SAGE: Smart home Agent with Grounded Execution

Dmitriy Rivkin, Francois Hogan, Amal Feriani, Abhisek Konar,
Adam Sigal, Steve Liu, Greg Dudek

November 3, 2023

1 Abstract

This article introduces **SAGE** (Smart home Agent with Grounded Execution), a framework designed to maximize the flexibility of smart home assistants by replacing manually-defined inference logic with an LLM-powered autonomous agent system. SAGE integrates information about user preferences, device states, and external factors (such as weather and TV schedules) through the orchestration of a collection of tools. SAGE's capabilities include learning user preferences from natural-language utterances, interacting with devices by reading their API documentation, writing code to continuously monitor devices, and understanding natural device references. To evaluate SAGE, we develop a benchmark of 43 highly challenging smart home tasks, where SAGE successfully achieves 23 tasks, significantly outperforming existing LLM-enabled baselines (5/43).

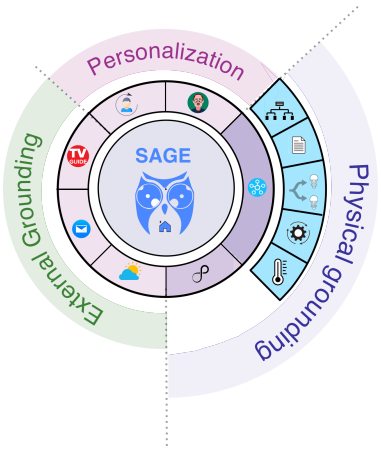

Figure 1: **SAGE - Smart home Agent with Grounded Execution.** Our system includes a collection of tools (icons described in Table 1) that can be sequentially called by the SAGE agent to accomplish a complex task. Some tools, such as the device interaction tool, follow a hierarchical structure and are themselves agents with their own set of tools, as illustrated in Figure 2.

2 Introduction

The application of LLM-based autonomous agents [1] in the smart home setting promises to revolutionize our living spaces. While voice recognition solutions have continuously improved over the last decade (Bixby, Alexa, Google Home, Siri, etc.), the planning and control of home devices have largely remained driven by the use of explicit commands such as "turn on kitchen light" and manually pre-programmed with applications such as IFTTT (If This Then That), where users specify how the system should react to a given state change through an app or online interface. These existing smart home solutions lack the flexibility to naturally interact with users, who are often required to memorize and recite specific phrases that follow a rigid syntax. This rigidity stems from the fact that the smart home planning algorithms require custom programs to be written for each use case and scenario, and are therefore unable to scale to the diverse set of needs of each user [2]. In contrast, LLM-driven autonomous agents use an LLM as the decision maker in a natural language based sequential decision making process, allowing them to react

to natural, complex instructions.

In this paper we introduce *Smart home Agent with Grounded Execution (SAGE)*, an LLM-based autonomous agent optimized for smart home applications. SAGE's long-term goal is to become an assistant that users can interact with as naturally as they would with a human companion. This requires the system to integrate three major categories of information: personal preferences, physcial grounding, and external grounding. Personalization is critical as it forms the foundation needed to understand natural and implicit requests by the user. For example, when asking a friend to "put the game on", a person assumes that their friend understands that their favorite sport is hockey, and that their favorite team is the Montreal Canadiens. Physical grounding informs the home assistant about the devices located in the user's home, their current states, and the possible actions that can be performed on them. Revisiting the "put the game on" example, a user would expect their friend to know which TV they are currently watching and put the game on that TV. Finally, *external grounding* gives the agent access to the external world outside the home such as weather information, TV schedules, emails, etc. In the "put the game on" example, the system requires access to a television schedule in order to understand if and how the request can be fulfilled.

The SAGE system architecture, shown in Figure 1, integrates a collection of tools, further detailed in Table 1, that allow it to exhibit personalization, physical grounding, and external grounding. A user command triggers a sequence of LLM queries, with the LLM's responses being used to determine which tools to employ, how to use them, and when to terminate execution. This approach to smart home automation shifts the control paradigm of home devices from a constrained setting, where users can choose among a predefined set of routines, to an unconstrained one, where users can successfully request novel behaviors that were not envisioned by the system's designers.

The main contributions of this paper are:

1. **An LLM-based agent for home automation (SAGE)** that can perform complex tasks within the home by autonomously using a sequence of tools which allow it to observe and act upon its environment.

2. **A collection of novel tools for home automation**. Our system develops a family of tools (see Table 1) that allow SAGE to be personalized and grounded.

3. **Evaluation on a real-world scenario**. We introduce a benchmark that consists of 43 highly challenging smart home tasks that test SAGE's personalization, intent resolution, device resolution, persistence, and command chaining capabilities. SAGE is shown to successfully perform 22 out of 43 of these tasks, significantly outperforming existing LLM-enabled baselines.

3 Related Work

3.1 Home Automation

A smart home system consists of connected IoT (Internet of Things) devices that enable the simultaneous monitoring, sensing, and control of the home environment. Automation of the control of these systems can lead to improvements in quality of life, comfort, and resource utilization [3]. Recent work has aimed to used machine learning to enhance the capabilities of smart home systems. For example, [4] proposed to perform home automation based on activity recognition, developing a deep learning algorithm to recognise users' activities from accelerometer data. Another focus is on voice-based home assistants, where the system is tasked with understanding users' voice utterances and executing the requested commands. For instance, [5] developed a voice controlled home automation system based on NLP (Natural Language Processing) and IoT to control four basic appliances. Leveraging advanced NLP techniques, current commercial solutions such as Bixby, Google Assistant, and Alexa offer a user-friendly interface capable of handling a variety of commands and questions from shopping and setting reminders to device control and home automation. However, these modern home assistants usually struggle with implicit and

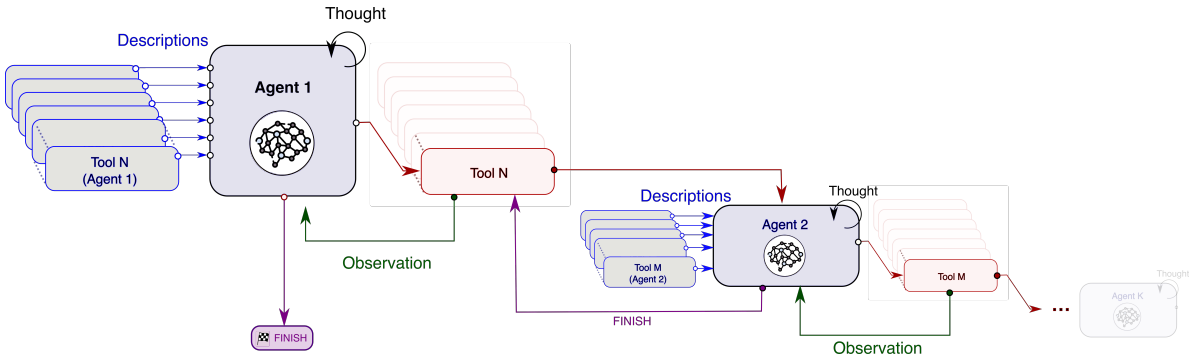

Figure 2: **LLM-based agent architecture.** Each agent is comprised of an LLM prompt template, a collection of tools, a variable that summarizes the results of previous reasoning steps of the agent, and a parser capable of interpreting the outputs of the LLM. Each tool is comprised of a single function where the inputs and outputs are both strings and an accompanying textual description that describes to the agent how to use the tool. The hierarchical nature of the system arises from the fact that some of the tools are implemented as more specialized autonomous agents themselves.

complex commands [6]. They are not capable of inferring the users' goals and acting accordingly.

In an attempt to overcome some of these challenges, recent work proposed to leverage the reasoning capabilities of LLMs to better understand and carry out user commands. In particular, Sasha [7] introduced the use of LLMs in smart home environments and showcased that the LLMs can be used to produce reasonable behaviors in response to complex or vague commands. Sasha implements a decision making pipeline where each step (such as selecting the device to use, or checking if a routine already exists) is implemented using an LLM. However, unlike SAGE, the stages of the Sasha pipeline are manually defined and fixed, limiting its flexibility. Today's smart home users also have the option of defining IFTTT-style routines, which connect trigger conditions to actions [8], in order to create complex behaviors. This approach is inconvenient because it must be implemented by users manually through an app or web interface. It is also limited by the fact that trigger conditions must be defined by device manufacturers. Users can also write their own apps to manage device states, such as SmartThings' SmartApps [9], but this requires a level of technical sophistication beyond the ability of most users, as well as a significant time investment. Web-based services such as IFTTT[1], Zapier[2], and Home Assistant [3] enable the user to create rules to control their smart devices. The advantage of these services is that they simplify the process of connecting various services and smart devices without the need for extensive programming knowledge. However, these solutions also lack reasoning and context-awareness offered by LLMs. Recently, IFTTT has begun to leverage the power of LLMs through the creation of ChatGPT plugin[4], which provide a more user-friendly and accessible interface to interact with the automation platform conversationally. However, this plugin does not allow GPT to generate new routines, but rather trigger existing ones. Zapier also offers an LLM-based offering, which allows users to describe action in a more natural way, but currently lacks significant reasoning sophistication. [7] created an LLM-based pipeline which could output trigger-action pairs to create simple IFTTT routines, but logical complexity of these routines, and well as the flexbility of the triggers, was limited. Earlier academic work investigated training

[1] https://ifttt.com/
[2] https://zapier.com/
[3] https://www.home-assistant.io/
[4] https://ifttt.com/explore/business/ifttt-ai

sequence-to-sequence models to synthesize IFTTT or Zapier routines from natural language descriptions [10]. Results were promising but somewhat limited in that the sequence models were able to generate the sequence of functions to call, but not the arguments to those functions.

3.2 Autonomous Agents

LLM-powered autonomous agents are designed to to perform complex and diverse tasks. Usually, this involves decomposing the task into multiple stages or subtasks. Several agent architecture designs have been proposed in the literature [1]. Chain-of-Thought (CoT) [11] is a well-known prompting technique that enables the agent to perform complex reasoning through step-by-step planing and acting. In the earlier CoT implementations, several CoT demonstrations are inserted in the prompt to guide the agent's reasoning process. Alternatively, zero-shot CoT [12] demonstrated the LLM reasoning capabilities by simply adding the sentence "think step by step" in the prompt. Another line of work extended CoT by adopting a tree-like reasoning structure where each intermediate step can have multiple subsequent steps (e.g., [13]). The aforementioned works did not consider feedback in the plan generation process which lead to the development of various agent designs where different feedback signals are considered. As an example, the ReAct agent [14] incorporates observations from the environment (e.g., outcomes of API calls or tools) received after taking an action. These observations are taken into consideration in the next reasoning step (i.e., thought). Human feedback can also help the agent adapt and refine its plan by asking for more details, preferences etc.

Another important part of the agent design is the use of external tools for the action execution. Tools enable the agent to go beyond its internal knowledge. APIs are the most common type of tools used in work such as Gorilla [15] and ToolLLM [16]. In addition to APIs, external knowledge bases can be used as a tool to acquire specific information or expert knowledge [17].

4 System Overview

The system described in this work is implemented as a hierarchical autonomous agent (illustrated in Figure 2), where large language models are used to coordinate the use of a multitude of tools to achieve their tasks. Each agent is comprised of an LLM prompt template, a collection of tools, a variable that summarizes the results of previous reasoning steps of the agent, and a parser capable of interpreting the outputs of the LLM. Each tool is comprised of a single function where the inputs and outputs are both strings and an accompanying textual description that describes to the agent how to use the tool. The hierarchical nature of the system arises from the fact that some of the tools are implemented as more specialized autonomous agents themselves.

We follow the prompt template proposed in the ReAct framework [14], where reasoning traces are generated in addition to task specifications. Since the output formats are conserved across all agents in this work, we make use of the standard ReAct output parser and history summarization function as implemented in LangChain [18].

We detail SAGE's main agent prompt in Figure 10, which includes a high-level description of the home automation task. The instructions are specialized for each sub-agent, as detailed further in Section 5. The prompt takes as arguments the human input, the tool names and descriptions, and the interaction history. The design parameters of the system include the behaviors of the tools, the descriptions of how to use the tools, the descriptions of the agents tasks, and the organization of the agent-tool hierarchy. The full agent-tool hierarchy used in this work is shown in Figure 1. The following section describes in detail the design of the individual tools.

5 Tools

In this section, we introduce a collection of novel tools developed for SAGE, see Table 1 for a comprehensive list and their descriptions. These tools span 4 categories: personalization, device interaction, device disambiguation, and persistent commands. SAGE also

Tool	Description
 Device interaction	*Use this to interact with smartthings. Accepts natural language commands. Works best with more detail. Does not know the names of users, so you can't refer to concepts like "Bob's kitchen."*
 Device interaction planner	*Used to generate a plan of API calls to make to execute some command.*
 API documentation retrieval	*Use this to get more details about specific components and capabilities of different devices. Using this tool before interacting with the API will increase your chance of success. Input to the tool should be a json string comprising a list of dictionaries. Each dictionary in the list should contain two keys: device_id (guid string), capability_id (str)*
 Device attribute retrieval	*Use this to get an attribute from a device capability. Input to the tool should be a json string with 4 keys: device_id (str), component (str), capability (str), attribute (str)*
 Device disambiguation	*Use this to pick the right device from a list of candidate devices. Works best when you describe the device and its location. Input to the tool should be a json string with 2 keys: devices (list of guid strings), disambiguation_information (str): device type and its surroundings*
 Device command execution	*Use this to execute a command on a device. Input to the tool should be a json string with 5 keys: device_id (str), component (str), capability (str), command (str), args (list)*
 Personalization	*Learn about the user preferences. The query should be clear, precise and formatted as a question. This tool is not capable of asking the user anything. Try using this tool before addressing subjective or ambiguous user commands.Input should be a json string with 2 keys: query and user_name.*
 Human interaction	*Use this tool to communicate with the user. This can be to interact with the user to ask for more information on a topic or clarification on a previously requested command.*
 Email and calendar	*Use this tool to perform actions with user's Gmail and Google calendar account, and get contacts' names and email addresses. This tool accepts natural language inputs. Do not specify the user in the query, assume it is known.*
 Weather	*Use this tool to get weather information of a given place. Always structure the query in the following format <City, Country>*
 TV schedule retrieval	*Search for programming currently playing on the TV. Input a json string with keys*
 Condition code writing	*Use this tool to check whether a certain condition is satisfied. Accepts simple natural language commands.Returns the name of a function which checks the condition. Inputs should be phrased as questions.*

Table 1: Tools used by SAGE agent along with written descriptions that inform the agent how to use the tool.

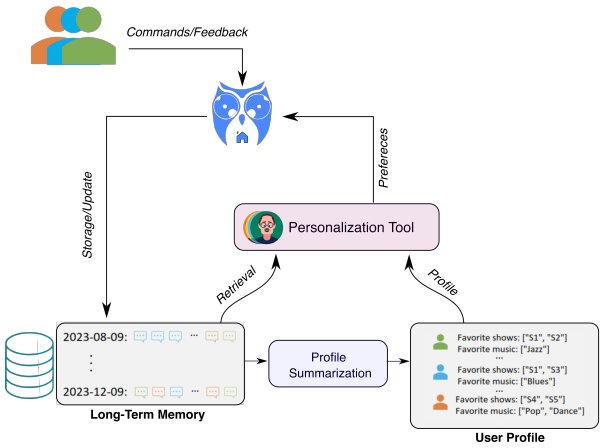

Figure 3: An overview of the personalization module.

integrates several other tools for external grounding, including a TV schedule search tool, a weather tool, and an email and calendar tool. We do not discuss these in detail because tools of this type have been well explored in the literature, and are readily available in libraries such as LangChain.

5.1 Personalization

This section introduces a personalization tool that enables the system to adapt and respond in a manner that is consistent with its user's preferences. The personalization tool, illustrated in Figure 3, is composed of two main sub-components: (i) the long-term memory that stores users' interactions, and (ii) the user profiler that constructs a hierarchical understanding of users' preferences.

5.1.1 Long-Term Memory

Long-term memory [1], shown in Figure 3, stores information about the user's past interactions and behavior. The memory records commands or feedback in chronological order and is used for retrieval by the agent to make decisions that are better aligned with the individuality of its user. Similar to information retrieval techniques for LLM augmentation summarized in [19], the long-term memory is used to retrieve memories relevant to the user query in order to augment the in-context information available to the personalization tool. Each entry in the memory is encoded using a dense retrieval embedding model (e.g., [20]). The vector representations are then indexed and stored in a vector database. In this work, we use the MiniLM embedding model [21].

5.1.2 User Profiler

The user profile, shown in Figure 3, provides a high-level summary of the interactions between the users and the agent to build a dynamic and holistic understanding of the users' preferences. We adopt a hierarchical approach, first proposed in [22], to build the users' profiles. The user profiler starts by splitting all the entries in the long-term memory by day and generates daily summaries to capture all the nuances of the users' preferences. Next, all the daily summaries are aggregated into a single global overview serving as the user profile. Our choice of a hierarchical approach is motivated by two main reasons: (i) scalability: as the long-term memory grows with time, a hierarchical approach is highly scalable because it is amenable to MapReduce-style processing [23], (ii) information loss: directly generating a concise summary from the long-term memory involves a long-context prompt. LLMs' ability to successfully retrieve and identify relevant information within the input context is known to degrade as the length of the input context increases [24].

5.1.3 Personalization tool

The personalization tool (depicted in Figure 3) combines the long-term memory and the user profiler to allow the SAGE agent to query them with questions about the user's preferences. When queried with a question about the user's preferences, this tool first encodes the question using the dense retrieval embedding model and similar memories are retrieved from the long-term memory, using cosine similarity in the embedding space as the distance metric. Next, the tool constructs an LLM prompt consisting of the retrieved memories, the user profile, and the question. Finally, it queries the LLM with the prompt and returns the response. The user profile and the

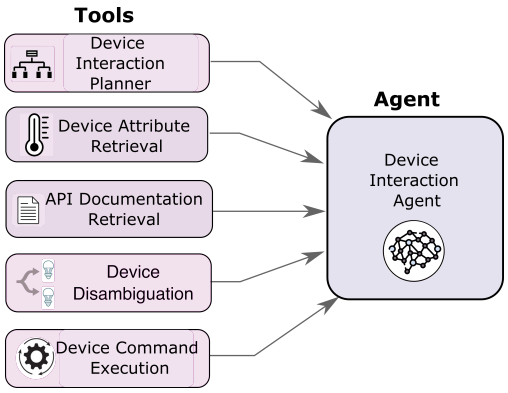

Figure 4: Device interaction agent architecture.

retrieved memories are complementary: the retrieval module identifies a small pool of candidate memories that provide narrow and precise information, while the user profile presents a more holistic view of the user. The LLM may draw upon both sources of information in answering the question about the user. Multiple users are supported by maintaining separate profiles and long-term memories for each. User identity is recognized using voice recognition software [25].

5.2 Device Interaction

In this section, we introduce the tool that SAGE uses to interact with smart devices. This tool enables flexible device interaction that is scalable, such that any device can be integrated into the system with minimal effort and its capabilities can be leveraged to the fullest extent. For example, if a user buys a smart fridge and adds it to their smart home, SAGE can integrate the presence of the fridge and all of its capabilities (e.g. temperature settings, door open detection, power consumption report, etc.) into its decision making process without the need for any fridge-specific code to be written.

Most smart home providers (SmartThings, Home-Assistant, Google Home, Alexa, etc.) provide APIs for interacting with the smart devices in users homes. These APIs are partially documented online, and code examples for using them are available, meaning that LLMs trained on web data (such as GPT-4) are likely to have some inherent knowledge of these APIs. In practice, we have found that LLMs often fail to use these APIs successfully due to minor errors such as forgetting the exact names of the attributes they need to retrieve. Furthermore, some devices have custom components for which no documentation is available online, and can only be retrieved by querying the device's API. These challenges can be overcome if details of API usage are injected into the prompt, but injecting the full documentation for all connected devices in not feasible, as it can amount to tens or hundreds of thousands of tokens, far exceeding the maximum prompt length of today's LLMs.

Motivated by LangChain's OpenAPI toolkit [26], the device interaction tool is implemented as an agent. This agent generates a high-level plan using only a high-level description of devices and their associated capabilities, retrieves detailed documentation for the subset of capabilities that are required by the plan, and uses these to construct API calls. This behavior is enabled through a collection of tools, detailed below.

Device interaction planner tool. Generates a sequence of steps that must be performed by device interaction tool agent to complete the given command. It is implemented using a single LLM query, exemplified in Figure 11. This query includes a list of devices, a full list of their capabilities, and a list of short descriptions of what each capability does. It also includes the input command and a description of how the plan should be structured. The query specifies that each step of the generated plan should include one or more device IDs, one or more capabilities, and a natural language description of what needs to be done in that step. A single step of the plan may include multiple devices if the planner cannot directly infer from the information it has which device the user was referring to. The agent can use the device disambiguation tool, detailed below, to resolve these ambiguities. The inclusion of multiple candidate capabilities in a step of the plan means that the planner cannot figure out exactly which one to use based on the short descriptions. In this case, the agent can retrieve detailed documentation for all of the proposed capabilities and then use that infor-

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

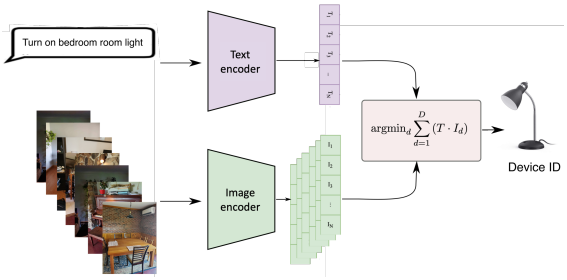

Figure 5: **Device disambiguation**. A method to resolve the device that best fits a user natural language device description using VLMs.

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.

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-

tecture and is simple to implement, has the downside of requiring the agent to constantly be running, incurring significant computational costs.

In order to increase the system's flexibility while minimizing cost, we propose a method by which SAGE can autonomously program conditional routines by enabling it to write python code that implements the condition checking logic. This method is summarized in Figure 6. Two tools are introduced to support this functionality: a condition code writing tool and a condition polling tool. The SAGE agent queries the condition code writing tool to write the necessary code, then registers this code with the condition polling tool, which runs it periodically. Along with the condition checking code, it also registers a description of the action that must be taken when the condition is met. Once the code returns "True", the polling process triggers a second execution of the SAGE agent with the command registered with it by the first execution.

The implementation of the condition code writing tool is complicated by the same challenge as the device interaction tool – the requirement to inject API details into the query that generates the code. We also overcome this challenge in a similar fashion by creating an agent which uses the device interaction planner tool and the API documentation retrieval tool. However, instead of a the device attribute retrieval and command execution tools, the condition code writing tool agent has access to a code execution tool which allows it to test its code. This tool also stores the code it has run in memory, so that it can be referred to by the name of the function. Similarly to the device attribute retrieval and command execution tools, the code execution tool handles exceptions by returning their messages to the to the code writing tool agent, facilitating recovery from faulty code.

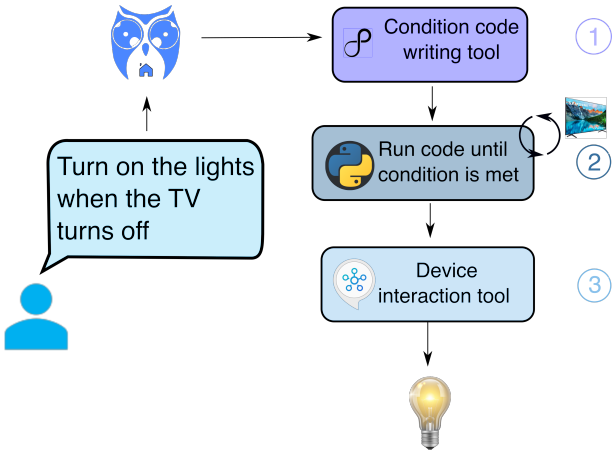

Figure 6: A summary of the persistent command handling mechanism. **1:** After receiving the user request, the SAGE agent extracts the condition ("is the TV off?") and uses the condition code writing tool to write code to check this condition. The tool registers this code in memory and responds with the name of the function (is_tv_off). **2:** The SAGE agent registers the function is_tv_off with the condition polling tool, along with the command reflecting the action to take once the condition is detected "turn the lights on". At this point the SAGE agent finishes executing. The condition polling tool periodically runs the is_tv_off function. Once the function outputs True, the condition polling tool triggers the SAGE agent again with the registered action "turn the lights on." Note that the agent will only be triggered when the status of the action transitions from False to True to avoid rerunning the agent the entire time the TV is off. **3:** The agent begins executing with the command "turn the lights on" and turns on the lights using the device interaction tool.

5.3.1 Working with other APIs.

The device interaction tool need only be capable of interaction with the SmartThings API, since interactions with other APIs (e.g. a weather API) can be handled by the top level agent through other tools. However, the code writer tool must have knowledge of all APIs required to check the condition. Luckily, high-performance LLMs such as GPT-4 are sometimes capable of writing correct code without reference material, especially for simple, popular APIs. However, we feel that the capability of the code-writing approach would be greatly expanded through the addition of a "code search" style functionality that would allow the code-writing agent to interactively access documentation for thousands of open

APIs. There are several projects pursuing this kind of capability (e.g. [28, 29]), and we believe that the integration of these ideas into SAGE is a promising avenue for future research.

5.4 Human interaction tool

The human interaction tool allows SAGE to ask the user questions, which it usually uses to clarify intent. Empirically, we have found that the introduction of the user interaction tool can cause the agent to become over-cautious, using this tool over other data sources to reduce uncertainty. We use prompt engineering in SAGE to encourage it not to over-use the tool, but how to best trade-off personalization, human interaction, and risk aversion is topic of active development.

6 Evaluation

To evaluate SAGE, we created a dataset of 43 test tasks. Test tasks are implemented by initializing device states and memories, running the home assistant with an input command, then evaluating whether the device state was modified appropriately. For tasks which involve answering user questions, the test code checks for the presence of specific sub-strings in the answer. These results of these tests are binary (pass/fail). To gain a better sense of the reasons for failure, we also manually analyze the results of the tests.

We classify the test cases according to five types of challenge that are difficult for existing systems to handle. Each test belongs to one or more of these categories. The categories are:

1. **Personalization**: Forming a response that it tailored to the user.

2. **Intent resolution**: Understanding unstructured commands and drawing logical conclusions.

3. **Device resolution**: Identifying the target device based on natural language description.

4. **Persistence**: Handling commands that require persistent monitoring of system states.

5. **Command chaining**: Parsing a complex command that consists of multiple instructions, breaking it into actionable parts and executing each parts in a coherent manner.

The test cases and their categories are summarized in Table 2. They are designed to be fully automated, without a human in the loop. For this reason, we disable the human interaction tool during testing.

6.1 Baselines

To contextualize SAGE's performance, we compare our method to two LLM-based baselines on our test tasks. The first method, called *one prompt*, involves creating a single prompt comprised of the states of all the devices and the command, and asking the LLM to generate updated states in response to the user query. The full device state, serialized to JSON format, exceeds GPT-4s token limit (8000 tokens), so we manually selected the parts of the device state involved in the tests. In addition, the model was asked to output the changes that need to be made, not the full new state.

The second baseline, called "Sasha," implements the pipeline described in [7], with some modifications. The original pipeline in [7] consists of 5 pipeline states – clarifying, filtering, planning, feedback, and execution. The clarifying and feedback stages required human intervention, and were thus not compatible with our fully automated testing framework, so they were removed in our implementation. Additionally, this pipeline distinguishes between "sensors" and actionable devices, allowing the pipeline to output sensor-based trigger-action pairs to handle persistent commands. This requires the manual definition of triggers, which our testing framework does not provide, since SAGE is able to generate its own triggers by writing code. As such, our implementation of Sasha does not include the trigger concept, and is therefore unable to handle persistent commands.

Both of these baselines are handicapped in that they are not able to access as much information as SAGE (e.g. user preferences, photos of the devices,

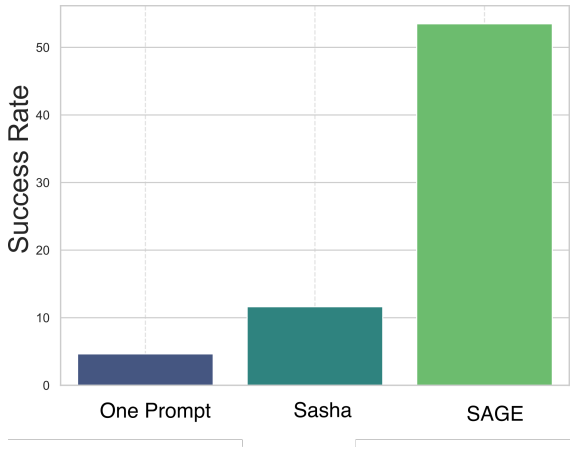

Figure 7: Overall success rates on 43 challenging tasks. SAGE leverages a collection of tools that allow it to integrate a large amount of grounding information into its decision making process, allowing it to outperform other LLM-based methods by a significant margin.

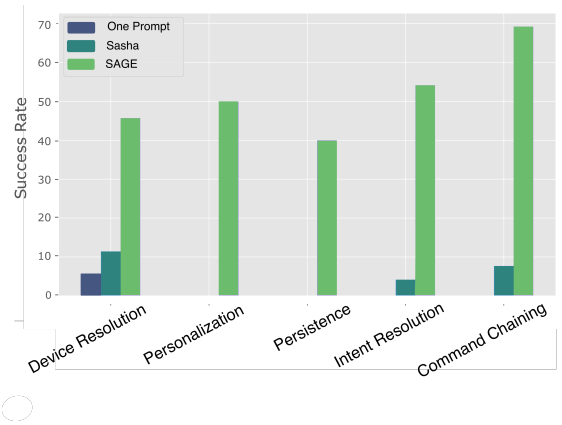

Figure 8: Success rate per task challenge type.

etc). Despite this, the baselines allow the reader to gauge the difficulty of the task set, and to appreciate the extent to which integrating information from a variety of sources can improve the performance of smart home automation systems. We do not provide a baseline with access to the same information as SAGE as, to our knowledge, there isn't any previous work that is capable of integrating all of these information sources.

7 Results

Overall success rates for the three methods, SAGE, one prompt, and Sasha, are presented in Figure 7, and success rates per task challenge type are presented in Figure 8. SAGE achieves an overall success rate of 51%, far beyond either of the baselines, demonstrating that it is indeed capable of integrating a variety of information sources through the use of its tools.

To further understand the reasons for failure in SAGE, we manually analyzed the failing cases, categorizing the failures according to the component causing the issue. These results are summarized in Figure 9, where each bar represents the ratio of tests failing due to failure of the component vs total tests using the component. Note that it is common for components to be reused multiple times within the execution of a single command, but these multiple uses are not reflected in Figure 9. Figure 9 shows that by far the least reliable part of the system is the condition code writing tool, which also helps to explain why persistent commands had the lowest success rate in Figure 8. We believe this to be in part due to issues with prompt length – each time the code writing agent makes a mistake which leads to an exception in the code, a copy of the code is added to the action history. Code takes up a relatively large number of tokens, and as the amount of code in the prompt becomes large the reasoning abilities of the model seem to degrade (as documented in [24]). Device disambiguation could also be improved, for example by using a better CLIP model or by using a captioning-based solution.

8 Conclusion

This article introduced SAGE, a grounded agent targeted at smart home applications. SAGE achieves its grounding by orchestrating the use of tools in a sequential decision making process. To enable SAGE to

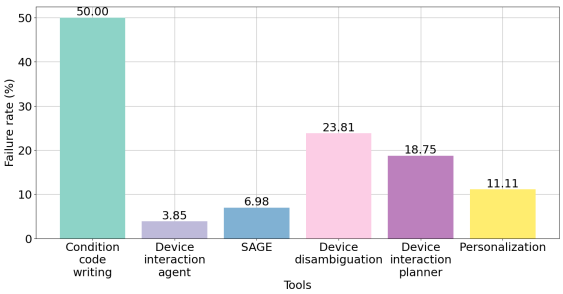

Figure 9: Failure rates for different components of the system. Failure rate is computed as the ratio of the number of tests which failed due to a failure in the given component and the total number of tests using the component. Failures are attributed to components by manual analysis.

be grounded with respect to user preferences, devices, and the external world, we devised and presented several novel tools. We also created a dataset of challenging smart home automation test cases which tested the system's ability to be personalized, to resolve users intents from unstructured queries, to resolve devices referred to in natural ways, and to command persistence and chaining. achieved a success rate of around 51% on these tasks. This value, while far from perfect, is much higher than that achieved by existing LLM-based home automation solutions. Each success required the successful sequential application of many tools, so in fact the number of successful tool uses is much larger than the number of failed ones. As such, SAGE represents a highly promising first step towards the creation of truly flexible smart home agents that users can interact with as naturally as they would with a close friend.

9 Appendix

You are an agent that assists with queries against some API.
Instructions:
 - Include a description of what you've done in the final answer, include device IDs
When reading or manipulating TV content, prefer using channel numbers.

Here are the tools to plan and execute API requests:
{tool_descriptions}

Starting below, you should follow this format:

User query: the query a User wants help with related to the API
Thought: you should always think about what to do
Action: the action to take, should be one of the tools [{tool_names}]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I am finished executing a plan and have the information the user asked for or the data the user asked to create
Final Answer: the final output from executing the plan
Your must always output a thought, action, and action input.

Begin!

User query: {input}
Thought: I should generate a plan to help with this query.
{agent_scratchpad}

Figure 10: Device interaction tool prompt.

You are a planner that helps users interact with their smart devices.
You are given a list of high level summaries of device capabilities ("all capabilities:").
You are also given a list of available devices ("devices you can use") which will tell you the name and ID of the device, as well as listing which capabilities the device has.

Your job is to figure out the sequence of which devices and capabilities to use in order to fulfill the user's request.
- If the user query cannot be performed on the devices explain why
- Restate the query 3 different ways
- The capability information you receive is not detailed. Often there will be multiple capabilities that sound like they might work. You should list all of the ones that might work to be safe
- It is often unclear exactly which device / devices the user is referring to. If you don't know exactly which device to use, list all of them, and provide a strategy for how to figure out the right one
- Include device IDs (guid strings), capability ids, and explanations of what needs to be done in your plan.
- When planning to modify device states, don't assume the devices are already on

Often a good strategy to figure out which device the user is talking about is to check which one is currently on.

all capabilities: {one_liners_string}
devices you can use: {device_capability_string}
user query: {query}

Figure 11: Device interaction planner tool prompt.

Table 2: All 43 test cases and associated challenge types.

Query	**Personalization**	**Intent resolution**	**Device resolution**	**Persistence**	**Command chaining**
Turn on the tv			✓		
What channel is playing on the tv?			✓		
Turn on the light by the bed			✓		
What is the current phase of the dish washing cycle?			✓		
Lower the volume of the tv by the light			✓		
What is the current temperature of the freezer?			✓		
Is the tv by the credenza on?			✓		
Play something for the kids on the tv by the plant		✓	✓		
Play something funny on the tv by the plant		✓	✓		
Set up lights for dinner		✓	✓		
Set the lights in the bedroom to a cozy setting	✓	✓	✓		
Darken the entire house		✓	✓		✓
Turn off all the lights that are dim			✓		✓
I am getting a call, adjust the volume of the tv		✓	✓		
Dishes are too greasy set an appropriate mode in the dishwasher.		✓			
Put something informative on the tv by the plant.	✓	✓	✓		
Change the lights of the house to represent my favourite hockey team. use the lights by the tv, the dining room and the fireplace.	✓	✓	✓		✓
I am going to sleep. change the bedroom light accordingly.	✓	✓	✓		
Turn off all the tvs and switch on the fireplace light			✓		✓

Continuing to the next page

Query	Personalization	Intent resolution	Device resolution	Persistence	Command chaining
Remind me to throw out the milk when the i open the fridge				✓	
Turn on light by the nightstand when the dishwasher is done			✓	✓	
Increase the volume of the tv by the credenza whenever the dishwasher is running			✓	✓	
Put the game on the tv by the credenza (User 1)	✓	✓	✓		
Put the game on the tv by the credenza (User 2)	✓	✓	✓		
Its been a long, tiring day. can you play something light and entertaining on the tv by the plant	✓	✓	✓		
Heading off to work. turn off all the non essential devices.		✓	✓		✓
It is too bright in the dining room.		✓	✓		
Setup a Christmassy mood by the fireplace.		✓	✓		
When the tv turns off turn on the light by the bed			✓	✓	
Move this channel to the other tv and turn this one off		✓	✓		✓
Put the game on the tv by the credenza and dim the lights by the tv	✓	✓	✓		✓
I am going to visit my mom. should i bring an umbrella?	✓				
If my mother is scheduled to visit this week, turn on national geographic on the tv by the credenza	✓		✓		✓
Turn on the light			✓	✓	✓
What did i miss?	✓	✓	✓		✓
What is my mother's email address?	✓	✓	✓		✓
Create a new event in my calendar - meeting with User 2 tomorrow at 4pm	✓				
When am I next scheduled to see my mother?	✓	✓			

Continuing to the next page

Query	Personalization	Intent resolution	Device resolution	Persistence	Command chaining
What does next week look like?	✓	✓			
Summarise the last email I received. Send the summary to User 3 via email.	✓				✓
If my father is not scheduled to visit next week, compose an email draft inviting him to come over one afternoon during that week.	✓				✓

References

[1] L. Wang, C. Ma, X. Feng, Z. Zhang, H. ran Yang, J. Zhang, Z.-Y. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. rong Wen, "A survey on large language model based autonomous agents," *ArXiv*, vol. abs/2308.11432, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261064713

[2] M. J. Kim, M. E. Cho, and H. J. Jun, "Developing design solutions for smart homes through user-centered scenarios," *Frontiers in psychology*, vol. 11, p. 335, 2020.

[3] "Can an intelligent personal assistant (ipa) be your friend? para-friendship development mechanism between ipas and their users," *Computers in Human Behavior*, vol. 111, p. 106412, 2020.

[4] R. D. Manu, S. Kumar, S. Snehashish, and K. Rekha, "Smart home automation using iot and deep learning," *International Research Journal of Engineering and Technology*, vol. 6, no. 4, pp. 1–4, 2019.

[5] P. J. Rani, J. Bakthakumar, B. P. Kumaar, U. P. Kumaar, and S. Kumar, "Voice controlled home automation system using natural language processing (nlp) and internet of things (iot)," in *2017 Third International Conference on Science Technology Engineering & Management (ICONSTEM)*, 2017, pp. 368–373.

[6] E. Luger and A. Sellen, "Like having a really bad pa: The gulf between user expectation and experience of conversational agents," in *Proceedings of CHI 2016*, 2016.

[7] E. King, H. Yu, S. Lee, and C. Julien, "Sasha: creative goal-oriented reasoning in smart homes with large language models," *arXiv preprint arXiv:2305.09802*, 2023.

[8] H. Yu, J. Hua, and C. Julien, "Dataset: Analysis of IFTTT recipes to study how humans use internet-of-things (iot) devices," *CoRR*, vol. abs/2110.00068, 2021.

[9] "Smartthings smartapp documentation," https://developer.smartthings.com/docs/connected-services/create-a-smartapp, accessed: 2023-10-27.

[10] D. Dalal and B. V. Galbraith, "Evaluating sequence-to-sequence learning models for if-then program synthesis," *CoRR*, vol. abs/2002.03485, 2020. [Online]. Available: https://arxiv.org/abs/2002.03485

[11] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[12] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[13] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *arXiv preprint arXiv:2305.10601*, 2023.

[14] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[15] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," *arXiv preprint arXiv:2305.15334*, 2023.

[16] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, R. Tian, R. Xie, J. Zhou, M. Gerstein, D. Li, Z. Liu, and M. Sun, "Toolllm: Facilitating large language models to master 16000+ real-world apis," 2023.

[17] Y. Ge, W. Hua, J. Ji, J. Tan, S. Xu, and Y. Zhang, "Openagi: When llm meets domain

experts," *arXiv preprint arXiv:2304.04370*, 2023.

[18] H. Chase, "Langchain," https://github.com/langchain-ai/langchain.

[19] Y. Zhu, H. Yuan, S. Wang, J. Liu, W. Liu, C. Deng, Z. Dou, and J.-R. Wen, "Large language models for information retrieval: A survey," *arXiv preprint arXiv:2308.07107*, 2023.

[20] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," *arXiv preprint arXiv:2004.04906*, 2020.

[21] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," *CoRR*, vol. abs/2002.10957, 2020.

[22] W. Zhong, L. Guo, Q. Gao, and Y. Wang, "Memorybank: Enhancing large language models with long-term memory," *arXiv preprint arXiv:2305.10250*, 2023.

[23] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107–113, jan 2008.

[24] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *arXiv preprint arXiv:2307.03172*, 2023.

[25] "Vosk speech recognition toolbox," https://github.com/alphacep/vosk-api, accessed: 2023-10-27.

[26] "Langchain openapi toolkit," https://python.langchain.com/docs/integrations/toolkits/openapi, accessed: 2023-10-26.

[27] "Smartthings api," https://developer.smartthings.com/docs/api/public, accessed: 2023-10-26.

[28] "Sourcegraph cody ai," https://github.com/sourcegraph/cody, accessed: 2023-10-26.

[29] "Gpt engineer," https://github.com/AntonOsika/gpt-engineer, accessed: 2023-10-26.