

AutoTask: Executing Arbitrary Voice Commands by Exploring and Learning from Mobile GUI

LIHANG PAN* and BOWEN WANG*, Department of Computer Science and Technology, Tsinghua University, China

CHUN YU†, Department of Computer Science and Technology, Tsinghua University, China

YUXUAN CHEN, Department of Computer Science and Technology, Tsinghua University, China

XIANGYU ZHANG, Department of Computer Science and Technology, Tsinghua University, China

YUANCHUN SHI, Department of Computer Science and Technology, Tsinghua University, China

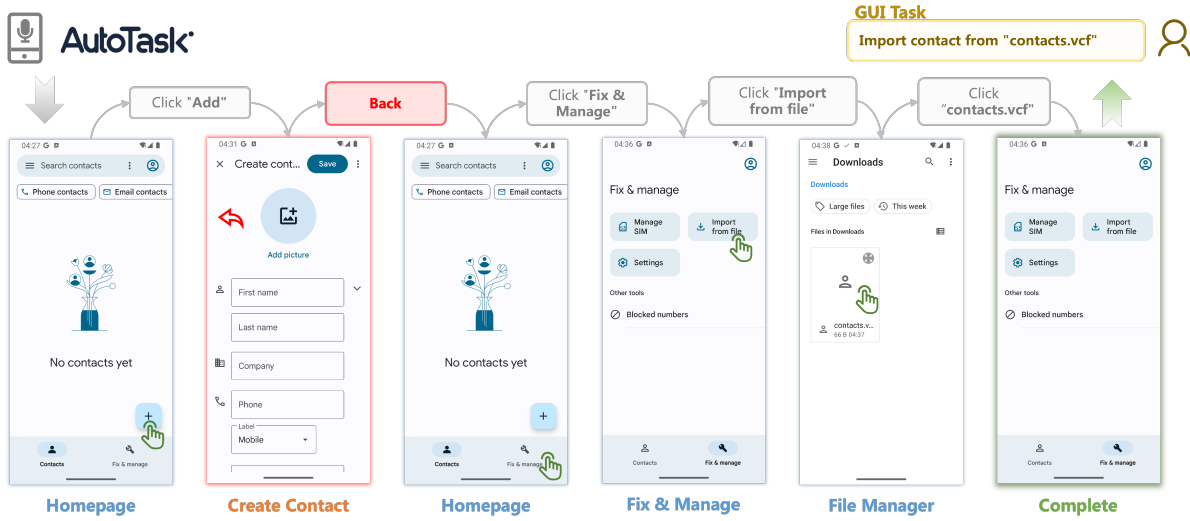


Fig. 1. To execute the command "import contacts from contacts.vcf" in the Contacts application (version 4.8.17), AutoTask first clicks the "Add" button, transitioning from Page 1 to Page 2. AutoTask finds that it can only manually add a contact on Page 2; hence, it reverts to Page 1 and clicks another button labeled "Fix & Manager". It then completes the import process through subsequent steps (4 & 5). After finishing the task, AutoTask synthesizes knowledge from its experiences, improving its ability for future commands.

Voice command interfaces (VCIs) have gained increasing importance, enabling hands-free and eyes-free interaction with digital devices. However, the inherent complexity in constructing effective voice interfaces has limited the VCIs' functionalities to only a small fraction of GUI applications and tasks. This paper presents AutoTask, a VCI capable of automating any task in any mobile application without configuration or modification from developers or end users. The primary challenge for AutoTask is the lack of knowledge, as it needs to accomplish unknown tasks (e.g., user commands) within an unknown environment (e.g., GUI). To address this challenge, AutoTask employs two strategies: (1) trial and error: AutoTask explores the GUI, attempts potential operation sequences, and recovers from errors through backtracking; (2) learning from the environment: AutoTask accumulates experiences during exploration and summarizes correct knowledge from these experiences. We implemented AutoTask on Android devices and conducted an evaluation study, which proved the feasibility of AutoTask.

CCS Concepts: • **Human-centered computing** → **Natural language interfaces**; *User interface programming*.

*Equal contribution

†Corresponding author.

1 INTRODUCTION

Voice interaction can effectively enhance the interactivity of applications, enabling end users to automate multi-step GUI tasks (e.g., setting an alarm for 9 AM) eyes-freely and hands-freely [15, 53, 61, 87, 97]. However, constructing voice command interfaces (VCIs) for existing GUI tasks is challenging and requires significant effort [30, 61]. Consequently, existing VCIs support only a limited set of predefined intents, failing to cover the actual needs of users [53, 61].

Large language models (LLMs) have been applied to numerous domains and have significantly reduced the cost of system development and deployment [13, 22, 24, 67, 96]. While LLMs help understand user commands and alleviate the development burden of VCIs, their application is limited in scope [83]. Developers are still required to pre-define a set of intents for the voice interface [3, 61]. Additionally, they must configure how these intents are executed [43, 47] and address potential errors arising from LLMs [75].

In this paper, we present AutoTask, a ready-to-use VCI that operates without any modifications or configurations by either developers or end users. It is capable of executing any intent within any application. AutoTask accomplishes arbitrary tasks (i.e., user commands) in an unknown environment (i.e., the GUI), the primary challenge of which is the lack of necessary knowledge. To overcome this, AutoTask (1) engages in trial and error: exploring the GUI, attempting possible operation sequences, and recovering from errors through backtracking; and (2) learns from the environment by accumulating experiences during exploration and summarizing knowledge from them.

As illustrated in Figure 1, AutoTask comprehends the semantics of the GUI and the user command and determines an operations sequence to carry out the given task. For example, to execute the command "Import contacts from contacts.vcf", AutoTask first chooses to click the "Add" button on Page 1. It then emulates this operation on the GUI, leading to content updates and a transition from Page 1 to Page 2. AutoTask subsequently evaluates the correctness of the executed operation sequence. If an error is detected, AutoTask revokes its actions to rectify it. For instance, upon reaching Page 2, AutoTask recognizes that it can only manually add a single contact there and cannot perform a batch import from a file. Consequently, it reverts to Page 1 and selects another button labeled "Fix & Manager". This process continues until the task is successfully completed. Additionally, AutoTask improves its performance by accumulating experiences while navigating the GUI and summarizing knowledge from these experiences, including:

- (1) Environmental knowledge: for example, in Figure 1, AutoTask can learn that clicking the "Add" button does not lead to batch importing of contacts. This knowledge can expedite AutoTask's execution of subsequent commands.
- (2) Task knowledge: for instance, in Figure 1, AutoTask can learn that the intent "Import contacts from a file" requires a parameter specifying the file name. This knowledge aids AutoTask in understanding the semantics of the commands.
- (3) Execution knowledge: as shown in Figure 1, AutoTask can learn the correct operation sequence for importing contacts from a file. This sequence can be directly replayed to accomplish similar tasks and help execute other commands.

This paper makes two main contributions:

- (1) We introduce a new paradigm in which an agent accomplishes unknown tasks in an unknown environment. The agent explores the environment to find a solution and summarizes its experiences into knowledge to enhance its capabilities.

- (2) We present a ready-to-use VCI named AutoTask, where end users can automate any intent with a single command. Experimental results proved its usability. We implemented AutoTask on Android smartphones and conducted an evaluation study. The experimental results proved its usability.

2 RELATED WORK

Voice command interfaces can effectively reduce the interaction burden, enabling users to interact with devices hands-freely and eyes-freely [15, 53, 61, 87, 97]. However, constructing a VCI for mobile devices requires significant effort [30, 47, 61], leading to the existing VCIs covering only a limited set of GUI functionalities. The workload primarily encompasses determining the supported intent set, understanding user natural language commands accurately, and executing tasks correctly. Self-improvement of VCIs during runtime is a crucial approach to reducing effort [61] but has yet to gain widespread support. Table 1 compares existing VCIs with AutoTask in these four aspects.

Table 1. Comparison of AutoTask and existing VCIs. The content within parentheses indicates the workload required by developers or end users. PBD stands for programming by demonstration, and CCG stands for combinatory categorial grammar.

| | Intent Set | Command Understanding | Task Execution | Self-Improvement |
|--------------------|-----------------------------------|---|---------------------------------------|---|
| SUGILITE [43] | Predefined (PBD) | CCG (Handcrafted rules) | Replay operations (PBD) | Improve executing (PBD) |
| SAVANT [3] | Predefined (specify a list) | Dialogflow agents (provide examples) | Search app screen (collect traces) | Not supported |
| AutoVCI [61] | Predefined (PBD) | BERT (extra dialogues) | Replay operations (PBD) | Improve understanding (extra dialogues) |
| LLM4Mobile [83] | Intents in the GUI (No effort) | LLM (No effort) | N/A | Not supported |
| AutoTask | Any intent (No effort) | LLM (No effort) | Explore the GUI (No effort) | Improve understanding Improve executing (No effort) |

2.1 Supported intents of VCIs

The earliest voice interfaces on smartphones only supported single-step GUI operations [4, 5, 81, 83, 97], for example, clicking a button already present on the GUI. The intent sets of this kind of VCI are limited to the current GUI contents. Although the VCIs can be applied to any mobile application without any configuration or modification, GUI tasks typically require multiple operations (e.g., clicking several buttons sequentially), and providing voice commands for each step would impose a significant interaction burden. Therefore, this approach is mainly used for accessibility purposes [29, 82, 97] and has not been widely adopted by ordinary users.

Task-oriented VCIs [3, 43, 47, 61, 69] (e.g., Siri) address the aforementioned issues and reduce the interaction burden. End users only provide a single voice command, and the virtual assistant can automatically complete a multi-step task on the GUI (e.g., setting a 9:00 am alarm). However, existing task-oriented voice assistants only support a limited set of intents. For instance, Siri does not support sending WhatsApp messages. This results in two challenges: on one hand, developers need to invest a significant amount of effort (e.g., conducting formative studies [3, 43]) to determine a useful intent set; on the other hand, end users often complain about the discoverability of functionalities [14, 55, 72, 90] and the lack of support for necessary intents [53].

AutoTask differs significantly from the two categories of voice interfaces mentioned above. AutoTask is a task-oriented voice assistant capable of automating multi-step tasks with a single command. However, AutoTask does not rely on a predefined set of intents and can accommodate any intent that can be executed on the GUI without requiring additional overhead from developers or end users.

2.2 Understanding User Commands

A task-oriented voice assistant understands the user command to: (1) classify the command into a specific intent (e.g., sending a message); and (2) identify parameters for the intent (e.g., message content and message recipient) [43, 47, 61]. The most traditional approach to command understanding is using context-free grammar (e.g., regular expressions [5, 97] and combinatory categorial grammar (CCG) [6, 43, 45, 46]). Additionally, researchers have employed more sophisticated algorithms (e.g., word dependency [69], n-grams [25, 37] and word embedding [69]) to extract features from commands and create natural language processing scripts. These solutions heavily rely on handcrafted rules [4, 85], necessitating significant developer effort; however, their performance is relatively poor [6, 53, 88, 91] and cannot satisfy users' needs.

Nowadays, many systems utilize deep neural networks to comprehend user commands, achieving satisfactory results [63, 68]. Before the widespread adoption of pre-trained models, researchers needed to collect a large amount of training data [11, 19, 77] and meticulously fine-tune the model's architecture and parameters to achieve good results [95]. This process involved a significant workload. With the development of pre-trained models, researchers only provide prompts and a few optional examples, and large language models (LLMs) can successfully understand the commands [50, 83].

An often overlooked issue is that interactive systems may inaccurately interpret user commands, leading to conversation breakdowns [7, 28, 61]. This problem increases the user's burden [33, 62] and reduces their willingness to engage in interactions [31, 53, 93]. Existing solutions all entail additional costs. For instance, developers can programmatically address breakdowns [8, 36]. AutoVCI [61] and SOVITE [44] require additional user interactions to ensure accurate command understanding.

AutoTask utilizes LLMs to comprehend user commands, enabling support for any task in any application with minimal development effort. To address the issue of LLM errors, AutoTask does not require developer or user involvement; instead, it automatically learns from the mobile GUI and continually adjusts its command understanding results.

2.3 Executing the Command

Existing VCIs search for and execute scripts in databases based on the command understanding results; the scripts can automatically carry out the user commands. These scripts can be categorized into two types based on their origins: those created by developers and those generated by end users.

The scripts for the majority of commercial voice assistants are manually created by developers [65]. For example, Siri directly invokes functions implemented by developers to execute voice commands. Because developers need to write a script for each intent, this approach significantly limits the number of functionalities available in the voice interface [61].

Since end users have a strong demand for voice assistants that can support their personalized needs, researchers have proposed different methods to collect execution scripts from users. One typical approach is program by demonstration (PBD) [1, 16, 20, 43, 47, 57, 61, 69, 74, 80]. Users can demonstrate how they complete tasks on the GUI; this process is recorded and automatically transformed into execution scripts. Researchers have also attempted to extract execution scripts from users' historical behavior records automatically [2, 3, 39]. While this approach avoids the explicit burden of demonstration, it also results in unpredictable system capabilities and relies on the time and quality of data accumulation.

Note that scripts collected from users are not always correct. For example, replaying the operation sequence demonstrated by users may fail due to pop-up windows or application version updates [43]. Existing solutions require users to handle exceptions manually [43], which introduces additional burdens.

AutoTask does not require any predefined scripts, whether they originate from developers or end users. It dynamically calculates a potential operation sequence on the GUI at runtime. Furthermore, it assesses whether the sequence is correct and automatically handles errors. The entire process does not necessitate any user intervention.

2.4 Self-Improvement

Many systems can learn and improve themselves through interactions with users. A typical application scenario is learning user preferences to provide better services (e.g., recommendations [54, 78], navigation [64], and scheduling [26]). AutoVCI [61] can enhance its semantic understanding ability through multi-turn dialogues with users [77]. In the field of machine learning, this approach is known as "human-in-the-loop" [73, 86, 99], in which users contribute to improving machine capabilities by providing annotations. This approach has achieved significant success in training large language models [58, 59].

Reinforcement learning is a common approach for improving machine capabilities without user intervention: AI-driven agents enhance themselves based on rewards provided by the environment [34]. This approach has been widely applied in fields such as gaming [35] and autonomous driving [38], but it has seen limited application in executing user commands on GUIs [9]. AppBuddy [71] is a preliminary attempt in this direction; however, it suffers from issues like sparse rewards [42] and excessive trial-and-error steps, making it unsuitable for direct application in interactive systems. AutoTask shares a similar concept with reinforcement learning: it autonomously summarizes knowledge from its explorations of the GUI, all without requiring user intervention.

3 PROBLEM FORMULATION & SOLUTION

This paper focuses on problems of the following form: an intelligent agent is required to complete an unknown task in an unknown environment. This type of problem is prevalent in applying artificial intelligence (AI) to daily tasks for two primary reasons. Firstly, it is impossible for the developers to collect corpora and pre-train an agent for every real-world scenario. Secondly, end users often struggle to, or choose not to, provide comprehensive, structured descriptions and step-by-step procedures for tasks. Solving problems characterized by these patterns can significantly broaden the application of AI in everyday tasks.

AutoTask is a solution to the problem in the field of VCIs: it automates the execution of interaction intents (i.e., tasks) expressed through natural language commands by simulating user operation sequences within the GUI (i.e., the environment) of mobile devices. AutoTask supports any applications and intents, ensuring comprehensive coverage of GUI tasks for voice assistants.

The core challenge of "completing unknown tasks in an unknown environment" lies in the lack of knowledge, which includes:

- (1) Lack of **environmental knowledge**. Although the intelligent agent can observe and interact with the environment, there is no prior knowledge about how the environment will change after interactions. For example, AutoTask can acquire GUI content; however, it lacks knowledge about how the GUI will change after simulating user actions (e.g., clicking a button on the screen).
- (2) Lack of **task knowledge**. The agent does not possess a set of supported tasks or have any predefined understanding related to task semantics or the interpretation of external inputs. For example, we have not provided AutoTask with a predefined set of intents or information about intent parameters. Additionally, we have not provided models or scripts to assist AutoTask in recognizing the intents and parameters within user commands.

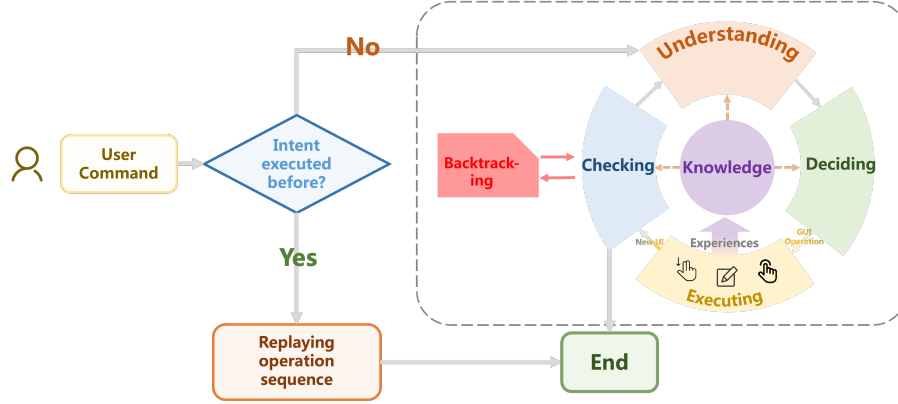


Fig. 2. AutoTask’s pipeline. It first checks if the intent has been executed. If so, it replays the recorded operation sequence. Otherwise, AutoTask explores the GUI and learns. It understands the GUI and command semantics, deciding the most probable operation. This operation is executed by simulating user actions. AutoTask checks the completed actions with the new GUI content. If an error is detected, it backtracks to undo previous actions. Throughout, AutoTask accumulates experience and summarizes knowledge to enhance its capabilities. The arrows pointing to "Experiences" are omitted in the figure for simplicity.

- (3) Lack of **execution knowledge**. Given the absence of environmental and task knowledge, the agent lacks the knowledge of how to execute tasks. For AutoTask, this is reflected in the absence of execution scripts (whether provided by developers or end users) for the intent.

To address this challenge, we propose an "explore-learn" strategy that comprises:

- (1) Trial and error: The agent explores the environment and attempts to execute the task. It recovers from errors through backtracking when necessary.
- (2) Learn from the environment: The agent accumulates experiences (records of actions and observations) during the exploration of the environment. From these experiences, the agent learns knowledge, enabling it to (a) directly execute intents that have been completed in the past and (b) expedite the exploration when executing unknown intents.

The system design of AutoTask is an application of this strategy to the field of voice assistants, which will be elaborated upon in the next section.

4 SYSTEM DESIGN

Figure 2 illustrates the AutoTask pipeline. Upon receiving a user command, AutoTask checks whether the intent expressed in the command has been previously executed. If it has, AutoTask automatically executes the task by replaying the operation sequence, with adjustments based on the current command’s parameter values [61]. If the command has not been executed previously or if the replay of the sequence fails, AutoTask enters the "explore-learn" mode, which can be divided into two parts:

- (1) Trial and error, which can be further subdivided into forward exploration and backward backtracking:
 - (a) During the forward exploration, AutoTask selects an optimal operation within the current GUI content (the understanding module and the deciding module), which is then automated by programmatically injecting an event into the GUI (the executing module). After obtaining the resulting GUI content, AutoTask assesses whether the current task is completed and whether the executed operation sequence

- is correct (the checking module). Based on this assessment, it decides whether to terminate the execution, continue forward exploration, or initiate backward backtracking.
- (b) During the backward backtracking, AutoTask undoes the last action (the backtracking module) and evaluates whether the current task is completed and whether the operation sequence is correct (the checking module). AutoTask decides accordingly to terminate the execution, continue backward backtracking, or start forward exploration.
- (2) Learning from the environment, which encompasses:
- (a) Accumulating experiences: AutoTask records all its decisions (e.g., outputs of the modules) and GUI observations.
- (b) Summarizing knowledge: AutoTask extracts correct knowledge from experiences at appropriate times. The details of AutoTask’s knowledge are illustrated in Table 2.
- (c) Applying knowledge: AutoTask utilizes its knowledge during task execution to directly execute previously completed intents or expedite the exploration process.

Table 2. AutoTask’s knowledge

| Knowledge | Type | Content | Goal | |
|-------------------------|------|--|------------------------------------|--|
| | | | Directly execute completed intents | Expedite Exploration for unknown intents |
| Environmental Knowledge | 1 | Contents and transitions within the GUI (used in understanding) | - | ✓ |
| | 2 | Contents or transitions that do not exist within the GUI (used in deciding and checking) | - | ✓ |
| Task Knowledge | - | Intents and parameters of commands (used in replaying and understanding) | ✓ | - |
| Execution Knowledge | 1 | Correct operation sequences (used in replaying, deciding, and checking) | ✓ | ✓ |
| | 2 | Lessons that prevent execution errors (used in deciding and checking) | - | ✓ |

4.1 Learning from the GUI: Summarizing Experiences into Knowledge

4.1.1 Experiences of AutoTask. AutoTask automatically records its experiences during runtime, which assists the system in executing the current task and is also summarized into knowledge to support subsequent tasks. As depicted in Figure 3, we represent AutoTask’s experiences as a graph, where nodes represent GUI pages, and edges record the results of the modules. Please refer to the corresponding sections for detailed results of each module.

4.1.2 Environmental knowledge. AutoTask’s environment is the GUI, and its environmental knowledge describes the contents of GUI pages and the transitions between pages. This knowledge can be categorized into the following two types:

- (1) (Type-1) Contents and page transitions of the GUI, which are stored in the form of triplets (S, O, D). S and D represent the GUI pages before and after the operation (represented through HTML, as indicated in Figure 5). O = (E, A, P), which denotes elements, actions, and parameters (e.g., the texts for the "text input"

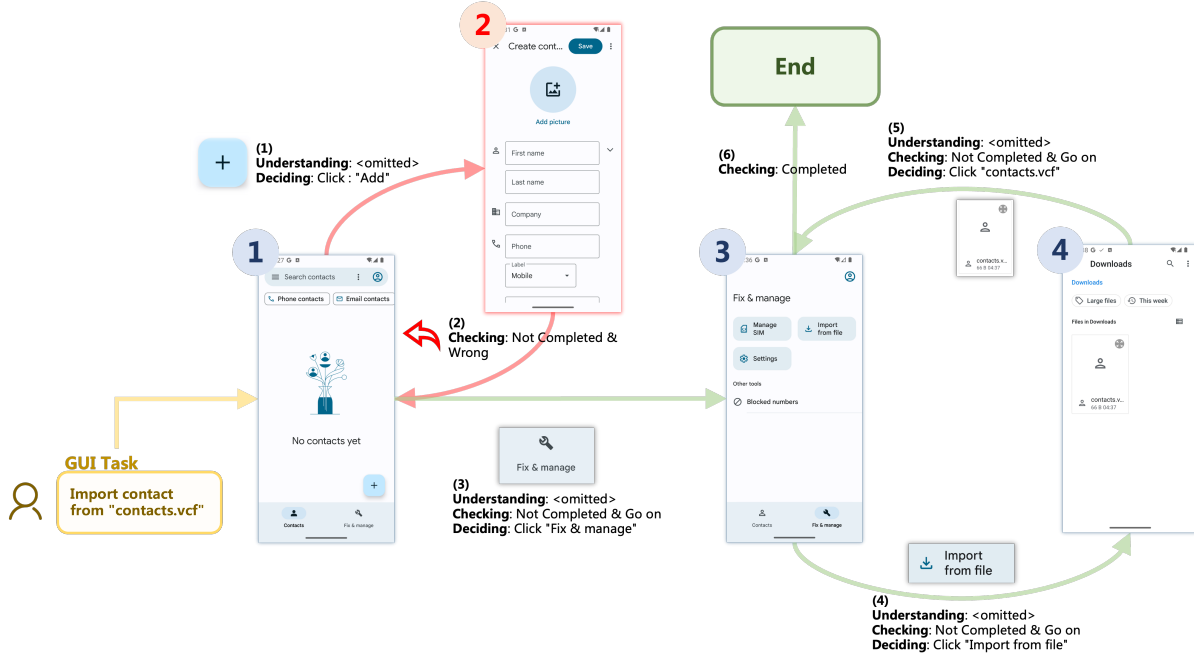


Fig. 3. AutoTask's execution experience for the command "Import Contacts from contacts.vcf". (1) AutoTask clicks the "Add" button on Page 1. (2) AutoTask detects an incorrect history operation based on the content of the new page (i.e., Page 2) and backtracks to Page 1, as indicated by the orange arrow. (3) AutoTask determines that the task remains incomplete and selects the "Fix & Manage" button. (4) AutoTask clicks "Import from file" on the new page. (5) AutoTask selects "contacts.vcf" following the user command. It is a parameter of the intent. (6) AutoTask concludes that the task is completed and finishes the execution.

- actions), respectively; these three components together describe a GUI operation. For example, the Edge (5) (in Figure 3) corresponds to the following triplet: (Page 4, (button labeled "contacts.vcf", click, null), Page 3).
- (2) (Type-2) Contents or transitions that do not exist in the GUI. This kind of knowledge is in natural language. For example, one piece of environmental knowledge (Type-2) summarized for Edge (1) (in Figure 3) could be, "By clicking the 'Add' button on the home page, you can only manually adding a single contact. Importing contacts from files is not supported". This knowledge may prevent AutoTask from attempting to click the "Add" button in future commands (e.g., "Import contacts from cloud backup"), thereby expediting the execution of subsequent tasks.

4.1.3 Task knowledge. Task knowledge helps understand the semantics of the tasks. AutoTask's task is to execute natural language commands, the semantics of which are typically described by intents and parameters (also called slots in some works [47]). A piece of task knowledge comprises (1) an intent name, (2) parameter names, (3) a command corresponding to the intent, and (4) values of the parameters in that command. An intent may appear in multiple pieces of task knowledge, as the commands inside are different. For example, the task knowledge from Figure 3 is (1) intent name - "import contacts from file"; (2) parameter name list - "file name"; (3) command - "import contacts from contacts.vcf"; (4) parameter values - "file name = 'contacts.vcf'".

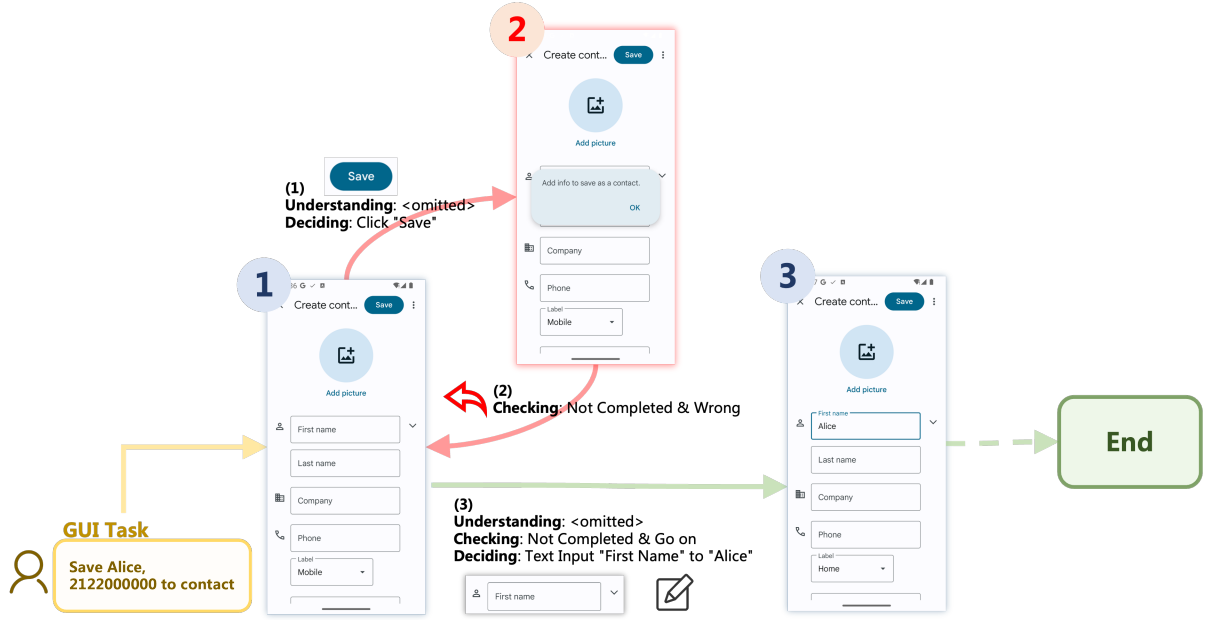


Fig. 4. AutoTask's execution experiences for the command "Save Alice, 2122000000 to contact". AutoTask makes a mistake: clicking the Save button (Edge 1) without entering the name and the phone number. This error is corrected through backtracking (Edge 2). AutoTask also summarizes a piece of knowledge to avoid similar mistakes: you should pay attention to the order of steps; actions that may finalize a task (e.g., clicking the Save button) should be performed last.

4.1.4 Execution knowledge. Execution knowledge describes how to execute the command in the GUI. It can be categorized into the following two types:

- (1) (Type-1) The correct operation sequence for the command. For example, the first type of execution knowledge for "import contacts from contacts.vcf" is "click 'Fix & manage', click 'Import from file', click <file name>". The angle brackets (< >) denote parameter values in the command.
- (2) (Type-2) Describing how to avoid incorrect operations. This knowledge comprises lessons summarized in natural language, corresponding to errors made by AutoTask during execution. As exemplified in Figure 4, AutoTask makes a mistake when executing "save Alice, 2122000000": it clicks "Save" without adding any information about the contact. A possible piece of knowledge summarized from this error could be, "You should pay attention to the order of steps; actions that may finalize a task (e.g., clicking the Save button) should be performed last".

4.1.5 From experiences to knowledge. AutoTask's experiences need to be further summarized into knowledge because some experiences may be redundant or erroneous.

Environmental knowledge (Type-1): When AutoTask simulates an operation on the GUI and obtains the contents of the resulting page, the experience is immediately transformed into environmental knowledge (Type-1). AutoTask's observations of the environment are always correct; as a result, AutoTask can directly and instantaneously convert the related experiences into knowledge.

Execution Knowledge (Type-1): AutoTask identifies the shortest path in its experiences that (1) connects the starting point and the endpoint¹ and (2) encompasses all parameters. For example, in Figure 3, the path "3-4-5" satisfies the aforementioned criteria. This path is considered the correct operation sequence for the task and is added to the database. Such knowledge is only summarized after the task is completed because AutoTask can only determine the task's endpoint at that time.

Task Knowledge: AutoTask records the command understanding result (to be discussed in 4.2.2) at the final step of the correct path as a piece of task knowledge. This type of knowledge will be summarized once AutoTask completes the task; otherwise, the command understanding result may be incorrect.

Environmental Knowledge (Type-2) & Execution Knowledge (Type-2): The purpose of these two types of knowledge is to prevent errors during task execution. AutoTask compares its experiences with the correct path to identify erroneous steps (e.g., Step 1 in Figure 3 & 4). AutoTask categorizes the reasons for errors into two types:

- (1) Lack of environmental knowledge. For example, the error in step 1 of Figure 3 occurs because AutoTask does not know whether there will be an "import from file" option after clicking the "Add" button. Since importing from a file is a way to "add" a batch of contacts, AutoTask considers trying the "Add" button worthwhile.
- (2) Lack of execution knowledge. For example, the error in step 1 of Figure 4 happens because AutoTask does not know how to determine the order of operations when multiple GUI actions are related to the command.

AutoTask summarizes a lesson in natural language for each error and utilizes it to avoid future errors. This knowledge can only be summarized after completing the task. Otherwise, AutoTask cannot accurately determine whether a step is correct. We employ LLM to compile this type of knowledge. The prompt will be discussed in section 5.3.

4.2 The Understanding Module

In the understanding module, AutoTask comprehends the GUI and the user command, the results of which serve as inputs for subsequent modules. This module enables AutoTask to augment information about the environment (i.e., the GUI) and the task (i.e., the user command) with its knowledge.

4.2.1 Understanding the GUI. The GUI semantics are formed by the contents of GUI pages and the transitions between pages. AutoTask can obtain page data through APIs provided by the operating system (e.g., Android AccessibilityService²). However, the incompleteness of GUI semantics arises because the contents after GUI operations cannot be foreseen. AutoTask addresses this issue by querying environmental knowledge to infer the elements "hidden" behind the current GUI elements. This process can be divided into two steps:

- (1) AutoTask retrieves elements from the environmental knowledge (Type-1) that can be reached through one or multiple operations starting from the current GUI elements.
- (2) To filter out irrelevant elements, these elements are transformed into vectors, and their similarities to the user command are computed (details are discussed in section 5.2). The semantic understanding result includes only elements with similarities exceeding a threshold. As exemplified in Figure 5 (C), the button "App pinning" is very related to the user command "enable app pinning" and is reachable by operating an element on the current GUI (clicking "Security & privacy" and then clicking "More security & privacy"). As a result, it is added to the "target" property of the button "Security & privacy".

The GUI semantics effectively guide AutoTask in selecting the correct operations. As illustrated in Figure 5, AutoTask successfully executes the command "enable SIM lock". This is not challenging since SIM lock and

¹The GUI screen where AutoTask thinks the task is completed

²<https://developer.android.com/reference/android/accessibilityservice/AccessibilityService>

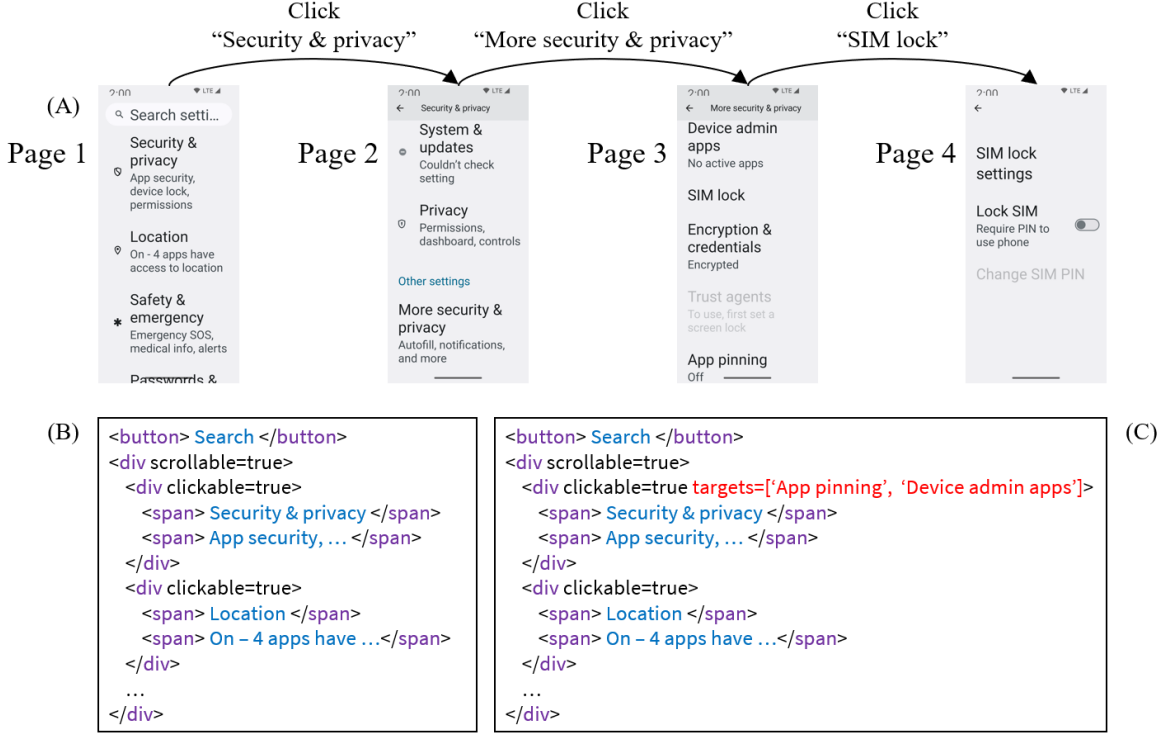


Fig. 5. An example result of AutoTask understanding the GUI. (A) The operation sequence for enabling SIM lock. We omit some operations, such as scrolling the screen. AutoTask learns environmental knowledge (Type-1) from the sequence. (B) The HTML representation of Page 1. We omit some GUI elements. (C) The HTML representation after AutoTask understanding the GUI when executing the command "enable app pinning". The understanding result corresponds to the "targets" property of the "Security & privacy" button.

security are highly semantically related. AutoTask also learns GUI-related knowledge during execution. Next, AutoTask executes the command "enable App pinning". Without relevant GUI knowledge, executing this command is challenging: AutoTask may blindly attempt to click on different buttons on the Android Settings homepage, such as "Application", "Display", and "Safety". However, GUI knowledge can assist AutoTask in directly choosing "Security & privacy" without additional explorations. During GUI understanding, AutoTask discovers a high semantic similarity between the "App pinning" button and the user command, and there exists an operation sequence from "Security & privacy" to "App pinning".

4.2.2 Understanding the command. During the process of understanding command semantics, AutoTask generates a natural language phrase that describes the intent conveyed in the user command. It also detects the parameters and their values from the command. Command understanding needs to be performed in each iteration of AutoTask because the understanding results may be updated as experiences accumulate [61].

The command semantics can guide AutoTask in taking correct operations in the GUI. For example, in the command "Save Alice, 2122000000 to contact" (Figure 4), if AutoTask realizes that "Alice" and "2122000000" are parameters for "create a new contact", it will use the two parameters during task execution, that is, entering them

into text boxes³. The results of command understanding may be stored as task knowledge (as already discussed in 4.1.5), which can assist AutoTask in comprehending subsequent commands.

AutoTask employs task knowledge and an LLM to understand the semantics of commands. We calculate the semantic similarities between the historical commands in the task knowledge and the current user command (details will be discussed in 5.2). Historical commands with similarities greater than a threshold will be selected as examples. The LLM will utilize these examples, the executed operation sequence, and current GUI contents to calculate the command understanding results. The prompts will be discussed in 5.3.

4.3 The Deciding Module

Table 3. Types of operations supported by AutoTask. Click, text input, and scroll forward are used in the deciding module, while the other three are used in the backtracking module.

| Action | GUI Elements | Parameters | Usage |
|-----------------|---------------------|--------------|---|
| Click | Clickable elements | N/A | The Deciding Module |
| Text Input | Editable elements | Text content | |
| Scroll Forward | Scrollable elements | N/A | |
| Navigate Up | N/A | N/A | The Backtracking Module (to undo a click operation) |
| Clear Text | Editable elements | N/A | The Backtracking Module (to undo a text input operation) |
| Scroll Backward | Scrollable elements | N/A | The Backtracking Module (to undo a scroll forward operation) |

In the deciding module, AutoTask calculates the most possible operation in the current GUI to complete the user command. AutoTask identifies all operations⁴ available in the GUI. Table 3 provides an overview of the types of supported operations. Subsequently, AutoTask assigns scores to these operations, with higher scores indicating a greater possibility of being the next operation. Each operation’s score is calculated based on a basic score and a penalty factor: $score = basic_score / (1 + penalty)$:

- (1) Basic Score (1.0 - 8.0), which is calculated by adding the following two components together:
 - (a) Likert scale (1.0 - 7.0). We employ an LLM to assess the relevance of the operations to the user’s command. We use a 7-point Likert scale, where 1 indicates extremely low relevance, and 7 denotes very high relevance. To assist the LLM in calculating the scores, we retrieve relevant environmental knowledge (Type-2) and execution knowledge (Type-1 & 2) from the knowledge base, as discussed in 5.2. Further details about the prompt will be provided in 5.3.
 - (b) Tie-breaking score (0.0 - 1.0). We calculate the semantic similarities between operations and tasks (please refer to section 5.2 for more details) as the tie-breaking scores. The tie-breaking scores increase the differentiation between different operations. While Likert scales are typically effective at identifying the most relevant operation, they may lack granularity when scoring less relevant options. The tie-breaking scores prevent operations from receiving identical scores, thereby avoiding AutoTask being reduced to brute-force searching.
- (2) Penalty factor (0.0 - positive infinity). It is important to note that penalizing an operation does not necessarily mean that the final score of the element will not be the highest. For example, when an operation is penalized

³Another way to use a parameter is to select an item with corresponding text in a list [61].

⁴The parameter for text input will be determined later

by the checking module, AutoTask may attempt other operations and find that these operations are even less relevant to the user command. In this case, AutoTask may retry the penalized operation. The penalty factor consists of two components:

- (a) Repetition penalty, used to penalize operations that have already appeared in the executed operation sequence⁵. The repetition penalty is fixed at 10.
- (b) Backtracking penalty, used to penalize operations considered incorrect by the checking module. The backtracking penalty is initialized at 0 and can be updated by the checking module (see 4.5 for details).

AutoTask selects the operation with the highest score as the next to be executed. If the current operation is text input, AutoTask utilizes an LLM to calculate the text content. Please refer to 5.3 for more details about the prompt.

4.4 The Executing Module & the Backtracking Module

In the executing module, we utilize the accessibility API to inject operations into the GUI based on the results from the deciding module. Conversely, the backtracking module injects interaction events (e.g., scrolling backward) to undo previous operations (e.g., scrolling forward, as indicated in Table 3). Both modules retrieve the GUI hierarchy after injecting events, which will be converted into HTML format (as shown in Figure 5(B)) and used by other modules. Operations may only cause some minor localized changes in the GUI, so we compare the GUI pages before and after the operations to identify newly appeared elements, which are then marked with a boolean property named "new" (as shown in Figure ?? in the Appendix).

We remove elements that meet both of the following two criteria from the GUI hierarchy:

- (1) Low interaction importance. This criterion applies when the element itself and its descendant elements (if any) cannot be interacted with and do not contain text or descriptions.
- (2) Low layout importance. This criterion applies when the element has no sibling elements or all sibling elements are considered to have "low interaction importance".

4.5 The Checking Module

After AutoTask performs GUI operations (both in the executing module and the backtracking module), the checking module conducts two checks on the completed operation sequence: completeness and correctness.

4.5.1 completeness check. The checking module employs an LLM to determine whether the current task is completed. This check is also performed during the backtracking process to address "overshoot" issues, where unnecessary operations are executed after task completion. For more prompt details, please refer to 5.3. The checking module also considers the task completed when the number of executed steps exceeds a threshold (set to 20 in our implementation).

When AutoTask considers the task as completed, we present the user with a list describing the shortest execution path (as discussed in 4.1.5). Each item in the list includes (1) a screenshot, (2) a rectangular bounding box used to highlight the operated element in the screenshot, and (3) a text description of the operation, e.g., "Text input: Alice". The user can choose one of the following options:

- (1) Confirming the correctness of the execution process. The system summarizes the knowledge and terminates.
- (2) Confirming that the task is not yet completed. The system continues running and starts the correctness check. The current page will not be considered as the endpoint for the current command.
- (3) Forcing termination. The system stops running directly without knowledge summarization. Note that the first type of environmental knowledge is still summarized and accumulated.

⁵Note that a combination of an action, a GUI element, and a parameter describes an operation. A repetitive operation implies that AutoTask has arrived at the current GUI screen.

- (4) Ignoring (default⁶). If the step threshold is exceeded, AutoTask stops running without knowledge summarization. Otherwise, the system summarizes knowledge and terminates.

4.5.2 correctness check. If a task is not completed, AutoTask checks whether the last step⁷ currently being executed is correct, i.e., whether AutoTask can continue to fulfill the user’s instruction. The essence of a correctness check is checking the correctness of the deciding module with more experiences and knowledge accumulated from the GUI. For example, in Figure 3, AutoTask clicks the "Add" button to import contacts from the file. However, after simulating user interaction and obtaining new GUI contents, the checking module can discover that the result of the deciding module is erroneous.

AutoTask applies an LLM to conduct the correctness check. Please refer to 5.3 for the detailed prompt of LLM. It is worth noting that AutoTask takes into account the backtracking penalty of the last step. If an operation has a high backtracking penalty but is still executed, the possibility of other operations may be lower. In such a case, the checking module should be more tolerant and consider it correct, providing an opportunity for further exploration.

If the last operation is considered to be incorrect, the checking module will calculate a penalty (0-9) to describe the severity of the error: 0 indicates that the error in the last operation is due to preceding steps already being incorrect; 9 indicates a very serious error in the last operation itself. This penalty will be accumulated into the current backtracking penalty of the last operation. Consequently, the backtracking penalty of an operation may be greater than 9, which indicates that it has been rejected by the checking module several times. We use LLM to calculate the penalty, with details of its prompt discussed in 5.3.

5 IMPLEMENTATION

In this section, we describe how AutoTask utilizes LLMs for computational purposes. Please refer to the Appendix for more detailed examples.

5.1 AutoTask’s context

AutoTask’s context describes its state and is widely used throughout the computational process, as shown in Figure ?? in the Appendix. It encompasses (1) the user command, (2) the executed operation sequence, and (3) the current GUI contents represented in HTML. The GUI content is augmented with environmental knowledge (Type-1), as discussed in 4.2.1. (4) the latest semantic comprehension result of the instruction (if any).

5.2 Embedding & Similarity: Choosing One or More Answers from Several Candidates

AutoTask utilizes the Embedding & Similarity approach to select one or multiple answers from several candidates for a given question. Both the question and the candidates are transformed into vectors using the embedding API provided by OpenAI. We regard each candidate’s cosine similarity with the question as its score. A higher score indicates that the candidate is more likely to be the correct answer. Table 4 summarizes the usage of this method. It is worth noting that the description of elements includes their surrounding elements, as they may exhibit strong semantic relevance.

5.3 Text Completion: Answering Questions

AutoTask utilizes the text completion approach to generate an answer for a given question. The question is passed as part of the prompt to the LLM (gpt-4), and the response generated by the LLM serves as the answer. Table 5

⁶In the evaluation study, we assumed that users would select this option.

⁷Backtracking steps or the steps being undone will not be considered as last steps. For example, if "A-B-C" forms an operation sequence and AutoTask uses operation D to undo operation C, then, even though the sequence is "A-B-C-D", we regard B as the last step.

Table 4. Usage of the Embedding & Similarity approach. To prevent the table from becoming too wide, we use "env." as an abbreviation for "environmental" and "exe." as an abbreviation for "execution".

| Purpose | Question | Candidate Answers |
|---|--|--|
| Retrieve elements from env. knowledge (Type-1) (4.2.1) | (1) context (2) "what element is related" | Elements in env. knowledge (Type-1) |
| Filter relevant pieces of task knowledge (4.2.1) | (1) context (2) "what task is similar to the command" | Tasks and its semantics from task knowledge |
| Filter relevant pieces of env. knowledge and exe. knowledge (4.3 & 4.5) | (1) context (2) "what knowledge is related" | Env. knowledge (Type-2) Exe. knowledge (Type-1 & 2) |
| Compute tie-breaking score (4.3) | (1) context (2) "what element should be operated" | Current GUI elements |

summarizes the usage of this method. The output template specifies the JSON format the LLM response should adhere to, which AutoTask can parse easily.

Table 5. Usage of the Text Completion method

| Module | Purpose (question) | Prompt Composition |
|---------------|---|--|
| Understanding | Understand command semantics | (1) Purpose (2) Task knowledge (3) Context (4) Output template |
| Deciding | Calculate the Likert scale | (1) Purpose (2) Execution knowledge (3) Context (4) Output template |
| | Calculate parameter for text input | (1) Purpose (2) Context (3) Textbox to be edited (4) Output template |
| Check | Checking completeness Check correctness Calculate penalty | (1) Purpose (2) Execution knowledge (3) GUI before last operation (4) Context (5) Output template |
| N/A | Summary Knowledge | (1) Purpose (2) AutoTask experiences (3) The erroneous step (4) The ground truth (5) Output template |

6 EVALUATION STUDY

The evaluation study has two goals: (1) to validate that AutoTask can correctly execute user instructions without predefined knowledge, and (2) to validate that knowledge accumulation can effectively accelerate AutoTask’s execution of user commands. We did not evaluate AutoTask’s feasibility regarding executing intents that have been carried out before. In such cases, AutoTask only replays the recorded operation sequences (with parameter adjustments), and the feasibility has already been validated in previous works [61].

6.1 Apparatus

We conducted the evaluation on an Android virtual machine (Pixel_XL running Android 11). No modifications were made to the virtual machine or system, and AutoTask can run on commercial physical devices.

6.2 Tasks

We validated the capabilities of AutoTask on the following datasets:

- (1) PixelHelp [48]. Consisting of natural language commands and their corresponding operation sequences, PixelHelp was revised for this study. Due to system and application upgrades, some outdated instructions were manually removed, leaving a total of 67 instructions.
- (2) UGIF [79]. This dataset also comprises natural language commands and their corresponding operation sequences. We concentrated on instructions related to Android Settings, as knowledge accumulation is more pronounced within the same application. We used these tasks to assess the impact of knowledge accumulation on AutoTask’s performance. Similar to PixelHelp, outdated instructions were removed, resulting in 100 remaining instructions.

We made adjustments to the commands in the datasets, including:

- (1) Changing the tone from inquiry to command. Both datasets were compiled from tutorials on the internet, where instructions were mostly presented in the form of inquiries. We modified the instructions to align them with how users typically interact with voice assistants. For example, we changed "how to turn off WiFi" to "turn off WiFi".
- (2) Supplementing missing parameters. Some commands lacked parameters and were not executable. We added parameters to these instructions. For example, we changed "how to delete a Google account" to "delete the Google account named Alice".

6.3 Metrics

Success rate, i.e., the ratio of the number of successfully completed tasks to the total number of tasks.

The **step accuracy** of a task, i.e., the ratio of the number of correct steps to the minimum number of steps required to complete the task. The correct steps are defined as the longest subsequence (instead of substring) of the actually executed steps that satisfy the following criterion: the minimum steps needed to complete the task start with the subsequence. For completed tasks, this metric is always 1. For tasks that were not successfully completed, this metric indicates the proximity of AutoTask to successful completion.

The **step redundancy rate** of a task, i.e., the ratio of the difference between the number of executed steps and the number of correct steps to the number of executed steps. This metric evaluates the system’s efficiency. Tasks that succeeded with a step redundancy rate of 0 are referred to as "tasks completed without redundancy".

Non-redundant completion rate, i.e., the ratio of the number of tasks completed without redundancy to the total number of tasks.

6.4 Baseline

We utilized the LLM approach proposed by Android in the Wild (AITW) [66] as the baseline. Similar to AutoTask, this approach employs an LLM to automate user instructions without requiring any configuration or modifications, making it applicable to arbitrary GUI intents. However, the baseline solution lacks an explicit self-checking and backtracking mechanism, although it can undo previous actions by performing certain actions (e.g., clicking the back button). Furthermore, it does not summarize and accumulate knowledge from the execution process. Note that in the original baseline solution, the prompt used only included information about the most recent five operations. We adjusted it to include the complete operation sequence to ensure consistency with AutoTask. When the number of execution steps in the baseline approach exceeds 20, we also forcibly terminate it.

6.5 Procedure

AutoTask and the baseline are executed in the same order for the commands from the two datasets (shuffled beforehand). After completing each task (normal completion or exceeding the maximum number of steps), the next instruction is automatically executed. Throughout this process, AutoTask accumulates knowledge when a task ends successfully but does not utilize knowledge derived from other tasks⁸. After completing all tasks, experimenters manually check whether each task has been executed correctly.

We categorized the tasks in UGIF into three types based on AutoTask’s execution results: (Type-1) tasks that AutoTask can complete without redundancy; (Type-2) tasks that AutoTask can complete with redundancy; (Type-3) tasks that AutoTask is unable to complete. Tasks of Type-1 and 2 are collectively referred to as Type-A tasks; AutoTask has already summarized knowledge for Type-A tasks in Phase 1. Tasks of Type-2 and 3 are collectively referred to as Type-B tasks; there is room for improvement in AutoTask’s performance on Type-B tasks.

We then evaluate the improvement of AutoTask’s performance with the accumulated knowledge. For each Type-B task, we randomly select a certain number of tasks from Type-A tasks⁹. We evaluate the performance of AutoTask after accumulating the knowledge from these tasks. We repeat the aforementioned random selection process ten times and compute the average results to mitigate the influence of random noise. To explore the impact of the amount of knowledge on AutoTask’s performance, we repeated the process several times with different percentages of the selected Type-A tasks: 20%, 40%, 60%, 80%, and 100%.

6.6 Results

6.6.1 Success rate. The accuracy of AutoTask in PixelHelp is 91.2% (6 errors in 67 commands) and that in UGIF is 93.0% (7 errors in 100 commands). The two metrics for the baseline are 52.2% (32 errors in 67 commands) and 67.0% (33 errors in 100 commands), respectively. The chi-square test ($p < 0.001$) proved that AutoTask significantly outperformed the baseline in both datasets. We identified the following two reasons:

- (1) AutoTask demonstrates excellent capability in detecting task completion, with precision at 99.4% and recall at 100%. Although the baseline achieves 100% accuracy in verifying task completion, its recall rate is relatively low: 36.1% (15 tasks from PixelHelp and 11 tasks from UGIF) of the errors are due to "overshoot", that is, the baseline executes unnecessary steps after the tasks had already completed.
- (2) AutoTask exhibits higher accuracy in its behavior on the GUI (results from the deciding and checking modules). The step accuracy of AutoTask is 93.5% (PixelHelp: 93.6%, UGIF: 93.4%), whereas the baseline stands at 82.9% (PixelHelp: 77.4%, UGIF: 87.2%).

We analyzed tasks that AutoTask did not complete correctly and categorized the reasons for errors into three main types:

⁸AutoTask still utilizes the first type of environment knowledge accumulated during the execution of the current task.

⁹We guarantee that the selected Type-A tasks do not include the Type-B task to be tested.

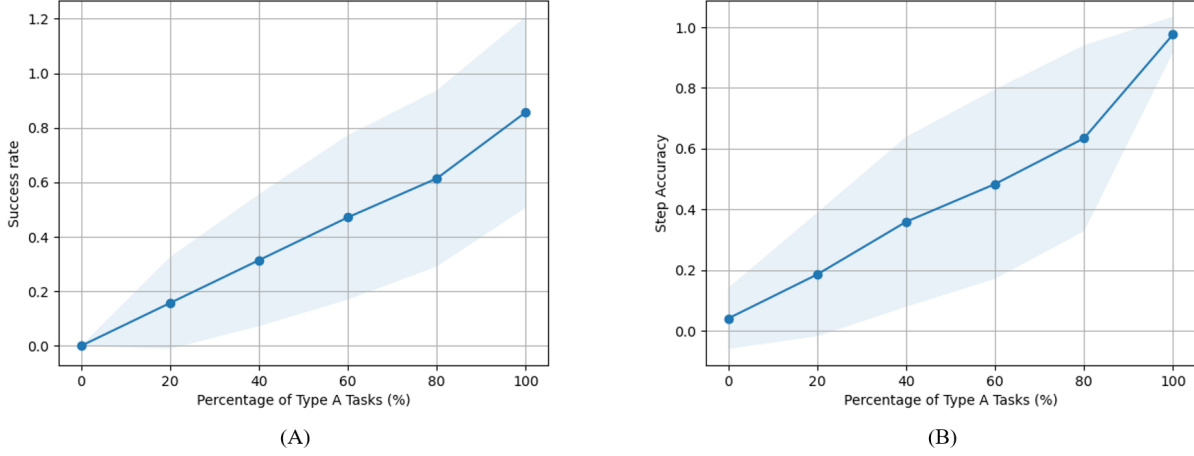


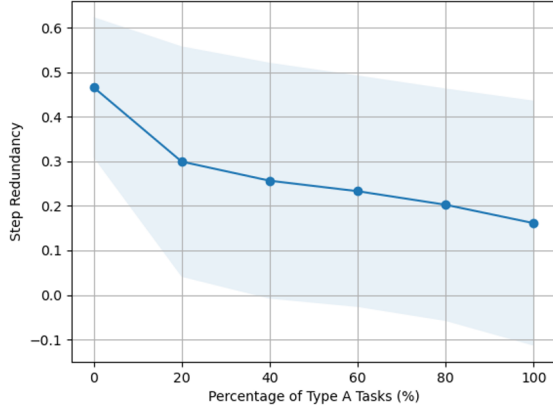
Fig. 6. The success rate (A) and step accuracy (B) of Type-3 tasks increase as the percentage of Type-A tasks from which AutoTask accumulates knowledge increases. The shadow represents the standard deviation.

- (1) AutoTask failed to properly ground instructions to the GUI, leading to blind attempts on the interface. A typical example is the command "check my chromebook if any", where AutoTask did not recognize "chromebook" as a connected device and instead kept clicking irrelevant buttons such as "Display" and "System". Three tasks failed due to this reason.
- (2) AutoTask was misled by information in the command, continuously trying to accomplish irrelevant tasks. For instance, with the command 'lock screen when app unpinning', AutoTask focused on what it perceived as the keyword "lock" and repeatedly tried to add a personal identification number (PIN) to the phone. Nine tasks failed due to this misunderstanding.
- (3) AutoTask mistakenly believed the task was completed. This occurred once in our experiments with the command "show system applications". AutoTask successfully navigated to the "Installed Applications" page and thought the task was finished. However, it was expected to click the "More Options" button and then select the "Show System" option.

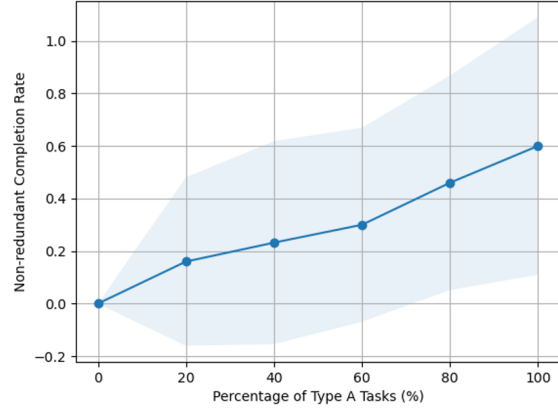
6.6.2 Redundancy. The step redundancy rates of AutoTask across both datasets are 8.54% and 8.96%, significantly lower than the corresponding baseline results (46.0%, $p < 0.001$; 32.0%, $p < 0.001$). AutoTask necessitates backtracking in only 12 (PixelHelp: 7, UGIF: 5) (7.14%) tasks, with an average backtrack count of 2.83 (min=1, max=8, sd = 5.18) within these tasks. This indicates that AutoTask requires minimal backtracking to accomplish tasks.

| Model | Dataset | Success Rate | Step Accuracy | Step Redundancy Rate | Non-redundant Completion Rate |
|----------|-----------|--------------|---------------|----------------------|-------------------------------|
| Baseline | PixelHelp | 52.2% | 77.4% | 46.0% | 65.3% |
| | UGIF | 67.0% | 87.2% | 32.0% | 78.4% |
| AutoTask | PixelHelp | 91.2% | 93.6% | 8.54% | 89.7% |
| | UGIF | 93.0% | 93.4% | 8.96% | 95.0% |

Table 6. Evaluation Results of Baseline LLM (AITW) and AutoTask on PixelHelp dataset and UGIF dataset.

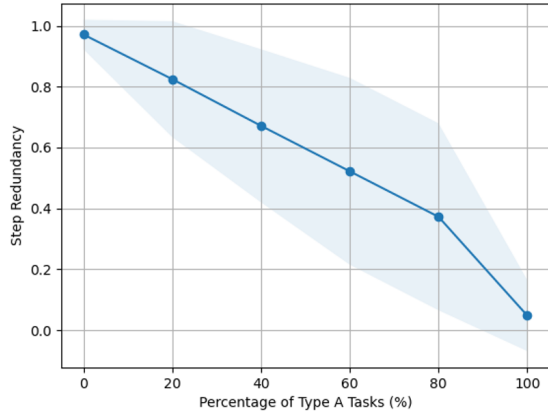


(A)

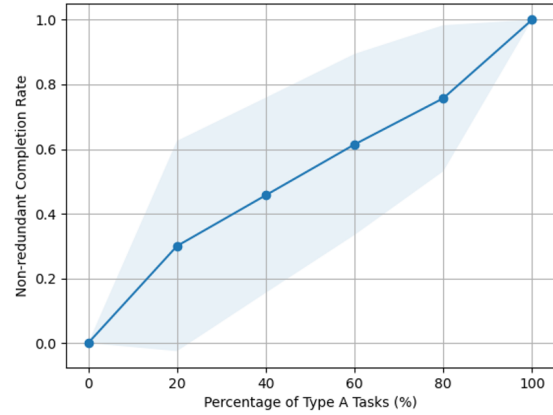


(B)

Fig. 7. The step redundancy rate (A) and non-redundant completion rate (B) of Type-2 tasks change as the percentage of Type-A tasks from which AutoTask accumulates knowledge increases. The shadow represents the standard deviation.



(A)



(B)

Fig. 8. The step redundancy rate (A) and non-redundant completion rate (B) of Type-3 tasks change as the percentage of Type-A tasks from which AutoTask accumulates knowledge increases. The shadow represents the standard deviation.

6.6.3 Performance improvement through knowledge accumulation. The quantities of the three task types in UGIF are 88, 5, and 7, respectively. Figure x illustrates how the success rate and the step accuracy increase with the accumulation of knowledge. When AutoTask learns all the knowledge, these metrics can reach 85.7% and 97.6%, respectively. Only one Type-3 task cannot be completed even after accumulating all the knowledge.

Furthermore, knowledge accumulation effectively improves the efficiency of task completion. Figure 7 & 8 demonstrates how the average step redundancy rate and non-redundant completion rate for Type-2 and Type-3 tasks change as the accumulated knowledge increases. Upon acquiring all available knowledge, the step redundancy rates for these two task types decrease from 46.7% to 16.1% and from 97.1% to 4.76%, respectively, with only 3 tasks (2 Type-2, 1 Type-3) still requiring backtracking.

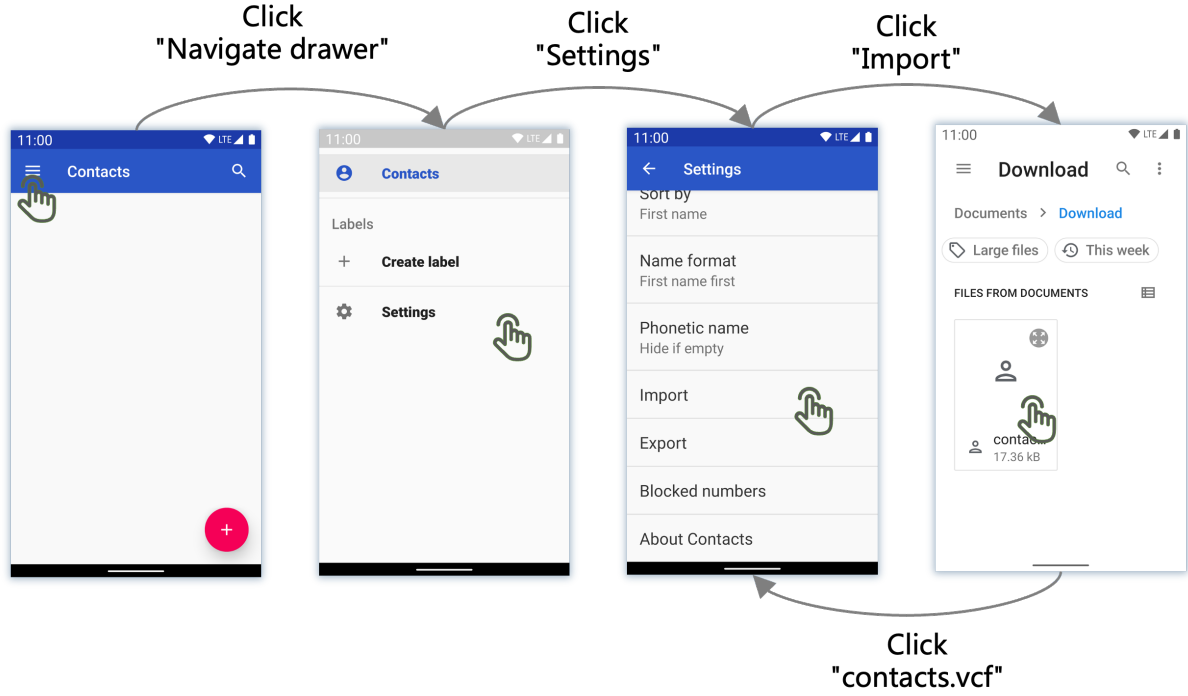


Fig. 9. How AutoTask executes "Import contacts from contacts.vcf" in version 1.7.31 of the Contacts application. Note that the location of this functionality has changed compared to Figure 3.

7 DISCUSSION

7.1 Generalization of Knowledge: Across Versions and Applications

While the study evaluated the performance improvement brought about by knowledge accumulation within the same application, this knowledge can be applied across different versions of the same application or even across applications with similar functionalities (e.g., iMessage vs. WhatsApp). These applications follow similar design principles and semantics [51]. For instance, Figure 9 depicts an earlier version of a contact application (version 1.7.31). In contrast to Figure 3, its home page does not contain "Fix & manage", and "Import from file" is hidden under "Settings". However, AutoTask can still benefit from the knowledge it has summarized. For example, it will refrain from attempting to click the "Add" button, thereby expediting the execution of the command.

The generalization of knowledge across versions and different applications also carries some risks, as the accuracy of such knowledge cannot be guaranteed. For example, if AutoTask first imports contacts from a file in version 1.7.31 of the Contacts application, it may summarize the knowledge that this functionality can be found by clicking the "Settings" button. However, when it attempts the same task in version 4.8.17 of the Contacts application based on this knowledge, it might experience decreased efficiency, even though it can still complete the task correctly through backtracking. This is because the functionality of importing contacts from a file has been relocated in the new version. One solution is to estimate the confidence level of its knowledge. While this goes beyond the scope of this paper, it is a promising direction for future research.

7.2 The Generalization of AutoTask: to Other Devices and Tasks

While we implemented AutoTask on Android smartphones and conducted evaluation experiments, AutoTask can generalize to other devices and platforms (e.g., web browsers [41, 49]) as long as they provide APIs, with which AutoTask can (1) retrieve the current contents on the GUI and (2) simulate user interactions in the GUI. AutoTask may also generalize to non-graphical interfaces, such as command line interfaces [52], to reduce the learning curve and interaction burdens. Besides, AutoTask can be viewed as a proxy for existing GUIs [30, 94], and its interaction modality is not limited to voice interaction. GUI mapping [40] (e.g., mapping a smartphone GUI to a smartwatch GUI [98]) is a typical example. Developers only need to be concerned with the visual mapping rules [12] instead of the execution logic of the applications. AutoTask can automatically operate the original GUI (e.g., smartphone GUI) based on the user’s interaction behaviors on the new GUI (e.g., smartwatch GUI).

AutoTask "accomplishes unknown tasks in an unknown environment", and all tasks that align with this pattern can benefit from the "explore-learn" strategy outlined in this paper with little effort required from developers or end users. For example, crafting prompts that yield high-performance results can be challenging for end users [17, 18, 32, 50, 84]. In this problem, the environment is the LLM that has not been fully explored and the task is to generate an effective prompt. However, the effectiveness is not well-defined. AutoTask can attempt an initial prompt to determine the capability boundaries of the LLM and then refine it (similar to the backtracking process described in this paper) based on the responses while accumulating knowledge to enhance its prompt-generation capabilities. Similar concepts [42] are found in ReAct [92] and Reflexion [70]. However, these approaches do not explicitly summarize knowledge to enhance their capabilities. AutoTask can also extend beyond the digital world into the physical world and be applied in various fields, such as embodied intelligence [10, 21, 56].

8 LIMITATION & FUTURE WORK

AutoTask does not interact with users during its execution. However, user commands may be incomplete or ambiguous [60], and AutoTask should request clarification or additional information when necessary. Additionally, it can proactively ask questions to prune its GUI exploration based on the user’s answers. For example, when a user needs to enable App pinning (Figure 5, Page 3), AutoTask may not be familiar with this feature and attempt various incorrect operations. AutoTask can significantly expedite command execution if it proactively asks the user questions like, "Is this feature related to Security?" to gain relevant insights.

AutoTask utilizes an online LLM (gpt-4) service provided by OpenAI via HTTPS requests, and the computational process is slow. Although optimizing the efficiency of LLMs is beyond the scope of this paper, future work could construct smaller and faster models, for example, through techniques such as knowledge distillation [27], to reduce waiting time for end users.

Users often have complex intents that cannot be covered by a single GUI task [23, 28, 53]. For example, in the command "Send my schedule to Alice", a voice interface is expected to accomplish two tasks (retrieve the schedule and send a message) sequentially. In future work, AutoTask can be combined with complex task decomposition systems [76, 89] to satisfy users’ complex needs.

9 CONCLUSION

In this paper, we present AutoTask, a voice command interface that automates voice commands by simulating GUI interactions. To make it applicable across different applications and GUI tasks, AutoTask requires no configuration or modification from developers or end users. Instead, AutoTask explores the GUI to attempt different operation sequences and accumulates knowledge from these explorations to enhance its capabilities. The evaluation study proves the feasibility of this approach: AutoTask performs significantly better than the baseline when no knowledge is accumulated, and knowledge accumulation further improves its performance. AutoTask addresses the special case in the voice assistant domain of accomplishing unknown tasks in an unknown environment, a

problem pattern that applies to many other similar scenarios. We hope that AutoTask can inspire future work to apply general artificial intelligence to everyday tasks with little effort from the developers or the end users.

REFERENCES

- [1] James Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary Swift, and William Taysom. 2007. Plow: A collaborative task learning agent. In *AAAI*, Vol. 7. 1514–1519.
- [2] Ville Anttila, Jussi Polet, Arttu Lämsä, and Jussi Liikka. 2012. RoutineMaker: Towards end-user automation of daily routines using smartphones. In *2012 IEEE International Conference on Pervasive Computing and Communications Workshops*. IEEE, 399–402.
- [3] Deniz Arsan, Ali Zaidi, Aravind Sagar, and Ranjitha Kumar. 2021. App-Based Task Shortcuts for Virtual Assistants. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 1089–1099.
- [4] Vikas Ashok, Yevgen Borodin, Yury Puzis, and IV Ramakrishnan. 2015. Capti-speak: a speech-enabled web screen reader. In *Proceedings of the 12th International Web for All Conference*. 1–10.
- [5] Vikas Ashok, Yury Puzis, Yevgen Borodin, and IV Ramakrishnan. 2017. Web screen reading automation assistance using semantic abstraction. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces*. 407–418.
- [6] Amos Azaria, Jayant Krishnamurthy, and Tom Mitchell. 2016. Instructable intelligent personal agent. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30.
- [7] Erin Beneteau, Olivia K Richards, Mingrui Zhang, Julie A Kientz, Jason Yip, and Alexis Hiniker. 2019. Communication breakdowns between families and Alexa. In *Proceedings of the 2019 CHI conference on human factors in computing systems*. 1–13.
- [8] Dan Bohus and Alexander Rudnicky. 2005. Error handling in the RavenClaw dialog management architecture. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*. 225–232.
- [9] SRK Branavan, Luke Zettlemoyer, and Regina Barzilay. 2010. Reading between the lines: Learning to map high-level instructions to commands. In *Proceedings of the 48th annual meeting of the association for computational linguistics*. 1268–1277.
- [10] Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, et al. 2023. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*. PMLR, 287–318.
- [11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [12] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676.
- [13] John Joon Young Chung, Woosok Kim, Kang Min Yoo, Hwaran Lee, Eytan Adar, and Minsuk Chang. 2022. TaleBrush: Sketching stories with generative pretrained language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–19.
- [14] Eric Corbett and Astrid Weber. 2016. What can I say? addressing user experience challenges of a mobile voice user interface for accessibility. In *Proceedings of the 18th international conference on human-computer interaction with mobile devices and services*. 72–82.
- [15] Benjamin R Cowan, Nadia Pantidi, David Coyle, Kellie Morrissey, Peter Clarke, Sara Al-Shehri, David Earley, and Natasha Bandeira. 2017. "What can i help you with?" infrequent users' experiences of intelligent personal assistants. In *Proceedings of the 19th international conference on human-computer interaction with mobile devices and services*. 1–12.
- [16] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration*. MIT press.
- [17] Hai Dang, Lukas Mecke, Florian Lehmann, Sven Goller, and Daniel Buschek. 2022. How to prompt? Opportunities and challenges of zero-and few-shot learning for human-AI interaction in creative applications of generative models. *arXiv preprint arXiv:2209.01390* (2022).
- [18] Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric P Xing, and Zhiting Hu. 2022. Rlprompt: Optimizing discrete text prompts with reinforcement learning. *arXiv preprint arXiv:2205.12548* (2022).
- [19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [20] Anind K Dey, Raffay Hamid, Chris Beckmann, Ian Li, and Daniel Hsu. 2004. a CAPpella: programming by demonstration of context-aware applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 33–40.
- [21] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. 2023. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378* (2023).
- [22] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. *arXiv preprint arXiv:2310.03533* (2023).
- [23] Ethan Fast, Binbin Chen, Julia Mendelsohn, Jonathan Bassen, and Michael S Bernstein. 2018. Iris: A conversational agent for complex tasks. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–12.

- [24] Alexander J Fiannaca, Chinmay Kulkarni, Carrie J Cai, and Michael Terry. 2023. Programming without a Programming Language: Challenges and Opportunities for Designing Developer Tools for Prompt Programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–7.
- [25] Tong Gao, Mira Dontcheva, Eytan Adar, Zhicheng Liu, and Karrie G Karahalios. 2015. Datatone: Managing ambiguity in natural language interfaces for data visualization. In *Proceedings of the 28th annual acm symposium on user interface software & technology*. 489–500.
- [26] Melinda T Gervasio, Michael D Moffitt, Martha E Pollack, Joseph M Taylor, and Tomas E Uribe. 2005. Active preference learning for personalized calendar scheduling assistance. In *Proceedings of the 10th international conference on Intelligent user interfaces*. 90–97.
- [27] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision* 129 (2021), 1789–1819.
- [28] Jonathan Grudin and Richard Jacques. 2019. Chatbots, humbots, and the quest for artificial general intelligence. In *Proceedings of the 2019 CHI conference on human factors in computing systems*. 1–11.
- [29] William Grussenmeyer and Eelke Folmer. 2017. Accessible touchscreen technology for people with visual impairments: a survey. *ACM Transactions on Accessible Computing (TACCESS)* 9, 2 (2017), 1–31.
- [30] Tian Huang, Chun Yu, Weinan Shi, Bowen Wang, David Yang, Yihao Zhu, Zhaoheng Li, and Yuanchun Shi. 2023. Interaction Proxy Manager: Semantic Model Generation and Run-Time Support for Reconstructing User Interfaces of Mobile Services. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 7, 3, Article 99 (sep 2023), 39 pages. <https://doi.org/10.1145/3610929>
- [31] Mohit Jain, Pratyush Kumar, Ramachandra Kota, and Shwetak N Patel. 2018. Evaluating and informing the design of chatbots. In *Proceedings of the 2018 designing interactive systems conference*. 895–906.
- [32] Ellen Jiang, Kristen Olson, Edwin Toh, Alejandra Molina, Aaron Donsbach, Michael Terry, and Carrie J Cai. 2022. Promptmaker: Prompt-based prototyping with large language models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–8.
- [33] Jiepu Jiang, Wei Jeng, and Daqing He. 2013. How do users respond to voice input errors? Lexical and phonetic query reformulation in voice search. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*. 143–152.
- [34] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.
- [35] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. 2019. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374* (2019).
- [36] Jihyun Kim, Meul Jeong, and Seul Chan Lee. 2019. " Why did this voice agent not understand me?" error recovery strategy for in-vehicle voice user interface. In *Proceedings of the 11th International Conference on Automotive User Interfaces and Interactive Vehicular Applications: Adjunct Proceedings*. 146–150.
- [37] Yea-Seul Kim, Mira Dontcheva, Eytan Adar, and Jessica Hullman. 2019. Vocal shortcuts for creative experts. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–14.
- [38] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. 2021. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems* 23, 6 (2021), 4909–4926.
- [39] Tessa Lau, Julian Cerruti, Guillermo Manzato, Mateo Bengualid, Jeffrey P Bigham, and Jeffrey Nichols. 2010. A conversational interface to web automation. In *Proceedings of the 23rd annual ACM symposium on User interface software and technology*. 229–238.
- [40] Sunjae Lee, Hayeon Lee, Hoyoung Kim, Sangmin Lee, Jeong Woon Choi, Yuseung Lee, Seono Lee, Ahyeon Kim, Jean Young Song, Sangeun Oh, Steven Y. Ko, and Insik Shin. 2021. FLUID-XP: Flexible User Interface Distribution for Cross-Platform Experience. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (New Orleans, Louisiana) (MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 762–774. <https://doi.org/10.1145/3447993.3483245>
- [41] Gilly Leshed, Eben M Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1719–1728.
- [42] Tao Li, Gang Li, Zhiwei Deng, Bryan Wang, and Yang Li. 2023. A Zero-Shot Language Agent for Computer Control with Structured Reflection. [arXiv:2310.08740](https://arxiv.org/abs/2310.08740) [cs.CL]
- [43] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. 2017. SUGILITE: creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI conference on human factors in computing systems*. 6038–6049.
- [44] Toby Jia-Jun Li, Jingya Chen, Haijun Xia, Tom M Mitchell, and Brad A Myers. 2020. Multi-modal repairs of conversational breakdowns in task-oriented dialogs. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 1094–1107.
- [45] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenzhe Shi, Wanling Ding, Tom M Mitchell, and Brad A Myers. 2018. Appinite: A multi-modal interface for specifying data descriptions in programming by demonstration using natural language instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 105–114.
- [46] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M Mitchell, and Brad A Myers. 2019. Pumice: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Proceedings of the 32nd annual ACM symposium on user interface software and technology*. 577–589.

- [47] Toby Jia-Jun Li and Oriana Riva. 2018. KITE: Building conversational bots from mobile apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 96–109.
- [48] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping natural language instructions to mobile UI action sequences. *arXiv preprint arXiv:2005.03776* (2020).
- [49] Greg Little, Tessa A Lau, Allen Cypher, James Lin, Eben M Haber, and Eser Kandogan. 2007. Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 943–946.
- [50] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. 2023. “What It Wants Me To Say”: Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–31.
- [51] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning Design Semantics for Mobile Apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) (UIST ’18). Association for Computing Machinery, New York, NY, USA, 569–579. <https://doi.org/10.1145/3242587.3242650>
- [52] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2023. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688* (2023).
- [53] Ewa Luger and Abigail Sellen. 2016. “Like Having a Really Bad PA” The Gulf between User Expectation and Experience of Conversational Agents. In *Proceedings of the 2016 CHI conference on human factors in computing systems*. 5286–5297.
- [54] David Massimo, Mehdi Elahi, and Francesco Ricci. 2017. Learning user preferences by observing user-items interactions in an IoT augmented space. In *Adjunct Publication of the 25th Conference on User Modeling, Adaptation and Personalization*. 35–40.
- [55] Christine Murad, Cosmin Munteanu, Benjamin R Cowan, and Leigh Clark. 2019. Revolution or evolution? Speech interaction and HCI design guidelines. *IEEE Pervasive Computing* 18, 2 (2019), 33–45.
- [56] Prasanth Murali, Ian Steenstra, Hye Sun Yun, Ameneh Shamekhi, and Timothy Bickmore. 2023. Improving multiparty interactions with a robot using large language models. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–8.
- [57] Brad A Myers. 1986. Visual programming, programming by example, and program visualization: a taxonomy. *ACM sigchi bulletin* 17, 4 (1986), 59–66.
- [58] OpenAI. 2023. GPT-4 Technical Report. *arXiv:2303.08774* [cs.CL]
- [59] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.
- [60] Lihang Pan, Chun Yu, Zhe He, and Yuanchun Shi. 2023. A Human-Computer Collaborative Editing Tool for Conceptual Diagrams. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–29.
- [61] Lihang Pan, Chun Yu, JiaHui Li, Tian Huang, Xiaojun Bi, and Yuanchun Shi. 2022. Automatically generating and improving voice command interface from operation sequences on smartphones. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–21.
- [62] Hannah RM Pelikan and Mathias Broth. 2016. Why that nao? how humans adapt to a conventional humanoid robot in taking turns-at-talk. In *Proceedings of the 2016 CHI conference on human factors in computing systems*. 4921–4932.
- [63] Chen Qu, Liu Yang, Minghui Qiu, W Bruce Croft, Yongfeng Zhang, and Mohit Iyyer. 2019. BERT with history answer embedding for conversational question answering. In *Proceedings of the 42nd international ACM SIGIR conference on research and development in information retrieval*. 1133–1136.
- [64] Arpit Rana and Derek Bridge. 2020. Navigation-by-preference: a new conversational recommender with preference-based feedback. In *Proceedings of the 25th International Conference on Intelligent User Interfaces*. 155–165.
- [65] Lenin Ravindranath, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. 2012. Code in the air: simplifying sensing and coordination tasks on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*. 1–6.
- [66] Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. 2023. Android in the wild: A large-scale dataset for android device control. *arXiv preprint arXiv:2307.10088* (2023).
- [67] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The programmer’s assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 491–514.
- [68] Minjoon Seo, Jinhyuk Lee, Tom Kwiatkowski, Ankur P Parikh, Ali Farhadi, and Hannaneh Hajishirzi. 2019. Real-time open-domain question answering with dense-sparse phrase index. *arXiv preprint arXiv:1906.05807* (2019).
- [69] Alborz Rezazadeh Sereshkeh, Gary Leung, Krish Perumal, Caleb Phillips, Minfan Zhang, Afsaneh Fazly, and Iqbal Mohamed. 2020. VASTA: a vision and language-assisted smartphone task automation system. In *Proceedings of the 25th international conference on intelligent user interfaces*. 22–32.
- [70] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

- [71] Maayan Shvo, Zhiming Hu, Rodrigo Toro Icarte, Iqbal Mohamed, Allan D Jepson, and Sheila A McIlraith. 2021. AppBuddy: Learning to Accomplish Tasks in Mobile Apps via Reinforcement Learning.. In *Canadian Conference on AI*.
- [72] Arjun Srinivasan, Mira Dontcheva, Eytan Adar, and Seth Walker. 2019. Discovering natural language commands in multimodal interfaces. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*. 661–672.
- [73] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems* 33 (2020), 3008–3021.
- [74] Atsushi Sugiura and Yoshiyuki Koseki. 1996. Simplifying macro definition in programming by demonstration. In *Proceedings of the 9th annual ACM symposium on User interface software and technology*. 173–182.
- [75] Niket Tandon, Aman Madaan, Peter Clark, and Yiming Yang. 2022. Learning to repair: Repairing model output errors after deployment using a dynamic memory of feedback. In *Findings of the Association for Computational Linguistics: NAACL 2022*, Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz (Eds.). Association for Computational Linguistics, Seattle, United States, 339–352. <https://doi.org/10.18653/v1/2022.findings-naacl.26>
- [76] Jaime Teevan, Shamsi T Iqbal, Carrie J Cai, Jeffrey P Bigham, Michael S Bernstein, and Elizabeth M Gerber. 2016. Productivity decomposed: Getting big things done with little microtasks. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. 3500–3507.
- [77] Jesse Thomason, Shiqi Zhang, Raymond J Mooney, and Peter Stone. 2015. Learning to interpret natural language commands through human-robot dialog. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.
- [78] Zhen Tu, Yali Fan, Yong Li, Xiang Chen, Li Su, and Depeng Jin. 2019. From fingerprint to footprint: Cold-start location recommendation by learning user interest from app data. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 3, 1 (2019), 1–22.
- [79] Sagar Gubbi Venkatesh, Partha Talukdar, and Srini Narayanan. 2022. UGIF: UI Grounded Instruction Following. *arXiv preprint arXiv:2211.07615* (2022).
- [80] Tom Veuskens, Kris Luyten, and Raf Ramakers. 2020. Rataplan: Resilient Automation of User Interface Actions with Multi-Modal Proxies. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 2, Article 60 (jun 2020), 23 pages. <https://doi.org/10.1145/3397329>
- [81] Minh Duc Vu, Han Wang, Zhuang Li, Gholamreza Haffari, Zhenchang Xing, and Chunyang Chen. 2023. Voicify Your UI: Towards Android App Control with Voice Commands. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 7, 1 (2023), 1–22.
- [82] Amber Wagner. 2013. Automation of VUI to GUI Mapping. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems* (Paris, France) (CHI EA '13). Association for Computing Machinery, New York, NY, USA, 1941–1944. <https://doi.org/10.1145/2468356.2468706>
- [83] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling conversational interaction with mobile ui using large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–17.
- [84] Yunlong Wang, Shuyuan Shen, and Brian Y Lim. 2023. RePrompt: Automatic Prompt Editing to Refine AI-Generative Art Towards Precise Expressions. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–29.
- [85] Wayne Ward and Sunil Issar. 1994. Recent improvements in the CMU spoken language understanding system. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*.
- [86] Xingjiao Wu, Luwei Xiao, Yixuan Sun, Junhang Zhang, Tianlong Ma, and Liang He. 2022. A survey of human-in-the-loop for machine learning. *Future Generation Computer Systems* 135 (2022), 364–381.
- [87] Linda Wulf, Markus Garschall, Julia Himmelsbach, and Manfred Tscheligi. 2014. Hands free-care free: elderly people taking advantage of speech-only interaction. In *Proceedings of the 8th Nordic Conference on Human-Computer Interaction: Fun, Fast, Foundational*. 203–206.
- [88] Liang Xu, Hai Hu, Xuanwei Zhang, Lu Li, Chenjie Cao, Yudong Li, Yechen Xu, Kai Sun, Dian Yu, Cong Yu, et al. 2020. CLUE: A Chinese language understanding evaluation benchmark. *arXiv preprint arXiv:2004.05986* (2020).
- [89] Hui Yang, Sifu Yue, and Yunzhong He. 2023. Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions. *arXiv preprint arXiv:2306.02224* (2023).
- [90] Nicole Yankelovich. 1996. How do users know what to say? *interactions* 3, 6 (1996), 32–43.
- [91] Nicole Yankelovich, Gina-Anne Levow, and Matt Marx. 1995. Designing SpeechActs: Issues in speech user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 369–376.
- [92] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).
- [93] Jennifer Zamora. 2017. I’m sorry, dave, i’m afraid i can’t do that: Chatbot perception and expectations. In *Proceedings of the 5th international conference on human agent interaction*. 253–260.
- [94] Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O Wobbrock. 2017. Interaction proxies for runtime repair and enhancement of mobile application accessibility. In *Proceedings of the 2017 CHI conference on human factors in computing systems*. 6024–6037.
- [95] Ye Zhang and Byron Wallace. 2015. A sensitivity analysis of (and practitioners’ guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820* (2015).

- [96] Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. 2023. Towards an understanding of large language models in software engineering tasks. *arXiv preprint arXiv:2308.11396* (2023).
- [97] Yu Zhong, TV Raman, Casey Burkhardt, Fadi Biadsy, and Jeffrey P Bigham. 2014. JustSpeak: enabling universal voice control on Android. In *Proceedings of the 11th Web for All Conference*. 1–4.
- [98] Zhilan Zhou, Jian Xu, Aruna Balasubramanian, and Donald E. Porter. 2020. A Survey of Patterns for Adapting Smartphone App UIs to Smart Watches. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services* (Oldenburg, Germany) (*MobileHCI '20*). Association for Computing Machinery, New York, NY, USA, Article 2, 11 pages. <https://doi.org/10.1145/3379503.3403564>
- [99] Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. 2019. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593* (2019).