RESEARCH-ARTICLE

VUI Testing of VPA Apps via Behavior Model-Enhanced LLM Agents

SUWAN LI, Nanjing University, Nanjing, Jiangsu, China

LEI BU, Nanjing University, Nanjing, Jiangsu, China

SHANGQING LIU, Nanjing University, Nanjing, Jiangsu, China

GUANGDONG BAI, The University of Queensland, Brisbane, QLD, Australia

FUMAN XIE, The University of Queensland, Brisbane, QLD, Australia

KAI CHEN, Institute of Information Engineering, Beijing, China

View all

Published: 10 October 2025
Accepted: 30 September 2025
Revised: 29 July 2025
Received: 08 March 2025

VUI Testing of VPA Apps via Behavior Model-Enhanced LLM Agents

SUWAN LI, State Key Laboratory for Novel Software Technology, Nanjing University, China
LEI BU*, State Key Laboratory for Novel Software Technology, Nanjing University, China
SHANGQING LIU*, State Key Laboratory for Novel Software Technology, Nanjing University, China
GUANGDONG BAI, the University of Queensland, Australia
FUMAN XIE, the University of Queensland, Australia
KAI CHEN, Institute of Information Engineering, Chinese Academy of Sciences, China
CHANG YUE, Institute of Information Engineering, Chinese Academy of Sciences, China

With the increasing adoption of smart speakers, Virtual Personal Assistant (VPA) applications have become integral to daily life, enabling users to access news, entertainment, and smart device control through Voice User Interfaces (VUI). However, many VPA apps suffer from quality issues, such as unexpected terminations and failures to process common user commands, highlighting the urgent need for systematic and efficient VUI testing. Existing chatbot-style and model-based testing approaches lack global and semantic awareness, resulting in ineffective test case generation and inefficient state exploration.

To address these challenges, we introduce Elevate, a model-enhanced, LLM-driven VPA testing framework that employs a multi-agent architecture to enhance VUI behavior testing. Elevate comprises three specialized LLM agents—Observer, Generator, and Planner—that collaboratively perform state extraction, test case generation, and guided state exploration. Additionally, a deterministic finite automaton (DFA)-based behavior model is designed to abstract app behavior and provide structured guidance to LLM agents, enhancing testing performance. Elevate also incorporates a feedback mechanism that refines testing strategies based on observed behaviors, ensuring continuous improvement.

Implemented using GPT-4-Turbo and DeepSeek-R1, Elevate has been evaluated on problem detection, sentence/semantic coverage, and large-scale testing. Experimental results show that Elevate outperforms state-of-the-art methods (Vitas and LLM-based chatbots), detecting at least 18 and 37 more problems, respectively, and achieving over 10% and 30% higher state coverage. In a large-scale evaluation on 4,000 Alexa skills, Elevate further demonstrated 15% higher coverage than Vitas, confirming its effectiveness, scalability, and potential for widespread application in VUI testing.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: VUI Testing, Large Language Model, Model-based Testing

*Corresponding authors.

Authors' addresses: Suwan Li, State Key Laboratory for Novel Software Technology, Nanjing University, China, lisuwan@smail.nju.edu.cn; Lei Bu, State Key Laboratory for Novel Software Technology, Nanjing University, China, bulei@.nju.edu.cn; Shangqing Liu, State Key Laboratory for Novel Software Technology, Nanjing University, China, shangqingliu@.nju.edu.cn; Guangdong Bai, the University of Queensland, Australia, g.bai@uq.edu.au; Fuman Xie, the University of Queensland, Australia, fuman.xie@uq.edu.au; Kai Chen, Institute of Information Engineering, Chinese Academy of Sciences, China, chenkai@iie.ac.cn; Chang Yue, Institute of Information Engineering, Chinese Academy of Sciences, China, yuechang@iie.ac.cn.

ACM 1557-7392/2025/10-ART
https://doi.org/10.1145/3771555

1 INTRODUCTION

With the widespread adoption of smart speakers, virtual personal assistants (VPAs) have become an integral part of daily life. Prominent examples include Amazon Alexa, Google Assistant, and Apple Siri, which are widely used to assist smart speaker users. To enhance their functionality, a vast ecosystem of virtual personal assistant applications (VPA apps) has emerged, providing a diverse range of services, including news updates, entertainment, and smart device control. One of the key advantages of VPA apps is their seamless accessibility—users can invoke these apps without installation, simply by speaking their designated invocation names. This ease of access has significantly contributed to their widespread popularity. For instance, the Alexa Skills Store, the largest marketplace for VPA apps, currently hosts over 200,000 applications [5]. A defining characteristic of VPA apps is their Voice User Interface (VUI), which enables users to interact solely through spoken dialogue, distinguishing them from traditional graphical or text-based applications. Despite their widespread adoption, the quality concerns surrounding VPA apps and their VUI have become increasingly prominent. Many VPA apps suffer from poor quality, exhibiting problems such as unexpected terminations [28] or failures to correctly process common user commands [41], leading to frustrating user experiences. The widespread prevalence of such VUI-related issues in VPA apps, underscores the urgent need for a comprehensive testing methodology.

Systematic and efficient VUI testing has been the recent subject of extensive research. Current methodologies predominantly employ chatbot-style testing techniques [15, 17, 21, 34, 38] or model-based testing [28]. Chatbot-style testing techniques rely on generating test cases for behavior exploration at the *textual level*. Specifically, chatbot-style testers are provided with *limited context*—such as the most recent output—to interact with VPA apps. Techniques like depth-first search (DFS) or strategies guided by local information are commonly utilized for exploring the state space. However, this local test cases generation and selection strategy prevents these testers from continuously refining the testing strategies based on relevant historical test information. The model-based testing technique is broadly used in various testing domains and has the advantage of effective exploration guided by the model. However, traditional model-based testing faces limitations when being applied to VUI testing. Semantic information is crucial in model-based VUI testing, especially in states identification and test cases generation. An inaccurate model could prevent the tester from adopting effective testing strategies for state space exploration. In conclusion, these techniques lack *global* and *semantic* information to conduct efficient VUI testing.

A promising solution to mitigate semantic loss in VUI testing is the integration of Large Language Models (LLMs). LLMs have been successfully applied in natural language processing (NLP) [23, 33, 35, 42] and semantics-based software testing [27, 30, 31], demonstrating strong contextual understanding and the ability to bridge gaps in semantic information. Inspired by these advancements, we leverage LLMs for VUI testing for enhancement. However, a systematic VUI testing process involves multiple interdependent tasks, such as extracting relevant information from VUI outputs, generating effective test cases, and strategically exploring app behavior. While a single LLM can handle simple tasks, it lacks the ability to perform multitasking, parallel processing, and collaborative decision-making required for systematic VUI testing. To address these challenges, we design a multi-agent LLM framework composed of *three specialized LLM agents*. Each agent is tasked with a distinct function, enhancing their collective ability to handle complex tasks.

Furthermore, to provide the testing system with global guidance, we propose embedding historical testing information into the LLM agents' context. However, LLMs have limited context length and may struggle to effectively summarize and conceptualize long, domain-specific contexts [22], such as previous communication logs in VPA app testing. To address this challenge, we introduce a model-enhanced approach that optimizes LLMs' capabilities in VUI testing. This model serves as an abstraction of VPA apps' behavior, progressively constructed through functionality-level states and context-related input events. During testing, task-specific model information is extracted and integrated into each agent's prompt, providing a structured overview of

app behavior and enabling the delivery of targeted feedback to the LLM agents. This ensures that the testing process remains informed, context-aware, and efficient, overcoming the limitations of LLMs' context length while enhancing their ability to explore VPA apps comprehensively.

To this end, we introduce Elevate (model-Enhanced Llm-drivEn Vpa App's vui TEsting), a framework designed to systematically and effectively test VUI behavior. Elevate integrates a multi-agent architecture with model-enhancement technology to compensate for semantic loss at every stage of VUI testing while providing global guidance to LLM agents through task-specific information derived from a dynamically constructed behavior model. The framework consists of three specialized LLM agents: Observer, which extracts and assimilates information from the VUI as a foundation for test generation; Generator, which produces contextually relevant test cases; and Planner, which conducts state space exploration, selecting efficient test cases to uncover novel behaviors. To enhance adaptability, Elevate incorporates a feedback mechanism, enabling agents to learn from VPA app responses and domain-specific knowledge, thereby continuously refining their performance. Additionally, a finite state machine (FSM)-based behavior model is incrementally built throughout the testing process, with states extracted from the Observer's real-time perception of VUI outputs and input events generated by the Generator based on contextual relevance. Only information relevant to the current state is embedded into the Planner's prompts to optimize exploration, while the behavior model also serves as a reference for the feedback mechanism, ensuring that LLM agents maintain consistency with the representation of the VPA app's behavior. By combining multi-agent collaboration, feedback-based mechanism, and an adaptive behavior model, Elevate provides a robust, semantically-aware, and efficient framework for comprehensive VUI testing.

Elevate has been implemented using GPT-4-Turbo [7] and DeepSeek-R1 [9] and evaluated to assess its problem detection capabilities and state space coverage. When compared to the state-of-the-art (SOTA) tool [28] and an LLM-based chatbot, Elevate detects at least 18 and 37 more problems, respectively, while achieving over 10% and 30% increases in state space coverage on the benchmarked apps. Additionally, an ablation study highlights the critical role of each LLM agent and the behavior model in achieving comprehensive state exploration. In a large-scale evaluation involving 4,000 Alexa skills, Elevate further demonstrated a 15% higher coverage than Vitas, reinforcing its superior performance and potential for widespread application in VUI testing. The key contributions of this paper are summarized as follows:

- We propose Elevate, a novel approach that employs three specialized LLM agents working in collaboration to address the complex challenges of VUI testing. By distributing responsibilities among these agents, Elevate effectively handles state extraction, test case generation, and guided state exploration, ensuring a systematic and semantics-aware testing process.
- We design a behavior model based on deterministic finite automata (DFA) to abstract VPA apps' behavior and integrate it into the LLM agents. This incorporation enhances the agents' testing performance by providing global guidance, enabling a more accurate testing process.
- Elevate is implemented using GPT-4-Turbo and DeepSeek-R1. Extensive experiments, evaluating problem detection, sentence/semantic coverage rate, and large-scale testing, confirm that Elevate significantly outperforms both the previous state-of-the-art approach, Vitas, and LLM-based chatbots. Our code is available at the website [10].

2 BACKGROUND

2.1 VPA Apps

VPA apps are cloud-based apps on smart speakers. Without downloading or installing, users can invoke these VPA apps by saying the invocation name. This unique invocation name allows VPAs built on these smart speakers to target the VPA apps and bridge the communication channel between users and apps. VPA apps are designed on the voice user interface (VUI). The VUI allows users to interact with VPA apps purely through voice. During the

communication, users' voice commands are firstly translated into texts through the speech recognition system. Through natural language processing, texts are analyzed to match a predefined intent. After that, the handler related to that intent is triggered and voice responses are returned.

To implement a VPA app, developers define front-end intents and back-end handlers. Intents specify the scope of inputs, while handlers define the functionalities and generate textual outputs. Each user input is processed by a fixed back-end handler because this input maps to a single intent, and each intent is handled by a deterministic handler. Non-deterministic behavior occurs only within the handler. For example, a handler may randomly select an output from a predefined list of responses. In summary, a VPA app's behavior is deterministic at the handler (functionality) level, but non-deterministic at the output (sentence) level.

Unlike the graphical user interface (GUI), VUIs are typically free of visible interfaces. Although the VUI brings convenience, its invisible feature introduces a range of quality, security and privacy concerns, such as unexpected exits [28], privacy violations [16, 40], and expected apps started [3, 26]. For this reason, thoroughly exploring VPA apps' behavior while testing the VUI's quality and security issues is of paramount importance.

However, VPA apps are not open source for normal testers. Every VPA app is composed of the front-end interaction model and the back-end processing code. The development platform provides storage for the front-end interaction model, while the back-end code of VPA apps is stored on the developer's server. As a result, dynamic black-box testing is a commonly used method for testing the VUI of VPA apps. Aimed at the official and individual testers, these testing techniques help explore the behavior of VPA apps. Any problems during runtime can be easily detected by implementing oracles on these testers.

2.2 Model-based Testing and Behavior Model

Since the front-end interaction model of VPA apps is designed based on implicit models [4], a model-based testing approach is feasible to explore the behavior of VPA apps. The traditional model-based testing approach is composed of two parts, model construction and test cases generation. The model is first built and then used to guide the exploration of the state space. However, when testing VPA apps, the initial model is difficult to acquire before interacting with VPA apps as the VPA apps are closed-source and most documents only provide a few lines to describe their functionalities. Therefore, VPA apps' behavior model is constructed on-the-fly, which means the model is built during the interaction.

In the scenario of VPA apps, models are constructed based on the communications between users and apps, which are essentially natural languages. Their communications involve several interaction rounds. A user's input and a VPA app's output form an interaction round. The VPA app's outputs may contain rich information, including expected inputs, the current context, apps' purposes and functionalities. By understanding and analyzing VPA app's outputs, states can be extracted. Users' inputs act as a trigger, which makes apps' states transfer from one to another. Based on the characteristics of VPA apps, the finite automaton is a commonly used approach to represent VPA apps' behavior [28]. As mentioned in Section 2.1, the behavior of VPA apps is deterministic at the functionality level, but non-deterministic at the sentence level. Our goal is to construct a high-level model to guide testing; therefore, we use a deterministic finite automaton (DFA) to describe VPA app behavior at the functionality level. This model is referred to as the **behavior model**. Here are the important elements and their compositions in the behavior model.

- **States.** A state represents a distinct functionality and is extracted based on the textual content of the VPA apps' outputs. Each state is represented by the text of the first output mapped to it. When an output cannot be matched to any existing state, a new state is created using the textual content of that output.
- **Input events.** Input events represent the user inputs received by the VPA apps under test. Each input event is represented by the textual content of the corresponding user utterance.

- **Transitions.** Transitions represent changes in the VPA apps' behavior triggered by user inputs. A transition is defined by a transition function of the form $\delta(s_1, input_1) = s_2$, which means that given the current state s_1, receiving the input event $input_1$ causes the VPA app to move to a new state s_2.

Fig. 1 illustrates an example to extract the behavior model from interaction records. The behavior model is incrementally built by mapping outputs to states, inputs to input events, and transitions that link outputs by inputs. Specifically, when the first output — "Ok, Here's Action movies. Welcome to action movies. Do you want me to tell you a movie?" — is received, a state is extracted based on this output. Since the only existing state in the behavior model is the initial state "<START>", a new state with the content "Do you want me to tell you a movie" is created. A transition from the initial state to this newly extracted state is then added to the transition set. This transition is represented in the graph as an edge labeled "Alexa, open action movies", going from the node "<START>" to the node "Do you want me to tell you a movie". As the interaction progresses, the behavior model is incrementally constructed through similar updates.

Different approaches extract states at varying levels of granularity. If the behavior model is constructed at the sentence level, responses such as "I tell you another movie." and "I tell you another one" would be treated as different states. As a comparison, the behavior model in Fig. 1 is built at a higher abstraction level, where outputs with different surface forms but similar functionalities are mapped to a single state. In this example, "I tell you another movie." and "I tell you another movie" are grouped into the same state because they convey the same functionality — asking if the user needs another movie recommendation.

2.3 Common Problems in VPA apps

As summarized in previous work [17, 21, 26, 28, 37–39], there are five common types of weaknesses in VPA apps: ① Unexpected Exit, ② Privacy Violation, ③ Unstoppable App, ④ Unexpected App Started, and ⑤ Unstartable App. Speech recognition and dialog flow bugs are not included from this section, because these issues are caused by the service provider rather than the developers. Among these weaknesses, the first three are runtime weaknesses, while the last two occur before runtime. Their manifestations and oracles are introduced below.

- Unexpected Exit. This weakness occurs when a VPA app quits without any explicit signals. It is detected if the VPA app fail to respond to user requests and does not provide explicit farewell messages, such as "Goodbye", beforehand.
- Privacy Violation. Privacy violation occurs when a VPA app requests users' privacy information without permission. It is detected if the app's outputs contain a wh-question or an instruction question that asks for sensitive privacy information.
- Unstoppable App. This weakness occurs when users send termination inputs but the app fails to exit. The VPA takes control after the VPA app quits. Therefore, this weakness is detected if the speaker in the following interaction is the VPA app instead of the VPA.
- Unexpected App Started. This weakness occurs when a different VPA app starts instead of the expected one. Before a VPA app launches, the VPA announces the app in a fixed format: "Ok, Here's <app name>". Therefore, this weakness is detected if the announced <app name> does not match the expected app's name.
- Unstartable App. This weakness occurs when an app is available in the store but fails to launch properly. It is detected when users use the correct phrase to request a VPA app, but no app starts.

As runtime problems are detected by analyzing the VPA apps' outputs, each problem is associated with the state that corresponds to the problematic output. In particular, Unexpected Exit is recorded at the last state before the app unexpectedly exits; Privacy Violation is recorded at the state where the app asks for privacy information; and Unstoppable App is recorded at the state preceding the user's attempt to terminate the interaction. Two problems are considered different only if differ in type or are associated with different states.

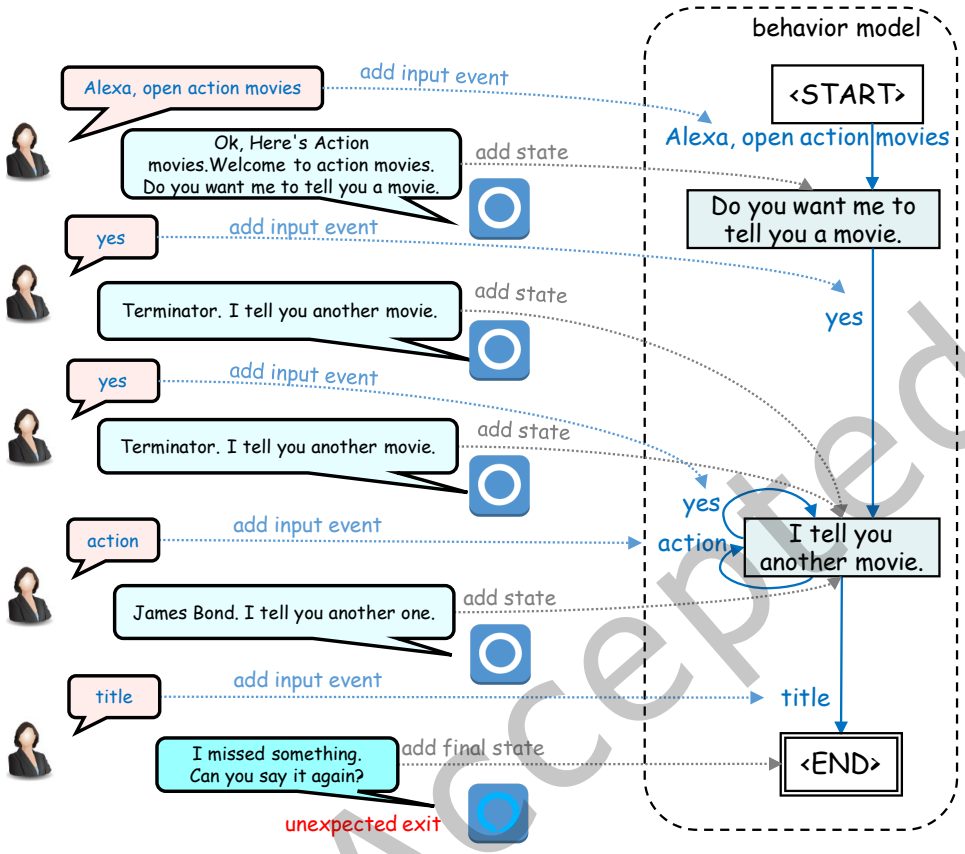


Fig. 1. The example to extract the behavior model from interaction records.

2.4 Motivation

Current VUI testing methodologies mainly follow two paradigms: chatbot-style techniques [17, 21, 34, 38] or model-based testing [28]. Chatbot-style methods generate test inputs at the textual level, typically relying only on the most recent output from the app under test. These methods often use pre-defined NLP rules to generate real-time inputs, but the limited context restricts their ability to adopt effective exploration strategies. Model-based testing methods attempt to compensate for the lack of an end-to-end testing overflow by constructing a behavior model. However, existing model-based methods usually build the behavior model at the sentence level, mapping any outputs with different textual content to separate states, regardless of their semantic similarity. This sentence-level granularity introduces two main limitations: 1) a redundant state space, and 2) the inability to perform testing at the functional level due to the lack of semantic analysis in state extraction.

In conclusion, existing VUI testing techniques have two major limitations: ① *Limited global perspective on the testing process* – These techniques lack a comprehensive, end-to-end view of the entire testing workflow. ② *Inability to accurately interpret semantic information in text responses and text inputs* – Current methods struggle to fully comprehend the semantic meaning embedded in VPA-generated textual outputs and tester-generated textual inputs.

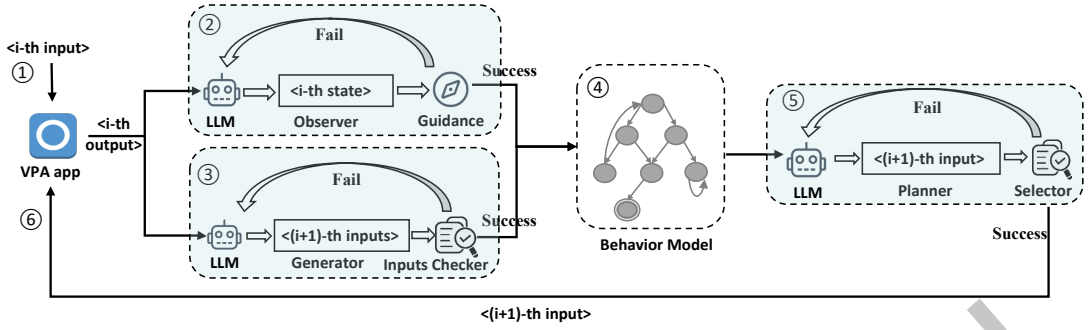


Fig. 2. The framework of Elevate.

However, providing a global perspective during testing is challenging. Using the entire context can lead to an explosion in context size, making processing inefficient and computationally expensive. Moreover, not all contextual information is relevant or necessary. Thus, a limited context is selected, but it leads to the limited context failing to provide a comprehensive view of VPA apps' behavior.

The semantics in the textual response and textual input cannot be comprehended by the existing techniques. This issue manifests in two key aspects: In terms of selecting the next-round input for testing, the previous techniques [15, 17, 21, 28, 34] usually consider quantitative factors such as invocation times or functionality coverage as the select criteria, it fails to comprehend the underlying semantics. For example, choosing the candidate "yes" or "no" for the response "I heard that you want your pet's name to be Shadow. Is that correct?", it is obvious that "yes" is more likely to trigger unseen behavior than "no", but the previous techniques fail to understand the semantics and consider "yes" and "no" with the same possibility to be selected. In terms of state extraction, the previous techniques just merge the outputs with identical content to one state, however, outputs with similar functionalities can be expressed in multiple ways. Although some NLP techniques such as text2vec [8] have certain abilities to group semantic similar text, when faced with significant textual variations, they remain incapable of handling the discrepancies effectively. For example, the semantic similarity of "Goodbye" and "Talk to ya later" calculated by text2vec is only 75.8%. But both of them express the similar semantic meaning.

To overcome the first limitation, we propose the construction of a behavior model at the functionality level to abstract the behavior of VPA applications. This model provides a global perspective for test case generation, ensuring a more structured and comprehensive testing process. To address the second limitation, we introduce three LLM-based agents designed to enhance semantic comprehension in textual responses and textual inputs. These agents work collaboratively to accurately comprehend the semantics during VUI testing.

3 BEHAVIOR MODEL-ENHANCED LLM AGENTS FOR VPA APPS TESTING

3.1 Overview

In this section, we introduce our proposed model-Enhanced LLM-driven vPA application testing technique for voice user interfaces (VUIs), referred to as Elevate. This approach leverages the collaboration of multiple LLM agents to interact with VUI outputs, generate context-aware inputs, construct a behavior model, and devise efficient planning strategies. As shown in Fig. 2, Elevate comprises three agents, each responsible for distinct functions, where the **Observer** agent is responsible for the state extraction, the **Generator** agent tasks with generating input events and the **Planner** agent dedicates to exploring the state space.

The general process of Elevate works as follows. The initial state <0th state> is set as "<START>" and the <1st input> is set as the invocation input (e.g., in Alexa skill the invocation input is always in the form of "Alexa open

+ <skill's invocation name>"). After ① executing the <i-th input>, Elevate receives the <i-th output> from VPA apps. Based on the <i-th output>, Elevate executes Observer and Generator to ② extract the <i-th state> and ③ generate the <(i+1)-th inputs>. The <i-th state> and <the (i+1)-th inputs> are used for the ④ behavior model construction. Subsequently, it extracts information related to the i-th state from the behavior model and embeds it to the prompt of Planner. Planner is expected to ⑤ explore VPA apps' new behavior by selecting an effective and efficient input event, labeled as the <(i+1)-th input>, from the <(i+1)-th inputs>. The <(i+1)-th input> will be used for next-round testing. The whole process will continue until the maximum iteration times are reached or the VPA apps quit. To facilitate a clear presentation, we introduce the following notations:

- <i-th output>: the VPA apps' output in the i-th interaction round. States are extracted for this output. Context-related inputs are generated based on its content.
- <i-th state>: the state extracted for <i-th output>.
- <i-th inputs>: the set of context-related inputs generated based on the content of the <(i-1)-th output>.
- <i-th input>: the input selected by Planner from <i-th inputs>to communicate with the VPA apps.
- <model>: the behavior model.
- <model.Q>: the set of states in the behavior model.
- <model.$\Sigma(s)$>: the input event information of state s, including their invocation times.
- <model.$\delta(s)$>: the set of transition functions that start from state s.
- <model.F>: the set of final states in the behavior model.

3.2 Observer Agent

The finite automaton has been utilized in VPA app testing [28] to represent changes in the application's states. Our behavior model is based on a deterministic finite automaton (DFA), as it is constructed at the functionality level. However, in practice, we can only obtain sentence-level information from the real-time outputs of VPA apps. Due to inherent randomness within handlers, VPA apps exhibit non-deterministic behavior at the sentence level. Therefore, it is necessary to analyze each output in order to extract its underlying functionality and corresponding state. We observe that outputs with similar functionalities tend to convey similar semantics; we refer to these outputs as **semantically similar outputs**. Observer is designed to detect such outputs and map them to a single state.

Unlike prior approaches [28], which primarily rely on text similarity for state extraction, we propose an innovative methodology that employs large language models (LLMs) to achieve a more precise state extraction and mergence process. The primary advantage of LLMs lies in their ability to understand and group semantically similar natural language outputs more effectively than traditional text similarity-based methods, thereby enabling more robust and accurate construction of the behavior model. In particular, as shown in Fig. 2, the observer agent has two interactive processes, namely conversation-based state extraction and mergence and rule-based guidance.

3.2.1 Conversation-based State Extraction and Mergence. For the i-th output of VPA apps, represented as <i-th output>, the prompt used for querying LLMs is defined in Fig.3. This prompt is designed to either align the <i-th output> with an existing state in the behavior model or generate a new state if no alignment is possible. As depicted in Fig.3, if the LLM outputs an existing state from the state list <model.Q> when given the input <i-th output>, it indicates that the LLM has determined the current input to be semantically similar to an existing state. Alternatively, if the model outputs a result identical to the input, it signifies that the LLM has identified the input as representing a new state, thereby generating it accordingly.

As illustrated in Fig. 3, during the initial query to the LLM with the input <i-th output > where $i = 1$, a long prompt is utilized. This prompt includes detailed task instructions along with three few-shot examples structured as "Input: <VPA app's output>, <model.Q>" and "Output: <state>" pairs, designed to assist the LLM

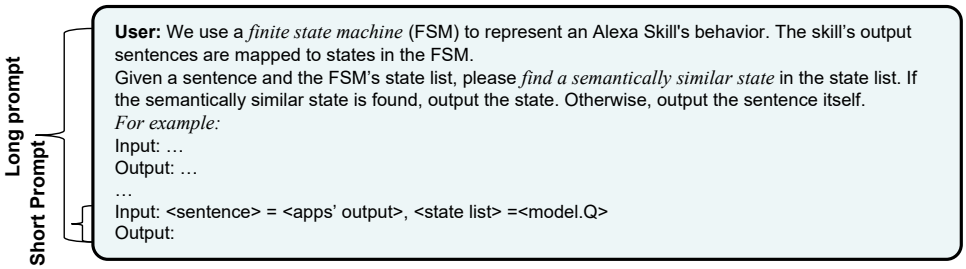


Fig. 3. The prompt used of Observer for the state extraction.

Feedback:
No State Error: The <state> is *not in* the state set <model.Q>. Please find a semantically similar state from the state set.
Not Merge Suggestion: The <apps' output> and <state> are *not semantically similar* because their input events do not have intersection.
Should Merge Suggestion: The <app's output> and <state> are *semantically similar*.

Fig. 4. The used feedback prompt for refinement.

in comprehending the task. For example, we include the following few shots: *Input: sentence: "Come on, ask for another animal.", state list: ["What are you interested in?"] Output: "Come on, ask for another animal."*. This example shows that "Come on, ask for another animal." conveys a different functionality from "What are you interested in?", and thus, a new state should be generated. Conversely, in the example *Input: sentence: "Alright, now ask me for another animal.", state list: ["What are you interested in?", "Come on, ask for another animal."] Output: "Come on, ask for another animal."*, the sentence "Alright, now ask me for another animal." and "Come on, ask for another animal." display the same functionality, so "Alright, now ask me for another animal." is mapped to the existing state "Come on, ask for another animal. Subsequently, for inputs <i-th output >where $i >= 2$, a shorter prompt is employed. This short prompt includes only the state list <model.Q> and the current input, ensuring efficiency while maintaining query accuracy.

For the <i-th output>, if the generated state does not satisfy the guidance (refer to Section 3.2.2), a feedback prompt containing the feedback information provided by the guidance is incorporated into the ongoing conversation turn. This mechanism is designed to refine and improve the LLM's output iteratively.

3.2.2 Rule-based Guidance. Although LLMs have demonstrated their capability to comprehend the outputs of applications, their inherent black-box nature and the resulting lack of control over generated results present significant challenges. To mitigate these issues, we propose a rule-based guidance approach aimed at directing the LLMs toward producing accurate outputs. Specifically, we design three checkers to evaluate the correctness of the outputs. If the LLM's output fails to meet the verification criteria of any checker, a corresponding error feedback, as illustrated in Fig. 4, is provided to the LLM to facilitate refinement and improve the output. The iteration will be continue until the LLM's output pass all checkers or the maximum iteration times is reached.

- *Availability Checker*. It guarantees that the extracted state by LLMs is in the right format. According to the designed prompt in Fig. 3, the LLM's output is restricted to either an existing state from the state set <model.Q> or the output itself, indicating a completely new state not present in the set. Any result that does not conform to this output format will trigger a No State Error.

- *Input Event Consistency Checker*. Two semantically similar outputs, both of which are non-wh outputs, should share the same input events in the subsequent turn inputs [21]. For example, there is an output1, "Some possible names are: Hunter, Cuddles or Biscuit. Either choose one of those names, or say, 'more', to hear more choices." and an output2 "Some other possible names are: Shadow, Hunter, or Snowball. Either choose one of those names, or say, 'more', to hear more choices.". The first output is a non-wh output and its potential input for the next turn is "Hunter", "Cuddles", "Biscuit" and "more". Similarly, the second output is also a non-wh output with the expected inputs "Shadow", "Hunter", "Snowball" and "more". Both of them share the input event "Hunter" and "more" for the subsequent potential input. We can find that output1 and output2 are semantically similar outputs. Thus, we compare the input events between two outputs to check the correctness of the LLM results. In particular, for the <(i+1)-th inputs> generated by the Generator agent (See Section 3.3) from the original output <i-th output>, we compare the input event sets between the extracted state <i-th state> and <(i+1)-th inputs>. If two sets have no intersection, the input event consistency check will raise the Not Merge Suggestion to indicate that Observer should not map the <i-th output> to the <i-th state>.
- *Transition Checker*. It assures that the determinism feature of the behavior model is not violated. The transition consistency check guarantees the correct structure of the behavior model. As a determined automaton, given the <i-th input> under the <(i-1)-th state>, the next state <i-th state> should be determined. For example, if the previous state <(i-1)-th state> is "Say stop to stop.", the current input <i-th input> is "stop", the current output <i-th output> is "Goodbye.". Observer judges the current output as a new state by the prompt in Fig. 3. Consequently, a new transition δ("Say stop to stop.", "stop") = "Goodbye." will be added to the behavior model. However, if the behavior model already has a transition δ("Say stop to stop.", "stop") = "See you.", the determinism of transitions is violated. To avoid breaking the determinism of transitions, the transition checker is used to ensure the current output "Goodbye." is correctly mapped to the state "See you." rather than a new state "Goodbye.". Therefore, the transition checker will firstly search if there is already a transition function δ(<(i-1)-th state>, <i-th input>) = <x-th state> in the behavior model. If found and the <x-th state> is not equal to the <i-th state>, the guidance will return the Should Merge Suggestion to suggest that the <i-th output> should be merged to the <x-th state> rather than the <i-th state>.

3.3 Generator Agent

To generate potential next-round inputs for testing, previous approaches have primarily relied on rule-based methods [21, 28]. However, these methods are limited in their ability to cover all possible output types and face scalability challenges, particularly when handling wh outputs. To address these limitations, we propose implementing the Generator using LLMs. However, a key challenge with LLM-based generation is that it may produce overly long sentences, which can introduce ambiguity and confusion for VPA applications. To mitigate this issue, we refine the LLM's outputs through a two-step process (see Fig. 2). First, we employ pattern-guided prompting to help the Generator understand the context of VPA applications. Second, we utilize the Inputs Checker to validate the generated inputs based on the feedback from VPA applications, ensuring accuracy and relevance.

3.3.1 Pattern-Guided Input Event Generation. For the <i-th output>, Elevate uses the prompt displayed in Fig. 5 to query LLMs. Similar to Section 3.2.1, during the initial query ($i = 1$), a long prompt is used, which includes some few-shot examples, the task instruction and the output format. After that ($i >= 2$), a short prompt which contains the <i-th output> is used for query.

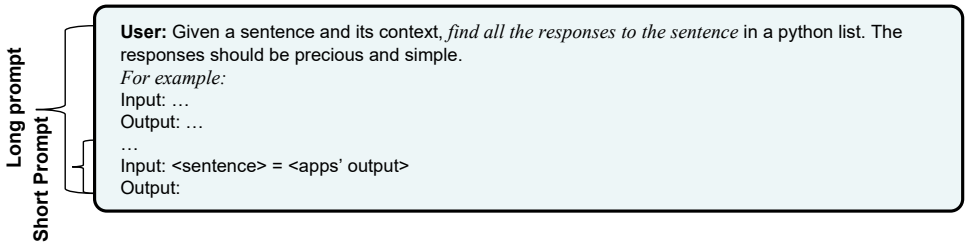


Fig. 5. The prompt used in Generator to generate the next-round possible inputs of VPA apps.

Specifically, the outputs of VPA apps can be categorized into five distinct types: yes-no outputs, selection outputs, instruction outputs, wh outputs, and mixed outputs [21]. Outputs within the same category exhibit similar formats.

- Yes-no outputs typically pose binary questions, such as "Are you ready?" or "Is that correct?", and expect responses like "yes" or "no".
- Selection outputs present multiple choices, often connected by conjunctions such as "and" or "or". A representative example is "Some possible names are: Hunter, Cuddles, Boomer, or Biscuit.". It expects inputs "Hunter", "Cuddles", "Boomer" and "Biscuit".
- Instruction outputs contain directive phrases with imperative verbs like "say". For instance, "Say 'more' to hear additional choices." expects 'more' as the input.
- Wh outputs are more open-ended, such as "What do you want to do with your pet today?". While some wh outputs allow free-form responses, others impose constraints on valid inputs. For example, the wh output "Which zodiac sign do you want to set as default?" expects a response limited to valid zodiac signs.
- Mixed outputs combine elements from multiple output types, incorporating characteristics of yes-no, selection, instruction, and wh outputs.

To enhance the accuracy of model-generated results, we include seven few-shot examples in the long prompt, leveraging these output categories to guide the model effectively. The seven examples encompass all five output categories. Each of the first four categories is represented by a single example. For mixed outputs, we identify and summarize the three most common patterns: instruction + selection output, wh + selection output, and yes-no + selection output, each of which is illustrated with a corresponding example. The example is structured in the form of "Input: <VPA apps' output>" and "Output: <inputs>" pair.

Finally, the <(i+1)-th inputs> are obtained by parsing the textual outputs of the Generator into a list. If the generated input fails to pass the Inputs Checker (refer to Section 3.3.2), a feedback prompt generated by the Inputs Checker will be provided to the LLM to refine and correct the output through re-prompting. The rules to generate inputs for different patterns of outputs are included in the feedback prompts. For yes-no outputs, the rule requires that the <(i+1)-th inputs> only contain "yes" and "no". For selection outputs, the rule determines that the <(i+1)-th inputs> are extracted from conjunctions. For instruction outputs, the rule asks for phrases after verbs like "say" or "tell" to form the input set. For wh outputs with constraints and in the form of "What <noun>..." , the rule requests that inputs are related to <noun>. For mixed outputs, the rule is combined of the above four rules based on the pattern of outputs. For other types of outputs, the rule asks for context-related inputs, so the inputs should be related to the <i-th output>.

3.3.2 Apps' feedback-guided checker. We design the Inputs Checker to ensure that the generated input set is both non-empty and contains only valid inputs. It operates in two steps: non-emptiness checker and the validity checker where non-emptiness checker verifies that the generated input set is not empty and the validity checker

Feedback:
Empty Error: The output *should be a non-empty python list* of the possible responses to the sentence <app's output>.
Invalid Suggestion: <input> is *not a valid response* for the sentence <app's output>. The output should be a python list of <rules>.

Fig. 6. The used feedback prompt in Generator.

ensures that all generated inputs conform to the expected format and constraints. Feedback prompts in Fig. 6 are returned to ask for another input set if the current input set fails to pass the Inputs Checker.

- *Non-emptiness Checker*. It ensures that the generated input set is not empty. If the <(i+1)-th inputs> is empty, the Empty Error will be returned to Generator.
- *Validity Checker*. It ensures that the inputs generated by the Generator do not contain any invalid entries. An input is deemed invalid if it fails to discover new states or if the state it leads to is not meaningful for further exploration. The validity of an input is determined based on the next state as follows: ① If the next state is identical to the current state or corresponds to a final state, the input is considered incapable of triggering new states. ② If the current output indicates confusion (e.g., responses such as "Sorry" or "I don't understand"), the generated input is deemed incomprehensible and, therefore, unhelpful for further exploration. If any input in <(i+1)-th inputs> is found to be invalid, the Inputs Checker returns an Invalid Suggestion feedback to prompt the model for a revised and valid set of <(i+1)-th inputs>.
  Since the validity of an input can only be determined after its resulting state is acquired, the Validity Checker does not execute immediately after Generator produces input events. Instead, it executes when one of the generated input events is selected for execution and the corresponding next state is obtained. If any input event is found to be invalid, Generator is queried to generate new input events for the current state, and the behavior model is updated accordingly.

3.4 Behavior Model Construction

In Section 3.2, Observer extracts the <i-th state>, which locates the testing progress. Meanwhile, Generator produces the candidate input set for further exploration. Although the LLM has a strong capability in selecting semantically related inputs during exploration, without global guidance information, such as history execution and new states discovery capability, the LLM agent cannot learn from previous trails to update its testing strategy. As a result, the behavior model is further constructed to provide a global overview of the entire testing process.

In particular, the behavior model is represented by a five-tuple deterministic finite automaton. Building the behavior model is to update the five tuples. The current state and the current input set are obtained by Observer and Generator, respectively, and added to the behavior model. The transition set is updated by adding a new transition function that links the previous state, the previous input and the current state. The final state set is modified when VPA apps quit. In detail, after Observer extracts the <i-th state> and Generator generates the <(i+1)-th inputs>, the behavior model is constructed as follows:

- The <i-th input>'s invocation times in the <model.Σ(<(i-1)-th state>)> is incremented by one.
- If the <i-th output> is not spoken by the tested VPA apps, the <(i-1)-th state> is added to the final state set <model.F>, and the following update is skipped.
- The <i-th state> is added to the state set <model.Q>.
- The input that is in the <(i+1)-th inputs> but not in the <model.Σ(<i-th state>)> is added to the <model.Σ(<i-th state>)>, and its initial invocation times is set as zero.
- The transition δ(<(i-1)-th state>, <i-th input>) = <i-th state> is added to the transition set <model.δ>.

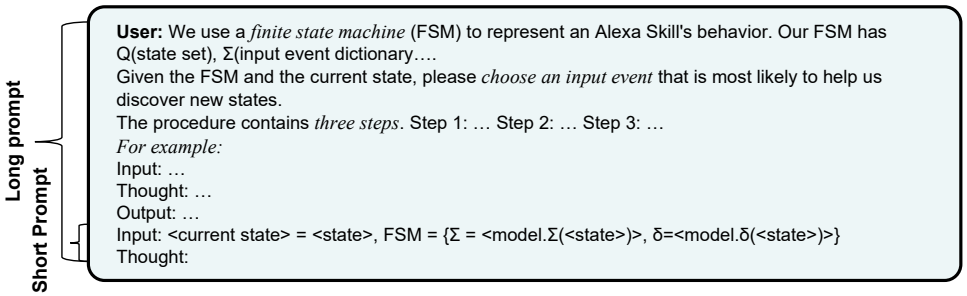


Fig. 7. The prompt used in Planner to generate an input from the input set.

3.5 Planner Agent

The testing strategy decides the effectiveness of testing. In the VUI testing scenario, semantic relevance of candidate inputs is especially significant, as the VPA apps' inputs and outputs are natural language. For example, when firstly visiting the "Is that correct?" state, "yes" is a better choice than "no", as "yes" is more likely to help cover new states and functionalities. Semantic information acts as the metric to decide which input to select when there is no related history testing information. While previous approaches [28] only adopt common metrics like functionality coverage, state space coverage to guide the exploration, Planner relies on LLMs to consider other metrics like text semantics when conducting state space exploration. It also incorporates the information from the behavior model to facilitate the input choice. We embed the behavior model information in the prompt so that Planner can understand the VPA apps' context and the global testing progress (see Section 3.5.1). A feedback mechanism called the Selector is established on Planner (see Section 3.5.2) for further guidance.

3.5.1 Behavior Model Guided State Space Exploration. Given the <i-th state> and its corresponding input set in the behavior model[1], we design prompts to query LLMs to select an effective input in Fig. 7. Behavior model is embedded into the prompt to offer information such as input set, invocation times and historical transitions. As introduced in Section 3.4, As we introduced in Section 3.4, the behavior model is updated before Planner is called to make decisions. Specifically, if an input $input_1$ is executed under a state $state_1$ in the previous interaction round, both the invocation times of $input_1$ and the corresponding transition $\delta(state_1, input_1)$ are updated in the behavior model. With this up-to-date information, Planner is equipped to make effective decisions at each step. However, including the entire behavior model in the prompt is unrealistic, as it occupies many tokens and disturbs Planner from identifying core information. Only the current state related information in the behavior model, including the current state <i-th state>, the input set <model.Σ(<i-th state>)> and transitions <model.δ(<i-th state>)>, is extracted to the prompt. To further improve Planner's capability, we employ a strategy combining in-context learning and chain-of-thought. In particular, as shown in Fig. 7, we prompt Planner to think step-by-step and show its thinking process. In step 1, Planner is asked to remove the invalid input events. In step 2, Planner finds a never-invoked input event that is mostly related to the context. In step 3, Planner finally chooses one input event from the never-invoked context-related input events in step 2 or the valid input events.

Similar to the Observer in Section 3.2 and Generator in Section 3.3, for the first query <i-th state> where $i = 1$, Planner uses a long prompt. It outlines the composition of the behavior model, task instructions, step-by-step guidelines and three few-shot examples in the form of "Input: <state>, <model.δ(<state>)>, <model.Σ(<state>)>", "Thought: step1: xxx, step2: xxx, step3: xxx" and "Output: <input>" triplets. When selecting inputs for the <i-th state> where $i \geq 2$, Elevate uses a short prompt, which only contains the <i-th state> and its related information

[1]Noted that it contains the <(i+1)-th input> generated by Generator.

Feedback:
No Input Error: <input> is *not in* the given input event set <inputs>. Please choose another input event from the input event set.
Better Input Suggestion: Choosing the input <input$_x$> *might be better than* the input <input>.Please choose another input event from the input event set.

Fig. 8. The used feedback prompt in Planner.

in the behavior model. Planner is expected to output its thinking process along with the selected <(i+1)-th input>. The selected input <(i+1)-th input> is acquired by identifying the text after the label "Output:". Then the <(i+1)-th input> is sent to the selector (refer to Section 3.5.2) for format and testing strategy checking. A feedback prompt with specific errors or recommendations is incorporated into the subsequent query. We implement the selector to double-check Planner's testing strategy.

3.5.2 Rule-based Selector. Although LLMs provide semantic information while selecting inputs, they may not always select the best one in terms of invocation times or new states discovery due to their unexplainability in rare cases. To address this limitation, we propose a selector to confirm that Planner's selected input is available and its testing strategy is good considering invocation times, new states discovery abilities based on our pre-defined rules. It is composed of three checkers: availability checker, invocation frequency checker and validity checker. If the LLM's selection strategy is considered worse, the feedback with a recommended better choice in Fig. 8 will be provided to the LLM for strategy improvement. The iteration continues until all checkers are passed or the maximum iteration times is reached.

- *Availability Checker*. It guarantees that the selected input (the <(i+1)-th input>) is available in the input set. Unavailable inputs may be irrelevant, so they are excluded. The availability check searches the <model.Σ(<i-th state>)> to find the <(i+1)-th input>. If the <(i+1)-th input> is not in the given set, it returns the No Input Error.
- *Invocation Frequency Checker*. It prioritizes selecting events that have not yet been explored. For example, if the <(i+1)-th input> is "more" and it is selected once, and there is another input called "Shadow" that has never been invoked before, "Shadow" is a better choice. In other words, if there is one <x-th input> that is both invoked less frequently and valid, the Better Input Suggestion recommends <x-th input> as a better input.
- *Validity Checker*. It ensures that the valid input always has higher priority than an invalid one. The validity check ensures that this rule is always obeyed. If such a valid <x-th input> is found and the <(i+1)-th input> is invalid, the validity check fails and the Better Input Suggestion recommends the <x-th input>.

The selector ensures that the <(i+1)-th input> can pass the above checkers and it will be sent to the VPA app for the next interaction round.

3.6 An Illustrative Example

To gain a better understanding of the workflow, we present an illustrative example in Fig.9, including extraction of states, generation of the input set, construction of behavior model and state exploration.

When the <i-th output> "Some other possible names are: Shadow, Spike, Rascal, or Snowball. Either choose one of those names, or say, 'more', to hear more choices." is acquired, Observer extracts a state for it from the behavior model's state list. Observer finds that there is an existing state with semantically similar content as the <i-th output>, which is exactly the same as the previous state <(i-1)-th state>. So it outputs "Either choose one of those names, or say, 'more', to hear more choices.". Meanwhile, Generator takes the <i-th output> as the input

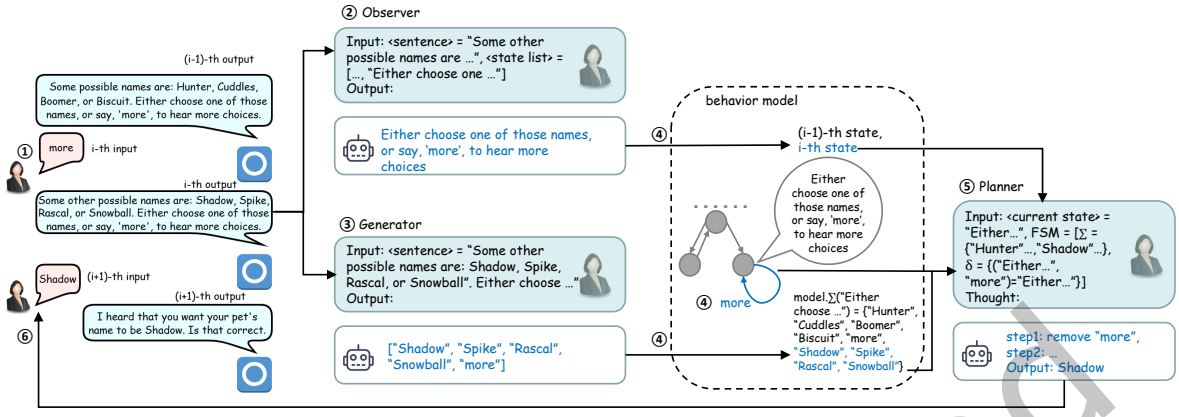


Fig. 9. The example to illustrate the workflow of Elevate. Only the Short Prompt and pass-check results are displayed. The feedback mechanism is omitted.

and generates the <(i+1)-th inputs>. Generator identifies the first sentence as a selection output and the second sentence as an instruction output. Based on the feature of these two types of outputs, it generates ["Shadow", "Spike", "Rascal", "Snowball" and "more"] as the <(i+1)-th inputs>.

The state extracted by Observer and the input set generated by Generator are used to construct the behavior model. The current state <i-th state> is located as the previous state <(i-1)-th state>, so the state set is not changed. The <(i+1)-th inputs> {"Shadow", "Spike", "Rascal", "Snowball", "more"} is added to the input set of the <i-th state>. A transition from the <(i-1)-th state> to the <i-th state> is added with the input event 'more'. The current state is the same as the previous state, so the <(i-1)-th state> related information in the behavior model can be used to help Planner select the <(i+1)-th input> at the <i-th state>. This information includes the input event set of the <(i-1)-th state > and the transition of the <(i-1)-th state>. The transition of the <(i-1)-th state> indicates that "more" is not a good choice as it cannot trigger effective transition. Planner takes this information as input. Planner understands that "more" is an invalid input, so its final choice is one of the recommended names "Shadow". "Shadow" is send to VPA apps for the next interaction round. If Observer fails to consider semantic information, and it generates a new state for the <i-th output>, Planner will not know that 'more' is a bad choice, and it is likely that 'more' will be selected again and the same functionality will be retested.

4 EVALUATION

In this section, we aim to answer the following research questions:
RQ1: How effective is Elevate compared to state-of-the-art techniques on VPA Apps' problem detection capacity?
RQ2: What is the performance of Elevate compared to state-of-the-art techniques in coverage rate?
RQ3: What are the contributions of different components of Elevate in improving its performance?
RQ4: How good is the generated behavior model?
RQ5: What is the performance of Elevate compared to state-of-the-art techniques in the large-scale test?

4.1 Setup

Benchmark. UQ-AAS21 dataset [6] contains basic information on real-world VPA apps and has been widely used in prior studies related to VPA apps [28, 36, 37]. Based on the information provided in the dataset, we could access the latest information pages and versions of the skills. Our evaluation was carried out on these latest

versions. From this dataset, we removed skills that lack ratings up to the time of the test based on the assumption that problems in popular skills have a greater impact on common users. Through preliminary filtering [12], 36,050 skills were removed. Due to the lax official certification process, a large number of low-quality skills remain in the store. Since testing all skills requires significant time and financial resources, we made a tradeoff by focusing on popular and influential ones.

Subsequently, we identify 4,000 popular skills with stable behaviors across various tests, forming a large dataset referred to as the **Stable-Large-Dataset**. This dataset spans all categories available on the Amazon skills website. Due to its large size, running evaluation on it incurs significant computational costs (nearly **one month** for a single round of evaluation). Additionally, many of the apps in this dataset exhibit simple behavior or are built using similar templates, making it meaningless to assess the performance. Thus, we create a **Complex-Small-Dataset**, selecting 50 Alexa skills from the Stable-Large-Dataset. These skills are chosen based on the following criteria: ① They represent a diverse range of categories. ② They do not suffer from network errors.③ They exhibit complex behaviors. We identify skills with potential network errors through a simple screening procedure. Specifically, each skill in the Stable-Large-Dataset is invoked using the official command "Alexa, open + <invocation name>" across 10 different accounts. If a skill fails to launch successfully on any of these accounts, it is removed. Skills exhibiting complex behavior are selected based on the size of their large semantic state space. After running Elevate on the Stable-Large-Dataset and excluding skills with network issues, we selected the 50 skills with the largest semantic state spaces to form the Complex-Small-Dataset. This selection ensures that the skills are complex enough to accurately reflect the evaluation methods, while also eliminating the influence of unstable behavior on the evaluation's effectiveness and fairness.

Baselines. We compare Elevate with two typical baselines as follows:

- **Vitas**. Vitas [28] is the state-of-the-art model-based testing framework for VPA apps which extracts states from outputs and generates input events through simple NLP rules. It explores the state space by managing weights and input, with the highest weight being selected in each round. When Vitas receives an input from a VPA app, it firstly generates a state for that output at the sentence level. Specifically, it compares the real-time output with previously discovered outputs, and creates a new state if the output textually differ from the discovered outputs. Next, Vitas generates candidate inputs for this output based on predefined rules. For example, if a sentence starts with an auxiliary verb, it is treated as a Yes-No question, and the fixed candidate inputs "yes" and "no" are generated. Each candidate input is then assigned with a weight based on the number of times it has been invoked and the input type. The input with the highest weight is sent to the VPA app for the next interaction round. Although Vitas uses the model to guide testing, the model is constructed at the sentence level. It was elevated to surpass the classic chatbot-style tester Skillexplorer [21] in terms of coverage and efficiency.
- **LLM-based chatbot**. Vitas has confirmed that the previous rule-based chatbot tester [21] is inferior to it. Essentially, rule-based methods rely on expert-defined rules to guide interactions, which can be rigid and difficult to scale. LLM, on the other hand, offers a more flexible and scalable alternative. We thus design the LLM-based chatbot as the baseline for evaluation. The LLM chatbot is instructed to act as a tester for VPA apps in the system prompt. When a real-time output is received, it is sent to the LLM as the prompt. Then, the LLM's response is directly used as the input for the next interaction with the VPA app. Except from the real-time output, no high-level information is provided to guide the LLM. This iterative process continues until the VPA app exits or terminates the session.

Test Settings: We conducted repeated tests to ensure the statistical significance of our evaluation results. Each evaluation method ran three trials to test each skill in the Complex-Small-Dataset. In each trial, the testers are allowed to test the skills in multiple sessions in case the skills exist unexpectedly, as long as the total testing time is within 10 minutes. Consequently, testing the entire Stable-Large-Dataset sequentially using a single evaluation

method would require 40,000 minutes (approximately one month). Due to time and financial costs, testers only run one trial on the Stable-Large-Dataset.

Metrics. For RQ1, we evaluate the approaches based on the number of problems detected, comparing the effectiveness of different methods in identifying issues. For RQ2, RQ3 and RQ5, we adopt two distinct metrics for evaluation: Sentence Coverage Rate – The sentence space, as defined by Vitas [28], is used as one of the metrics. Given an output, its sentence state is extracted by first comparing its content with the contents of existing sentence states. If no existing sentence state has the same content as the output, a new sentence state is created using the output itself as its content. Therefore, the sentence space size is calculated by counting the unique sentences in the skill outputs. Semantic Coverage Rate – To address the limitations of the sentence space, which fails to account for semantic similarity between states, we introduce the semantic state space. In this space, semantically similar or equivalent outputs are consolidated into a single state. To ensure fairness, the semantic states of all evaluation methods are extracted using GPT-4. Specifically, GPT-4 is prompted with an output $output_x$ and a list of existing states, and it determines the semantic state corresponding to $output_x$. It either selects a state $state_y$ from the state list to indicate that $output_x$ should be mapped to $state_y$, or returns $output_x$ itself, indicating that a new state $state_x$ should be created with $output_x$ as its content. Furthermore, the total state space of sentence or semantic is the union of the generated states by Elevate and the baselines for sentence or semantic respectively. The coverage rate is then calculated by dividing the size of the state space (either sentence or semantic) by the total state space size.

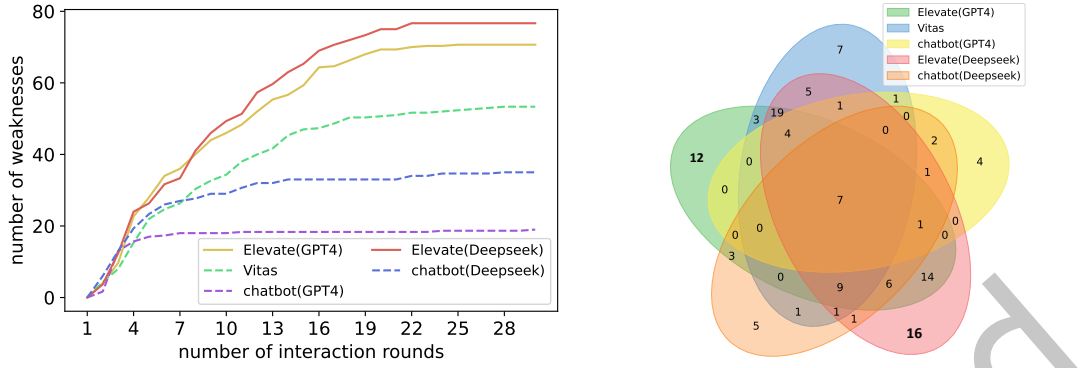
Implementation. In our experiments, we implement Elevate using two different LLMs: GPT-4-Turbo-2024-04-09 [7] and DeepSeek-R1 [9]. These implementations are named as Elevate (GPT4) and Elevate (DeepSeek), respectively. In both versions, for the Observer, the temperature for querying the LLM is set to 0, ensuring that state extraction is as deterministic as possible. In contrast, the Generator and Planner use a temperature of 0.1, which introduces a degree of flexibility to allow for more diverse input sets and exploration strategies. The LLM-based chatbot is also implemented using these two LLMs, referred to as chatbot(GPT4) and chatbot(DeepSeek). For consistency, the temperature for querying the chatbot is set to 0, ensuring that its performance remains stable in evaluation. The Amazon simulator [2] is employed as the testing platform, which supports text-based input and output. This design choice avoids potential inaccuracies introduced by speech-to-text processes. The experiments were conducted on Ubuntu 18.04.4 machines equipped with an AMD EPYC 7702P 64-Core Processor CPU@1.996GHz and 4GB RAM, providing a consistent environment for evaluation.

4.2 Evaluation Results

4.2.1 RQ1: Problems Detection.

The interaction round, defined as a pair of an input and its corresponding output, is used as the evaluation metric instead of response time. This choice is made for the following reasons: ① Network variability – The simulator's response time to the same input can fluctuate due to network conditions, leading to inconsistencies in timing-based evaluations. ② LLM inference time differences – Different LLMs require varying amounts of time to generate responses, making response time an unreliable metric for comparison.

Detected problems with interaction rounds. Fig. 10a presents the average number of problems of the three trials detected by Elevate and baseline methods on the Complex-Small-Dataset. Overall, we observe that as the number of interaction rounds increases, the number of detected problems also rises. Among all methods, Elevate (DeepSeek) identifies the highest number of issues, detecting an average of 76.67 problems across 50 apps, followed closely by Elevate (GPT4), which finds an average of 70.67 problems. In contrast, Vitas and chatbot(LLM) perform significantly worse, detecting no more than 55 problems each. These results highlight the effectiveness of Elevate, demonstrating that it significantly outperforms the previous state-of-the-art baselines in detecting weaknesses in VPA applications.



(a) The average number of problems detected with the increasing number of rounds.

(b) Different problems detected shown in venn diagram.

Fig. 10. Problems detected by Elevate and baselines.

Comparing Elevate with the state-of-the-art Vitas, we observe that Elevate maintains a continuous problem detection capability, whereas both Vitas and chatbot(LLM) experience a significant slowdown in discovering new issues after approximately 10 interaction rounds. Notably, between the 10th and 20th interaction rounds, Elevate (GPT4) and Elevate (DeepSeek) identified 23.33 and 25.67 new problems, respectively. In contrast, Vitas detected only 16.33 new problems, while chatbot(GPT4) and chatbot(DeepSeek) found just 0.33 and 4.00 problems, respectively. These results demonstrate that Elevate possesses a sustained ability to uncover issues by continuously exploring new states and functionalities. This capability enables it to detect deep and complex problems that other methods fail to reveal. Furthermore, the findings highlight that unguided LLMs, such as chatbots, perform poorly in VUI testing, as they lack a structured exploration strategy and traditional approaches like Vitas struggle to comprehend the semantic meaning of the context, leading to suboptimal performance in problem detection.

Unique problems found by Elevate and baselines. Fig. 10b presents a Venn diagram illustrating the number of weaknesses detected by Elevate and the baseline approaches. The problems for comparison are the union of problems detected by each tester in three trials. From the results, we observe that Elevate (GPT4) and Elevate (DeepSeek) uncover 12 and 16 unique problems, respectively, which are missed by both Vitas and chatbot-style LLM testers. Additionally, Elevate successfully identifies 83.7% of all detected issues, demonstrating its effectiveness in VPA testing. However, some problems remain undetected by Elevate. This phenomenon arises due to differences in testing strategies between Elevate and the baselines. Elevate prioritizes thorough exploration of an app's behavior by generating and selecting contextually relevant inputs. Vitas and chatbot(LLM) focus on detecting problems by using unusual or templated inputs. To illustrate this distinction, we conduct a case study (Fig. 11) showing how different testing strategies impact problem detection. For example, when the Amateur Impressions skill prompts the user to select a portrayed impression (e.g., Arnold, Christopher Walken, Gollum), we observe that Elevate (GPT4) selects a contextually relevant input ("Arnold") and successfully discovers a new state and functionality. Vitas, which uses a template-based approach, extracts part of the output as input, leading to a meaningless response. Chatbot(LLM) lacks contextual awareness, often generating irrelevant outputs that do not align with the semantic meaning of the response. However, as these skills usually have difficulty in understanding strange inputs, Vitas and Chatbot(LLM) detect the problems (i.e., unexpected exit). These cases are rare on the testset.

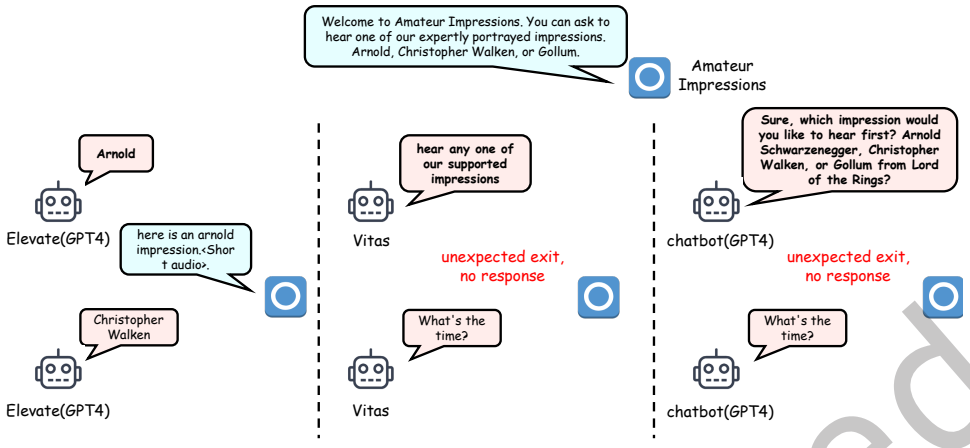


Fig. 11. The case study that illustrates the different testing strategy adopted by Elevate, Vitas and chatbot(LLM).

Distribution of problems across different types found by Elevate and baselines. Table 1 presents the number of runtime problems detected by Elevate and baseline methods across various problem types. Only runtime issues are included in the analysis, as skills that fail to launch correctly were excluded from the Complex-Small-Dataset. The values in the table are the confidence intervals when the confidence level is set as 95%. Among all the testers, Elevate (Deepseek) and Elevate (GPT4) detect the highest number of problems across all categories. Specifically, Elevate (Deepseek) detects more than 75 issues, with the majority being Unexpected Exit problems. Elevate (GPT4) demonstrates particular strength in uncovering Privacy Violations and Unstoppable App issues, detecting 8 and 7 instances, respectively. Vitas and chatbot-style testers detect significantly fewer problems in all categories — approximately 33% fewer than those found by Elevate.

Although different testers are used, Unexpected Exit problem accounts for the largest proportion. However, there are skills that consistently show no issues across multiple tests by various testers. typically provide clear guidance and status notifications. In detail, they recommend inputs aligned with their core functionality, which helps prevent users from using irrelevant commands. Before termination, they offer clear status updates such as "Goodbye" or "See you again", and terminate immediately after these farewell messages. In addition, these robust skills are equipped with error handlers by saying things like "Sorry, I don't understand your commend. Please say <recommended inputs> to continue.", thus avoiding unexpected exits caused by irrelevant or malformed inputs. In contract, skills that are more susceptible to issues often lack such error-handling strategies and fail to gracefully manage unexpected inputs. Meanwhile, a part of skills terminate sessions immediately after responding to a user input without providing any status update. Given that VPA apps operate on a voice-only interface, such invisible terminations can lead to user confusion. Therefore, providing user-friendly outputs and explicit status updates is crucial for improving the overall user experience.

Qualitative evaluation of problems found by Elevate and baselines. To further analyze the features of the problems identified by Elevate and the baselines, we define two metrics: 1) the shortest paths required to trigger a problem, and 2) the similarity between tester-generated inputs and typical user inputs along the problem-triggering path. To compute the first metric, we construct a graph from the communication logs between each evaluation method and each skill. For Elevate, this graph is identical to the behavior model. For other evaluation methods, each node in the graph represents a semantic state extracted by GPT-4, and each edge

Table 1. The number of problems detected by Elevate and baselines across different types. The value in the table is set as the confidence interval when the confidence is 95%.

Approach	Total	Unexpected Exit	Privacy Violation	Unstoppable App
Elevate (GPT4)	[68.7, 72.6]	[53.7, 57.6]	[8.0, 8.0]	[7.0, 7.0]
Elevate (Deepseek)	[75.3, 78.1]	[61.3, 64.1]	[6.1, 7.2]	[6.8, 7.9]
Vitas	[51.9, 54.7]	[43.9, 46.7]	[4.0, 4.0]	[4.0, 4.0]
chatbot(GPT4)	[17.4, 20.6]	[17.4, 20.6]	[0.0, 0.0]	[0.0, 0.0]
chatbot(Deepseek)	[34.1, 35.9]	[26.1, 27.9]	[5.0, 5.0]	[3.0, 3.0]

Table 2. The number of problems detected by Elevate and baselines with short, middle or long inputs paths.

tester	short([0, 3))	middle([3, 6])	long((6,+inf))
Elevate (GPT4)	39	**34**	**5**
Elevate (Deepseek)	**46**	**34**	**5**
Vitas	30	26	2
chatbot(GPT4)	12	8	1
chatbot(Deepseek)	21	15	1
total	61	56	6

corresponds to an input that links two states. The shortest path to a problem is defined as the shortest path from the first node to the node representing the problematic state.

To calculate the second metric, we invited two annotators from our laboratory who are familiar with VPA apps to evaluate the inputs along the shortest problem-triggering path. Each input is scored with 0, 0.5 or 1 based on its similarity to user-generated inputs and the app's response to it. The final score of a problem is defined as the average of the input score along its shortest path. To assist annotation, we provide background examples illustrating inputs that VPA apps can fully understand, partially understand or fail to understand, based on the apps' feedback. Inputs that are contextually relevant, similar to user-generated inputs, and well comprehensive by the VPA app are scored with 1. Inputs meeting one or two of these requirements are scored with 0.5, while inputs failing to meet any are scored with 0. For each problem, both annotators independently assign scores. We then compute the relative difference rate between the two scores by dividing their absolute difference by their average. If the relative difference rate exceeds 0.15, the annotators are asked to re-evaluate until consensus is reached. The final similarity score of each problem is the average of the two agreed-upon scores.

Tab.2 presents the number of problems categorized by the length of the shortest path from the initial node: short([0, 3)), middle([3, 6]), and long((6, +inf)). Both Elevate (GPT4) and Elevate (Deepseek) detect the highest number of problems across all categories, demonstrating superior overall effectiveness. In particular, they show strong performance in uncovering deeply hidden issues: among the six problems requiring at least six inputs to be triggered, Elevate (GPT4) and Elevate (Deepseek) detect 83.3% of them. In constrast, Vitas and the LLM-based chatbot methods only detect 33.3% and 16.7%, respectively. These results demonstrate that Elevate is notably more capable of discovering hard-to-reach problems in VPA apps.

Tab.3 shows the number of problems categorized by the similarity of their triggering inputs to those commonly generated by users: low([0, 0.7]), middle((0.7, 0.85]) and high((0.85, 1]). Elevate (GPT4) and Elevate (Deepseek) detect the most problems triggered by inputs that are highly similar to typical user inputs. Specifically, among

Table 3. The number of problems detected by Elevate and baselines whose triggering inputs have low, middle or high similarity with user-generated inputs.

tester	low([0, 0.7])	middle((0.7, 0.85])	high((0.85, 1])
Elevate (GPT4)	8	**15**	55
Elevate (Deepseek)	12	14	**59**
Vitas	**13**	8	37
chatbot(GPT4)	7	3	11
chatbot(Deepseek)	8	7	22
total	28	21	74

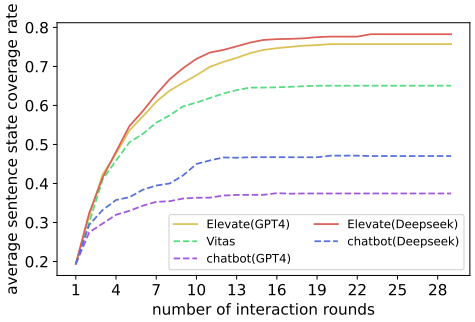
Table 4. The average significance and impact score of ten randomly selected weaknesses in a user study.

problem id	skill's name	problem type	average significance score	average impact score
1	Zyrtec	Unexpected Exit	4.43	4.40
2	Book One	Unexpected Exit	4.48	4.53
3	Action Movies	Unexpected Exit	4.40	4.48
4	Amateur Impressions	Unexpected Exit	4.43	4.48
5	Zyrtec	Privacy Violation	4.65	4.68
6	Spelling Bee	Privacy Violation	4.60	4.75
7	ZauberBot	Unstoppable App	4.55	4.58
8	Sous Chef	Unstoppable App	4.55	4.68
9	MY HOME	Unexpected App Started	4.53	4.55
10	Central Bucks	Unavailable App	4.10	4.13
Average	-	-	4.47	4.53

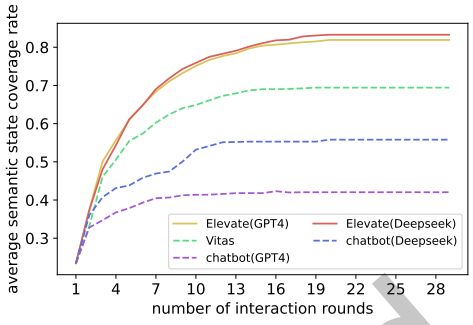
the problems with the highest similarity scores, Elevate (GPT4) and Elevate (Deepseek) detect 74.3% and 79.7%, respectively. In contrast, Vitas detects only 50%, while chatbot-style methods perform the worst, identified fewer than 30% of such problems. These findings indicate that Elevate's behavior more closely resembles that of real users, enabling it to uncover issues that are more likely to be encountered in real-world usage. The qualitative evaluation highlights that Elevate and the baselines adopt distinct testing strategies: Elevate tends to use user-familiar inputs to discover deeply embedded problems, while Vitas and chatbot-style methods often generate less irrelevant inputs, leading them to quickly uncover more superficial weaknesses.

We also conducted a user study to evaluate the weaknesses detected by Elevate (GPT4) and Elevate (Deepseek). A total of 40 users participated in the study, with ages ranging from 18 to 78 and diverse occupational and educational background. All participants had prior experience using mobile applications, and 80% have used voice-based applications. Each participant was asked to assess ten randomly selected weaknesses identified by Elevate. These ten weaknesses spanned all five problem categories and came from nine different skills. For each weakness, participants rated significance and impact on a scale from 1 (least) to 5 (most). Participants scored significance based on the extent to which the issue deserves developers' attention. For the impact, participants scored it according to the severity of the weakness.

Table 4 presents the average significance and impact score of the weaknesses evaluated in the user study. The ten selected weaknesses include four Unexpected Exit issues, two Privacy Violation issues, two Unstoppable App



(a) The average sentence state coverage rate with the increasing number of interaction rounds.



(b) The average semantic state coverage rate with the increasing number of interaction rounds.

Fig. 12. Average sentence state and semantic state coverage rate calculated by Elevate and baselines.

issues, one Unexpected App Started issue and one Unstartable App issue. From the users' perspective, Privacy Violation weaknesses are perceived as the most critical, with average significance and impact scores of 4.63 and 4.72, respectively. Unstoppable App problems also raise serious concern, receiving average scores of 4.55 for significance and 4.63 for impact. Users also rated the Unexpected App Started issue highly, with scores of 4.53 (significance) and 4.55 (impact), indicating their concern. Unexpected Exit issues follow closely, with both average significance and impact scores exceeding 4.40. Although the Unstartable App weakness received the lowest ratings among them, its average significance and impact scores still exceed 4.10, suggesting that users generally consider it non-negligible. In summary, the results indicate that users regard all detected weaknesses as both significant and impactful, with Privacy Violation and Unstoppable App issues standing out as particularly severe.

Answers to RQ1: Elevate (GPT4) and Elevate (DeepSeek) detect at least 17.33 and 35.67 more problems than Vitas and chatbot(LLM), respectively, across 50 apps. Additionally, Elevate (GPT4) identifies 12 unique problems, while Elevate (DeepSeek) uncovers 16 unique problems that were missed by baseline methods. This improvement stems from Elevate 's superior contextual semantic comprehension during interactions. After conducting a qualitative evaluation on detected weaknesses, we observe that Elevate (GPT4) and Elevate (Deepseek) are good at detecting hidden problems. The user study demonstrates that weaknesses found by Elevate are significance and impactful from the users' perspective.

4.2.2 RQ2: Coverage Analysis.

Sentence state coverage. Fig. 12a presents the average coverage rate over interaction rounds achieved by Elevate and the baseline methods. The results show that Elevate (GPT4) and Elevate (Deepseek) achieve the highest coverage, reaching an average of 75.7% and 78.3%, respectively. Vitas follows with a an average final coverage of 65.1%, while chatbot-style LLM testers rank the lowest, with coverage below 50%. However, it is important to note that the total sentence state space is redundant, as semantically similar or equivalent states are not merged. Consequently, the absolute values of sentence state coverage appear relatively low. In this metric, a sentence state is considered covered only if the exact same sentence is outputted by the skill. However, VPA skill outputs are not fixed—the same state can have multiple linguistic variations, making this metric less reliable for assessing true coverage.

Semantic state coverage.

Fig. 12b presents the average semantic state coverage achieved by different testers. Notably, Elevate (DeepSeek) achieves the highest average coverage at 83.3%, surpassing all other methods. Elevate (GPT4) ranks second with an average coverage of 81.9%, demonstrating a significant advantage in comprehensively exploring the VPA state space. Elevate outperform Vitas, achieving approximately 12% higher average coverage than the third-best tester, Vitas. In contrast, chatbot-style LLM testers perform the worst, with chatbot(DeepSeek) and chatbot(GPT4) achieving only an average of 55.8% and 42.0% semantic state coverage, respectively.

We can conclude that Elevate outperforms all other testers with a clear advantage in VPA testing. Compared to Vitas, Elevate covers 12% more of the state space than the state-of-the-art non-LLM-based tester. This performance improvement is primarily attributed to Elevate 's multi-agent design, which leverages LLM agents and a behavior model to prioritize logical and comprehensible inputs. By selecting inputs that effectively interact with skills, Elevate enables a more thorough exploration of skill behavior. In contrast, Vitas employs a weight-guided strategy to generate inputs by combining phrases from skill documentation and outputs. However, this approach often leads to unusual or irrelevant inputs, limiting its ability to explore meaningful state transitions. Compared to chatbot-style LLM testers, Elevate covers at least 26% more of the state space. This significant gap arises because chatbot(LLM) testers simply use VPA app outputs as the next-round input without a well-designed exploration strategy. While this straightforward approach enables basic interactions, it lacks the dedicated design that allows Elevate to explore deeper and more complex state transitions.

Furthermore, the average token usage per skill in Elevate is 9,078 tokens for inputs and 1,282 tokens for outputs. This results in an average cost of approximately 0.13 dollars per skill when using GPT-4. Given the significant improvements by Elevate, this cost is reasonable and acceptable.

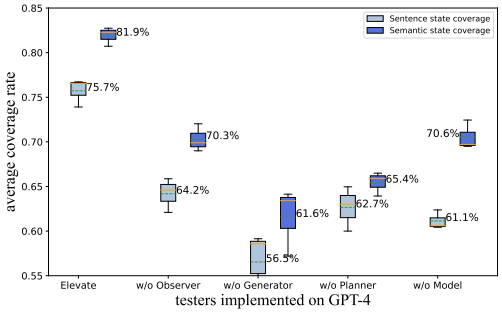
> **Answers to RQ2**: As the number of interaction rounds increases, Elevate (GPT4) and Elevate (DeepSeek) exhibit a clear advantage in sentence/semantic state coverage compared to Vitas, chatbot(GPT4), and chatbot(DeepSeek). This improvement is primarily driven by Elevate 's multi-agent architecture and behavior model design.

4.2.3 RQ3: Ablation Study.

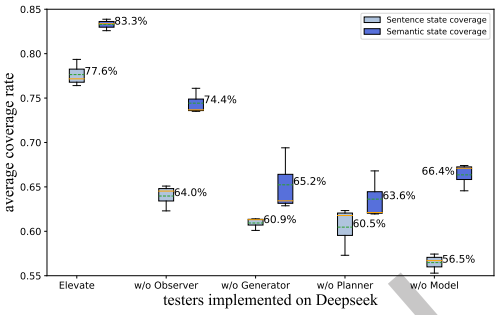
We conduct an ablation study to analyze the impact of each component in Elevate. Specifically, It consists of three LLM-based agents and a behavior model. To assess their contributions, we perform the following modifications: Evaluating the impact of agents – We disable one agent at a time and replace it with a simplified LLM The simplified LLM is implemented by removing two components from the original agent: 1) the entire feedback mechanism, and 2) the few-shot examples in the original prompts. Evaluating the impact of the behavior model – We remove behavior model information in the prompt for evaluation. All other settings remain identical to Elevate to ensure a fair comparison.

Coverage achieved by Elevate, w/o Observer, w/o Generator, w/o Planner and w/o Behavior Model.

Fig. 13 presents the coverage distribution across three trials on the Complex-Small-Dataset. The results indicate that Elevate achieves the highest coverage, with an average sentence state coverage of 75.7%/77.6% and a semantic state coverage of 81.9%/83.3% on GPT-4 and DeepSeek respectively, demonstrating its effectiveness. The ablation study further reveals key differences in the contributions of individual agents: For GPT-4, the Generator has the most significant impact on coverage (i.e., removing it, the sentence and semantic rate drop to 56.5% and 61.6%, respectively, which are the lowest in three agents). For DeepSeek, the Planner contributes the most to state coverage and removing it, the sentence and semantic rate drop to 60.5% and 63.6%, respectively. The variances



(a) The distribution of coverage rate achieved by evaluation methods implemented on GPT-4.



(b) The distribution of coverage rate achieved by evaluation methods implemented on Deepseek.

Fig. 13. The distribution of coverage rates in three trials.

in semantic and sentence coverage achieved by all testers across the three trials are small, as indicated by the relatively flat box plots.

We hypothesize that the primary reason for these variations stems from the intrinsic capabilities of the models. Compared to GPT-4 Turbo, DeepSeek demonstrates superior logical reasoning and planning capabilities. As a result, DeepSeek is able to generate a higher-quality candidate set of inputs. In this condition, the Planner's role in space exploration becomes more critical, playing a key role in enhancing overall performance. In contrast, since the input sets generated by GPT-4 may not be as high-quality as those produced by DeepSeek, the Planner's role is relatively less and Generator plays a more critical role. As a result, the impact of removing Generator is more significant for GPT-4. Compared to the Generator and Planner, the Observer has a relatively smaller direct impact on state coverage. However, its role remains crucial in ensuring the accuracy and efficiency of the testing process. The Observer is specifically designed to extract and merge semantically similar states. Without an effective Observer, it will be a challenge for the Planner to generate accurate and effective inputs for subsequent interactions. Overall, each agent contributes to enhancing the system's performance. While their relative impact may vary depending on the underlying LLM, all three agents work together to improve the performance.

Furthermore, the impact of the behavior model is also analyzed in Fig. 13. The results show that removing the behavior model results in at least a 10% decrease in both sentence state coverage and semantic state coverage compared to Elevate. This finding highlights the importance of incorporating context information in the testing process. The behavior model helps guide input selection and state exploration by leveraging historical context. Without it, the system degrades into a simple multi-agent conversational framework, which is insufficient for effective VPA testing.

Problems detected by Elevate, w/o Observer, w/o Generator, w/o Planner and w/o Behavior Model. Fig.14 presents the distribution of problems detected across three trials on the Complex-Small-Dataset. Both Elevate (GPT4) and Elevate (Deepseek) detect the most problems, with averages of 70.7 and 76.7 respectively. Among the components, removing Generator has the largest negative impact on problem detection: w/o Generator(GPT4) and w/o Deepseek(GPT4) detect no more than 66% of the problems found by Elevate. Removing Planner has the second largest impact, leading to an average decrease of 20 and 25 problems for GPT-4 and Deepseek, respectively. This is expected as Generator and Planner are directly responsible for generating and selecting inputs that explore new behaviors and reveal problems, making them crucial for effective problem detection. Eliminating Observer also reduces problem detection performance, resulting in approximately 20 fewer problems discovered on average. Although the Observer's impact is more indirect, it plays a key role by extracting states used to build the behavior

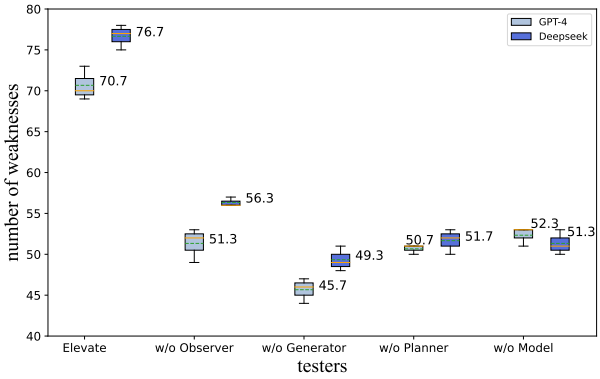


Fig. 14. The distribution of detected weaknesses in three trials.

Table 5. The accuracy rate of states extraction and transitions generated of Elevate (GPT4) and Elevate (Deepseek) compared with users. The value in the table is set as the confidence interval when the confidence is set as 95%.

Composition	Elevate (GPT4)	Elevate (DeepSeek)
States	[99.0%, 99.5%]	[99.0%, 99.5%]
Transitions	[99.3%, 99.7%]	[99.0%, 99.5%]

model; an inaccurate behavior model can mislead exploration and reduce both coverage and detection efficiency. Finally, the behavior model design also contributes to the problem discovery. Without the high-level guidance it provides, the problem number of detected problems drops to 52.3 for GPT-4 and 51.3 for Deepseek.

Answers to RQ3: Each component plays a crucial role in enhancing Elevate's performance. The ablation results demonstrate that both the multi-agent framework and the behavior model contribute significantly to achieve optimal performance in Elevate and removing any of these components leads to a noticeable performance decline.

4.2.4 *RQ4: Behavior Model Quality.*

We conducted an additional evaluation to assess the quality of the behavior model generated by Elevate. The behavior model serves to represent the behavior of VPA apps, offering an end-to-end overview that guides the testing process. There are no "correct" behavior models, only good ones, which can explicitly represent VPA apps' behavior. To assess their quality, we asked domain experts to manually construct behavior models at the functionality level using the same interaction records. We then compared those expert-generated models with those generated by Elevate.

Our comparison was conduct on two aspects: 1) state extraction, and 2) transition generation. For state extraction, we calculated accuracy by counting the number of states extracted by Observer that were consistent with the expert-generated states. For transition generation, a transition $\delta(s_1, input_1) = s_2$ was recognized correct only if both s_1 and s_2 matched the corresponding exper-defined states. Only Elevate (GPT4) and Elevate (Deepseek) were involved in this evaluation because Vitas constructs the behavior model at the sentence level and chatbot style testers do not construct behavior models during testing.

Tab.5 shows the accuracy of state extraction and transition generation achieved by Elevate (GPT4) and Elevate (Deepseek). Both implementations achieve over 99% accuracy in state extraction and transition generation compared with expert-generated behavior models. This high accuracy demonstrates the effectiveness of LLMs in semantic understanding, enhanced by our conversation-based prompting and feedback mechanisms. The prompts guide Observer to consider semantics (functionalities) when extracting the state, and output it in a fixed form. Without the feedback mechanism, Observer achieves an average state extraction accuracy of 87.4% and 88.5% on GPT-4 and Deepseek, respectively. The feedback mechanism, Guidance, further boosts this accuracy by over 10%. Guidance enforces domain-specific rules to ensure the availability, consistency of input events and deterministic feature of the behavior model. These rules are informed by knowledge of VPA app design and semantically similar outputs characteristics, helping double-check and correct state extraction when ambiguity arises.

Although there are few interpretations in the behavior model, they do not greatly influence the overall system stability. In fact, there are ambiguous cases making it inherently hard to determine whether two outputs convey similar functionality. We present a case study in Section 5.1 to further discuss Elevate's failure and evaluate the effect of these misinterpretations on the effectiveness of the overall testing.

Answers to RQ4: The state extraction and transition generation accuracy exceed 99% for both Elevate implementations. This high performance is largely attributed to the use of conversation based prompts, the feedback mechanism, and the intrinsic semantic understanding capabilities of large language models, all of which contribute significantly to constructing a high-quality behavior model.

4.2.5 RQ5: Large-scale Evaluation.

We further conduct a large-scale evaluation on the Stable-Large-Dataset, which consists of 4,000 apps. Although the Stable-Large-Dataset includes apps from all categories listed on the Amazon's website, the distribution of apps across these categories is uneven but reflects the distribution in the Amazon store. For example, the "Local" category only contains 18 skills, whereas the "Games Trivia" category contains 530 skills. We limit the testing time to 10 minutes per skill. Each evaluation method requires 40,000 minutes to test the Stable-Large-Dataset sequentially. When Elevate is implemented with Deepseek and GPT-4, the total cost to test the Stable-Large-Dataset amounts to 310.5 dollars and 7.9 dollars, respectively. The cost difference mainly comes from the varying prices of the underlying LLMs. Due to the high computational and financial cost , we limit the comparison to Elevate and the previous state-of-the-art baseline, Vitas.

Table 6 presents the number of apps identified with different types of problems. As summarized by Vitas [28], five common issues are frequently observed in VPA applications. A VPA app should not terminate abruptly without an explicit termination signal (e.g., saying "Goodbye"). If it exits unexpectedly, it is classified as an unexpected exit problem. If a VPA app requests personal or sensitive information without the user's permission, it is flagged for violating privacy policies. Unstoppable app refers to when users say stop words like "Stop", "Exit", or "Cancel", the VPA app is expected to exit. If it fails to terminate as instructed, it is classified under exit problems. If a VPA app launches an unintended application instead of responding correctly to user input, it falls under the unexpected app started category. If a VPA app fails to launch when prompted, it is classified as an unstartable app problem. We can find that in total 4000 apps, Elevate (DeepSeek) and Elevate (GPT4) detected 3541 and 3486 problems respectively. In contrast, Vitas only has 2876 problems detected. Furthermore, we find that the majority of detected problems are in the category of unexpected exit and unstartable app. In contrast, the other categories account for a small proportion. The experimental results indicate that most of the detected problems arise during the startup and termination phases of VPA applications. In general, we can conclude that Elevate demonstrates superior effectiveness in identifying problems.

Table 6. The number of problems detected by Vitas, Elevate (GPT4) and Elevate (Deepseek) on the Stable-Large-Dataset.

Approach	Total	Unexpected Exit	Privacy Violation	Unstoppable App	Unexpected App Started	Unstartable App
Vitas	2,878	1,736	67	60	154	1,007
Elevate (GPT4)	3,486	1,899	78	133	192	1,430
Elevate (DeepSeek)	**3,541**	**1,957**	**79**	**157**	**211**	**1,508**

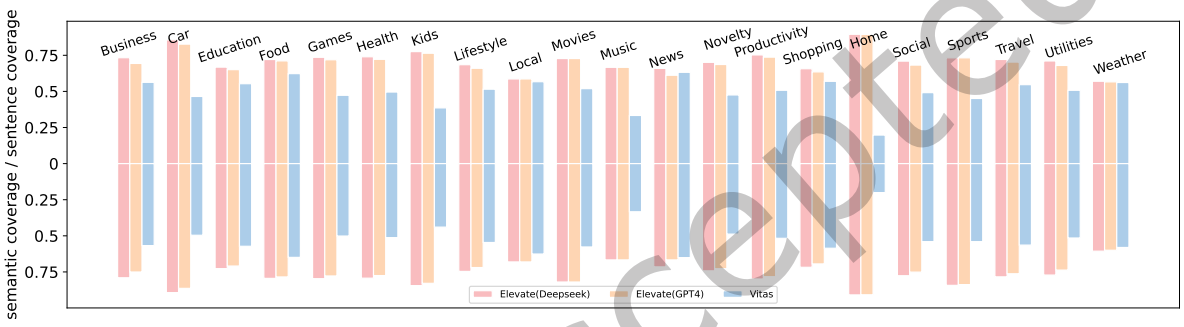


Fig. 15. Sentence state and semantic state coverage rate calculated by Elevate and Vitas on the Stable-Large-Dataset.

The coverage rate analysis on the Stable-Large-Dataset is presented in Fig. 15, where the x-axis represents the app categories, as listed on the official Alexa skill website [1]. From the results, we observe that Elevate achieves a sentence state coverage of over 65% and a semantic state coverage of over 70% across most categories. In comparison, Vitas performs significantly worse, with sentence and semantic state coverage rates below 55% and 60%, respectively, in most categories.unication, Elevate performs more better than Vitas. In categories “Music”, “Home”, “Car” and “Kids”, Elevate exhibits a substantial advantage over Vitas, achieving nearly 30% higher semantic state coverage. We hypothesize that LLMs possess richer domain knowledge about these applications. For other categories, while the improvement of Elevate may not be as substantial as in the aforementioned categories, the results still demonstrate a consistent performance advantage, confirming the robustness of Elevate in VPA testing across diverse skill domains.

Answers to RQ5: The evaluation results on the Stable-Large-Dataset, which contains 4,000 skills, demonstrate that Elevate significantly outperforms Vitas in both the number of detected problems and sentence/semantic state coverage. These results further confirm Elevate ’s superiority in comprehensive VPA testing, highlighting its ability to uncover more problems than Vitas.

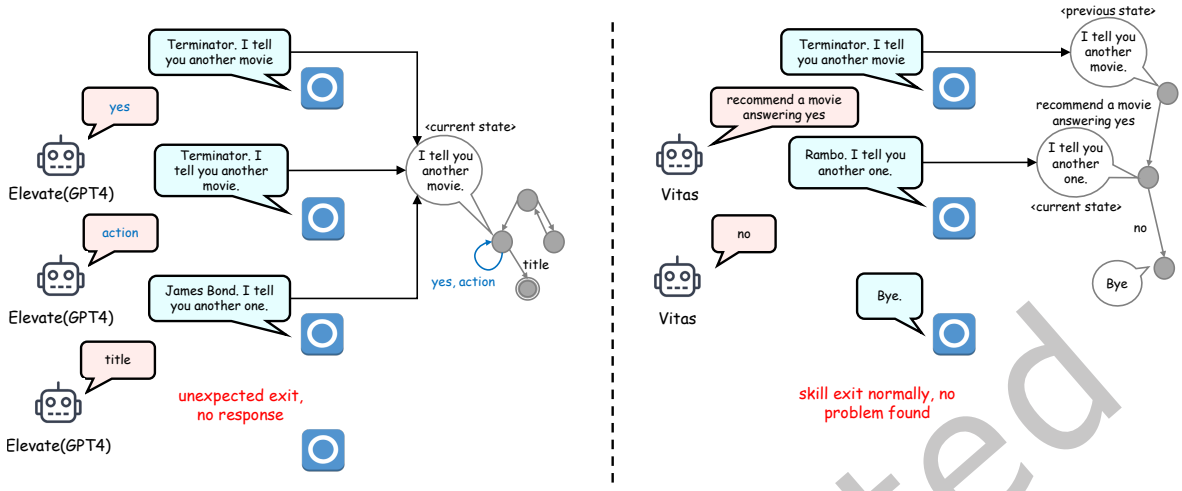


Fig. 16. Considering semantic relevance when analyzing outputs and inputs.

5 DISCUSSION

5.1 Case Study

Diverse and context-related test cases. VPA applications require specific inputs to trigger new behaviors, otherwise, they may repeat the same responses, preventing testers from uncovering hidden functionalities and weaknesses. Elevate leverages LLM's generation and learning capabilities to automatically generate concise and meaningful test cases that are contextually relevant to the VPA app's expected input. For example, in the Alexa skill "My Horoscope", when the skill prompts is "Which zodiac sign do you want to set as default?" A valid response must be a zodiac sign. Elevate correctly generates all twelve zodiac signs (e.g., Aries, Taurus, Gemini, etc.) as candidate inputs, ensuring that the test inputs align with the app's expected behavior. In contrast, Vitas generates incomprehensible candidate inputs such as "transform", "confidence", and "reinforces", which do not match the skill's expected input format. This failure to understand the semantic context prevents Vitas from effectively testing the skill's behavior.

Semantic state extraction. An accurate behavior model is essential for effective state exploration by the Planner. In contrast, previous approaches like Vitas rely on textual equivalence to distinguish different states, which often leads to redundant state spaces and repeated tests. Fig. 16 illustrates the differences between Elevate and Vitas in state extraction and their subsequent impact on testing. Elevate 's Observer merges semantically similar outputs. For example, the sentences "I tell you another movie." and "I tell you another one." are merged into a single state since they convey the same meaning. This allows the system to connect prior testing information from "I tell you another movie." with the current output "I tell you another one.", ensuring better state continuity. Elevate 's Planner benefits from a behavior model. When queried in the third round, after the skill outputs "James Bond. I tell you another one.", the Planner has full access to the current state's history. This includes previous transitions such as δ(<current state>, "yes") = <current state>and δ(<current state>, "action") = <current state>. The Planner learns that "yes" and "action" are invalid inputs, allowing it to avoid redundant selections, improving testing efficiency. Vitas treats semantically similar outputs as different states. Vitas incorrectly classifies "I tell you another movie." and "I tell you another one." as separate states, failing to recognize their semantic equivalence. As a result, when reaching the third round, Vitas considers it a completely new state and lacks any historical execution data

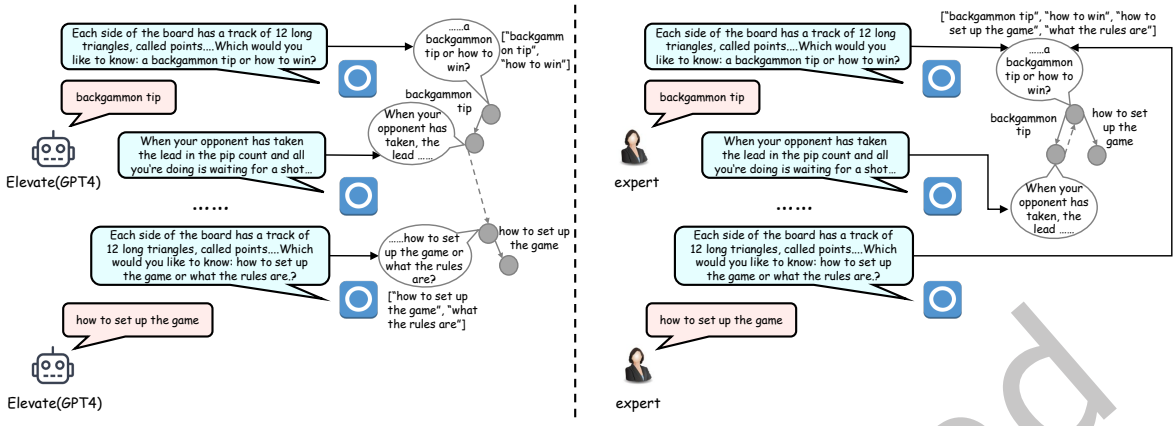


Fig. 17. Elevate may fail to extract semantic states for ambiguous outputs.

to inform its decisions. This case highlights that Elevate 's semantic-aware state merging significantly enhances testing efficiency, allowing the Planner to make smarter decisions, while Vitas' reliance on textual equivalence results in redundant and inefficient testing.

Semantic-relevant input generation. We observe that Elevate 's Planner initially prioritizes selecting inputs that are most semantically related to the context. However, as interaction rounds increase, the Planner adapts its selection strategy by considering historical transitions and invocation frequency to optimize state exploration. Fig. 16 illustrates the different exploration strategies employed by Elevate and Vitas. In the first interaction round, the Planner selects "yes" because it is the most contextually relevant response. However, contrary to expectations, "yes" fails to uncover new states. To continue exploration, the Planner dynamically adjusts its strategy and subsequently tries "action" and "title" as inputs. This strategy ultimately leads Elevate to detect an unexpected exit problem, demonstrating its effectiveness in systematically uncovering hidden issues. In contrast, Vitas' rigid selection strategy leads to missed issues, for example, Vitas selects "no" as the input, which prematurely ends the conversation.

Failure in extracting states for ambiguous outputs. Fig. 17 presents a case where Elevate generates semantic states that differ from those identified by human experts. The apps' outputs—"Each side of the board has a track of 12 long triangles, called points....Which would you like to know: a backgammon tip or how to win?" and "Each side of the board has a track of 12 long triangles, called points....Which would you like to know: how to set up the game or what the rules are?"—are mapped to two different states, S_1 and S_2, by Elevate, but merged into a single state (S_1) by a testing expert. This case is considered ambiguous because the first part of the outputs is identical, while the second part suggests entirely different follow-up inputs. The "Should Merge Suggestion" by the Transition Checker in Guidance is not applicable here, as there are no conflicting transitions in the behavior model.

As we mentioned in the above case studies, Planner relies on the behavior model to make efficient decisions, and inaccurate state extraction can lead to repeated tests. Nevertheless, Elevate's stability is not affected by minor misinterpretations. In this case, although Observer "mistakenly" maps the two outputs to different states, their candidate input events do not have overlap. Specifically, the knowledge learned from S_1 — that "a backgammon tip" or "how to win" can trigger new states — is not applicable to S_2, because these input events are not among S_2's candidate input events. Therefore, the misinterpretation has no impact on subsequent testing.

5.2 Limitations

Despite its effectiveness, Elevate has several limitations. ① Text-Only Input Handling. Similar to previous approaches [28], Elevate processes only textual outputs provided by the simulator. It does not consider extra modalities, such as audio files or images, which may contain additional context useful for testing. ② LLM as a Black-Box. Elevate treats LLMs as black-box models for VUI testing. It does not fully utilize internal LLM information, such as token probabilities or confidence scores, which could potentially enhance state exploration and input selection strategies. ③ LLM Hallucinations. LLMs are prone to hallucinations, which may lead to incorrect or irrelevant inputs during testing. To mitigate this issue, Elevate employs a feedback mechanism that detects and corrects hallucinated outputs, ensuring more reliable interactions. Future work could focus on incorporating multimodal testing, leveraging internal LLM insights, and enhancing robustness against hallucinations to further improve Elevate 's capabilities.

5.3 Threats to Validity

The first potential threat to our evaluation is that some VPA skills are still under development, leading to unstable behavior. As a result, the testing results may change over time, affecting the reliability of our findings. To mitigate this threat, we pre-filter skills with unstable behavior from the initial dataset. Elevate is then evaluated only on the remaining stable skills, ensuring that the results reflect consistent and reproducible testing performance. Another threat arises from the dataset skew. Since the distribution of skills on the Amazon store is uneven, the Stable-Large-Dataset inherits this imbalance. Categories such as "Local" and "Connected Car" contain fewer than 30 skills each, which may affect the representativeness of the coverage rate within these categories. The third potential threat is that our approach has been evaluated using only two LLMs, namely GPT-4 and DeepSeek. We acknowledge that other LLMs exist and could also be used for evaluation. However, we believe that the selected LLMs are representative, as both GPT-4 and DeepSeek possess strong capabilities, making them well-suited for VPA testing. Future work could extend the evaluation to additional LLMs to further validate the generalizability of our approach. Another potential threat arises from the prompt settings, as experimental results can be influenced by different prompt engineering strategies. To mitigate this threat, our prompts were iteratively refined and manually tested to ensure stability and reliability. This process helps minimize the impact of prompt variations on the experimental results, ensuring that our findings are robust and not overly dependent on specific prompt formulations.

6 RELATED WORK

VPA Apps Testing: Several studies have explored VPA app behavior using dynamic testing techniques [15, 17, 21, 28, 29, 34, 38]. These approaches can be broadly classified into chatbot-style testers, model-based testing, and static analysis methods. SkillExplorer[21], VerHealth[34], and SkillDetective [38] are classic chatbot-style testers. These methods generate test cases based on the latest VPA app outputs and employ a DFS-based test case selection strategy for exploration. VUI-UPSET [15, 17] also follows a chatbot-style testing approach but focuses on generating paraphrases to reveal bugs and functional inconsistencies in VPA applications. Vitas [28] is a model-based testing approach designed for VUI testing. While Vitas improves coverage and efficiency, it relies on simple rule-based model construction and fails to incorporate semantic information, limiting its effectiveness in complex scenarios. SkillScanner [29] is the first static analysis approach designed to detect privacy leaks in VPA applications. It employs taint analysis to track potential privacy violations and conducts experiments on an open-source dataset collected from GitHub. Compared to these methods, Elevate introduces a multi-agent framework to mitigate semantic loss at every stage of VUI testing. It is further enhanced by a behavior model, which provides global testing insights for guidance.

Large Language Model for Software Testing: As LLMs continue to advance, they have been increasingly applied to different fields [11, 18–20, 24, 25, 32] of software engineering. They have also been widely adopted in software testing. Several studies have explored their potential in test case generation, fuzzing, and GUI testing. Codet [11] leverages LLMs to automatically generate test cases for assessing the quality of code solutions. CodaMosa [27] utilizes Codex to generate test cases when search-based testing methods reach bottlenecks, improving test effectiveness. TitanFuzz [13] employs LLMs to generate and mutate deep learning (DL) program inputs for fuzz testing DL libraries. FuzzGPT [14], a follow-up to TitanFuzz, primes LLMs to synthesize bug-triggering programs, demonstrating improved bug detection performance. Other research has focused on testing mobile app GUIs, where LLMs generate context-aware textual inputs or human-like actions for more realistic testing scenarios [30, 31]. Building on these advancements, our work introduces LLM-based VUI testing, which extends LLM applications into the domain of VPA application testing.

7 CONCLUSION

In this work, we propose Elevate, a model-enhanced LLM-driven model-based testing framework for VPA apps. Elevate proposes a new paradigm of VUI testing that involves three LLM agents' collaboration. Each of them processes a task of perceiving outputs to extract states, understanding VPA apps' context to generate effective input events or making efficient exploration decisions. Additionally, we innovatively propose a model-enhancement technology. Different from traditional model-based testing, the model is not only used for guidance, but also embedded in agents' prompts to provide global testing information and feedback. The experimental results indicate that Elevate detects more problems and achieves a higher coverage rate than the previous state-of-the-art works.

ACKNOWLEDGMENTS

We are grateful for the constructive feedback of all the anonymous reviewers to improve this manuscript. The authors from Nanjing University are supported in part by the National Key Research and Development Program of China (2024YFB2505604) and the National Natural Science Foundation of China (No.62232008, 62172200). The authors from IIE, Chinese Academy of Science, are supported in part by NSFC (U24A20236), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118). The authors from University of Queensland are supported in part by the National Natural Science Foundation of China (Grant No. 62572256).

REFERENCES

[1] 2014. Amazon.com: Alexa Skills. https://www.amazon.com/alexa-skills/.
[2] 2017. Alexa Simulator limitations. https://developer.amazon.com/en-US/docs/alexa/devconsole/test-your-skill.html#use-simulator.
[3] 2018. Portland Family Says Their Amazon Alexa Recorded Private Conversations. https://www.wweek.com/news/2018/05/26/portland-family-says-their-amazon-alexa-recorded-private-conversations-and-sent-them-to-a-random-contact-in-seattle/.
[4] 2021. Scenes|Conversational Actions|Google Developers. https://developers.google.com/assistant/conversational/scenes.
[5] 2022. Total number of Amazon Alexa skills in selected countries as of January 2021. https://www.statista.com/statistics/917900/selected-countries-amazon-alexa-skill-count/.
[6] 2023. Amazon-Alexa-UQ-AAS21-datasets. https://github.com/xie00059/Amazon-Alexa-UQ-AAS21-datasets/tree/main.
[7] 2023. GPT-4. https://openai.com/gpt-4.
[8] 2024. text2vec. https://github.com/shibing624/text2vec.
[9] 2025. DeepSeek. https://www.deepseek.com/.
[10] 2025. Elevate. https://github.com/Elevate-12/Elevate-2025.
[11] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.
[12] Long Cheng, Christin Wilson, Song Liao, Jeffrey Young, Daniel Dong, and Hongxin Hu. 2020. Dangerous Skills Got Certified: Measuring the Trustworthiness of Skill Certification in Voice Personal Assistant Platforms. In *CCS '20: 2020 ACM SIGSAC Conference on Computer*

and Communications Security, Virtual Event, USA, November 9-13, 2020, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1699–1716.

[13] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, Ren'é Just and Gordon Fraser (Eds.). ACM, 423–435.

[14] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 70:1–70:13. https://doi.org/10.1145/3597503.3623343

[15] Emanuela Guglielmi and Giovanni Rosa and Simone Scalabrino and Gabriele Bavota and Rocco Oliveto. 2024. Help Them Understand: Testing and Improving Voice User Interfaces. *ACM Trans. Softw. Eng. Methodol.* 33, 6, Article 143 (2024), 33 pages. https://doi.org/10.1145/3654438

[16] Marcia Ford and William Palmer. 2019. Alexa, are you listening to me? An analysis of Alexa voice service network traffic. *Pers. Ubiquitous Comput.* 23, 1 (2019), 67–79.

[17] Emanuela Guglielmi, Giovanni Rosa, Simone Scalabrino, Gabriele Bavota, and Rocco Oliveto. 2022. Sorry, I don't Understand: Improving Voice User Interface Testing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 96:1–96:12.

[18] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[19] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 34:1–34:13. https://doi.org/10.1145/3597503.3623306

[20] Qi Guo, Xiaofei Xie, Shangqing Liu, Ming Hu, Xiaohong Li, and Lei Bu. 2025. Intention is All You Need: Refining Your Code from Your Intention. *arXiv preprint arXiv:2502.08172* (2025).

[21] Zhixiu Guo, Zijin Lin, Pan Li, and Kai Chen. 2020. SkillExplorer: Understanding the Behavior of Skills in Large Scale. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2649–2666.

[22] Mathav Raj J, Kushala VM, Harikrishna Warrier, and Yogesh Gupta. 2024. Fine Tuning LLM for Enterprise: Practical Guidelines and Recommendations. *CoRR* abs/2404.10779 (2024).

[23] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. In *NeurIPS*.

[24] Jiaolong Kong, Mingfei Cheng, Xiaofei Xie, Shangqing Liu, Xiaoning Du, and Qi Guo. 2024. Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs. *arXiv preprint arXiv:2403.01971* (2024).

[25] Jiaolong Kong, Xiaofei Xie, and Shangqing Liu. 2025. Demystifying Memorization in LLM-Based Program Repair via a General Hypothesis Testing Framework. *Proc. ACM Softw. Eng.* 2, FSE (2025), 2712–2734. https://doi.org/10.1145/3729390

[26] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. 2018. Skill Squatting Attacks on Amazon Alexa. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 33–47.

[27] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931.

[28] Suwan Li, Lei Bu, Guangdong Bai, Zhixiu Guo, Kai Chen, and Hanlin Wei. 2022. VITAS : Guided Model-based VUI Testing of VPA Apps. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 115:1–115:12.

[29] Song Liao, Long Cheng, Haipeng Cai, Linke Guo, and Hongxin Hu. 2023. SkillScanner: Detecting Policy-Violating Voice Applications Through Static Analysis at the Development Phase. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2321–2335.

[30] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1355–1367.

[31] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 100:1–100:13.

https://doi.org/10.1145/3597503.3639180

[32] Lezhi Ma, Shangqing Liu, Lei Bu, Shangru Li, Yida Wang, and Yang Liu. 2024. SpecEval: Evaluating Code Comprehension in Large Language Models via Program Specifications. *arXiv preprint arXiv:2409.12866* (2024).

[33] Murray Shanahan. 2024. Talking about Large Language Models. *Commun. ACM* 67, 2 (2024), 68–79. https://doi.org/10.1145/3624724

[34] Faysal Hossain Shezan, Hang Hu, Gang Wang, and Yuan Tian. 2020. VerHealth: Vetting Medical Voice Applications through Policy Enforcement. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 4 (2020), 153:1–153:21. https://doi.org/10.1145/3432233

[35] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[36] Fuman Xie, Chuan Yan, Mark Huasong Meng, Shao-Ming Teng, Yanjun Zhang, and Guangdong Bai. 2024. Are Your Requests Your True Needs? Checking Excessive Data Collection in VPA App. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 205:1–205:12.

[37] Fuman Xie, Yanjun Zhang, Chuan Yan, Suwan Li, Lei Bu, Kai Chen, Zi Huang, and Guangdong Bai. 2022. Scrutinizing Privacy Policy Compliance of Virtual Personal Assistant Apps. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 90:1–90:13.

[38] Jeffrey Young, Song Liao, Long Cheng, Hongxin Hu, and Huixing Deng. 2022. SkillDetective: Automated Policy-Violation Detection of Voice Assistant Applications in the Wild. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 1113–1130.

[39] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. 2019. Dangerous Skills: Understanding and Mitigating Security Risks of Voice-Controlled Third-Party Functions on Virtual Personal Assistant Systems. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1381–1396.

[40] Nan Zhang, Xianghang Mi, Xuan Feng, Xiao Feng Wang, Yuan Tian, and Feng Qian. 2019. Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems. *IEEE Symposium on Security and Privacy* (2019).

[41] Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. 2019. Life after Speech Recognition: Fuzzing Semantic Misinterpretation for Voice Assistant Applications. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

[42] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. *CoRR* abs/2303.18223 (2023).