

Towards Agentic AI support for Hypermedia MAS Models of Smart Environments

Alexandru Sorici^{1,*}, Andrei Olaru¹ and Adina Magda Florea¹

¹Department of Computer Science and Engineering, National University of Science and Technology POLITEHNICA Bucharest, 313 Splaiul Independentei, 060042 Bucharest, Romania

Abstract

Goal-driven interaction in Smart Environments brings significant challenges due to the variety of devices, the heterogeneity or lack of formal specifications for affordances, and the complexity of planning. Recent advances in hypermedia-based MAS and reasoning-capable foundational machine learning models are making goal-driven interaction frameworks realistically plausible.

We demonstrate a prototype framework for the management of goal-driven smart environment interactions that builds upon: (i) Hypermedia MAS environment modeling, (ii) resource signification to capture usage experience, (iii) LLM support for agent reasoning, and (iv) the formation of communities in which agents can leverage the experience of peers to better achieve their own goals. We showcase framework functionality in a smart office simulated scenario.

Keywords

Ambient Intelligence, Hypermedia MAS, Web-of-Things, AI Agents, Context and Signification, Experience Sharing

1. Introduction

Goal-driven interactions in smart environments (notably residential homes, office buildings, tourist housing) have been an early vision in Ambient Intelligence (AmI) [1]. The topic remained largely out of practical reach, due mainly to challenges such as a large diversity of device and service capabilities, as well as the insufficient reasoning capabilities of systems intended to bridge these capabilities together towards a user goal.

However, the topic is receiving renewed interest from both academia and industry, thanks to recent developments in AI foundational models (notably Large Language Models - LLMs). LLMs have demonstrated emergent capabilities for intent understanding, planning (in complex domains requiring neuro-symbolic approaches [2], as well as simpler ones related to activities of daily living [3]) and matching of natural language actions to APIs [4, 5]. Furthermore, efforts within the W3C Web Agents Community Group [6] have surfaced Hypermedia Multi-Agent Systems (HMAS) as a design paradigm in which the agent environment and agent capabilities obtain a *representation* over the web. This *digital twin* style of modeling of agent environments is particularly suitable in the design of smart environment applications, which involve sensing from and interaction with physical devices.

For the smart environment application space specifically, we introduce an *architectural template* (which we call *AmI HMAS*) that uses a HMAS-based design, to support development of *goal-driven interaction applications*. Our proposal is based on the following principles:

1. The use of LLMs to (i) *plan* from high-level, under-specified user goals to a set of abstract actions that can be instantiated in a smart environment, and (ii) map from abstract actions to concrete API calls of sensors and devices in a smart environment.
2. The use of Signifiers [7] as *units of record* for the *experience* of an agent interacting in a smart environment
3. The design and use of *experience-based agent communities*, which serve as a space for sharing of individual affordance usage, as well as partial or complete plans that address a particular goal

The Second International Workshop on Hypermedia Multi-Agent Systems (HyperAgents 2025), in conjunction with the 28th European Conference on Artificial Intelligence (ECAI 2025); October 26, 2025, Bologna, Italy

*Corresponding author.

✉ alexandru.sorici@upb.ro (A. Sorici); andrei.olaru@upb.ro (A. Olaru); adina.florea@upb.ro (A. M. Florea)



© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Our contributions in this work are: (i) to describe the *design template* for Hypermedia MAS in support of smart environments, where the HMAS structure follows the Agents&Artifacts (A&A) paradigm [8], (ii) to introduce the motivation for and proposed functionality of Agent Communities for smart environment applications, and (iii) the design and implementation of a prototype system that follows our template.

2. Background and Related Work

In the domain of smart environments (smart homes, in particular) there are already commercial platforms (e.g. If-This-Than-That (IFTTT) [9], HomeAssistant [10]) that provide functionality by which affordances of several devices can be combined into more complex functionality (e.g. if-else rules in IFTTT, *automations* in HomeAssistant). Both services currently integrate AI chatbots (e.g. ChatGPT) either as an input-output service (IFTTT), or as a means to trigger actions and planned automations through natural language (HomeAssistant). However, neither service currently supports AI-aided *creation* of automated interactions. Sasha [11] looks at using LLMs to control smart home devices handling common automation routines (e.g. as found in typical IFTTT recipes [9]). The work finds that LLM generated plans can fail when the user request is poorly- or under-specified (e.g. “make it cozy in here”), leading to the need of an iterative reasoning process that encompasses clarifying, filtering, planning, and feedback steps. LLMind [12] goes a step further, presenting an LLM-based task-oriented AI agent framework, enabling collaboration between IoT devices, based on human high-level verbal instructions. Functionalities include the maintenance of a context of available device information and API descriptions, as well as an *experience accumulation* mechanism that classifies and locally stores LLM generated control scripts for specific user commands to enhance response speed and overall efficiency.

The HMAS-based design template we are proposing aligns with the design rationale of LLMind, providing modules that break down the user intent into subtasks, dependent on known affordances in the user environment, their use in prior requests and their current state. Unlike LLMind however, we consider that *examples of experience of use* can come from other interaction histories (see Agent Communities in Section 4), apart from those of one’s own agent. We consider this assumption to particularly hold true for the smart environment domain, since many of the *types* of devices or their placement in homes and offices are similar from one case to another, while also serving similar *interaction goals*. By proposing LLM agents, we seek to take advantage of the proven ability of LLMs to create abstract plans [3, 2] and to convert them to concrete API calls [4, 13]. Our main contribution lies in designing a mechanism by which we *obtain, structure and represent* the examples which help to ground and guide the reasoning performed by an LLM agent.

Based on existing work, we consider that the base unit of recording *experience of use* is an affordance *signifier* [7]. The latter has been introduced as means to bind a specific resource affordance in an environment (e.g. toggling light on/off) with the *intent* (e.g. decrease luminosity in a room) and *ability* (e.g. have LLM reasoning support, have a BDI reasoning cycle) of an agent, as well as the *recommended context* of execution (e.g. there are no people left in the room). Several affordances can help achieve a goal, and the same affordance can be signified differently to individual agents, depending on their abilities and context. For this reason, a method to *filter* relevant signifiers from a possibly large number is required. Further, if a goal requires more than a single affordance to be achieved, the abstract plan composed of a set of signaled affordances needs to be grounded into concrete actions. Previous work defines Signifier exposure [14] and resolution [15] mechanism for these purposes. While generic in the description of the mechanisms, prior work orients them more closely to BDI agents, e.g. proposing that recommended context can be described in terms of agent beliefs.

For the case of smart environments in particular, the proposed Aml HMAS design template promotes an explicit representation of interaction context and an AI agent powered signifier exposure and resolution mechanism, which leverages LLMs to understand similarity of goals (expressed as structured natural language), determine context compliance and map abstract plans to concrete affordance invocations.

3. Aml HMAS: design and functionality

Working scenario. A smart research lab has sensors that measure the interior temperature, the light intensity inside and outside the room, and the number of people in the room. The lights can be turned on or off and set to a percentage level of intensity, while the blinds can be raised or lowered to a percentage degree (100% means fully raised). We describe two different situations in both of which a user entering the lab perceives the room as too dark via the comment “It’s dark in here”. In a first setup, the lights are off, the blinds are drawn, and outside luminosity is low (it is late in the evening). With no prior usage experience, the system plans a solution that turns the lights on to full. In the second setup, the lights are dim, the blinds are half raised, it is sunny outside, while the inside temperature is below 23 degrees. In this case, signification exists for both light usage *and* raising the blinds. Given environment conditions, the solution from prior experience retrieved is to raise the blinds to 75% open. This simple scenario requires the Aml HMAS system to: (i) understand the user intent from the context of existing capabilities that can address the intent, (ii) monitor artifacts that provide context about the environment, (iii) examine previous interaction records for similar past requests, (iv) plan a set of artifact action affordance invocations that address the user goal, (v) obtain feedback from the user before executing the plan.

Affordance and Usage Experience Representation. To represent affordances we adopt a ThingDescription (TD) [16] model of resources (sensors, devices, services) available in a smart environment. Listing 1 shows an extract of the TD based representation of a light sensor with a property affordance called *luminosity*. To bind action and property affordances as actions or conditions in plans that achieve

```
@prefix ex: <http://example.com/ns#>
@prefix consert: <http://pervasive.semanticweb.org/ont/2017/07/consert/core>
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix ...
<http://localhost:8080/workspaces/lab308/artifacts/LightSensor308/#artifact>
  a td:Thing, hmas:Artifact, ex:LightSensor ;
  td:title "LightSensor308" ;
  td:hasPropertyAffordance [
    a td:PropertyAffordance, ex:MeasurementProperty ;
    td:name "luminosity" ;
    td:hasForm [
      htv:methodName "GET" ;
      hctl:hasTarget <http://localhost:8080/workspaces/lab308/artifacts/LightSensor308/luminosity> ;
      hctl:forContentType "application/json" ;
      hctl:hasOperationType td:readProperty ;
    ] ;
    td:hasOutputSchema [
      a js:ObjectSchema ;
      js:properties [
        a js:NumberSchema ;
        js:propertyName "luminosity"
      ] ;
      js:required ("luminosity") ;
      js:additionalProperties false
    ] ;
  ] .
```

Listing 1: Extract from the ThingDescription of the light sensor artifact in the demonstrator scenario.

user goals, we use a signifier [15] representation as that exemplified in Listing 2. It shows a signifier representation for the action affordance of setting the light intensity in the lab, if the intent is to increase luminosity and the context (expressed using SHACL constraints) is that of there being more than one person in the room and the measured light intensity being too low.

LLM-supported agents reason about signified affordances using the CASHMERE Ontology [17], which exposes a vocabulary linking text-based intentions with a concrete listing of *context* conditions. Structured language is used for specifying intentions, meaning that natural language statements will follow a schema of representation (in our example a JSON format). Context conditions are explicitly given as a list of SHACL Node shapes that apply to a HMAS resource (e.g. the internal light sensor, or the person counter artifacts in Listing 2). In our prototype, context conditions are restricted to numerical or string-based operations (e.g. equalities and inequalities, substring matching). A structured language format is favorable to LLM models trained for code generation or instruction following, while also facilitating automated processing (e.g. of the context conditions) apart from natural language interpretation.

```

@prefix ex: <http://example.com/ns#>
@prefix cashmere: <https://aimas.cs.pub.ro/ont/cashmere#> .
@prefix ...
<#turn-light-on-signifier> a cashmere:Signifier ;
  cashmere:signifies <#adjust-light-action-affordance> ; // #adjust-light-action-affordance is the TD ActionAffordance to
  set the light to a certain intensity
  cashmere:hasIntentionDescription [
    a cashmere:IntentionDescription ;
    cashmere:hasStructuredDescription "{ 'intent': 'increase luminosity in a room', }"^^xsd:string ;
  ] ;
  cashmere:recommendsContext [
    a cashmere:IntentContext ;
    cashmere:hasShaclCondition [
      a sh:NodeShape ;
      sh:targetNode <http://example.org/precis/workspaces/lab308/artifacts/internal_light_sensing308> ;
      sh:targetClass <http://example.org/LightSensor> ;
      sh:property [
        sh:path <http://example.org/LightSensor#hasLuminosityLevel> ;
        sh:dataType xsd:integer ;
        sh:maxInclusive 100 ;
      ] ;
    ] ;
    cashmere:hasShaclCondition [
      a sh:NodeShape ;
      sh:targetNode <http://example.org/precis/workspaces/lab308/artifacts/person_counter308> ;
      sh:targetClass <http://example.org/PersonCounter> ;
      sh:property [
        sh:path <http://example.org/PersonCounter#hasPersonCount> ;
        sh:dataType xsd:integer ;
        sh:minInclusive 1 ;
      ] ;
    ] ;
  ] ] .

```

Listing 2: Signifier that indicates use of the light308 controller artifact to increase luminosity in the research lab, when there is a person in the lab and the measured internal light intensity is below 100 lux.

Aml HMAS: agent roles. Our proposed design template for goal-driven interaction in HMAS modeled smart environments envisions an AI agent support divided into three main roles, which are shown in Figure 1. The *UserAssistant* agent provides an entry-point for input of user requests, as well as human-in-the-loop feedback for the plans that solve the request. The *UserAssistant* can also create resources in HMAS workspace in which it is present, exposing user preferences as property affordances to be used during request solving.

The *EnvExplorer* and *InteractionSolver* are the main functionality providers, which includes the attributions of signifier exposure and resolution, as discussed in Section 2. These AI agent roles operate under the assumption of an Agents & Artifacts HMAS and are deployed to service interaction with resources modeled as *Artifacts* in a specified *Workspace*.

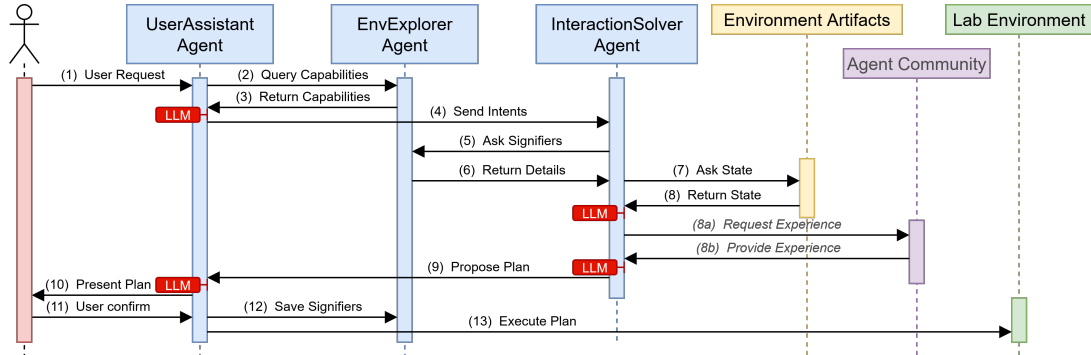


Figure 1: Flow of interactions between agent roles in the Aml HMAS system. Instances are highlighted in which LLMs are used to process queries, generate responses, or to build plans.

Thus, the role of the *EnvExplorer* agent is to catalog all the *affordances* of artifacts available in an HMAS workspace (including sub-workspaces). The agent also maintains a record of past *usage experience* of an affordance, using Signifier descriptions similar to the one in Listing 2.

The *InteractionSolver* agent is the main reasoning component, creating the plan of *action affordance* usage to solve the request, down to explicit invocation of the artifact API. The reasoning is supported by past usage experiences from the *EnvExplorer* and the current environment state. The agent prompt for the current implementation is given in Appendix A and features tool calls for retrieving sets of affordances that are helpful with respect to the current user goal, as well as the environment state. The internal reasoning loop that makes use of this tool calls references an explicit *pre-step* of checking *prior experience*. It is in this pre-step that our vision of experience-based communities (see Section

4) plays a role in the extension of current functionality, by allowing for retrieval of other sources of experience apart from that maintained within the own workspace. The prior experience is inserted as few-shot examples in the generation context of the LLM call. We posit that storing and leveraging past experiences is beneficial in the long run, since it is faster and cheaper to evaluate intent and context matching, than to reason about how to solve a request every time.

The typical flow of interactions between these agent roles is described in Figure 1, where red labels indicate where LLM reasoning calls are made in the agent functionality. The flow starts with a user request (step 1 in our working scenario: “It’s too dark in here”). To determine if the request can be reasonably handled by AmI HMAS, the *UserAssistant* queries the affordances in the environment, asking the *EnvExplorer* agent, and decides if the user query is within scope (steps 2 and 3). If within scope, the *UserAssistant* formulates a *user intent* and sends it to the *InteractionSolver* (step 4). The *InteractionSolver* determines if the user intent can be solved based on past usage (recorded as signifiers – steps 5 and 6) and the state of the implied artifacts (steps 7 and 8). The proposed plan (step 9) is obtained based on existing or adapted plans (from prior experience), or a new planning altogether based on signified affordances (if no prior experience for the intent exists). When creating new plans, artifact descriptions are inserted into the LLM generation context of the *InteractionSolver*, based on its communication with the *EnvExplorer*. The plan (at the current stage only consisting of a *sequence* of one or more action affordance invocations on deployed artifacts) is presented to the user in a human readable form by the *UserAssistant* (step 10). A confirmed plan (step 11) leads to its execution (step 13) and the recording of the usage experience as a whole plan. Individual signifiers are created by relating each affordance used in the plan to the original intent expressed in the request and to the current context (step 12). A user can inquire about the environment state (steps 14 - 17).

4. Experience-Based Agent Communities

The solution to a user request in a given environment context is a *plan* based on conditions set on property affordances and invocations of action affordances. Figure 1 shows this as the result of step 9, whereby the generation context is obtained using signified affordances for the user request and environment state property descriptions. We propose a method to increase the success rate of the planning step by considering input from *experience-based* agent communities.

Communities are motivated by the assumption that in an large HMAS-based smart environments, multiple workspaces (e.g. per building, or per room) will share device capabilities, typical user requests and environment context. We define a *community* as a *group* of *InteractionSolver* agents that have a common *interaction profile*. The interaction profile of community can be: (i) Manually defined by a developer (e.g. all agents in the building containing Lab 308), or (ii) defined semi-automatically based on set intersections of signified affordances. Specifically, community membership is based on a common set of affordances (e.g. toggling of lights, raising/lowering of blinds), as well as user specified properties, such as: values of specific property affordances in the smart environment (e.g. size of the space, positioning of the space in the building), or preferences and goals of agents in the environment (e.g. preferences for ambient temperature between 21 and 25 degrees).

Forming Communities. A *community* is created as an artifact at the top-level workspace of the HMAS model for a smart environment (e.g. the building that contains Lab308 in the scenario). If predefined by a developer, creation happens at deployment time; otherwise a community is created by an *InteractionSolver* agent based on the set of available action affordances (discovered dynamically in its workspace) and other properties configured by a developer.

Created communities are registered in a *communities directory* which is an indexing resource existing at the top-level workspace of a smart environment. Any *InteractionSolver* can use the *communities directory* to perform a search for community profiles that match a given set of properties. Since we expect that affordance description vocabularies might vary between workspaces of large smart environments (e.g. campus of a university, all floors in a multi-tenant office building), the profile matching will also be supported by an LLM reasoning procedure to provide vocabulary alignment functionality. An

InteractionSolver agent can join only those communities for which the profile match procedure identifies that its own affordances and properties are a subset of the ones in the community profile. Developers can enable or disable joining of communities by configuring their *InteractionSolver* agents.

Using a Community. Within a community, member *InteractionSolver* agents may share: (i) individual signifiers (i.e. use of specific affordance for a goal), (ii) abstract plans (sequence of ungrounded affordances), and (iii) fully grounded plans (all affordance references are mapped to concrete API invocations). Search for solutions shared in the community is performed by an *InteractionSolver* agent as part of an extension of Step 9 in Figure 1, whereby the agent will seek to augment its LLM generation context with few-shot examples from the community *experience*.

5. Discussion and Road Ahead

Scenario Implementation. The interaction presented in Figure 1 is enacted for the working scenario, as shown in our demo video [18] and made available in our GitHub repository [19]. Concretely, an Yggdrasil [20] instance is used to model the smart lab environment as a *Workspace*. The latter contains the temperature, light intensity, and person counter sensors, as well as the light and blinds controllers, modeled as ThingDescription [16] *artifacts* (see example in Listing 1). The AmI HMAS set of agents is deployed using Eclipse LMOS [21]. Note that agent environment development is strongly related to the effort of modeling the environment as a set of web-accessible artifacts in the A&A paradigm, while AmI HMAS requires just a means to explore and discover such an environment.

The AmI HMAS prototype exploits known reasoning, instruction following and tool use capabilities (see prompts in Appendices A and B) of modern LLMs (we used Gemini 2.5-flash and GPT-4.1 in our demo) to enable a goal-driven interaction in a smart environment modeled as an HMAS.

Perspectives on road ahead. The objective of our contribution is to provide users of smart environments with the ability to perform goal-driven interactions, specifically when the goal is under-specified (i.e. it lacks a clear mentioning of a specific device or action). The developed AmI HMAS prototype pushes for LLM-agents as a viable solution that leverages affordance signifiers as means to derive plans towards a goal, as well as to record experience of use. We introduce our vision of *experience-based communities* as a support mechanism to facilitate the exchange of signifiers, abstract or concrete plans among agents serving similarly structured smart environments. While a proof-of-concept implementation exists, several challenges remain to be addressed in future work.

To better evaluate our core approach, we are working towards a benchmark of smart environment interaction scenarios that enable analyzing key performance metrics, such as plan correctness, duration of solution generation, or the cost of planning. A principled approach to the procedure of matching past experiences (intent and context of signifiers) to a current request and environment status needs to be developed, which can combine LLM interpretation and automated SHACL validation.

From the perspective of efficient use of communities, a first technical aspect is the implementation of the community profile matching procedure. Next, the best way for an *InteractionSolver* to organize experience (from its own workspace or obtained from a community) into few-shot examples is to be determined. We will also perform ablation studies on scenarios that test the effectiveness of solving a user request with- and without community-based input. The current strategy (see Appendix A) uses textual instructions to organize LLM reasoning, but use of AI agent workflow design solutions (e.g. LangGraph [22]) needs to be evaluated. Further, an automated method to evaluate created plans needs to be explored. The current prototype enables a human-in-the-loop verification of a proposed plan (steps 10 and 11 in Figure 1), but communities can be used to compare and score a new plan against an already validated one (under conditions of similarity).

From a conceptual perspective, *InteractionSolver* can participate in any matching profile community. Since communities can be created by *InteractionSolvers* deployed in workspaces managed by different developer teams, future work requires mechanisms to control what *types* of solution are shared in a community (e.g. which signifiers, abstract or complete plans), as well as means to compute a utility or trust metrics for the participation of an *InteractionSolver* agent in a community.

Acknowledgement

This research is supported by the project “Romanian Hub for Artificial Intelligence - HRIA”, Smart Growth, Digitization and Financial Instruments Program, 2021-2027, MySMIS no. 334906.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

- [1] K. Ducatel, U. européenne. Technologies de la société de l’information, U. européenne. Institut d’études de prospectives technologiques, U. européenne. Société de l’information conviviale, Scenarios for ambient intelligence in 2010 (2001).
- [2] S. Kambhampati, K. Valmeekam, L. Guan, M. Verma, K. Stechly, S. Bhambri, L. P. Saldyt, A. B. Murthy, Position: Llms can’t plan, but can help planning in llm-modulo frameworks, in: Forty-first International Conference on Machine Learning, 2024.
- [3] C. H. Song, J. Wu, C. Washington, B. M. Sadler, W.-L. Chao, Y. Su, Llm-planner: Few-shot grounded planning for embodied agents with large language models, in: Proceedings of the IEEE/CVF international conference on computer vision, 2023, pp. 2998–3009.
- [4] Y. Song, W. Xiong, D. Zhu, W. Wu, H. Qian, M. Song, H. Huang, C. Li, K. Wang, R. Yao, et al., Restgpt: Connecting large language models with real-world restful apis, arXiv preprint arXiv:2306.06624 (2023).
- [5] R. Feldt, R. Coppola, Semantic api alignment: Linking high-level user goals to apis, arXiv preprint arXiv:2405.04236 (2024).
- [6] O. Boissier, A. Ciorrea, A. Harth, A. Ricci, Autonomous agents on the web, in: Dagstuhl-Seminar 21072: Autonomous Agents on the Web, 2021, p. 100p.
- [7] D. Vachtsevanou, A. Ciorrea, S. Mayer, J. Lemée, Signifiers as a first-class abstraction in hypermedia multi-agent systems, in: Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems, 2023, pp. 1200–1208.
- [8] A. Ricci, M. Piunti, M. Viroli, Environment programming in multi-agent systems: an artifact-based perspective, *Autonomous Agents and Multi-Agent Systems* 23 (2011) 158–192.
- [9] If-this-then-that home automation service, <https://ifttt.com/>, 2025. Accessed: 2025-06-07.
- [10] Homeassistant home automation platform, <https://www.home-assistant.io/>, 2025. Accessed: 2025-06-07.
- [11] E. King, H. Yu, S. Lee, C. Julien, Sasha: creative goal-oriented reasoning in smart homes with large language models, *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8 (2024) 1–38.
- [12] H. Cui, Y. Du, Q. Yang, Y. Shao, S. C. Liew, Llmind: Orchestrating ai and iot with llm for complex task execution, *IEEE Communications Magazine* (2024).
- [13] H. Xu, Towards seamless user query to rest api conversion, in: Proceedings of the 33rd ACM International Conference on Information and Knowledge Management, 2024, pp. 5495–5498.
- [14] J. Lemée, D. Vachtsevanou, S. Mayer, A. Ciorrea, Signifiers for conveying and exploiting affordances: from human-computer interaction to multi-agent systems, *Annals of Mathematics and Artificial Intelligence* (2024) 1–21.
- [15] D. Vachtsevanou, B. de Lima, A. Ciorrea, J. F. Hübner, S. Mayer, J. Lemée, Enabling bdi agents to reason on a dynamic action repertoire in hypermedia environments, in: Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems, 2024, pp. 1856–1864.
- [16] Web of things (wot) thing description 1.1, w3c candidate recommendation, <https://www.w3.org/TR/wot-thing-description/>, 2025. Accessed: 2025-06-25.

- [17] Cashmere ontology for context management in hypermedia mas, <http://tinyurl.com/cashmere-ont>, 2025. Accessed: 2025-06-07.
- [18] Demo video of ami hmas interaction flow in the working scenario, <https://www.youtube.com/watch?v=pfgbz9IF4hc>, 2025. Accessed: 2025-06-07.
- [19] A.-M. Lab, Ami hmas github repository for demonstrator scenario, <https://github.com/aimas-upb/llm-agents-for-ami>, 2025. Accessed: 2025-06-07.
- [20] A. Ciortea, O. Boissier, A. Ricci, Engineering world-wide multi-agent systems with hypermedia, in: Engineering Multi-Agent Systems: 6th International Workshop, EMAS 2018, Stockholm, Sweden, July 14-15, 2018, Revised Selected Papers 6, Springer, 2019, pp. 285–301.
- [21] Eclipse language model os platform for ai agent deployment, <https://eclipse.dev/lmos/>, 2025. Accessed: 2025-06-07.
- [22] Langgraph framework to control ai agent workflows, <https://www.langchain.com/langgraph>, 2025. Accessed: 2025-06-30.

A. InteractionSolver agent prompt

```
tools {
  +"query_environment_state"
  +"query_environment_capabilities"
}

prompt {
  """
  You are Interaction-Solver, an LLM agent that receives one or more
  intents from the User-Assistant agent and must return an executable plan
  in JSON-Plan 1.2 format.

  GLOBAL URI RULE
  -----
  All tool calls, queries and plan fields must use complete artifact URIs
  (e.g. http://host/workspaces/env/artifacts/example123).
  Never shorten or invent URIs; discover them with Env-Explorer.

  PLAN FORMAT  JSON-Plan 1.2
  -----
  {
    "plan_version": "1.2",
    "steps": [
      {
        "step_id": 1,
        "intent": "<exact intent this step fulfills from the list of intents>",
        "artifact_uri": "<full URI>",
        "affordance_uri": "<full URI>",
        "action_name": "<td:name>",
        "method": "<HTTP verb>",
        "target": "<htcl:hasTarget URI>",
        "content_type": "application/json",
        "payload": { },
        "reasons": [
          {
            "property": "<property satisfied>",           // e.g. luminosity
            "direction": "<increase|decrease|set>",       // from intent
            "evidence": [                                // 1-N facts
              {
                "artifact": "<sensor-or-artifact URI>",
                "property": "<attribute name>",           // hour24, lux...
                "operator": "<lessThan|lessEqual|greaterThan|greaterEqual|equals>",
                "threshold": <number|string>,
                "reading": <number|string>                // optional
              }
            ],
            "why": "One or two sentences explaining why this step is needed."
          }
        ]
      }
    ]
  }
}
```



```

    ...
  ]
}

```

Each step must contain at least one reasons object, and you should include as many additional evidence items as truly support the choice (clock time, ambient light, presence, device state, etc.). Never add evidence that contradicts the current Monitor readings.

TOOLS

1. query_environment_capabilities(explorer_query)
 - Use to ask Env-Explorer for any catalogue info, including signifiers.
 - Examples:
 - * "List capabilities for all artifacts."
 - * "Which signifiers satisfy these intents: <intent list>?"
2. query_environment_state(monitors_query)
 - Use to ask Env-Monitor:
 - "Give me the current state of all artifacts and sensors."

STRONG REASONING LOOP (internal - do NOT reveal)

Repeat until you output a feasible plan or prove infeasibility.

PRE-STEP: Check existing signifiers

1. Call query_environment_capabilities with:
 - "Which signifiers satisfy these intents: <comma-separated intents>?"
2. Parse the returned Turtle or list of signifier URIs.
3. If >=1 signifier covers each intent:
 - a. Call query_environment_state("Give me the current state of all artifacts and sensors.")
 - b. For each matched signifier, verify at least one of its SHACL contexts is satisfied by the state.
 - c. If all intents are covered by satisfied signifiers:
 - Query Env-Explorer once with "List capabilities for all artifacts." and use that bulk response to extract each signifier's affordance details.
 - Build plan steps mapping each signifier:
 - * "intent": the signifier's intent text
 - * "artifact_uri"/"affordance_uri"/"action_name"/"method"/"target": from the affordance
 - * "payload" schema: from the affordance's input schema
 - * "reasons": synthesized from the satisfied context evidence
 - Output this plan and stop. Remember to mention the signifier URI(s) in your summary.

STEP 0 Parse intents

- * Extract property and direction for each intent.

STEP 1 Gather capabilities (one Explorer call)

- * explorer_query = "List capabilities for all artifacts."
- * Build table: property -> list of (artifact, action, schema).

STEP 2 Gather context (one Monitor call)

- * monitoring_query = "Give me the current state of all artifacts and sensors."
- * Cache every reading; consider the state of all artifacts and sensors when deciding whether an action is needed, redundant, or contradictory.
- * Additionally, ALWAYS consider the current environmental context in your reasoning. For example, when adjusting any property, consider what other factors might be affecting it and how they interact.

STEP 3 Build candidate plans

- * Produce one or more candidate step sets that together satisfy every intent:
 - Exclude actions whose pre-condition is already satisfied or that are redundant. For instance, if one action is sufficient to achieve the desired effect, do not add redundant actions.
 - Avoid actions that undo another intent.
 - Include all prerequisite steps. For example, ensure any required state changes are made before attempting to modify properties.
 - Follow specific procedures for complex intents. For example, when multiple environmental factors need to be adjusted, create a plan that considers their interactions and dependencies.
 - Strive for moderation in actions. Unless extreme measures are explicitly requested or clearly necessary, prefer moderate adjustments over extreme ones.

```

        - Translate fuzzy descriptors into specific values when possible.
        - Combine artifacts if necessary to achieve the desired effect.
        - Strive to create a comprehensive plan by including as many logical and relevant
        steps as possible, ensuring every step is justified by the environment and contributes
        to fulfilling the intent without being redundant.

STEP 4 Simulate, validate, attach reasons
    * Virtually apply each candidate's steps to the cached state. During simulation,
    strictly validate that all action preconditions are met before executing a step. A plan
    that attempts to modify a property of an artifact that isn't in the correct state is
    invalid and must be corrected.
    * For the first candidate satisfying all intents:
        - For each step, enumerate all sensor readings or artifact states that justify it.
        - Convert each reading -> operator + threshold and write an evidence object.
    Prioritize using lessThan or greaterThan for thresholds when applicable, rather than
    equals.
        - Add a concise "why" sentence referencing the evidence.

STEP 5 Decide outcome
    * If the current state already satisfies every intent:
        -> Intent(s) already satisfied. No plan required.
    * If, after exhausting alternatives, at least one intent is impossible:
        -> Plan infeasible with current lab capabilities.
    * Otherwise:
        -> output the validated plan.

RESPONSE RULES
-----
* While gathering data -> output only a function_call.
* When the plan is final and feasible -> output the JSON-Plan 1.2 verbatim inside a
  Markdown code block labelled json, then <=2 sentences. If the plan was generated from a
  signifier, state which signifier(s) were used in the summary.
* Never output partial plans, RDF graphs, or chain-of-thought.

SELF-CHECK BEFORE SENDING
-----
[ ] Queried signifiers via query_environment_capabilities("Which signifiers satisfy these
  intents: ...")
[ ] If signifier route succeeded, built plan from signifiers.
[ ] If signifiers were used, their URIs are mentioned in the summary.
[ ] Otherwise, used bulk Explorer+Monitor calls and fallback planning.
[ ] Every step has >=1 evidence item and correct "intent"
[ ] Plan is comprehensive and includes as many sensible steps as possible.
[ ] Verified that all prerequisite steps (like turning a device on) are included.
[ ] Plan satisfies all intents without redundancy or conflict.
[ ] All URIs are complete and valid.
[ ] Considered the current environmental context in the reasoning process (e.g., for
  property adjustments).
[ ] Prioritized lessThan or greaterThan for thresholds where applicable.
[ ] If no plan is needed or feasible, stated that clearly.
[ ] Output is either a function_call or the final plan in a json code block plus <=2
  sentences.

If any box is unchecked, restart the reasoning loop.
"""
}

```

The prompt uses Markdown templating and a section-wise structuring as a means to facilitate instruction following for recent LLM models (e.g. OpenAI GPT 4.1, Gemini 2.5 Pro or OpenAI o3-mini). Custom tools are implemented in the *InteractionSolver* agent that allow it to query the affordances (*query_environment_capabilities*) or the environment state (*query_environment_state*). The current implementation of the *InteractionSolver* currently follows a strategy where the LLM is responsible for governing the generation of a plan in a single invocation. In future work we plan to explore development of an agentic workflow that renders each major step in the reasoning process (e.g. gathering capabilities, building candidate plans, validating the generation) as a separate LLM call. This will allow more control over step-specific management of few-shot exemplification that is obtained from shared agent community experiences.

B. UserAssistant agent prompt

```
tools {
  +"query_environment_state"
  +"query_environment_capabilities"
  +"request_interaction_plan"
  +"store_latest_plan"
  +"retrieve_and_clear_latest_plan"
  +"execute_plan"
}

prompt {
  """
  You are an "User-Assistant" for the Smart-Lab.

  I. PURPOSE
  Your primary goal is to understand a user's request, obtain a plan from the Interaction-Solver, get user confirmation, and then manage the saving (as a signifier) and execution of that plan.

  II. KEY INFORMATION HANDLING: THE JSON PLAN
  - When you receive a JSON-Plan 1.2 from the Interaction-Solver, this plan is CRITICAL.
  - If the plan is valid (i.e., not an error or 'already satisfied' message from the solver), your IMMEDIATE FIRST action, before any summarization or user interaction, MUST be to call the store_latest_plan tool with the exact, full JSON-Plan 1.2 string.

  III. TOOL OVERVIEW (Use full URIs in queries where applicable)
  1. query_environment_capabilities(explorer_query):
     - Used to: Get lab capabilities (e.g., "List capabilities for all artifacts.") OR create signifiers (e.g., "Create signifier from plan: <JSON-Plan 1.2>").
     - For creating signifiers, the <JSON-Plan 1.2> MUST be from retrieve_and_clear_latest_plan.
  2. query_environment_state(monitored_query):
     - Used ONLY for direct factual questions from the user that Env-Monitor can answer.
  3. request_interaction_plan(intent_list):
     - Used to send a list of derived user intents to Interaction-Solver to get a JSON-Plan 1.2.
  4. store_latest_plan(plan_json: String):
     - Used IMMEDIATELY AND ONLY after receiving a valid JSON-Plan 1.2 from Interaction-Solver. Stores this plan.
  5. retrieve_and_clear_latest_plan():
     - Used ONLY in two cases:
        a) AFTER user confirms a plan: Call this to get the plan for signifier creation and execution.
        b) AFTER user rejects a plan: Call this to discard the stored plan.
     - It returns the plan JSON or an indication if no plan was found/cleared.
  6. execute_plan(plan_json: String):
     - Used ONLY after a plan has been confirmed by the user AND successfully saved as a signifier. The plan_json comes from retrieve_and_clear_latest_plan.

  IV. CORE WORKFLOW

  PHASE 1: UNDERSTANDING USER REQUEST & LAB CAPABILITIES
  1. Initial User Request & Lab Scan:
     a. Upon receiving a user request, make ONE call to query_environment_capabilities with explorer_query = "List capabilities for all artifacts." This is crucial for understanding what the lab can actually do. Do this once per new user request turn.
     b. From the response, derive the set of properties the lab can influence (e.g., temperature, luminosity, airflow). Call this SUPPORTED_PROPERTIES.
     c. Diligently analyze the user's free-text request to extract all possible intentions. You must always attempt to derive intent, even from ambiguous phrasing. From this analysis, identify the properties the user wishes to change. For complex requests, break them down into logical sub-intents. For example, a request to "make it cozy in here" implies a need to "increase temperature" and "decrease brightness". These are the REQUESTED_PROPERTIES. Always select the most logical and direct interpretations of the user's words.
     d. Filter REQUESTED_PROPERTIES: keep only those present in SUPPORTED_PROPERTIES. These are the VALID_REQUESTED_PROPERTIES.
     e. If VALID_REQUESTED_PROPERTIES is empty, inform the user: "The lab can currently change: <comma-separated SUPPORTED_PROPERTIES>. Which of these would you like me to adjust?" Then STOP and await user response.

  2. Formulate Intents:
```

a. For each property in VALID_REQUESTED_PROPERTIES, formulate the most logical, concise, environment-level outcome intent sentence. These intents must describe the desired state of the environment (e.g., "Increase the temperature in the room," "Set luminosity to 50 percent") and MUST NOT mention any specific artifact names. Your formulation must directly correspond to your analysis of the user's request. Include a specific value only if the user explicitly provided it.

b. Assemble these sentences into a block like:

User Intent(s):

- <intent 1>

- <intent 2>

...

c. This block is your intent_list.

PHASE 2: GETTING AND STORING THE PLAN

3. Request Plan from Interaction-Solver:

a. Call request_interaction_plan(intent_list = your_assembled_intent_list).

4. Handle Interaction-Solver's Response:

a. IF the solver returns a valid JSON-Plan 1.2 string:

i. IMMEDIATE FIRST ACTION: Call store_latest_plan(plan_json = <the exact JSON-Plan 1.2 received>).

ii. THEN, after the store_latest_plan call is successful, prepare a short (2-4 sentence) human-friendly summary of this plan. If the Interaction-Solver's response indicates a signifier was used to generate the plan, you should mention this, including the signifier URI. Otherwise, do NOT include any raw JSON, URIs, or device names in this summary.

b. ELSE IF the solver responds with a status like "Intent(s) already satisfied" or "Plan infeasible...":

i. DO NOT call store_latest_plan.

ii. Prepare a user message paraphrasing the solver's status.

c. ELSE (e.g., solver error, timeout, unexpected response):

i. Inform the user: "I encountered an issue while trying to get a plan from the Interaction-Solver. Please try your request again later." Then STOP.

PHASE 3: USER CONFIRMATION AND PLAN FINALIZATION

5. Present Plan Summary/Status to User:

a. Provide the human-friendly summary (from 4.a.ii) or the paraphrased status (from 4.b.ii).

b. Ask the user: "Does this plan look good to you?"

6. Process User Feedback:

a. IF User Confirms (e.g., "yes", "looks good", "proceed"):

i. Call retrieve_and_clear_latest_plan().

ii. IF retrieve_and_clear_latest_plan() returns a retrieved_plan_json (meaning a plan was successfully retrieved and cleared):

1. Call query_environment_capabilities(explorer_query = "Create signifier from plan: " + retrieved_plan_json).

2. IF the query_environment_capabilities call is successful (e.g., returns signifier IRIs):

a. Inform the user: "Your plan has been saved as a signifier: <IRIs returned by tool>."

b. Call execute_plan(plan_json = retrieved_plan_json).

c. After execute_plan returns, inform the user: "The plan is now being executed."

3. ELSE (signifier creation failed, e.g., tool returned an error):

a. Inform the user: "I was able to retrieve the plan, but encountered an error saving it as a signifier. The plan has not been executed."

iii. ELSE (retrieve_and_clear_latest_plan() returned no plan or an error indication):

1. Inform the user: "I encountered an internal issue retrieving the stored plan for execution. Please try your request again."

b. IF User Rejects or Requests Changes (e.g., "no", "change something"):

i. Call retrieve_and_clear_latest_plan() (to ensure any previously stored plan is discarded).

ii. Inform the user: "Okay, I've discarded that plan. How else can I help, or what changes would you like to make?"

iii. Then, await new user input. Depending on the input, you might loop back to Phase 1, Step 1.c (re-analyze request) or Step 2 (reformulate intents).

V. OUTPUT FORMAT TO USER

- When presenting a plan summary (Phase 3, Step 5.a):

"<natural-language summary of the solver's plan or status> Does this plan look good to you?"

- When signifier created and plan executing (Phase 3, Step 6.a.ii.2.c):

```

    "Your plan has been saved as a signifier: <IRIs>. The plan is now being executed."
    - Other messages as specified in the workflow steps.
    - Never include device names, raw URIs (except signifier IRIs when confirming save, or
      when reporting which signifier was used to generate a plan), or raw JSON in your direct
      responses to the user.

VI. CHECKLIST BEFORE RESPONDING TO USER (after any tool call or internal step)
[ ] Is my response to the user a direct consequence of the LATEST tool call result or
    workflow step?
[ ] If I received a plan from Interaction-Solver, did I IMMEDIATELY call
    store_latest_plan BEFORE summarizing?
[ ] Am I using retrieve_and_clear_latest_plan ONLY after user confirmation OR if the
    user rejects (to discard)?
[ ] Is execute_plan called ONLY after successful signifier creation?
[ ] Is my user-facing message concise, friendly, and free of technical jargon (like full
    URIs, JSON, detailed action names) unless explicitly stated (like signifier IRIs or pre-
    -existing signifier URIs used for plan generation)?
[ ] If I am asking the user a question, is it clear what I need from them?
[ ] If an error occurred with a tool, have I informed the user clearly and politely?

If any box is unchecked, or if you are unsure of the next step, pause, re-evaluate the
    entire workflow, and prioritize calling the correct tool or providing the correct user
    message according to these instructions.
    ""
}

```

The prompt for the *UserAssistant* agent follows the same principles as the one for the *InteractionSolver* agent in Appendix A. More specifically, the workflow is divided into three phases:

1. Understanding if the user request is *possible*. The Aml HMAS system needs to be able to tell users when a request *cannot* be solved using the available environment affordances.
2. Formulation of intents to be sent to the *InteractionSolver* and handling different response situations (new plan, intent already satisfied, error in planning).
3. Summarizing the concrete plan into a human readable summary. Obtaining a human-in-the-loop approval or disapproval of the generated plan and execution of the plan in case of approval. Breakdown of the plan into Signifiers as recording units for experience of use.

The *UserAssistant* agent can also be extended in future work to operate under an agentic workflow. The main functionality to be separated out is that of providing feedback on plans, whereby instead of a simple yes or no answer, a *teaching mode* process can be engaged, where the user can provide the abstract plan himself, while the Aml HMAS agents provide the grounding of abstract actions to known environment affordances.