mation to make a final choice.

API documentation retrieval tool. Retrieves documentation about a requested device's capabilities. The documentation is scraped from the web when available, otherwise it is retrieved from the device using the API. While documentation extracted from the device API is often lacking detailed natural language descriptions of usage, it contains the names of attributes, commands, and command arguments, the meaning of many of which can be inferred from the name alone. Takes as input a list of capabilities, and returns detailed documentation for each of these in JSON format. The JSON format is used for convenience, since this is the format returned by the documentation scraper and device APIs.

Device attribute retrieval tool and device command execution tool. These tools allow the agent to communicate with the API to read attributes and execute commands. We format the tool as a wrapper around SmartThings REST API to query a device's state [27]. In order to use these tools, the documentation retrieval tool must first be called in order to retrieve the capability details and format the inputs properly. Note that in the event that the inputs are not formatted properly and the API throws an exception, we have found empirically that if the text associated with this exception is propagated to the device interaction tool agent, it can often react and correct its usage accordingly.

Device disambiguation. Allows the system to resolve which devices the user wants to control in scenarios when there is more than one instance of a given device (e.g. multiple smart lights). We propose a method that can determine which device is relevant to the task by leveraging visual context. By capturing a photograph of the device within its surroundings (during setup), we can resolve the ambiguity of the device ID without requiring the user to hard-code a unique identifier to the device (which users often forget following setup and usually do not communicate to guests). For example, in Figure 5, it is obvious from the picture alone that the light is located with the dining room. The device identity is disambiguated using a Visual Language Model (VLM), as visualized in Figure 5, where a multimodal VLM is used to compute embeddings for the user's natural language description of the device and each of the device photographs. The device whose image embedding has maximum cosine similarity to the text embedding is selected.

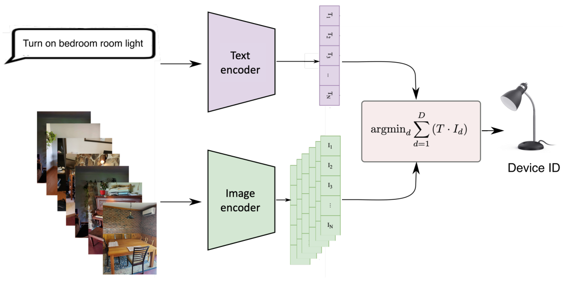
Turn on bedroom room light
Text encoder
Image encoder
T_1 T_2 T_3 ⋮ T_N
I_1 I_2 I_3 ⋮ I_N
$\text{argmin}_d \sum_{d=1}^{D} (T \cdot I_d)$
Device ID

Figure 5: **Device disambiguation**. A method to resolve the device that best fits a user natural language device description using VLMs.

5.3 Flexible persistent command handling via LLM code writing

Many of the more powerful smart home behaviors are unlocked by the ability to monitor the state of some device and react to state changes. These behaviors are referred to as persistent commands [7] or routines, as the system should *persistently* behave in a desired manner following a conditional event (e.g. the coffee machine should turn on whenever the morning alarm rings). Smart home solutions typically approach this problem using conditional statements in applications such as IFTTT. Once the specification is made, condition checking can be performed using low-cost compute resources. A drawback of this approach is the lack of flexibility afforded by the system because IFTTT routines rely on conditional triggers that must be pre-defined by the manufacturer and manually activated by the user.

Highly flexible persistent command handling could be implemented within the SAGE architecture simply by periodically running SAGE with the persistent command as input. Each time it is run, the agent could check whether the command is satisfied and if it is execute the desired behavior. This approach, which retains the full capability of the agent archi-