GenieWizard: Multimodal App Feature Discovery with Large Language Models

Jackie (Junrui) Yang
Computer Science Department
Stanford University
Stanford, California, USA
Skywalk Inc.
Palo Alto, California, USA
jackie@jackieyang.me

Yingtian Shi
School of Interactive Computing
Georgia Institute of Technology
Atlanta, Georgia, USA
yshi457@gatech.edu

Chris Gu
Computer Science Department
Stanford University
Stanford, California, USA
cgu26@stanford.edu

Zhang Zheng
Computer Science Department
Stanford University
Stanford, California, USA
zhangzhengzr@gmail.com

Anisha Jain
Independent Researcher
Stanford, California, USA
anishaj037@gmail.com

Tianshi Li
Khoury College of Computer Sciences
Northeastern University
Boston, Massachusetts, USA
tia.li@northeastern.edu

Monica S. Lam
Computer Science Department
Stanford University
Stanford, California, USA
lam@cs.stanford.edu

James A. Landay
Computer Science Department
Stanford University
Stanford, California, USA
landay@stanford.edu

Abstract

Multimodal interactions are more flexible, efficient, and adaptable than graphical interactions, allowing users to execute commands beyond simply tapping GUI buttons. However, the flexibility of multimodal commands makes it hard for designers to prototype and provide design specifications for developers. It is also hard for developers to anticipate what actions users may want. We present GenieWizard, a tool to aid developers in discovering potential features to implement in multimodal interfaces. GenieWizard supports user-desired command discovery early in the implementation process, streamlining the development process. GenieWizard uses an LLM to generate potential user interactions and parse these interactions into a form that can be used to discover the missing features for developers. Our evaluations showed that GenieWizard can reliably simulate user interactions and identify missing features. Also, in a study (N = 12), we demonstrated that developers using GenieWizard can identify and implement 42% of the missing features of multimodal apps compared to only 10% without GenieWizard.

CCS Concepts

• **Human-centered computing** → **Interaction paradigms**; *Systems and tools for interaction design*; *Natural language interfaces*; • **Software and its engineering** → Software creation and management; • **Computing methodologies** → *Natural language processing*.

Keywords

Multimodal interfaces, developer tools, large language models, feature discovery, interaction simulation, voice interfaces, touch interfaces, semantic parsing, multimodal app development

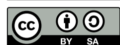
ACM Reference Format:

Jackie (Junrui) Yang, Yingtian Shi, Chris Gu, Zhang Zheng, Anisha Jain, Tianshi Li, Monica S. Lam, and James A. Landay. 2025. GenieWizard: Multimodal App Feature Discovery with Large Language Models. In *CHI Conference on Human Factors in Computing Systems (CHI '25), April 26–May 01, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3706598.3714327

1 Introduction

Multimodal interactions[1] allow users to engage with computer systems using a combination of multiple input modalities, such as touch and voice. These interactions have been proven to offer more flexibility, efficiency, and adaptability for various users and tasks [64]. Unlike touch-only interactions, where users are limited to actions displayed on a graphical user interface (GUI), multimodal interactions allow users to express their intentions using a combination of modalities. Our prior research has shown that even apps developed with a state-of-the-art multimodal framework may fail to support 41% of the desired commands from real users [68]. This result is caused by users voicing any command that comes to mind,

[1]Specifically, this paper targets multimodal apps that use deictic gesture + speech interactions. This is a common category of multimodal applications proposed by Oviatt [51].



This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

CHI '25, Yokohama, Japan

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1394-1/25/04
https://doi.org/10.1145/3706598.3714327

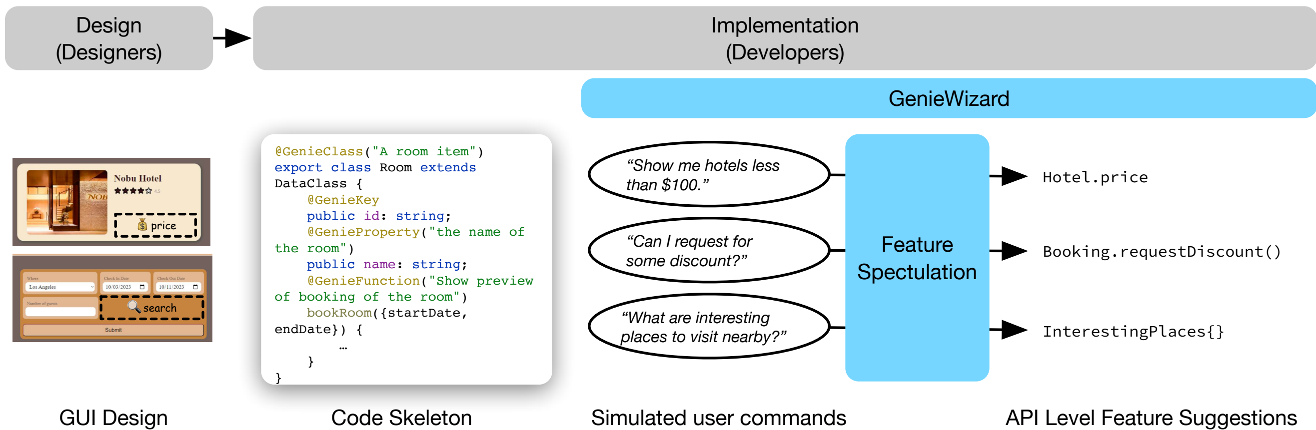


Figure 1: GenieWizard helps developers discover commonly expected features in multimodal apps: Based on the *GUI design* provided by the designers, multimodal app developers write an initial *code skeleton*. GenieWizard can help developers by suggesting features that need to be implemented to provide a good initial app experience. GenieWizard achieves this by *simulating user commands* using a large-language model(LLM), speculating about missing features using a zero-shot parser and abstract interpretation, and providing actionable *API-level feature suggestions* to developers.

not being limited to what is displayed on the GUI. As such, it is significantly more difficult for multimodal apps to provide a smooth user experience without interruptions from unsupported command errors. If an app cannot handle the commands a user issues, it may lead to frustration and discourage natural exploration [43].

Feature discovery for GUI apps can be supported by user testing using design and prototyping tools like Figma without implementing the full system and UI. However, the same cannot be done for multimodal UIs due to the complexity of implementing multimodal interactions. This paper shows how we can support feature discovery for *multimodal* apps without implementing a full prototype.

1.1 Feature Discovery for GUI and Multimodal Apps

The Software Development Life Cycle (SDLC) [4] has several variations of workflows, such as the waterfall model [55], spiral model [9], incremental model [54], and agile methods [33]. These models include common stages, including design, implementation, and testing. Designers can address most GUI usability issues at the design stage using GUI design prototypes. GUI design tools such as Figma support user testing with an interface prototype. As users interact with the prototype, a prepared screen representing the GUI design will appear, allowing the user to test the app without needing a code-based implementation.

Consider the development of a hotel search application, as shown in Figure 1. Designers can usually make a clickable GUI prototype with little to no code. From user feedback during usability testing, they can discover necessary GUI improvements like adding a button (e.g., a search button) or a label (e.g., a price label). The missing corresponding functions, such as a search function `hotel.search()` and a price property `hotel.price` can be captured in a design specification, such as a Figma document, which is then used to inform the implementation.

Multimodal commands are inherently richer and more flexible. In this hotel search example, the user may ask "Show me hotels for less than $100", or "Can I request a discount for this hotel" while simultaneously tapping on a hotel entry. These are both reasonable commands to use when searching for a hotel, but their implementation is nontrivial. Multimodal apps can better address the exponentially many combinations of features conveyed in multimodal commands through an LLM-based neural semantic parser. For example, ReactGenie [68] uses such a parser to translate natural language into an executable domain-specific language (DSL) that uses compositional constructs to connect implemented functions and the user's current interaction (e.g., voice and touch) to execute the user's command. In contrast to what is needed to support the more limited functionality of GUI applications, multimodal applications necessitate more extensive and more expensive user research during the initial development phase. This is because developers need a set of common user commands that align with user expectations to plan out the functions to implement. So, discovering commonly used features early in the development process is even more crucial for multimodal apps.

With GUI applications, features[2] are closely tied to interface elements, enabling developers to easily link interface operations with their corresponding implementations. However, the features in multimodal apps can be nontrivial to conceptualize, meaning that without having a deep understanding of how the app will be implemented, developers cannot anticipate whether a desired multimodal command can be supported and which group of commands can be supported through the same underlying features. For example, a hotel search app probably already has logic dealing with hotel pricing information in its database. If the user says, "Show me hotels for less than $100," the semantic parser can automatically translate the command into code that filters hotels based on pricing

[2]Here, "features" means lower-level programming features, i.e., specific classes, properties, and functions in an app.

information. In contrast, a command such as "What are interesting places to visit nearby?" likely requires more code changes. For example, this might require creating a new `attractions` class, and the semantic parser would produce a query involving the hotel and nearby attractions.

In summary, **for multimodal apps, it is both more challenging and important to discover features without a prototype implementation**. **Multimodal interactions present two major challenges to the development workflow:**

(1) How can we prototype multimodal interactions without extensive implementations?
(2) How can missing features be conveyed in a way that is actionable for developers?

1.2 Introduction to GenieWizard

We present GenieWizard, a developer tool (see Figure 4 for the user interface) that aids developers in discovering the features needed to support common multimodal interactions in the early stages of implementation. GenieWizard provides an IDE that allows developers to code as usual, while also providing feature suggestions and an interface to track their implementation progress towards these features. To use GenieWizard, developers only need to provide early-stage code skeletons (i.e., class/property/function definitions). They do not need to provide full implementations, and there are no GUI designs required. Through an automated feature suggestion pipeline, GenieWizard can suggest potential features to implement potential user commands. Figure 2 provides an overview of GenieWizard.

1.2.1 Feature discovery before prototyping. Our approach is to leverage the generative power of large language models (LLMs) to *simulate* user testing with a *simulated* app. GenieWizard first derives an app description from the code skeleton. It uses an LLM to generate personas of potential users based on the app description and randomly sampled demographic data. Then, it uses an LLM to simulate commands from each persona interacting with an app instructed to behave according to the same app description.

1.2.2 Suggesting missing features to developers. GenieWizard's next step is to suggest features for the app implementation. Note that, as stated in Section 1.1, multimodal features are nontrivial to deduce. We cannot directly tell what features are missing from generated user interactions without first parsing them into actual commands. Therefore, the suggestions need to be at the API level to be effective. We discuss more about this in Section 6.1.

We leverage the hallucination of LLMs as generative power to design the missing features. An LLM-based neural semantic parser is instructed to use a set of APIs given by developers. What happens if a user command cannot be implemented with any of the given APIs? The parser will hallucinate, referring to classes, properties, and functions that do not exist. In production, we need to suppress the hallucinations and force the parser to recognize that the requested command is not yet supported.

Our novel design turns this LLM bug into a feature by explicitly encouraging the hallucination behavior when the app cannot fulfill the interaction required by the simulated user. This is achieved with a carefully crafted prompt we supply to the LLM. Next, GenieWizard scans the generated API calls for missing classes, properties, and functions, clusters similar ones, and provides a concrete and concise list of suggested features (i.e., classes, properties, and functions) for developers to implement.

1.2.3 Workflow Integration of GenieWizard. We imagine GenieWizard will be used in the initial implementation stage of a multimodal app's development cycle (see Figure 1). Designers will provide a full GUI design and list the app's target use cases. Developers first design an initial app architecture—especially the state code skeleton—based on the GUI design and specified use cases. They then provide this initial code (along with the use case details as part of the app description) to GenieWizard, which helps refine the architecture and implementation until the app can support the predicted common user requests. The goal is that the first version of the new multimodal app will be much more functional (less unsupported commands) for end-users with the help of GenieWizard.

1.2.4 Contributions. Our main contributions include:

- A novel system, GenieWizard, that helps developers discover a rich and useful feature set for a multimodal application, automatically, early in the implementation cycle, through LLM-based user/app interaction simulations.
- A technical approach incorporating a zero-shot parser and a dry run utility that can understand simulated multimodal commands and translate them into missing properties, functions, and classes.
- An evaluation of GenieWizard showed that generated user commands cover, on average, 71% of the desired user commands, the zero-shot parser can match the performance of the few-shot parser of the prior work, and developers using the GenieWizard plugin can implement more of the desired commands (42%) than using a baseline tool (10%).

2 Related Work

Our work builds upon and extends many prior research projects. The topics include: UI design and testing tools, UI feedback tools with AI, developer tools with AI, multimodal development frameworks, and multimodal interactions frameworks with AI. In this section, we will review key contributions to these topics and show how our approach differs.

2.1 UI Design and Testing

The evaluation of user experience is an important part of software development [32]. The feedback obtained from testing with users can be utilized for program and interface improvements. Nielsen and Molich [48] first proposed seven factors, including error prevention, user control, and freedom, among others, as usability heuristics for designers and developers to test. Designers and developers can use analytical evaluation techniques to collect user feedback and examine the user experience. Thinking aloud [40] requires users to make spoken comments while accomplishing specific interaction tasks. Keystroke-level analysis [17] predicts how long it will take an expert user to accomplish a routine task without errors using an interactive computer system to help evaluate user experience.

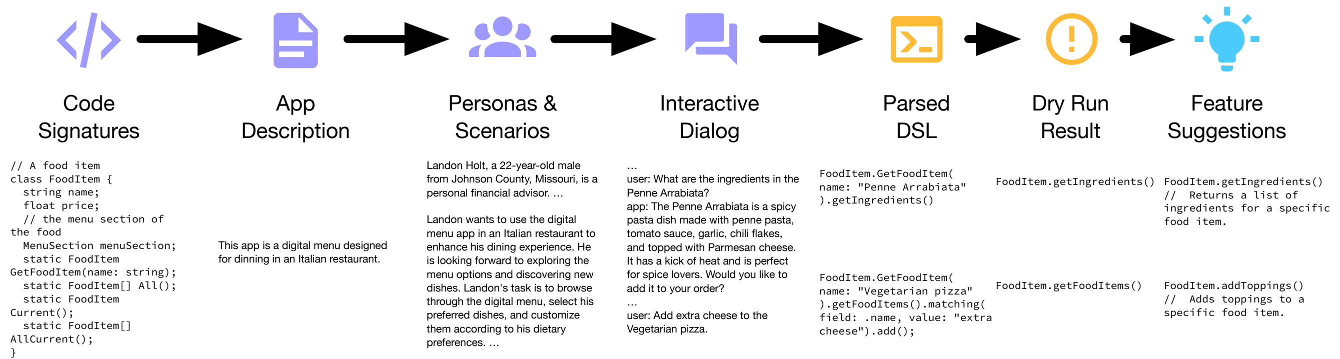


Figure 2: GenieWizard Suggestion Generation Pipeline: GenieWizard's suggestion generation pipeline starts from developer-provided code signatures and generates feature suggestions through an automated pipeline. It involves three general stages: simulating user commands (purple, left four), speculating on missing features (yellow, center two), and suggesting features to implement (blue, right one).

Other methods, such as long-term diary research [10], daily reconstruction methods [35], and the experience sampling method (ESM) [46] are used to assess the usability of software during use.

However, most of these methods generally require users to experience a functioning application, which requires much of the implementation to be finished before obtaining user feedback. Considering the perspectives of risk management and development efficiency [18], prototyping can help developers conduct functional tests of various industrial product outputs during the development stage [16]. Budde et al. [14] classify prototypes according to the manner of their construction. Different prototyping methods allow developers to test user interfaces (horizontal prototypes) or individual features (vertical prototypes) without completing the entire program. The MENULAY [15] and the Dynamic Interface Creation Environment [56, 57] were two of the earliest UI prototyping tools that allowed developers to test the layout of interface elements, such as the placement of text boxes and buttons. Floyd [25] and Naumann and Jenkins [47] all believe that prototyping starts with determining requirements and features, followed by implementation and testing.

Prior research [6, 7, 19, 25, 31, 44] generally identified three categories of prototyping models: Exploratory, Experimental, and Evolutionary Prototyping, tailored to different stages of application development. These models perform well for GUI applications, but the functional requirements brought by multimodal user input are likely to go beyond the visual elements of the interface itself. Faced with the uncertainty of requirements in multimodal applications, these models may struggle to help developers complete the prototyping of multimodal applications. Bourguet [11] attempted to use a Finite State Machine (FSM) to build prototypes of multimodal applications, but the multimodal inputs, such as voice input and clicks, also need to be anticipated by developers in advance, which is difficult given the flexible nature of multimodal interactions.

Therefore, we need a method to gather possible user input, understand what features are required, and present this information in a concise way to developers. Also, all of the above should be performed as early as possible in the development process. GenieWizard simulates the user input-gathering process by using an LLM-powered pipeline that only requires a skeleton of the *state code*[3], which can be created early in the implementation process before any UI code or a functioning implementation. GenieWizard then analyzes the required features using its zero-shot parser and dry run utilities to convert simulated user inputs to a list of required code elements. Finally, GenieWizard provides concise feedback for developers.

2.2 UI Development Feedback Generation with AI

GenieWizard suggests features for multimodal apps to improve the user experience. There has also been some recent work using large language models for testing GUI design and implementation and generating feedback. Liu et al. [42] used LLMs for zero-shot human-like interaction generation for detecting crashing bugs triggered by GUI actions. Duan et al. [23] automated heuristic evaluation of UI designs by feeding an LLM with prompts containing the design guidelines and the UI representation.

Prior work has yet to explore providing feedback on multimodal app development, and generating feedback for such development is more challenging than generating feedback for GUI development. First, no design guidelines are currently available for conducting heuristic evaluations on multimodal apps. Second, the flexibility of user actions possible in multimodal apps makes the potential set of user interactions that should be supported significantly larger than those in GUI apps. GenieWizard tackles these challenges by simulating user actions to bootstrap the feedback process when heuristic evaluation is not feasible. It also leverages persona generation techniques to cover a wide range of user types in the simulated actions. This allows GenieWizard to generate possible user command suggestions based only on the skeleton of the app's state code before developers write functioning implementations.

[3] State code, in the context of GUI development, refers to the part of the application that manages the data and the logic that determines the state of the user interface.

2.3 AI-assisted Developer Tools

The past decade has seen a large body of literature about building developer tools using AI [29]. Below, we summarize major AI-assisted developer tools, synthesize the evolution of AI techniques used to build these tools, and highlight the novel techniques proposed by this work for integrating LLMs into developer tools.

Earlier work on developer tools used traditional machine learning techniques, including supervised and unsupervised methods. For example, Lal and Pahwa [38] used Decision Trees and Support Vector Machine (SVM) models to predict if certain code contained bugs, helping with code review. Nucci et al. [49] used source code metrics, historical data, and manual annotations to train Naive Bayes and J48 probabilistic models to identify bad coding practices. Bader et al. [5] used documented software bug fixes from codebases to power a hierarchal clustering algorithm that identifies bugs in programs and suggests possible bug fixes in a ranked order based on probability. More recent research applies deep-learning-based approaches to building developer tools. These approaches not only allow for better performance in detection tasks but also further enable more novel generation tasks, such as UI generation and documentation generation. Generative Adversarial Networks (GANs) have been researched to improve and automate the process of designing graphical layouts [41, 62]. LayoutTransformer [28] and GUILGET [63] are self-attention-based transformer models that generate and complete design and UI layouts. Jing et al. [34] uses variational autoencoders (VAE) to generate layouts for various product listing pages encountered in mobile shopping applications. Khomh et al. [36] presents a deep neural network that analyzes the structural information of Java methods for code comments generation.

Pre-trained LLMs have the potential to empower more innovative AI-assisted developer tools. Given their code analysis ability and flexible text-based interaction paradigms that support both human languages and programming languages, off-the-shelf LLM-powered chatbots are used by developers to comprehend, write, and debug code. Some researchers have further developed LLM-powered tools that support code understanding [45], the information searching and foraging process [12], code generation based on conversational interactions [58], translating natural language commands to domain-specific language code [68], as well as software testing tasks [67] such as unit test generation [39], test input generation [69], and program repair [53]. In addition, systems have been developed to assist writers in providing feedback during their writing process [8, 20].

GenieWizard has similarities with other LLM-powered software testing tools in that it also helps developers identify and fix issues in the code. At the same time, the novel pipeline it proposes differentiates it from other LLM-powered developer tools that primarily leverage the model's code knowledge. Combining the model's commonsense reasoning and code knowledge automates the entire process, from simulating user behaviors to suggesting missing function code signatures for programming multimodal interactions. We envision this pipeline will enable a new paradigm of AI-based developer tools, which we discuss later in this paper.

2.4 Multimodal App Development Frameworks

Multimodal app development is a challenging task. Before LLMs became widely accessible, researchers had created development frameworks to facilitate the implementation of specific voice commands on top of a GUI. Sarmah et al. [60] developed a tool for adding the voice input modality to existing web apps without requiring significant NLP expertise. However, it only supports template-based matching of multimodal commands (e.g., "add <song name> to the playlist"), and each individual command needs to be added separately. This not only increases the development cost, but also does not fully realize the potential of achieving flexibility, efficiency, and adaptability with multimodal interactions.

With LLMs, researchers have proposed novel programming frameworks to streamline the implementation of multimodal apps that can generalize from a small number of examples to a large set of commands. Wang et al. [66] explore a generalizable approach to adapt an LLM to mobile UIs to support conversational interactions with the UIs. In our prior work, we [68] proposed a pipeline to translate human voice commands to DSL code that can compose the app functions exposed by the developers to support flexible user intentions.

While this recent work has addressed the barrier to implementing a large set of multimodal commands, another significant issue remains unsolved: it is difficult to determine a comprehensive set of functions to implement for supporting the multimodal commands in the code. GenieWizard tackles this problem by analyzing an early version of the code to infer the required functions that can cover common interactions for varied types of users. It also helps developers prioritize suggested features based on their relevance and potential impact on the user experience. This feature ranking is crucial as it guides developers on which functionality to implement first, optimizing the development process and resource allocation.

2.5 Multimodal Interactions with AI Models

While apps such as ChatGPT [50] or Google Gemini [26] support multimodal input and output (typically images and text), they work best for general knowledge questions. People have also tried to make an AI model that allows a user to give voice commands and emulates user clicks/keystrokes as input to a traditional GUI app [2, 66]. While these solutions are easier to code, they suffer from mistakes and the inability to integrate actions on different screens [66]. Another solution is to allow an LLM to generate interfaces on the fly [65] according to the user's command. However, it is harder to make an app conform to a consistent look and feel and this is also more prone to LLM generation errors. Therefore, GenieWizard is based on a more traditional software development process to help developers implement more features to deliver a more usable first iteration of an application that has reliable features and a consistent look and feel.

3 GenieWizard System Design

The goal of GenieWizard is to help developers reduce the "unsupported" errors in their multimodal apps by suggesting features to implement. In a simplified software engineering development lifecycle [3, 59], designers first create prototypes of an app to verify its features and make improvements. Designers then make design

specifications from that process and present them to developers. Developers implement logic (state code) that supports all the features and then implement a UI (UI code) that renders information present in the logic in a way that conforms to the design specs. Finally, the app can be tested by end-users through app user testing.

However, with the introduction of multimodal interactions, prototyping is much harder for designers. For GUI apps, there are many tools to help designers make partially-functioning prototypes that will respond to a user's clicks for testing. Additionally, designers have less ambiguous representations of the design specifications, usually via a graphical storyboard. When developers implement the app to behave exactly like the graphical storyboard, users have a higher chance of having the user experience intended by the designers. However, to create design specifications for multimodal apps, designers must carry out laborious Wizard-of-Oz studies [37] to simulate the system to respond to various multimodal commands and learn the users' desired commands. This is due to the fact that the GUI interaction space is usually limited, i.e., if there are five buttons on a screen, there are only five possible interactions that must be pre-programmed. Multimodal interaction involves not only a large number of actions available on almost every screen but also the exponential number of possible combinations of these actions and references to objects on the screen. Developers must also understand the mapping between user commands and the required properties, functions, and classes to implement them.

In the following, we first give an overview of the multimodal interaction framework ReactGenie (which GenieWizard is based on) and then introduce the reader to the multimodal feature discovery problem GenieWizard is trying to solve. This is followed by an overview of the GenieWizard system and the details of the three key stages in the GenieWizard pipeline.

3.1 Overview of the ReactGenie Framework

To familiarize the readers with multimodal apps, we first describe the ReactGenie framework [68], a state-of-the-art multimodal app framework that supports arbitrary combinations of GUI actions and API calls via voice commands. Our GenieWizard prototype is developed on top of ReactGenie. ReactGenie streamlines the creation of multimodal mobile apps by allowing developers to focus on implementing the app features and the GUI without having to handle each possible multimodal intention manually.

3.1.1 The developer interface. As shown in Figure 3, the developer simply defines the *state code*, which contains classes with properties and functions that define the app's features. Only properties annotated with `@GenieProperty` and functions annotated with `@GenieFunction` will be exposed to the multimodal runtime. Hence, developers can prevent the internal helper functions and properties from being exposed to the user by simply not annotating them, e.g., `reserveRoom()` and `imageUrl`. The developer then defines the UI code, which can be typical React[4] code that renders the GUI based on the state code.

The developer also needs to provide a set of few-shot examples of how natural language commands are to be translated into correct *ReactGenieDSL* code from user commands. For example, as shown in `Examples` in Figure 3, the sentence "*I want to book this room till the end of the week*" is represented in ReactGenieDSL as

```
Room.Current().bookRoom(startDate: DateTime.
today(), endDate: DateTime.today().setDayOf
TheWeek(day: DateTime.Sunday))
```

ReactGenieDSL is a DSL designed to support the composition of GUI actions and developer-provided functions in a syntax that is easy to translate from natural language.

Notice that in this example, a special function that is not required to be provided by the developer is the `Room.Current()` function. This function supports the user's multimodal deictic touch gesture. ReactGenie provides this function by understanding what object in memory is mapped to the user's click point (x,y) using the developer's UI code. This function then returns the same object under the user's touch point. In this way, the user can refer to the object that they are tapping in their multimodal commands.

3.1.2 ReactGenie Framework. ReactGenie uses the developer-supplied examples of translations from English to ReactGenieDSL, along with the extracted state code function signatures, to create an LLM-based semantic parser that turns user utterances into ReactGenieDSL code. When a user issues a command, ReactGenie uses the semantic parser to turn the command into ReactGenieDSL code. The ReactGenieDSL interpreter executes the command within the context of the developer-provided state code. The execution will automatically update the state and trigger a re-rendering of the relevant GUI. ReactGenie further analyzes the execution result and, depending on the result, optionally composes a GUI interface using developer-provided UI code to display the result to the user. Parallel to the GUI rendering, ReactGenie also generates a text response using human-readable descriptions of the execution result.

3.2 The Multimodal Feature Discovery Problem

The problem GenieWizard attempts to address is the following: **Given a prototype of a multimodal app, propose a set of new programming features to implement so that typical end-user requests can be satisfied.**

To illustrate the complexity of this problem, let us consider the example of a functional food-ordering GUI app. Suppose the app provides information about each food item, including ingredients, on an item detail page. An app implementing the GUI may simply have a text field called `Food.description`. A user would like to avoid peanuts because of allergies, so they ask, "Show me all the food that doesn't contain peanuts." To handle this request, we need to add a `Food.ingredients` field and translate the command to `Food.All().ingredients.notContains(item: "peanut")`.

In a traditional development lifecycle, such a problem might only be detected at the software usability testing phase. Our prior work [68] showed that a multimodal app developed with the ReactGenie framework still leaves 41% of the commands desired by users unsupported by the study participants' implementations. As demonstrated by [43], the portion of unsupported commands affects people's willingness to use the system and makes early testing of multimodal systems in the wild infeasible.

[4] https://react.dev/

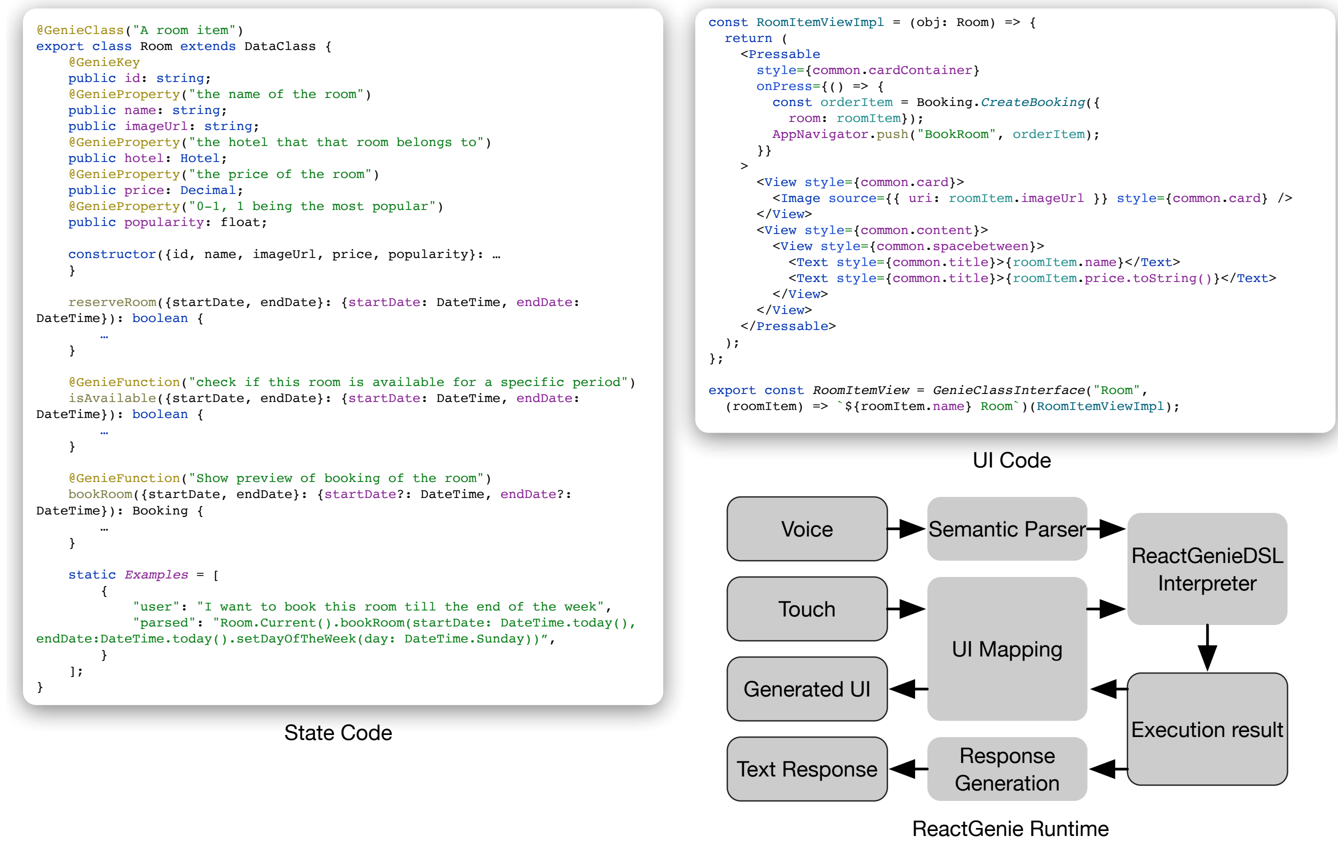


Figure 3: ReactGenie [68] System Overview: GenieWizard is built upon the state-of-the-art multimodal app implementation framework ReactGenie. ReactGenie offers a basis for implementing multimodal apps when the required functions are known, so with GenieWizard we focus on detecting missing functions and generating suggestions.

3.3 Overview of GenieWizard

GenieWizard assists in multimodal feature discovery by bridging the gap between the design and implementation of multimodal apps. This tool *simulates* user testing in the early implementation phases so that the developers can improve their implementation rapidly without going through the full software engineering life cycle. To achieve this, GenieWizard decomposes the problem into these stages.

(1) **Simulate User Interactions**: Simulate the user's multimodal interactions with the app based on the early version of the state code (e.g., an incomplete implementation, without few-shot examples).
(2) **Speculate on Missing Features from Interactions**: Analyze the generated user interactions and speculate on the missing features in the limited state code.
(3) **Suggest Features to Implement During Development**: To help the app developers using the GenieWizard system, GenieWizard suggests features and keeps track of the progress of the implementation for the developers in a convenient way.

3.4 Simulate User Interactions

Prior work on app development feedback typically uses a set of heuristic rules to provide suggestions for the designers and the developers [1] and/or retrieving similar designs to help developers make their design [1, 12]. However, given that there are not many existing multimodal apps, it is hard to build a set of heuristic rules or retrieve similar designs. GenieWizard takes a different approach. We simulate user interactions with the app using LLMs, similar to prior work on using LLMs to simulate social behaviors [1, 52].

The goal is to simulate diverse user behaviors that invoke unimplemented actions related to the app. The user interaction simulation is listed in the four blocks on the left of Figure 2.

First, we prompt the LLM with the extracted function signatures in the app's state code to generate a brief one-sentence description of the app. This description is used to confine the generated interactions to be related to the app but not specific to the existing implementation. The description can be further edited if the developer intends for a specific use case.

Second, to create diverse behaviors, we create a set of personas that represent different types of users. Targeting the US demographics, we created a system that can automatically generate profiles

representative of the occupation, gender, age, and name distributions of a US population from data collected from the US Census, Social Security Administration, and Bureau of Labor Statistics. To guide the persona's behavior, GenieWizard also prompts the LLM to imagine a scenario for each profile, including a specific task related to the app description that a user fitting that profile may want to do.

Third, to simulate real-world usage, we prompt the LLM for an interaction dialog between the user and the app using the persona, scenario, and app description. We prompt the model to simulate the app by providing a textual response and a description of what is being rendered on the screen to simulate the feedback from an actual multimodal interaction process.

Lastly, we extract the simulated user utterances from the resulting dialog and use it as a set of user multimodal commands that should be supported by the app.

3.5 Speculate on Missing Features from Interactions

GenieWizard's second stage is to derive the features needed to support the simulated user multimodal commands. Note that we do not simply map each user's command into a single API call. The expressiveness of ReactGenieDSL can support many possible commands with a set of given fields and APIs. For example, it has sorting and filtering functions, and combinations thereof at its disposal. Therefore, given the location and price for each hotel, ReactGenieDSL can generate code to find the cheapest hotel in a location, or the closest hotel within a price range. Figuring out the features to add is thus nontrivial, and cannot be handled solely by an LLM.

3.5.1 Speculative parsing. To tackle this problem, we use an LLM-based neural semantic parser to translate user requests into ReactGenieDSL code. We prompt the parser to speculate and use new APIs if none of the provided APIs suffice. We refer to this technique as *speculative parsing*. All speculated APIs are potential features to implement.

3.5.2 Zero-shot neural semantic parser. The semantic parser in ReactGenie requires the developer to provide some few-shot example pairs of user conversation and corresponding ReactGenieDSL. Not only is this tedious, but these examples must be updated whenever the code is changed, and it is not possible to supply these descriptions for newly speculated APIs. Thus, it is desirable to create a *zero-shot* parser, which requires *no examples* of how each API is to be used.

Through experimentation with prompts, we observed a zero-shot parser already performs quite well because of the familiar syntax of ReactGenieDSL and the presence of the function signatures of available functions in the prompt. However, there are problems with some unique design choices in ReactGenieDSL that caused some syntax errors, including the removal lambda expressions [22] and pervasive use of method chaining [27] (see the details in the ReactGenie paper [68]).

Our solution is to teach the LLM-based semantic parser the unusual syntactic design of ReactGenieDSL using a small app. GenieWizard provides, as a fixed preamble to the LLM prompt, a small predefined app's declaration of functions, few-shot examples of that app, and some tips for generating correct ReactGenieDSL code. The prompt then includes function declarations extracted from the developer's app without any app-specific examples. We evaluated the accuracy of this zero-shot parser in Section 5.2.

3.5.3 Feature identification. As shown in the center two blocks in Figure 2, our next step is to analyze the parsed ReactGenieDSL code, which may contain unimplemented functions, to understand what specific features are missing. Similar to in previous sections, the feature we mean here is a specific class, function, or property that the developer can add to support the unsupported command. We want to do this in a way that does not require the developer to provide a fully working implementation of the app, so we cannot call any of the developer's functions to get a result.

We use the concept of abstract interpretation [21] in program analysis to identify the missing feature. The idea is to execute the code abstractly by just computing the types in the program. Because ReactGenieDSL is strongly typed, we can abstract out the implementation of a function by using its signature. We refer to the process of using abstract interpretation to identify missing functions as a "dry run." We built a version of the ReactGenieDSL interpreter that "dry runs" the generated commands and outputs the first encountered classes, functions, or properties the developer has not yet declared. Using the ReactGenieDSL code `Room.All().price` as an example, GenieWizard's dry run module will first find the class name `Room` and its properties and functions. Then, it will find the function `Room.All`. Rather than calling the function that may not be fully implemented yet, it will directly check the return value type, which is `Room[]`. Finally, it will try to find the property price in the room class. In this case, if the property is not found, it will output a dry run error along with the missing app feature (class/property/function) mentioned in the app.

Note that once the parser reaches the first unimplemented feature, it cannot speculate any further because the system does not have the missing feature's type information. We found in our testing that the majority of unsupported commands only require one missing feature.

3.6 Suggest Features to Implement During Development

We discovered through experimentation that the user simulation can often generate around a hundred unsupported commands per app. If this directly translates to an equal number of missing features, it would be overwhelming to show all of them to the developer.

One naive approach is to group the same missing classes/properties/functions together by name and only show one of them to the developers. However, under different generated commands, the same feature may be implemented using different functions (e.g., hotel's review score can be implemented as `Hotel.rating` or `Hotel.reviewScore`). In some cases, different features may be speculatively parsed into the same function (e.g., the user's user ID and the user's identification document can be parsed to `User.ID`). These are caused by the fact that the missing classes/properties/functions names alone are not enough to represent what the feature is for.

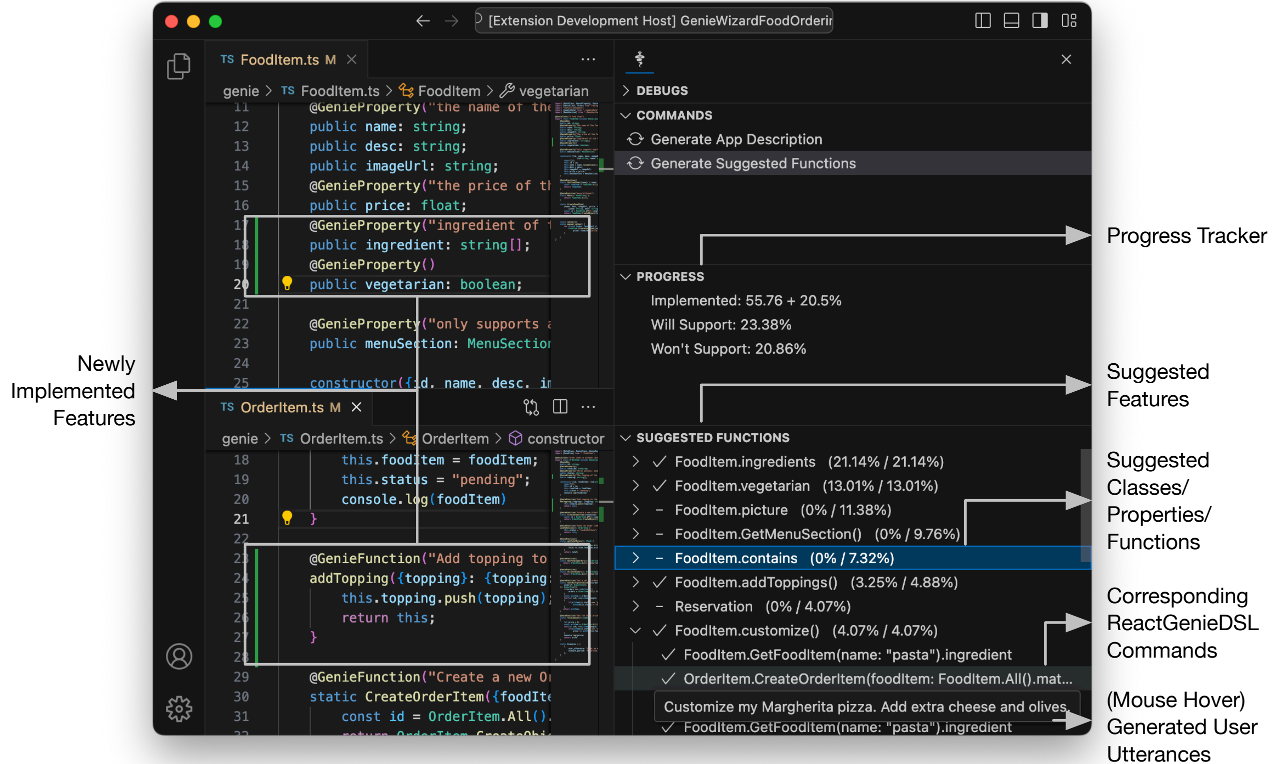


Figure 4: GenieWizard IDE Plugin: The GenieWizard IDE plugin provides developers with feature suggestions in the same place where they are writing code. It can provide suggestions of potential unsupported features through a list of suggested functions (unique features). Under each suggested function, developers can see the generated parsed commands in the same group. By hovering the mouse over a parsed command, developers can see the original generated user utterance to better contextualize the feature they are implementing. They can set goals by clicking on each unique feature, and a checkmark will appear next to it. While working on the implementation, checkmarks will appear next to the parsed command to indicate a generated user utterance that is now supported. The IDE also renders a progress tracker showing the percentage of commands implemented, the percentage of commands marked as will support, and the percentage of unmarked commands.

To solve the problems with this naive approach, we first provide the originally generated command, the parsed ReactGenieDSL code, and the missing feature to the LLM and ask it to generate a brief description of the missing feature in the form of a comment (`User.ID // The user's identification document`). This process is illustrated in the rightmost block in Figure 2. We then cluster the generated commands using agglomerative clustering with the cosine distance between the embedding vector of the feature and description to form a list of clusters of unique features [24] (shown in Figure 4). We show each cluster's representative feature to the developer and list every ReactGenieDSL statement in that category below it.

To provide better context, the developer can hover above a ReactGenieDSL statement to see the originally generated user utterance that parsed to this statement. To further help the developer keep track of the implementation progress, we also provide a progress tracker that shows the percentage of the missing features that have been implemented, along with which commands are implemented and which are not. Developers can click on the features they would like to implement, and the plugin will automatically re-parse the generated user utterance under that category when it detects a function signature change in the state code. It will keep track of the developer's progress and show their live progress in the IDE interface.

4 Implementation

We built a Python Flask server that handles every part of the GenieWizard pipeline except for the speculative parser, which has to be implemented using a TypeScript environment where the developer's state code is written. We used OpenAI GPT-3.5-turbo for everything to save on inference costs and increase responsiveness except for the app description generation, for which we used GPT-4 because we found GPT-3.5-turbo frequently generates irrelevant

app descriptions. We chose to generate 40 personas + scenario combinations and asked the model to produce 9-18 conversation turns for each. These parameters were chosen based on empirical results to establish a balance between speed and suggestion quality, and the resulting performance was systematically evaluated in Section 5.3. The model does not always follow the conversation-turn requirements we gave. After filtering out out-of-scope commands, such as "open the app", using empirically-designed regular expressions, the system generates 410 potential user commands every time from the 40 personas + scenarios. Each time the developer requests a new feature discovery process, the entire utterance generation task costs around 0.03 USD to run, the missing feature speculation task costs around 2.5 USD[5], and the clustering task costs 0.01 USD. This cost is relatively low compared to the labor cost[6], so many developers can adopt it from a financial perspective.

Separately, we implemented a VS Code Plugin as the developer's user interface for the GenieWizard IDE in TypeScript.

5 Evaluation

We conducted evaluations on three major parts of the GenieWizard system: 1) the zero-shot parser, 2) the utterance generation pipeline, and 3) the suggestion generation and IDE user interfaces. To facilitate the evaluation, we also built two example multimodal apps for developers to improve upon in a user study.

5.1 Example Apps for the Evaluation

We built two example apps for the evaluation: a food menu ordering app and a hotel booking app. The food menu app is a web app that allows users to browse food options, place an order, and keep track of the food's progress until it reaches the customer's table. The hotel booking app is a web app that allows users to book hotels by selecting from/to dates and the number of guests, viewing availability, and creating bookings. We built both apps with a simple GUI implementation with functions for rendering content on the GUI with the ReactGenie framework (see Figure 5). These apps have a basic state code implementation to represent early versions of commercial apps.

We conducted an IRB-approved crowd-based elicitation study similar to that described in our previous work [68] to create a gold standard for missing app features. We showed screenshots of the app being tested and asked people to demonstrate how they would interact with the app multimodally (see Figure 5). We improved upon our previous work's methodology by making the interface accept voice and touch interaction inputs instead of using text to simulate voice input. This is because, in our pilot study, we found that text-based input frequently limited user interactions to relatively short and simple voice commands due to the high cost of typing, while using voice recognition better simulates a real-world multimodal interaction environment.

The two apps we built represent common app categories that people frequently use (e.g., food menu apps — Toast/Yelp, and hotel booking apps — Hotels.com). We try to use simple screenshots to elicit questions based on participants' experiences with apps in the same category. For example, many people asked about room amenities even if they were not present in our test app screenshots.

We recruited 40 participants (20 male and 20 female) on the research recruitment platform Prolific for each app and filtered out attention check failures and some entries where people misunderstood the task (several participants treated the interface as a regular GUI app and the voice interface as a feedback recorder). Each survey submission took around 7 minutes to complete, and we paid each participant 1.4 USD. We gathered 298 user utterances for the hotel booking app and 367 for the food ordering app.

5.2 Zero-Shot Parser Performance

The goal of the zero-shot parser is to parse generated and end-user commands to generate the correct ReactGenieDSL code and speculate on the missing features (classes/properties/functions) for unsupported commands, all *without* requiring the developer to provide example user command-ReactGenieDSL pairs. In this way, the zero-shot parser can be used both as a component in the app used by end-users and as part of GenieWizard's pipeline. In this section, we will measure the accuracy of a few variants of the zero-shot parser we created and compare them against the few-shot parser from our prior work [68].

5.2.1 Measures. We would like to know what percentage of the parsed DSL from each parser is semantically and syntactically correct. Syntactically, the DSL is checked using the ReactGenieDSL interpreter's syntax check. Semantically, for supported commands, the DSL has to use the right functions and achieve what the user wants. For unsupported commands, the function speculated by the parser has to fulfill the user's request, and assuming that the speculated function exists, it has to achieve what the user wants.

Notably, there may be different ways to satisfy the user's request, and we would consider all of them correct. For example, if the user asks "Recommend me a main dish.", that can be translated to either `FoodItem.All().matching(field: .menuSection, value: MenuSection.GetMenuSection(name: "main dish")).sort(field: .price, ascending: true)[0]` or `MenuSection.GetMenuSection(name: "main dish").getFoodItems().sort(field: .rating, ascending: false)[0]`. Although the former and the latter use different functions to retrieve the main dish, they are both correct. As for the recommendation, the former recommends a low-price option, and the latter recommends a high-rating option, both of which fulfill the user's desire.

5.2.2 Procedure. We created three variants of our zero-shot parser with GPT-4, GPT-3.5, and Codex[7]. We used ReactGenie's few-shot parser[8] built with Codex as a baseline for comparison. We randomly sampled 50 utterances from the user commands elicited for the food menu and hotel reservation apps, giving 100 utterances in total.

[5]This can be further reduced to approximately 0.8 USD by using the latest GPT-4o mini model.

[6]The US country-wise minimal wage at the time of writing is 7.25 USD.

[7]OpenAI Codex model (`code-davanci-002`) has about a one-call-per-second rate limit on OpenAI's endpoints. Azure OpenAI API has much higher rate limits, but their Codex model is 20-30 times as expensive as GPT-3.5 and 2-3 times as expensive as GPT-4, making the cost prohibitive. This makes the Codex model infeasible for testing several hundred generated utterances.

[8]For the main study, our app was developed without the few-shot examples. To test the few-shot Codex parser, we temporarily added seven examples for each app to produce the few-shot parser results for comparison.

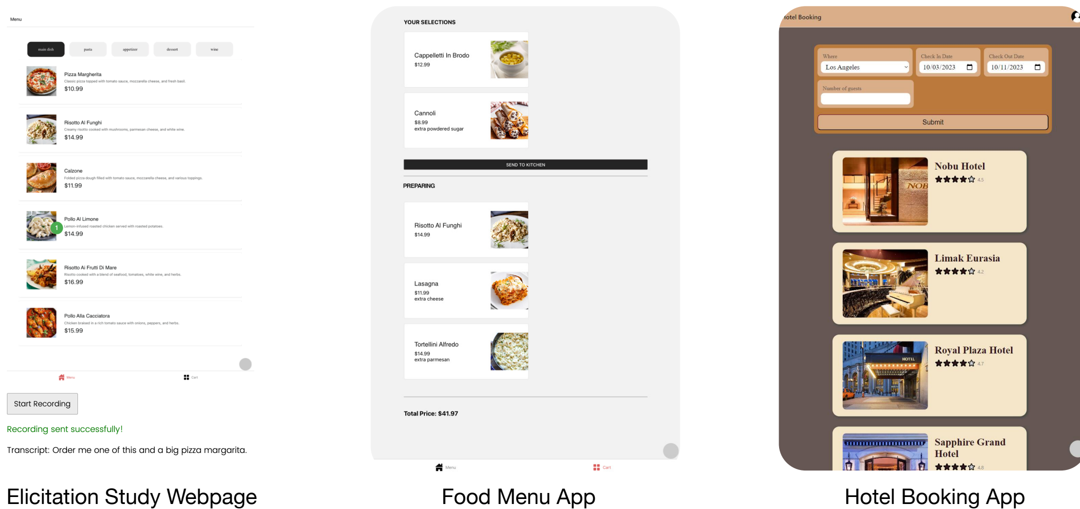


Figure 5: GenieWizard Example Apps and Elicitation Study Interface. We built two example apps for developers to improve upon: a food menu ordering app and a hotel booking app. We conducted an elicitation study with a multimodal command collection interface to collect possible real user commands for both apps.

Category	Zero-shot GPT-4	Zero-shot GPT-3.5	Zero-shot Codex	Few-shot Codex
Total (100)	77% (77)	69% (69)	80% (80)	72% (72)
Supported (64)	90% (56)	84% (52)	90% (56)	89% (55)
Unsupported (36)	55% (21)	45% (17)	63% (24)	45% (17)

Table 1: Parser Accuracy: We compared GenieWizard's zero-shot parser implemented on GPT-4, GPT-3.5, and OpenAI Codex (expensive and slow) with the few-shot Codex parser implemented in ReactGenie [68]. The results showed that the zero-shot GPT-4-based and zero-shot GPT-3.5-based parsers perform similarly to the few-shot Codex-based parser on supported commands. On unsupported commands, all GenieWizard models match or surpass the few-shot Codex model.

5.2.3 Results. As shown in Table 1, our zero-shot parser demonstrated an accuracy similar to that of a few-shot Codex parser (90%) on supported commands (64% of total commands), indicating that our new zero-shot prompt is effective at understanding users' requests while reducing development costs. For unsupported commands (36% of total commands), all three zero-shot models have superior or similar performance compared to the few-shot parser. This is likely because we actively encouraged missing feature generation (speculative parsing) in the newly designed zero-shot prompt. Moreover, all parsers performed worse on unsupported commands than on supported commands. This is expected since the parser must consider the missing feature while parsing the user input.

Among the zero-shot parsers, the GPT-3.5-based parser has much lower costs, lower latency, and reasonable accuracy, so we selected that as the model to use in the final tool.

In the following evaluations, we need to find unsupported commands in elicited user commands. The parser can be actually used as a filter for possible unsupported commands. If the parser generates a syntax error or there is no syntax error but it refers to a non-existing class/property/function (discovered through the dry run), we treat it as potentially having unsupported commands, and we can then manually inspect it. Due to Codex's high price and low throughput, we used zero-shot GPT-4 as the model to do the initial filter. We evaluated the accuracy of this method as the indicator of potential unsupported commands. This method achieved 81% precision and 68% recall.

5.3 Generation Pipeline Performance

The goal of the utterance generation pipeline is to generate feature suggestions that can cover a broader range of user-desired commands that are currently unsupported. We would like to know what proportion of real users' unsupported command types we can discover using our pipeline.

5.3.1 Measures. To evaluate the utterance generation pipeline, we would like to know the percentage of real users' unsupported commands that can be covered by GenieWizard's pipeline versus simply prompting GPT-4 to generate feature suggestions using the same code skeleton and app description as context.

Specifically, we would like to extract unsupported commands in the real-user datasets we collected from the elicitation study described in Section 5.1, and see if these commands can be covered by commands generated by the two different systems (GenieWizard or prompting GPT-4). We define distinct groups of commands as commands that can mutually be supported by the app by adding a single class/property/function. We consider a group of real users'

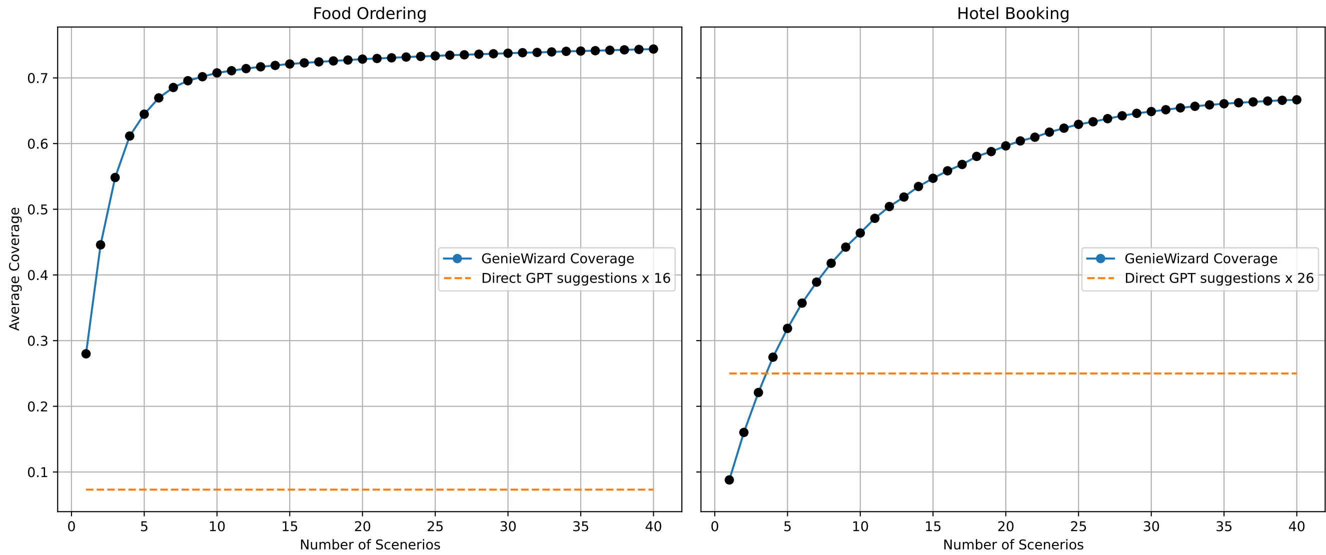


Figure 6: GenieWizard Pipeline Performance: GenieWizard automatically generated commands have high coverage over real user-elicited commands. GenieWizard's commands can cover 74% and 67% of the user-elicited commands of the food ordering and hotel booking app, respectively. Prompting GPT-4 to generate the same number of suggestions can only generate 7% and 25% of the elicited commands.

commands covered if a system-generated suggested command also belongs to the group. The higher the coverage, the better the system is given the same number of suggestions/groups.

5.3.2 Procedure. To extract real users' unsupported commands, we used the aforementioned GenieWizard's GPT-4-based parser + dry run filter to retrieve possible unsupported user commands. We removed the supported ones and were left with 82 user-elicited commands from the food ordering app and 60 from the hotel booking app. We further labeled them collaboratively to reveal different missing feature groups (i.e., each group could be implemented with the same class/property/function). We found 19 groups of commands for the food ordering app and 27 for the hotel booking app.

For GenieWizard's generation pipeline, we generated user utterances with 40 different persona and scenario combos. We extracted 117 generated commands for the food ordering app and 103 for the hotel booking app. We then assigned generated commands that belong to an existing group of user-expected commands to that group and created new groups for commands that do not belong in an existing group. In the end, we got 16 and 26 groups of generated commands for the two apps, respectively. Among them, 8 and 11 groups fall within the same groups as in the user-elicited commands. Note that these groups have an uneven distribution of commands, meaning that a few groups have a lot of commands while many other groups have very few commands. As a result, the coverage of GenieWizard for the food ordering app and the hotel booking app are 74% and 67%, respectively.

As a comparison, we asked GPT-4 to also generate 16 and 26 features to add for both apps. Note that we limit the features to the same number of groups that the GenieWizard pipeline generates because each suggestion of GPT-4 is always in its unique group. In this way, we are simulating the case where the developers received the same number of suggestions directly from the LLM as in the GenieWizard condition. The GPT-4-based suggestion only covers 7% and 25% of the user-elicited commands for the two apps.

To further evaluate the system's performance when fewer persona and scenario sets are used to save compute, we conducted an additional experiment where we sampled randomly drawing a subset of different numbers from the 40 persona + scenario combinations and measured how much coverage of the total number of unsupported commands we can achieve with a subset of what we have generated. As shown in Figure 6, GenieWizard easily beat direct GPT suggestions starting at only four persona + scenario sets. This indicates the advantage of GenieWizard is that we can better support the developer in bridging the gap between the supported commands and the user's desired intention space.

5.4 Developer Experience with GenieWizard Plugin

To evaluate the efficacy of GenieWizard, we conducted an IRB-approved user study asking developers ($N = 12$) to identify missing functions of the two example apps (see Figure 5) with the regular VS Code IDE[9] as a baseline and compared it against what they identified as missing when using our GenieWizard IDE.

5.4.1 Study Design. The study was facilitated using a remote desktop to ensure all participants completed the coding tasks in the same environment. The experimenter first introduced the study goals and explained study-related concepts such as multimodal

[9]Both have GitHub Copilot enabled to save the developer's time.

apps since most participants did not have multimodal development experience.

As our participants are expected to modify ReactGenie apps during the main tasks, we first provided a short tutorial to demonstrate how to use ReactGenie by building a simple counter app. This task helped familiarize our participants with the basic architecture of a ReactGenie app, including state code, UI code, and code that links UI and state code. In the main tasks, they mostly needed to add functions to the state code, through doing so requires a more holistic understanding of how the entire system works. The tutorial took about 30 minutes to complete.

The main study process contains two multimodal app improvement tasks with the goal of "optimizing these two multimodal applications using different methods to support as many multimodal interactions as possible for the users" within a limited amount of time (30 minutes for each app). In the two tasks, the developer used a different tool (VS Code vs. GenieWizard IDE) on different apps. We used a counterbalanced design to control learning effects due to the order of the tools and the combinations of tools and apps.

Finally, participants were required to complete a post-study survey that included their demographic information, and they were asked to fill out the SUS usability scale [13] and the NASA-TLX cognitive load scale [30]. We recorded audio and the shared screen during the entire process with the participant's permission. Each participant received a $50 gift card as compensation after the study.

5.4.2 Participants. We recruited 12 React developer participants (8 males) by distributing the recruitment link using a convenience sample. Our participants include student developers and professional developers with an average age of 24.17 (σ = 2.37). Three participants had TypeScript development experience, and one developer had experience with ReactGenie.

5.4.3 Results. We labeled developers' implemented features into the same user-elicited command groups as above. As shown in Table 2, the GenieWizard conditions resulted in a significantly larger increase in the supported command coverage than with the baseline condition (paired t-test, $t = 6.077, p < 0.001$). The unsupported command reduction averaged 42%. The statistics in Section 5.2 showed that, from the randomly sampled 100 commands, our initial app prototypes provided to the developers support 64% of the user's commands (36% unsupported). Therefore, combining these two statistics, the unsupported commands can be reduced by $36\% \times 42\% \approx 15\%$. In other words, developers with GenieWizard can potentially increase the supported command percentage from 64% (about one in three user commands are unsupported) to 79% (about one in five user commands are unsupported), which should be noticeable from a user experience perspective. When looking at the individual developer's performance when using GenieWizard, only P7's unsupported command percentage did not decrease. We observed that this participant did not follow the suggestions provided by GenieWizard but instead optimized both applications according to their own ideas. The participant stated that although GenieWizard provided good advice, they preferred to explore the potential features of the application on their own.

GenieWizard also showed a higher level of usability and a lower participant burden than the baseline. The participants' mean SUS score (see Figure 7, middle) for the GenieWizard plugin is 77.5 (σ = 8.52). For the baseline, the mean score is 41.46 (σ = 18.26). GenieWizard's SUS score is significantly higher than the baseline (paired t-test, $p < 0.001, t = 6.292$), showing its better usability for this task. The average NASA-TLX (see Figure 7, left) score from participants with the GenieWizard plugin is 29.53 (σ = 12.24). With the baseline, the score is 60.49 (σ = 13.27). The score of GenieWizard is significantly lower than the baseline (paired t-test, $p < 0.001, t = -5.145$).

Participant ID	Baseline	GenieWizard
0	[Food] 19/233 (08%)	[Hotel] 086/163 (53%)
1	[Hotel] 31/163 (19%)	[Food] 125/233 (54%)
2	[Food] 07/233 (03%)	[Hotel] 051/163 (31%)
3	[Hotel] 22/163 (13%)	[Food] 113/233 (49%)
4	[Food] 00/233 (00%)	[Hotel] 074/163 (45%)
5	[Hotel] 11/163 (07%)	[Food] 081/233 (35%)
6	[Food] 12/233 (05%)	[Hotel] 067/163 (41%)
7	[Hotel] 25/163 (15%)	[Food] 000/233 (00%)
8	[Food] 30/233 (13%)	[Hotel] 048/163 (29%)
9	[Hotel] 18/163 (11%)	[Food] 155/233 (67%)
10	[Food] 33/233 (14%)	[Hotel] 070/163 (43%)
11	[Hotel] 19/163 (12%)	[Food] 137/233 (59%)

Table 2: Comparison of Baseline and GenieWizard on Helping Developers Address Unsupported Features: The results showed that GenieWizard, on average, can help developers implement 42% of the unsupported actions compared to a baseline IDE solution of 10%.

We used a seven-point Likert scale questionnaire to evaluate whether the participants would use these tools in real life[10] (see Figure 7, right). The median Likert-scale rating (1-strongly disagree, 7-strongly agree) for GenieWizard and the baseline methods are 7.0 and 3.0, respectively. We found a statistically significantly higher rating for GenieWizard than the baseline method ($W = 0.000, p = 0.003$).

All participants expressed that their greatest difficulty in using the baseline was not knowing what potential needs the end-users may have and not knowing which features should be supported. They hoped to have actual test cases to assist them in improving the application. One participant even chooses to refer to other apps to get inspiration. The participants agreed that GenieWizard can help them improve their application quickly and efficiently. While using GenieWizard, participant 1 said, *"It provides me with various insights from other users and then prevents*[11] *me from exhaustive searching of possible user demands."*

However, some participants also found some parts of the interactions simulated by GenieWizard to be difficult to understand. For example, in one study session, a generated suggestion asks for a `specialRequest` function for a food item. The participant in that session found it hard to understand the requirements associated with this suggestion. Furthermore, one participant wished that GenieWizard not only provided suggestions but also helped them implement some of the functions.

[10]The prompt is "Would you consider using this method to build multimodal application in real life?"
[11]Note by authors: the participant probably meant "alleviates".

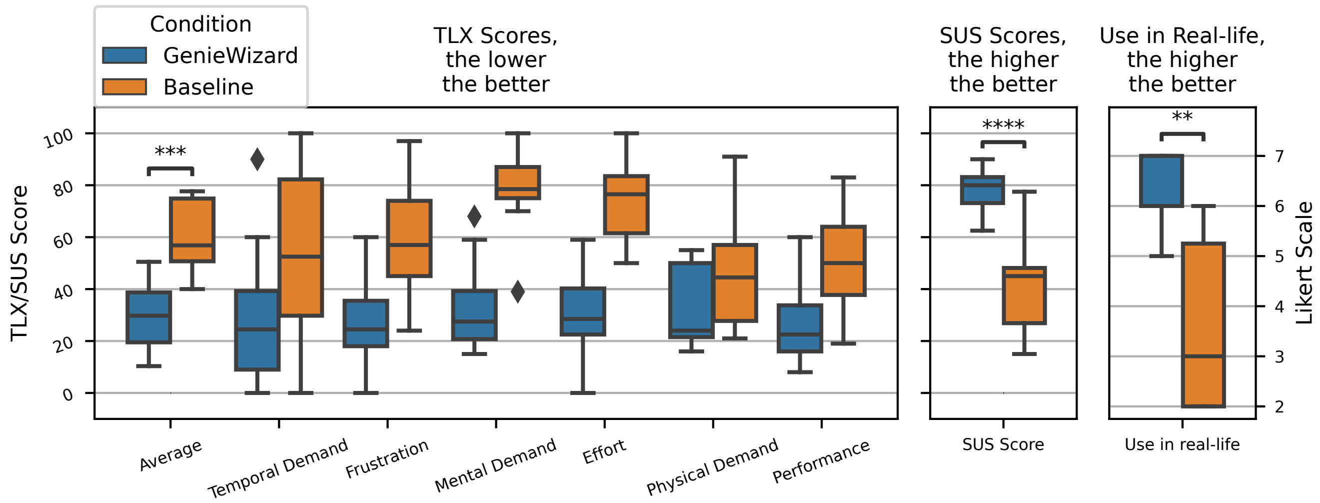


Figure 7: Comparing mental load, usability, and participant preference of developing with GenieWizard compared to without GenieWizard. An $N = 12$ study has shown that GenieWizard induced less mental load ($p < 0.001, t = -5.145$). It has also shown GenieWizard is more usable for improving a multimodal app ($p < 0.001, t = 6.292$). Participants also expressed willingness to use GenieWizard in their real life. **: p < 0.01 ***: p < 0.001 ****: p < 0.0001

6 Discussion

The development of multimodal applications presents unique challenges due to the vast interaction space and the difficulty in anticipating user behaviors. Our work with GenieWizard demonstrates the potential of AI-powered tools to address these challenges and support developers throughout the implementation process. In this section, we discuss our rationale for why we chose an API-level suggestion scheme. Then, we discuss the implications of our findings, contextualize GenieWizard within the broader landscape of AI-assisted development tools for multimodal apps, and explore the limitations and future directions of this approach. We begin by examining how GenieWizard leverages AI to bridge the gap between user expectations and developer implementations, then consider the limitations of our current approach, and finally propose avenues for future research and development in this rapidly evolving field.

6.1 API Level Suggestions vs. Concept Level Suggestions

GenieWizard suggest features to developers in the format of APIs to implement. An alternative method could be providing high-level features (concepts) directly based on synthesized user interactions. However, this alternative method will not work well for multimodal interactions.

For multimodal interactions, there is not a one-to-one mapping from user requests to app feature implementations (see Section 1.1). The alternative method will not work because, multimodal apps are unlike traditional GUI apps, where features are usually user behaviors that the app supports through a series of controls. In multimodal interactions, the developer's code is composed together to support a great variety of possible multimodal user requests, and features can only be clearly defined at a lower level, i.e., function level.

A typical user request, function call, and app feature look like the following:

User request: What is the cheapest main dish that I have ordered from this restaurant before?
Function calls:
```
Order.All().filter(field: .restaurant, object: Restaurant.Current()).mainDishes.sorted(field: .price, ascending: true)[0]
```
Feature missing: `Order.mainDishes`

User requests are supported by composing different app features together through a series of function calls. To understand whether a user request is supported, we have to decompose the user request down to the function call level to see if any functions are missing. At this point, proposing features as concepts vs features as functional calls are equivalent. In fact, the GenieWizard proposed function calls are only helping developers keep track of which user requests can be supported by their implementation, even if they implement them differently. For example, the above missing features can also be implemented by adding a field to the `Food` class called `isMainDish`, and GenieWizard can also keep track of the implementation progress accordingly.

6.2 AI-powered Tools for Multimodal App Development

The adoption of multimodal apps has been limited by the prohibitive development efforts required to support the potentially exponential number of commands users can ask. In other words, developers need to write all the functions separately to support all the possible user queries. Our prior research [68] alleviated this problem by using LLMs to compose different features built by developers to support users' specific commands. However, the multimodal interface means that users may expect features not originally built into

the app, resulting in 41% of unsupported commands in ReactGenie's evaluations [68].

GenieWizard tries to address this unsupported command problem by using AI-powered tools. We have demonstrated that developers may only implement 10% of missing functions through their common sense reasoning (Section 5.4). We also found that simply prompting GPT-4 for feature suggestions can only cover 16% of missing functions. Therefore, we need a better pipeline to simulate user behavior and app responses, and give concrete, actionable suggestions.

We built GenieWizard's pipeline to start with the developer's program skeleton to support feature discovery early in the implementation phase. GenieWizard then generates possible user interactions by sampling from user personas and simulating app behavior through a text-based conversation using LLMs. GenieWizard further identifies missing features in the generated commands and clusters them to form actionable suggestions for developers. We found GenieWizard's suggestions can cover 71% of the missing features in real user commands (Section 5.3).

To make developers' lives easier, we present these suggested features in an IDE plugin so that developers can see these suggestions right where they write code. GenieWizard also presents relevant user commands and parsed ReactGenieDSL lines to demonstrate how these suggested features may be used. In addition, GenieWizard's IDE plugin can show the percentage of generated commands that are currently supported while the developer is changing their code, giving them a more direct feeling of the progress they have made. Overall, we demonstrated in Section 5.4 that developers with GenieWizard can implement 40% of missing features in actual users' commands, which is a huge improvement over the 10% when completing the same task without GenieWizard.

GenieWizard shows great promise for AI to help in the development of multimodal apps. In addition to helping the *program* better understand the user at *run time*, it can also help the *programmer* better understand and support users at *development time*.

6.3 Limitations

One limitation of our evaluation is that the grouping process is ambiguous in nature. For example, the customization of "crust" and the customization of "toppings" can be implemented as two separate functions (`FoodItem.setCrust()` and `FoodItem.setTopping()`) or a single function (`FoodItem.addCustomizations()`). We used our experience in engineering apps with ReactGenie to select the best implementation path for the grouping process.

Our pipeline generates personas from a general US population distribution, and our survey is also limited to US populations. Within the scope of the paper, we cannot be sure whether this finding of high alignment between generated personas and target population can be generalized to other sub-populations.

Another concern is the potential bias in LLMs [61] may skew the generated commands and, therefore, steer the development of the next generation of multimodal apps. In an early pipeline version, we tried to ask the LLM to directly generate personas and scenarios. However, we discovered that the LLM generated many male persona named "James." To mitigate this issue, we wrote a module to sample user profiles according to a US representative population. There may be more bias that is yet undiscovered in GenieWizard, so more caution needs to be taken when using developer tools such as GenieWizard.

6.4 Future Work

This paper demonstrated GenieWizard as a tool to improve multimodal apps. Recently, the voice assistant industry has also moved towards using LLMs and API calling to implement voice chatbots. Future work can investigate a similar user interaction generation pipeline that may also be able to help chatbot developers implement more APIs to improve chatbot user experiences.

Currently, we only tested using GenieWizard to help with the first round in the app's development lifecycle. We have not yet tested the tool when an app has gone through a round of actual user feedback and redesign and whether GenieWizard can still provide benefits to the developer. Once there is an implemented version of the app, future iterations with GenieWizard can also consider how to take an already available UI into consideration to generate more representative user utterances.

7 Conclusion

Multimodal interactions allow users to utilize an app in more flexible and efficient ways. However, this also raises the bar for developers to implement usable multimodal apps, as the space of user actions can significantly expand now that the GUI does not constrain user actions. If the required functions are unimplemented, the user experience may suffer. In this research, we aim to bridge this gap by introducing a novel developer tool, GenieWizard. This tool provides early feedback on the missing functions of a multimodal app, enabling developers to receive auto-generated suggestions that can cover a majority of user actions, even with just a basic skeleton version of the app. The GenieWizard pipeline consists of the following sequential components. First, GenieWizard generates user personas and simulates user actions (commands) based on these personas. This helps bootstrap the design feedback process in the absence of established design principles for multimodal apps. Then, our zero-shot parser parses the simulated actions into DSL code without any developer input. Finally, our dry run module can automatically test the generated actions and commands and offer suggestions about unsupported functions. Our evaluation has demonstrated the strong performance of the different components of the pipeline, as well as the end-to-end impact. This is best represented by the significant increase in the number of missing functions developers can identify and fix during a short lab study session, showing that GenieWizard is both effective and usable.

Acknowledgments

We would like to thank the reviewers for their insightful feedback and the participants of our user studies for their invaluable input. We also want to acknowledge Meta Platforms, Inc., the Verdant Foundation, and Stanford HAI for their generous financial support. We are also grateful to Microsoft for providing Azure AI credits, which have been instrumental in advancing our research.

References

[1] Gati Aher, Rosa I. Arriaga, and Adam Tauman Kalai. 2023. Using Large Language Models to Simulate Multiple Humans and Replicate Human Subject Studies. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 17, 35 pages.

[2] Adept AI. 2024. Adept: AI That Powers the Workforce. https://www.adept.ai/. Accessed: 2024-08-23.

[3] Intakhab Alam, Nadeem Sarwar, and Iram Noreen. 2022. Statistical analysis of software development models by six-pointed star framework. *PLOS ONE* 17, 4 (Apr 2022), e0264420. doi:10.1371/journal.pone.0264420

[4] Adel Alshamrani and Abdullah Bahattab. 2015. A comparison between three SDLC models: waterfall model, spiral model, and Incremental/Iterative model. *International Journal of Computer Science Issues (IJCSI)* 12, 1 (2015), 106.

[5] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27. doi:10.1145/3360585

[6] D. Baumer, W. Bischofberger, H. Lichter, and H. Zullighoven. 1996. User interface prototyping-concepts, tools, and experience. In *Proceedings of IEEE 18th International Conference on Software Engineering*. 532–541. doi:10.1109/ICSE.1996.493447

[7] Michel Beaudouin-Lafon and Wendy E Mackay. 2007. Prototyping tools and techniques. In *The human-computer interaction handbook*. CRC Press, 1043–1066.

[8] Karim Benharrak, Tim Zindulka, Florian Lehmann, Hendrik Heuer, and Daniel Buschek. 2024. Writer-Defined AI Personas for On-Demand Feedback Generation. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 1049, 18 pages. doi:10.1145/3613904.3642406

[9] Barry W. Boehm. 1988. A spiral model of software development and enhancement. *Computer* 21, 5 (1988), 61–72.

[10] Niall Bolger, Angelina Davis, and Eshkol Rafaeli. 2003. Diary methods: Capturing life as it is lived. *Annual review of psychology* 54, 1 (2003), 579–616.

[11] Marie-Luce Bourguet. 2003. Designing and Prototyping Multimodal Commands.. In *Interact*, Vol. 3. Citeseer, 717–720.

[12] Paul Brie, Nicolas Burny, Arthur Sluÿters, and Jean Vanderdonckt. 2023. Evaluating a Large Language Model on Searching for GUI Layouts. *Proceedings of the ACM on Human-Computer Interaction* 7, EICS (2023), 1–37. doi:10.1145/3593230

[13] John Brooke. 1996. SUS: A 'Quick and Dirty' Usability Scale. In *Usability Evaluation In Industry*. CRC Press, 207–212. doi:10.1201/9781498710411-35

[14] Reinhard Budde, Karlheinz Kautz, Karin Kuhlenkamp, and Heinz Züllighoven. 1990. What is prototyping? *Information Technology & People* 6, 2/3 (1990), 89–95.

[15] W. Buxton, M. R. Lamb, D. Sherman, and K. C. Smith. 1983. Towards a comprehensive user interface management system. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques* (Detroit, Michigan, USA) *(SIGGRAPH '83)*. Association for Computing Machinery, New York, NY, USA, 35–42. doi:10.1145/800059.801130

[16] Bradley Camburn, Vimal Viswanathan, Julie Linsey, David Anderson, Daniel Jensen, Richard Crawford, Kevin Otto, and Kristin Wood. 2017. Design prototyping methods: State of the art in strategies, techniques, and guidelines. *Design Science* 3 (08 2017). doi:10.1017/dsj.2017.10

[17] Stuart K. Card, Thomas P. Moran, and Allen Newell. 1980. The keystroke-level model for user performance time with interactive systems. *Commun. ACM* 23, 7 (July 1980), 396–410. doi:10.1145/358886.358895

[18] Mahil Carr and June Verner. 1997. Prototyping and software development approaches. *Department of Information Systems, City University of Hong Kong, Hong Kong* (1997), 319–338.

[19] Mahil Carr and June Verner. 1997. Prototyping and software development approaches. *Department of Information Systems, City University of Hong Kong, Hong Kong*, 319–338.

[20] Yoonseo Choi, Eun Jeong Kang, Seulgi Choi, Min Kyung Lee, and Juho Kim. 2024. Proxona: Leveraging LLM-Driven Personas to Enhance Creators' Understanding of Their Audience. doi:10.48550/ARXIV.2408.10937

[21] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973

[22] cppreference.com. 2024. Lambda Expressions (since C++11). https://en.cppreference.com/w/cpp/language/lambda. Accessed: 2024-08-23.

[23] Peitong Duan, Jeremy Warner, and Bjoern Hartmann. 2023. Towards Generating UI Design Feedback with LLMs. *Adjunct Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (2023), 1–3. doi:10.1145/3586182.3615810

[24] David Eppstein. 2001. Fast hierarchical clustering and other applications of dynamic closest pairs. *ACM J. Exp. Algorithmics* 5 (dec 2001), 1–es. doi:10.1145/351827.351829

[25] Christiane Floyd. 1984. A Systematic Look at Prototyping. In *Approaches to Prototyping*, Reinhard Budde, Karin Kuhlenkamp, Lars Mathiassen, and Heinz Züllighoven (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–18.

[26] Google. 2024. Google Gemini. https://gemini.google.com/. Accessed: 2024-08-23.

[27] Kasper B. Graversen. 2008. Method Chaining. First Class Thoughts, Web Archive. https://web.archive.org/web/20110222112016/http://firstclassthoughts.co.uk/java/method_chaining.html Accessed: 2024-08-23.

[28] Kamal Gupta, Justin Lazarow, Alessandro Achille, Larry Davis, Vijay Mahadevan, and Abhinav Shrivastava. 2021. LayoutTransformer: Layout Generation and Completion with Self-attention. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)* 00 (2021), 984–994. doi:10.1109/iccv48922.2021.00104

[29] Mark Harman. 2012. The Role of Artificial Intelligence in Software Engineering. *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)* 1 (2012), 1–6. doi:10.1109/raise.2012.6227961

[30] Sandra G. Hart. 2006. NASA-Task Load Index (NASA-TLX); 20 Years Later. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 50, 9 (October 2006), 904–908. doi:10.1177/154193120605000909

[31] Björn Hartmann. 2009. *Gaining Design Insight Through Interaction Prototyping Tools*. Ph.D. dissertation. CS Department, Stanford University, Stanford, CA.

[32] Marc Hassenzahl and Noam Tractinsky. 2006. User experience-a research agenda. *Behaviour & information technology* 25, 2 (2006), 91–97.

[33] John Hunt. 2006. *Agile Methods and the Agile Manifesto*. Springer, London, 9–30. doi:10.1007/1-84628-262-4_2

[34] Qianzhi Jing, Tingting Zhou, Yixin Tsang, Liuqing Chen, Lingyun Sun, Yankun Zhen, and Yichun Du. 2023. Layout Generation for Various Scenarios in Mobile Shopping Applications. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (2023), 1–18. doi:10.1145/3544548.3581446

[35] Evangelos Karapanos, John Zimmerman, Jodi Forlizzi, and Jean-Bernard Martens. 2009. User experience over time: an initial framework. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 729–738.

[36] Foutse Khomh, Chanchal K Roy, Janet Siegmund, Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. *Proceedings of the 26th Conference on Program Comprehension* (2018), 200–210. doi:10.1145/3196321.3196334

[37] Scott R. Klemmer, Anoop K. Sinha, Jack Chen, James A. Landay, Nadeem Aboobaker, and Annie Wang. 2000. Suede: a Wizard of Oz prototyping tool for speech user interfaces. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology* (San Diego, California, USA) *(UIST '00)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/354401.354406

[38] Harsh Lal and Gaurav Pahwa. 2017. Code Review Analysis of Software System Using Machine Learning Techniques. *2017 11th International Conference on Intelligent Systems and Control (ISCO)* (2017), 8–13. doi:10.1109/isco.2017.7855962

[39] 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 *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. doi:10.1109/icse48619.2023.00085

[40] Clayton Lewis. 1982. *Using the "Thinking-aloud" Method in Cognitive Interface Design*. Technical Report RC9265. IBM Research. https://dominoweb.draco.res.ibm.com/2513e349e05372cc852574ec0051eea4.html Accessed: 2024-08-23.

[41] Jianan Li, Jimei Yang, Aaron Hertzmann, Jianming Zhang, and Tingfa Xu. 2021. LayoutGAN: Synthesizing Graphic Layouts With Vector-Wireframe Adversarial Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43, 7 (July 2021), 2388–2399. doi:10.1109/tpami.2019.2963663

[42] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Chatting with GPT-3 for Zero-Shot Human-Like Mobile Automated GUI Testing. *arXiv* (2023). doi:10.48550/arxiv.2305.09434 arXiv:2305.09434

[43] Lina Mavrina, Jessica Szczuka, Clara Strathmann, Lisa Michelle Bohnenkamp, Nicole Krämer, and Stefan Kopp. 2022. "Alexa, You're Really Stupid": A Longitudinal Field Study on Communication Breakdowns Between Family Members and a Voice Assistant. *Frontiers in Computer Science* 4 (Jan 2022). doi:10.3389/fcomp.2022.791704

[44] Radka Nacheva. 2017. Prototyping approach in user interface development. In *Proceedings of the 2nd Conference on Innovative Teaching Methods (ITM 2017)*, Vol. 28. 78.

[45] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 97, 13 pages. doi:10.1145/3597503.3639187

[46] Christie Napa Scollon, Chu-Kim Prieto, and Ed Diener. 2009. Experience sampling: promises and pitfalls, strength and weaknesses. In *Assessing well-being: The collected works of Ed Diener*. Springer, 157–180.

[47] Justus D. Naumann and A. Milton Jenkins. 1982. Prototyping: the new paradigm for systems development. *Management Information Systems Quarterly* 6 (1982), 29–44. https://api.semanticscholar.org/CorpusID:261071872

[48] Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 249–256.

[49] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting Code Smells Using Machine Learning Techniques: Are We There Yet? *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), 612–621. doi:10.1109/saner.2018.8330266

[50] OpenAI. 2024. ChatGPT. https://openai.com/chatgpt/. Accessed: 2024-08-23.

[51] Sharon Oviatt. 2009. Multimodal Interfaces. In *Human-Computer Interaction* (1st edition ed.). CRC Press, Chapter 5, 20. doi:10.1201/9781420088861-10 Published 2 March 2009.

[52] Joon Sung Park, Lindsay Popowski, Carrie Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2022. Social Simulacra: Creating Populated Prototypes for Social Computing Systems. *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Oct 2022). doi:10.1145/3526113.3545616

[53] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. doi:10.1109/sp46215.2023.10179324

[54] Witold Pedrycz and Keun-Chang Kwak. 2007. The development of incremental models. *IEEE Transactions on Fuzzy Systems* 15, 3 (2007), 507–518.

[55] Kai Petersen, Claes Wohlin, and Dejan Baca. 2009. *The Waterfall Model in Large-Scale Development.* Springer Berlin Heidelberg, 386–400. doi:10.1007/978-3-642-02152-7_29

[56] Wolfgang Pree. 1992. Integration of Object-Oriented Software Development and Prototyping: Approaches and Consequences. In *Shifting Paradigms in Software Engineering*, Roland Mittermeir (Ed.). Springer Vienna, Vienna, 215–222.

[57] W Pree and G Pomberger. 1992. Object-oriented versus conventional software development: A comparative case study. *Microprocessing and Microprogramming* 35, 1 (1992), 203–211. doi:10.1016/0165-6074(92)90318-2 Software and Hardware: Specification and Design.

[58] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. *Proceedings of the 28th International Conference on Intelligent User Interfaces* (2023), 491–514. doi:10.1145/3581641.3584037 arXiv:2302.07080

[59] W. W. Royce. 1987. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of the 9th International Conference on Software Engineering* (Monterey, California, USA) *(ICSE '87)*. IEEE Computer Society Press, Washington, DC, USA, 328–338.

[60] Ritam Jyoti Sarmah, Yunpeng Ding, Di Wang, Cheuk Yin Phipson Lee, Toby Jia-Jun Li, and Xiang "Anthony" Chen. 2020. Geno: A Developer Tool for Authoring Multimodal Interaction on Existing Web Applications. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. ACM. doi:10.1145/3379337.3415848

[61] Omar Shaikh, Hongxin Zhang, William Held, Michael Bernstein, and Diyi Yang. 2023. On Second Thought, Let's Not Think Step by Step! Bias and Toxicity in Zero-Shot Reasoning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics. doi:10.18653/v1/2023.acl-long.244

[62] Heng Tao Shen, Yueting Zhuang, John R Smith, Yang Yang, Pablo Cesar, Florian Metze, Balakrishnan Prabhakaran, Kotaro Kikuchi, Edgar Simo-Serra, Mayu Otani, and Kota Yamaguchi. 2021. Constrained Graphic Layout Generation via Latent Optimization. *Proceedings of the 29th ACM International Conference on Multimedia* (2021), 88–96. doi:10.1145/3474085.3475497 arXiv:2108.00871

[63] Andrey Sobolevsky, Guillaume-Alexandre Bilodeau, Jinghui Cheng, and Jin L.C. Guo. 2023. GUILGET: GUI Layout GEneration with Transformer. *Proceedings of the Canadian Conference on Artificial Intelligence* (June 2023). doi:10.21428/594757db.08fe0a25

[64] Matthew Turk. 2014. Multimodal interaction: A review. *Pattern Recognition Letters* 36 (January 2014), 189–195. doi:10.1016/j.patrec.2013.07.003

[65] wandb. 2024. OpenUI: Describe UI Using Your Imagination, Render It Live. https://github.com/wandb/openui/. Accessed: 2024-08-23.

[66] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling Conversational Interaction with Mobile UI using Large Language Models. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (2023), 1–17. doi:10.1145/3544548.3580895

[67] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Trans. Softw. Eng.* 50, 4 (April 2024), 911–936. doi:10.1109/TSE.2024.3368208

[68] Jackie (Junrui) Yang, Yingtian Shi, Yuhan Zhang, Karina Li, Daniel Wan Rosli, Anisha Jain, Shuning Zhang, Tianshi Li, James A. Landay, and Monica S. Lam. 2024. ReactGenie: A Development Framework for Complex Multimodal Interactions Using Large Language Models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 483, 23 pages. doi:10.1145/3613904.3642517

[69] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. ACM. doi:10.1145/3453483.3454054