AIoT Smart Home via Autonomous LLM Agents

Dmitriy Rivkin, Francois Hogan, Amal Feriani, Abhisek Konar, Adam Sigal, Xue Liu, Gregory Dudek

***Abstract*—The common-sense reasoning abilities and vast general knowledge of Large Language Models (LLMs) make them a natural fit for interpreting user requests in a Smart Home assistant context. LLMs, however, lack specific knowledge about the user and their home, which limits their potential impact. SAGE (Smart Home Agent with Grounded Execution), overcomes these and other limitations by using a scheme in which a user request triggers an LLM-controlled sequence of discrete actions. These actions can be used to retrieve information, interact with the user, or manipulate device states. SAGE controls this process through a dynamically constructed tree of LLM prompts, which help it decide which action to take next, whether an action was successful, and when to terminate the process. The SAGE action set augments an LLM's capabilities to support some of the most critical requirements for a Smart Home assistant. These include: flexible and scalable user preference management (*"Is my team playing tonight?"*), access to any smart device's full functionality without device-specific code via API reading (*"Turn down the screen brightness on my dryer"*), persistent device state monitoring (*"Remind me to throw out the milk when I open the fridge"*), natural device references using only a photo of the room (*"Turn on the lamp on the dresser"*), and more. We introduce a benchmark of 50 new and challenging smart home tasks where SAGE achieves a 76% success rate, significantly outperforming existing *LLM-enabled* baselines (30% success rate).**

***Index Terms*—Autonomous LLM Agents, Smart Home, IoT, Generative AI, Embodied AI, Personalized AI, AI Assistant.**

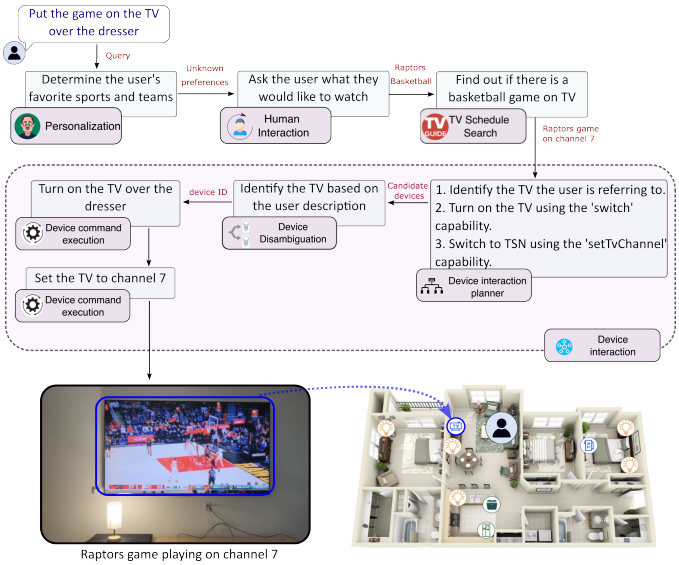

Fig. 1. **Demonstration of SAGE response process to user command: *"Put the game on by the dresser"*.** The figure illustrates the sequence of tools used by SAGE to complete the task. Note that each control flow decision was made by an LLM, not by hand-coded logic. The control flow is described step by step in Section V. This demo was executed using real SmartThings-enabled devices.

I. Introduction

Smart home assistants have become increasingly flexible and capable in recent years [31], [32]. However, despite significant advancements, there remains a chasm between the capabilities of smart home assistants and those of human users [33]–[35]. The limitations of existing smart home assistants can be broken down into three categories: (1) difficulty interpreting unconstrained natural language of user commands, (2) limitations in interaction with its environment (i.e. devices) and external data sources, and (3) lack of knowledge of the user's habits and preferences. Overcoming these shortcomings would represent a significant step forward for smart home assistants, bringing them closer to matching the capabilities of a human assistant. To address these issues, we present SAGE (Smart Home Agent with Grounded Execution).

When making a request to a friend, they may use language which is imprecise, but of which the meaning is inherently understood by both parties. This is possible because humans in this situation have knowledge of the other person, the environment, and the world in general, allowing them to infer intent based on context. For example, it is trivial to ask a friend to *"Turn on the TV over the dresser"*, but doing so with a smart home assistant necessitates some manual set-up. Existing smart home systems (e.g. Bixby, Alexa, Google Assistant, etc.) need users to define device names for the assistant to be able to interpret commands relating to their smart devices. Understanding of unconstrained natural language in the context of smart home assistants has seen great progress with the rise of large language models (LLMs), such as GPT4. LLMs are powerful language models which are trained on vast amounts of text scraped from across the internet [36]. As such, they have exceptional performance on diverse text generation tasks from writing code to poetry, without the need for finetuning [37]. In light of this, recent approaches such as [2] leverage LLMs in a fixed decision-making pipeline to interpret human commands, and output changes to be made to the smart home in a structured text format.

LLMs even enable the interpretation of open-ended, subjective requests which require creative reasoning (e.g. *"Set up the lights for Halloween"*). Without prior manual setup by the user, existing smart home systems are unable to carry out such a request in any meaningful way. Conversely, an LLM-enabled smart home assistant could respond by putting on spooky music and setting the lights to a dim orange, approaching how a human might respond in the same context.

However, interpreting and carrying out user commands is not a one-size-fits-all task; using a fixed process for all user commands is inflexible. Consider the steps needed to carry out

This work was done while the authors were with the Samsung AI Center Montréal. Corresponding author: dmitriy.rivkin@independentrobotics.com.

commands such as *"Tell me the current time"*, as compared with *"Put the game on by the dresser"*. The first command is a simple and unambiguous request, while the second requires the ability to understand imprecise language and the intricate control of the user's smart devices. This issue motivates our use of an "autonomous LLM agent" [1] strategy because it allows for a higher degree of LLM autonomy than previous, more rigid frameworks, such as [2] and [3] (see Section II).

In the autonomous agent setting, the agent generates a plan for addressing the task through chain-of-thought reasoning [4], and chooses which *tools* to use at each step of the process. These tools have both inputs (arguments) and outputs. At each step of the command execution process, the LLM is prompted to decide which tool and arguments to use. Internally, a tool may query a database, control a device, interact with the user, or even spawn a separate, specialized autonomous LLM agent to carry out specific types of tasks. When the tool finishes execution, the outputs of the tool are used to construct the subsequent prompt. In this framework, LLM augmentation is achieved through the construction of tool selection prompts, tool interfaces, tool implementations, and tool output formats. With these augmentations, SAGE is able to overcome inherent shortcomings of LLMs such as the lack of specific knowledge about the user, their home, and their devices. The overall approach is described in more detail in Section III. The most important tools are briefly introduced in the following paragraphs, and described in detail in Section IV.

To address the aforementioned issue of ambiguous user command phrasing, we present the *device disambiguation tool*, which is used to interpret implicit user intent in such situations, with minimal manual set-up. For example, this allows SAGE to understand which device the user means when saying "Turn on the TV over the dresser", without defining a device name.

Existing smart home solutions require manually written code to define the interaction between the assistant and devices. Due to the wide range of devices available on the market, it becomes infeasible for the developers of the assistant to tightly integrate the full spectrum of functionality provided by each one. We equip SAGE with the *device interaction tool*, which leverages the ability of LLMs to generate structured text to make API calls for fine-grained control of diverse smart devices.

In existing smart home systems, users can use simple conditional logic to create persistent commands (also known as automations or routines) for their devices, such as those used in IFTTT[1] (If This Then That). These allow a certain set of actions to be performed in response to a condition being met. Existing smart home systems' restricted complexity of device interaction leads to similarly restricted options for persistent commands and their trigger conditions, especially when doing so by interacting with the voice assistant. In SAGE, this is not only alleviated by the device interaction tool's ability to exploit device APIs more fully, but also by a set of tools for *persistent command handling via code-writing*. This set of tools leverages the ability of LLMs to generate code, allowing for the creation of complex triggers, which in turn enable more elaborate persistent commands. In existing systems, these must be defined manually, usually using a rigid proprietary app. SAGE's code writing tool is therefore both easier to use and more powerful in comparison.

The more an assistant understands a user's preferences, the more it can understand the user's intent and tailor its responses accordingly. However, existing smart home assistants are limited in their understanding of their users' preferences, which in turn limits the ways they can leverage these preferences in carrying out their tasks. To address this, we propose the *personalization tool*, which is the first LLM memory augmentation approach to be tailored to the smart home context, allowing SAGE to leverage global information about the user, as well as finer details relevant to the current context.

The upshot of the integration of these tools within the SAGE design is a decision making flow controlled entirely by an LLM but not limited by its inherent shortcomings, enabling an unprecedented level of flexibility and naturalness in users' interaction with their smart homes. An example of such a decision process is illustrated in Figure 1. The key takeaway of this example is that each step was decided by an LLM, with no manually defined decision logic whatsoever. Because no manual decision logic exists, users are not limited to the use cases envisioned by the developers of such a smart home assistant. As such, SAGE represents a significant paradigm shift from existing smart home systems and even more recent advancements which make use of LLMs (e.g. [2]).

Since SAGE is equipped with a range of functionalities which are not present in existing smart home systems, we present a new benchmark to appropriately evaluate their performance.

The benchmark is composed of 50 tasks which gauge performance based on challenges engendered by the aforementioned limitations of existing smart home systems (see Section VI). We evaluate SAGE with several LLMs and compare it to two LLM-enabled baselines in Section VII, and demonstrate a performance improvement of over $2\times$.

The main contributions of this paper are:

1) **SAGE:** Our smart home assistant framework which integrates diverse tools, of which some are completely novel, some are adapted and extended from existing tools, and some are taken off-the-shelf, into a cohesive and powerful system tailored to smart home use. This strategy enables SAGE to overcome key limitations of existing commercial smart home systems, as well as previously proposed LLM-enabled approaches.
2) **Device disambiguation tool:** A mechanism for allowing users to refer to their devices in a natural and non-rigid way when speaking to their smart assistant, as they normally would to another human, based on a single photo of that device in the home.
3) **Device interaction tool:** A module which leverages the ability of LLMs to generate structured text to make API calls for fine-grained control of diverse smart devices.
4) **Persistent command handling via code-writing:** A scheme for allowing the system to persistently monitor device states and execute instructions using code generated by the LLM.

[1]https://ifttt.com/

5) **Personalization tool:** A module that combines two types of user interaction memory to answer questions about user preferences.
6) **Performance benchmark:** A benchmark of 50 tasks to validate SAGE's performance.

We also validate our findings with a proof-of-concept demonstration of SAGE in a real-world smart home setting. A video of this demonstration, as well as the GitHub repository of the project's code, and more, can be found on the SAGE webpage: `saic-montreal.github.io/`.

II. Related Work

A. Home Automation

A smart home system consists of connected IoT (Internet of Things) smart devices that enable the simultaneous monitoring, sensing, and control of the home environment. Automating the control of these devices can lead to improvements in quality of life, comfort, and resource usage [19]. Recent work has aimed to use machine learning to enhance the capabilities of smart home systems. For example, [20] 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. For instance, [21] 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 complex commands [22]. In other words, these systems tend to fail in situations where a user utterance can not easily be mapped to a pre-programmed routine.

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 [2] 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. In contrast, SAGE relies on an LLM to decide the sequence of steps to perform.

Today's smart home users also have the option of manually defining IFTTT-style routines, which connect trigger conditions to actions [23], 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 with SmartThings SmartApps [24], 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, 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 the reasoning and context-awareness offered by LLMs.

Recently, IFTTT has begun to leverage LLMs through the creation of a ChatGPT plugin[4], which provides a more user-friendly and accessible interface to interact with the automation platform. However, this plugin does not allow ChatGPT to generate new routines, rather only to trigger existing ones. Similarly, Zapier is aiming to develop an LLM-based product, which will allow users to create automations based on natural language prompts, but currently lacks significant reasoning and sophistication. [2] created an LLM-based pipeline which outputs trigger-action pairs to create simple IFTTT routines, but the logical complexity of these routines, as well as the flexibility of the triggers, is limited. Earlier academic work investigated training sequence-to-sequence models to synthesize IFTTT or Zapier routines from natural language descriptions [25]. Results were promising but limited in that the sequence models were able to generate the sequence of functions to call, but not the arguments to those functions.

B. Autonomous Agents

LLM-powered autonomous agents are designed 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) [4] is a well-known prompting technique that enables the agent to perform complex reasoning through step-by-step planning and acting. In the CoT implementations, several CoT demonstrations are inserted in the prompt to guide the agent's reasoning process. Alternatively, zero-shot CoT [26] demonstrated the reasoning capabilities of LLMs 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 its own set of sub-steps (e.g. [27]). The aforementioned work did not consider feedback from the environment (i.e. the outcomes of actions already taken) in the plan generation process. Subsequent work did incorporate this feedback. For example, ReAct agent [5] 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 each subsequent reasoning step. 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 action execution. These enable the agent to go beyond its internal knowledge. APIs are the most common

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

type of tool, and LLMs (such as Gorilla [28] and ToolLLM [29]) have been trained specifically for API use. In addition to APIs, external knowledge bases (e.g. databases of documents) can be used as a tool to acquire specific information or expert knowledge [30].

III. System Overview

The structure of SAGE's sequential decision-making process is described in Algorithm 1. Some tools are themselves implemented as agents, which execute their own sequential decision making processes. As previously indicated, such processes are often referred to as *"agents"* [1]. Therefore, we refer to tools which are implemented through such a sequence as *"agent-tools"*, with a single top-level agent-tool providing the entry point for user interaction.

Internally, a tool may query a database, control a device, interact with the user, or even spawn a separate, specialized autonomous LLM agent to carry out specific types of tasks.

The decision functions d_i are implemented using an LLM. The LLM prompt $\mathcal{P}_i$ for the decision function d_i is constructed as follows:

$$\mathcal{P}_i = \text{Concatenate}(\texttt{TaskInfo}_i, \texttt{ToolInstructions}(S_i), \texttt{FormatInfo}, a, \texttt{HistoryInfo}(H_i))$$

where:

- $\texttt{TaskInfo}_i$ provides contextual information about the task to be performed by the meta-tool.
- $\texttt{ToolInstructions}(S_i)$ includes instructions on how to use each tool in the subset S_i of tools available to meta-tool m_i.
- $\texttt{FormatInfo}$ specifies instructions on how the output should be formatted to facilitate parsing.
- a is the input received by the meta-tool.
- $\texttt{HistoryInfo}(H_i)$ compiles the decision history of the meta-tool into a format that can be understood by the LLM.

Given the prompt $\mathcal{P}_i$, the LLM is then sampled:

$$(response, probability) \leftarrow \text{SampleLLM}(\mathcal{P}_i)$$

Here, SampleLLM represents the process of querying the language model with the generated prompt $\mathcal{P}_i$. The language model then returns a response along with an associated probability indicating the confidence of the model in its generated response. The output generated by the LLM needs to be parsed into a format suitable for the algorithm:

$$(action, output) \leftarrow \text{ParseLLMOutput}(response)$$

ParseLLMOutput is a function that interprets the response from the LLM into a defined $action$ (either τ or a tuple (s, a) where s is a tool and a is an argument) and associated $output$ data that is meaningful within the context of the algorithm. If the LLM fails to follow the instructions in $\texttt{FormatInfo}$ this is not possible and an error is thrown.

Algorithm 1 SAGE Decision Process

Require: Set of all tools $T = \{t_1, t_2, \ldots, t_n\}$.
Require: Subset of agent-tools $M = \{m_0, m_1, \ldots, m_p\} \subseteq T$ with m_0 as the fixed entry point agent-tool.
Require: Set of decision functions $D = \{d_0, d_1, \ldots, d_p\}$, where d_i corresponds to m_i.
Require: Subsets $S_i \subseteq T$, the tools available to agent-tool m_i.
Require: Universal set of possible arguments A.
Require: Special terminate action τ indicating an agent-tool should terminate its process and return its output.

Initialize decision history H_0 for the entry point agent-tool's decision function d_0.
loop ▷ Infinite loop awaiting user input
  Await and receive user input *userInput* $\in A$
  $output \leftarrow$ CallAgentTool$(m_0, H_0,$ *userInput*$)$
  Respond to the user with $output$.
end loop

function CallAgentTool(m_i, H_i, $a \in A$)
  loop
    $(action, output) \leftarrow d_i(H_i, input)$
    if $action = \tau$ **then**
      $H_i \leftarrow emptylist$
      return $output$
    else
      Parse action as (s, a^*) where $s \in S_i$, $a^* \in A$
      if $s \in M$ **then**
        Identify the index j such that $s = m_j$
        if not H_j initialized **then**
          Initialize H_j as an empty list
        end if
        $o \leftarrow$ CallAgentTool(m_j, H_j, a^*)
      else
        $o \leftarrow \text{S}(a^*)$
      end if
      $H_i \leftarrow H_i + [(s, a, o)]$
    end if
  end loop
end function

Combining these steps the decision function d_i using an LLM can be encapsulated as:

$$d_i(H_i, input) = \text{ParseLLMOutput}(\text{SampleLLM}(\mathcal{P}_i))$$

Within this framework, planning (the process of breaking down a high-level goal into sub-steps) is implemented using a technique called "chain-of-thought" [4]. In this approach, $\texttt{ToolInstructions}$ and $\texttt{FormatInfo}$ encourage the LLM to output a plan before proceeding to specify exactly how to execute the steps of the plan. A highly simplified example is provided in Figure 2. Observe that the multi-step plan output by the decision function (d_i) the first time it is called is persisted via the history (H_i) so the agent-tool can continue to execute the same plan through multiple LLM calls.

Prompt:
[TaskInfo]
You are an AI that helps the user turn their lights on and off. Think step-by-step about what to do before you execute.

[ToolInstructions]
Tools:
- light ID tool (input: light common name)
- turn on tool (input: light uuid)
- turn off tool (input: light uuid)

[FormatInfo]
Use the following output format
Thought: the steps you need to execute to handle the request
Action: the next tool to use OR τ
Action input: the argument to the tool OR final response (if action is τ)

[a]
User input: Turn on the fancy light

[HistoryInfo]
Previous LLM response
Thought: First I need to use the light ID tool to find the uuid of the fancy light, then I need to turn it on using the turn on tool.
Action: light ID tool
Action input: fancy light
Previous tool output
Observation: the UUID of "fancy light" is 12e4df...bc4
Response:
Thought: I now know the UUID of fancy light. Now I need to use the turn on tool to turn it on.
Action: turn on tool
Action input: 12e4df...bc4

Fig. 2. Simplified LLM prompt construction and response example. Blue text indicates the prompt, orange text indicates the response, and black text is the explanation of the section of the prompt below. In this example, the decision function (d_i) is being called for the second time, so the history (H_i) is populated with the results of the first call.

It may also alter the plan in response to unexpected results, facilitating failure recovery.

In this work, FormatInfo and ParseLLMOutput are taken from ReAct [5]. The HistoryInfo function is a concatenation operation. The set of possible arguments, A, is the set of all strings. TaskInfo, ToolInstructions, the implementations of all tools in set T, and the assignment of sub-tools to meta-tools S_i comprise the method referred to as SAGE. Implementations for several critical tools are described in Section IV.

IV. Tools

In this section, we introduce a collection of tools developed for SAGE (see Table I for a comprehensive list). This table indicates which tools are agent-tools, as well as the sub-tools

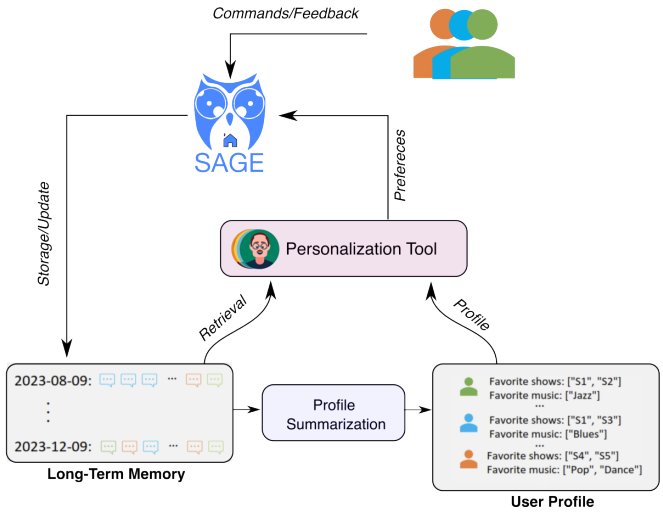

Fig. 3. An overview of the personalization tool. Long-term memory remembers and retrieves all past user utterances. The user profile contains more general trends about the user's preferences.

for those agent-tools, thereby providing an overview of the entire tool hierarchy. To help organize these tools for clarity, we group them into 4 functionality categories: personalization (accounting for user preferences), device interaction (interacting with smart devices), monitoring (continuous monitoring of device states to handle persistent commands), and external interaction (interaction with APIs external to the smart home, e.g. email, weather). The rest of this section provides implementation details for tools in the personalization, device interaction, and monitoring categories. Details on external interaction tools are omitted for brevity, as these types of tools have been well explored in previous work (e.g. in LangChain [6]).

A. Personalization

This section introduces tools with personalization-related functionality: the personalization and human interaction tools. The personalization tool, illustrated in Figure 3, is composed of two main sub-components: (1) the long-term memory that stores the history of all past user interactions, and (2) the user profiler that constructs a hierarchical understanding of user preferences. We first describe these two components, and then detail how they are integrated. Finally, we conclude with a brief description of the human interaction tool.

1) Long-Term Memory: Long-term memory [1], shown on the bottom-left of Figure 3, stores information about the user's past interactions and behavior. The memory records a history of user commands and feedback. Similar to existing information retrieval techniques for LLM augmentation [7], 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 [8]. The vector representations are then indexed and stored in a vector database. In this work, we use the MiniLM embedding model [9].

TABLE I
SAGE TOOL HIERARCHY. EACH ROW CORRESPONDS TO A TOOL IN SAGE AND INDICATES ITS FUNCTIONALITY CATEGORY, NAME, WHETHER IT IS A TOOL-AGENT, AND WHICH SUB-TOOLS (IF ANY) IT HAS ACCESS TO.

Category	Name	Agent-Tool	Sub-Tools
entry point	SAGE	yes	personalization, device interaction, condition code writing, condition polling, email and calendar, weather, TV schedule search
personalization	personalization	no	N/A
	human interaction	no	N/A
device interaction	device interaction	yes	device interaction planner, API documentation retrieval, device attribute retrieval, device command execution, device disambiguation
	device interaction planner	no	N/A
	API documentation retrieval	no	N/A
	device attribute retrieval	no	N/A
	device command execution	no	N/A
	device disambiguation	no	N/A
monitoring	condition code writing	yes	device interaction planner, API documentation retrieval, device disambiguation, code execution
	code execution	no	N/A
	condition polling	no	N/A
external interaction	email and calendar	yes	get contacts, create a calendar event, list calendar events, create email draft, send an email message, search email, get an email message, get email thread
	get contacts	no	N/A
	create calendar event	no	N/A
	list calendar events	no	N/A
	create email draft	no	N/A
	send email message	no	N/A
	search email	no	N/A
	get email message	no	N/A
	get email thread	no	N/A
	weather	no	N/A
	TV schedule search	no	N/A

2) User Profiler: The user profile, shown on the bottom-right of 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 [10], to build the users' profiles. The user profiler starts by grouping all memory entries by date and generating daily user preference summaries to capture user preferences at high granularity. Next, the daily summaries are aggregated into a single global summary serving as the user profile. Our choice of a hierarchical approach is motivated by two main reasons: (1) scalability: as the long-term memory grows with time, a hierarchical approach is highly scalable because it is amenable to MapReduce-style processing [11], (2) information loss: directly generating a concise summary from the long-term memory involves a long-context prompt. The ability of LLMs to successfully identify relevant information within the input context is known to degrade as the length of the input context increases [12].

3) Personalization tool: Leveraging LLMs to create responses tailored to users' preferences has been studied in different fields such as recommendation systems [38] and personal companion systems [39]. Our work represents, however, the first endeavor to incorporate personalization in LLM-based smart home control systems. It is common to augment an LLM-based agent with memory to store information over extended periods. One straightforward approach is to append previous interactions directly into the agent's prompt to add more contextual information. However, this approach is hamstrung by the LLM's limited ability to handle long contexts.

Building on prior research on memory-augmented LLMs [40], our work combines two types of memory structures: *the long-term memory* and *user profile*. The user profile and the 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. These two memory types are combined through a distinct prompting strategy. Given the agent's query, the most similar memories are retrieved from the long-term memory using cosine similarity in the embedding space as the distance metric. These retrieved interactions and the profile are added to the prompt alongside the agent query and the user name. The LLM is instructed to understand and infer user preferences and answer the agent's query as faithfully as possible. Multiple users are supported by maintaining separate profiles and long-term memories. User identity is recognized using voice recognition software [13].

4) Human interaction tool: The human interaction tool allows SAGE to ask the user questions, which it usually uses to clarify intent. The tool is called with a string which is communicated to the user via a text-to-speech interface. The tool waits for the user to reply, and transcribes their spoken answer to text using an off-the-shelf model [14]. Empirically, we have found that the introduction of the human 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 a topic of active research.

B. Device Interaction

In this section, we introduce the set of tools that SAGE uses to interact with smart devices. These tools enable flexible

device interaction that is scalable, in the sense that new devices can be integrated into the system with negligible extra development effort and its capabilities can be leveraged to the fullest extent without the need for device-specific code. For example, if a user adds a smart fridge to their smart home ecosystem, SAGE can integrate the presence of the fridge and all of its capabilities (e.g. temperature settings, door opening detection, power consumption, etc.) into its decision making process without the need for any fridge-specific code to be written.

Most smart home systems (SmartThings, Home Assistant, Google Home, Alexa, etc.) provide APIs for interacting with the smart devices in users' homes. These APIs are documented online, and code examples for using them are available, meaning that LLMs trained on web data (such as GPT4) 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 functionality 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 is not feasible, as doing so would significantly exceed the maximum prompt length of today's LLMs.

Motivated by LangChain's OpenAPI toolkit [15], the device interaction tool is implemented as an agent-tool (the prefix "agent-" may be omitted for brevity). This agent-tool generates a high-level plan using only a general description of devices and their associated capabilities, retrieves detailed documentation for the subset of capabilities that are required by the plan, and then uses these to construct API calls. This behavior is enabled through a collection of tools detailed below.

1) Device interaction planner tool: : Generates a sequence of steps that must be performed by the device interaction agent-tool to complete the given command. The tool is implemented using a single LLM query. This query includes a list of devices, their capabilities, and 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. If the planner cannot directly infer the correct device from the information it has been given, it can supply multiple candidate devices and/or capabilities to the device disambiguation tool (detailed below) to decide on the correct one. The Device Disambiguation tool retrieves detailed documentation for each proposed capability, giving the agent sufficient information to make a final choice. The use of a planning tool, as opposed to relying on the chain-of-thought planning (described in Section III) of the device interaction agent-tool, is motivated by the fact that this planning process requires a large amount of information injected into the prompt. Adding all of this information directly to the device interaction agent-tool prompt would lead to significantly higher LLM query costs, as the agent-tool prompt is usually called many times within the course of a single use of the device interaction tool.

2) 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. This tool takes as input a list of capabilities, and returns detailed documentation for each in JSON format. The JSON format is used for convenience, since this is the format returned by the documentation scraper and device APIs.

3) 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 implement these tools as wrappers around the SmartThings REST API to query and modify device states [16]. 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 back to the device interaction tool agent, it can often react and correct the API request accordingly.

4) Device disambiguation tool: : We introduce a novel method that is capable of aligning a spoken command to its intended device. While current systems require users to formulate explicit commands such as *"Turn on TV #1"*, we propose a method that allows users to use implicit commands that do not include a hard-coded name such as "*Turn on the TV over the dresser*". This new capability changes the way in which humans can interact with their devices by allowing them to express themselves in a more natural manner, and without the need to define and remember names for each device. Our methods build on Contrastive Language-Image Pre-training (CLIP) [17], a visual-language model (VLM) architecture which transforms text and visual information into a common embedding space. This is the first time, to our knowledge, that this method is used within the smart home control space.

The device disambiguation tool 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 taking a photo of the device within its surroundings (during initial device setup), we can resolve the device ID without requiring the user to hard-code a unique device name (which can easily be forgotten and may not be known to guests). For example, in Figure 4, it is obvious from the picture alone that the light is located in the dining room. The device identity is disambiguated using a VLM, as shown in Figure 4, where a multimodal VLM (OpenClip ViT-B-32 laion2b_s34b_b79k, [17]) is used to compute embeddings for the user's natural language description of the device and each of the device images. The device whose image embedding has maximum cosine similarity to the text embedding of the

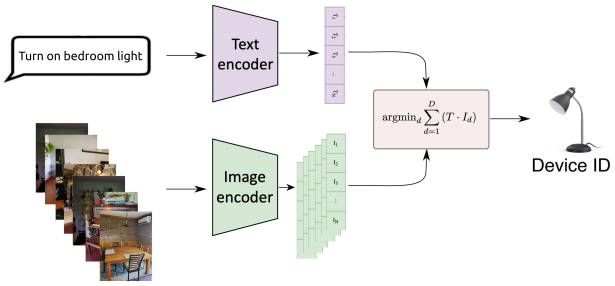

Fig. 4. **Device disambiguation**. A method to resolve the device that best fits a natural language device description using VLMs.

device's description is selected.

5) Comparison with LangChain OpenAPI toolkit: Though the device interaction agent-tool was inspired by the LangChain OpenAPI toolkit, SAGE's device interaction tool includes several key adaptations to make it appropriate for the smart home use case. First, the LangChain toolkit assumes a certain structure to the API, where each unit of functionality is associated with a separate, well-documented HTTP endpoint. In the SmartThings API and other smart home control APIs, each device type exposes significantly different functionality. As each device type is not assigned its own set of HTTP endpoints, the assumptions of the toolkit break down. Our device interaction tool is built around the concepts of `devices` and `capabilities` in the API structure, as opposed to HTTP endpoints.

Second, the LangChain toolkit can only make a sequence of API calls, while our solution is able to interweave calls to other tools (such as the device disambiguation tool).

Finally and most importantly, the prompts of the device interaction planner tool and the device interaction agent-tool are optimized to the smart home scenario and include directives that help produce more correct behaviors in common smart home applications, e.g. making sure that the device is on before modifying its state, or coming up with schemes to determine which device the user is talking about. These schemes do not always rely on the device disambiguation tool, but may also depend on which device is currently on, the names of the devices, or other information.

C. Monitoring

This section introduces a set of tools whose functionality relates to device state monitoring. Many of the more powerful smart home behaviors are unlocked by the ability to monitor the state of some devices and react to state changes. These behaviors are referred to as persistent commands [2] 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 condition is defined, condition checking can be performed using low-cost computing 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 predefined 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 executing the desired behavior. This approach, which retains all of the capabilities of the agent architecture and is simple to implement, has the downside of requiring the agent to constantly be running, incurring significant computational costs.

Our solution to this conundrum is to exploit the ability of LLMs to generate high-quality code. This has been demonstrated using a number of popular benchmarks, such as HumanEval [41]. Such benchmarks feature a text description of what the code must accomplish and a test suite which ensures that the code behaves as expected. The most commonly reported statistic is the success rate on the first attempt. Top-tier LLMs such as GPT4 are currently achieving success rates of approximately 80% on these benchmarks [42]. Therefore, in order to increase the system's flexibility while minimizing cost, we propose a method by which SAGE can autonomously program conditional routines by writing Python code which implements condition-checking logic.

Two tools are introduced to support this functionality: the condition code-writing tool and the 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 (once every few seconds). 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 entire approach, and an example thereof, is summarized in Figure 5.

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 overcome this challenge in a similar fashion: by creating an agent-tool which uses the device interaction planner tool and the API documentation retrieval tool. However, instead of the device attribute retrieval and command execution tools (as in the case of the device interaction tool), the condition code writing tool agent-tool 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 code writing agent-tool, facilitating recovery from faulty code.

V. Execution Example

Figure 1 visualizes an execution trace of SAGE agent-tool handling a single command, illustrating the sequence of tools that are called to complete the task. After checking long-term user interaction memory and user profile information with the *personalization* tool, the system doesn't find any relevant

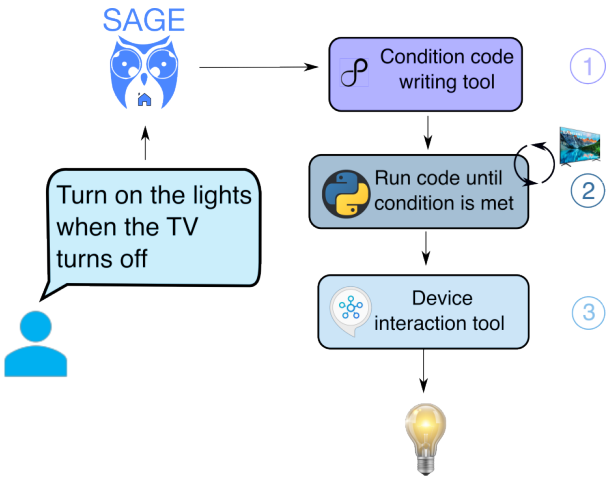

Fig. 5. A summary of the persistent command handling mechanism. **1:** After receiving the user request, the SAGE agent-tool extracts the condition (*"Is the TV off?"*) and uses the condition code writing tool to write code to check this condition. The tool registers this code in memory and responds with the name of the function (is_tv_off). **2:** The SAGE agent-tool 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-tool 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-tool again with the registered action *"Turn the lights on."* Note that the agent-tool will only be triggered when the status of the action transitions from False to True to avoid re-running the agent-tool the entire time the TV is off. **3:** The agent-tool begins executing with the command *"Turn the lights on"* and turns on the lights using the device interaction tool.

information and so must call the *human interaction* tool to directly ask the user about their sports preferences. The SAGE agent-tool then finds the relevant TV program and channel using the *TV schedule search* tool. Once the user's concrete goal is established (to change the TV to a particular channel), SAGE calls the *device interaction* tool, an agent-tool. The *device interaction* tool starts by calling the *device interaction planner* tool to establish the sequence of steps that must be performed. The plan it generates is as follows: First, the *device disambiguation* tool is called to determine which TV the user is referring to. Second, the *device command execution* tool turns the TV on. Finally, the *device command execution* tool is used again to change the channel. The *device interaction* tool successfully executes this plan, then returns execution to the top level SAGE agent-tool, which responds to the user and terminates the run.

VI. Evaluation

Since SAGE is equipped with a range of functionalities which are not present in existing smart home systems, we present a new benchmark to appropriately evaluate their performance. The benchmark is composed of 50 tasks which gauge performance based on technical challenges engendered by the previously discussed limitations of existing smart home systems.

These test cases are implemented by initializing device states and memories, running the SAGE with an input command, and then evaluating whether the device state was modified appropriately. For tasks that involve answering a user's questions (as opposed to modifying device states), the tests are designed such that to answer the question the agent must retrieve a specific piece of information (which it is unlikely to be able to guess). Table VI lists a selection of such tasks used in the evaluation. An LLM-based evaluator is then used to check whether the answer contains the expected information. The results of all tests are binary (pass/fail).

The smart device configuration (device types, IDs, etc) was created by configuring a home with real SmartThings devices, then saving the state of these devices. The devices included: 2 televisions, 1 refrigerator, 1 dishwasher, and 4 lights. The initial states of the devices are modified by the initialization routine of each task. Photographs of the real devices in their real locations are used in the device disambiguation tool.

We classify the test cases according to five types of technical challenges which are difficult for existing systems. Most tests in the set belong to one or more of these categories. The categories are:

1) **Personalization**: Integrating knowledge of user preference to interpret the request correctly.
2) **Intent resolution**: Understanding vague commands and drawing logical conclusions.
3) **Device resolution**: Identifying the desired device ID 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 steps and executing each step in a coherent manner.

We also include a sixth category, **Direct command**, to indicate test cases that are simpler to execute in that they do not feature any of the 5 challenges listed above. These are more comparable to the tasks used to evaluate previous methods such as [43].

The test cases are designed to be run in a completely automated fashion. For this reason, we disable the human interaction tool during testing. We test SAGE with 5 different LLMs: GPT4, GPT4-turbo, GPT3.5-turbo (ChatGPT) [5], Lemur [18], and Claude2.1[6]. These are chosen to represent different points along the performance (and consequently cost) spectrum. GPT4 was the most powerful LLM available at the time SAGE was developed. All prompts in SAGE are optimized specifically for GPT4. GPT4-turbo was released during the evaluation phases of SAGE, and we include it as a less costly alternative. GPT3.5-turbo costs an order of magnitude less than GPT4. Claude2.1 was included as the most competitive commercially available non-OpenAI LLM at the time (note that since the evaluation of SAGE was completed, Claude3[7] and Gemini[8] have been released and are reportedly competitive with GPT4). Finally, Lemur is included to represent the best open source LLMs, having been tuned specifically for autonomous agent type tasks.

[5]https://platform.openai.com/docs/models
[6]https://www.anthropic.com/index/claude-2-1
[7]https://www.anthropic.com/news/claude-3-family
[8]https://deepmind.google/technologies/gemini/#introduction

TABLE II
AN ILLUSTRATIVE SUBSET OF THE 50 TASKS USED FOR EVALUATION. WE CLASSIFY THE TEST CASES ACCORDING TO FIVE TYPES OF TECHNICAL CHALLENGES WHICH ARE DIFFICULT FOR EXISTING SYSTEMS: PERSONALIZATION (PR), PERSISTENCE (PT), DEVICE RESOLUTION (DR), INTENT RESOLUTION (IR), AND COMMAND CHAINING (CC). "SERVICE CATEGORY" DRAWS A PARALLEL BETWEEN TASKS AND POPULAR IFTTT RECIPE CATEGORIES. THE FULL LIST CAN BE FOUND ON THE SAGE CODE REPOSITORY: GITHUB.COM/SAIC-MONTREAL/SAGE.

User command	Challenge category					Service Category
	PR	PT	DR	IR	CC	
It is too bright in the dining room.			✓	✓		Lighting
Turn on the light by the bed.			✓			Lighting
Set up a Christmassy mood by the fireplace.			✓	✓		Lighting
I am getting a call, adjust the volume of the TV.			✓	✓		Television
Put the game on the TV by the credenza and dim the lights by the TV.	✓		✓	✓	✓	Television, Lighting
I am going to sleep. Change the bedroom light accordingly.	✓		✓	✓		Lighting
Dishes are too greasy, set an appropriate mode in the dishwasher.				✓		Appliances
Put something informative on the TV by the plant.	✓		✓	✓		Television
Change the lights of the house to represent my favorite hockey team. Use the lights by the TV, the dining room and the fireplace.	✓		✓	✓	✓	Lighting
Create a new event in my calendar - build a spaceship tomorrow at 4pm.	✓					Internet
Turn on the light in the dining room when I open the fridge.		✓				Lighting, Appliances
Turn on light by the nightstand when the dishwasher is done.		✓	✓			Lighting

For all LLMs, we set the temperature parameter to 0. For each LLM, we run each test case 3 separate times, since LLM performance is somewhat stochastic, even with 0 temperature.

In addition to the main set of 50 tasks, we also created a set of 10 extra "test set" tasks after the development of SAGE was complete. The aim of these tasks was to verify that the prompts had not been over-engineered for the task set. The author who developed these tasks was familiar with the SAGE architecture, but was not involved in the final prompt engineering stages. These test set tasks evaluate the performance on the same five categories of challenges as the main set.

To provide more insights on the key challenges faced by each LLM, we manually annotated the failures that consistently occurred across all three runs. The failures were categorized based on the nomenclature defined in Table III. Since the success of a given test case is contingent upon multiple steps of decision-making, failures can occur in more than one step in the LLM's line of reasoning. For our analysis, the classification of a failure of a given test case is based on the *first* mistake in the execution.

We contextualize the failure categories by grouping them into tiers. These tiers are organized such that, most of the time, a failure in tier n implies success in tiers 1 through $n-1$. For example, if an annotator marked a test case as failing due to a failure in planning (tier 2 failure), this implies that it succeeded at command understanding and formatting (tier 1 failures).

A. Baselines

To contextualize SAGE's performance, we compare our method to two LLM-based smart home automation baselines on our test tasks. The first method, called "One Prompt", involves creating a single prompt comprised of the user command as well as the states of all devices, and asking the LLM to generate updated states in response. The full device state, serialized to JSON format, exceeds GPT4's 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 [2], with some modifications. The original pipeline in [2] 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 support, 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 at a disadvantage in that they are not able to integrate all of the different sources of information that SAGE uses (e.g. user preferences, photos of the devices, 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 because, to our knowledge, there is no previous work that is capable of integrating all of these information sources.

VII. RESULTS

In this section we present results for the competing LLMs and methods discussed in Section VI. We also provide a discussion of potential reasons for performance differences.

Overall success rates for the three methods, SAGE, Sasha, and One Prompt on the 50 task set are presented in Figure 6. SAGE achieves an overall success rate of 76% with GPT4, 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. Unsurprisingly, GPT4

TABLE III
DESCRIPTION OF THE FAILURE CATEGORIES. THE FAILURE TIERS COLUMN HELPS CONTEXTUALIZE THE FAILURE TYPE CATEGORY. IN MOST CASES, A FAILURE ON TIER N IMPLIES THAT FAILURES IN LOWER TIERS WERE AVOIDED (4 = HIGHEST, 1 = LOWEST).

Failure Tier	Failure Type	Description
1	Formatting	Agent fails to follow the template required by ToolInstructions.
	Command understanding	Agent fails to understand what the user is asking.
2	Planning	Agent's plan is incorrect or missing steps.
3	Plan execution	Agent proposes a correct plan but the execution is missing steps.
	Tool selection	Agent chooses the wrong tool at a given stage of the execution.
	Tool population	Agent provides incorrect arguments to the tool.
	API request	Agent makes incorrect SmartThings API request (wrong attribute, command, or component).
	Code writing	Agent's generated code does not work properly.
4	Faulty tool	Non-agent failure (usually a failure of the VLM in the device disambiguation tool).
	LLM limitation	LLM lacking key common knowledge or context length.
	Hallucination	Agent invents concepts (e.g. devices, components, user requirements) that do not exist.
NA	Other	All other failures.

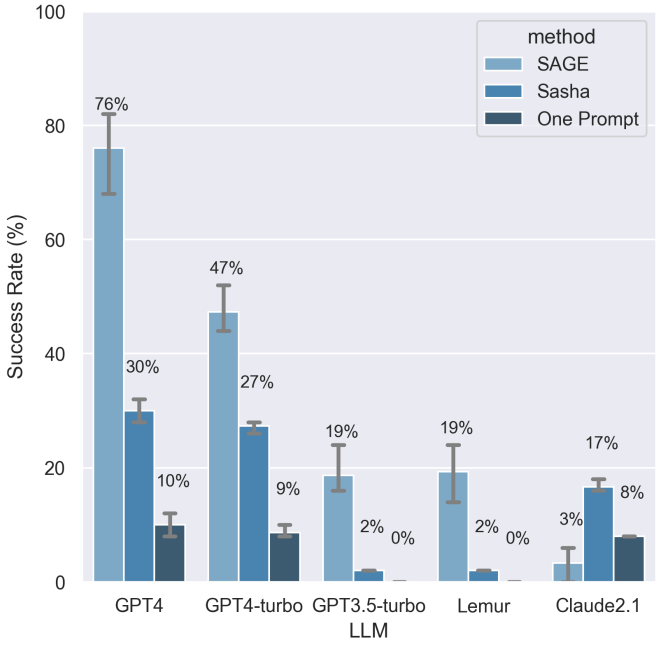

Fig. 6. Overall success rates on 50 challenging tasks. SAGE leverages a collection of tools that allow it to integrate a large amount of information into its decision making process, allowing it to outperform other LLM-based methods by a significant margin. Designed primarily for use with GPT4, it outperforms other methods with a variety of LLMs, including open source models such as Lemur. Error bars indicate max and min scores over the three runs.

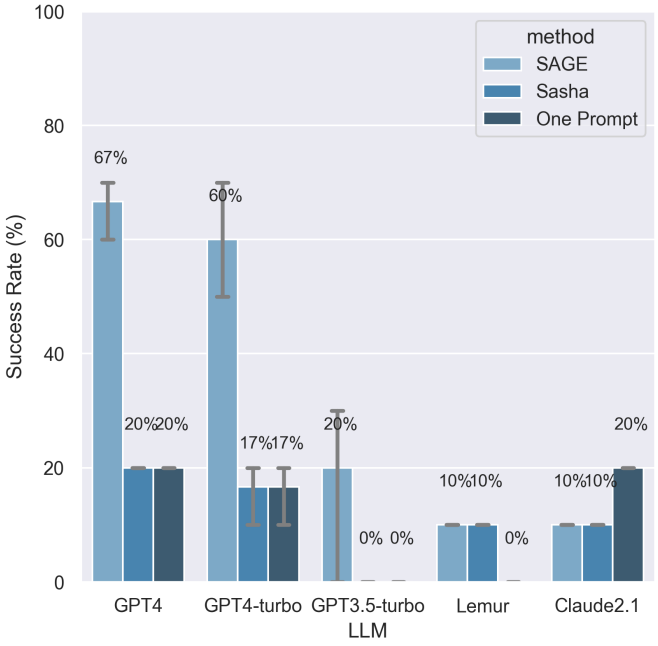

Fig. 7. Overall success rates on a "test set" of 10 extra tasks. These tasks were not seen by the SAGE's designers during the iteration process, and serve to validate that the prompts used in SAGE are not over-engineered to a particular task set. Error bars indicate max and min scores over the three runs.

achieves the highest performance of all LLMs regardless of method. This is largely because it is the most powerful, but also because the prompts were optimized for it. GPT4-turbo appears to be significantly worse than GPT4 on the 50 task set, though it is purported to have similar performance. There are several potential explanations for this phenomenon. First, the details of this model have not been publicly released, but based on the fact that it is several times less expensive to run, we can guess that it has fewer parameters than GPT4. Second, our experience has been that GPT4-turbo is more conversational than GPT4, making it less likely to follow instructions, and therefore worse at agent-style tasks. Finally, the SAGE prompts are optimized for GPT4. It is likely that if we invested equal effort in optimizing them for GPT4-turbo, its performance would increase significantly. On the 10-task "test set" illustrated in Figure 7, the two models attain approximately equal performance. However, given the small size of this set, it may be less accurate in resolving performance differences than the primary set.

Another interesting observation from Figure 6 is that Claude2.1 performs very poorly with SAGE, even compared to both baselines. This can primarily be attributed to Claude2.1's inability to follow formatting instructions and poor tool selection abilities, as illustrated by Table IV, which presents the results of the manual failure analysis. In order to successfully function as the backbone of an agent, an LLM must be capable of following instructions related to formatting and tool selection, as an agent must be capable of converting natural language reasoning into concrete real-world actions. Claude2.1 does not seem to have significant reasoning issues

TABLE IV

RESULTS OF MANUAL FAILURE ANALYSIS. THIS ANALYSIS CATEGORIZES FAILURES INTO ONE OF 12 FAILURE TYPES. THE FAILURE TIERS COLUMN HELPS TO CONTEXTUALIZE THE FAILURE TYPE CATEGORY. IN MOST CASES, A FAILURE IN TIER N IMPLIES THAT FAILURES IN LOWER TIERS WERE AVOIDED (4 = HIGHEST, 1 = LOWEST). ONLY TEST CASES THAT FAILED CONSISTENTLY ACROSS ALL THREE RUNS WERE ANALYZED.

Failure Tier	Failure Type	GPT4	GPT4 Turbo	GPT-3.5 Turbo	Lemur	Claude2.1
		Total Failures = # failures per run ×3				
		9	51	96	111	141
		Failure rate (%)				
1	Formatting	0	0	12	0	35
	Command understanding	67	14	2	0	1
2	Planning	0	6	32	43	0
3	Plan execution	0	2	5	10	2
	Tool selection	0	4	8	3	37
	Tool population	0	18	27	24	15
	API request formatting	33	51	0	0	1
	Code writing	0	6	6	0	0
4	Faulty tool	0	0	5	12	0
	LLM limitation	0	0	1	7	0
	Hallucination	0	0	0	0	2
NA	Other	0	0	0	1	7

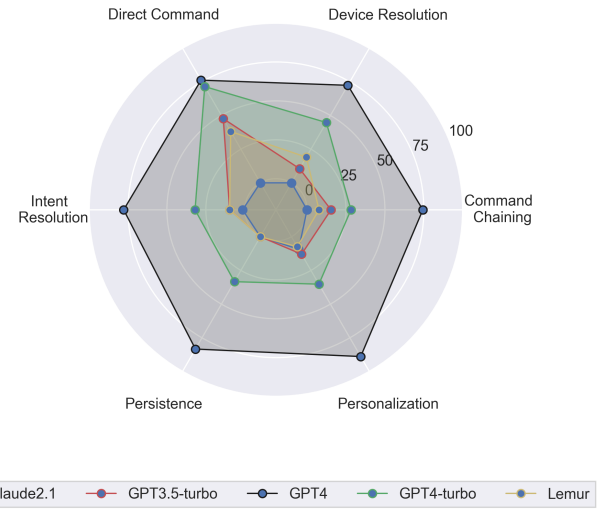

Fig. 8. SAGE success rates for each category of challenging tasks on the 50 task set. Note the inner ring of the chart indicates zero success rate, not the center.

(as evidenced by a lack of failure in the planning stages), but its lack of ability to perform this basic functionality makes it a poor candidate for SAGE and other agent architectures.

Parity of GPT3.5-turbo and Lemur is an encouraging result for those interested in the use of open source LLMs for agent applications.

Examination of Figure 8, which presents SAGE success rates for each category of challenging tasks, reveals that SAGE with GPT4-turbo maintains fairly consistent performance across challenge categories. On the other hand, lesser LLMs tend to perform better on direct commands than more challenging tasks. This confirms that the categories we have identified do indeed present significant difficulties for smart home automation systems.

A. *Failure analysis*

Table IV summarizes the results of the manual failure analysis. In this section, we review each of the LLMs and provide some discussion about the primary reasons they fail.

Out of GPT4's 9 failures that were analyzed, 6 are due to misunderstandings of the user command. These commands are somewhat vague, and the LLM was not able to apply the common sense expected by the test design. The other 3 failures were due to the LLM using the wrong device component (in the SmartThings API some devices have multiple "components", e.g. one for the freezer part and one for the refrigerator part of a smart fridge).

GPT4-turbo performance is hindered by tool usage issues, which account for 69% of the failures (API request formatting and tool population). Indeed, GPT4-turbo most often fails due to problems formulating a request that respects the SmartThings API. This includes providing the right argument types, components, and capabilities.

GPT3.5-turbo predominantly fails to generate proper plans and struggles with passing the right parameters to tools. It also usually fails to use the state of the available devices to inform its decision making process. It also often fails to correctly interpret the user's request, which tends to require significant common-sense reasoning abilities.

Lemur struggles to correctly interpret the user command, given the context of the current device state, and thus also struggles to generate good plans. The plans generated by Lemur are missing vital steps like using device disambiguation to find the right tool or recognizing the need to create a trigger for a persistent command. It also utilizes multiple rounds of trial and error to find the right API calls for device control.

Claude2.1 often struggles to follow the expected format of the response. This involves outputting a thought followed by an action based on previous observations. Although we provide verbose error log and debugging hints to the agent, Claude2.1 fails to recover from its mistakes. Another common failure type for Claude is tool selection. For example, Claude2.1 often uses the personalization tool for direct commands - where it is not needed - instead of the device interaction tool. Note that tool use is a new feature recently added to Claude and is still under development.

Many of these failure modes are instances of the LLM failing to respect the instructions provided within the prompt.

Weaker LLMs tend to fail to attain the bare minimum criterion of correctly formatting their responses, and as such their contents cannot be used. Stronger LLMs can usually respect the response format, but often fail to attend to the details of response content. One commonly occurring example of such behavior is using the incorrect parameter name (API request formatting). The parameter names that are actually used by the LLM may be similar (e.g. temperature_measurement rather than temperatureMeasurement). This example reveals the limitations of in-context learning (the capacity to modify the behavior of an LLM by modifying its context but not its weights), and highlights the way it breaks down. Specifically, as the context becomes longer, the model ceases to attend as carefully to any particular directive in the context. Attempting to tackle highly complex tasks such as smart home control using LLM agents tends to lead to long, complex prompts which push the boundaries of in-context learning.

As such, managing prompt length is a key challenge that drove the SAGE design. SAGE's hierarchy is critical here – it allows for the distribution of functionality among a set of different agent-tools, such that the knowledge (and therefore prompt length) required by each is reduced. Another critical factor in design is the adaptation of the tool interfaces to align with the LLMs preferences. As discussed above, a common failure mode is for the LLM to attempt tool usage in a way which seems reasonable, but which is not actually correct. As a simple example, an LLM using a tool which takes two numbers as inputs and outputs their quotient may invent another argument which controls how rounding is handled. In some cases, it is easier to overcome this by adapting tools to conform to the LLMs' preferred usage. This type of optimization – making sure the prompts are only as long as the LLM can handle and adapting the structure of the hierarchy to be well aligned with the LLM's expectations, gives the LLM that was used for development of the system (in this case GPT4) a big performance advantage. We expect that if the system was optimized for one of the other LLMs, it would likely see a significant performance boost, albeit unlikely that it would be able to perform as well as GPT4 in the same scenario.

VIII. Conclusion

This article introduced SAGE, an LLM agent framework targeted at smart home applications. SAGE orchestrates the use of tools in a sequential decision making process. SAGE integrates a collection of novel tools designed to address key challenges in smart home automation including difficulty interpreting unconstrained natural language of user commands, limitations in interaction with the environment and external data sources, and lack of knowledge of the user's habits and preferences. We also created a dataset of challenging smart home automation test cases which tested the system's ability to be personalized, to resolve user intent from unstructured queries, to resolve devices referred to in natural ways (e.g. *"the TV over the dresser"*), and to appropriately handle command persistence and chaining. These challenges are very difficult for today's smart home automation systems, but reflect the sophistication smart home users will demand of next-generation systems.

SAGE achieved a success rate of 76% on these tasks. This value, while imperfect, is 2.5× better than the next-best LLM baseline, let alone existing commercial smart home assistant systems. Each success required the successful sequential use of many tools, meaning that in fact the number of successful tool uses is much larger than the number of failed ones. To understand the underlying causes of failures, we manually analyzed and categorized each one. This analysis revealed that there are many ways that such a complex decision-making process can go wrong, and that today's most powerful LLMs are needed in order to achieve acceptable performance. However, the respectable performance attained by Lemur, an open-source model fine-tuned for agent tasks, is encouraging. We expect that in one or two years the performance of SAGE with an open source LLM will be comparable to that of GPT4 today, at a fraction of the cost.

As such, SAGE represents a promising first step towards the creation of truly flexible smart home automation systems that users can interact with as naturally as they would with a close friend.

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] 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.

[3] Zapier, "Zapier," https://actions.zapier.com/.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

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

[14] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," 2022. [Online]. Available: https://arxiv.org/abs/2212.04356

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

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

[17] G. Ilharco, M. Wortsman, R. Wightman, C. Gordon, N. Carlini, R. Taori, A. Dave, V. Shankar, H. Namkoong, J. Miller, H. Hajishirzi, A. Farhadi, and L. Schmidt, "Openclip," Jul. 2021, if you use this software, please cite it as below. [Online]. Available: https://doi.org/10.5281/zenodo.5143773

[18] Y. Xu, H. Su, C. Xing, B. Mi, Q. Liu, W. Shi, B. Hui, F. Zhou, Y. Liu, T. Xie, Z. Cheng, S. Zhao, L. Kong, B. Wang, C. Xiong, and T. Yu, "Lemur: Harmonizing natural language and code for language agents," 2023.

[19] C.-W. C. Ki, E. Cho, and J.-E. Lee, "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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] A. de Barcelos Silva, M. M. Gomes, C. A. da Costa, R. da Rosa Righi, J. L. V. Barbosa, G. Pessin, G. De Doncker, and G. Federizzi, "Intelligent personal assistants: A systematic literature review," *Expert Systems with Applications*, vol. 147, p. 113193, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417420300191

[32] B. L. Risteska Stojkoska and K. V. Trivodaliev, "A review of internet of things for smart home: Challenges and solutions," *Journal of Cleaner Production*, vol. 140, pp. 1454–1464, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095965261631589X

[33] D. Bonino and F. Corno, "What would you ask to your home if it were intelligent? exploring user expectations about next-generation homes," *Journal of Ambient Intelligence and Smart Environments*, vol. 3, no. 2, pp. 111–126, 2011.

[34] C. Meurisch, C. A. Mihale-Wilson, A. Hawlitschek, F. Giger, F. Müller, O. Hinz, and M. Mühlhäuser, "Exploring user expectations of proactive ai systems," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 4, no. 4, dec 2020. [Online]. Available: https://doi.org/10.1145/3432193

[35] K. Adeyeye, "The householder is king: Engendering householder participation in bridging the performance gap in homes," *Energy Research and Social Science*, vol. 103, p. 103199, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214629623002591

[36] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[37] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[38] J. Chen, "A survey on large language models for personalized and explainable recommendations," *CoRR*, vol. abs/2311.12338, 2023.

[39] W. Zhong, L. Guo, Q. Gao, H. Ye, and Y. Wang, "Memorybank: Enhancing large language models with long-term memory," *CoRR*, vol. abs/2305.10250, 2023.

[40] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen, "A survey on large language model based autonomous agents," *CoRR*, vol. abs/2308.11432, 2023.

[41] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[42] "Evalplus leaderboard," https://evalplus.github.io/leaderboard.html, accessed on April 3, 2024.

[43] E. King, H. Yu, S. Lee, and C. Julien, "Sasha: creative goal-oriented reasoning in smart homes with large language models," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 8, no. 1, pp. 1–38, 2024.