation pipeline (Figure 16), regardless of the number of samples used in optimization.



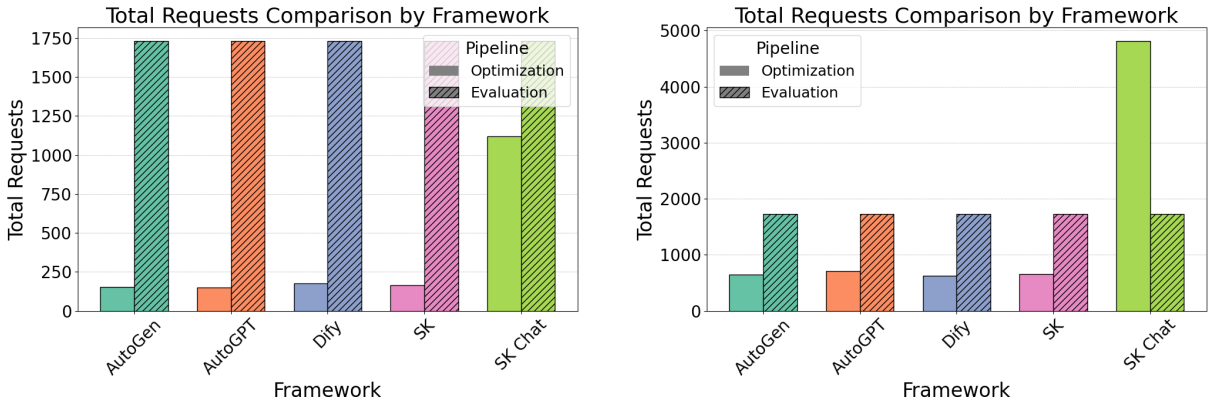
(a) TS_{10} (b) TS_{50}

Figure 16: Token usage on the evaluation pipeline.

Semantic Kernel Chat exhibits substantially a higher token usage compared to others–approximately 6 and 28 million tokens for TS_{10} and TS_{50}, respectively. In contrast, the total token usage for the other frameworks remains much lower, averaging around 100,000 tokens for TS_{10} and 400,000 tokens for TS_{50}. This elevated token usage in Semantic Kernel Chat is likely due to its internal agent orchestration mechanism. Specifically, during orchestration, a part of the chat history must be sent to the model to enable agents to generate replies. Additionally, operations such as agent selection and function calling rely heavily on models, which further increases token consumption.

Table 19: Wilcoxon signed-rank test results for ROUGE-L across framework pairs for TS_{10}.

Framework 1	Framework 2	p-value	Significant
AutoGen	AutoGPT	0.002	✔
AutoGen	Dify	0.000	✔
AutoGen	Semantic Kernel	0.000	✔
AutoGen	Semantic Kernel Chat	0.017	✔
AutoGPT	Dify	0.000	✔
AutoGPT	Semantic Kernel	0.000	✔
AutoGPT	Semantic Kernel Chat	0.552	✖
Dify	Semantic Kernel	0.368	✖
Dify	Semantic Kernel Chat	0.000	✔
Semantic Kernel	Semantic Kernel Chat	0.000	✔



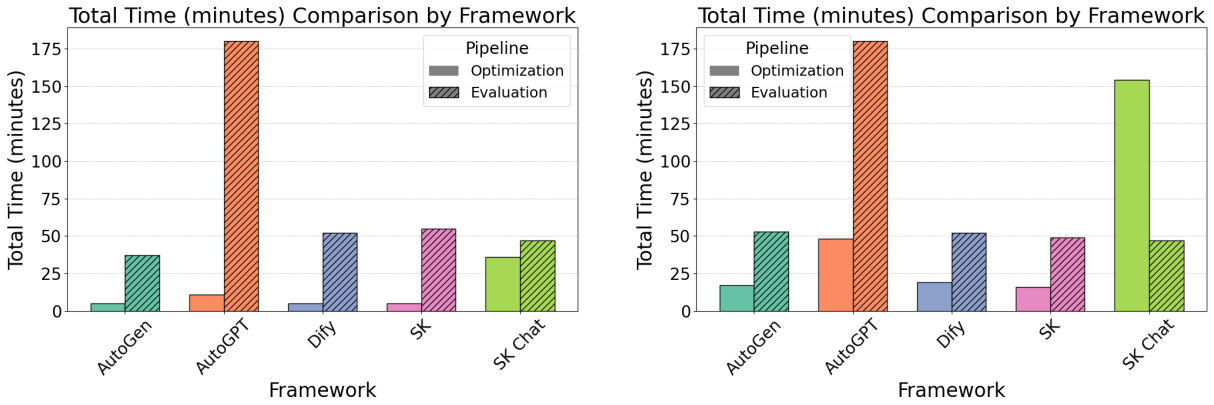
(a) TS_{10} (b) TS_{50}

Figure 17: Number of requests.

In contrast, the other analyzed frameworks only send the current prompt to the model. Their orchestration logic is handled entirely by the framework itself, without relying on the model for agent selection or function execution. As a result, these frameworks maintain significantly lower and more predictable token usage.

Regarding the evaluation pipeline (Figure 16), as mentioned earlier, total token usage remains the same across all the tested frameworks, around 1.9 million tokens, regardless of the optimization set analyzed. This consistency is expected, as the number of evaluation samples is similar across all frameworks and utilized optimization sets. Moreover, token usage across frameworks is comparable because the orchestration of the evaluation process does not rely on models; instead, it uses static prompts or predefined logic, leading to uniform resource consumption.

In addition, we measured the number of requests that the frameworks made to the model provider platform with a prompt. As expected, the number of requests follows a pattern similar to that of token usage. Figure 17 shows the number of requests by framework and by optimization sample size. We observe that the Semantic Kernel has the highest number of requests, with



(a) TS_{10} (b) TS_{50}

Figure 18: Elapsed Time.

1,121 and 4,813 requests for the TS_{10} and TS_{50} optimizations, respectively. In contrast, the other frameworks show significantly lower request counts, ranging between 149 and 179 for the TS_{10} set and 667 and 715 for the TS_{50} set.

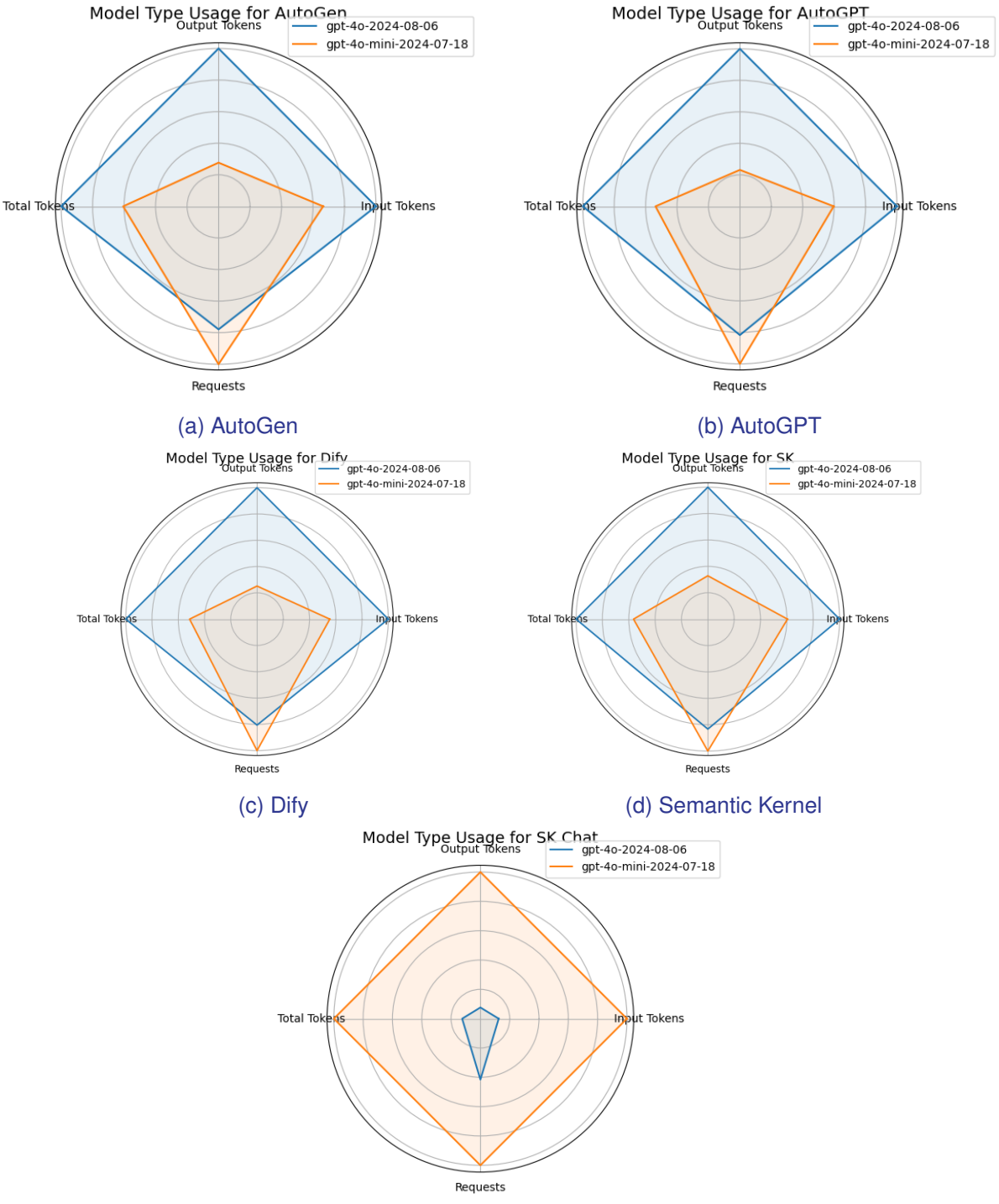
Notably, the number of requests differs among frameworks during the optimization phase. This variation is likely due to the non-deterministic nature of prompt generation, which may lead to differing numbers of optimization loops depending on the prompt generation sequence. However, this variation does not occur in the evaluation phase, where the number of requests is exactly 1,730 for all frameworks. This consistency arises because, for each of the 865 evaluation samples, two requests are made: one by `Extractor Agent` and one by `Summarizer Agent`.

The total time usage was estimated based on the start and end timestamps of service interactions on the model platform. As such, the actual time spent by the framework may be slightly higher, though the difference is not expected to be significant. It is also important to highlight that these results are approximations, as they may be influenced by external factors such as internet transmission latency and system load of the computer running the experiments.

Figure 18 illustrates the time spent by each framework during both the optimization and evaluation pipelines. As shown, execution time varies considerably across frameworks and phases. Semantic Kernel Chat framework recorded the longest optimization duration, with 36 minutes for TS_{10} set and 154 minutes for TS_{50}. The second-highest optimization times were observed with AutoGPT, which took 11 minutes and 48 minutes, respectively. The remaining three frameworks had similar optimization times, averaging around 5 minutes for 10 samples and 16–19 minutes for 50 samples.

In contrast, the evaluation pipeline shows a different pattern. AutoGPT exhibited the longest evaluation time, taking approximately 180 minutes for both the TS_{10} and TS_{50} evaluations. All other frameworks completed evaluation within a narrower range of 37 to 55 minutes, with no consistent pattern across optimization sample sizes.

These results suggest different underlying causes for the extended execution times observed in certain frameworks. In the case of Semantic Kernel Chat, debugging and terminal outputs indicate that agents were not communicating effectively during the optimization phase, which led to longer iterations and delays, a behavior not observed during evaluation. For AutoGPT, the extended duration was consistent across both phases, likely due to its internal application implementation, which appears to take longer to process each task of the pipeline type.



(a) AutoGen

(b) AutoGPT

(c) Dify

(d) Semantic Kernel

(e) Semantic Kernel Chat

Figure 19: Model usage comparison across five frameworks.

To further analyze the efficiency of the selected frameworks, we collect the usage of the underpinning agents used in the experiments, i.e., GPT4o and GPT4o-mini. At the time of conducting the experiments, OpenAI charges 5.00 USD per 1 million input tokens for the former, whereas the latter gets 0.60 USD per 1 million input tokens. Therefore, analyzing model usage is essential for interpreting both operational cost and guiding architectural decisions.

Figure 19 depicts the token usage and number of requests per model type across all frameworks. As shown in Figure 19, most frameworks exhibit higher usage of the GPT-4o model compared to GPT-4o-mini, with the notable exception of the Semantic Kernel Chat framework, which shows higher usage of GPT-4o-mini. However, this exception is not reflected in the number of requests, where GPT-4o-mini also records a higher request count for all frameworks. A likely explanation for this pattern is related to the specific agents assigned to each model. In all frameworks, the agents Teacher and PromptCombiner use GPT-4o. In particular, the Teacher agent consistently generates the longest prompts, contributing heavily to the total token usage. In the case of Semantic Kernel Chat, however, the default model used for orchestration and function calling is GPT-4o-mini. Because these steps occur frequently and are tied to model-based decision-making, this leads to higher overall usage of GPT-4o-mini in this framework.

4.3.5 Summary of the quantitative analysis

The quantitative evaluation reveals that there is no noticeable difference in the recommendation accuracy among the frameworks. With respect to efficiency, Semantic Kernel Chat requires a large number of used tokens, while AutoGPT takes much longer time to commit compared to the others. The experimental results demonstrated that Dify consistently outperforms the other frameworks in terms of ROUGE scores, achieving the highest values across all metrics. Semantic Kernel also performs well, particularly in ROUGE-1 and ROUGE-L, but falls short of Dify's performance. AutoGen and Semantic Kernel Chat show comparable results, while AutoGPT ranks lowest in all metrics. The Wilcoxon signed-rank test confirms significant differences between most framework pairs, indicating that Dify and Semantic Kernel are the most effective for the `GitHub` README summarization task.

Despite the statistically significant differences highlighted in our evaluation, two threats to validity must be acknowledged. First, LLM-based frameworks inherently exhibit *non-determinism* due to stochastic decoding, temperature, and sampling variability. Although our evaluation relied on repeated runs, residual randomness may still influence outcomes. A more extensive experimental design (e.g., multiple seeds, larger sample sizes) would further mitigate this effect. Second, while statistical tests indicated significance, the *effect sizes* of some differences were relatively small, raising questions about their practical impact. Future evaluations should therefore complement significance testing with standardized effect size measures to provide a more balanced view of both statistical and practical significance.

4.4 Conclusion

In this section, we conducted a comprehensive evaluation of selected MAS frameworks, focusing on both qualitative and quantitative aspects, with a particular emphasis on the summarization of README.MD files as a common scenario. Concerning the qualitative evaluation of the frameworks, low-code MAS frameworks offer more built-in functions, even though limitations in terms of agent selection and sharing represent a drawback. In contrast, high-code frameworks provide more flexibility and control over agent behavior, but require more development effort. The choice of framework depends on the specific requirements of the application, the level of

expertise of the developers, and the desired balance between ease of use and customization. It is worth noting that monitoring capabilities often require external telemetry tools or services, which can add complexity to the development process.

In addition, the quantitative evaluation reveals that there is no noticeable difference in the recommendation accuracy among the frameworks. With respect to Efficiency, Semantic Kernel Chat requires a large number of used tokens, while AutoGPT takes a much longer time to commit compared to the others.

In the context of MOSAICO, this analysis provides an initial overview of the technical features provided by most of the popular MAS frameworks. While we acknowledge that further investigations are needed, the identified characteristics underpin the design choices for devising the MOSAICO agent taxonomy and the corresponding repository.

5 Analysis of Model Repositories and Marketplaces

Even though LLMs and pre-trained models (PTM) are becoming increasingly adopted [17, 57, 1], developers still face several challenges when integrating these models seamlessly into software engineering workflows. Current model repositories lack standardization, making it difficult to discover, compare, and evaluate models according to specific software engineering needs.

As an explanatory example, Figure 20 depicts the Hugging Face (HF) data model, which provides a comprehensive and structured representation of the core entities and relationships within the Hugging Face Hub. This platform is recognized as a central repository for sharing, collaborating, and managing AI assets, including machine learning models, datasets, and lightweight applications (Spaces). The model highlights the dual focus of Hugging Face on both *data versioning* and *collaborative development*. At the heart of the data model lies the `repository` entity, which acts as a container for different resource types, distinguished by the `RepositoryType` enumeration (i.e., `model`, `dataset`, or `space`). Each repository maintains metadata fields such as `id`, `name`, `type`, `cardData` (for documentation), and flags like `private`, `disabled`, or `gated`, which control visibility and access. Social signals such as `likes` are also captured, emphasizing the community-driven nature of the platform.

Repositories may contain multiple files (`RepoFile`) and track versioned changes through the `Commit` entity, which logs messages, timestamps, committers, and code deltas (e.g., `deletions`, `insertions`). Each commit links to a set of `ModifiedFile` entries, detailing file-level changes, including path updates and code diffs. This structure mirrors typical version control systems like Git and provides a robust foundation for reproducibility and traceability.

In addition to versioning, the model places strong emphasis on collaboration. Each repository can host multiple `Discussion` threads, enabling asynchronous communication among contributors. These discussions are composed of various `DiscussionEvent` types, such as comments, status changes, title edits, and commit references, which collectively document the decision-making process. The `Author` entity captures information about both human and bot contributors, reflecting the hybrid human–AI development model increasingly prevalent in open-source AI. The model also includes specializations for each repository type:

- `Model` and `Dataset` repositories capture domain-specific attributes such as `pipeline_tag`, `downloads`, `citation`, and configuration files.
- `Spaces` include runtime metadata like `stage`, `hardware`, and `storage` requirements, making them suitable for interactive or deployed applications.

The Hugging Face data model provides a foundation for organizing AI assets in a collaborative environment. However, this schema lacks features that would allow developers to filter models by software engineering tasks, agent behaviors, or qualitative properties (e.g., governance, fairness, or compliance). Furthermore, existing platforms rarely support comparison across runtime requirements or advanced monitoring capabilities, complicating the reuse of models by non-expert developers. In addition to these limitations, most repositories do not expose evaluation metrics or governance attributes essential to assessing an agent's reliability and compliance. Ethical implications and biases remain also under-addressed.

Several prior works investigate model repositories and provided metadata for developers. Castano et al. [58] investigate the carbon footprints of 1,417 different models hosted on Hugging Face. The measured emission correlated with different factors such as model size, dataset size, and application domains. The same authors [58] also provide an empirical investigation on the evolution of PTMs in terms of maintenance, popularity, and usage.

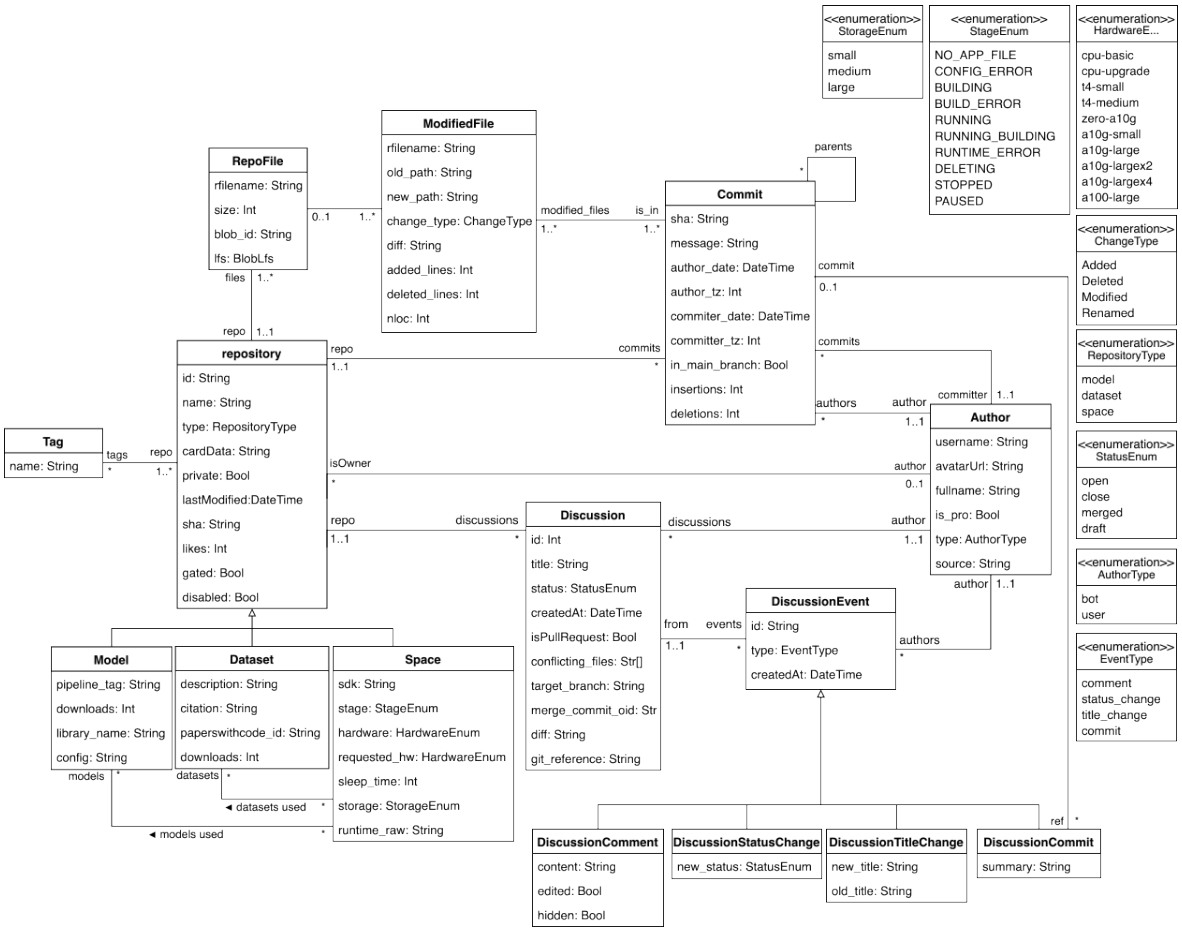


Figure 20: The Hugging Face data model.

Gong et al. [59] conducted a comprehensive study of the reuse of PTMs for software engineering tasks by considering those managed by six different model repositories, including Hugging Face. After data cleaning and labeling steps, the authors propose a code contract composed of pre- and post-conditions for the reuse in software development, e.g., input data, intended usage, and performance.

Montes et al. [60] highlight discrepancies in the documentation of 36 PTMs that support image classification across four different model repositories (i.e., TensorFlow Model Garden, ONNX Model Zoo, Torchvision Models, and Keras Applications), highlighting the need for standardized documentation. Pepe et al. [61] conducted a large-scale study on 159,132 models stored on HF by focusing on the documentation, licenses, and fairness aspects. Overall, only a few PTMs provide permissive licenses and mention potential bias in the documentation. Gao et al. [62] investigate ethical concerns in HF models leveraging the API and KeyBERT model, ending up with a dedicated taxonomy. In our previous work [63], we investigated to what extent traditional ML models, namely Naive Bayesian and SVC, can classify PTM given their model card.

Compared with the abovementioned works, we expand the analysis by *(i)* considering new challenges introduced by the *AgentWare* paradigm and *(ii)* incorporating additional repositories and marketplaces launched in recent years, which contain MAS-specific features compared to traditional model repositories.

5.1 Reusability Features

Table 20: Reusability features.

Alias	Feature	Description
R1	Model categorization	The platform supports the categorization of AI agents considering the task, e.g., summarization, or code generation.
R2	Integration of popularity, ranking mechanisms	The platform provides a mechanism to evaluate the popularity of models, e.g., number of stars, downloads, or the number of user using a specific agent.
R3	Model search	The platform provides a search functionality to find models based on specific criteria, e.g., name, category.
R4	Datasets	The platform provides datasets or use cases for training and evaluating models.
R5	Community support	The platform has an active community that contributes to its development and provides support.
R6	License	The platform provides information about the licensing of the models and datasets.
R7	Hardware and software requirements	The platform provides information about the hardware and software requirements for running the models.
R8	Agent monitoring	The platform provides evaluation metrics or benchmarks for assessing model performance.
R9	Versioning system	The platform supports versioning of models and datasets.
R10	Accessing agents	The platform provides dedicated APIs to test, train, or reuse agents without deploying them locally. Alternatively, the platform allows the reuse of the agents locally.

To systematically evaluate the reusability of AI agents across repositories, we identified a set of *reusability features* that are essential for software engineering practitioners. These features were derived from an analysis of existing repositories, frameworks, and marketplaces, as well as insights from the literature on multi-agent systems and software engineering. The identified features are summarized in Table 20. The features cover various aspects of agent reusability, including:

- **Categorization and Search** (R1, R2): Basic features for classifying agents by type, task, or domain, and enabling search functionality.
- **Metadata and Documentation** (R3, R4, R9): Detailed information about agents, including descriptions, usage examples, and documentation to facilitate understanding and integration.
- **Licensing and Compliance** (R5, R6): Clear licensing information to ensure legal compliance and proper usage rights.

Table 21: Comparison of model repositories and marketplaces.

Model repository	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Hugging Face [64]	●	●	●	●	●	◐	◐	◐	●	●
PyTorch Hub [65]	●	●	●	○	●	○	◐	○	○	●
Model Zoo [66]	●	●	●	○	○	○	○	○	○	○
Wolfram Neural Net [67]	●	○	●	◐	○	○	◐	●	○	●
Nvidia catalog [68]	●	○	●	●	○	●	○	○	●	○
Kaggle [69]	●	●	●	●	●	●	◐	○	●	○
Ollama Library [70]	◐	●	●	○	●	●	○	○	○	○
MindSpore Hub [71]	●	○	●	●	◐	●	●	○	●	○
Onnx Zoo [72]	●	○	●	○	○	○	○	○	○	○
Agent repository/marketplace										
SuperAGi marketplace [73]	●	○	●	○	○	○	○	○	●	●
OpenAI GPT store [74]	●	●	●	○	◐	○	○	○	●	●
Dify Marketplace [75]	●	●	●	○	●	●	●	○	●	●
BeeAI [76]	●	○	●	○	○	●	○	○	○	●
Agent Locker [77]	●	●	●	○	●	○	○	○	○	●
Agent List [78]	●	●	●	○	●	◐	○	○	○	●
AI toolkit [79]	●	○	●	○	●	○	○	●	○	●

● Provided ◐ Partially Provided ○ Not Provided

- **Hardware/Software Requirements** (R7): Specifications for the computational resources needed to run the agents effectively.
- **Benchmarking and Evaluation** (R8): Mechanisms for evaluating agent performance against established metrics or benchmarks.
- **Interoperability and Standards** (10): Support for standard protocols or frameworks to ensure agents can work together seamlessly.

These features will be used to compare existing repositories and frameworks in Section 5.2. The analysis will highlight gaps in current offerings and inform the design of the MOSAICO taxonomy and repository, ensuring they meet the needs of software engineering practitioners for effective agent reuse.

5.2 Comparison of Existing Repositories

Table 21 compares popular repositories against the identified reusability features. The legend uses ● to denote provided features, ◐ for partially provided features, and ○ for missing features. Most platforms provide basic model categorization (R1) and search (R3), but only a few support advanced features such as hardware/software requirements (R7), agent monitoring (R8), or detailed licensing (R6). Furthermore, agent discovery is often limited to keyword searches without semantic filters, hindering the identification of agents suited for specific SE tasks. Advanced discovery and benchmarking remain critical gaps for enabling systematic reuse in SE workflows.

Hugging Face [64] is the most widely adopted platform for hosting and sharing models. It provides APIs (Playground[27]) and community support. However, it lacks detailed licensing information [61], metadata for SE tasks [63], and continuous monitoring capabilities.

[27] https://huggingface.co/playground

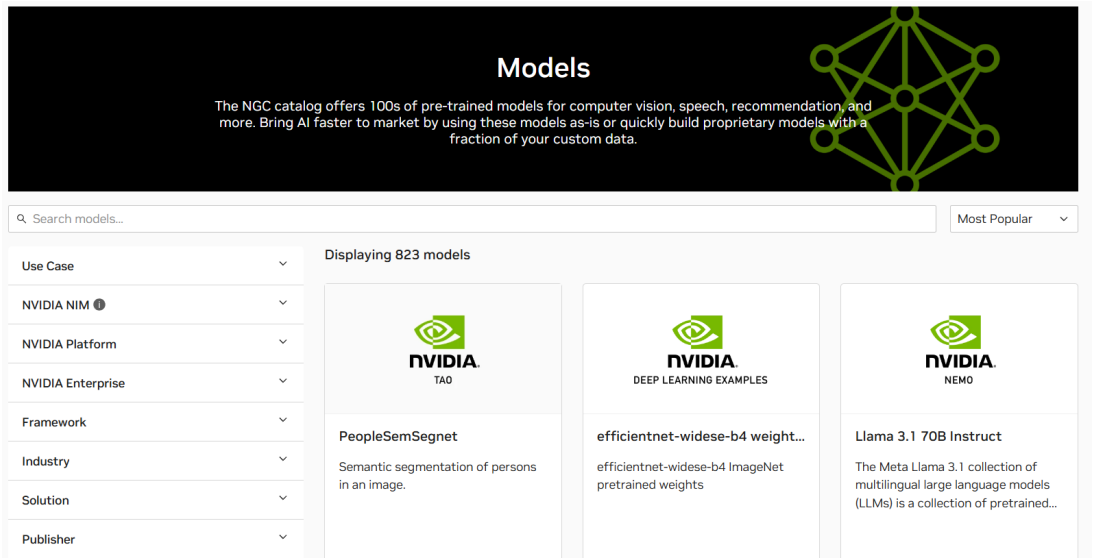


Figure 21: The Nvidia catalog.

PyTorch Hub [65] is a repository for PyTorch models, offering basic search and categorization. It supports model versioning and provides tutorials, but lacks advanced metadata and evaluation features for software engineering tasks.

ModelZoo [66] is a repository managed by Meta that provides frameworks and models but offers only basic browsing without evaluation or monitoring capabilities. It lacks comprehensive metadata for software engineering tasks, making it less suitable for SE practitioners.

Wolfram Neural Net [67] is a platform that provides a variety of pre-trained models and tools for neural network development. It offers a user-friendly interface and integration with other Wolfram products but lacks extensive metadata and evaluation features for software engineering tasks.

Nvidia Catalog (see Figure 21) provides a wide range of models and datasets, including those optimized for Nvidia hardware. It offers advanced filtering and collections but lacks comprehensive evaluation and benchmarking features for agent comparison.

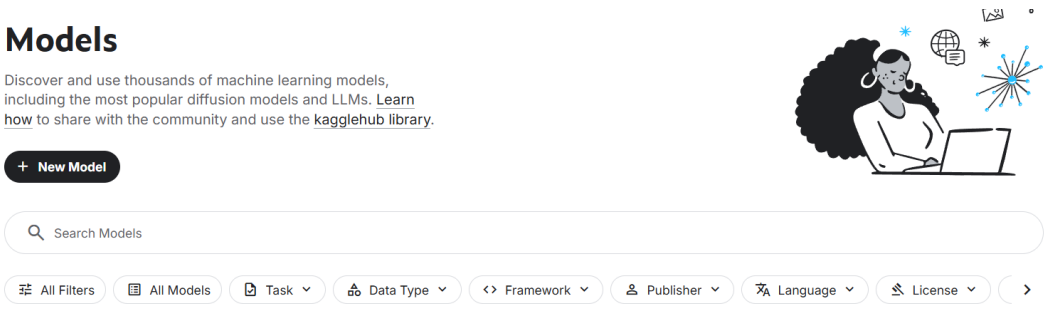


Figure 22: Model searching capabilities provided by Kaggle.

Kaggle [69] is a popular platform for data science and machine learning, hosting a variety of models and datasets. It provides community usability ratings (see (Figure 22)) and basic search features, but lacks runtime testing APIs and complete metadata for agent evaluation.

Compared to the others, *Ollama* [70] is model-oriented, providing good coverage for basic features but neglecting most advanced ones, e.g., model requirements and licensing, which are instead covered by *MindSpore Hub* [71]. The model repository with fewer features is *ONNX*

Zoo [72], which simply collects and categorizes models without providing any additional relevant metadata.

In addition to traditional model repositories, various *agent-specific marketplaces* have emerged, offering valuable insights for the MOSAICO taxonomy. *SuperAGI marketplace* [73] focuses on MAS but lacks discovery and monitoring capabilities. The *OpenAI GPT store* [74] allows easy creation and sharing of GPTs, i.e., custom versions of ChatGPT[28], but provides minimal metadata and no benchmarking.

Integrated with the *Dify* [22] MAS framework, the *Dify marketplace* [75] is a platform that allows users to create and deploy AI agents, tools, and extensions for the workflows, providing an advanced categorization feature, searching functionalities based on the artifact type, and hardware and software requirements. Furthermore, it stores and curates tools, extensions, and agent strategies, i.e., predefined workflows for particular tasks. However, it does not provide benchmarking and monitoring capabilities, including datasets and use cases.

Similar to LlamaIndex, the *BeeAI marketplace* [76] contains a list of agents that can be reused in the BeeAI platform both locally and remotely. While it provides basic features like tags, searching, and a short description of the agents, it contains only nine models at the time of writing. In addition, no community and benchmarking features are supported.

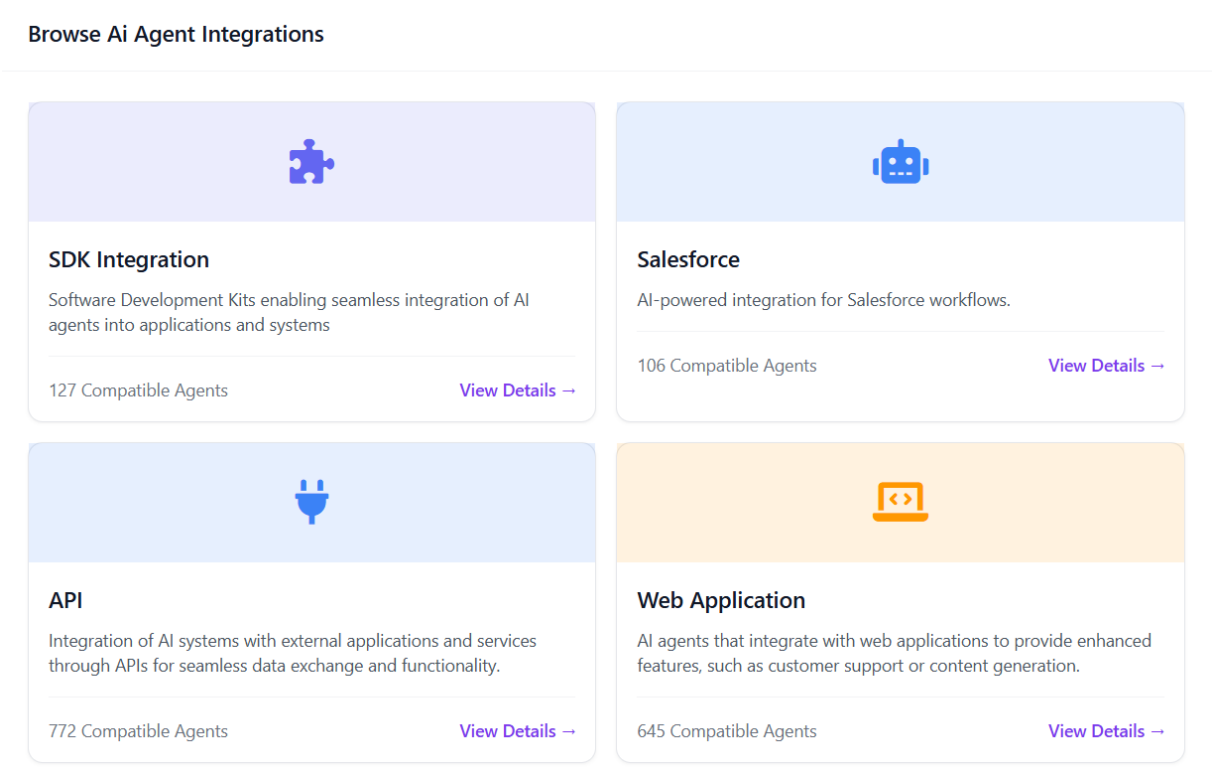


Figure 23: AgentLocker integrations

AgentLocker [77] provides a curated marketplace with structured categorization features, i.e., the agents can be classified with respect to performed tasks, use cases, and integrations with existing technologies as shown in Figure 23. Similar to the OpenAI GPT store, it also provides a list of similar agents given a selected one. However, datasets and versioning capabilities are missing, thus preventing developers from monitoring agents over time. Concerning

[28] https://help.openai.com/en/articles/8554397-creating-a-gpt

the community support, the platform allows contributions after the creation of a dedicated account.

AI Agent List platform [78] provides overall similar features to the other AI marketplaces. While the agents miss proper license files, the tagging system categorizes them with respect to their level of openness [80], allowing users to select different types of agents. Notably, the platform contains MCP servers, which are needed to build agentic applications.

Microsoft's AI Toolkit for Visual Studio Code [79] provides a marketplace for AI agents that can be used within the IDE. Unlike the other agent repositories, it provides tracing and evaluation features in the IDE environment, even though it does not provide datasets or license information for the hosted models. Concerning the contribution features, the catalog includes third-party agents as well as the possibility of creating custom models, with the support of a fine-tuning process.

These marketplaces highlight the need for standardized metadata and interoperability features, and emphasize the importance of categorizing agents by their capabilities, tasks, and deployment environments to enhance discoverability and reuse.

5.3 Conclusion

The presented analysis shows that existing platforms only partially support the needs of software engineering practitioners. Advanced metadata such as governance rules, evaluation metrics, and competency mappings are either absent or inconsistently implemented. In MOSAICO, these gaps will be addressed through a repository that integrates:

- reusability features (R1–R10) as part of the metadata model,
- discovery and benchmarking capabilities to ensure effective agent selection.

6 The MOSAICO Agent Taxonomy

This section introduces the taxonomy that characterizes how the envisioned MOSAICO AI agents are described, stored, and managed within the repository that will be devised in the context of Task 2.4. The definition of a structured taxonomy of AI agents represents a preparatory step towards the design and implementation of the MOSAICO repository. Such a taxonomy provides the conceptual foundations for systematically classifying, discovering, and benchmarking agents within the platform.

As discussed in Section 2, the need for taxonomies of AI assets is already recognized in both academic and industrial contexts. For example, the *Agntcy Directory*[29] offers a curated classification of agent skills organized into hierarchical categories, whereas the *VS Code AI Toolkit*[30] provides a practical taxonomy of models by task and framework, enabling developers to more easily integrate them within software environments. These initiatives demonstrate the importance of taxonomies as documentation artifacts and as enablers of integration and reuse of agents stored in curated marketplaces. Similarly, the Hugging Face Hub has become a de facto standard repository for models, datasets, and applications, showing how structured metadata and consistent classification can foster large-scale adoption and community engagement.

Building on these experiences, the MOSAICO taxonomy aims to capture the specific requirements of *multi-agent systems for software engineering*. Unlike the conceptual taxonomy derived from MAS frameworks (Section 3), this taxonomy focuses on the *operational aspects* necessary to enable discovery, evaluation, governance, and orchestration of agents in software engineering workflows. Its novelty lies in combining structural dimensions (e.g., roles, coordination, interaction mechanisms) with quality-related aspects (e.g., accuracy, fairness, latency), thus enabling not only classification but also systematic evaluation and benchmarking. Moreover, the taxonomy provides the basis for the MOSAICO *data model*, ensuring a seamless transition from conceptual foundations to implementation of the MOSAICO repository (see Section 7).

The taxonomy is formalized through a *metamodel* (see Figure 24) that defines the core conceptual entities that will be managed in the repository and the relationships between them. The metamodel serves as the schema for storing agent metadata in the MOSAICO Repository, enabling advanced search, benchmarking, and policy enforcement. The taxonomy follows two main principles:

- *capability-centric discovery*: agents are indexed by the software engineering tasks (skills) they support (e.g., code summarization, code generation, vulnerability detection), enabling task-driven search and composition. This aligns with external taxonomies such as A2A's `AgentSkill` model and Agntcy's capability hierarchy;
- *measurable performance*: benchmarking and telemetry are linked to agents to support their comparison, ranking, and recommendation;

[29] https://docs.agntcy.org/dir/overview/
[30] https://code.visualstudio.com/docs/intelligentapps/models

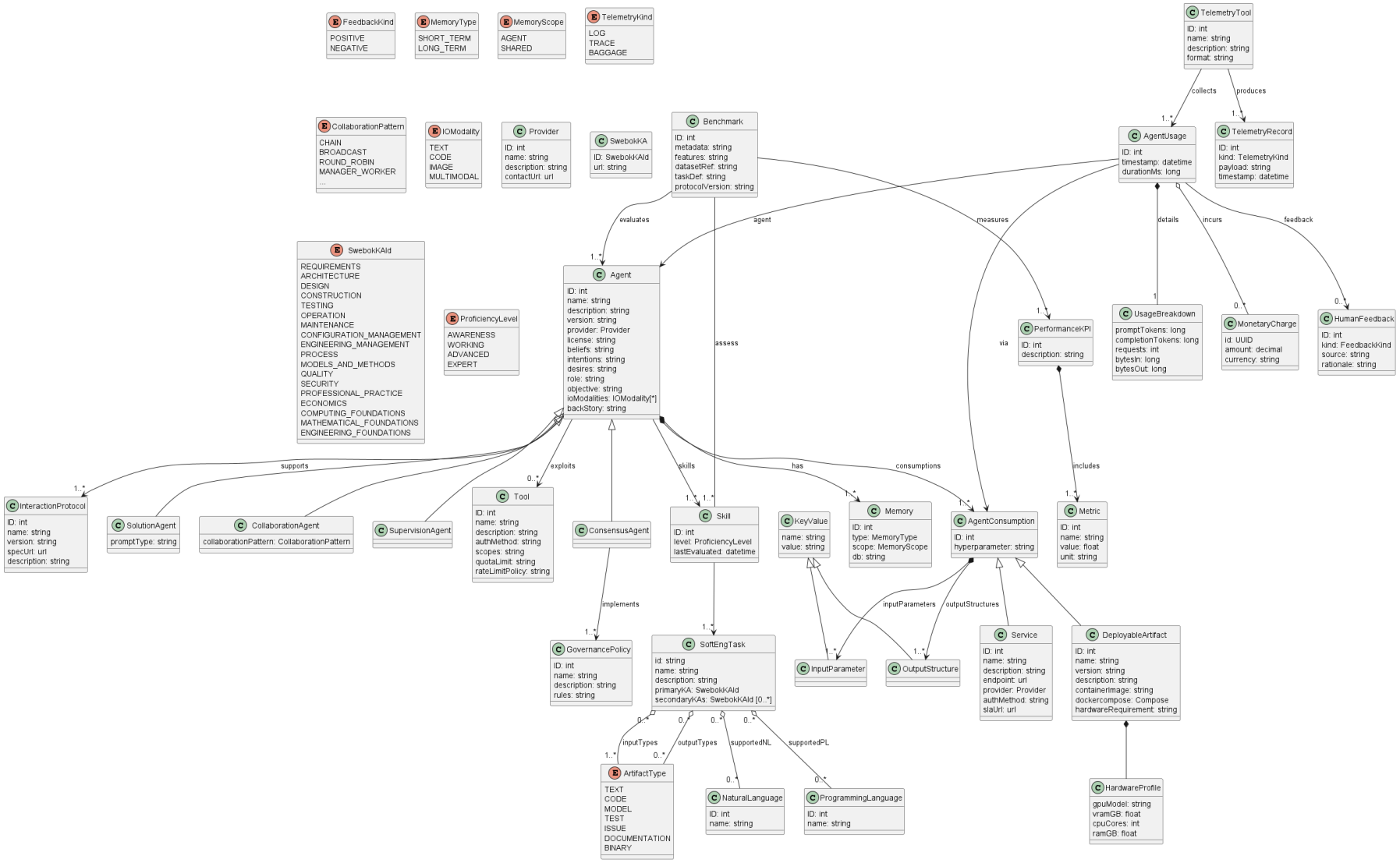


Figure 24: Metamodel representing the MOSAICO agent taxonomy.

At the core of the metamodel is the `Agent` entity, which represents an AI agent with its capabilities and metadata. Agents can be specialized into different types (i.e., `SolutionAgent`, `CollaborationAgent`, `SupervisionAgent`, `ConsensusAgent`) based on their roles and functionalities. Each agent is associated with a provider, license, and can exploit external tools while maintaining memory for stateful interactions. According to Fig. 24, each `Agent`:

- is uniquely identified and associated with metadata such as `name`, `version`, `provider`, and `license`;
- exposes one or more skills with a given `proficiency level` to perform specific software engineering tasks (`SoftEngTasks`). Software engineering tasks are indexed by their primary and secondary knowledge areas (`SwebokKAId`), associated with input/output artifact types (`ArtifactTypes`), and supported natural and programming languages. The software knowledge areas are based on the Software Engineering Body of Knowledge (SWEBOK),[31] which provides a comprehensive classification of software engineering knowledge. Thus, each `Skill` is linked to one or more `SoftEngTasks`, which represent specific software engineering tasks that an agent can perform. This distinction allows for a more granular representation of an agent's capabilities, facilitating better matching and discovery based on specific tasks required for software engineering;
- can exploit external `Tools` (with authentication, scopes, and rate limits);
- maintains `Memory` typed by *short/long* term and scoped as *agent-private/shared*;
- supports one or more `InteractionProtocols` for inter-agent communication; and
- declares one or more `AgentConsumption` options, either as a network `Service` (with `Provider`, endpoint, and optional SLA) or as a `DeployableArtifact` packaged for local execution (with `HardwareProfile` and container metadata).

Additionally, the metamodel introduces several operational entities to support evaluation, and monitoring. In particular, `Benchmark` defines datasets and procedures; its execution produces `PerformanceKPIs` composed of atomic `Metrics` (with units), yielding comparable results across versions and deployments. `Benchmarks` are also linked to `Skills`, enabling task-specific assessment. This aspect is crucial to enable performance-based ranking and recommendation of agents within the repository. Measureable benchmarks to assess and compare agents are the main focus of Deliverable D2.2 and consequently, the `Benchmark`, `PerformanceKPI`, and `Metric` classes will be further refined in that deliverable.

`AgentUsage` captures runtime invocations (inputs, outputs, context, duration, cost) and may carry `HumanFeedback` (explicit or implicit). In particular, `UsageBreakdown` holds token usage in/out, number of requests, and of received and sent bytes. `MonetaryCharge` is a separate concept: it translates usage into cost related aspects, such as currency, and amount.

`TelemetryTool` collects `TelemetryRecords` of a given kind (log, trace, baggage), enabling monitoring and diagnosis. It is important to remark that `TelemetryRecord` will be refined and extended in Task 2.4 to align with existing standards and practices in observability (e.g., OpenTelemetry[32]) and to support the specific monitoring needs of MOSAICO multi-agent systems. Finally, `GovernancePolicy` codifies consensus rules (e.g., majority voting, weighted preference, arbitration criteria) that are implemented by `ConsensusAgent` to resolve divergences among supervision agents.

[31] `https://www.computer.org/education/bodies-of-knowledge/software-engineering`
[32] `https://opentelemetry.io/`

Concerning the relationships shown in the metamodel, we can identify several key interactions between entities:

- *Discovery*. Agents are indexed by `SoftEngTask` and supported `InteractionProtocol`s. Moreover, `Provider`, `license`, and consumption options enrich search and filtering;
- *Execution*. A chosen `AgentConsumption` (`Service` or `DeployableArtifact`) determines runtime access; `Tool` usage constitutes the agent's actions within a protocol.
- *Consensus*. When supervision agents propose multiple candidate solutions, a `ConsensusAgent` enforces the defined `GovernancePolicy` to mediate and select the agreed outcome.
- *Evaluation*. `Benchmark` runs bind `Agent` versions to `PerformanceKPI` and `Metric`s; `AgentUsage` and `TelemetryRecord`s support continuous monitoring.

Table 22: Mapping between foundational concepts (FC1–FC5) and the MOSAICO taxonomy.

Dimension	Conceptual Focus	Metamodel Realization
FC1	Foundations of an agent system: application domain, agent type, memory, tools, roles	`Agent` (core entity with metadata: `name`, `version`, `provider`, `license`); `SoftEngTask` (skills/capabilities); `Memory` (short-/long term, private/shared); `Tool` (external APIs/services); specialization into `SolutionAgent`, `CollaborationAgent`, etc.
FC2	Coordination and orchestration patterns among agents	`InteractionProtocol` (supported protocols); `AgentConsumption` (`Service`, `DeployableArtifact`); consensus-driven coordination through `SupervisionAgent`, `ConsensusAgent`, and `GovernancePolicy` capturing rules for conflict resolution and orchestration.
FC3	Evaluation and continuous improvement	`Benchmark` (datasets, procedures); `PerformanceKPI` and `Metric` (accuracy, latency, fairness, resource usage); `AgentUsage` (inputs/outputs, cost, duration, feedback); `TelemetryTool` and `TelemetryRecord` for monitoring.
FC4	Individual agent behavior: goals, roles, perception, action, and consensus rules	BDI attributes in `Agent` (`beliefs`, `desires`, `intentions`); `SolutionAgent` (role/prompt type), `CollaborationAgent` (collaboration pattern); `SupervisionAgent` and `ConsensusAgent` operationalize governance by applying `GovernancePolicy` rules to reconcile multiple solutions.
FC5	Repository-level functionalities: discovery, benchmarking, comparison, and reuse	`AgentConsumption` options (`Service`, `DeployableArtifact`); `InputParameter`/`OutputStructure` for standardized I/O; benchmarking workflows and discovery mechanisms based on `SoftEngTask`, `Provider`, and consumption mode.

The presented taxonomy links the foundational dimensions (FC1–FC5) identified in the conceptual taxonomy presented in Section 3 by providing concrete entities and relationships that can be directly stored and managed in the MOSAICO Repository. Table 22 summarizes this mapping. It shows that each conceptual feature from FC1–FC5 is realized by one or more concrete classes in the metamodel, making the taxonomy executable within the repository infrastructure. The metamodel extends the conceptual taxonomy by introducing operational entities such as `TelemetryRecord`, `HumanFeedback`, and `AgentUsage`, as well as external alignments with interoperability standards (`A2AProtocol`) and provenance information (`Provider`). These extensions are crucial to support repository functionalities including *runtime monitoring*, *benchmark-driven evaluation*, and *performance-based recommendations*.

Governance is explicitly addressed under FC2 and FC4 through the interplay between `SupervisionAgent`, `ConsensusAgent`, and `GovernancePolicy`. In this setting, governance captures the consensus rules (e.g., voting, arbitration, or weighting criteria) that guide the resolution of conflicts among multiple solution proposals, ensuring that orchestration proceeds in a principled and reproducible way.

6.1 Application of the taxonomy to the summarization example

This section instantiates the MOSAICO taxonomy on the GitHub `README` summarization task introduced in Section 2. The goal is to show how the presented taxonomy supports the representation of a concrete multi-agent system, as in the simplified diagram shown in Fig. 25. To make the example more concrete, we assume the availability of a repository of agents, each exposing specific capabilities (skills) and consumable via network services or deployable artifacts. For simplicity, we focus on a subset of the taxonomy entities and relationships that are most relevant to the summarization task, without making a clear distinction between static and dynamic aspects.

We assume a repository containing multiple `Agents` including some specialized for text understanding and summarization, all interoperating through the *Agent2Agent* `InteractionProtocol`. Capabilities are indexed via `SoftEngTask` entries such as `Summarization` and `PromptOptimization`. Benchmarks (e.g., collections of `README` files with gold "About" sections) are available as `Benchmark` artifacts with associated `PerformanceKPIs` (e.g., ROUGE-L, factual consistency) and atomic `Metrics`. Runtime behaviors are observable through `TelemetryTools` that produce `TelemetryRecords` (logs, traces, baggage), and optional `HumanFeedback`.

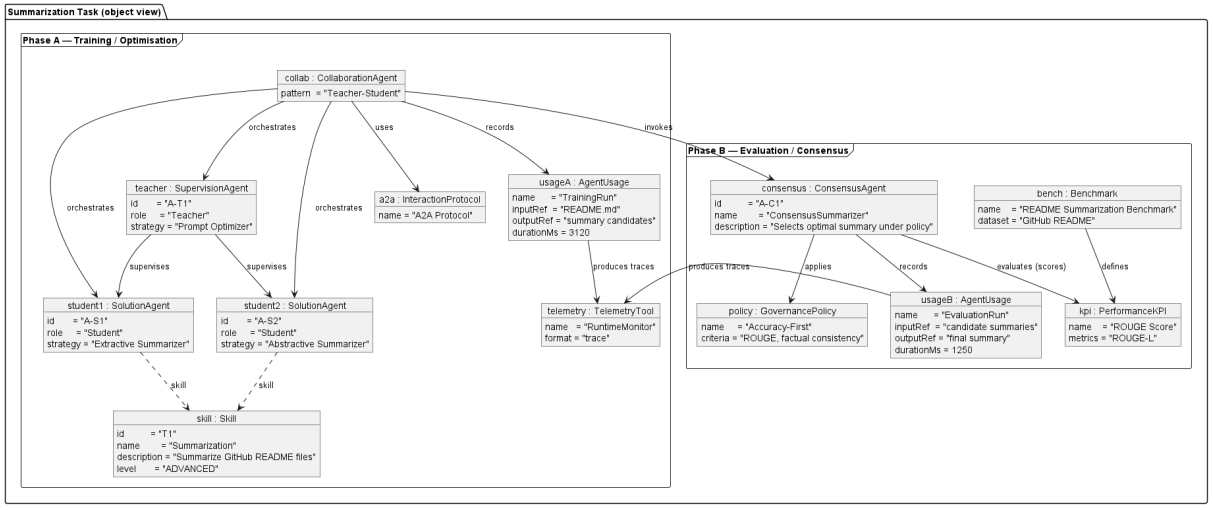


Figure 25: Fragment of a multi-agent system for summarizing GitHub `README` files, represented using concepts from the MOSAICO taxonomy.

The UML object diagram shown in Fig. 25, represents a multi-agent system organized into two main phases: *Phase A - Training/Optimization* and *Phase B - Evaluation/Consensus*. Each phase involves different types of agents, skills, protocols, and evaluation metrics, all represented using the MOSAICO taxonomy. During the training phase, a `CollaborationAgent` orchestrates a `SupervisionAgent` (teacher) and multiple `SolutionAgents` (students) to optimize summarization prompts. The agents communicate via an `InteractionProtocol` (A2A), and their performance is tracked through `AgentUsage` and `TelemetryTool`. In the evaluation

phase, a `ConsensusAgent` selects the best summary based on a defined `GovernancePolicy` and evaluates it against a `Benchmark` using specific `PerformanceKPIs`.

Listing 5 shows an explanatory JSON representation of the `SolutionAgent` implementing an *abstractive summarization* strategy (i.e., `student2` in Fig. 25). It declares its capabilities, inputs/outputs, tools, memory, and runtime configuration. The agent is consumable as a network service via an LLM provider (i.e., OpenAI), with specified model parameters.

Listing 5: Example JSON representation of a `SolutionAgent` specialized in abstractive summarization.

```
{
  "id": "A-S2",
  "name": "StudentAgent_Abstractive",
  "metaclass": "SolutionAgent",
  "role": "student",
  "swebokKA": ["MODELS_AND_METHODS"],
  "skills": ["Summarization.Abstractive","Paraphrasing","Compression"],
  "inputs": [{"name":"doc","type":"TEXT"},{"name":"prompt","type":"TEXT"
     }],
  "outputs": [{"name":"summary","type":"TEXT"}],
  "tools": [{"id":"t-openai","kind":"LLM"}],
  "memory": {"shortTerm":"disabled"},
  "runConfig": {"modelId":"gpt-4o-2024-08-06","temperature":0.3,"
     maxTokens":520},
}
```

The example usage log shown in Listing 6 contains an instance of `AgentUsage` produced when the agent is invoked during the testing phase (Phase B in Fig. 25). It records the input/output references, duration, along with telemetry attributes for traceability. The usage breakdown details token consumption, requests, and data transfer, which are essential for monitoring and billing purposes.

Listing 6: Example JSON representation of `AgentUsage` for the `SolutionAgent` during the testing phase.

```
{
  "agentUsage": {
    "id": "run-1732",
    "timestamp": "2025-09-26T09:12:10Z",
    "agentId": "A-S2",
    "modelId": "gpt-4o",
    "durationMs": 1180,
    "traceId": "5e7a..."
    ...
  },
  "usageBreakdown": {
    "promptTokens": 1420,
    "completionTokens": 480,
    "cachedTokens": 0,
    "totalTokens": 1900,
    "requests": 1,
    "bytesIn": 26800,
    "bytesOut": 9200
    ...
  }
}
```

It is important to remark that the JSON snippets are simplified for clarity. In a real-world implementation, additional fields and nested structures may be necessary to capture all relevant metadata and configurations. The examples also assume certain conventions for naming and structuring the JSON objects, which may vary based on specific implementation choices and on the telemetry and logging platforms that will be selected and used. However, the example illustrates how the MOSAICO taxonomy can be concretely applied to represent agents and their interactions in a multi-agent system for software engineering tasks.

6.2 Conclusion

The proposed taxonomy provides a comprehensive framework for understanding and organizing the various components and interactions within the MOSAICO ecosystem. By clearly defining foundational entities, their relationships, and the overarching governance structures, the taxonomy facilitates better communication, collaboration, and coordination among the different agents involved in software engineering tasks. The taxonomy serves as a blueprint for the repository's data model and aligns with the conceptual dimensions identified in the earlier analysis (FC1–FC5). It enables capability-based discovery, policy-driven orchestration, and performance-aware evaluation of agents, thus supporting the overarching goals of the MOSAICO project.

It is important to remark that the proposed taxonomy is intended to be extensible and adaptable, allowing for future enhancements as new capabilities and requirements emerge. It also aligns with existing standards and practices in the field, ensuring interoperability and compatibility with other systems and frameworks (e.g., Agent2Agent (A2A) Protocol, the Agntcy taxonomy), and industrial marketplaces, such as BeeAI, and Microsoft's AI Toolkit for VS Code.

7 The MOSAICO Agent Repository: a Preliminary Architecture

In this section we present the initial version of the MOSAICO Agent repository, by relying on the concepts defined in the taxonomy previously defined. The main components of the repository, including the agent architecture, metadata management, and integration with external services, are outlined. The architecture is presented in Section 7.1 and provides the technical foundation for agent discovery, governance enforcement, benchmarking, and orchestration. The presented architecture is intended to be iteratively refined during the project, based on feedback from use cases and evolving requirements.

The repository will be implemented as a modular, service-oriented platform based on the *Spring Framework* ecosystem, with `Spring Boot` orchestrating backend services and integrating external capabilities. The repository will expose RESTful endpoints (Section 7.2) for metadata management, taxonomy-driven search, recommendation queries, governance rule retrieval, and MCP metadata provisioning for use by MCP-compliant agents. These services are backed by the technology stack described in Section 7.3, ensuring robust integration, flexible deployment, and consistent access to persistent metadata, governance constraints, and MCP configurations. The architecture will also incorporate optional observability through `OpenTelemetry`, enabling the collection of operational data via its API to support benchmarking and recommendation generation (Section 7.5). This telemetry pipeline will feed *Benchmarking Analysis* modules.

By embedding the core concepts of the metadata model agents and of taxonomy entities, the repository transforms design abstractions into active components that drive discovery and context-aware recommendations. REST and MCP interfaces guarantee open, standardised access to this information, promoting interoperability with external agents and orchestration platforms.

7.1 Architecture Views

To present the MOSAICO Agent Repository, we provide two complementary views: a component diagram and a deployment diagram.

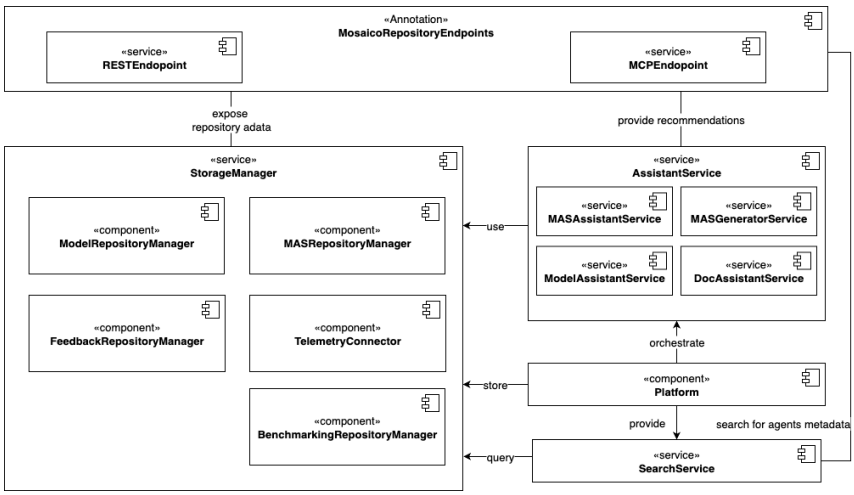


Figure 26: Preliminary component architecture of the MOSAICO Agent Repository.

Figure 26 depicts the component-level view of the repository. The architecture is organised into three main groups of services:

- *External Access Layer*, represented by the `MosaicoRepositoryEndpoints` annotation, exposes both REST and MCP endpoints for interoperable access to repository resources.
- *Core Management Layer*, centred around the `StorageManager` and its associated components (e.g., `ModelRepositoryManager`, `MASRepositoryManager`, `TelemetryConnector`), is responsible for managing metadata, telemetry connections, and benchmarking results.
- *Assistance Layer*, where the `AssistantService` orchestrates specialised assistant services (MAS, model, documentation, code generation) and interacts with the `SearchService` to retrieve metadata from the centralised search engine.

The diagram also shows the interaction flows between components, highlighting how metadata is stored, queried, and consumed.

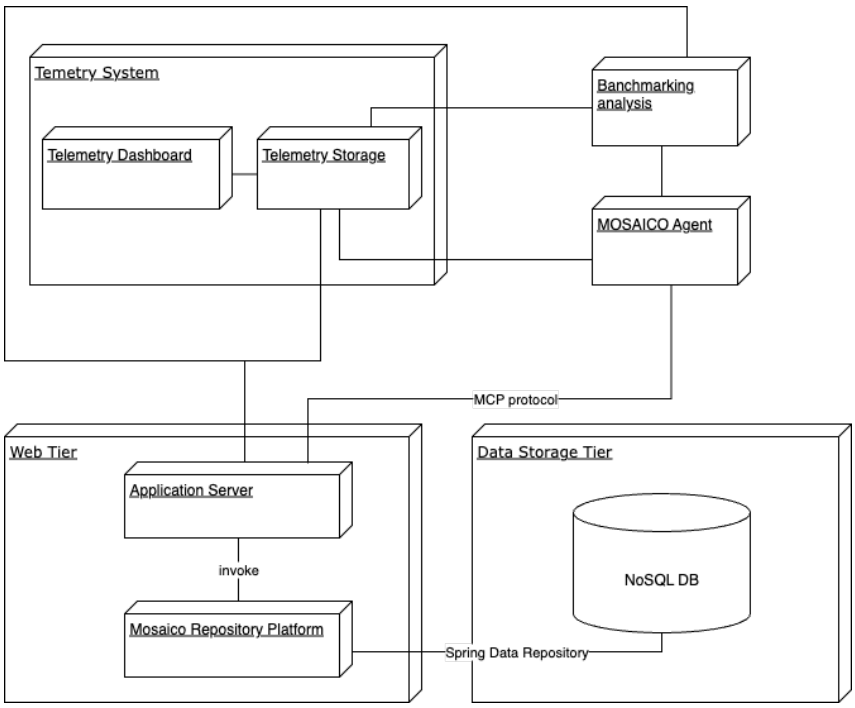


Figure 27: Preliminary deployment architecture of the MOSAICO Agent Repository.

Figure 27 presents the deployment view of the MOSAICO Agent Repository. The `Web Tier` hosts the application server running the MOSAICO Repository Platform, which interacts with the `Data Storage Tier` backed by a NoSQL DB via Spring Data repositories. `Telemetry Systems`, deployed as separate containers, provide LLM-focused dashboards and storage through `OpenTelemetry`. These systems feed into the `Benchmarking Analysis` module, whose outputs update metadata of the managed agents. Communication between tiers uses HTTP/HTTPS, Spring Data, and the MCP protocol.

7.2 External Access Interfaces: REST and MCP

The REST and MCP interfaces address the need for open, standardised access to repository metadata, enabling interoperability with external agents and orchestration platforms. The entry

point to the repository services is provided via RESTful APIs exposed through `Spring MVC`. These interfaces support the full spectrum of repository operations, including the management of metadata such as taxonomy entities, agent descriptions, and benchmarking information; the execution of search, filtering, and recommendation queries on registered agents; and the retrieval of performance metrics.

The REST API follows a consistent pattern, providing *Create, Read, Update, and Delete* (CRUD) operations for each element defined in the metadata model. This uniformity ensures that all the entities of taxonomies presented in the previous section can be programmatically managed. By exposing these operations through RESTful endpoints, the repository ensures that models, agents, datasets, and related resources can be programmatically managed in a coherent manner, facilitating integration with external tools and services within the MOSAICO ecosystem. In addition to generic CRUD capabilities, the API offers a centralised search engine as a dedicated resource, enabling advanced cross-entity queries that combine taxonomy-based filtering with metadata attributes. Agent recommendation services are accessible through dedicated endpoints; however, the corresponding interfaces are still under definition.

Beyond REST, the repository also supports the Model Context Protocol (MCP) as an external access mechanism. In this mode, the repository acts as an MCP-compliant provider, exposing its curated metadata (e.g., indexed agents, governance rules, and performance metrics) in a standardised format that can be consumed by MCP-aware clients. This allows external agents and orchestration platforms to dynamically discover, query, and integrate MOSAICO's resources without the need for custom adapters. The MCP endpoints are backed by the same metadata model described in Figure 24, ensuring consistency between internal operations and externally exposed data.

All REST interfaces are documented through an `OpenAPI` specification where applicable, enabling precise client generation in multiple programming languages. For REST development and debugging, a dedicated Swagger-based frontend is provided, while MCP access can be tested using standard MCP client tooling.

7.3 Technology Stack

The MOSAICO Agent Repository will leverage a modern, containerized technology stack to ensure scalability, maintainability, and interoperability:

- **Backend:**
  - `Spring Framework` for modular application development.
  - `Spring Boot` for backend service orchestration.
  - `Spring MVC` to expose RESTful APIs.
  - `Spring Data` for persistence and database interaction.
  - `Spring AI` and `Spring MCP` for AI integration and MCP server management.
- **Persistence:** `NoSQL database` for flexible, schema-less data storage. All persistent information is stored in a `NoSQL database`, accessed through `Spring Data` repositories. The usage of vector databases will be also investigated to store and search vector embeddings of agents and models.
- **Build and CI/CD:**
  - `Maven` for project management and builds.
  - Unit and mock testing integrated in CI/CD pipelines.

  - Automatic Docker image generation for containerized deployment.

- **Telemetry service.**
  - `OpenTelemetry` for telemetry collection; dashboards via Grafana and LangFuse for LLM-specific visualisations.

7.4 Semantic and Vector-Based Search in the MOSAICO Repository

Semantic search functionalities of the repository will be designed with the aim of supporting retrieval of agents and related entities. The core idea is to enable users to express their information needs in natural language, while the system leverages vector embeddings to find relevant objects based on semantic similarity rather than exact keyword matches. The architecture will be built around a containerised architecture in which the *embedding service*, the *vector database*, and the *backend application* are deployed as separate but interoperating components. In a typical deployment, the embedding service is provided by a dedicated container exposing an HTTP API. The backend itself orchestrates these services: it sends data to the embedding service for vectorisation, persists the resulting vectors in the database, and queries the index when semantic endpoints are invoked.

The following interaction sequence illustrates the envisioned process in practice:

1. A client submits a new entity (e.g., an `Agent` or `Skill`) to the repository API.
2. The backend forwards the entity's textual attributes to the embedding service container, which returns a vector representation.
3. The vector is stored in the vector database container, alongside the relational metadata.
4. When a user issues a semantic query, the backend generates an embedding of the query through the embedding service, retrieves the nearest neighbours from the vector database, and returns the ranked results to the client.

When a user submits a natural-language query through the `/api/search/semantic` endpoint, the system:

1. Embeds the query into the same vector space as repository objects.
2. Executes a similarity search in the vector database to find the most relevant entities.
3. Returns a ranked list of objects, potentially spanning multiple taxonomy classes.

The semantic search mechanism can be refined by combining free-text similarity with structured filters derived from the repository's metadata model. In this way, queries expressed in natural language are complemented with precise constraints that narrow the result set without sacrificing flexibility. Typical filters include restrictions on the type of entity being retrieved (e.g., limiting results to `Agents` or `Skills`), and thresholds on performance metrics (for instance, requiring $\texttt{metrics.accuracy} \geq 0.9$).

To illustrate, consider a user interested in finding agents that can support UML model validation and traceability. By means of the envisioned repository, the user can formulate a query that captures both the high-level intent and the specific requirements as shown in Listing 7. The free-text part of the query expresses the task in natural language, while structured filters restrict results to entities of type `Agent` or `Skill` and require that only those with an accuracy above 0.85 are returned. This combination ensures that the retrieved objects are not only semantically related to the user's intent but also satisfy explicit quality constraints.

Listing 7: Example repository query.

```
curl -X POST https://<host>/api/search/semantic \
-H "Content-Type: application/json" \
-d '{
        "query": "agents supporting UML model validation and
           traceability",
        "topK": 5,
        "filters": {
                "entityTypes": ["Agent","Skill"],
                "metrics.accuracy": ">=0.85"
        }
}'
```

7.5 Telemetry and Benchmarking (External Integration)

As shown in the deployment diagram in Figure 27, telemetry services will be designed as an *external* component of the MOSAICO architecture. Their integration is based on the *OpenTelemetry* standard, which provides a vendor-neutral, extensible mechanism for collecting, processing, and exporting observability data. The overall flow follows the model:

$$\text{Application (Agent)} \longrightarrow \text{OpenTelemetry Back-End} \longrightarrow \text{Dashboard}$$

In this configuration, the repository does not store or process raw telemetry data internally. Instead, it provisions a *pre-configured container* running an OpenTelemetry-compliant backend and an additional *dashboard container* dedicated to the visualization and analysis of collected data. This dashboard is customised to include a predefined frontend for monitoring telemetry from managed Agents, in line with recent developments in AI agent observability [33]. Such dashboards can display metrics such as token usage, response latency, context window utilisation, and prompt–completion correlations, providing valuable insights for the evaluation and benchmarking of AI-based agents.

Agents that require monitoring must be explicitly configured with the OpenTelemetry endpoint exposed by the MOSAICO backend. This introduces a minimal but explicit instrumentation requirement on the client side, ensuring that only those agents designed for observability participate in the telemetry workflow.

Furthermore, the repository application will be able to query the OpenTelemetry backend through its API to retrieve aggregated or filtered telemetry data. This capability will enable the generation of context-aware recommendations based on actual agent performance, operational patterns, and observed usage metrics. By incorporating live telemetry into the recommendation pipeline, the platform can refine agent suggestions over time, prioritising those that demonstrate higher efficiency, reliability, or compliance with governance rules.

The high-level interaction for telemetry integration is illustrated in Figure 28. The process starts with the developer configuring the agent with the telemetry server endpoint. The agent sends traces, metrics, and logs via the OpenTelemetry API to the OpenTelemetry backend container, which stores and processes the observability data. The telemetry dashboard (LLM-focused) retrieves and visualises these metrics for the developer, while the MOSAICO Repository application queries the backend to obtain processed telemetry, which it uses to generate context-aware recommendations.

[33] https://opentelemetry.io/blog/2025/ai-agent-observability/

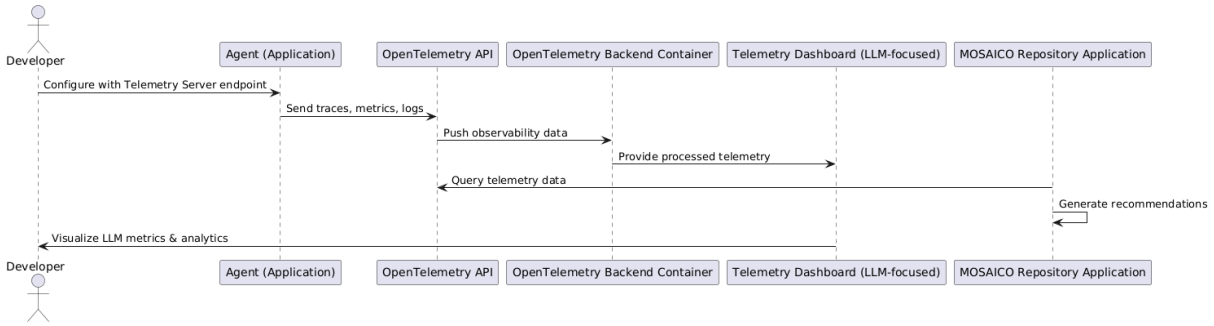


Figure 28: Sequence of interactions for telemetry integration between an instrumented agent and the MOSAICO Repository.

7.6 Conclusion

This section presented an initial version of the MOSAICO Agent Repository, which is intended to be refined throughout the project lifecycle. The repository aims to provide a comprehensive framework for managing AI agents in software engineering contexts, leveraging the proposed agent taxonomy as its foundation. By incorporating advanced telemetry and observability features, the repository will facilitate continuous improvement and adaptation of AI agents, ensuring their effectiveness and alignment with user needs.

8 Conclusion and Future Work

This document presented the agent taxonomy as the foundation for the MOSAICO repository of AI agents. By combining capability-centric discovery, measurable performance, and governed reuse, the proposed model addresses critical requirements for managing AI agents in SE contexts.

The analysis shows that existing industrial marketplaces (BeeAI, AgentLocker, Agntcy) and developer ecosystems (VS Code AI Toolkit, HuggingFace Hub) provide useful starting points, but lack crucial features such as benchmark integration, governance policies, and fine-grained metadata for SE tasks.

The deliverable also demonstrates, through a summarization case study, how the taxonomy can be concretely applied. The example highlights the value of the taxonomy in supporting collaboration patterns, benchmark-based evaluation, and policy-driven consensus in multi-agent settings.

This deliverable will guide:

- The implementation of the MOSAICO Repository, where the metamodel will be realized in a concrete storage and query architecture.
- The population of the repository with initial agent entries, enabling search, comparison, and recommendation.
- The integration with agent interaction protocols, collaboration patterns and governance mechanisms (WP1, WP3, WP4), ensuring that repository entries can be reused in practical SE workflows.

In conclusion, this document sets the stage for MOSAICO's long-term ambition: to create a repository of AI agents for Software Engineering, bridging academic innovation with industrial practice.

References

[1] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang, "Large Language Models for Software Engineering: A Systematic Literature Review", *in*: *ACM Trans. Softw. Eng. Methodol.* 33.8 (Dec. 2024), ISSN: 1049-331X, DOI: 10.1145/3695988, URL: https://doi.org/10.1145/3695988. [Cited on pages 9, 58]

[2] Ipek Ozkaya, "Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications", *in*: *IEEE Software* 40.3 (2023), pp. 4–8, DOI: 10.1109/MS.2023.3248401. [Cited on page 9]

[3] Bingyang Wei, "Requirements are All You Need: From Requirements to Code with LLMs", *in*: *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, 2024, pp. 416–422, DOI: 10.1109/RE59067.2024.00049. [Cited on page 9]

[4] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, et al., "Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks", en, *in*: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, Madrid, ES: IEEE, May 2021, pp. 336–347, ISBN: 978-1-66540-296-5, DOI: 10.1109/ICSE43902.2021.00041, URL: https://ieeexplore.ieee.org/document/9401982/ (visited on 01/30/2024). [Cited on page 9]

[5] Dagmar Monett and Claudia Lemke, "AI-ware: Bridging AI and Software Engineering for responsible and sustainable intelligent artefacts", *in*: Sept. 2021. [Cited on page 9]

[6] Nienke Nijkamp, June Sallou, Niels van der Heijden, and Luís Cruz, "Green AI in Action: Strategic Model Selection for Ensembles in Production", *in*: *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, Porto de Galinhas, Brazil: Association for Computing Machinery, 2024, pp. 50–58, ISBN: 9798400706851, DOI: 10.1145/3664646.3664763, URL: https://doi.org/10.1145/3664646.3664763. [Cited on page 9]

[7] Filipe Roseiro Cogo, Gopi Krishnan Rajbahadur, Dayi Lin, and Ahmed E. Hassan, "A Tutorial on Software Engineering for FMware", *in*: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, Porto de Galinhas, Brazil: Association for Computing Machinery, 2024, pp. 710–712, ISBN: 9798400706585, DOI: 10.1145/3663529.3663820. [Cited on pages 9, 13]

[8] James Kirkpatrick, Razvan Pascanu, Neil C. Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell, "Overcoming catastrophic forgetting in neural networks", *in*: *CoRR* abs/1612.00796 (2016), arXiv: 1612.00796, URL: http://arxiv.org/abs/1612.00796. [Cited on page 9]

[9] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu, "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions", en, *in*: *ACM Trans. Inf. Syst.* 43.2 (Jan. 2025), Read_Status: New Read_Status_Date: 2025-02-04T10:31:43.947Z, p. 3703155, ISSN: 1046-8188, DOI: 10.1145/3703155, URL: https://doi.org/10.1145/3703155 (visited on 01/17/2025). [Cited on page 9]

[10] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang, "A survey on large language model (LLM) security and privacy: The Good, The Bad, and The Ugly", *in*: *High-Confidence Computing* 4.2 (2024), p. 100211, ISSN: 2667-2952, DOI: https://doi.org/10.1016/j.hcc.2024.100211, URL: https://www.sciencedirect.com/science/article/pii/S266729522400014X. [Cited on page 9]

[11] Miao Yu, Fanci Meng, Xinyun Zhou, Shilong Wang, Junyuan Mao, Linsey Pang, Tianlong Chen, Kun Wang, Xinfeng Li, Yongfeng Zhang, Bo An, and Qingsong Wen, *A Survey on Trustworthy LLM Agents: Threats and Countermeasures*, 2025, arXiv: 2503.09648 [cs.MA], URL: https://arxiv.org/abs/2503.09648. [Cited on page 9]

[12] Junda He, Christoph Treude, and David Lo, "LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead", *in*: *ACM Trans. Softw. Eng. Methodol.* (Jan. 2025), Just Accepted, ISSN: 1049-331X, DOI: 10.1145/3712003. [Cited on page 9]

[13] Duc S. H. Nguyen, Bach G. Truong, Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio, *Teamwork makes the dream work: LLMs-Based Agents for GitHub README.MD Summarization*, 2025, arXiv: 2503.10876 [cs.SE], URL: https://arxiv.org/abs/2503.10876. [Cited on page 9]

[14] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Awadallah, Ryen W White, Doug Burger, and Chi Wang, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations", en, *in*: (2024), Read_Status: New Read_Status_Date: 2025-02-04T10:31:43.946Z. [Cited on pages 9, 13]

[15] Lijun Sun, Yijun Yang, Qiqi Duan, Yuhui Shi, Chao Lyu, Yu-Cheng Chang, Chin-Teng Lin, and Yang Shen, *Multi-Agent Coordination across Diverse Applications: A Survey*, 2025, arXiv: 2502.14743 [cs.MA], URL: https://arxiv.org/abs/2502.14743. [Cited on page 9]

[16] Bingyu Yan, Xiaoming Zhang, Litian Zhang, Lian Zhang, Ziyi Zhou, Dezhuang Miao, and Chaozhuo Li, *Beyond Self-Talk: A Communication-Centric Survey of LLM-Based Multi-Agent Systems*, 2025, arXiv: 2502.14321 [cs.MA], URL: https://arxiv.org/abs/2502.14321. [Cited on page 9]

[17] Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, Qi He, Yao Ma, Ming Huang, and Suhang Wang, *A Comprehensive Survey of Small Language Models in the Era of Large Language Models: Techniques, Enhancements, Applications, Collaboration with LLMs, and Trustworthiness*, arXiv:2411.03350 [cs] Read_Status: New Read_Status_Date: 2025-02-04T10:31:49.501Z, 2024, DOI: 10.48550/arXiv.2411.03350, URL: http://arxiv.org/abs/2411.03350 (visited on 01/17/2025). [Cited on pages 9, 58]

[18] LangChain Team, *Langchain: Building applications with LLMs through composability*, https://github.com/langchain-ai/langchain, GitHub repository, last accessed 11-02-2025, 2024. [Cited on pages 10, 25]

[19] *Llama Index: Data framework for LLM applications*, https://github.com/run-llama/llama_index, GitHub repository, last accessed 11-02-2025, 2024. [Cited on pages 10, 19]

[20] *Microsoft Semantic Kernel: Integration framework for AI models*, https://github.com/microsoft/semantic-kernel, GitHub repository, last accessed 11-02-2025, 2024. [Cited on pages 10, 20]

[21] Smolagents Team, *Smolagents: Minimalist framework for building powerful agents*, https://github.com/huggingface/smolagents, GitHub repository, last accessed 11-02-2025, 2024. [Cited on pages 10, 23]

[22] Dify Contributors, *Dify: Open-source framework for LLM applications*, https://github.com/langgenius/dify, GitHub repository, last accessed 11-02-2025, 2024. [Cited on pages 10, 23, 63]

[23] Ahmed E. Hassan, Dayi Lin, Gopi Krishnan Rajbahadur, Keheliya Gallaba, Filipe Roseiro Cogo, Boyuan Chen, Haoxiang Zhang, Kishanthan Thangarajah, Gustavo Oliva, Jiahuei (Justina) Lin, Wali Mohammad Abdullah, and Zhen Ming (Jack) Jiang, "Rethinking Software Engineering in the Era of Foundation Models: A Curated Catalogue of Challenges in the Development of Trustworthy FMware", *in*: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, Porto de Galinhas, Brazil: Association for Computing Machinery, 2024, pp. 294–305, ISBN: 9798400706585, DOI: 10.1145/3663529.3663849. [Cited on page 13]

[24] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann, "Software Engineering for Machine Learning: A Case Study", *in*: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 291–300, DOI: 10.1109/ICSE-SEIP.2019.00042. [Cited on page 13]

[25] Nikhil Kandpal, Brian Lester, Mohammed Muqeeth, Anisha Mascarenhas, Monty Evans, Vishal Baskaran, Tenghao Huang, Haokun Liu, and Colin Raffel, "Git-Theta: a git extension for collaborative development of machine learning models", *in*: *Proceedings of the 40th International Conference on Machine Learning*, ICML'23, Honolulu, Hawaii, USA: JMLR.org, 2023. [Cited on page 13]

[26] Haiyin Zhang, Luís Cruz, and Arie van Deursen, "Code smells for machine learning applications", *in*: *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*, CAIN '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 217–228, ISBN: 9781450392754, DOI: 10.1145/3522664.3528620. [Cited on page 13]

[27] Vincenzo De Martino and Fabio Palomba, "Classification and challenges of non-functional requirements in ML-enabled systems: A systematic literature review", *in*: *Information and Software Technology* 181 (2025), p. 107678, ISSN: 0950-5849, DOI: https://doi.org/10.1016/j.infsof.2025.107678, URL: https://www.sciencedirect.com/science/article/pii/S0950584925000175. [Cited on page 13]

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, ukasz Kaiser, and Illia Polosukhin, "Attention is All you Need", *in*: *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017, URL: https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html (visited on 03/31/2025). [Cited on page 13]

[29] Bernardino Romera-Paredes and Philip H. S. Torr, "An embarrassingly simple approach to zero-shot learning", *in*: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, Lille, France: JMLR.org, 2015, pp. 2152–2161. [Cited on page 13]

[30] Xuan Li, Shuai Yuan, Xiaodong Gu, Yuting Chen, and Beijun Shen, “Few-shot code translation via task-adapted prompt learning”, *in*: *Journal of Systems and Software* 212 (2024), p. 112002, ISSN: 0164-1212, DOI: https://doi.org/10.1016/j.jss.2024.112002, URL: https://www.sciencedirect.com/science/article/pii/S0164121224000451. [Cited on page 13]

[31] Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, and Taolue Chen, *Chain-of-Thought in Neural Code Generation: From and For Lightweight Language Models*, arXiv:2312.05562 [cs], 2023, URL: http://arxiv.org/abs/2312.05562 (visited on 12/18/2023). [Cited on page 13]

[32] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo, “Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues”, *in*: *ACM Trans. Softw. Eng. Methodol.* 33.5 (2024), Read_Status: New Read_Status_Date: 2025-02-04T10:31:26.839Z, 116:1–116:26, ISSN: 1049-331X, DOI: 10.1145/3643674, URL: https://dl.acm.org/doi/10.1145/3643674 (visited on 01/17/2025). [Cited on page 13]

[33] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung, “Survey of Hallucination in Natural Language Generation”, *in*: *ACM Comput. Surv.* 55.12 (2023), Read_Status: New Read_Status_Date: 2025-02-04T10:31:25.093Z tex.ids= jiSurveyHallucinationNatural2023, 248:1–248:38, ISSN: 0360-0300, DOI: 10.1145/3571730, URL: https://dl.acm.org/doi/10.1145/3571730 (visited on 01/17/2025). [Cited on page 13]

[34] Vlad-Andrei Cursaru, Laura Duits, Joel Milligan, Damla Ural, Berta Rodriguez Sanchez, Vincenzo Stoico, and Ivano Malavolta, “A Controlled Experiment on the Energy Efficiency of the Source Code Generated by Code Llama”, *in*: *Quality of Information and Communications Technology*, ed. by Antonia Bertolino, João Pascoal Faria, Patricia Lago, and Laura Semini, Cham: Springer Nature Switzerland, 2024, pp. 161–176, ISBN: 978-3-031-70245-7. [Cited on page 13]

[35] Joel Castaño, Silverio Martínez-Fernández, Xavier Franch, et al., “Exploring the Carbon Footprint of Hugging Face’s ML Models: A Repository Mining Study”, *in*: *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, arXiv:2305.11164 [cs, stat], Oct. 2023, pp. 1–12, DOI: 10.1109/ESEM56168.2023.10304801, URL: http://arxiv.org/abs/2305.11164 (visited on 01/30/2024). [Cited on page 13]

[36] Rishabh Bhardwaj, Navonil Majumder, and Soujanya Poria, “Investigating Gender Bias in BERT”, en, *in*: *Cognitive Computation* 13.4 (2021), Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 4 Publisher: Springer US, pp. 1008–1018, ISSN: 1866-9964, DOI: 10.1007/s12559-021-09881-2, URL: https://link.springer.com/article/10.1007/s12559-021-09881-2 (visited on 01/27/2025). [Cited on page 13]

[37] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber, “MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework”, en, *in*: 2023, arXiv: 2308.00352 [cs.AI], URL: https://openreview.net/forum?id=VtmBAGCN7o (visited on 12/27/2024). [Cited on page 13]

[38] Adam J. Hepworth, Daniel P. Baxter, and Hussein A. Abbass, "Onto4MAT: A Swarm Shepherding Ontology for Generalized Multiagent Teaming", en, *in*: *IEEE Access* 10 (2022), pp. 59843–59861, ISSN: 2169-3536, DOI: 10.1109/ACCESS.2022.3180032, URL: https://ieeexplore.ieee.org/document/9787500/ (visited on 02/09/2024). [Cited on page 13]

[39] Iman Naja, Milan Markovic, Peter Edwards, Wei Pang, Caitlin Cottrill, and Rebecca Williams, "Using Knowledge Graphs to Unlock Practical Collection, Integration, and Audit of AI Accountability Information", en, *in*: *IEEE Access* 10 (2022), pp. 74383–74411, ISSN: 2169-3536, DOI: 10.1109/ACCESS.2022.3188967, URL: https://ieeexplore.ieee.org/document/9815594/ (visited on 02/09/2024). [Cited on page 13]

[40] Giovanni Maria Biancofiore, Dario Di Palma, Claudio Pomo, Fedelucio Narducci, and Tommaso Di Noia, "Conversational User Interfaces and Agents", en, *in*: *Human-Centered AI: An Illustrated Scientific Quest*, ed. by Panagiotis Germanakos, Monika Juhasz, Aparna Kongot, Devashree Marathe, and Dimitris Sacharidis, Series Title: Human–Computer Interaction Series Read_Status: New Read_Status_Date: 2025-04-03T07:39:06.520Z, Cham: Springer Nature Switzerland, 2025, pp. 399–438, ISBN: 978-3-031-61374-6 978-3-031-61375-3, DOI: 10.1007/978-3-031-61375-3_4, URL: https://link.springer.com/10.1007/978-3-031-61375-3_4 (visited on 04/03/2025). [Cited on page 14]

[41] Thorsten Händler, "A Taxonomy for Autonomous LLM-Powered Multi-Agent Architectures:" en, *in*: *Proceedings of the 15th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, Read_Status: New Read_Status_Date: 2025-02-10T21:50:37.118Z, Rome, Italy: SCITEPRESS - Science and Technology Publications, 2023, pp. 85–98, ISBN: 978-989-758-671-2, DOI: 10.5220/0012239100003598, URL: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0012239100003598 (visited on 02/10/2025). [Cited on page 14]

[42] P.B. Kruchten, "The 4+1 View Model of architecture", *in*: *IEEE Software* 12.6 (1995), pp. 42–50, DOI: 10.1109/52.469759. [Cited on page 14]

[43] Hudson Borges and Marco Tulio Valente, "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform", *in*: *Journal of Systems and Software* 146 (2018), pp. 112–129, ISSN: 0164-1212, DOI: https://doi.org/10.1016/j.jss.2018.09.016, URL: https://www.sciencedirect.com/science/article/pii/S0164121218301961. [Cited on page 16]

[44] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez, *MemGPT: Towards LLMs as Operating Systems*, 2024, arXiv: 2310.08560 [cs.AI], URL: https://arxiv.org/abs/2310.08560. [Cited on page 16]

[45] Prakrit Pathak and Prashant Singh Rana, "Comparative Analysis of Pretrained Models for Text Classification, Generation and Summarization: A Detailed Analysis", *in*: *Pattern Recognition*, ed. by Apostolos Antonacopoulos, Subhasis Chaudhuri, Rama Chellappa, Cheng-Lin Liu, Saumik Bhattacharya, and Umapada Pal, Cham: Springer Nature Switzerland, 2025, pp. 151–166, ISBN: 978-3-031-78107-0. [Cited on page 25]

[46] Thu T. H. Doan, Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio, "Too long; didn't read: Automatic summarization of GitHub README.MD with Transformers", *in*: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, Oulu, Finland: Association for Computing Machinery, June 2023, pp. 267–272, ISBN: 9798400700446, DOI: 10.1145/3593434.3593448, URL: https:

//dl.acm.org/doi/10.1145/3593434.3593448 (visited on 02/01/2024). [Cited on pages 25, 45]

[47] Duc S. H. Nguyen, Bach G. Truong, Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio, "Teamwork makes the dream work: LLMs-Based Agents for GitHub README.MD Summarization", *in*: *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, Clarion Hotel Trondheim, Trondheim, Norway: Association for Computing Machinery, 2025, pp. 621–625, ISBN: 9798400712760, DOI: 10.1145/3696630.3728511, URL: https://doi.org/10.1145/3696630.3728511. [Cited on pages 25, 45, 47]

[48] Anonymous, *EASE2024-Replication-Package*, version 2.0.4, Mar. 2024, URL: https://anonymous.4open.science/r/EASE2024-HF-ReplicationPackage-B4B0/. [Cited on page 26]

[49] Chin-Yew Lin, "ROUGE: A Package for Automatic Evaluation of Summaries", *in*: *Text Summarization Branches Out*, Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81, URL: https://aclanthology.org/W04-1013/. [Cited on pages 26, 46]

[50] Francisco Zanartu, Christoph Treude, Bruno Cartaxo, Hudson Silva Borges, Pedro Moura, Markus Wagner, and Gustavo Pinto, *Automatically Categorising GitHub Repositories by Application Domain*, arXiv:2208.00269 [cs], July 2022, DOI: 10.48550/arXiv.2208.00269, URL: http://arxiv.org/abs/2208.00269 (visited on 11/24/2023). [Cited on page 45]

[51] 2021-03-23-popular-3k-python, *Dataset — Software Heritage documentation*, Dec. 2023, URL: https://docs.softwareheritage.org/devel/swh-dataset/graph/dataset.html (visited on 12/05/2023). [Cited on page 45]

[52] Adam Rule, Aurélien Tabard, and James D. Hollan, "Exploration and Explanation in Computational Notebooks", *in*: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, Montreal QC, Canada: Association for Computing Machinery, 2018, pp. 1–12, ISBN: 9781450356206, DOI: 10.1145/3173574.3173606. [Cited on page 45]

[53] Hudson Borges, Andre Hora, and Marco Tulio Valente, "Understanding the Factors That Impact the Popularity of GitHub Repositories", en, *in*: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Raleigh, NC, USA: IEEE, Oct. 2016, pp. 334–344, ISBN: 978-1-5090-3806-0, DOI: 10.1109/ICSME.2016.31, URL: http://ieeexplore.ieee.org/document/7816479/ (visited on 11/01/2023). [Cited on page 45]

[54] Tianlei Wang, Shaowei Wang, and Tse-Hsun (Peter) Chen, *Study the Correlation between the Readme File of Github Projects and Their Popularity*, en, SSRN Scholarly Paper, Rochester, NY, Nov. 2022, DOI: 10.2139/ssrn.4281782, URL: https://papers.ssrn.com/abstract=4281782 (visited on 12/01/2023). [Cited on page 45]

[55] S. S. Shapiro and M. B. Wilk, "An Analysis of Variance Test for Normality (Complete Samples)", *in*: *Biometrika* 52.3/4 (1965), pp. 591–611, ISSN: 00063444, 14643510, URL: http://www.jstor.org/stable/2333709 (visited on 05/28/2025). [Cited on page 46]

[56] Frank Wilcoxon, "Individual Comparisons by Ranking Methods", *in*: *Biometrics Bulletin* 1.6 (1945), pp. 80–83, ISSN: 00994987, URL: http://www.jstor.org/stable/3001968 (visited on 05/26/2025). [Cited on page 46]

[57] Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang, "A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges", en, *in*: *Vicinagearth* 1.1 (2024), Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 1 Publisher: Springer Nature Singapore Read_Status: New Read_Status_Date: 2025-02-04T10:31:49.500Z, pp. 1–43, ISSN: 3005-060X, DOI: 10.1007/s44336-024-00009-2, URL: https://link.springer.com/article/10.1007/s44336-024-00009-2 (visited on 01/27/2025). [Cited on page 58]

[58] Joel Castaño, Silverio Martínez-Fernández, Xavier Franch, and Justus Bogner, *Analyzing the Evolution and Maintenance of ML Models on Hugging Face*, arXiv:2311.13380 [cs], Nov. 2023, DOI: 10.48550/arXiv.2311.13380, URL: http://arxiv.org/abs/2311.13380 (visited on 01/30/2024). [Cited on page 58]

[59] Lina Gong, Jingxuan Zhang, Mingqiang Wei, Haoxiang Zhang, and Zhiqiu Huang, "What Is the Intended Usage Context of This Model? An Exploratory Study of Pre-Trained Models on Various Model Repositories", *in*: *ACM Trans. Softw. Eng. Methodol.* 32.3 (2023), 69:1–69:57, ISSN: 1049-331X, DOI: 10.1145/3569934, (visited on 02/17/2024). [Cited on page 59]

[60] Diego Montes, Pongpatapee Peerapatanapokin, Jeff Schultz, Chengjun Guo, Wenxin Jiang, et al., "Discrepancies among pre-trained deep neural networks: a new threat to model zoo reliability", *in*: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 1605–1609, ISBN: 978-1-4503-9413-0, DOI: 10.1145/3540250.3560881, URL: https://dl.acm.org/doi/10.1145/3540250.3560881 (visited on 01/29/2024). [Cited on page 59]

[61] Federica Pepe, Vittoria Nardone, Antonio Mastropaolo, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta, "How do Hugging Face Models Document Datasets, Bias, and Licenses? An Empirical Study", *in*: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, ICPC '24, Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 370–381, ISBN: 9798400705861, DOI: 10.1145/3643916.3644412, URL: https://doi.org/10.1145/3643916.3644412. [Cited on pages 59, 61]

[62] Haoyu Gao, Mansooreh Zahedi, Christoph Treude, Sarita Rosenstock, and Marc Cheong, "Documenting Ethical Considerations in Open Source AI Models", *in*: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '24, Barcelona, Spain: Association for Computing Machinery, 2024, pp. 177–188, ISBN: 9798400710476, DOI: 10.1145/3674805.3686679, URL: https://doi.org/10.1145/3674805.3686679. [Cited on page 59]

[63] Claudio Di Sipio, Riccardo Rubei, Juri Di Rocco, Davide Di Ruscio, and Phuong T. Nguyen, "Automated categorization of pre-trained models in software engineering: A case study with a Hugging Face dataset", *in*: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, Salerno, Italy: Association for Computing Machinery, 2024, pp. 351–356, ISBN: 9798400717017, DOI: 10.1145/3661167.3661215, URL: https://doi.org/10.1145/3661167.3661215. [Cited on pages 59, 61]

[64] *Hugging Face*, https://huggingface.co/, Accessed: Mar 19, 2025. [Cited on page 61]

[65] *PyTorch Hub*, https://pytorch.org/hub/, Accessed: Mar 19, 2025. [Cited on pages 61, 62]

[66] *Model Zoo*, https://modelzoo.co/, Accessed: Mar 19, 2025. [Cited on pages 61, 62]

[67] *Wolfram Neural Net Repository*, https://resources.wolframcloud.com/NeuralNetRepository, Accessed: Mar 19, 2025. [Cited on pages 61, 62]

[68] *Nvidia NGC Catalog*, https://catalog.ngc.nvidia.com/models, Accessed: Mar 19, 2025. [Cited on page 61]

[69] *Kaggle Models*, https://www.kaggle.com/models, Accessed: Mar 19, 2025. [Cited on pages 61, 62]

[70] *Ollama Library*, https://ollama.com/library, Accessed: Mar 19, 2025. [Cited on pages 61, 62]

[71] Mindspore Community, *Mindspore*, https://www.mindspore.cn/resources/hub/en, GitHub repository, last accessed 11-02-2025, 2025. [Cited on pages 61, 62]

[72] Onnx Community, *Onnx model zoo*, https://onnx.ai/models/, GitHub repository, last accessed 11-02-2025, 2025. [Cited on pages 61, 63]

[73] *SuperAGi Marketplace*, https://superagi.com/docs/Marketplace/agent_template_marketplace, Accessed: Mar 19, 2025. [Cited on pages 61, 63]

[74] *OpenAI GPT store*, https://chatgpt.com/gpts, Accessed: Mar 19, 2025. [Cited on pages 61, 63]

[75] Dify, *Dify Marketplace*, https://marketplace.dify.ai/, Last accessed: Aug 18, 2025. [Cited on pages 61, 63]

[76] BeeAI marketplace, *BeeAI*, https://beeai.dev/agents, Last accessed: Aug 7, 2025. [Cited on pages 61, 63]

[77] AgentLocker marketplace, *AgentLocker*, https://www.agentlocker.ai/, Last accessed: Aug 7, 2025. [Cited on pages 61, 63]

[78] AgentList marketplace, *AgentList*, https://aiagentslist.com/, Last accessed: Aug 7, 2025. [Cited on pages 61, 64]

[79] *Microsoft AI Toolkit*, https://code.visualstudio.com/docs/intelligentapps/models, Accessed: Aug 7, 2025. [Cited on pages 61, 64]

[80] Matt White, Ibrahim Haddad, Cailean Osborne, Xiao-Yang Yanglet Liu, Ahmed Abdelmonsef, Sachin Varghese, and Arnaud Le Hors, *The Model Openness Framework: Promoting Completeness and Openness for Reproducibility, Transparency, and Usability in Artificial Intelligence*, 2024, arXiv: 2403.13784 [cs.LG], URL: https://arxiv.org/abs/2403.13784. [Cited on page 64]

List of selected papers

[P1] J. He, C. Treude, D. Lo, Llm-based multi-agent systems for software engineering: Literature review, vision and the road ahead, ACM Trans. Softw. Eng. Methodol. (2025). DOI: doi:10.1145/3712003.

[P2] Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N. V., Wiest, O., and Zhang, X. (2024). Large Language Model Based Multi-agents: A Survey of Progress and Challenges. 9, 8048–8057. 10.24963/ijcai.2024/890.

[P3] Dhamani, D., and Maher, M. L. (2023). The Tyranny of Possibilities in the Design of Task-Oriented LLM Systems: A Scoping Survey (arXiv:2312.17601; Versione 1). arXiv. 10.48550/arXiv.2312.17601.

[P4] Xi, Z., Chen, W., Guo, X. et al. The rise and potential of large language model based agents: a survey. Sci. China Inf. Sci. 68, 121101 (2025). 10.1007/s11432-024-4222-0.

[P5] Mu, C., Guo, H., Chen, Y., Shen, C., Hu, D., Hu, S., and Wang, Z. (2024). Multi-agent, human–agent and beyond: A survey on cooperation in social dilemmas. Neurocomputing, 610, 128514. 10.1016/j.neucom.2024.128514.

[P6] Sun, C., Huang, S., and Pompili, D. (2024). LLM-based Multi-Agent Reinforcement Learning: Current and Future Directions (arXiv:2405.11106; Versione 1). arXiv. 10.48550/arXiv.2405.11106.

[P7] Cheng, Y., Zhang, C., Zhang, Z., Meng, X., Hong, S., Li, W., Wang, Z., Wang, Z., Yin, F., Zhao, J., and He, X. (2024). Exploring Large Language Model based Intelligent Agents: Definitions, Methods, and Prospects (arXiv:2401.03428; Versione 1). arXiv. 10.48550/arXiv.2401.03428.

[P8] Sun, M., Han, R., Jiang, B., Qi, H., Sun, D., Yuan, Y., and Huang, J. (2024). A Survey on Large Language Model-based Agents for Statistics and Data Science (arXiv:2412.14222; Versione 1). arXiv. 10.48550/arXiv.2412.14222.

[P9] H. Du, S. Thudumu, R. Vasa, K. Mouzakis, A Survey on Context-Aware Multi-Agent Systems: Techniques, Challenges and Future Directions, 2024. DOI: 10.48550/arXiv.2402.01968.

[P10] Jin, W., Du, H., Zhao, B., Tian, X., Shi, B., and Yang, G. (2025). A Comprehensive Survey on Multi-Agent Cooperative Decision-Making: Scenarios, Approaches, Challenges and Perspectives (SSRN Scholarly Paper 5106265). Social Science Research Network. 10.2139/ssrn.5106265.

[P11] Yu, M., Meng, F., Zhou, X., Wang, S., Mao, J., Pang, L., Chen, T., Wang, K., Li, X., Zhang, Y., An, B., and Wen, Q. (2025). A Survey on Trustworthy LLM Agents: Threats and Countermeasures (arXiv:2503.09648). arXiv. 10.48550/arXiv.2503.09648.

[P12] Wu, Y., Li, D., Chen, Y., Jiang, R., Zou, H. P., Fang, L., Wang, Z., and Yu, P. S. (2025). Multi-Agent Autonomous Driving Systems with Large Language Models: A Survey of Recent Advances (arXiv:2502.16804; Versione 1). arXiv. 10.48550/arXiv.2502.16804.

[P13] Sun, L., Yang, Y., Duan, Q., Shi, Y., Lyu, C., Chang, Y.-C., Lin, C.-T., and Shen, Y. (2025). Multi-Agent Coordination across Diverse Applications: A Survey (arXiv:2502.14743; Versione 2). arXiv. 10.48550/arXiv.2502.14743.

[P14] Yan, B., Zhang, X., Zhang, L., Zhang, L., Zhou, Z., Miao, D., and Li, C. (2025). Beyond Self-Talk: A Communication-Centric Survey of LLM-Based Multi-Agent Systems (arXiv:2502.14321; Versione 1). arXiv. 10.48550/arXiv.2502.14321.

[P15] Singh, A., Ehtesham, A., Kumar, S., and Khoei, T. T. (2025). Agentic Retrieval-Augmented Generation: A Survey on Agentic RAG (arXiv:2501.09136; Versione 3). arXiv. 10.48550/arXiv.2501.09136.

[P16] Sun, C., Huang, S., and Pompili, D. (2025). LLM-Based Multi-Agent Decision-Making: Challenges and Future Directions. IEEE Robotics and Automation Letters, 10(6), 5681–5688. 10.1109/LRA.2025.3562371

[P17] Aratchige, R. M., and Ilmini, W. M. K. S. (2025). LLMs Working in Harmony: A Survey on the Technological Aspects of Building Effective LLM-Based Multi Agent Systems (arXiv:2504.01963). arXiv. 10.48550/arXiv.2504.01963

[P18] Tran, K.-T., Dao, D., Nguyen, M.-D., Pham, Q.-V., O'Sullivan, B., and Nguyen, H. D. (2025). Multi-Agent Collaboration Mechanisms: A Survey of LLMs (arXiv:2501.06322). arXiv. 10.48550/arXiv.2501.06322

List of selected frameworks

[MF1] **AutoGPT**. (2023). *AutoGPT*. `https://github.com/Significant-Gravitas/AutoGPT`. Last accessed 2025-09-23.

[MF2] **LangChain**. (2023). *LangChain*. `https://github.com/langchain-ai/langchain`. Last accessed 2025-09-23.

[MF3] **Dify**. (2023). *Dify*. `https://github.com/langgenius/dify`. Last accessed 2025-09-23.

[MF4] **MetaGPT**. (2023). *MetaGPT*. `https://github.com/geekan/MetaGPT`. Last accessed 2025-09-23.

[MF5] **AutoGen**. (2023). *AutoGen*. `https://github.com/microsoft/autogen`. Last accessed 2025-09-23.

[MF6] **Llama Index**. (2023). *Llama Index*. `https://github.com/run-llama/llama_index`. Last accessed 2025-09-23.

[MF7] **Flowise**. (2023). *Flowise*. `https://github.com/FlowiseAI/Flowise`. Last accessed 2025-09-23.

[MF8] **CrewAI**. (2023). *CrewAI*. `https://github.com/joaomdmoura/crewai`. Last accessed 2025-09-23.

[MF9] **Semantic Kernel**. (2023). *Semantic Kernel*. `https://github.com/microsoft/semantic-kernel`. Last accessed 2025-09-23.

[MF10] **Agno**. (2025). *Agno*. `https://github.com/agno-agi/agno`. Last accessed 2025-09-23.

[MF11] **Haystack**. (2019). *Haystack*. `https://github.com/deepset-ai/haystack`. Last accessed 2025-09-23.

[MF12] **OpenAI Agents**. (2025). *OpenAI Agents*.[34] `https://github.com/openai/openai-agents-python`. Last accessed 2025-09-25.

[MF13] **Smolagents**. (2024). *Smolagents*. `https://github.com/huggingface/smolagents`. Last accessed 2025-09-25.

[MF14] **Letta AI**. (2023). *Letta*. `https://github.com/letta-ai/letta`. Last accessed 2025-09-23.

[34] Formerly referred to as *SwarmAI*.