AmI HMAS: Hybrid Agents with Individual and Collective Experience-Aware Code-based Planning for Smart Environments

Vlad-Alexandru Florea[1], Alexandru Sorici[1], Vlad-Matei Drăghici[1], Andrei-Cătălin Barbu[1], and Andrei Olaru[1]

Department of Computer Science and Engineering, National University of Science and Technology POLITEHNICA Bucharest, Romania
{vlad.florea1709, alexandru.sorici, vlad_matei.draghici, andrei.olaru}@upb.ro, andrei.barbu2607@stud.acs.upb.ro

Abstract. We describe the functionality and implementation of AmI HMAS, a framework for agent-based, goal-driven, LLM-supported interactions with smart environments. AmI HMAS maps existing HomeAssistant deployments into semantically represented, navigable Hypermedia Environments, enabling discovery of real-world smart devices. The framework combines classic agency with LLM reasoning to perform environment exploration, request interpretation, community-based exchange of experience, and action planning. AmI HMAS leverages an engine that enables storage and reuse of past interaction experiences during reasoning, distinguishing between environment state requests, explicit commands and implicit / ambiguous requests. The planning approach is designed to produce BehaviorTree code-based procedural plans, that enable plan life cycle management and reuse. Plan components can be exchanged in a community of agents that manage different smart environments, leveraging the power of the community to improve solving requests. We evaluate the system quantitatively across two distinct setups (simulated homes in the HomeBench benchmark and cross-environment transfer in a smart research lab simulation), measuring planning success rates, signifier fast-path hit rates, LLM call reduction, and planning latency across different request types (explicit, ambiguous, single or multi-command, achievable or impossible) and experience reuse settings.

Keywords: Ambient Intelligence · Hybrid Agent Architecture · Behavior Trees · LLM Planning · Signifiers · Hypermedia MAS · Smart Environments

1 Introduction

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

reasoning capabilities of systems intended to bridge these capabilities together towards a user goal. However, the topic is receiving renewed interest thanks to Large Language Models (LLMs), which demonstrate capabilities for intent understanding [13,15], planning [2,3], and matching natural language to APIs [4,5]. Furthermore, the W3C Web Agents Community Group [8] has surfaced Hypermedia Multi-Agent Systems (HMAS) as a paradigm where agent environments obtain a web-based *representation* – a *digital twin* approach particularly suitable for smart environments.

Several recent benchmarks [13,15,16] evaluate LLM-based smart environment systems on affordance identification, filtering of impossible requests, and multi-device coordination. However, these interactions are considered in isolation: there is no persistent learning, no standards-based environment representation, and no mechanism for sharing experience across deployments.

In this paper, we present AmI HMAS[1], a framework for goal-driven, LLM-supported interactions with smart environments that addresses these gaps.

Our contributions are:

1. **HMAS integration pipeline**: A mapping engine converts HomeAssistant deployments into W3C WoT Thing Descriptions served via an HMAS platform, enabling semantic discovery of real-world devices.
2. **A hybrid agent architecture for BehaviorTree (BT) code-based planning**: A deterministic state machine (UserAssistant) invokes the LLM only at controlled injection points – Natural Language Understanding (NLU) and Natural Language Generation (NLG) – while an InteractionSolver generates compositional BT-specific *intermediate representations* (IR) of plans in JSON—with sequence, selector, parallel, action, and condition nodes—which are compiled to `py_trees`[2] objects and executed via a tick-based lifecycle.
3. **Signifier-enhanced dual-path planning with experience sharing**: Past successful interaction records are captured as structured experience units called *signifiers* (see Section 4).
4. **Community-based experience sharing**: InteractionSolver agents form communities based on shared environment affordance types and use the *signifier* experience of other agents as assistance with *ambiguous or incomplete* requests for which no prior personal experience exists.

2 Background and Related Work

AmI HMAS combines advances from three main related domains: hypermedia-driven multi-agent environment representation, *affordance signification* as a mechanism for structured experience recording, and LLM-based planning for smart environment interactions.

[1] Demo video: https://youtu.be/M7_sPTWmBFc; Source code: https://github.com/aimas-upb/llm-agents-for-ami

[2] https://py-trees.readthedocs.io/

2.1 Hypermedia MAS and Signifiers

Hypermedia Multi-Agent Systems (HMAS) [18] provide a design paradigm for engineering worldwide multi-agent systems using hypermedia, i.e. resources interlinked through typed REST-style references that agents can follow to discover and interact with their environment. The paradigm follows the Agents & Artifacts (A&A) approach [10], where environments are modeled as web-accessible *artifacts* (encapsulated functional resources representing devices or services) organized in *workspaces* (logical groupings of related artifacts, e.g. a room). The W3C Web of Things Thing Description [20] specification provides a standardized metadata format for describing smart device capabilities: properties (readable state), actions (invokable commands), and events (push notifications). We argue that an HMAS-based representation is a suitable choice for smart environments, given the inherent hierarchy and logical separation of living spaces, as well as the availability of smart environment management platforms (e.g. HomeAssistant [12]) which expose device controls over REST APIs.

Building on the notion of *interaction affordances* [19,28] – action possibilities that agents can perceive and act upon – *signifiers* have been introduced as first-class abstractions in HMAS [9] to bind specific resource affordances in environments to intended use. The term originates from Affordance Theory [28], where signifiers are the perceivable cues through which a user discovers *how* an object can be used; in HMAS the concept is lifted to agents, so that a signifier records the association between an intent and the affordance that can fulfill it. A signifier captures: the *intent* (e.g. "increase luminosity in a room"), the *affordance* being used (e.g. toggling a light), the *context* of execution (e.g. there are people in the room and light intensity is below 100 lux), and the recommended *ability* (e.g. having LLM reasoning support). Signifier exposure [22] and resolution [23] mechanisms have been proposed for BDI agents reasoning on dynamic action repertoires in hypermedia environments.

In our work, we interpret *signification* from the perspective of human-computer interaction, where the agent perceives the *possible* or *past experience* use of the *affordance* of a device, if even the actual device capabilities do not have a one-to-one correspondence between intent and action. For example, an agent may *record* that a smart light affords *notification*, event though there is no singular `notify` action affordance that can be invoked on the device. However, turning the light on and off in a pattern can count-as a visual notification. To make such *context-aware experience of use* associations concrete, the CASHMERE ontology [24] provides a vocabulary linking an intention with an appropriate affordance and context conditions (expressed using SHACL [21], a W3C language for specifying and validating constraints over RDF graphs).

2.2 LLM-Based Planning and Behavior Trees

LLM planning for API-grounded actions has been explored early on by approaches such as LLM-Planner [3] (few-shot grounded planning for embodied

agents) or RestGPT [4] (connecting LLMs with real-world REST APIs). Semantic API Alignment [5] maps high-level user goals to API calls. These works demonstrate that LLMs can generate executable plans but use unstructured or flat output formats (action lists, API call sequences) that lack composability and lifecycle management. Kambhampati et al. [2] argue that LLMs "can't plan, but can help planning" – suggesting their use as plan generators within *structured frameworks* rather than as autonomous planners. Moreover, it has been shown that LLM pretraining that favors code generation makes task solving more successful when the LLM invocation requests solutions as code [6].

To facilitate a code-based approach for the type of procedural planning we require in smart environment interactions, we consider Behavior Trees (BTs), which are well-established in robotics and game AI [17]. BT nodes are divided into control flow (composite), decorator, and leaf (action) nodes. Control flow nodes decide which children run: *sequence* executes left to right and fails on the first failure, *selector* returns the first success, and *parallel* runs children simultaneously according to a success policy. Decorators modify a single child's result (e.g., invert or retry), while *leaf nodes* perform actions or condition checks. Nodes return one of three states: RUNNING, SUCCESS, or FAILURE. Execution proceeds by *ticking* the tree from the root (or a running node) until the root returns SUCCESS or FAILURE. Many behavior tree frameworks also provide a blackboard mechanism that enables key–value read and write operations to coordinate node execution across different branches.

Our approach thus follows this principle: the LLM generates a BT specification within a constrained JSON IR schema, which is then validated, compiled, and executed deterministically. Recent EMAS community work further supports this direction: Asici et al. [33] propose hybrid role-based architectures for LLM-enhanced MAS, while Ichida et al. [31] and Gatti et al. [32] explore combining BDI agents with LLM capabilities for natural language environments.

2.3 Smart Environment Interaction Platforms and Benchmarks

Commercial platforms such as IFTTT [11] and HomeAssistant [12] combine device affordances through rule-based automations and are beginning to integrate AI chatbots, but neither currently supports AI-aided *creation* of automated interactions.

Sasha [13] uses LLMs for smart home control, finding that plans can fail when requests are under-specified (e.g. "make it cozy in here"), motivating iterative reasoning. LLMind [14] presents an LLM-based agent framework for IoT devices, including an *experience accumulation* mechanism that locally stores LLM-generated control scripts. While LLMind demonstrates the value of experience reuse, its scripts are environment-specific and not structured for cross-environment transfer. HomeBench [15] benchmarks valid and invalid instructions across single and multiple devices, evaluating affordance identification, impossible-request filtering, and multi-device coordination. SimuHome [16] adds temporal and environment awareness, evaluating agents under a ReAct framework and finding that reasoning models require significantly longer inference

times. The key gap these works leave open is the combination of (i) a standards-based environment representation enabling discovery and grounding, (ii) persistent learning from past interactions via formally represented usage experience, and (iii) cross-environment experience sharing.

In AmI HMAS we aim to addresses all three. We use an HMAS-based model of the environment where devices, services and sensors are represented as TD artifacts (see Section 3.2). Signification is used as a means to record *experience of use* that can be reused locally (in both explicit and vague request formulations) or shared through a community-based mechanism (see Section 5). The planning procedure leverages the ability to semantically navigate an HMAS environment and constrain the planning context through past experience affordance hints, while producing a BT-based procedural plan (see Section 4.2).

3 AmI HMAS: Architecture and Design

To better explain the AmI HMAS agent architecture and functionality, we consider two reference environments with the following setup. *Lab308* is a smart research lab with indoor temperature and light intensity sensors, plus controllable lights and blinds. We build a HomeAssistant deployment using virtual instances of devices in the *Lab308* setup using the HACS integration[3]. We showcase both explicit and ambiguous interactions by first asking the AmI HMAS system to bring Lab308 in a low-light condition (close blinds and turn off lights) and then complain about it "being to dark in here" (an implicit intent of increasing the luminosity in the lab). We further create a second HomeAssistant virtual deployment, whose room and device setup is taken from HomeBench's [15] *Home17* environment, featuring study room lights, climate control devices, and entertainment services.

We choose a use case from a smart home interactions benchmark to showcase that community-based *experience-of-use* is not limited to similar physical environments (e.g. all teaching or research labs in a faculty building), but rather anchored in shared *types* of devices and their affordances. More specifically, community-informed solutions are used in the case of implicit or ambiguous requests, in which the community provides *suggestions* of affordance use when local experience of *interpreting* the ambiguous requests is not available (see Algorithm 1 and the discussion in Section 5).

In our scenario, we consider the setup in which the AmI HMAS instances of these two smart environments are part of an agent community built around the shared use of the same affordance *types* (e.g. smart lights). When handling a similar ambiguous complaint about low light in *Home17* (e.g. "I cannot see at my desk") appropriate affordance hints can be obtained from the agent system of *Lab308* through the community experience sharing protocol.

[3] Home Assistant Community Store: https://www.hacs.xyz/

3.1 System Overview

AmI HMAS targets goal-driven interaction in smart environments modeled as Hypermedia MAS following the A&A paradigm [10]. On deployment, the system launches an environment discovery pipeline (top row of Figure 1) where existing HomeAssistant deployments are first mapped into semantically represented HMAS environments with TD-based artifacts. The HMAS view of the HomeAssistant deployment can then be discovered and navigated by software agents that collaborate to fulfill user requests.

The agent system has three specialized roles, the *User Assistant*, *Environment Explorer* and *Interaction Solver* agents, which operate in a "hybrid" mode, i.e. combining deterministic behaviors with LLM-supported reasoning in key injection points (see arrows pointing to LLM in Figure 1): intent extraction, plan generation (as structured code in the form of BTs), matching of new intents to past experiences, and plan summarization. The agents are implemented using the SPADE [25] framework. The latter is an engineering choice, given its Python-native development (allowing for easier mixing of deterministic state-machine like behavior and modules for LLM-based reasoning) and its support of a communication infrastructure over XMPP.

Experience reuse and sharing is achieved by using the Signifier Memory Engine and interacting with the Communities, formed of other InteractionSolver agents (see bottom components in Figure 1), also interacting via SPADE/XMPP. The former facilitates recording of *affordance usage experience*, as well as matching of explicit or ambiguous (implicit) intents to previous experiences. The Community Interaction API extends the space of possibility for relevant experience collection, specifically for ambiguous or incomplete user requests, thereby alleviating a *cold start* problem, where the local AmI HMAS instance does not possess prior experience or user preference feedback on a particular request formulation.

3.2 HomeAssistant to HMAS Mapping

The integration engine bridges HomeAssistant deployments and the HMAS environment. The HomeAssistant platform is designed to group entities (smart devices and sensors) by rooms where they are deployed. All rooms are part of a home. This default hierarchical logical grouping facilitates an easy mapping to the core concepts of an A&A Hypermedia MAS environment model: HomeAssistant *areas* map to HMAS *workspaces*, while *devices* within each area become *artifacts* with W3C WoT Thing Descriptions. Entity attributes and state values are exposed as *PropertyAffordances* (readable via HTTP GET), while entity services are exposed as *ActionAffordances* (invokable via HTTP POST with JSON input schemas). Apart from artifact-specific property or action invocations, the integration engine implements A&A operations such as *joining* a workspace and *focusing* on an artifact. Such operations allow an agent to receive event updates from all a single artifact (focus) or all artifacts in a workspace (join). The engine achieves this by implementing a listener on state changes in the HomeAssistant platform and relaying them via WebSub event notifications to all agents subscribed by *join* or *focus* operations.

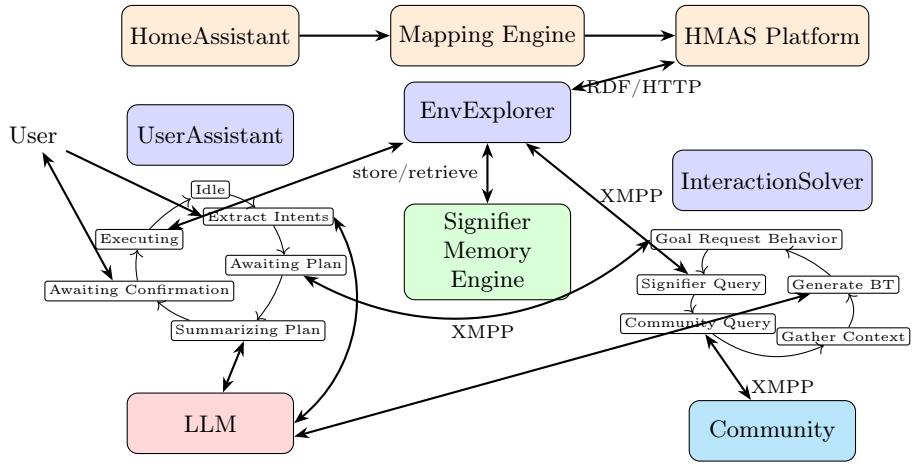

Fig. 1. AmI HMAS system architecture. HomeAssistant deployments are mapped to W3C WoT Thing Descriptions served via an HMAS Platform. Three agents roles –*UserAssistant*, *EnvExplorer* and *InteractionSolver* – collaborate using the SPADE communication infrastructure over XMPP. The Signifier Memory Engine (managed foremost by the EnvExplorer) and Community API enable storage and sharing of affordance use experience. Thin pointed arrows indicate the main operation cycle of the UserAssistant and the InteractionSolver, detailed in Section 3.3

Each artifact receives a W3C WoT Thing Description in RDF/Turtle. For instance, `LightSensor308` in Lab308 exposes a *PropertyAffordance* named *luminosity* with an HTTP GET endpoint returning a JSON object with a numeric value in Lux. At the same time `Light308` exposes *ActionAffordances* for turning the light on or off and changing its brightness (see Appendix B for example Thing Descriptions).

3.3 Agent Roles and Interaction Flow

AmI HMAS employs three agents, each with a distinct role, that communicate using the SPADE/XMPP infrastructure.

The EnvExplorer is the agent that *discovers* and catalogs the environment, indexing the *affordances* of available artifacts and tracking changes in PropertyAffordance values. The agent also manages the Signifier Memory Engine that records and recalls past affordance *usage experiences*. On startup, it crawls the HMAS platform via the integration engine, following the W3C WoT Discovery protocol [29], performing a four-phase discovery: (i) map workspaces, (ii) map artifacts, (iii) extract affordances from Thing Descriptions, and (iv) fetch initial device state. Following discovery, the EnvExplorer executes a `focus` operation on each artifact in turn to maintain an up-to-date view of artifact state via WebSub push events. The EnvExplorer has three main interactions with the other AmI HMAS agents: (i) It will respond to capability and state

check requests from the UserAssistant agent, (ii) it will deliver selected artifact state values to the InteractionSolver, when they are needed for planning, and (iii) on request from the InteractionSolver, it will perform matching of a new intent with past experience using the Signifier Memory Engine (see Section 4.1).

The UserAssistant agent provides the interface to the user and has the main responsibility of *understanding* requests and *managing* the plans created in consequence. A view of its life-cycle is shown in Figure 1. The user-facing interactions of the UserAssistant start with request understanding and structured intent extraction. Requests are first broken down into *atomic* sub-goals, where atomicity implies that the natural language phrasing can no further be separated into simpler independent commands without undermining command dependency (e.g. condition → action rules, temporal ordering of commands). Each atomic sub-goal is then classified into one of four main categories: (i) *state-check* – interrogation of any environment state (e.g. "How humid is it in the room?"), (ii) *explicit goal request* – the action to take and the affordances to invoke are clearly specified (e.g. "lower the living room blinds to 50% and turn off the lights in the kitchen"), (iii) *implicit goal request* – the actions are vague or underspecified (e.g. "Make the room comfortable for my baby", "It's kind of dark in here", "Make the living room brighter"), and (iv) *preference stating* – the user specifies a preference with respect to environment conditions or actions to be carried out (e.g. "I prefer natural light to artificial one", "I feel comfortable at 25 Celsius and with a low AC fan setting").

The UserAssistant makes use of LLM reasoning over the TD description of environment artifacts to parse each sub-goal into a *structured* intent form which specifies: the `intent_text` of the sub-goal, the `artifact` that is targeted (e.g. `Light308`), the `workspace` (e.g. `Lab308`) in which the artifact is included, the `action` and `parameter` to be invoked or altered on the artifact (e.g. turn on, brightness), the parameter `value` (e.g. the value 20 for the brightness percentage) and the action semantics in the sub-goal: one of `check` (query the status), `set` (set a parameter to a value), or `modify` (alter a parameter value – e.g. "dim by 20%"). The list of structured intents resulting from a request parsing are then sent as `GOAL_REQUEST` messages to the InteractionSolver for planning.

The InteractionSolver is the agent responsible for reasoning about solutions to a user requests, creating the plan of required *ActionAffordance* invocations.

The main task is to generate code-based structured plans modeled as BTs. The key workflow, shown in Figure 1, implements a dual-path strategy (detailed in Section 4.2). Both paths first involve asking the EnvExplorer for signifiers that match to the new sub-goals. Depending on the information in each structured intent, the InteractionSolver will use:

1. Signifier fast-path: For *explicit* sub-goals with a action of type `set`. If the sub-goal matches a previous affordance use, the agent builds a BT directly using the *ActionAffordance* type and parametrization available in the structured intent (see Section 3.4).

2. LLM planning path: For *implicit* sub-goals or *explicit* ones whose action type is `modify`, agent will first perform a *context gathering* procedure. Context gathering involves (i) retrieving *ActionAffordance* suggestions based on signifier matches from the EnvExplorer or obtained through the Community API, (ii) obtaining a *selected environment state* view by asking the EnvExplorer to get the state of artifacts whose *ActionAffordances* have been suggested, as well as that indicated by the *context* of matched Signifiers (see Section 4.1). The gathered context and the structured intent information constitute the input for LLM-based reasoning to generate the BT plan.

3.4 BehaviorTree Code-Based Planning and Execution

Behavior Tree based procedural plans are an advancement over our previous work in plan generation and representation [26] and have conceptual and technical advantages compared to structure-free function generation (e.g. as in SAGE [34]) or Finite-State Machine (FSM) representation (e.g. as in LLMind [14]). BT plans are *modular* and *reusable*, since BTs can be easily added / replaced as sub-behaviors of other BTs. This makes *plan sharing* between agents more manageable than with a FSM approach. BTs support a simple life cycle management through step-wise "ticking", enabling continuous state checking control over execution. Nodes can synchronize across branches through a shared key-value store (blackboard), which allows modeling of read–compute–set behaviors (e.g. "Increase the brightness of the kitchen light by 20%") or requests which require temporal alignment (e.g. "Run the humidifier at 50% for 10 minutes after the oven has finished"). BT plans are also more interpretable and easier to debug, since node semantics facilitate simpler LLM-based conversion into human-readable descriptions for user-in-the-loop validation performed by the UserAssistant.

The IR supports three composite node types (`sequence`, `selector`, `parallel`) and three leaf types: `action` for *ActionAffordance* invocations (HTTP POST), `property` for *PropertyAffordance* reads (HTTP GET), and `condition` for property-value checks. The full mapping to `py_trees` classes is given in Table 5 (Appendix D). The LLM receives a system prompt[4] instructing on BT node types, common patterns (e.g. idempotent actions via selector; sequential and parallel composition), and the available affordances with HTTP endpoints and schemas.

The `AsyncBTPlanner` of the InteractionSolver generates BT plans via custom OpenAI function calling. The LLM receives a system prompt[5] teaching BT node types, common patterns (idempotent actions via selector – condition-action, sequential or parallel composition), and the available affordances with HTTP endpoints and schemas. The `generate_behavior_tree` tool constrains output to the JSON IR schema. If validation fails, the planner retries up to three times with feedback injection. A normalization step auto-fills missing `name` fields for smaller LLMs that occasionally omit them.

[4] Full system prompts are available in the source code repository at https://github.com/aimas-upb/llm-agents-for-ami/tree/emas2026.

[5] Full system prompts are available in the source code repository.

Listing 1.1 shows an example BT intermediate representation for the request "It's too dark in here" in Lab308. The `parallel` root executes three independent actions concurrently: toggling the light on, setting brightness to maximum, and opening the blinds.

Listing 1.1. BT JSON IR for "it's too dark in here" in Lab308. Three independent actions wrapped in a parallel node.

```
{
  "type": "parallel", "name": "IncreaseBrightness",
  "policy": "success_on_all",
  "children": [
    { "type": "action", "name": "ToggleLight",
      "action_url": "http://localhost:8080/.../light308/toggle",
      "parameters": {} },
    { "type": "action", "name": "SetBrightness",
      "action_url": "http://localhost:8080/.../light308/setBrightness",
      "parameters": {"brightness": 100} },
    { "type": "action", "name": "OpenBlinds",
      "action_url": "http://localhost:8080/.../blinds308/setPosition",
      "parameters": {"closedPercentage": 0} }
  ]
}
```

When the UserAssistant receives the JSON IR of a Behavior Tree plan, it uses it as input to an LLM call for human-readable plan summarization to be delivered for confirmation to the user. On confirmation, it compiles the BT back into a `py_trees` object using its `IRExecutor` engine. The UserAssistant can subsequently exert full control over plan execution by *ticking* the tree up to a configurable limit, returning an `ExecutionResult` with success status, tick count, and per-tick node history.

4 Signifier Memory Engine

In AmI HMAS, a *signifier* is a unit of record for usage experience, with the general meaning that: "for an atomic sub-goal X, affordance Y with parameters Z was used". Optionally (e.g. in the case of planning for ambiguous / implicit goal phrasing), the signifier record also contains information on the *context* in which the affordance was applied. The reasoning behind including *context* information for affordance to intent matching is that, in many cases, the correctness or efficiency of this association can depend on user preferences or environment conditions. For example, a request to "make Lab308 brighter" can be satisfied by both opening the blinds or turning on the lights (or both), but the way to achieve this depends on factors such as the time of day which affects outside luminosity, the weather or the preference of the user for natural light. In the current implementation, context is recorded only for *implicit* sub-goals and comprises the *PropertyAffordance* values of all artifacts in the same workspace as the one whose affordance is recorded. We leave a more elaborate mechanism of *context selection* (e.g. based on semantic modeling of action effects on more general environment variables) to future work.

4.1 Signifier Representation and Matching

The signifier model binds an affordance to a structured intent and an execution context (see Appendix C for an example record). Each record captures the structured intent (action, artifact, parameter, value), the affordance URI invoked, a payload hint (if parameterized), and – for *implicit* intents – structured conditions recording the environment state at execution time.

Signifiers are created by the UserAssistant after BT plan execution: the agent walks the tree and extracts one signifier per *ActionAffordance* invocation node – analogous to remembering which "ingredients" went into a meal with a desired flavor. Signifiers are sent to the EnvExplorer, which uses the Signifier Memory Engine to store them for future retrieval. To support context-aware matching, structured conditions are converted to SHACL shapes that can be validated against a current environment state graph.

When a new goal arrives, the system queries stored signifiers to find relevant past experience. The matching procedure depends on the intent type. For an *explicit* intent, the `intent_text` from a signifier is compared using sentence embedding vector cosine similarity with the same field from the new goal. If the similarity exceeds a configured threshold, the `action`, `artifact` and `parameter` structured fields are used as additional hard match requirements.

For *implicit* requests, the procedure is similar, but the final list of matching signifiers is further filtered by a context match. The current environment context is converted to an RDF data graph and validated against SHACL shapes obtained from the signifier structured conditions; signifiers whose preconditions are violated are excluded from the final result set. Note that, in the current version, context matching is considered from a *strict* match perspective. Options to perform context matching in a more flexible (e.g. machine learned) manner are left for future work.

4.2 Planning with Signifiers

As described in Section 3.3, use of signifiers can result in fast matches for previous *explicit* goals (which lead to deterministic new plan construction), or they can result in *affordance filters* for intents that are implicit or involve a relative change to the state of an artifact. The InteractionSolver implements a dual-path strategy based on intent type and signifier matches, formalized in Algorithm 1.

The entry to the algorithm is the list of *atomic* structured intents (`si_list`) together with the workspace identifier (`ws_id`), the URI of the top-level HMAS workspace that the local AmI HMAS instance manages (e.g. `lab308` or `home17`). The `ws_id` is needed both to scope the local signifier search (QuerySignifiers) and to drive environment state collection (GatherContext). For each intent, signifier matching is performed once against local experience (line 3). Lines 4–6 show the *fast path*, in which an `explicit` intent with action type `set` and at least one validated signifier match leads to a directly built BT and an early continuation of the loop. For all remaining cases (`implicit` intents or `explicit` ones with action type `modify`), the matched signifiers serve as *affordance hints*

Algorithm 1 Dual-Path Planning Decision

Require: si_list, ws_id
Ensure: bt_plan
1: $bt_list \leftarrow [\,]$ ▷ list of BT for each structured intent
2: **for** each si in si_list **do**
3: $aff_hints \leftarrow$ QuerySignifiers(si, ws_id) ▷ match against local experience
4: **if** $si.type =$ `explicit` $\wedge$ $si.$`action` $=$ `set` $\wedge$ $aff_hints \neq \emptyset$ **then**
5: $bt_{si} \leftarrow$ BuildBTFromSignifiers(si, aff_hints) ▷ fast-path
6: $bt_list.append(bt_{si})$; **continue**
7: **end if**
8: **if** $si.type =$ `implicit` $\wedge$ $empty(aff_hints)$ **then**
9: SendCommunityRequest(si)
10: **wait until** 75% of community responded $\vee$ timeout
11: $aff_hints \leftarrow$ GatherCommunityResponses(si)
12: **end if**
13: $ctx \leftarrow$ GatherContext(si, ws_id, aff_hints) ▷ gather environment state
14: $bt_{si} \leftarrow$ LLMGenerateBT(si, ctx)
15: $bt_list.append(bt_{si})$
16: **end for**
17: $bt_plan \leftarrow$ MergeBTList(bt_list)
18: **return** bt_plan

for LLM-based generation. For `implicit` intents whose local search returned no match, the search is extended to community peers (lines 7–10, see also Section 5). The affordance hints inform the environment state collection on line 11; the affordance suggestions together with the gathered state constitute the input for LLM-based BT generation (lines 12–13).

For `set` action types the LLM is instructed to build idempotency through `selector` control nodes, which first check the value of the parameter changed by the corresponding affordance (e.g. `brightness` for `set_brightness`) and then set it to the value from the intent on mismatch. Intents with `modify` action types require a read-compute-set instruction, where the *read* uses *PropertyAffordance* nodes to access values and write them to the BT blackboard, *compute* implements a custom modification logic, and *set* uses *ActionAffordance* nodes to adjust to the customly computed value from the blackboard.

The BTs generated for each independent structured intent are merged into a single plan for the entire request (line 14) using a `parallel` control node with `success-on-one` policy (i.e. node failures are independent of each other).

5 Community-Based Experience Sharing

As introduced in our prior work [26], we envision *experience-based agent communities* as groups of InteractionSolver agents with a common interaction profile.

Community Architecture. Communities are organized around shared *affordance types*, such as "the community of agents having access to smart lightbulbs" or "the community of agents that can manage humidity". InteractionSolver

agents communicate via the SPADE/XMPP infrastructure to query other such agents for *signifiers* that could be used for the given list of *intents*. If the InteractionSolver agent searches for relevant signifiers in planning for the achievement of a goal, if no useful signifiers are found *locally*, the agent sends a query to all agents in communities that it is part of, sending the intents that it needs to solve. Any relevant signifiers that are sent back by the community are integrated in the planning process as possible hints on achieving the goal. After successful BT execution, signifiers are recorded locally. Whenever a query arrives in relation to a list of intents, the agent responds with a list of signifiers obtained in the same manner as for its own intents.

Intent-Level Transfer. A key design principle is that signifiers are queried at the *intent level*, not at the exact affordance level. For example, in Lab308 a user says "It's too dark in here," resulting in signifiers for intents like "turn on the light" and "set brightness to maximum" being published to the community. Later, in HomeBench Home 17, a user says "I can't read anything at my desk." The InteractionSolver finds Lab308's signifiers via embedding similarity, but since Home 17 has different affordance URIs (e.g. `studyRoomLight/turn_on`), they become *hints* in the LLM prompt rather than fast-path candidates. The LLM uses the intent-action *pattern* ("for increasing light, use a turn-on action") and maps it to Home 17's local affordances. Vocabulary alignment between environments is currently implicit (embedding similarity); explicit ontology-based alignment is left for future work.

6 Evaluation

We evaluate AmI HMAS from two perspectives. The first experiment evaluates the ability of LLM-supported reasoning to understand simple or compound, valid or invalid *explicit* user requests, and to create appropriate code-based BT plans for them.

The second experiment constructs a scenario that evaluates all the contributions discussed in this work, making quantitative and qualitative assessments on planning path selection, latency, and LLM usage across two environments, followed by a cross-environment transfer analysis.

6.1 Explicit Request Understanding and BT planning

We use a modified version of the HomeBench [15] benchmark in which we convert each home into a Hypermedia MAS environment, and give each element (room, device, action affordance) a semantic representation (e.g. `ex:LivingRoom`, `ex:AirPurifier`, `ex:SetModeCommand` – see more details about experiment setup in Appendix A). We select 400 test cases that cover request phrasings for single and compound goals, which can be feasible (i.e. the requested device and affordance exist in the environment) or not. All atomic intent formulations are `explicit` (e.g. "Set the curtain to 0 percent open and set the air purifiers fan speed to low in the living room.").

Table 1. Overall performance summary across all 400 test cases per model. *Full success*: fraction of tests where all expected actions were correctly executed. *Partial success*: tests where execution was partially verifiable and at least one action matched. *Full+Partial*: combined success rate. *Impossible detected*: fraction of infeasible intents correctly identified. Precision (P), recall (R), F1, and duration are aggregated over all tests. Best value per column in **bold** (lowest for duration).

Model	Full succ.	Partial succ.	Full+ Partial	Imp. det.	P	R	F1	Dur. (s)
GPT-4o	76.8%	19.0%	95.8%	99.3%	**0.952**	0.908	0.929	15.1
GPT-4o-mini	**80.5%**	15.8%	96.3%	99.6%	0.949	0.934	0.941	17.8
GPT-5-mini	63.8%	**32.5%**	**96.3%**	**100%**	**0.952**	**0.954**	**0.953**	16.1
GPT-5-nano	64.0%	24.3%	88.3%	**100%**	0.943	0.811	0.872	**12.1**

We propose an evaluation pipeline that is similar (but not completely identical) to the processing in Algorithm 1. A first LLM call parses a user utterance into atomic structured intents – each capturing a verb (`set` or `modify`), a target artifact type, an action affordance, and optional parameter–value pairs. Our focus in this experiment is the evaluation of correct parsing of single or compound phrases into structured intents when a semantic vocabulary (an ontology for the types of rooms, devices and affordances in HomeBench) serves as reference. Therefore, instead of matching against past experience, we use SPARQL queries that leverage a TD HMAS semantics and the structured intent to identify concrete affordance target URIs and parameter names (see Appendix A). Intents returning no bindings are classified as infeasible and cached in a per-environment experience store to avoid redundant future queries. Resolved intents are routed into two Behavior Tree (BT) construction paths in a manner identical to Algorithm 1: `set` intents are translated directly into action leaf nodes, while `modify` intents are delegated to an LLM code-generation step. The resulting subtrees are combined under a deterministic `Parallel` composite and executed as an in-memory `py_trees` object. Table 1 shows the overall performance of the experiment accross four different models (two general instruction following models – GPT-4o-mini, GPT-4o –, and two reasoning models – GPT-5-mini, GPT-5-nano with low reasoning effort). The meaning of precision and recall in this case is given by formulas:

$$prec = \frac{\text{total matched actions}}{\text{total executed actions}}, rec = \frac{\text{total matched actions}}{\text{total expected actions}}$$

The results show that models achieve a high F1 score and a near perfect percentage of infeasible intent identification. Error analysis reveals that most cases of partial success are for `modify` actions, where the modification is misinterpreted (e.g. a decrease by 50% is interpreted as percentage-relative instead of absolute, even though the measurement unit *is* percentages) or wrongly computed. Differences between models show that in the case of *explicit* intents, small instruction following models (4o-mini) outperform small reasoning models (5-nano) when the code template (BTs) is clearly explained. Large reasoning models (5-mini)

achieve the best results overall owing to a much better pre-training for code generation. More details to the experiment results are available in Appendix A.

6.2 Experience Reuse

To test all the AmI HMAS functionality we define a more constrained setup, equivalent to the scenario described at the start of Section 3. We define 17 test requests across two environments: *Lab308* (a smart research lab with 4 affordances on a light and motorized blinds) and *HomeBench Home 17* (a simulated study room with 3 affordances on a light). Requests span 7 categories: simple, implicit, explicit (with parameter), multi-intent, modify (relative change), check (status query), and impossible (no affordance match). Table 2 lists all 17 requests grouped by evaluation phase and environment.

Table 2. Evaluation requests grouped by environment. Columns show the planning path in each phase: A (cold start), B (warm start), C (cross-environment transfer with Lab308 community hints; Home 17 only).

ID	Env.	Category	Request text	A	B	C
L01	Lab308	simple	"turn on the light"	LLM	fast	–
L02	Lab308	implicit	"it's too dark in here"	LLM	fast	–
L03	Lab308	explicit	"set brightness to 80"	LLM	fast	–
L04	Lab308	simple	"open the blinds"	LLM	fast	–
L05	Lab308	implicit	"make the room bright"	LLM	fast	–
L06	Lab308	multi	"turn off light and close blinds"	LLM	fast	–
L07	Lab308	modify	"dim the light by 20"	LLM	LLM	–
L08	Lab308	check	"is the light on?"	LLM	LLM	–
L09	Lab308	impossible	"turn on the heater"	imp.	imp.	–
L10	Lab308	explicit	"close blinds to 50%"	LLM	fast	–
H01	Home 17	simple	"turn on the study room light"	LLM	fast	hints
H02	Home 17	implicit	"I can't read anything at my desk"	LLM	fast	hints
H03	Home 17	explicit	"set the light color to warm white"	LLM	fast	hints
H04	Home 17	simple	"turn off the light"	LLM	fast	hints
H05	Home 17	implicit	"make the room cozy"	LLM	fast	hints
H06	Home 17	impossible	"increase the brightness"	imp.	imp.	hints*
H07	Home 17	impossible	"turn on the heater"	imp.	imp.	imp.

*H06 is correctly identified as impossible in Phases A/B, but in Phase C the injected Lab308 hints cause the LLM to attempt a workaround via `set_color`.

The evaluation runs three phases: **A. Cold start:** No signifiers are available. All actionable requests go through the LLM planning path. **B. Warm start:** Signifiers extracted from Phase A successes are used for signifier matching. Requests matching prior experience may hit the fast path (Algorithm 1). **C. Cross-env transfer:** Lab308 signifiers from Phase A serve as community hints

for HomeBench requests. Since affordance URIs differ, the fast path should not fire; instead, hints guide the LLM. For each request, we record: the planning path (fast/LLM/impossible), number of LLM API calls (including retries), wall-clock latency, structural validity of the generated BT, and correctness of affordance URL usage.

Table 3 summarizes the results. In Phase A (cold start), all 14 actionable requests produce valid BTs through the LLM path, with 3 impossible requests correctly identified, establishing a 2449 ms baseline. Phase B demonstrates signifier reuse: 12 of 14 actionable requests hit the fast path with zero LLM calls and sub-second latency. Notably, implicit and multi-intent requests also benefit–e.g., "it's too dark" matches three stored signifiers (light-on, brightness, blinds) which the fast path assembles into a composite BT. Only `modify` and `check` requests correctly fall through to the LLM. Phase C confirms intent-level transfer: all 6 actionable HomeBench requests succeed using Lab308 community hints, and all generated BTs correctly use HomeBench affordance URLs—confirming transfer without affordance-level contamination. Comparing against the Phase A baseline restricted to the same Home 17 requests (2174 ms rather than the full 2449 ms average, which includes the slower Lab308 environment), Phase C yields a 17.7% latency reduction (1789 ms), suggesting that community hints help the LLM converge faster.

The cross-environment scenario further reveals both a strength and a limitation of intent-level transfer. Although Lab308's signifiers enable HomeBench to select correct affordances despite differing artifact names and URIs (all five matching-intent requests yield correct BTs), *capability* mismatches introduce *risk*. For example, when evaluated with "increase the brightness", the LLM maps the request to `set_color` (since HomeBench lacks `set_brightness`), instead of plainly rejecting it as impossible. This speaks to the broader discussion of the difference between *affordance* and *capability* (in this case, assuming that color changes affect brightness). At the technical level, a caveat of experience-augmented LLM planning is that injected hints can override inherent impossibility detection, leading to plausible but possibly incorrect workarounds.

Table 3. Summary results across three evaluation phases. Requests: total requests; Fast/LLM/Imp.: number of requests per planning path; Succ.: success rate; Avg Lat.: average planning latency for actionable (non-impossible) requests; LLM Calls: total LLM API calls.

Phase	Req.	Fast	LLM	Imp.	Succ.	Avg Lat.	LLM Calls
A (Cold)	17	0	14	3	100%	2449 ms	18
B (Warm)	17	12	2	3	100%	337 ms	5
C (Cross)	7	0	6	1	100%	1789 ms	7

7 Conclusions and Future Work

The AmI HMAS framework presented in this work shows how research efforts stemming from the Web Agents community (Hypermedia MAS environment engineering principles, signification as a method to record experience of use) suitably facilitate development of goal-driven smart environment interactions. We build our framework around classical agent functionality that is carefully and specifically augmented with LLM-based reasoning where this is most impactful: goal *understanding* and *structuring*, and *code-based procedural planning*. We leverage signifiers as units for experience tracking and note their greatest impact in supporting faster planning especially for *implicit* or underspecified requests (e.g. "turn on the light", "it's too dark in here"). We support experience sharing through communities organized around shared *affordance types* and use it to support *cold start* cases of implicit requests where no prior experience exists in the local AmI HMAS system. We further prove through experimentation that using structured intents and tasking even small LLMs (e.g. GPT-5-nano with a low reasoning effort) to produce Behavior Tree plans *as code* yields good performance on *explicit* request management, and that signifer-based experience reuse leads to reduced planning latency for *implicit* requests.

At the same time we acknowledge that the current work has limitations in terms of scope and technical solution choice. The tests in HomeBench and our two-environment scenario cover cases that have a one-to-one mapping between the atomic intent and the affordance that can satisfy it. We plan to expand evaluations to cases where atomic intents involve actions that are logically (e.g. if-then rules) or temporally conditioned and develop an accompanying environment simulator. From technical perspectives, we aim to improve the context scoping in signifiers (what information gets recorded as context for a intent – affordance binding) by adding explicit modeling of environmental variables (e.g. luminosity, temperature) and the action- and property affordances of artifacts that impact / are impacted by their change. This effort will draw inspiration from existing work [35]. We will also look into more flexible context matching procedures that go beyond the current SHACL-based strict equivalence or value range verifications. Furthermore, we aim to improve planning speed by means of smaller, fine-tuned LLM models that cater to BT-based procedural plan generation.

Use of local LLMs. The current evaluation is run against the OpenAI API, but the choice of model provider is orthogonal to the architecture: the InteractionSolver, UserAssistant and EnvExplorer interact with the LLM through a thin OpenAI-compatible client and any back-end exposing the same chat-completions surface (e.g. Ollama, vLLM, llama.cpp servers) can be substituted by swapping a base URL and a model name in the configuration. We expect three main effects when moving to a locally-served model.

First, in line with our results on small reasoning models (Section 6.1), the BT-as-code planning task is feasible for small open-weight models: GPT-5-nano with low reasoning effort already reaches an F1 of 0.872 on *explicit* requests,

suggesting that 7–14B parameter local models tuned for code generation should be similarly competitive on `set`-type intents, while `modify` (read–compute–set) and `check` intents – which the warm-start phase shows to be the residual LLM users – are likely to benefit most from a stronger model.

Second, end-to-end latency depends on local hardware rather than network round-trips, and is therefore more variable; we expect the experience-fast-path hit rate observed in Phase B (12/14 actionable, sub-second) to amplify the practical value of local deployments by removing most LLM calls for already-seen intents and concentrating model use on novel or vague requests.

Third, privacy and data-locality benefits are particularly relevant for smart-home contexts, since LLM prompts in our pipeline contain device descriptions and ambient state. A systematic comparison between hosted GPT models and locally-served open-weight models is left for future work.

Apart from addressing the above limitations, future work also involves efforts to improve community-based experience sharing. Our focus will be on developing a method for automatic semantic vocabulary creation for the affordances of any given HomeAssistant deployment, complemented by an explicit vocabulary alignment procedure as part of the community protocol for signifier and plan sharing.

Acknowledgments. This paper is supported by the European Union's HORIZON Research and Innovation Programme under grant agreement No 101120657, project ENFIELD (European Lighthouse to Manifest Trustworthy and Green AI).

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

References

1. Ducatel, K., et al.: Scenarios for ambient intelligence in 2010. ISTAG Report, European Commission (2001)
2. Kambhampati, S., Valmeekam, K., Guan, L., Verma, M., Stechly, K., Bhambri, S., Saldyt, L.P., Murthy, A.B.: Position: LLMs can't plan, but can help planning frameworks. In: Forty-first International Conference on Machine Learning (2024)
3. Song, C.H., Wu, J., Washington, C., Sadler, B.M., Chao, W.-L., Su, Y.: LLM-Planner: few-shot grounded planning for embodied agents with large language models. In: Proc. IEEE/CVF ICCV, pp. 2998–3009 (2023)
4. Song, Y., Xiong, W., Zhu, D., Wu, W., Qian, H., et al.: Connecting large language models with real-world REST APIs. arXiv:2306.06624 (2023)
5. Feldt, R., Coppola, R.: Semantic API alignment: linking high-level user goals to APIs. arXiv:2405.04236 (2024)
6. Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., Ji, H.: Executable code actions elicit better LLM agents. In: Proc. 41st International Conference on Machine Learning (ICML) (2024)
7. Colledanchise, M., Ögren, P.: Behavior trees in robotics and AI: An introduction. CRC Press (2018)

8. Boissier, O., Ciortea, A., Harth, A., Ricci, A.: Autonomous agents on the web. In: Dagstuhl-Seminar 21072 (2021)
9. Vachtsevanou, D., Ciortea, A., Mayer, S., Lemée, J.: Signifiers as a first-class abstraction in hypermedia multi-agent systems. In: Proc. AAMAS 2023, pp. 1200–1208 (2023)
10. Ricci, A., Piunti, M., Viroli, M.: Environment programming in multi-agent systems: an artifact-based perspective. Autonomous Agents and Multi-Agent Systems **23**, 158–192 (2011)
11. If-This-Than-That home automation service, https://ifttt.com/, last accessed 2026-01-05
12. HomeAssistant home automation platform, https://www.home-assistant.io/, last accessed 2026-01-05
13. King, E., Yu, H., Lee, S., Julien, C.: Sasha: creative goal-oriented reasoning in smart homes with large language models. Proc. ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies **8**(1), 1–38 (2024)
14. Cui, H., Du, Y., Yang, Q., Shao, Y., Liew, S.C.: LLMind: orchestrating AI and IoT with LLM for complex task execution. IEEE Communications Magazine (2024)
15. Li, S., Guo, Y., Yao, J., Liu, Z., Wang, H.: HomeBench: evaluating LLMs in smart homes with valid and invalid instructions across single and multiple devices. arXiv:2505.19629 (2025)
16. Seo, G., Yang, J., Pyo, J., Kim, N., Lee, J., Jo, Y.: SimuHome: a temporal- and environment-aware benchmark for smart home LLM agents. arXiv:2509.24282 (2025)
17. Colledanchise, M., Ögren, P.: Behavior Trees in Robotics and AI: An Introduction. CRC Press (2018)
18. Ciortea, A., Boissier, O., Ricci, A.: Engineering world-wide multi-agent systems with hypermedia. In: Engineering Multi-Agent Systems, EMAS 2018. LNCS, vol. 11375, pp. 285–301. Springer (2019)
19. Ciortea, A., Mayer, S., Boissier, O., Gandon, F.: Exploiting interaction affordances: on engineering autonomous agents for the web of things (2019)
20. Web of Things (WoT) Thing Description 1.1, W3C Candidate Recommendation, https://www.w3.org/TR/wot-thing-description/, last accessed 2026-01-05
21. SHACL W3C Recommendation, https://www.w3.org/TR/shacl/, last accessed 2026-01-05
22. Lemée, J., Vachtsevanou, D., Mayer, S., Ciortea, A.: Signifiers for conveying and exploiting affordances: from human-computer interaction to multi-agent systems. Annals of Mathematics and Artificial Intelligence (2024)
23. Vachtsevanou, D., de Lima, B., Ciortea, A., Hübner, J.F., Mayer, S., Lemée, J.: Enabling BDI agents to reason on a dynamic action repertoire in hypermedia environments. In: Proc. AAMAS 2024, pp. 1856–1864 (2024)
24. CASHMERE ontology for context management in hypermedia MAS, http://tinyurl.com/cashmere-ont, last accessed 2026-01-05
25. SPADE Agent Development Framework, https://spadeagents.eu/, last accessed 2026-01-05
26. Sorici, A., Olaru, A., Florea, A.M.: Towards agentic AI support for Hypermedia MAS models of smart environments. In: 2nd International Workshop on Hypermedia Multi-Agent Systems (HyperAgents 2025), in conjunction with ECAI 2025 (2025)
27. Sorici, A., Udrăstoiu, V.-V., Cordos, C., Olaru, A.: AmI HMAS: a Hypermedia MAS for goal-driven interactions with every-day smart environments. In: Proc. of

the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), pp. 4149-4151. (2026)
28. Norman, D.: The Design of Everyday Things: Revised and Expanded Edition. Basic Books (2013)
29. W3C Web-of-Things Discovery Protocol Specification, https://www.w3.org/TR/wot-discovery/, last accessed 2026-01-05
30. LangGraph framework to control AI agent workflows, https://www.langchain.com/langgraph, last accessed 2026-01-05
31. Ichida, A.Y., Meneguzzi, F., Cardoso, R.C.: BDI Agents in Natural Language Environments. In: Proc. AAMAS 2024, pp. 880–888. IFAAMAS (2024)
32. Gatti, A., Ciatto, G., Calegari, R., Omicini, A.: ChatBDI: Think BDI, Talk LLM. In: Proc. AAMAS 2025, pp. 2541–2543. IFAAMAS (2025)
33. Asici, T.Z., Acar, E., Dennis, L.A., Bordini, R.H.: Towards Engineering LLM-Enhanced Multi-Agent Systems: A Critical Examination of Roles. In: Proc. EMAS 2025. LNCS, Springer (2025)
34. Rivkin, D., Hogan, F., Feriani, A., Konar, A., Sigal, A., Liu, S., Dudek, G.: Sage: Smart home agent with grounded execution. arXiv preprint arXiv:2311.00772 (2023)
35. Ramanathan, G., Mayer, S.: Towards achieving adaptive behaviour of agents through physics-infused descriptions of cyber-physical devices (2025)

A Explicit Intent Understanding and BT plans - Experiment Details

The Explicit Intent Understanding and BT planning performance experiments are conducted on four curated test suites drawn from the HomeBench dataset, each containing 100 tasks targeting *single-feasible*, *single-infeasible*, *multi-feasible*, and *multi-mixed* request categories. Single tasks require one device action, whereas multi tasks compose two to three concurrent or sequential sub-goals spanning different rooms. Infeasible tasks—either entirely (*single-infeasible*) or partially (*multi-mixed*)—request affordances absent from the target home, testing the system's ability to correctly detect and skip impossible sub-goals while still executing the feasible remainder. Each smart home is described as an RDF knowledge graph combining the W3C WoT TD (`td:`) and HMAS (`hmas:`) vocabularies, structured as a two-level workspace–artifact hierarchy. We built a domain ontology defining 15 device classes and 23 action affordance classes for the HomeBench dataset, with individual artifacts exposing concrete `td:ActionAffordance` and `td:PropertyAffordance` entries with JSON Schema-typed parameters and HTTP controls.

SPARQL queries for affordance retrieval. One aim of our experiment is to validate the idea that augmenting a TD-based description of an HMAS smart environment with a minimal semantic vocabulary of artifact and affordance *types* and providing this vocabulary to an LLM (as a Turtle-serialized file) is effective in retrieving proper *intent structure*.

We therefore impose a *strict* SPARQL query, shown in Listing 1.2, to evaluate LLM ability to match natural language phrasing to the corresponding semantic vocabulary. A failure to produce bindings is interpreted as an *impossible* request.

Detailed, per request category, experiment results. Table 4 shows a more detailed per-request category performance breakdown that complements the general results discussed in Section 6.1. Notice that average planning duration highly varies by request type and model applied (e.g. ∼5 seconds for GPT-4o on single

Table 4. Per-category action precision, recall, F1-score, and average task duration (seconds) for each model variant on the neuro-symbolic pipeline. Categories: *SF* = single feasible, *MF* = multi-action feasible, *MM* = multi-action mixed (partially infeasible). Metrics are computed at the corpus level (aggregated action counts across all 100 tests per category). Best value per column in **bold** (lowest for duration).

	Single Feasible (SF)				**Multi Feasible (MF)**				**Multi Mixed (MM)**			
Model	P	R	F1	Dur.(s)	P	R	F1	Dur.(s)	P	R	F1	Dur.(s)
GPT-4o	**0.990**	**0.990**	**0.990**	**5.1**	**0.983**	0.907	0.943	25.8	0.947	0.881	0.913	28.1
GPT-4o-mini	0.980	**0.990**	0.985	6.5	0.976	0.920	0.947	26.9	**0.949**	0.931	0.940	35.8
GPT-5-mini	**0.990**	**0.990**	**0.990**	11.1	**0.983**	**0.942**	**0.962**	23.2	0.947	**0.954**	**0.951**	25.1
GPT-5-nano	0.988	0.810	0.890	8.7	0.976	0.795	0.876	**16.9**	0.940	0.828	0.880	**18.7**

```
PREFIX ex: <http://example.org/>
PREFIX hctl: <https://www.w3.org/2019/wot/hypermedia#>
PREFIX hmas: <https://purl.org/hmas/>
PREFIX http: <http://www.w3.org/2011/http#>
PREFIX jsonschema: <https://www.w3.org/2019/wot/json-schema#>
PREFIX td: <https://www.w3.org/2019/wot/td#>

SELECT ?workspace ?artifact ?affordance_name ?target_uri
       ?parameter_name ?parameter_schema_type
WHERE {
    ?workspace a <wsp_placeholder> ;
              hmas:contains <artifact_placeholder> .
    ?artifact a ex:Dehumidifiers ;
              td:hasActionAffordance <aff_placeholder> .
    ?affordance a ex:SetIntensityCommand ;
                td:name ?affordance_name ;
                td:hasForm ?form .
    ?form hctl:hasTarget ?target_uri .
        OPTIONAL {
        ?affordance td:hasInputSchema ?inputSchema .
        ?inputSchema jsonschema:properties ?property .
        ?property jsonschema:propertyName ?parameter_name .
        ?property a ?parameter_schema_type .
        FILTER (?parameter_name = <param_placeholder>)
    }
}
```

Listing 1.2. SPARQL query for structured intent field identification using an ontology vocabulary. The placeholders in the query are filled in by the LLM intent parsing prompt

task requests, and up to ∼36 seconds for multi-mixed requests which can contain up to 8 sub-goals in one phrasing, of which up to half can be impossible to achieve).

The F1 scores achieved by models across all categories confirm our hypotheses that procedural planning for smart environment interaction benefits from LLM-based reasoning, to the extent to which the task of the LLM is to generate *code* directly in a templated form (as a BT). The actual code-based solution in the case of the HomeBench explicit requests is in the *compute* nodes of intents that involve a modification to artifact state. The TD-based semantics means that reading and setting of corresponding state parameters can be templated through *ProperAffordance* and *ActionAffordance* nodes.

B Thing Description Examples

Listing 1.3 shows an excerpt of the W3C WoT Thing Description generated by the mapping engine for `Light308` in Lab308, illustrating both *ActionAffordances* (`turn_on`, `set_brightness`) and *PropertyAffordances* (`on_off`, `brightness`).

Listing 1.3. Thing Description excerpt for `Light308` in Lab308 (selected affordances).

```
<http://localhost:8080/.../artifacts/light308>
    a hmas:Artifact ;
    td:title "Light308" ;
    td:hasActionAffordance [
```

```
        a td:ActionAffordance ;
        td:name "turn_on" ;
        td:title "turn_on" ;
        td:hasForm [
            htv:methodName "POST" ;
            hctl:hasTarget <http://localhost:8080/.../light308/
                turn_on> ;
            hctl:forContentType "application/json" ;
            hctl:hasOperationType td:invokeAction
        ]
    ] ;
    td:hasActionAffordance [
        a td:ActionAffordance ;
        td:name "set_brightness" ;
        td:title "set_brightness" ;
        td:hasInputSchema [
            a js:ObjectSchema ;
            js:properties [
                a js:NumberSchema ;
                js:propertyName "brightness" ;
                js:minimum 0 ;
                js:maximum 100
            ] ;
            js:required "brightness"
        ] ;
        td:hasForm [
            htv:methodName "POST" ;
            hctl:hasTarget <http://localhost:8080/.../light308/
                set_brightness> ;
            hctl:forContentType "application/json" ;
            hctl:hasOperationType td:invokeAction
        ]
    ] ;
    td:hasPropertyAffordance [
        a td:PropertyAffordance ;
        td:name "on_off" ;
        td:isObservable true ;
        td:hasOutputSchema [
            a js:StringSchema ;
            js:enum "on", "off"
        ] ;
        td:hasForm [
            htv:methodName "GET" ;
            hctl:hasTarget <http://localhost:8080/.../light308/
                on_off> ;
            hctl:forContentType "application/json" ;
            hctl:hasOperationType td:readProperty
        ]
    ] ;
    td:hasPropertyAffordance [
        a td:PropertyAffordance ;
        td:name "brightness" ;
        td:isObservable true ;
        td:hasOutputSchema [
            a js:NumberSchema ;
            js:minimum 0 ;
            js:maximum 100
        ] ;
        td:hasForm [
            htv:methodName "GET" ;
            hctl:hasTarget <http://localhost:8080/.../light308/
                brightness> ;
            hctl:forContentType "application/json" ;
            hctl:hasOperationType td:readProperty
        ]
    ] .
```

C Signifier Record Example

Listing 1.4 shows an example signifier record extracted after successful BT execution for the request "turn on the light" in Lab308. The intent was marked as `implicit` because the specific light is not named concretely; based on the known workspace model, the system resolved it to `light308`. The `structured_conditions` field captures the environment state at execution time, later converted to SHACL shapes for context-aware matching.

Listing 1.4. Signifier record extracted after successful BT execution. Fields capture the intent-affordance binding and execution context (structured_conditions).

```
{
  "signifier_id": "exec-a1b2c3",
  "intent": {"action":"set", "artifact":"light308", "parameter":"on_off",
            "value": true, "intent_text": "turn on the light"},
  "intent_type": "IMPLICIT",
  "affordance_uri": "http://localhost:8080/.../light308/turn_on",
  "action_name": "turn_on",
  "payload_hint": {},
  "workspace_id": "http://localhost:8080/workspaces/lab308",
  "was_successful": true,
  "source": "bt_execution",
  "structured_conditions": [
    {
      "artifact": "http://localhost:8080/.../artifacts/light308",
      "property_affordance": "on_off",
      "value_conditions": [{ "operator": "equals", "value": false }]
    },
    {
      "artifact": "http://localhost:8080/.../artifacts/light308",
      "property_affordance": "brightness",
      "value_conditions": [{ "operator": "equals", "value": 40 }]
    }
  ],
}
```

D BT Node Types

Table 5. BT node types in the JSON IR and their mapping to `py_trees` execution.

IR Type	py_trees Class	Runtime Behavior
`sequence`	`Sequence(memory=True)`	Execute children left-to-right; fail on first failure
`selector`	`Selector(memory=False)`	Try children left-to-right; succeed on first success
`parallel`	`Parallel(policy)`	Run all children concurrently; `success_on_all` or `success_on_one`
`action`	`ActionAffordanceNode`	HTTP POST to `action_url` with `parameters`
`property`	`PropertyAffordanceNode`	HTTP GET `property_url`
`condition`	`PropertyConditionNode`	HTTP GET `property_url`; compare to `expected_value` ($=, \neq, >, <, \geq, \leq$)