A Hybrid Role-Based Reference Architecture for LLM-Enhanced Multi-Agent Systems

Tansu Zafer Asici[1[0000−0003−0495−8198]], Önder Gürcan[2[0000−0001−6982−5658]], and Geylani Kardas[1[0000−0001−6975−305X]]

[1] Ege University,
International Computer Institute,
35100, Izmir, Türkiye
{tansu.asici,geylani.kardas}@ege.edu.tr
[2] NORCE Research Center AS
Center for Modeling Social Systems
Kristiansand, Norway
onder.gurcan@norceresearch.no

Abstract. Large language models (LLMs) are transforming how we build multi-agent systems (MAS); yet, many LLM-centric frameworks still lack the engineering rigour that agent-oriented software engineering (AOSE) provides, resulting in systems that are powerful but difficult to maintain and scale. In our previous work, we critically examined the "role" concept across definition, specification, and implementation, and proposed a preliminary hybrid role-based architecture where roles are treated as first-class run-time entities that support four different action implementation types. However, that earlier work remained at a conceptual level: it identified the need for typed actions and runtime roles but did not provide a formal meta-model specifying how these constructs relate to one another, nor did it offer a concrete realization or validation. Building on that foundation, this paper closes this gap by defining a role meta-model for LLM-enhanced agents that specifies the core role constructs, their interfaces, constraints, and interaction relationships, with clear variation points for design-time and run-time implementation. We realize this meta-model as a framework-agnostic Java annotation set: any Java-based agent framework can adopt the annotations to expose roles, actions, and interaction points declaratively in code and validate them at run-time. We demonstrate the applicability of our approach by implementing a hotel reservation scenario in the SCOP framework, where each agent type is realized through dedicated role specifications and role implementations combining hybrid action types. Finally, we discuss practical design considerations—deliberation–execution separation, action-type boundary decisions, and observability and debugging—offering guidance toward production-grade LLM-enhanced MAS.

Keywords: Multi-agent systems · large language models · agent-oriented software engineering · role-based architecture · software design patterns

1 Introduction

Large language models (LLMs) are rapidly reshaping how multi-agent systems (MAS) are built, enabling agents that can interpret natural language, generate plans, and interact flexibly with humans and external tools. In practice, however, many LLM-centric MAS frameworks emphasize orchestration and prompt engineering over systematic software engineering structure [22,21]. This often produces systems that are impressive in demonstrations but difficult to maintain, test, debug, and scale in real deployments—where predictability, traceability, scalability, and security constraints matter [37,41,31].

A core reason for this gap is that "roles" in many LLM-enabled frameworks remain primarily prompt-level descriptions (e.g., "act as a travel agent"), rather than explicit, queryable, run-time constructs with well-defined interfaces, constraints, and interaction relationships [5]. This design choice undermines observability and safety: without run-time role representations, developers lack the tools to reliably inspect capabilities in a running system, enforce boundaries on what an agent is allowed to do, validate action availability and preconditions, or evolve behavior in a controlled way without introducing unintended or unsafe side effects. The issue is compounded by LLM non-determinism: small prompt changes can induce large behavioral shifts [30,10], which complicates regression testing and systematic debugging. As a result, teams often rely on iterative trial-and-error rather than principled design-time artifacts that can be traced to implementation and verified at run-time.

In our previous study [5], we tackled this problem directly by critically examining the role concept across definition, specification, and implementation, and by surveying common LLM-powered MAS tools/frameworks to expose where ad hoc "role prompting" departs from established Agent-Oriented Software Engineering (AOSE) foundations. We then proposed an initial hybrid role-based architecture in which roles are treated as first-class run-time entities, capable of encapsulating both AOSE-informed structure and LLM-driven functionality, and we illustrated how different action realization types (`LOCAL`, `WEB_SERVICE`, `LLM`, and `MCP_TOOL`) can coexist within a unified role abstraction.

Building on that foundation, this paper advances the architecture into a reusable reference architecture for engineering LLM-enhanced MAS. The key idea is to make role specifications actionable across design-time and run-time: roles expose explicit interfaces and constraints; each action declares its implementation type; and interaction points are made uniform so that LLM capabilities can be integrated without sacrificing engineering rigor. The resulting approach targets production-grade LLM-enhanced MAS by improving consistency, maintainability, observability, and security isolation.

This paper makes the following explicit contributions:

1. A prescriptive, reusable reference architecture for LLM-enhanced MAS centered on roles as first-class runtime entities, operationalizing the earlier hybrid role-based concept into an actionable architectural blueprint.
2. A role meta-model for LLM-enhanced agents that specifies core role constructs (interfaces, constraints, and interaction relationships) and highlights

clear variation points so roles can be implemented consistently from design-time to run-time.
3. A Java annotation-based implementation of the meta-model for declarative roles, actions, and relationships, with runtime validation and framework-agnostic support for Java agent platforms.
4. An end-to-end demonstration in a hotel reservation scenario implemented in the SCOP[3] framework, showing how agent types can be realized through role specifications and role implementations that combine hybrid action types.
5. A discussion of practical design considerations for LLM-enhanced MAS, covering deliberation–execution separation, action-type boundary decisions, and observability and debugging strategies.

Section 2 surveys related work. Section 3 presents the hybrid role-based reference architecture, defining roles as runtime entities and introducing typed action implementation strategies. Section 4 defines the role meta-model that formalizes core constructs, constraints, and interaction relationships. Section 5 realizes this meta-model in Java through declarative annotations. Section 6 demonstrates the approach through a concrete hotel reservation scenario implemented in SCOP. Section 7 discusses design considerations, actionable insights, and practical challenges. Section 8 concludes the paper with directions for future work.

2 Related Work

Recent LLM-based agent frameworks provide powerful orchestration mechanisms but typically treat roles as workflow nodes, prompt descriptions, or implementation-specific classes rather than as explicit, queryable software engineering constructs [20]. *LangGraph*[4] models agent behaviour as state graphs, *CrewAI*[5] exposes role, goal, and backstory fields as natural-language context with only post-hoc output guardrails, *MetaGPT*[6] [23] implements roles as Python classes within SOP-driven workflows where all actions are LLM-driven, and the *Microsoft Agent Framework*[7] [40] offers graph-based orchestration with MCP tool integration. None of them provides a framework-agnostic role meta-model with typed action semantics, runtime capability inspection, and explicit safety constraints—the gap our architecture fills by making roles first-class runtime entities whose actions, states, responsibilities, and lifecycle hooks are structured metadata.

Hybrid AOSE and neurosymbolic approaches keep parts of the agent architecture outside the LLM: Borghoff et al. [8] restrict LLMs to topic extraction while Coloured Petri Nets govern workflow execution; Al Owayyed et al. [3] combine BDI control with LLM capabilities in educational agents; and Aguzzi et al. [2] treat LLMs as tools invoked by external software in PlanchBDI. We

[3] `https://scop-framework.netlify.app/`, last access on Jul. 7, 2026.
[4] `https://www.langchain.com/langgraph`, last access on Jul. 7, 2026.
[5] `https://www.crewai.com/`, last access on Jul. 7, 2026.
[6] `https://github.com/geekan/MetaGPT`, last access on Jul. 7, 2026.
[7] `https://github.com/microsoft/agent-framework`, last access on Jul. 7, 2026.

share their principle that LLMs should not control the whole agent loop, but our contribution is architectural rather than paradigm-specific: we encapsulate LLM and non-LLM capabilities as typed actions attached to inspectable roles, letting deterministic code, web services, LLM calls, and MCP tools coexist under a common abstraction.

Traditional AOSE has long modelled roles and organisational structures explicitly. $\mathcal{M}$oise$^+$ [24] defines organisations through structural, functional, and deontic specifications, and related normative models include OperA [14] and Electronic Institutions [15]; Lorini [28] similarly argues that conversational agents require formal structures for intentional communication and norm compliance. These approaches were designed for closed symbolic systems with predefined plans and norm-monitoring infrastructures, and do not directly address non-deterministic LLM actions or tool-calling workflows. We adopt a lighter mechanism: annotation-driven responsibilities, must-always/must-never safety properties, and before/after action hooks that operate at the execution boundary regardless of whether the action is deterministic or LLM-generated.

Recent BDI+LLM work further explores neurosymbolic combinations: NatBDI [25] represents beliefs, contexts, and plans in natural language with inference-based plan selection; ChatBDI [18] adds natural-language interaction to Jason/-JaCaMo agents through LLM-based translation between human utterances and agent communication; Schulz and Jander [35] use LLMs to generate BDI plans at runtime; and Frering et al. [17] combine BDI safety checks with LLM-based command interpretation for human–robot interaction. These works focus on plan generation, natural-language interaction, or single-agent safety mediation, whereas we target a reusable role-based reference architecture for LLM-enhanced MAS in which each role exposes typed capabilities, safety metadata, communication preferences, and lifecycle hooks inspectable and enforceable at runtime. Beyond runtime architectures, Hajjar Zadeh et al. [21] use LLMs at design-time to support MaSE/JADE-based MAS code generation; our work is complementary, defining runtime structures that govern how LLM and non-LLM actions are represented, selected, executed, and monitored inside deployed systems.

3 Hybrid Role-based Reference Architecture

In AOSE, a role is an abstract definition that encapsulates three essential elements. First, *responsibilities* define what the agent must accomplish—the goals and obligations associated with the role. Second, *behaviors* (i.e. higher-level patterns or policies composed of *actions*) specify how the agent should act to fulfill those responsibilities. Third, *interaction protocols* govern how agents communicate and coordinate with one another. This abstraction is supported across methodologies, highlighting both static design-time specifications and dynamic, runtime role adaptations that enhance multi-agent system robustness and flexibility [34,9,16,32,39]. Roles promote modularity, reusability, and flexibility: an agent can adopt and switch between multiple roles as needed, adapting to changing circumstances while maintaining well-defined boundaries.

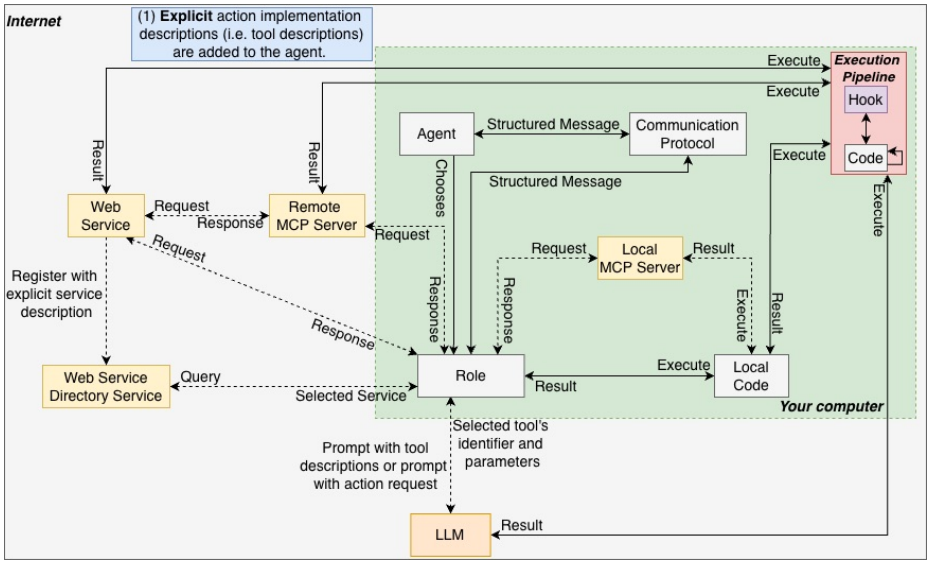

Fig. 1. An Hybrid Role-based MAS Reference Architecture. The four action types and the role-as-runtime-entity principle originate from [5] while this improved architecture introduces action lifecycle hooks ([B.E.]/[A.E.])

Based on this observation, we treat roles as *explicit*, *run-time* entities that bridge traditional AOSE principles with LLM capabilities [5]—a concrete architectural decision with practical implications for how systems are built and maintained. In our reference architecture (Figure 1), a role instance is a uniquely identified run-time entity that persists across invocations, is played by one concrete agent in a specific domain, maintains its own state, and can perform typed actions whose declared implementation type determines the execution strategy. Since different tasks require different strategies, we define four action implementation types and allow a role to mix them as required by each operation.

We make this design choice explicit: a role is a *computational* run-time entity, not merely a passive knowledge structure. A role is realized as an addressable object that an agent plays; it owns mutable state, receives and handles messages, and executes its typed actions, thereby actively processing data rather than only describing it. The motivation is an engineering one: treating roles as inspectable run-time objects—with identity, lifecycle, and queryable state—is what enables capability inspection, dynamic role adoption and switching, safety-constraint enforcement, and observability in deployed systems, properties that are lost when roles exist only as prompt text.

At the same time, the meta-model is deliberately layered so as not to over-constrain how roles are realized: it specifies *what* a role exposes (its identity, responsibilities, safety constraints, and typed-action interface) as declarative metadata, while leaving *how* this specification is mapped to a concrete run-time representation as a variation point. The same responsibilities and action specifi-

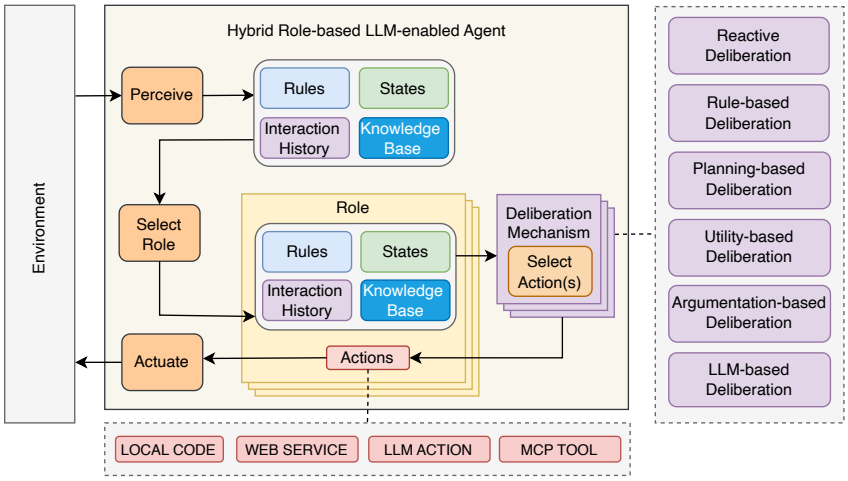

Fig. 2. A Hybrid Role-based LLM-enabled Agent Internal Architecture

cations can therefore be read as factual and procedural knowledge at design time, yet drive a computational entity at run time; the two views are complementary. Because the specification is metadata rather than a mandated class hierarchy (Section 5.2), alternative realizations—e.g., roles backed by composition or non-object representations—remain possible without changing the architecture.

The architecture adopts the definitions of four action types and the role-as-runtime-entity principle previously discussed in [5] and introduces *action lifecycle hooks* through `[B.E.]` (i.e. before execution) and `[A.E.]` (i.e. after execution) that execute before and after each action invocation. The `[B.E.]` hook enables parameter transformation, validation, and critically, *secure credential injection*. For example, when a *web service* action requires an API key, the `[B.E.]` hook can retrieve it from the environment variables and injects it into the request—the LLM never sees the credential, only the action's public interface (details are given in Section 5.1). This way, the reference architecture ensures that sensitive information such as API keys, tokens, and passwords remain isolated from the LLM context, addressing a significant security concern in LLM-enhanced systems. Similarly, the `[A.E.]` hooks support auditing, caching, and result transformation, enabling comprehensive action monitoring without polluting the core action logic.

Figure 2 illustrates the internal architecture of a hybrid role-based LLM-enabled agent. The agent follows the classic perceive–deliberate–act cycle. Both the agent and each role it plays encapsulate their own **memory**, and this agent-level and role-level memory informs the hybrid deliberation process [33,27,1,19].

We make memory an explicit, first-class component that both the agent and each role encapsulate: agent-level memory holds the agent's own state, while each role's memory holds the role's state, so that memory remains scoped by encapsu-

lation. We keep the internal structure of memory deliberately lightweight in this paper—each memory groups the entity's *state* with a *knowledgebase* (domain facts and knowledge) and an *interaction history* (past messages and actions), while the role's rules and typed actions remain first-class alongside it—and leave a finer-grained treatment of memory to future work. In the hotel reservation case study, a role's state corresponds to its `@State` fields (e.g., the current negotiation state), its knowledgebase to the hotel and weather data retrieved from external services, and its interaction history to the accumulated negotiation history and market analytics.

Communication is handled through the same perceive–act interface: receiving a message is treated as a form of perception and sending a message as a form of actuation, so the agent's input and output steps uniformly cover both environmental interaction and inter-agent messaging. Both options exist in AOSE—message handling can instead be modeled as a separate concern, as in Jason [7], architecture-based MAS design [38], and interaction-protocol enactment such as BSPL/Kiko [12]—but we adopt the uniform perceive/act model for simplicity. In the case study, incoming Contract Net Protocol messages (e.g., a call for proposals) are received during the perceive step and dispatched to the corresponding handler (e.g., `handleCFPMessage`), while outgoing messages (proposals, awards) are emitted during the act step.

Crucially, the deliberation mechanism is pluggable—it can employ reactive [29], rule-based [11], planning-based [4,13], utility-based, argumentation-based [26,36], or LLM-based [42] strategies depending on the application requirements—and is orthogonal to the action type, so any strategy can select any of the four typed actions. Once an action is selected, its execution is determined by its declared type, enforcing a clear separation between *what to do* (deliberation) and *how to do it* (typed execution).

These mechanisms are not bound to a particular role; rather, they operate over the structure each role exposes. A role supplies three elements that any deliberation strategy consumes: its *state* (the agent- and role-level fields that form the decision context), its *responsibilities* (duties together with must-always and must-never constraints that bound which actions are admissible), and its set of *typed actions* (the selectable action space). A rule-based mechanism thus matches role state against responsibility rules, a planning-based mechanism chains role actions toward a goal, a utility-based mechanism scores the admissible role actions, and an LLM-based mechanism reasons over the role's action descriptions and state—yet all of them read the same role-provided state and responsibilities and choose among the same role-declared actions. The role therefore defines *what can be decided*, the pluggable mechanism decides *which action* is selected, and the action's declared type determines *how* it executes.

In this paper, we focus on LLM-based deliberation and extend our previous work [5] by formalizing the related architectural elements into a prescriptive meta-model (Section 4).

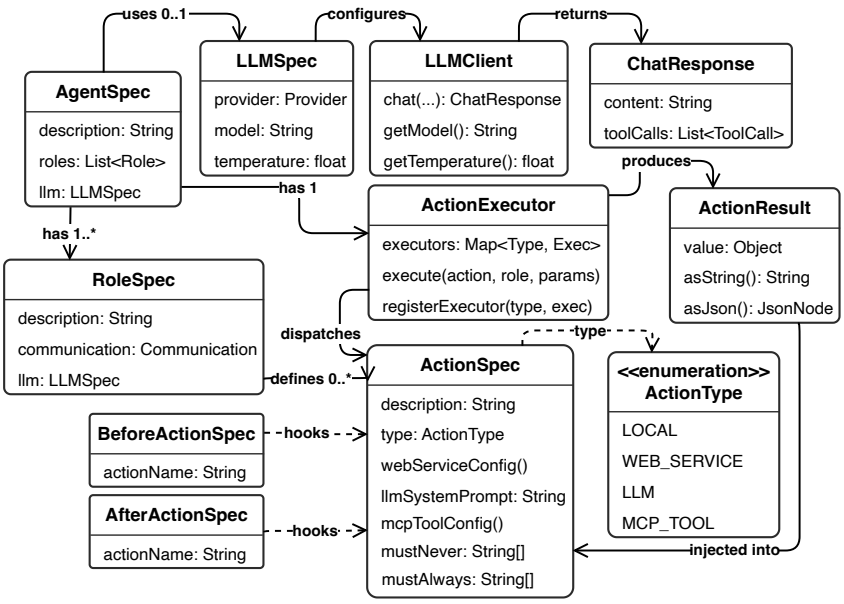

Fig. 3. Role-based Meta-Model for LLM-Enhanced Agents

4 A Meta-Model for Role-based Reference Architecture

To make the reference architecture prescriptive and actionable, we define a role meta-model for LLM-enhanced agents that specifies the core role constructs, their interfaces, constraints, and interaction relationships. Figure 3 presents this meta-model as a UML class diagram. The meta-model formalizes the core role constructs: the agent-level and role-level *rules* correspond to responsibilities with their duties, must-always invariants, and must-never safety constraints; *state* is captured through annotated fields that expose queryable runtime metadata; and *actions* are realized through typed action specifications whose declared type determines the execution strategy. The remainder of this section details the core constructs (Section 4.1), correctness and safety constraints (Section 4.2), and relationships and runtime mapping (Section 4.3).

4.1 Core constructs and interfaces

The meta-model centers on two complementary abstractions: the `AgentSpec`, which represents identity (the "who"), and the `RoleSpec`, which encapsulates capabilities (the "what"). An `AgentSpec` declares a natural-language `description` identifying the agent and references to one or more roles. Optionally, it may hold a reference to an `LLMSpec`, providing a default language model for all its roles. This separation ensures that the same agent can play multiple roles—potentially in different environments—while maintaining a single identity.

The `RoleSpec` is the central construct of the meta-model. It captures a role's identity through a natural-language `description` and may bind an optional `@LLMSpec` and a `@Communication` block that set the role's default language model and interaction style. A *Responsibility* is derived from each *ActionSpec* rather than declared separately: it takes the action's `name` and `description` and carries the *must-always* invariants and *must-never* safety constraints the action declares, attaching both positive capability specification and negative safety bounding directly to the action that realizes it. Each *ActionSpec* declares the action's description, its `ActionType` from the four-variant enumeration introduced in Section 3 (`LOCAL`, `WEB_SERVICE`, `LLM`, `MCP_TOOL`), and type-specific configuration for external calls (`WebServiceConfig` for `WEB_SERVICE` and `MCPToolConfig` for `MCP_TOOL`); the action's name, its parameters (`ParamSpec` entries capturing name, type, optionality, and default value), and its return type are discovered from the annotated method's signature rather than restated in the annotation. `LLM` actions instead carry an optional system prompt and temperature directly on the `ActionSpec`. The `ActionExecutor` routes each action to the appropriate typed executor, decoupling specification from execution strategy. The raw output of a typed executor is wrapped in an `ActionResult` and injected back into the action method whenever its signature declares such a parameter, so the method body can post-process the execution result—parsing an LLM response or an HTTP body, for instance—before returning; this is what lets a single action combine framework-orchestrated execution with developer-defined post-processing.

A `RoleSpec` may optionally override the agent-level `LLMSpec` with its own instance, enabling role-specific model selection. The `LLMSpec` is a declarative configuration (provider, model, temperature); from it the framework instantiates an `LLMClient` that provides a uniform contract: given a message, an optional system prompt, conversation history, and tool definitions, it returns a `ChatResponse` containing either text content or structured tool calls, insulating the role layer from provider-specific details. The meta-model intentionally leaves certain elements open for extension: `LLMSpec` can bind to any provider (e.g., OpenAI, Anthropic, Google), making the meta-model both prescriptive and adaptable.

4.2 Constraints and guardrails

A distinguishing feature of the meta-model is its integrated support for correctness and safety constraints at the action level.

Action lifecycle hooks. The meta-model formalizes the lifecycle hooks introduced in Section 3 as two named constructs: *BeforeActionSpec* and *AfterActionSpec*, each linked to a target action by name. A *BeforeActionSpec* receives and returns the action's parameter map, while an *AfterActionSpec* receives both the parameters and the result. This name-based binding keeps hooks decoupled from action implementations, enabling their independent evolution.

Safety properties. As noted in Section 4.1, actions declare *must-always* and *must-never* safety properties as string constraints on the action. These con-

straints are carried into the responsibilities derived from the actions and are thus queryable at runtime, enabling programmatic inspection of a role's declared safety bounds.

Execution pipeline. The `ActionExecutor` processes each action invocation through a defined pipeline: (1) BeforeActionSpec hook, (2) parameter validation and type conversion, (3) action execution via the typed executor, and (4) AfterActionSpec hook. This pipeline ensures that every action—regardless of its type—passes through the same validation and hook checkpoints, providing uniform guarantees across deterministic and non-deterministic actions alike.

4.3 Relationships and runtime mapping

The meta-model defines three categories of interaction relationships. An agent *plays* one or more roles, adopted dynamically at runtime: an agent may acquire new roles or deactivate existing ones as the system evolves, and the same agent can play different roles in different environments. Interaction protocols—such as the Contract Net Protocol (`CNP`) used in our case study—are realized as sequences of typed messages following a defined protocol flow. The meta-model delegates inter-agent messaging to the underlying framework; for instance, in SCOP the framework provides its own communication infrastructure for message routing and delivery.

A key design goal is *continuity*: every design-time artifact must have a corresponding runtime entity that faithfully represents it. This mapping is achieved through *reflective metadata discovery*: when a role is initialized, the framework scans its class definition, discovers action methods and their associated metadata, constructs `ActionSpec` objects with resolved parameter specifications and type-specific configurations, and registers them with the `ActionExecutor`. The practical consequence is *runtime inspectability*: unlike prompt-only approaches where role definitions are ephemeral text in an LLM context window, our meta-model produces persistent, queryable runtime entities supporting observability, debugging, and governance in production-grade LLM-enhanced MAS.

5 Java Realization via Annotations

The meta-model constructs defined in Section 4 are realized in Java through a set of declarative annotations. Rather than encoding agent structure in external configuration files or XML descriptors, developers annotate standard Java classes and methods; the framework extracts this metadata at runtime via reflection and constructs the corresponding runtime entities. This section presents the annotation set, demonstrates its use with concrete examples from the hotel reservation case study, and explains how an existing MAS framework adopts it.

Table 1. Annotation set and meta-model mapping

Meta-model	Annotation	Target	Key attributes
AgentSpec	`@AgentSpec`	Class	`description`, `llm`
RoleSpec	`@RoleSpec`	Class	`description`, `llm`, `communication`
ActionSpec	`@ActionSpec`	Method	`type`, `description`*
BeforeActionSpec	`@BeforeActionSpec`	Method	`value`
AfterActionSpec	`@AfterActionSpec`	Method	`value`
LLMSpec	`@LLMSpec`	Nested	`provider`, `model`, `temperature`
Communication	`@Communication`	Nested	`style`, `languages`
WebServiceConfig	`@WebService`	Nested	`endpoint`, `method`, `timeout`
MCPToolConfig	`@MCPTool`	Nested	`serverUrl`, `toolName`, `timeout`
State	`@State`	Field	`description`

*Plus `webService`, `llmSystemPrompt`, or `mcpTool` for the corresponding action types.

5.1 Annotation set and semantics

Table 1 summarizes the annotation set and its correspondence to the meta-model constructs introduced in Section 4. The annotations form a compositional hierarchy organized around three levels: agent, role, and action.

Agent level. `@AgentSpec` is applied to the class that represents an agent. It declares the agent's identity through a natural-language `description` and optionally embeds an `@LLMSpec` that serves as the default language model configuration for all roles the agent adopts. A role-level `@LLMSpec`, if present, overrides this default. Agents that require no LLM reasoning (e.g., pure data fetchers) may omit `@LLMSpec` entirely. Crucially, the annotation itself carries no dependency on any particular class hierarchy: `@AgentSpec` is pure metadata that the extraction layer reads via reflection. When a specific MAS framework is used, the annotated class *also* extends that framework's agent base class (e.g., `ai.scop.core.Agent` in the SCOP case study); the two concerns—declarative role metadata and framework-specific lifecycle—are independent and connected only through the thin bridge described in Section 5.2.

Role level. `@RoleSpec` annotates the class that represents a role and is the richest annotation in the hierarchy. As with `@AgentSpec`, the annotation is pure metadata, independent of any class hierarchy, although in our SCOP integration the annotated class extends `ai.scop.core.Role`. It composites three concerns into a single declaration: (i) a natural-language `description` of the role's purpose, (ii) an optional `@LLMSpec` that overrides the agent-level default when a role requires a different model, and (iii) a `@Communication` block that specifies tone (e.g., `PROFESSIONAL` vs. `FRIENDLY`), verbosity level, and supported natural languages. The role's responsibilities are not declared here; they are derived automatically from its `@ActionSpec` methods (see below), reflecting the Gaia notion of responsibilities while avoiding a hand-maintained action grouping. This composite structure ensures that the role's behavioral contract—what it does, how it reasons, and how it communicates—is captured in a single, inspectable site.

How responsibilities are operationalized. Responsibilities are not written by hand; the metadata extraction layer (Section 5.2) derives them automat-

ically from the role's `@ActionSpec` methods, producing one responsibility per LLM-exposed action. Each derived responsibility takes its readable name from the method, its `description` from the action, and the *must-always*/*must-never* safety constraints the action declares, and is read into the runtime `RoleSpec` so that at run time each obligation resolves to a concrete, inspectable typed action—without the developer maintaining a separate obligation-to-action mapping. In the case study, for example, the hotel provider role's responsibilities are derived from its action methods—`decidePricingStrategy`, `sendProposal`, `registerWithDF`, and the negotiation handlers, among others—each carrying the type and constraints of the action that realizes it. Responsibilities thus provide a queryable map from a role's obligations to the actions—and their declared types—that realize them, which the must-always/must-never safety properties (Section 4.2) can further bound.

Action level. `@ActionSpec` marks a method as an executable action and declares its `type`. The meta-model defines four action types: `LOCAL` for in-process business logic, `WEB_SERVICE` for external HTTP calls, `LLM` for tasks requiring natural language generation, subjective evaluation, or dynamic tool selection, and `MCP_TOOL` for Model Context Protocol server invocations. Each non-local type carries its own configuration: `@WebService` specifies endpoint URL, HTTP method, and timeout; an `LLM` action may set an `llmSystemPrompt` (and an optional `llmTemperature`) directly on `@ActionSpec`, while tool exposure is configured at the agent level; `@MCPTool` provides the MCP server URL, tool name, and timeout. This design lets the framework generate the entire call pipeline—HTTP client, LLM invocation, or MCP handshake—without developer-written boilerplate.

Lifecycle hooks. `@BeforeActionSpec` and `@AfterActionSpec` realize the meta-model's lifecycle hooks (Section 4.2) as Java annotations. Both receive an `ActionParams` map; `@BeforeActionSpec` returns the (possibly modified) map, while `@AfterActionSpec` performs post-execution side effects. Hooks are resolved by name: the annotation's `value` must match the target method name.

State. `@State` annotates fields within a role class as queryable metadata, letting the framework or external monitoring tools inspect a role's current state without exposing internal details.

5.2 How a Java-based MAS framework adopts it

The annotation set is designed to be *framework-agnostic*: it specifies *what* an agent, role, or action is, not how a particular framework should instantiate or execute it. A MAS framework adopts the annotations by providing a metadata extraction layer and a thin bridge that maps annotation metadata to the framework's own lifecycle.

Metadata extraction. The extraction layer centres on `ActionDiscovery.discoverActions(Class)`, which scans a role class for methods annotated with `@ActionSpec` and produces, for each, a resolved `ActionSpec` carrying a method reference, parameter specifications, action type, and type-specific configuration.

Role-level metadata—the role's `@RoleSpec` description, its `@LLMSpec` configuration, and its `@Communication` preferences—is read directly from the class's annotations via reflection. No XML files or external configuration are required: annotations serve as the single source of truth.

Action execution routing. The `ActionExecutor` described in Section 4.1 routes each action to the appropriate typed executor: `LOCAL` actions are invoked via reflection, `WEB_SERVICE` actions through an HTTP client, and so forth. The full execution pipeline—hooks, validation, dispatch, and post-processing—is transparent to the developer: action methods are ordinary Java methods whose lifecycle is managed entirely by the framework.

Framework adoption. Because the core extraction and execution components (`ActionDiscovery` and `ActionExecutor`) reside in a framework-agnostic core module, the only artefact a framework implementor must write is a *bridge* class in a dedicated integration module. The bridge reads role and action metadata at startup, resolves LLM configurations at run-time, and translates the framework's agent life-cycle (e.g., setup, role registration) into calls to the core extraction and execution layer. The same pattern applies to any Java-based MAS platform—SCOP, JADE [6] or Jason [7]. Section 6 demonstrates implementation for SCOP.

5.3 Security-by-construction patterns

The hook mechanism introduced in Section 5.1 enables a *security-by-construction* pattern: sensitive data can be injected into action parameters via `@BeforeActionSpec` hooks at runtime, ensuring that credentials and secrets never appear in LLM-visible contexts such as system prompts, tool descriptions, or conversation history. Consider a `WEB_SERVICE` action that requires an API key for authentication. Rather than hard-coding the key in the `@WebService` annotation or passing it through the LLM's tool-calling interface, a `@BeforeActionSpec` hook retrieves the credential from a secure source (e.g., an environment variable or a vault) and injects it into the `ActionParams` (Listing 1.6). Because the hook executes *outside* the LLM reasoning loop, the secret is available to the action executor but is never serialized into a prompt or tool response. In a production deployment, the same pattern would be used to inject database credentials, OAuth tokens, or encryption keys without exposing them to the language model.

6 Case Study: Hotel Reservation Scenario in SCOP

We consider a hotel reservation system where multiple hotel agents compete to serve customer agents.[8] This scenario demonstrates how the `Contract Net Protocol (CNP)`—a classic AOSE interaction pattern—can be integrated with hybrid action types to create a robust, LLM-enhanced MAS. The scenario is

[8] The complete implementation is available at `https://gitlab.com/scop-framework/model-library/negotiation-and-markets/scop-lib-nm-hotel-reservation-via-cnp`, last access on Jul. 7, 2026.

implemented using SCOP, an agent-based modelling and simulation framework following the Agent/Group/Role approach [16].

Step 1: Define agents and roles with hybrid actions.

The system involves two primary roles. The HotelRole manages reservations using `LOCAL` actions for availability checks (`canFulfillRequest`) and booking confirmation, `LLM` for competitive pricing decisions, and `MCP_TOOL` for calendar synchronization with external platforms. The CustomerRole seeks optimal deals using `LOCAL` to broadcast requests and validate budgets, `WEB_SERVICE` to retrieve weather forecasts, and `LLM` to dynamically select search strategies and evaluate competing offers. Each action is annotated with its implementation type via `@ActionSpec(type=...)`, making the execution strategy explicit and enabling the framework to route invocations to suitable executors.

Listing 1.1 shows how the hotel agent is declared with `@AgentSpec`, binding it to a specific LLM provider and model. Different agents can bind to different models: the customer agent uses `glm-5:cloud` with a higher temperature (0.7) for more creative negotiation, while the data-fetcher agent declares no `@LLMSpec` at all, since it performs only deterministic REST API calls.

```
@AgentSpec(
    description = "Hotel service provider agent that handles room reservations",
    llm = @LLMSpec(provider = Provider.OLLAMA, model = "minimax-m2.5:cloud", temperature =
          0.5f))
public class HotelAgent extends Agent { ... }
```

Listing 1.1. Agent declaration with `@AgentSpec` annotation.

Listing 1.2 illustrates the `@RoleSpec` for the hotel provider role, declaring a role-level language model—`kimi-k2.5:cloud`, which overrides the agent-level `minimax-m2.5:cloud` default of Listing 1.1—together with its communication preferences. The customer role likewise overrides its agent's `glm-5:cloud` default with `deepseek-v3.2:cloud`, illustrating per-role model selection. The `@Communication` block allows each role to declare a distinct interaction style—here the hotel provider adopts a `PROFESSIONAL` tone with `CONCISE` verbosity, guiding the model toward formal yet brief responses during negotiations. The role's responsibilities are not declared explicitly; they are derived automatically from its `@ActionSpec` methods.

```
@RoleSpec(
    description = "Hotel service provider that responds to reservation requests",
    llm = @LLMSpec(provider = Provider.OLLAMA, model = "kimi-k2.5:cloud", temperature = 0.5f)
          ,
    communication = @Communication(
        style = @Communication.Style(tone = Communication.Tone.PROFESSIONAL, verbosity =
              Communication.Verbosity.CONCISE),
        languages = {"tr", "en"}))
public class HotelProviderRole extends Role { ... }
```

Listing 1.2. Role declaration with LLM and communication preferences.

Listing 1.3 shows two concrete action implementations: a `LOCAL` action for processing incoming `CFP` messages and a `WEB_SERVICE` action for fetching hotel data from an external API.

```
@ActionSpec(type = ActionType.LOCAL,
            description = "Process incoming CFP and decide whether to make a proposal")
public void handleCFPMessage(Message<RoomQuery> msg) {
```

```
    RoomQuery q = msg.getPayload();
    if (!canFulfillRequest(q)) { sendRefusal(msg.getSender(), "Does not match"); return; }
    sendProposal(msg.getSender(), q);
}

@ActionSpec(type = ActionType.WEB_SERVICE,
    description = "Fetch all hotels from Hotel Data API",
    webService = @WebService(endpoint = "http://localhost:8000/api/hotels",
                             method = HttpMethod.GET, timeout = 5000))
public List<Hotel> fetchAllHotels() { ... }
```

Listing 1.3. `LOCAL` and `WEB_SERVICE` action declarations.

Step 2: SCOP framework integration.

Following the adoption pattern of Section 5.2, SCOP integration is realised through a single bridge class, `SCOPBridge`, that reads action metadata via `ActionDiscovery`, resolves LLM configurations from `@RoleSpec`/`@LLMSpec`, and dispatches actions through `ActionExecutor`. Agent classes extend `ai.scop.core.Agent`, role classes extend `ai.scop.core.Role`, and the agent's `setup()` method calls `adopt(role)` to register annotated roles with SCOP's tick-based execution model (Listing 1.4).

```
@AgentSpec(description = "Customer agent", ...)
public class CustomerAgent extends Agent {
    @Override
    protected void setup() {
        adopt(new CustomerRole(this, "HotelEnv", desiredLocation, desiredRank, maxPrice,
            desiredPrice, numberOfRooms, amenities, new LinearPricingStrategy()));
        adopt(new Conversation(this, getPlayground()));
    }
}
```

Listing 1.4. SCOP framework adoption: agent setup with role adoption.

Step 3: Contract Net Protocol execution.

The `CNP` orchestrates the negotiation through structured message exchange. The customer broadcasts a `CFP` specifying requirements: location, minimum star rating, budget, number of rooms, and required amenities. Each hotel agent receives the `CFP` and evaluates whether to participate. The `canFulfillRequest()` action (`LOCAL`) checks hard constraints—location match, room availability, amenity requirements. If constraints are met, the `decidePricingStrategy()` action (`LLM`) determines competitive pricing based on current occupancy, competitor landscape, and market conditions. Hotels respond with `PROPOSAL` (including price and terms) or `REFUSE` messages.

Step 4: LLM-enhanced decision making.

The customer gathers proposals until a timeout and calls `evaluateProposals()` (an `LLM` action that ranks offers considering price, star rating, amenities, and overall value). The LLM receives structured proposal data and returns a reasoned selection. If the LLM fails or returns an invalid response, a fallback `LOCAL` scoring function ensures the system remains operational. The customer sends `AWARD` to the winner and `REJECT` to others. The winning hotel then atomically updates its inventory via `LOCAL`.

Step 5: Life-cycle hooks for cross-cutting concerns.

The case study exercises life-cycle hooks for concerns such as dynamic pricing, seasonal adjustment, negotiation strategy, proposal logging, and anomaly detection. Listing 1.5 shows a `@BeforeActionSpec` hook that computes a dy-

namic price adjustment based on occupancy *before* the proposal is sent, and an `@AfterActionSpec` hook that accumulates market analytics *after* each proposal is received. All hooks are linked solely by action name strings without modifying core action logic.

```
@BeforeActionSpec("sendProposal")
private ActionParams beforeSendProposal(ActionParams p) {
    double multiplier = decidePricingStrategy(query, null); // LLM occupancy-aware pricing
    double dynamicPrice = basePrice * Math.max(0.85, Math.min(1.30, multiplier));
    p.set("dynamicPrice", dynamicPrice);
    p.set("demandLevel", multiplier < 1.0 ? "LOW" : multiplier < 1.10 ? "NORMAL" : "HIGH");
    return p;
}

@AfterActionSpec("handleProposalMessage")
private void afterHandleProposalMessage(ActionParams p) {
    double price = p.getDouble("proposalPrice");
    ttlPropCount++;
    proposalPriceSum += price;
    double avg = proposalPriceSum / ttlPropCount;
    if (ttlPropCount > 1 && price > avg * 1.5) getLogger().warn("PRICE_ANOMALY detected");
}
```

Listing 1.5. Lifecycle hooks for dynamic pricing and market analytics.

The hook mechanism also enables the *security-by-construction* pattern described in Section 5.3: Listing 1.6 shows credential injection via a `@BeforeActionSpec` hook that keeps secrets outside the LLM context.

```
@BeforeActionSpec("fetchWeather")
private ActionParams beforeFetchWeather(ActionParams p) {
    p.set("apiKey", EnvConfig.openWeatherApiKey());
    p.set("weatherBase", EnvConfig.weatherApiBase());
    return p;
}
```

Listing 1.6. Credential injection via `@BeforeActionSpec`.

This example demonstrates the architecture's key principles: deterministic code handles critical operations, LLM actions provide intelligent flexibility, `CNP` messages ensure semantic clarity, and lifecycle hooks enable cross-cutting concerns without polluting core action logic. The complete source code and runtime results of this case study are publicly available in the project repository.[9]

7 Discussion

Separating deliberation from execution prevents the LLM from controlling the full agent loop. Deliberation—deciding which action to perform and when—remains deterministic and role-managed via an explicit control flow whose transitions depend only on programmatic conditions, ensuring predictable control flow. The LLM is invoked only within explicitly typed `LLM` action implementations to determine how a selected action is carried out, while `LOCAL` fallbacks preserve liveness when the LLM fails. This separation improves auditability through fully loggable state transitions, contains failures by localizing

[9] `https://gitlab.com/scop-framework/model-library/negotiation-and-markets/scop-lib-nm-hotel-reservation-via-cnp`, last access on Jul. 7, 2026.

Table 2. Action type selection guidelines.

Action Type	Select when...	Avoid when...
`LOCAL`	Output must be deterministic and reproducible; operation is safety-critical or performance-sensitive (e.g., inventory update, constraint check)	Task requires natural language understanding or creative reasoning
`WEB_SERVICE`	Operation is a well-defined HTTP API call with a stable contract (e.g., data retrieval, third-party integration)	Response format varies unpredictably or requires interpretation
`LLM`	Task requires natural language generation, subjective evaluation, open-ended reasoning, or dynamic tool selection based on context (e.g., ranking proposals, composing messages, selecting search strategies)	Output must be bit-exact or operation is latency-critical; tool selection is static and can be determined programmatically
`MCP_TOOL`	External tool ecosystem must be accessed through a standardized protocol; tools are managed independently of the agent	Tool is internal to the system and can be called directly

LLM issues to individual actions, and strengthens testability by enabling conventional unit tests for deliberation logic while isolating probabilistic evaluation to LLM-dependent actions.

Choosing among the hybrid action types is a design decision, and the boundaries between the action types involve trade-offs across determinism vs. flexibility, latency vs. adaptability, and testability vs. expressiveness. Our implementation experience suggests treating action selection as a *determinism-first* heuristic: prefer `LOCAL` whenever an operation can be specified algorithmically—especially for safety-critical, performance-sensitive, or state-mutating steps—because *predictable* behavior simplifies testing, debugging, and compliance. `LLM` actions become appropriate when the task inherently requires natural language generation, subjective judgment, open-ended reasoning, or dynamic tool selection (e.g., evaluating competing options, selecting search strategies) (Table 2).

Observability and debugging are improved due to distinct explicit action types: when a failure occurs, the action type immediately indicates the likely source and its characteristic signature (e.g., exceptions for `LOCAL`, HTTP/-timeout signals for `WEB_SERVICE`, malformed outputs or tool-resolution issues for `LLM`, and protocol/connection errors for `MCP_TOOL`), enabling the executor to select appropriate retry, fallback, or escalation strategies without relying on ad-hoc logs or parsing free-text outputs. The annotation-driven meta-model further strengthens debugging with externally inspectable `@State` snapshots, structured pre/post-action audit trails via `@BeforeActionSpec` / `@AfterActionSpec`, and deterministic state-machine transitions that support traceability, replay, and dashboard-style monitoring from framework metadata.

Lifecycle hooks currently bind to actions using method-name strings rather than typed references, since Java annotations cannot use method references as attribute values (see Listing 1.5 line 1 and line 10). As a result, renaming an action without updating the hook's `value` silently breaks the binding. Hooks are resolved lazily at execution time; if no matching `@ActionSpec` method is found, the hook is skipped, avoiding crashes but potentially hiding refactoring errors. This fragility could be reduced through eager validation during role initialization, compile-time annotation processing, or build-integrated static anal-

ysis. Thus, adopters should weigh this design's decoupling benefits against its refactoring risks.

8 Conclusion and Future Work

This paper extends our earlier role-based perspective [5] into actionable engineering guidance for LLM-enhanced multi-agent systems, making *how* an action executes a first-class design decision rather than treating all behavior as prompt-driven inference. Explicitly typing actions (`LOCAL`, `WEB_SERVICE`, `LLM`, `MCP_TOOL`) and separating typed execution from LLM-driven execution improves maintainability, observability, and safety: failures become classifiable before invocation, role state becomes inspectable, and behavior is auditable through structured pre/post hooks and explicit state transitions. Overall, the architecture offers a pragmatic middle ground that preserves LLM flexibility while restoring the engineering discipline needed for long-lived deployments.

We see four main directions to strengthen and validate the architecture. We will expand empirical evaluation through multi-domain case studies, including negotiation and task allocation with the Contract Net Protocol, decentralized coordination patterns, and hierarchical task decomposition. We will also benchmark against frameworks such as CrewAI and the Microsoft Agent Framework, focusing on reliability, observability, and maintainability trade-offs. In addition, we plan to extend the "typed execution + inspectable state" approach beyond roles to additional AOSE concepts—environments, plans, and organizations—to support organizational structure, governance, and multi-role reasoning.

Long-running workflows also require systematic context management, since persistent role state must coexist with bounded LLM context windows. This motivates principled summarization, retrieval, and memory policies. We will further deepen the security and safety analysis through threat modeling for prompt injection, tool misuse, and cross-role data leakage, while investigating enforcement mechanisms such as capability-based tool access, sandboxing for local code, and policy checks at action pre/post boundaries. Together, these steps aim to move from a reference architecture toward repeatable engineering methods and measurable guarantees for production MAS.

Acknowledgments. This study was supported by Scientific and Technological Research Council of Türkiye (TUBITAK) under the Grant Number 125E975. The authors thank to TUBITAK for their supports.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aaron, E., Admoni, H.: Action selection and task sequence learning for hybrid dynamical cognitive agents. Robotics and Autonomous Systems **58**(9), 1049 – 1056 (2010). https://doi.org/10.1016/j.robot.2010.05.006
2. Aguzzi, G., Ciatto, G., Ferrando, A., Gatti, A., Mascardi, V.: LLMs as agents, LLMs at the service of agents, or agents at the service of LLMs? In: Proceedings of the 26th International Workshop on Objects to Agents (WOA) (2025)
3. Al Owayyed, M., Denga, A., Brinkman, W.P.: Controlled yet natural: A hybrid bdi-llm conversational agent for child helpline training. In: Proceedings of the 25th ACM International Conference on Intelligent Virtual Agents. pp. 1–10 (2025)
4. Alford, R., Shivashankar, V., Roberts, M., Frank, J., Aha, D.W.: Hierarchical planning: Relating task and goal decomposition with task sharing. vol. 2016-January, p. 3022 – 3028 (2016)
5. Asici, T.Z., Gürcan, Ö., Kardas, G.: Towards engineering llm-enhanced multi-agent systems: A critical examination of roles. In: Rodriguez, S., Feng, L., Müller, J.P. (eds.) Proceedings of the 13th International Workshop on Engineering Multi-Agent Systems (EMAS 2025), held in conjunction with the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), May 19–20, 2025, Detroit, Michigan, USA. Lecture Notes in Artificial Intelligence, vol. 16407, pp. 200–220. Springer Nature Switzerland, Cham (2026). https://doi.org/10.1007/978-3-032-18011-7_12
6. Bellifemine, F., Poggi, A., Rimassa, G.: Jade–a fipa-compliant agent framework. In: Proceedings of PAAM. vol. 99, p. 33. London (1999)
7. Bordini, R.H., Hübner, J.F., Wooldridge, M.J., Wooldridge, M.J.: Programming multi-agent systems in AgentSpeak using Jason, vol. 8. Wiley Online Library (2007)
8. Borghoff, U.M., Bottoni, P., Pareschi, R.: Beyond prompt chaining: The tb-cspn architecture for agentic ai. Future Internet **17**(8), 363 (2025)
9. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems **8**, 203–236 (2004)
10. Chatterjee, A., Kowndinya Renduchintala, H., Bhatia, S., Chakraborty, T.: Posix: A prompt sensitivity index for large language models. p. 14550 – 14565 (2024). https://doi.org/10.18653/v1/2024.findings-emnlp.852
11. Chien, S.A., Gervasio, M.T., DeJong, G.F.: On becoming decreasingly reactive: Learning to deliberate minimally. p. 288 – 292 (1991). https://doi.org/10.1016/B978-1-55860-200-7.50060-X
12. Christie, S.H., Singh, M.P., Chopra, A.K.: Kiko: Programming agents to enact interaction models. In: Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 1154–1163. IFAAMAS (2023)
13. Debenham, J.: Information-based planning and strategies. IFIP International Federation for Information Processing **276**, 45 – 54 (2008). https://doi.org/10.1007/978-0-387-09695-7_5
14. Dignum, V.: A Model for Organizational Interaction: Based on Agents, Founded in Logic. Ph.D. thesis, Utrecht University (2004)
15. Esteva, M., de la Cruz, D., Sierra, C.: ISLANDER: An electronic institutions editor. In: Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 1045–1052 (2002). https://doi.org/10.1145/545056.545069

16. Ferber, J., Michel, F., Báez, J.: Agre: Integrating environments with organizations. In: Int. Work. on Env. for Multi-Agent Systems. pp. 48–56. Springer (2004)
17. Frering, L., Steinbauer-Wagner, G., Holzinger, A.: Integrating belief-desire-intention agents with large language models for reliable human–robot interaction and explainable artificial intelligence. Engineering Applications of Artificial Intelligence **141**, 109771 (2025). https://doi.org/10.1016/j.engappai.2024.109771
18. Gatti, A., Mascardi, V., Ferrando, A.: Let me talk to you! Natural language interaction between humans and BDI agents via ChatBDI. In: Proceedings of the 28th European Conference on Artificial Intelligence (ECAI) (2025). https://doi.org/10.3233/FAIA251242
19. Guessoum, Z.: Hybrid agent model: A reactive and cognitive behavior. p. 25 – 32 (1997)
20. Guo, T., et al.: Large language model based multi-agents: A survey of progress and challenges. pp. 8048–8057 (2024)
21. Hajjarzadeh, S., Shakeri Hossein Abad, Z., Far, B.: Llm-based mase, a software development framework for developing multi-agent systems. Lecture Notes in Computer Science **15706 LNAI**, 432 – 443 (2026). https://doi.org/10.1007/978-981-96-8889-0_37
22. Hewing, M., Leinhos, V.: Human-guided ai: Designing prompts in llm for effective human-computer collaboration. Lecture Notes in Computer Science **16345 LNCS**, 23 – 36 (2026). https://doi.org/10.1007/978-3-032-13184-3_2
23. Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S.K.S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., Schmidhuber, J.: MetaGPT: Meta programming for a multi-agent collaborative framework. In: Proceedings of the Twelfth International Conference on Learning Representations (ICLR) (2024)
24. Hübner, J.F., Sichman, J.S., Boissier, O.: Developing organised multi-agent systems using the $\mathcal{M}$oise$^+$ model: Programming issues at the system and agent levels. International Journal of Agent-Oriented Software Engineering **1**(3/4), 370–395 (2007). https://doi.org/10.1504/IJAOSE.2007.016266
25. Ichida, A.Y., Meneguzzi, F., Cardoso, R.C.: BDI agents in natural language environments. In: Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 880–888. IFAAMAS (2024)
26. Kok, E.M., Meyer, J.J.C., Prakken, H., Vreeswijk, G.A.W.: A formal argumentation framework for deliberation dialogues. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **6614 LNAI**, 31 – 48 (2011). https://doi.org/10.1007/978-3-642-21940-5_3
27. Lloyd-Kelly, M., Gobet, F., Lane, P.C.R.: The art of balance: Problem-solving vs. pattern-recognition. vol. 2, p. 131 – 142 (2015). https://doi.org/10.5220/0005215901310142
28. Lorini, E.: Designing artificial reasoners for communication. In: Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems. p. 2690–2695. AAMAS '24, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2024)
29. Navarro, M., Heras, S., Julián, V.: Guidelines to apply cbr in real-time multi-agent systems. Journal of Physical Agents **3**(3), 39 – 47 (2009). https://doi.org/10.14198/jopha.2009.3.3.07
30. Ouyang, S., Zhang, J.M., Harman, M., Wang, M.: An empirical study of the non-determinism of chatgpt in code generation. ACM Transactions on Software Engineering and Methodology **34**(2) (2025). https://doi.org/10.1145/3697010

31. Owotogbe, J.: Assessing and enhancing the robustness of llm-based multi-agent systems through chaos engineering. p. 250 – 252 (2025). https://doi.org/10.1109/CAIN66642.2025.00039
32. Padgham, L., Winikoff, M.: Prometheus: A practical agent-oriented methodology. In: Agent-oriented methodologies, pp. 107–135. IGI Global (2005)
33. Palanca, J., Rincon, J.A., Carrascosa, C., Julian, V.J., Terrasa, A.: Flexible agent architecture: Mixing reactive and deliberative behaviors in spade. Electronics (Switzerland) **12**(3) (2023). https://doi.org/10.3390/electronics12030659
34. Pavón, J., Gómez-Sanz, J.J., Fuentes, R.: The ingenias methodology and tools. In: Agent-oriented methodologies, pp. 236–276. IGI Global (2005)
35. Schulz, T., Jander, K.: Dynamic plan generation with LLMs: Automatic execution of abstract BDI-agent goals. International Journal of Parallel, Emergent and Distributed Systems (2025). https://doi.org/10.1080/17445760.2025.2541956
36. Tang, Y., Parsons, S.: Argumentation-based dialogues for deliberation. p. 683 – 690 (2005)
37. Thakur, J.P., Moharir, A.K.: Trustworthy design patterns for multi-agent software systems. Communications in Computer and Information Science **2720 CCIS**, 381 – 392 (2026). https://doi.org/10.1007/978-3-032-08649-5_24
38. Weyns, D.: Architecture-Based Design of Multi-Agent Systems. Springer (2010). https://doi.org/10.1007/978-3-642-01064-4
39. Wooldridge, M., Jennings, N.R., Kinny, D.: The gaia methodology for agent-oriented analysis and design. Autonomous Agents and multi-agent systems **3**, 285–312 (2000)
40. Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., Wang, C.: AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv preprint arXiv:2308.08155 (2023)
41. Xu, A., Du, M., Yu, T., Puvvadi, M., Yu, T., Guo, Y., Chen, X., Gottschlich, J.G.: Agentic ai for enterprise: Emerging applications and real-world challenges. vol. 2, p. 6300 – 6301 (2025). https://doi.org/10.1145/3711896.3737871
42. Yu, J., Ding, Y., Sato, H.: Dyntaskmas: A dynamic task graph-driven framework for asynchronous and parallel llm-based multi-agent systems. vol. 35, p. 288 – 296 (2025). https://doi.org/10.1609/icaps.v35i1.36130