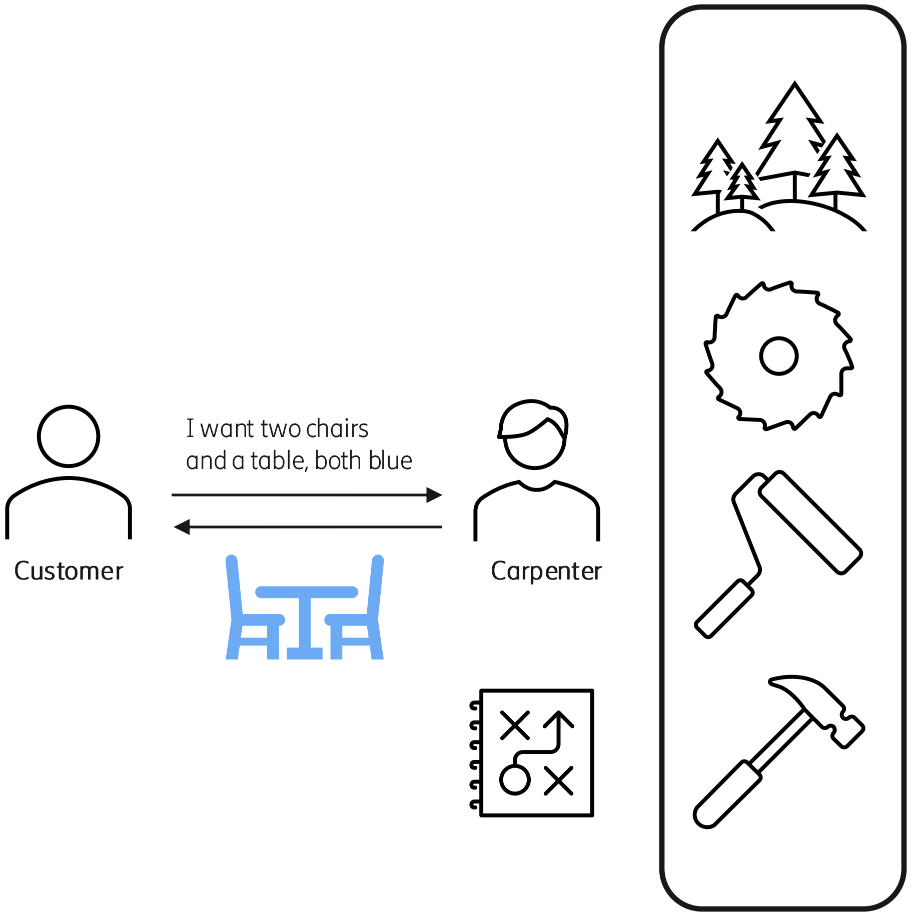
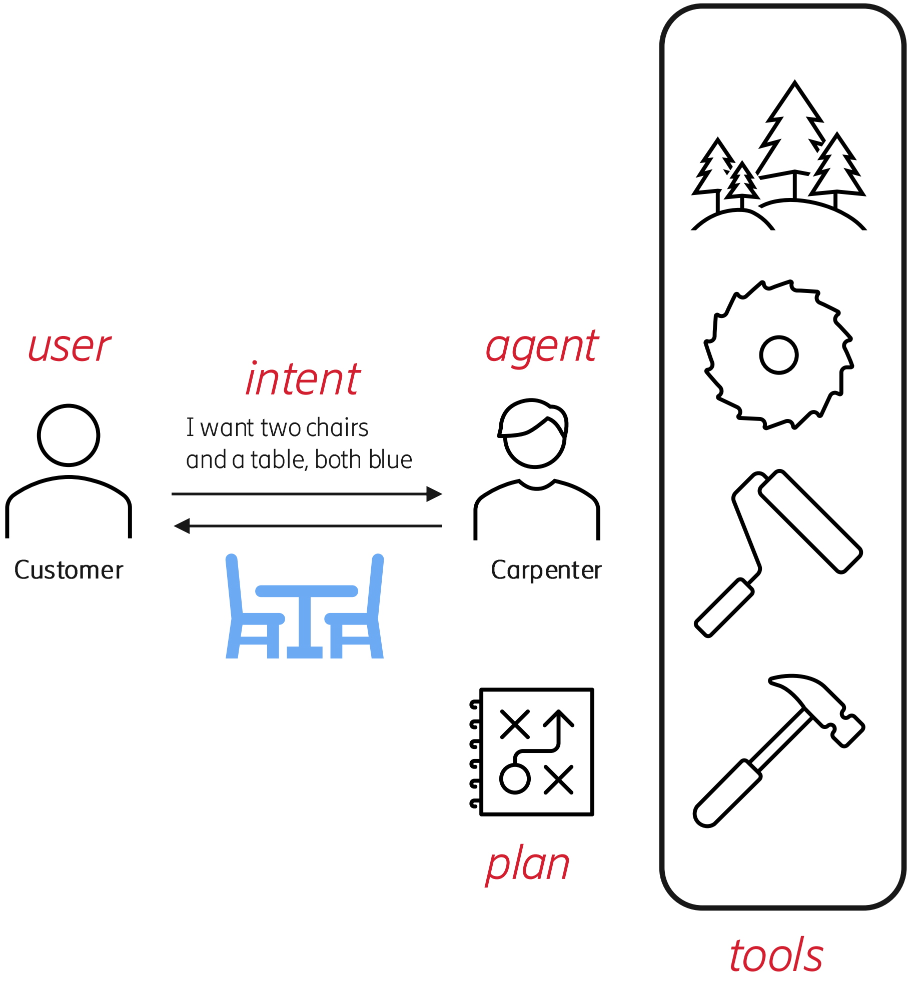
Agentic AI operation of IoT Systems

Jaime Jiménez
Tung Nguyen
Lorenzo Corneo

Context/Intro



Context/Intro



System Architecture

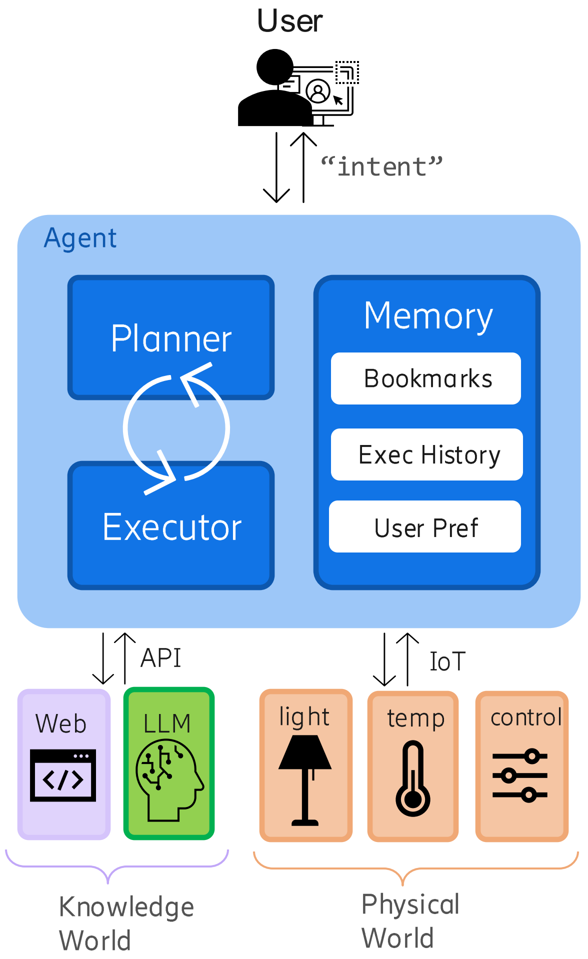
User: Interacts with the agent using natural language or **"intents"**.

Knowledge base: Source of ground truth, for example the web or other knowledge base.

Physical system: IoT devices communicating over standardized IoT APIs and data models.

LLM: Large Language Model that exposes an API endpoint for querying.

Agent: A system powered by an LLM, designed to break down user intents expressed in natural language into smaller, actionable steps. These steps enable the agent to effectively interact with various tools, such as databases, APIs, or devices.



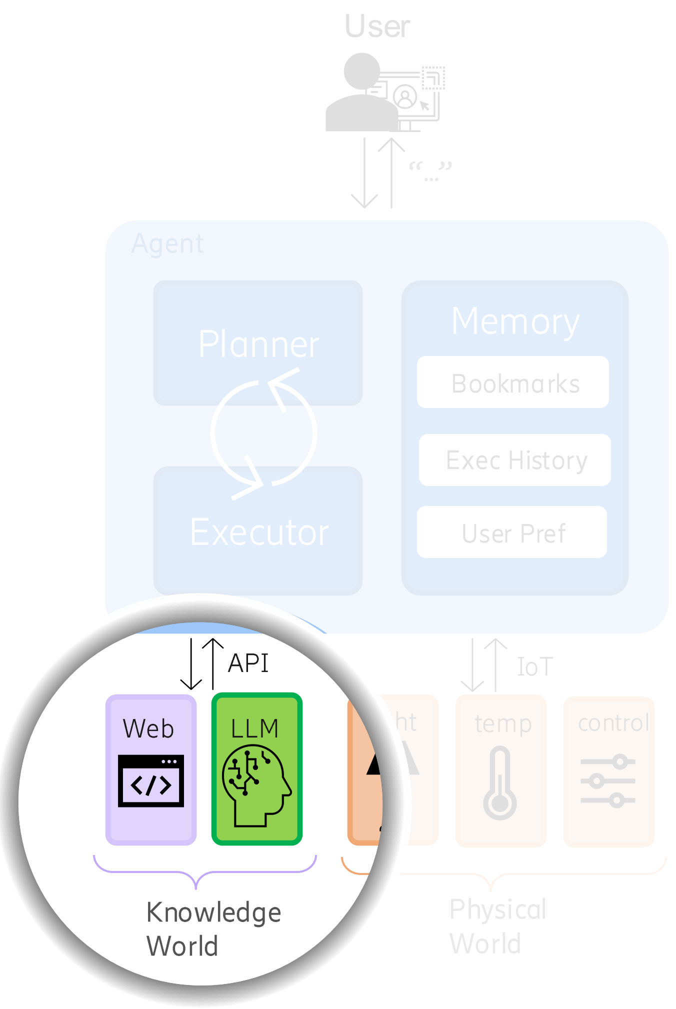
Knowledge base and LLM

The agent requires to interact with the web to gather fresh information that is not available in the initial prompt or in the memory.

It uses the function `websearch()` to browse the internet for clarifications. Other use case dependent tools (CPI, specifications, etc.) can be added.

The agent may also use the training data directly from the Large Language Model (LLM).

LLMs we have tried: OpenAI GPT4o, Deepseek-R1, Llama 3.3 (local).



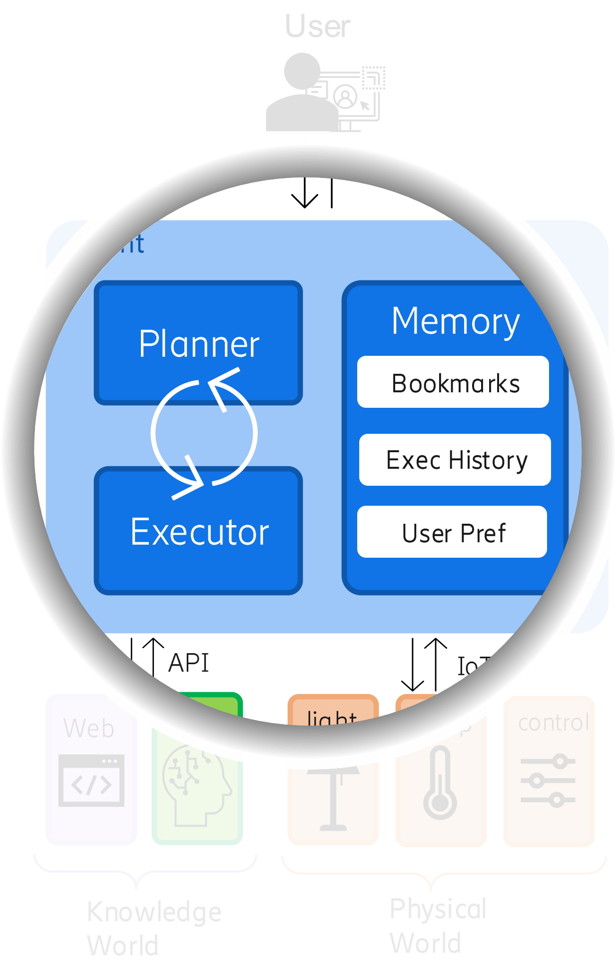
The Agent

Our agent:

- Reasoning and Acting (ReAct) **Code Agent**, returning actions as python code snippets using the function calls we have defined.
- **Model agnostic**, with pluggable LLMs.
- **Multimodal**, supports support text, vision, video and audio.

To help the agent we built **Memory** tools:

- **Bookmarking**: Maps device links (e.g., coap://) and their logical relationships.
- **Execution History**: Stores and reuses previous Python code executions for determinism and traceability.
- **User Preferences**: Maintains system behavior within user-defined boundaries in closed-loop automation.



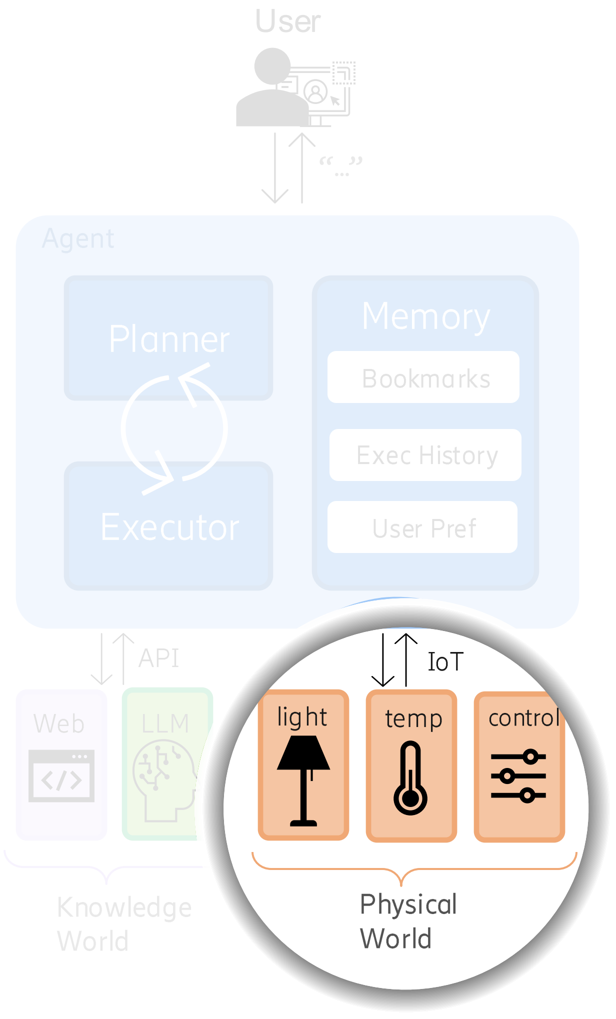
Physical system

aiocoap

- A virtualized **docker** environment with few IoT devices: temperature sensor, thermostat, air quality sensor/actuator, lamp, air purifier.
- Each device runs a CoAP server, is addressable and a simple device logic.
- Serialization is JSON and based on a simple common device data model.

Simple abstractions for the agent to interact directly over CoAP:

Feature	REST mapping	Agent Function Call
`Discover “things”`	`GET to /.well-known/core`	`list_resources(…)`
`Get current value`	`GET to resource URI`	`get_resource (…)`
`Set new value`	`PUT/POST to resource URI`	`set_resource (…)`
`Subscribe to a value`	`GET with Observe: 0`	`observe_resource(…)`



Demo

User intent demo

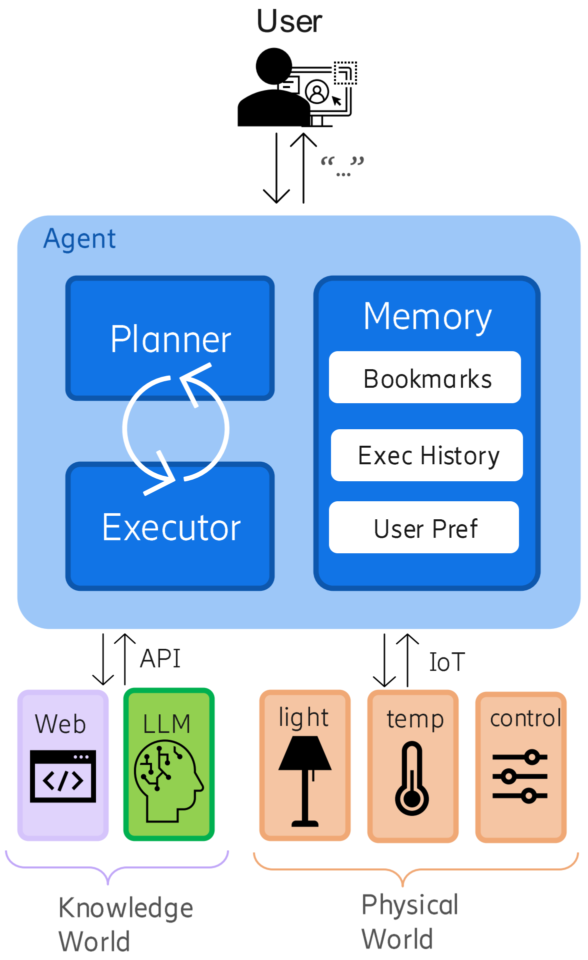
Very minimal a priori knowledge is given to the agent:

```
agent = CodeAgent(
  tools=[list_resources, get_resource, set_resource, observe_resource,
         list_observers, WebSearchTool()],
  model=model,
  planning_interval=5,
  max_steps=10
)
```

- In principle, the agent is **not IoT-specific.** It is IoT only because it uses CoAP and CoAP today is used mainly for IoT.

Knowledge comes from:

- User intents.
- Tool discovery at runtime.
- Device metadata when doing discovery.
- LLM general knowledge.



User intent demo

The system is capable of:

- Interpreting vague user intents: *"It is cold"*.
- Listing the facts and making a step-by-step plan that uses a subset of the **tools**.
- Search a knowledge base (web) in real time for current information.
- Generate executable python code using the functions…

```
# Set the light level to a higher value, e.g., 100 lx
set_light_response = set_resource(resource_name="light", value="100")
print(set_light_response)
```

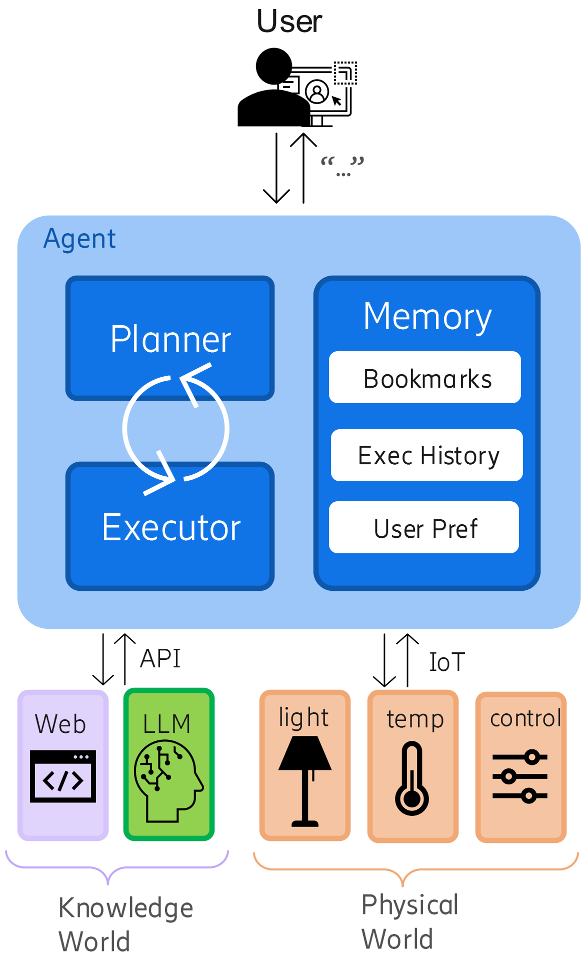
… detect execution errors in the code …

```
set_resource(resource_name="thermostat", value=22)
print("Thermostat setting increased to 22°C.")
```

```
Code execution failed at line 'set_resource(resource_name="thermostat",
value=22)' due to: AttributeError: 'int' object has no attribute 'encode'
```

… and try to fix them.

```
set_resource(resource_name="thermostat", value="22")
print("Thermostat setting increased to 22°C.")
```



User intent demo

The system is capable of:

- Find connected devices using hypermedia syntax (i.e. link attributes, rt, media types, etc) to build a set of traversed links or bookmarks.
- Discovering device capabilities in real time.

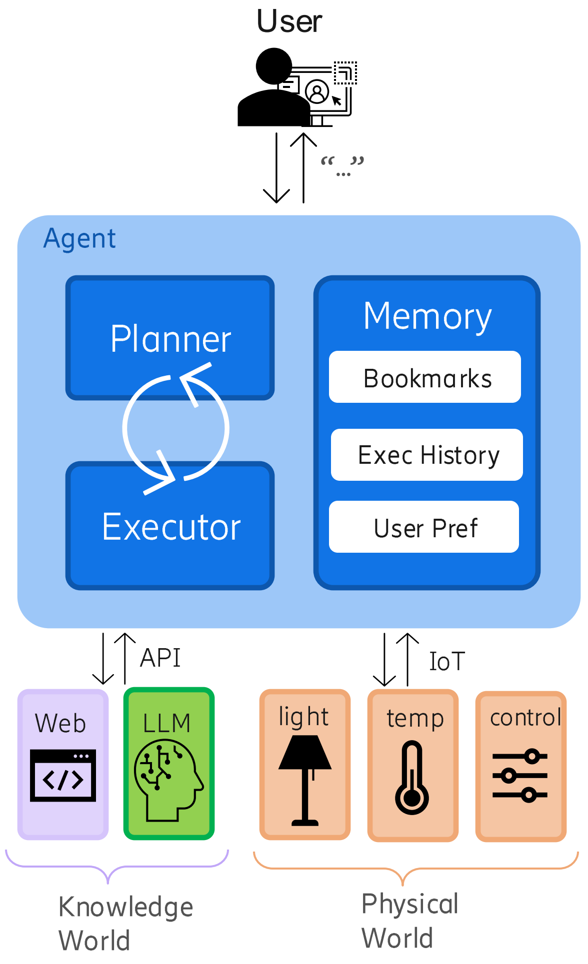
```
temperature: sensor (°C) - Type: number (Observable)
-  Description: A temperature sensor that returns Celsius degrees. The property
   is observable and can provide notifications.
-  Available requests: GET
-  URI: coap://127.0.0.1:5683/temperature
```

- Retrieving device state in real time and *incorporating* that information to the planner.

```
Current Temperature: {"n": "thermostat", "v": 20, "u": "Cel", "t": 1749737720}
```

- Observing device state until task is completed.

```
Setting up observation for temperature at coap://127.0.0.1:5683/temperature
Sending observation request for temperature
Waiting for observation responses for temperature
Resource temperature has reached target value 22
```



Continuous closed loop demo

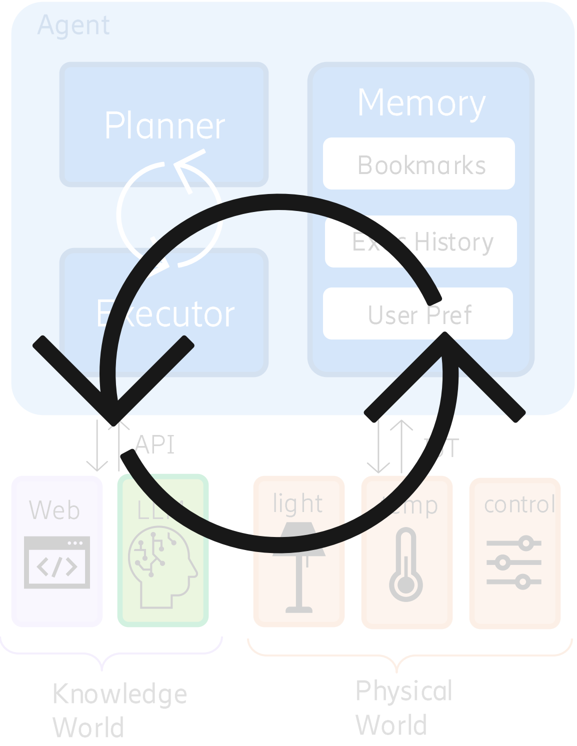
The system is capable of:

- Permanently observe devices that contain resources of relevance to the user.
- Load a user profile based on previous interactions and keep the system in equilibrium based on the user profile.

```
Resource light out of comfort zone: {'n': 'light', 'v': 0.0, 'u': 'lx', 't': 1749647426}
New run
Adjust light to bring it back to comfort zone and observe if it reaches the comfort. The comfort zone for light is from 300 to 800
PatchedAzureOpenAIServerModel - se-gpt-4o
```

- Here we do have specific configuration prompts like:

```
run_agent_in_thread(
    f"Adjust {resource_name} to match user preference. The user prefers {resource_name} to be
    {preferred_value}. Please observe to confirm if {resource_name} reached {preferred_value}",
    resource_name=resource_name
)
```



Learnings

HATEOAS-like Client for CoAP Devices

- Agents require very little **a-priori knowledge** about the IoT system.
- They can operate effectively with **unknown endpoints**, needing only an **entry point** to begin discovering and using available functionality.
- Agents are **guided by hypermedia links** encountered during interactions with the physical system.
- The **planner** (or agent controller) also leverages **real-time data** obtained from server interactions to inform decisions.
- Not tied to a specific deployment, it should work other CoAP deployments.

Agent Capabilities

- Agents are **not task-specific**; they are **proactive** and capable of general-purpose reasoning and action.
- **A minimal toolset** (discover, get, set, subscribe) over CoAP is **sufficient**, no need for extra frameworks.
- **IETF specs** are already present in the **training data of most LLMs**, helping reduce hallucinations.
- **Standardized affordances** help agents understand functionality.
- CoAP **device to device design** is very well suited for agents.

Agent Robustness

- Agents are **non-deterministic**, so their behavior may vary between runs.
- In built **memory tools** help agents to become **hybrid or deterministic** (by replaying successful prior workflows).
- Agent can recover from different errors:
  - **execution** errors → feed python interpreter error back to planner
  - **interaction** errors → feed CoAP 4.XX back to planner
  - **outcome** errors → agent keeps trying until user intent is fulfilled

Agentic Systems vs. Traditional Software

They are fundamentally different things.

- **Traditional applications** require **explicit development cycles and software updates** to support each new feature.

- In contrast, **agentic systems** can adapt to new feature requests (e.g., new intents, new devices, new device state, new knowledge base) at **runtime**, using **hypermedia** and results from **real-time interactions**.