ReactGenie: A Development Framework for Complex Multimodal Interactions Using Large Language Models

Jackie Junrui Yang
jackiey@stanford.edu
Stanford University
Stanford, CA, USA

Yingtian Shi
shiyt0313@gmail.com
Tsinghua University
Beijing, China

Yuhan Zhang
zhangyh@stanford.edu
Stanford University
Stanford, CA, USA

Karina Li
karinali@stanford.edu
Stanford University
Stanford, CA, USA

Daniel Wan Rosli
danwr@stanford.edu
Stanford University
Stanford, CA, USA

Anisha Jain
anishaj037@gmail.com
Independent Researcher
USA

Shuning Zhang
zhang-sn19@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Tianshi Li
tia.li@northeastern.edu
Northeastern University
Boston, MA, USA

James A. Landay
landay@stanford.edu
Stanford University
Stanford, CA, USA

Monica S. Lam
lam@cs.stanford.edu
Stanford University
Stanford, CA, USA

ABSTRACT

By combining voice and touch interactions, multimodal interfaces can surpass the efficiency of either modality alone. Traditional multimodal frameworks require laborious developer work to support rich multimodal commands where the user's multimodal command involves possibly exponential combinations of actions/function invocations. This paper presents ReactGenie, a programming framework that better separates multimodal input from the computational model to enable developers to create efficient and capable multimodal interfaces with ease. ReactGenie translates multimodal user commands into NLPL (Natural Language Programming Language), a programming language we created, using a neural semantic parser based on large-language models. The ReactGenie runtime interprets the parsed NLPL and composes primitives in the computational model to implement complex user commands. As a result, ReactGenie allows easy implementation and unprecedented richness in commands for end-users of multimodal apps. Our evaluation showed that 12 developers can learn and build a non-trivial ReactGenie application in under 2.5 hours on average. In addition, compared with a traditional GUI, end-users can complete tasks faster and with less task load using ReactGenie apps.

CHI '24, May 11–16, 2024, Honolulu, HI, USA

ACM ISBN 979-8-4007-0330-0/24/05
https://doi.org/10.1145/3613904.3642517

CCS CONCEPTS

• **General and reference** → **Design**; • **Software and its engineering** → *Graphical user interfaces*; *Object oriented frameworks*; • **Information systems** → *Multimedia and multimodal retrieval*; • **Human-centered computing** → *User interface programming*.

KEYWORDS

multimodal interactions, development frameworks, programming framework, large-language model, natural language processing

ACM Reference Format:
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 (CHI '24), May 11–16, 2024, Honolulu, HI, USA.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3613904.3642517

1 INTRODUCTION

Multimodal interactions, combining multiple different input and output modalities, such as touch, voice, and graphical user interfaces (GUIs), offer increased flexibility, efficiency, and adaptability for diverse users and tasks [52]. However, the development of multimodal applications remains challenging for developers due to the complexity of managing multimodal commands and handling the low-level control logic for interactions. Existing frameworks [12, 14, 32, 41, 42, 49, 50] often require developers to manually handle these complexities, significantly increasing development costs and time. The voice modality, in particular, presents a unique challenge due to the compositionality and expressiveness of natural

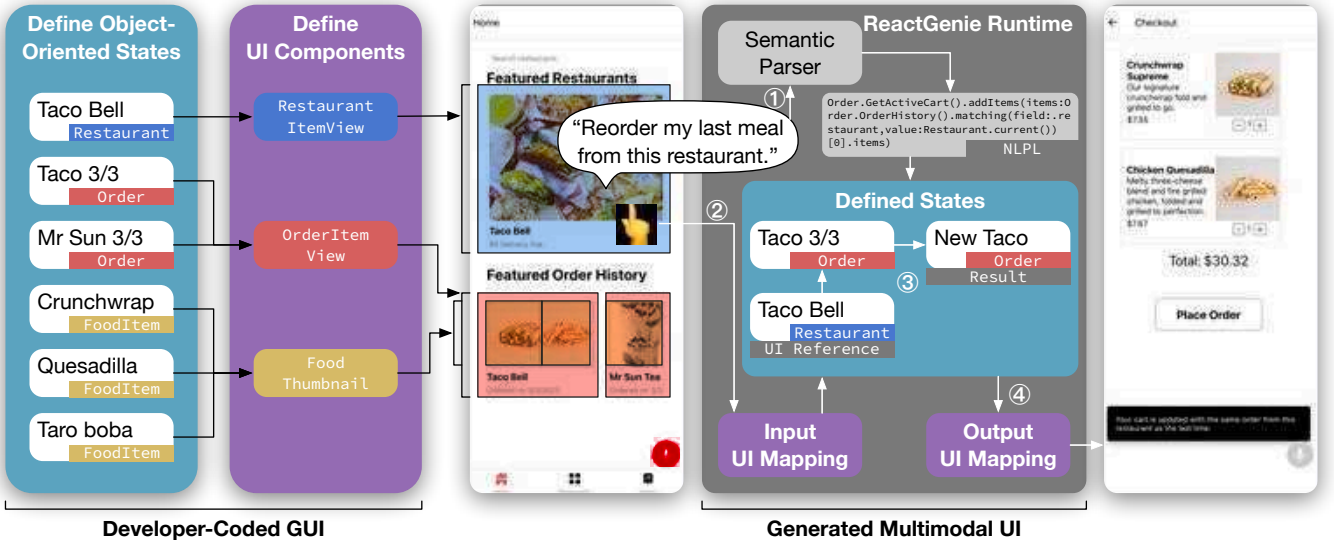

Figure 1: ReactGenie allows developers to easily build multimodal applications by better-separating interfaces (UI components) from computational models (object-oriented state). The demo (two screenshots) shows the user performing a multimodal (speech + touch) command (left screenshot), with the system executing the command by parsing the voice, understanding the reference in touch, and presenting the user with the appropriate UI interface and text feedback (right screenshot). (Left) ReactGenie provides this new yet familiar interface to create a GUI by defining states (data and logic) and UI components (transformation from data to UI representation). (Right) ReactGenie automatically generates a natural semantic parser from developer-defined states and generates input and output UI mappings from developer-defined UI components. ReactGenie can then execute rich multimodal commands by composing the methods and properties of states and presenting the results using existing UI components.

language. Sub-par implementations often greatly reduce the expressiveness of these multimodal interfaces. Various systems [28, 53] can automatically handle voice commands by converting them to UI actions, but they are prone to error and do not allow developers to fully control the app's behavior.

The research described in this paper aims to provide developers with a simple programming abstraction (see Figure 1) by hiding the complexity of natural language understanding and supporting the composition of different modalities automatically. Our goal is to enable users to access off-screen content/actions and complete tasks that normally involve multiple GUI taps in a single multimodal command, as illustrated in Figure 2. This flexibility is achieved with little additional effort from developers compared to traditional GUI apps. This approach encourages the adoption of multimodal interactions and makes multimodal interactions more accessible to end-users.

This paper presents ReactGenie[1], a declarative programming framework for developing multimodal applications. The core concept behind ReactGenie is a better abstraction that separates the multimodal input and output interfaces from the underlying computation models. ReactGenie uses an object-oriented state abstraction to represent the computation model of the app and uses declarative UI components to represent the UI. Users' compound multimodal commands are translated into a composition of multiple function calls using large language models (LLMs), e.g., to find the referred object/objects and make the right state change.

Existing declarative UI state management frameworks, such as Redux [6], use a single global state store to manage all of the state changes of the UI. The straightforward way to implement rich multimodal user commands in these existing frameworks is by making many imperative-style function calls. However, these function calls require the error-prone creation of many intermediate variables to store return values that are then used in the next function call as the programmer traverses the complex state stored in the monolithic object. These intermediate variables commonly cause missing references to variables when the neural semantic parser translates the user's natural language input into code [36]. In contrast, the object-oriented state abstraction in ReactGenie encourages componentized classes instead of a single global state store. The componentized classes result in smaller objects, each equipped with methods for relevant operations. This design supports multiple chained method calls/property accesses (method chaining) and provides a straightforward representation of the user's command with no need for intermediate variables (as shown in the example NLPL command in Figure 1). This allows ReactGenie to accurately compose the methods and properties of existing states needed for executing rich multimodal commands.

With ReactGenie, developers build graphical interfaces using a development workflow similar to a typical React + Redux [5] application. To add multimodality, the developer simply adds a few annotations to their code and example parses (pairs of expected

[1]project website (including source code): https://jya.ng/reactgenie, https://hci.stanford.edu/research/reactgenie/

end-user voice command examples and the corresponding function calls). These command examples indicate what methods/properties can be used in voice and how. By using the extracted class definitions and example parses from the developer's state code, ReactGenie creates a parser that leverages an LLM [17] to translate the user's natural language to a new domain-specific language (DSL), *NLPL*, Natural Language Programming Language. Combined with a custom-designed interpreter, ReactGenie can seamlessly handle multimodal commands and present the results in the graphical UI that the developer builds as usual.

As shown in Figure 1 left, developers can define both object-oriented state abstraction classes to handle data changes and UI components that explicitly map the state to the UI. Similar to React, when the user interacts with the app, the app's state will be updated, and the UI will be re-rendered. What sets ReactGenie apart is its unique ability to support rich multimodal input, as shown in Figure 1 right.

The main contributions of this research are as follows:

- ReactGenie, a multimodal app development framework based on an object-oriented state abstraction that is easy for developers to learn and use and generates apps that support rich multimodal interactions.
- A programming language, NLPL, used to represent user's multimodal commands. This involves the design of the high-level annotation of user-accessible functions, the automatic generation of a natural semantic parser using LLMs that targets NLPL, a new DSL for rich multimodal commands, and an interpreter that executes NLPL. These systems support automatic and accurate handling of natural language understanding in ReactGenie.
- Evaluations of ReactGenie:
  - For developers, we demonstrated its expressiveness through building three representative demo apps in different domains, its low development cost by comparing it with GPT-3 function calling, and its usability and learnability through a study with 12 developers successfully building a demo app.
  - For end-users, we measured the parser accuracy to be 90% with elicited commands from 50 participants and evaluated the usability of apps built using ReactGenie in a user study with 16 participants. We found users had a reduced cognitive load when using an app with ReactGenie-supported multimodal interactions compared to using a graphical user interface (GUI) app. They also preferred the multimodal app to the GUI-based app.

1.1 Targeted Interactions

ReactGenie supports *rich* interactions that are *complex* for current computer systems, but are *intuitive* for users. One example of a rich multimodal interaction is shown in the center of Figure 1: the user says, "Reorder my last meal from this restaurant" while touching the restaurant displayed on the screen. Such commands are common in human-to-human communication. Still, they involve multiple steps (retrieving the history of orders from the restaurant, creating an order, and adding food to the order) for the app. These commands are *complex* to implement today as they require combining inputs from both modalities and/or composition of different features.

ReactGenie supports a typical family of gesture + speech multimodal interactions. This aligns with one of the categories of speech and gesture multimodal applications proposed by Sharon Oviatt's seminal work [9]: The *recognition modes* ReactGenie supports are *simultaneous and individual modes*, meaning that ReactGenie supports users to use either speech-only interactions, gesture-only interactions, or both interactions at the same time ("What is the last time I ordered from this *[touch on a restaurant]* restaurant"). The *supported gesture input type* is *touch/pen input*, and the *size of the gesture vocabulary* is a *deictic selection*. This means that ReactGenie focuses on scenarios where the user's gesture input resolves object references through pointing in a multimodal command. The *size of speech vocabulary* is *arbitrary human sentences*, and the type of linguistic processing is *large-language model processing*. The last two terms are new types we invented to better describe ReactGenie's support for rich commands and the use of highly generalizable large-language models. Following Oviatt's original classification, ReactGenie would be classified as *large vocabulary* and *statistical language processing*. ReactGenie uses *late semantic fusion* to fuse input from different modalities, which means the system integrates and interprets the meaning of inputs from multiple modalities only after each input has been independently processed and understood.

With ReactGenie, the developer simply provides a small amount of additional information associated with each input method and function. Our system supports the full compositionality of input modalities and functions by automatically translating a user command into one of exponentially many possible action sequences. The richness of user interaction afforded by our system is unprecedented, as traditional multimodal programming frameworks require developers to hard-code every combination of features supported.

ReactGenie lets the programmer simply *describe* the functionality of their code, including actions they support and the relationship between UI and data. This allows ReactGenie to handle these rich multimodal commands in arbitrary combinations of actions without requiring direct developer input. The example in Figure 1 is supported by:

(1) ReactGenie first translates the user's voice command to the NLPL code. For example, the user refers to an element in the UI by voice ("this restaurant"), and the semantic parser generates a special reference `Restaurant.current()`.
(2) ReactGenie extracts the tap point from the UI and uses the UI component code to map the tap point back to a state object `Restaurant(name:"TacoBell")`.
(3) With the parsed DSL and UI context, ReactGenie's interpreter can execute the generated NLPL using developer-defined states. It first retrieves the most recent order from "Taco Bell", designated as "Taco 3/3". Then, it creates a new order, designated as "New Taco". Finally, the interpreter adds all the food items from "Taco 3/3" to "New Taco" and returns the new order.
(4) ReactGenie passes the return value of the NLPL statement to the output UI mapping module. Because the return value is an `Order` object, ReactGenie searches in the developer's UI component code to find a corresponding representation

(Output UI Mapping) to present the result to the user. ReactGenie also generates a text response using the LLM based on the user's input, parsed NLPL, and the return value: "Your cart is updated with the same order from this restaurant as the last time."

During this process, the ReactGenie framework uses its knowledge about the developer's app to automatically understand a multimodal compositional command, compose actions to execute, and find the appropriate interface to present the results to the user. This pipeline allows ReactGenie to handle more commands than prior frameworks with little developer input.

2 RELATED WORK

In this section, we review related work on multimodal interaction systems, Graphical and Voice UI frameworks, and multimodal interaction frameworks.

2.1 Multimodal Interaction Systems

Many researchers have proposed multimodal interaction systems. The earliest multimodal interaction systems, such as Bolt's "Put-that-there", were developed in the 1980s [15]. They demonstrated that users can interact with a computer using voice and gestures. QuickSet [21] further demonstrated use cases of multimodal interaction on a mobile device and showed military and medical applications.

Recent work has explored different applications of multimodal interaction, including care-taking of older adults [48, 51], photo editing [38], and digital virtual assistants [33]. Researchers have also explored different devices and environments for multimodal interaction, including augmented reality [59], virtual reality [39, 56], wearables [16], and the Internet of Things [25, 34, 55, 58].

These projects have demonstrated the great potential of multimodal interaction systems. However, multimodal systems still have limited adoption in the real world due to the development complexity they currently require.

2.2 Graphical UI frameworks

ReactGenie is built on top of an existing graphical UI framework to provide a familiar development experience. Model–view–controller (MVC) [35] is the traditional basis of UI development frameworks and is used in frameworks such as Microsoft's Windows Forms [29], and Apple's UIKit [1]. The model stores data while the controller manages GUI input and updates the GUI view based on data changes. Typically implemented in object-oriented programming languages, MVC can be compared to a shadow play, where objects (controllers) manipulate GUIs and data to maintain synchronization. However, updating the model with alternative modalities, such as voice, is not feasible due to the strong entanglement between models and GUI updates.

Garnet [43, 45], a user interface development environment introduced in the late 1980s, is another notable approach to GUI development. Garnet introduced concepts like data binding, which allows the GUI to be updated automatically when the data changes. It also tries abstracting the GUI state away from the presentation using interactors [44]. While interactors allow the UI state to be rewired and thus to be updated using another modality like voice or gesture [37], they do not enable manipulation of more abstract states (e.g., foods in a delivery order) that are not directly mapped to a single UI control.

Declarative UI frameworks, such as React [2], Flutter [3], and SwiftUI [7], are a more recent approach to UI development. With declarative UI frameworks, programmers write functions to transform data into UI interfaces, and the system automatically manages updates. To ease the management of states that may be updated by and reflected on multiple UI interfaces, centralized state management frameworks, such as Redux [6], Flux [10], and Pinia [4], are often used together with these declarative UI frameworks. They provide a single source of truth for the application state and allow state updates to be reflected across all presented UIs. This approach can be likened to an overhead projector, where the centralized state represents the writing and the transform functions represent the lens projecting the UI to the user. While this approach improves separation and UI updating, it sacrifices the object-oriented nature of the data model. This centralized state works well with button pushes but comes short in dynamically composing multiple actions to support rich multimodal commands.

ReactGenie reintroduces object-orientedness to centralized state management systems by representing the state as a sum of all class instances in the memory. Developers can declare classes and describe actions as member functions of the classes. ReactGenie captures all instantiated classes and stores them in a central state. This more modularized model is analogous to actors (class instances) in a movie set, with views (UI components) acting as cameras capturing different angles of the centralized state. In this way, ReactGenie enables rich action composition through type-checked function calls. Furthermore, developers can tag specific cameras to point at certain objects, enabling automatic UI updates from state changes. These features allow ReactGenie apps to easily support the compositionality of multimodal input and enable the interleaving of multimodal input with other graphical UI actions.

2.3 Voice UI frameworks

Commercial voice or chatbot frameworks, such as Amazon Lex, Google Dialogflow, and Rasa, are designed to handle natural language understanding and generation. These frameworks allow developers to define intents and entities and then train the model to recognize the intents and entities from the user's input. In this context, intents refer to categories of the user's action, such as making a reservation or asking for weather information, and one action can only be mapped to one intent. Intents are usually mapped to different programming implementations to handle commands in the corresponding intent categories. These frameworks require a complete redevelopment of an application to support voice-only input. Frameworks such as Alexa Skills Kit and Google Actions allow developers to extend existing applications to support voice input. However, these still require manual work to build functions only for voice, and the visual UI updates are limited to simple text and a few pre-defined UI elements. Additionally, the one-intent-one-implementation nature of the intent-based architecture limits the compositionality of the voice commands.

Research-focused voice/natural language frameworks, such as Genie [19, 54] and other semantic parsers [13, 46], are designed to

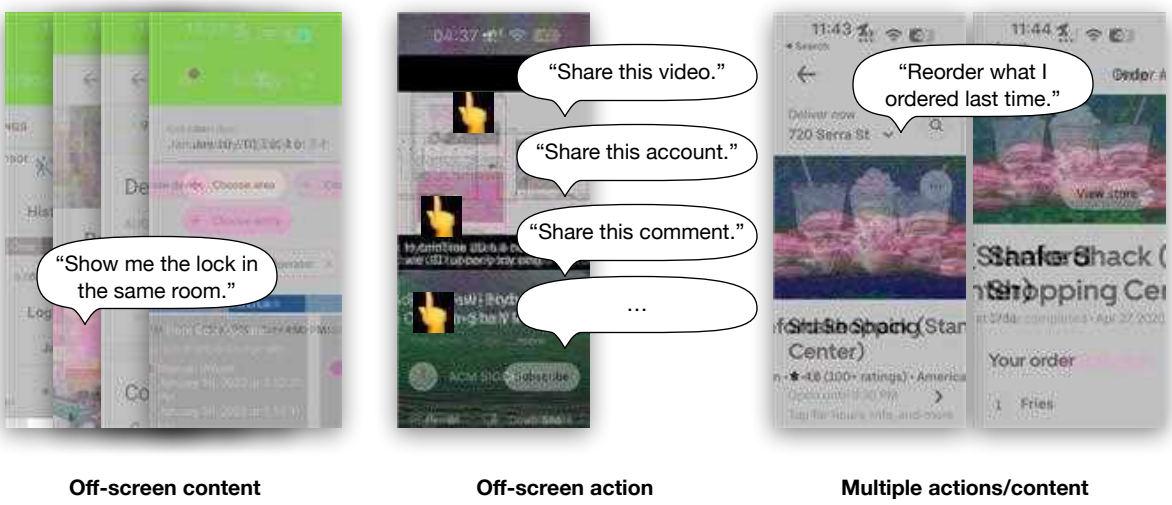

Figure 2: ReactGenie's targeted interaction scenarios.

support better compositionality of voice commands. However, given that today's app development is primarily geared toward mobile and graphical interfaces, these frameworks require extra work from the developer and do not support multimodal features. ReactGenie improves this experience by integrating the development of voice and graphical UIs, allowing developers to extensively reuse existing code and support multimodal interactions.

2.4 Multimodal Interaction Frameworks

Prior work has also proposed multimodal interaction frameworks that allow developers to build multimodal applications. One of the earliest works is presented by Cohen et al. [20]. It includes ideas like forming the user's voice command as a function call and using the user's touch point as a parameter to the function call. Later, researchers created standards [23, 24] and frameworks [12, 14, 32, 41, 42, 49, 50] to help developers build apps that can handle multiple inputs across different devices. Although these frameworks provide scaffolding for developers to build multimodal applications, they mostly treated voice as an event source that can trigger functions the developer has to explicitly implement for voice. Developers also have to manually update the UI to reflect the result of the voice command. This manual process limits voice commands to simple single-action commands and makes it difficult for developers to build richer multimodal applications.

Recently, there are research projects on generating voice commands by learning from demonstration [27, 40, 47], extracting from graphical user interfaces with large language models [28, 53], or building multimodal applications using existing voice skills [57]. The first approach still requires developers to manually create demonstrations for each action and limits the compositionality of the voice commands. The second approach is useful for accessibility purposes, but it relies on the features being easily extractable from the GUI. It is uncertain how well the first two approaches can generalize to more complex UI tasks that require multiple UI actions. The third approach is constrained by what is provided by the voice skills and, traditionally, these have been limited due to the added development effort.

In comparison, ReactGenie leverages the existing GUI development workflow and requires only minimal annotations to the code to generate multimodal applications. Having access to the full object-oriented state programming codebase, ReactGenie can handle the natural complexity of multimodal input, compose the right series of function calls, and update the UI to reflect the result automatically.

3 SYSTEM DESIGN

In this section, we first define the design goals of the framework. Then, we describe the theory of operation that addresses the design goals. Finally, we discuss the implementation of the system components and workflow.

3.1 Design goals

Our design goals include aspects of the interaction design of ReactGenie apps as well as the design of the framework itself.

3.1.1 Interaction Design. ReactGenie is primarily designed to enhance user interaction with mobile applications, but the concept should also apply to apps on other platforms. Today, mobile applications are well-optimized for touch and graphical interactions. Users can use the graphical interface to see content on the screen and use touch to access actions on the screen. To further enhance the user's performance and reduce cognitive load, ReactGenie focuses on supporting interactions that often involve touch actions used together with a voice command.

Here is a series of example commands in interactions with a food ordering app between user A and their friend user B:

C1 A knows what they want, so A says, "Show me what I ordered last week from McDonald's." The app responds with the order history.

C2 A wants to add a previously ordered food into the cart (not available on UI). A says, "Order this hamburger," with a tap on the "Big Mac" entry in the order history, and the app adds a "Big Mac" to the shopping cart.
C3 B wants to order something different, so they tap on the restaurant to view the menu.
C4 B doesn't like beef, so they say, "Show me food without beef," the app displays options accordingly.
C5 B says, "Order a meal with this sandwich," with a tap on the "McChicken sandwich," the app adds a McChicken meal to the cart.

This interaction demonstrates the power of these multimodal commands where voice and touch are used interchangeably or in conjunction. These commands can be categorized into three interaction design goals (see Figure 2):

I1 Access off-screen content (C1, C4): For example, the user is looking at the smart home app and notices abnormal motion on the smart home app's living room security camera. So they can talk to the smart home app, "Show me the lock status history in the same room of this device." This interaction usually requires multiple GUI navigation steps (go to the device's room page, navigate to the door lock section, check door lock history) to access the content.
I2 Access off-screen actions (C2): For example, the user says, "Share this creator/comment" while watching a YouTube video. Some actions are hidden behind a menu or a button, and some are not accessible at all on mobile devices.
I3 Combine multiple actions/content (C5): For example, the user says, "Order what I ordered last time" while looking at a food delivery app. This usually requires the user to go back and forth between an order detail page and a menu page.

The common theme among these interactions is that they require the multimodal interaction framework to understand the content and actions available in the app. For content, the framework needs to know what is the content on the screen, how to access it, and how to represent returned content (from user-initiated commands) to show the retrieved content. For actions, the framework needs to know the list of available actions and how to render changes on the user interface after the action is triggered. Finally, the framework needs to translate the user's intent, which may be rich, into possibly a series of actions and content displays.

With ReactGenie, apps will have a microphone button on the screen. When the user taps on the button, the user can say their command and refer to the content on the screen by tapping it. The app will then parse the voice command and touch input and execute the corresponding actions to help the user with the scenarios described above.

3.1.2 Framework Design. To translate the content and actions in the user's rich multimodal intent, ReactGenie needs to obtain information about the app's capabilities from the code. The design goal for ReactGenie is to do this in a way that causes minimal disruption for the application developer.

Without a proper framework for multimodal apps, the developer must design their own mechanisms to handle voice, handle the complexity of multimodal commands, and maintain control of the app's behavior. Our goals for the ReactGenie framework include handling these issues:

F1 Maintain the expressiveness of the developer for their GUI appearances and specific app functionality.
F2 Reduce development cost by maximizing the reuse of existing GUI code and hiding the complexity of handling multimodal commands from developers.
F3 Ease the learning curve by providing a similar programming experience to existing GUI frameworks.

3.2 Theory of Operation

ReactGenie presents an object-oriented state programming model to the developer. *State code*, in the context of GUI development, refers to the part of the application that manages the data and logic that determine the state of the user interface. The global state store in Redux is typically represented as a tree of stored variables and the associated functions to transform them. The concept of state in declarative UIs is similar to the model in the model-view-controller (MVC) programming models.

As mentioned in Section 2.2, in frameworks like React, UI development is moving towards separating the UI from the state. Developers define functions (*components*) that convert the state into UI. Each UI component receives part of the state and renders the UI in an HTML-like format. Components can sometimes render a part of their state using another component in a compositional way (e.g., a restaurant menu component can use menu item components to render each food item, and the restaurant menu components host a list to organize the menu item components). Therefore, there is a unidirectional data flow from the state to the UI so that the state update is automatically reflected on the UI without any additional code from the developers. In comparison, in a typical MVC paradigm, the controller simultaneously updates the model (state) and the UI to keep them in sync. The unidirectional data flow feature in declarative UIs (compared to MVC) allows data to be updated outside of GUI input because data updates are no longer entangled with UI update code in MVC's controller code. This feature allows ReactGenie to use multimodal commands to change the same state and update the UI accordingly, maximizing the reuse of existing GUI code (**F2**).

However, in these existing declarative UI frameworks, the state is represented by a single global data store. This data store contains a tree-like data structure with all the content and status of the app (state) and a list of functions to manipulate the state data. The functions usually contain parameters for indexing into the state object and parameters to further specify the actions. For example, an `ADD_FOOD_TO_ORDER` action needs to have the parameters of `food_id`, `order_id`, and `quantity`. This works well for GUI design, as developers can implement an action for each button press, and each button in the GUI stores the corresponding ID that it needs to call when pressed. However, this makes it difficult to handle typical multimodal commands, which require composing multiple actions (**I3**). To translate a user command of "Add two hamburgers to a new order", an example translated program in React-Redux would be:

```
// create the order
dispatch(CREATE_ORDER())
```

```
export const orderReducer = (state = {orders: []}, action: any) => {
    switch (action.type) {
        case FETCH_ORDERS:
            return {...state, orders: fetchOrdersFromServer()};
        case CREATE_ORDER:
            const newOrder =
                {id: state.orders.length + 1, items: [], date: Date.now()}
            return {...state,orders: [...state.orders,newOrder]};
        case ADD_FOOD_TO_ORDER:
            const {foodId} = action.payload;
            const updatedOrders = state.orders.map((order) => {
                if (order.id === state.orders.length) {
                    return {
                        ...order,items: [...order.items, {id: foodId}]
                    };
                }
                updateServer();
                return order;
            });
            return {...state,orders: updatedOrders,};
        default:
            return state;
    }
};
```

React-Redux's Monolithic State Implementation

```
@GenieClass("Past order or a shopping cart")
class Order extends DataClass {
    @GenieKey()
    public orderId: string;
    @GenieProperty("Items in the order")
    public orderItems: FoodItem[];
    @GenieProperty("When order is created")
    public orderDate: DateTime;
    constructor({orderId, orderItems, orderDate}: {orderId: string,
orderItems: FoodItem[], orderDate: DateTime}) {
        super({orderId, orderItems}); this.orderId = orderId;
this.orderItems = orderItems; this.dateTime = DateTime();
    }
    @GenieFunction()
    All(): Order[] {
        return fetchOrdersFromServer();
    }
    @GenieFunction("Create a new order")
    static CreateOrder(): Order {
        return new Order({orderId: randomId(), orderItems: []});
    }
    @GenieFunction("Add an item to the order")
    addItem({foodItem}: {foodItem: FoodItem}) {
        this.orderItems.push(foodItem); updateServer();
    }
}
// Example parses omitted here, see appendix
```

ReactGenie's Object-Oriented State Implementation

Figure 3: A comparison between state code in React-Redux and in ReactGenie. (Left) Part of an example state code in Redux. Data is stored in the state variable, and the state can be mutated by the actions defined. These actions (stored in a `Reducer`) do not have explicit types, and they directly manipulate the state, so no return values are defined. Note that the return values of case statements in a `Reducer` indicate the new state variable after the state changes; actions do not have return values. Due to its monolithic design, it is hard to compose functions together to achieve some multimodal actions. (Right) Part of an example state code in ReactGenie. Automatically managed by ReactGenie, the state is composed of all the instantiated objects' `DataClasses`. Actions in the state code are defined as methods of the class. All the methods have explicit parameter types and return types. These functions can be composed together to achieve multimodal actions.

```
// find the id of the created order
const order_id = store.orders.last().id
// find the food id of hamburger
const food_id = store.foods.
  filter(food => food.name === 'hamburger')[0].id
// add food to order
dispatch(ADD_FOOD_TO_ORDER(order_id, food_id, 2))
```

This process involves the creation of multiple intermediary variables and queries to the state object. When the neural semantic parser generates code for this process, we have found that LLMs will often generate a line of code referencing an intermediary variable that has not been declared before, causing an error in response to the user's query. The problem is that even if we have already created a new order, we would have to retrieve the order ID from the state object, save it in an intermediary variable, and feed that ID into the imperative style actions. This issue is solved in ReactGenie by applying concepts of object-oriented programming where an order can be represented by a separate object that has both the data and all of the relevant actions (e.g., adding a food item to the order) associated with the object, so a retrieved order can directly be used to call the add food item action.

So, ReactGenie introduces the object-oriented programming model to componentize the state of a declarative UI app. In componentizing the state object, developers implement smaller objects containing its relevant content stored in properties and actions defined as methods. This componentized design allows ReactGenie to translate a typical multimodal command into a single statement with method chaining. Using the example above, the user's command can be translated as: `Order.CreateOrder().addFoodItem(foodItem:FoodItem.All().matching(field:.name,value:"hamburger"),quantity:2`. With ReactGenie's state abstraction, LLMs can generate code that directly calls `addFoodItem` after creating the new order. These methods are also strictly typed, which helps the natural semantic parser develop the correct combination of methods with fewer runtime errors. This succinctness improves the accuracy of the neural semantic parser that translates the user's natural language command to executable code. Furthermore, as the GUI and voice modality share the same content and state, this representation supports interchangeability in user input modality for each part of the rich multimodal command.

In practice, to work with ReactGenie's object-oriented state abstraction, the developer identifies the user-accessible content (object or object properties) and actions (functions) with the `GenieProperty` and `GenieFunction` annotations, respectively, along with an example of how it may be referred to in English as shown in Figure 3 right. The GUI is programmed using compositional components (similar to other declarative UI frameworks described above) rendered from the user-accessible state objects. This allows the internal state to be rendered to the user in a GUI. The high-level model of ReactGenie resembles popular GUI development frameworks (React + Redux), which makes it easy for developers to learn and use (**F3**). ReactGenie automatically handles the retrieval of content (objects/properties) off-screen (**I1**), the execution of actions off-screen (**I2**), and combinations of the two (**I3**). In this way, voice and multimodal commands will be handled

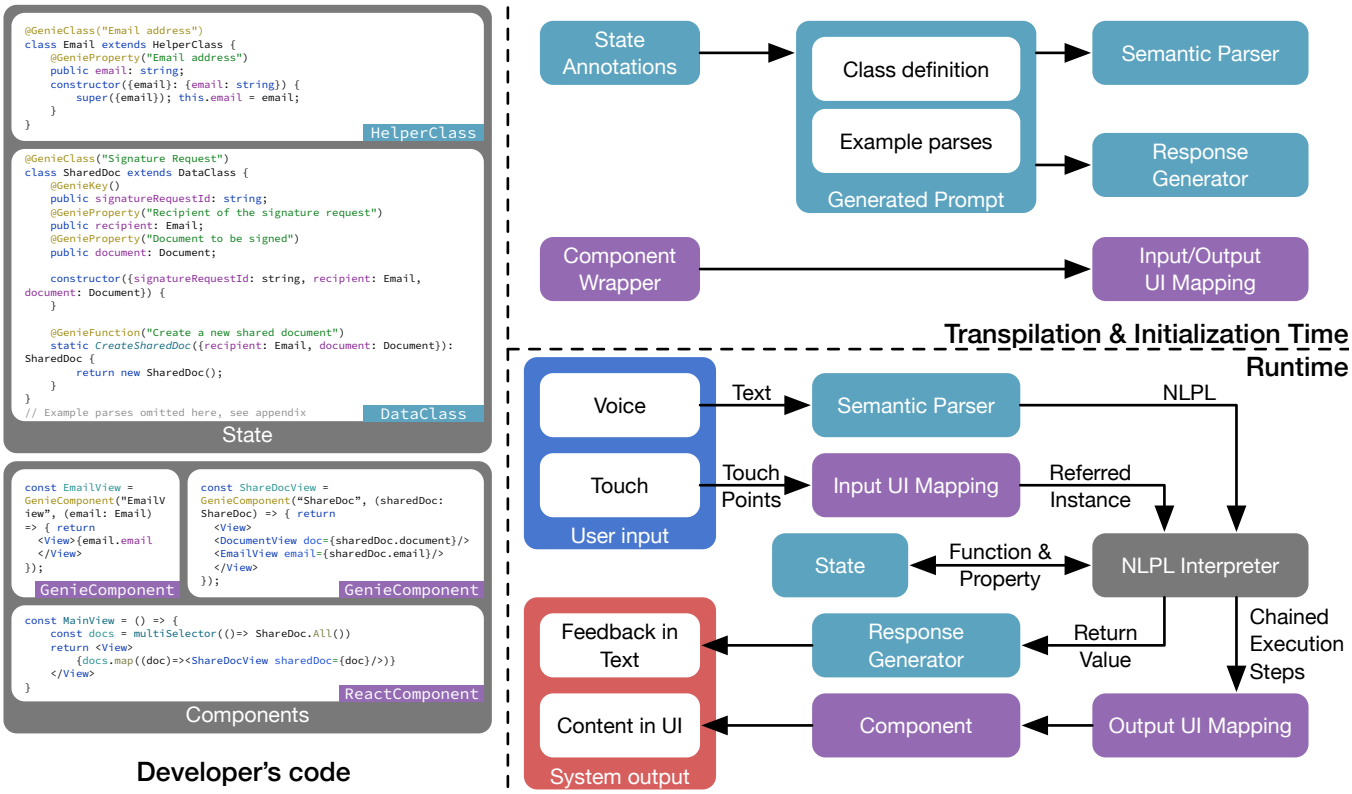

Figure 4: Overview of the ReactGenie system: (Left) Developers write object-oriented state code for programming content and actions and define the UI as cascading components. (Right top) ReactGenie operates at transpilation and initialization time to generate runtime modules. (Right bottom) Developer modules, generated modules, and ReactGenie modules come together to process rich multimodal commands from the user. This workflow is similar to regular GUI development, maximizes code reuse, and allows full control of the app behavior.

by LLM-generated programs and touch commands can be handled in the traditional way where developers write programs to handle each individual user input event.

By reusing a programming model similar to existing UI frameworks, ReactGenie allows developers to control the look and feel of their app to a similar degree as typical declarative UI frameworks. Meanwhile, developers can also control what function the end-users can access in multimodal interactions through whether or not to add annotations. Hence, developers have full control over the functionality of the app. These designs help developers maintain full control of their app while enjoying the benefits multimodal interaction offers to end-users (**F1**).

3.3 The Developer's Programming Model

From the developers' point of view, using ReactGenie is similar to any declarative UI framework. They need to implement the state code that provides the content and actions supported in the app. They also need to specify the UI components that translates current state classes into UI interfaces that the user can see and manipulate. Table 1 provides a list of functions and annotations that developers need to provide for ReactGenie (see Figure 4 left for an example).

In Section 3.5, we will describe in detail the right side of Figure 4, which is how the ReactGenie system uses the developer-supplied code described in this section, (1) transpiles (source-to-source compiles) it into ReactGenie modules (Right top), and (2) uses the generated ReactGenie modules to handle end-users' input (Right bottom).

3.3.1 State Code. Developers provide the content and actions in an app through ReactGenie's object-oriented state model, implemented in TypeScript[2]. As with all object-oriented programming models, ReactGenie's state code includes the definition and implementation of classes. Classes have declarations, methods, and properties, which can be labeled as `GenieClass`, `GenieFunction`, and `GenieProperty`. These annotations in the state code (or *State Annotations* for short) indicate that they are user-accessible via multimodal commands. All ReactGenie annotations have an optional parameter that denotes the purpose of that class/function/property, similar to comments in code. These code annotations or decorators in TypeScript are tags written before the class, method, and property declarations. This allows the relevant annotation code to be executed at initialization time to transform the capabilities of the annotated classes or methods.

[2] https://www.typescriptlang.org/

Table 1: Annotations and Functions for programming ReactGenie.

Annotation/Function	Location	Type	Parameters	Purpose
`GenieClass`	DataClass/ HelperClass	Annotation	`comment` (optional): describing the class's purpose.	Indicates the class to be user-accessible through multimodal interactions, essential for determining which parts of the code are relevant to the multimodal experience.
`GenieFunction`	DataClass/ HelperClass	Annotation	`comment` (optional): describing the function's purpose.	Tags a method to be exposed to multimodal interaction, marking this method as something the user may call directly.
`GenieProperty`	DataClass/ HelperClass	Annotation	`comment` (optional): describing the property's purpose.	Exposes the property for multimodal interaction, letting users interact with this property through multimodal commands.
`GenieKey`	DataClass	Annotation	None required.	Identifies a unique ID property that uniquely identifies an instance within a DataClass, crucial for managing and retrieving data instances.
`constructor`	DataClass/ HelperClass	Function	Varies based on class requirements.	Initializes a DataClass instance with required data, setting up the basic data structure of an object.
`GenieComponent`	UI Component	Wrapper Function	`target`: DataClass or HelperClass to be displayed. `component`: The UI component to be wrapped.	Wraps a UI component and maps it to a DataClass or HelperClass instance, telling how to present data/state visually and enable touch references in multimodal interactions.

All states that are accessible via multimodal interactions must be declared as instances of the `DataClass` and `HelperClass` provided by ReactGenie. A `DataClass` stores the app's data, and a `HelperClass` provides definitions to ease the user's interaction with the data. The properties of a `DataClass` can be of TypeScript primitive types, a `DataClass`, or a `HelperClass`.

Figure 4 left shows an example of a document signing app; examples of each class are shown in the top left. As shown in the figure, the `SharedDoc` `DataClass` tracks the lifecycle of a document's signature request. An `Email` class is a `HelperClass` that helps manage users' email addresses. ReactGenie also has system-provided `HelperClass` such as `DateTime` and `TimeDelta` to help developers manage date, time, and period of time. The introduction of the `HelperClass` class not only allows developers to define helper functions but also helps with type-checking, which is useful for developers to write less buggy code and for ReactGenie's semantic parser to generate more correct NLPL.

All `DataClasses` start with the class declaration, which begins with the `GenieClass` annotation and then the class name and a required inheritance of `DataClass`. See the first two lines of `ShareDoc` in Figure 4. Developers need to implement one method and one property:

(1) `constructor` method: The constructor of the class initializes the instance with all the required data.
(2) `id` property: A unique identifier of the instance, annotated by the `GenieKey`.

Developers can also implement additional functions and properties to complete the `DataClass`. If developers want the user to be able to use the functionality directly, they need to add the `GenieFunction` and `GenieProperty` annotation. An example of that can be found in Figure 4 left (see `document` and the `CreateSharedDoc` in `SharedDoc`). Sometimes, developers may want to implement internal housekeeping functions, such as clearing the application cache or managing internal database transactions. Those functions/properties should not be annotated and will not be called by ReactGenie.

The `HelperClass` allows developers to define new types that can be used in the `DataClass`. A specific example is the ReactGenie-supplied `DateTime` `HelperClass`. It can support operations like offsetting the time by a certain amount or setting the year/month/day/hour/minute/second/day of the week to a certain value. It allows commands such as "last Thursday" to be translated to "`DateTime.Current.offset(week:-1).set(weekOfTheDay:4)`". The developer can define other `HelperClass` instances to support more complex operations such as length unit conversion. Similarly, the `HelperClass` needs to have a `GenieClass` annotation and needs to inherit from `HelperClass`. The `HelperClass` only requires a `constructor` method that takes the data as input and initializes the instance. ReactGenie will also generate a `Current` property for the `HelperClass` that returns the instance that is being referred to by the user. ReactGenie does not keep track of the instances of `HelperClass` separately in memory, but instead, they will be part of the `DataClass` that uses them.

For both `DataClass` and `HelperClass`, the developer can define a `description` method to customize the string representation of the instance for response generation. By default, ReactGenie will generate a JSON-like representation using all the instance's properties.

Finally, developers need to provide example parses for the `DataClass` and `HelperClass`. *Example parses* or *few-shot examples* are pairs of expected end-user voice command examples and the corresponding translated NLPL. Developers provide them as few-shot examples for ReactGenie's neural semantic parser. These examples are helpful for the parser to learn about what the app is about and the syntax of NLPL. In practice, around 10 example parses are sufficient. Developers do not have to cover all use cases, and ReactGenie's semantic parser can automatically generalize to most of the app's features and user expressions.

3.3.2 Built-in Dataclass Methods. ReactGenie automatically generate three methods for each `DataClass`:

(1) `All` method: For voice inputs, the user needs to refer to all or a select set of instances of certain objects. A static `All` method is provided that returns an array of all the instances of the DataClass. There are built-in functions that support filtering array elements based on value or value ranges, sorting based on an order on a property, and extract an element by its index.
(2) `Get` method: A static method that takes an `id` as input and returns the instance with the corresponding `id`.
(3) `Current` property: A static method that returns the instance that is being referred to by the user. This is automatically annotated with `GenieProperty`.

Like many of the common state management frameworks, when any of the properties of a `DataClass` instance are changed, UI components on screen that refer to that property will be automatically re-rendered with the most up-to-date data. This ensures that the UI is always in sync with what is being represented in the state.

ReactGenie automatically maintains the instances of `DataClass` in memory to form the app's state. `DataClass` can be backed up remotely, which is common in modern app development.

3.3.3 Component/UI Code. ReactGenie developers need to define the GUI (as shown in the three code boxes in the bottom left of Figure 4) as a set of functional components[3], similar with React. These components can refer to each other to facilitate reuse. It is common for every single instance of a `DataClass` or `HelperClass` to be represented by a component. For example, `EmailView` represents an `Email` instance, while `SharedDocView` represents a `SharedDoc` instance. Therefore, ReactGenie introduces a special wrapper function (or *Component Wrapper* for short) called `GenieComponent` to show that explicit mapping. Instead of the arbitrary parameters of a normal functional component, components wrapped by `GenieComponent` take a `DataClass` or `HelperClass` instance as input. `GenieComponent` allows the ReactGenie runtime to understand which component is mapped to which instance in memory to facilitate reference by touch. It also allows ReactGenie to render the result of the user's request using the developer-defined component. While defining `GenieComponent`, the developer can also specify an optional title and priority (both can be a method of the state instance) for the interface, which is relevant for choosing the interface to render multimodal command responses.

[3] https://react.dev/learn/your-first-component

3.4 NLPL

We use a neural network to translate natural language commands into NLPL, a domain-specific language (DSL) we created. We feed the large-language model the automatically-extracted developer's class skeleton (only declaration and type information, no implementations to save tokens, and reduce distractions), developer-supplied few-shot examples, and the user's current voice command and ask it to generate NLPL code for understanding the user's intention.

We do not generate JavaScript directly because the expressiveness of JavaScript may cause unintended changes to the app's behavior (contradicting **F1**). We cannot use a simple intent classifier, such as the one used by traditional voice assistants, because of the complexity of our user's multimodal commands. The *DSL interpreter* module runs the generated NLPL code and calls the corresponding methods in the developer's state definition code (called *state code* in the following sections). This also allows ReactGenie to handle users' verbal references for simultaneous touch input through special reference functions (see Section 3.5.2).

The NLPL is designed to meet these language design goals:

L1 *Easy to generate*: LLMs can generate syntactically correct NLPL code.
L2 *Robust against generation errors*: LLMs can generate semantically correct NLPL code.
L3 *Able to express multimodal commands*: NLPL can express diverse multimodal commands.

Therefore, because of **L1**, NLPL has to be in a form similar to existing programming languages that LLMs are trained on. To help with **L2**, we also want NLPL to be strongly typed. We tried a syntax similar to TypeScript, but we noticed the LLM-based semantic parser occasionally generated the correct parameters but in the wrong order. Therefore, we decided to use a syntax similar to Swift, a strongly typed language that requires parameter names to be specified in the function call.

To support the expressiveness of human languages **L3**, a simple data formatting language such as JSON is not enough. JSON is good at representing structured data and is sometimes used to represent simple intent with a single function call and parameters. However, the user's commands can consist of multiple method invocations and represent complex logic flow, so we looked at the language structures of what people may say to a ReactGenie app. When interacting with a virtual assistant, people typically utter an imperative or interrogative sentence.

An imperative sentence must have a verb and an object. Some more complex imperative sentences can have object modifiers and verb modifiers. For example, "[Change the background color](verb) of [all the yellow](object modifier) [textboxes](object) [to orange](verb modifier)". Objects can be translated to function calls to retrieve the corresponding objects, e.g., "this food" `->Food.Current()` and "textboxes" `->TextBox.All()`. Object modifiers can be translated to SQL-like operations, e.g., "food with a rating above 3" `->Food.All().between(field:.rating,from:3)`, "all the yellow textboxes" `->TextBox.All().equals(field:.color,value:"yellow")`. Note that we did not use SQL syntax because SQL does not have easy support for calling functions of objects, so it would have trouble translating verbs/actions. Verb and verb modifiers are translated to function calls.

For example, changing the background color to orange would be `.setBackgroundColor(color:"orange")`.

We also avoid supporting lambda expressions, variable declaration, and control flow statements to avoid reference errors to increase robustness (**L2**) in translation and reduce complexity. For lambda expressions, NLPL automatically distributes function calls to individual elements of objects to support plural objects: "make all textboxes orange" `->TextBox.All().setBackgroundColor(color:"orange")`. For variable declaration, we use method chaining to avoid reference errors. This is because, in early testing, we found that OpenAI Codex frequently refers to undeclared variables. Prior work on building a target language for a neural semantic parser also observed similar problems [18, 36]. For control flow statements, we can use function call distributions for for-loops and use the SQL-like syntax described above for if-statements.

In an interrogative sentence, the verb matters less (typically, the verb is just "is"), and there are still object and object modifiers. We can translate the object and the object modifiers similarly: "When is [the last time](subject) [I ordered from this restaurant](subject modifier)" `->Order.All().equals(field:.restaurant,value:Restaurant.Current()).sort(field:.date,ascending:false)[0].date`.

Human languages are flexible and do not always have to follow the exact grammar. Still, an LLM can automatically compose the NLPL features above to accommodate the user's request as long as the features are in the developer's program. The full grammar of NLPL is listed in Appendix A. We implemented the *DSL interpreter* module using the peggy[4] parser generator.

3.5 System Workflow

As shown in Figure 4 right, ReactGenie provides libraries that automatically perform actions during two phases of the development and usage process: 1) transpilation and initialization time and 2) runtime.

At transpilation and initialization time (Figure 4 right top), ReactGenie uses the annotations in the state code to generate LLM prompts containing class definitions and example parses for the semantic parser and response generator modules (Section 3.5.1). ReactGenie also reads the component wrapper code to determine the mapping between the components and the state objects for the input and output UI mapping modules (Section 3.5.2).

At runtime (Figure 4 right bottom), ReactGenie processes the user's multimodal voice and touch input. The voice part is translated to NLPL using the semantic parser, and the touch points are translated to the referred state instances using the input UI mapping module. Knowing the NLPL and the referred state instances, the NLPL interpreter can call the relevant methods and properties in the developer's state code to achieve the user's request. The NLPL interpreter records the final return value and the intermediary execution steps from executing each part of the composed method-chaining statement. ReactGenie further uses the final return value to generate text feedback for the user. It also uses the execution steps to graphically present the answer to a query-type request or the effect of an action-type request.

[4] https://peggyjs.org/

Note that *Transpilation* is a source-to-source translation process from the TypeScript the developer writes to Javascript that the machine executes. Typically, a TypeScript app is transpiled to JavaScript to run in a mobile app or a browser. However, during the transpilation process, the metadata, like typing and function parameter names, are removed. The metadata lost in transpilation is required by ReactGenie to create a large language model-based semantic parser/response generator and the UI mapping. Therefore, we built our transpliation plugins to use the developer's code before transpilation to generate the modules that ReactGenie uses at runtime.

3.5.1 Transpilation and Initialization for State Code. ReactGenie uses a custom transpiler plugin that generates extra metadata for `@GenieProperty`, and `@GenieFunction` of `DataClass` and `HelperClass`.

We use in-context learning to implement the semantic parser and the response generator. During initialization of the app, ReactGenie will load injected metadata from *state classes* (classes in state code) to generate a base prompt shared by both the semantic parser and the response generator. LLMs work by generating text continuations given a paragraph of previous text. The provided previous text is often called the *prompt*. By controlling the prompt, we change the information the LLM has access to and guide the LLM to do what we want (generate the corresponding NLPL of the user's command). ReactGenie's *generated prompt* contains two parts: 1) The *class definitions* contain all the `DataClass` and `HelperClass` method and property definitions with the implementation stripped out. It is rendered in a format similar to Swift syntax. 2) The *example parses* provided by the developer are also included as few-shot examples.

The user input is then appended to the generated prompt and used in the LLM-based semantic parser for NLPL translation. The *response generator* prompts the LLM with the generated prompt, the user input, the parsed NLPL, and the `description` of the return value from the execution of NLPL to produce a short text response.

We built the semantic parser using the OpenAI Codex model `code-davanci-2` and the response generator using the OpenAI GPT 3.5 model `text-davanci-3`.

3.5.2 Transpilation and Initialization for UI Code. At initialization time, we also process `GenieComponent` functions to save a mapping between the `GenieComponent` and the `GenieClass` that they are representing. We generate *input and output UI mapping* modules from this information. We monitor the bounding box of all `GenieComponents` for input mapping. When the user touches the screen while expressing a multimodal command, ReactGenie will use the bounding box information to determine which component the user is pointing to.

It is common for multiple UI components to cover the area where the user taps on the screen. For example, in Figure 1, all the `FoodThumbnail` components overlap with the `OrderItemView` components. ReactGenie allows the user to use their voice to disambiguate the reference: If the user mentions food, such as "this food" (`FoodItem.Current()`), or actions that can only be done with food, like "what is the price for this" (`FoodItem.Current().price`), ReactGenie will use the `FoodItem` object and vice versa. In the special

case where multiple components of the same type cover the tapped area, ReactGenie uses the one with the smallest bounding box.

Another common scenario is that if one object is clearly in the "foreground" of the graphical UI, the user may naturally refer to it as "this" without explicitly specifying the component via touch. So, when the user refers to a state class and either there is no touch point, or the touch point does not match any component representing that class, ReactGenie will use the largest component on the screen representing that class as the reference.

We also use `GenieComponent` to generate *output UI mapping* modules. We gather all the `GenieComponents` with supplied priority and title and group them by the state class they represent. When the result of the executed NLPL is a state class instance, ReactGenie enumerates through all the `GenieComponents` representing that class and renders the one with the highest priority.

There are two types of execution results. The first type is for query-type requests where the translated NLPL returns an instance that can be rendered by a `GenieComponent`. This is common when the user asks to either retrieve some data "what are my most recent orders from this restaurant?" or to perform some action with a clear result "what vegetarian food does this restaurant offer?". In that case, rendering the result on the screen would be intuitive. So, when the return value can be represented by a `GenieComponent`, ReactGenie will always find the highest priority `GenieComponent` and render it.

The second type is for action-type requests where the translated NLPL returns a value that cannot be rendered by a `GenieComponent`. For example, if the user asks to "add a hamburger to the cart" (NLPL: `Order.GetActiveCart().addItem([Food.Named("hamburger")])`, it would return `void` which cannot be rendered. For these actions, the return value is less important, and the user is more interested in the side effects of the action (e.g., the cart has been updated). In that case, ReactGenie will back trace the chained execution steps and find the last renderable result is the return value of `Order.GetActiveCart()` (i.e., an instance of a cart). ReactGenie will also check all the currently visible components to see if there is any that already represent the same instance of that cart on screen. ReactGenie would only render this result if the current page has no component representing the same instance. For example, when the user is already on a restaurant page where they can see an indicator of the number of items in the cart (e.g., the cart icon with a counter also represents the cart instance), it would be redundant to show the cart again. However, if the user is on the past order page where they cannot see any representation of the cart, it would be useful to show the cart to ensure the user understands the action being performed.

3.5.3 Runtime. Like normal React or React-Native apps, when users interact with buttons and visual controls, the app calls the corresponding methods to update data in the state instances. In turn, the state instances trigger the `GenieComponents` to update their UI.

As shown in the bottom right of Figure 4, the multimodal interactions are handled through developer modules (Section 3.3), the NLPL modules (Section 3.4), and the generated modules (Section 3.5.1), collectively. When the user touches the microphone button on the UI, ReactGenie starts listening to the user's voice command and intercepts all touch events on the screen. From this, we gather two inputs: the user's voice command and the touch point(s). We use speech recognition from Azure to transcribe the user's speech to text. The voice command transcript is then passed to the *semantic parser* module to generate the NLPL code. The touch point(s) are passed to the *input UI mapping* module to determine which component and state instance the user can refer to. Both pieces of information are then passed to the NLPL interpreter to execute the NLPL code with the corresponding relevant state instance. ReactGenie uses the methods and properties of the developer-provided *state classes* to execute the NLPL code. After the execution, we record both the final return value and the intermediate values during execution. ReactGenie uses the return value and the parsed DSL to generate a text response using the *response generator*. ReactGenie also passes execution steps to the *output UI mapping* module to determine whether and how to render the result on the screen. Finally, the text response and the rendered UI are used to generate *Feedback in Text* and *Content in UI*.

4 FRAMEWORK EVALUATION

We first evaluate the development framework by checking whether our design goals have been reached.

F-RQ1 How expressive is the ReactGenie framework? (**F1**)
F-RQ2 How much time is needed for *expert developers* to develop multimodal apps using ReactGenie compared with existing frameworks? (**F2**)
F-RQ3 How easy is it to learn and use ReactGenie to develop multimodal apps for *novice developers*? (**F3**)

4.1 Expressiveness of the Framework

To answer whether ReactGenie can support the expressiveness of mobile apps (**F-RQ1**), we built three example apps across three major categories of apps: food & drink, social networking, and business in different interface styles, as shown in Figure 5. The implementation statistics are shown in Table 2.

4.1.1 ReactGenieFoodOrdering. ReactGenieFoodOrdering is a food ordering app that allows users to order food from a restaurant. It allows users to browse menus, manage shopping carts, and check order history. For example, users can say "Reorder my last order", click on a food item and say "Add three of this to my cart", or click on the restaurant and say "Show me the menu of this restaurant". The app is comprised of 2689 lines of code, with only 88 (3%) related to building the multimodal UI. Note that every example parse provided by the developer takes four lines of code, and every `GenieClass`, `GenieFunction`, and `GenieProperty` annotation takes just one line of code.

4.1.2 ReactGenieSocial. ReactGenieSocial is a social networking app that allows users to post pictures, comment on pictures, and share pictures with friends. It allows users to browse, interact with, and share posts. For example, users can say "Show me posts from John", "Can you show me posts from Mark that have been liked before?", or click on the screen and say "Share this post with Emma". The app is comprised of 1034 lines of code, with only 49 (5%) related to building the multimodal UI.

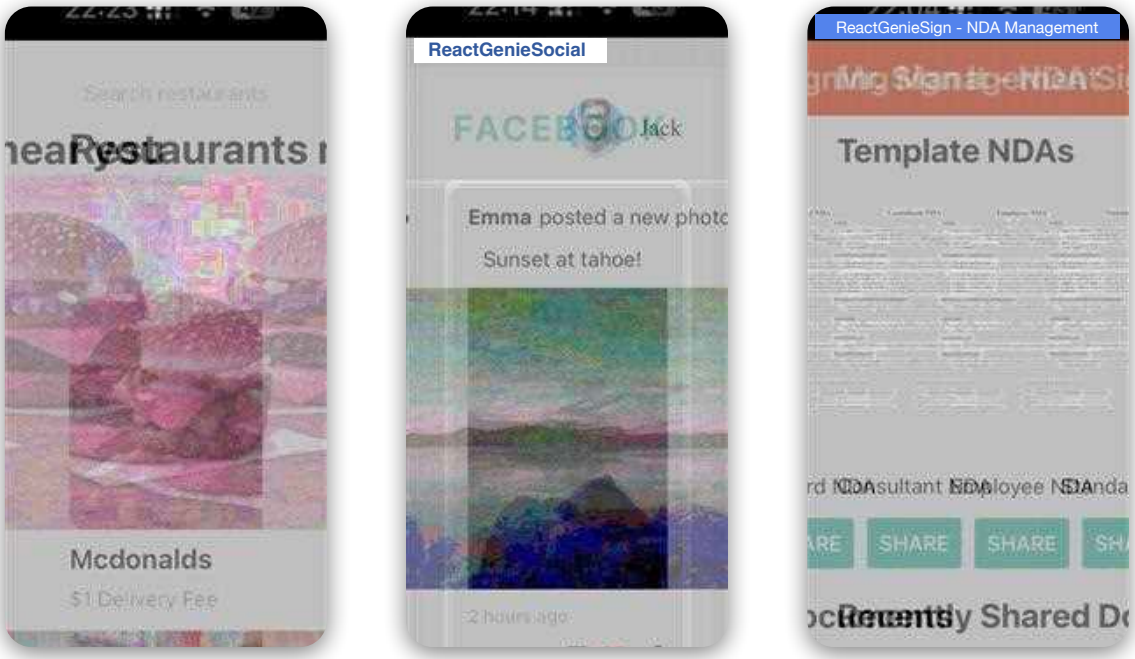

Figure 5: Example apps built with ReactGenie. Left: ReactGenieFoodOrdering, a food ordering app. Middle: ReactGenieSocial, a social networking app. Right: ReactGenieSign, a business app for distributing and collecting signed NDAs.

App Name	FoodOrdering	Social	Sign
`DataClass`	Order, FoodItem, Restaurant	Post, User, Message	User, Document, SignatureRequest
`HelperClass`	OrderItem		EmailAddress
`GenieComponent`	11	7	6
OtherComponents	8	2	3
`GenieFunction`	22	8	11
`GenieProperty`	18	10	16
State Code (lines)	835	449	421
Component Code (lines)	1854	585	446
Examples (count)	11	6	6

Table 2: Implementation statistics for demo apps. We listed all the `DataClass`, `HelperClass`, and the number of `GenieComponent` and `GenieFunction` used in the apps. We also listed the number of lines of code for the state and component code and the number of example parses provided for the voice parser.

4.1.3 ReactGenieSign. ReactGenieSign is a business app that manages NDAs and contracts. It allows users to create documents, share documents with clients for signing, and manage clients. For example, users can say "Show me the signature request from John", click on the email address and say "Only show me requests from this email", or click on the email address and say "Share the document in the most recent signature request to this email". The app is comprised of 867 lines of code, with only 51 (6%) related to building the multimodal UI.

4.1.4 Summary. Implementing the three apps in distinct domains that support a wide range of multimodal commands demonstrates the expressiveness of ReactGenie. While building these demo apps, we noticed that different UIs are naturally decomposed into components that represent different ReactGenie state instances, which made it easy to decompose the UI into `GenieComponents`. We were also able to lay out the graphical UI in the most appropriate way and then decompose the UI layout into individual components representing different state classes for UI mapping purposes. We also noticed that only a small fraction (5% on average) of the code must be written to handle multimodal interactions. This is particularly impressive since defining multimodal interactions can be intricate and typically requires substantial code to support.

4.2 Development Time: Expert Evaluation

To evaluate the development time required for ReactGenie apps (**F-RQ1**), we conducted an expert evaluation comparing the time to build apps with ReactGenie to the time required for building similar features with baseline tools.

4.2.1 Study Design. There is no readily available multimodal framework that adds multimodal capabilities to a multimodal app that has comparable capabilities to ReactGenie. Therefore, we used GPT-3 function calling[5] and React to implement the baseline app. GPT-3 function calling is a service supported by large language models that can convert natural language into function calls. We chose it as the baseline because a developer can use it to translate users' voice commands into function calls defined in the app, which makes it the closest off-the-shelf solution to help with the multimodal app development tasks that ReactGenie supports.

For the GPT-3 function-calling condition, we used a typical React-Redux architecture where there is no object-oriented state abstraction and the UI state is stored in a monolithic data store. For example, in the function-calling condition, the developer would implement an action on the monolithic data store called `RATE_FOOD(food_name:string,rating:number)` and pass the same function to the GPT-3.5-turbo endpoint as a function candidate. When the user says "Rate the hamburger five star", GPT returns a function call action that it wants to call `RATE_FOOD(hamburger,5)`, and the developer can execute the function for the monolithic state.

In the study, we asked an expert user[6] of ReactGenie to build a multimodal timer app (ReactGenieTimer, Figure 6 right). The timer app allows users to create timers and start/stop timers with voice and touch.

We asked the developer to first write the part of the app that is agnostic to ReactGenie (the boilerplate code), such as the React UI code, CSS files, and basic configurations. From there, we timed how long the expert developer finished the multimodal app with ReactGenie. From the same starting point, we also timed how long the expert developer could implement similar features with GPT-3 function calling.

4.2.2 Results. The implementation of both versions of the app starts with a boilerplate project that contains the framework-agnostic part of the code (337 lines of code). The expert developer took 45 minutes to complete the ReactGenie multimodal app and added 159 lines of code to complete the app. In comparison, it took the same developer 177 minutes to implement similar functionality with GPT-3 function calling and an additional 523 lines of code. Within that period of time, 52 minutes and 166 lines of code were spent on implementing the basic react-redux-based state management.

The additional code and time for GPT-3 function calling is due to the following:

(1) **The developer needs to provide the function signature to the model manually**, and when the model thinks a function call is necessary, the developer needs to call the corresponding function call.

(2) **The developer needs to build extra convenient functions for GPT-3.** Since GPT-3 can only call one function at a time, it has trouble executing actions like "Start the exercise timer" because that involves two steps: 1) retrieve the timer called exercise, and 2) start the retrieved timer. For GPT-3 to work, the developer has to implement a function that can start a timer by its name.

Even with the additional lines of code, the GPT-3 function calling version lacks a few significant benefits of the ReactGenie version:

(1) It **cannot support references by touch** due to a lack of the UI mapping module. For example, it cannot support "Start this timer." while tapping on a timer.
(2) It **cannot support rich commands** unless explicitly provided by the app developer. For example, it cannot support "Pause all timers with less than two minutes left."
(3) It **cannot navigate to the relevant page** after executing a command due to the lack of UI mapping. For example, when the user says "Start the cooking timer" while the cooking timer is not currently on the page, the app will not show the user the cooking timer visually.

The 3.9x time used and 3.3x lines of code to implement similar features in the baseline condition show that ReactGenie drastically reduced the implementation time for developers to build multimodal apps. In addition, the missing features in the GPT version also demonstrate the usefulness of ReactGenie for expressive multimodal app development.

4.3 Usability and Learnability: Developer Studies

To evaluate the usability and learnability of ReactGenie (**F-RQ3**), we conducted an IRB-approved user study asking novice developers to build multimodal applications with our framework. Considering that developers cannot complete multimodal applications within an acceptable time frame through direct API calls, our study was focused on the framework-specific part of the implementation to measure the usability of ReactGenie for building multimodal apps. We also evaluated developers' comprehension of the framework.

4.3.1 Study Design. The study was facilitated using a remote desktop to ensure all participants completed the coding tasks in the same environment. During the study, the experimenter introduced the study goals and explained study-related concepts such as multimodal apps since most participants did not have multimodal development experience. Then, the experimenter helped the developer connect to the remote experimental environment via video conferencing software.

The main study process contains two parts: the first, where developers learned ReactGenie and the second, where they built an app in ReactGenie. In the first part, participants familiarized themselves with ReactGenie by creating a multimodal counter application (ReactGenieCounter, Figure 6 left) guided by a tutorial. In the second part, participants were asked to leverage their knowledge from the example counter app development to construct a multimodal timer application (the same ReactGenieTimer app described in Section 4.2) independently. To make it feasible to complete the tasks within an acceptable amount of time and to ensure that participants

[5] https://openai.com/blog/function-calling-and-other-api-updates
[6] The expert developer was the first author of this paper.

ReactGenieCounter **ReactGenieTimer**

Figure 6: The ReactGenieCounter and ReactGenieTimer apps built in the developer study. The ReactGenieCounter app allows users to create counters and increment/decrement counters. The ReactGenieTimer app allows users to create/edit/delete timers and start/stop timers. Both apps support a variety of multimodal commands.

could focus on using the ReactGenie framework to implement multimodal features, we provided boilerplate code with the basic GUI implementation (similar to what was built by the expert developer in Section 4.2) for the two development tasks.

During the study, participants were required to complete a ReactGenie comprehension quiz and a post-study survey that recorded their demographic information and asked them to fill out the SUS usability scale [8] and the NASA-TLX cognitive load scale [31]. Finally, we have a short interview about their experience using the framework. We recorded the audio and the developer's screen during the entire process with the user's permission. Each user received a $60 gift card as compensation after the study.

4.3.2 Participants. We recruited developers with React development experience to ensure they could handle the non-multimodal-related coding tasks beyond what ReactGenie is designed for. Therefore, we designed a quiz with seven programming questions and deployed a recruitment screener. These questions cover basic React skills (such as how to use a React component, and how to manipulate state in React) and basic object-oriented programming skills (e.g., what is `this` pointer, and how to write class constructors). Only developers who answered at least five of the seven questions correctly were invited to participate in the study.

We recruited 12 participants (10 males and two females) in the remote user study by distributing the recruitment screener link using a convenience sample. Our participants include student developers and professional developers with an average age of 23.8 (σ = 3.43). All participants had React development experience; ten developers also had TypeScript development experience. The most experienced developer had seven years of React development experience.

4.3.3 Results. All participants successfully built the applications within 150 minutes. The average completion time was 109.7 minutes (σ = 24.19), with 42.3 minutes (σ = 14.90) for the first phase and 67.3 minutes (σ = 17.52) for the second phase. This shows that developers with React and object-oriented programming experience can quickly learn and use the ReactGenie framework, illustrating the framework's ease of learning and high usability. The user with the shortest task completion time needed only 82 minutes to complete both tasks.

Almost all participants (11/12) answered the seven post-task quiz questions correctly. A single participant got one question wrong related to predefining the interface. This shows that developers could develop a multimodal app and correctly understand how the ReactGenie code they wrote corresponds to the multimodal features, which suggests the design of the ReactGenie framework is easy to understand. The main parts that participants found relatively difficult to understand are the GUI declarations and providing example parses. Participants said it took them more time to understand the `GenieComponent` function and how to bind data types to the corresponding interfaces. Regarding example editing, participants thought it was hard to determine what should be included in the examples. Although participants found these parts difficult to understand, they could use ReactGenie smoothly after learning and testing their apps. Participant 11 said, "ReactGenie incorporates

natural language to UI building, and it is not hard to do that when programming."

We evaluated ReactGenie's usability using a part of the seven-point SUS scale. ReactGenie received high SUS scores in terms of ease of use ($\mu = 6.16$, $\sigma = 0.58$), enjoyable ($\mu = 6.08$, $\sigma = 0.79$), intuitive ($\mu = 6.08$, $\sigma = 0.90$), and willing to use ($\mu = 6.08$, $\sigma = 0.79$). Participant 10 says, "(ReactGenie was) easy to pick up if you know React." At the end of the experiment, participant 12 commented that he enjoyed programming with ReactGenie and asked if there was any library he could use to access ReactGenie in his daily programming. All participants agreed that ReactGenie is easy to use (medium 6 out of 7 points Likert scale), and all participants were willing to use ReactGenie to build real-life multimodal applications (medium 6 out of 7 points Likert scale). The average NASA-TLX score for the overall study was 21.99 out of 100 (the lower, the better), showing the practicality and low usage burden to program with ReactGenie.

5 INTERACTION EVALUATION

We evaluated the interaction provided by the ReactGenie framework through two aspects: the performance of the generated neural semantic parser and the overall user experience of a ReactGenie app.

5.1 Parser Performance

To understand how well the ReactGenie parser works with information extracted from the developer's code, we elicited commands from crowd workers for the ReactGenieFoodOrdering app and tested our parser. Specifically, we would like to know:

RQ1 What percentage of the commands 1) is achievable with a single UI interaction on screen, 2) fall into the three targeted interactions mentioned in Section 3.1, or 3) are out of scope of ReactGenie.

RQ2 How accurate the parser is when parsing commands in the targeted interactions.

5.1.1 Elicitation. We wanted to obtain multimodal commands that users may use in a real-world scenario. We adopted a similar method as described as Cloudlicit [11]. We provided the user with three screenshots (restaurant listing page, restaurant menu page, and past orders page) of the US's two most popular food ordering apps: DoorDash and UberEats. In our pilot study, we found that many participants' thoughts on what they can do in these apps are limited to what's on-screen and what they think the current generation of voice assistants can do. Therefore, we showed the final study participants 12 videos randomly, containing four videos for each of the three categories of interactions being executed on a different app (home page of the Apple app store). Among these 12 videos, we also ensured half involved only voice and the other half contained voice and touch.

We recruited 50 participants from Prolific, a crowdsourcing platform. We used the balanced sample options when finding participants, so we had 25 female and 25 male participants. The age range of the participants was 20 to 79, with a median age of 29. The survey took approximately five minutes, and we paid $2 for each participant.

From these 50 participants, we obtained 300 commands. We filtered out 12 unclear or unrelated responses to the survey. For example, one participant wrote "various good foods to order or view that can be good" as a command. After filtering, 288 commands remained in the dataset.

*5.1.2 **RQ1**: Percentages of categories of commands.* We classify the commands into three categories:

(1) **Simple UI interaction**: The command can be achieved with a single UI interaction on screen. For example, *"Look at the Curry Up Now Menu"* when the restaurant is visible on screen.
(2) **Within the three targeted interaction categories**: The command falls into the three targeted interactions mentioned in Section 3.1. For example, one participant wrote *"Order me two big macs and large fries from Mcdonald's for pickup."* With a GUI, this command would typically be achieved via multiple taps to find the restaurant, add the foods, and configure the delivery options.
(3) **Out of scope of ReactGenie**: The command is out of scope of ReactGenie. For example, "How do I repeat past orders?". ReactGenie tries to help people complete complex tasks, but it does not have built-in knowledge about how to use the UI of the app.

Two researchers collaboratively labeled 30 commands to get a rubric for the rest of the commands. They then labeled the rest of the commands (258 commands) using the rubric separately. Both labelers labeled the same label for 224 commands and different labels for 34 commands. Because the labels have a skewed distribution, we used Gwet's AC1 [30] to measure the inter-rater reliability. The AC1 score is 0.83, which means the labels are highly consistent. They resolved the disagreement and got a final label for each command.

From this analysis, we found that 100 of the elicited commands were simple UI interactions, 172 commands fell into the three targeted interactions, and 16 commands were out of the scope of ReactGenie.

This shows that users can come up with tasks beyond just simple UI interactions even when the type of multimodal interfaces that ReactGenie supports are not available in commercial apps. It may also hint at user interest in the types of interactions that we propose here.

*5.1.3 **RQ2**: Accuracy of the parser.* We tested the parser on the 172 commands that fall into the three targeted interactions. We ran the parser based on the ReactGenieFoodOrdering app and read the generated NLPL to see if the parses are correct. While working on labeling the correctness, we also noticed that many of the commands are not supported by our simple demo app, e.g., ReactGenieFoodOrdering only knows delivery fees for different restaurants, but not estimated delivery times. So we also labeled whether the feature that the command tries to use is supported by ReactGenieFoodOrdering.

Our analysis showed that 101 commands are supported by ReactGenieFoodOrdering, and 71 commands are not supported. Some features that are missing from ReactGenieFoodOrdering are 1) toppings/customization of a food item; 2) reviews of a restaurant or a food item; and 3) delivery time estimates for restaurants.

From the 101 commands supported by ReactGenieFoodOrdering, we found that 91 commands are parsed correctly by the parser, and ten are not. Therefore, on this dataset, the parser has an accuracy of 90%. A parser accuracy of 90% should be considered very high compared to prior work on neural semantic parsers' accuracy on compound commands [19]. Seven of the ten incorrect parses would result in syntax errors (such as used `.first()` rather than NLPL supported `[0]`) or runtime errors (such as ordering the last meal from "A" has been translated to order the food called "A", thus the system will not be able to find "A" as a food). Three of the incorrect parses would be helpful but not give the desired behavior. For example, "Find me the closest pizza restaurant" was translated to "find the closest restaurant". None of the errors resulted in behavior completely different from the user's expectations.

We also looked at the 71 commands that are not supported by ReactGenieFoodOrdering. These commands mention features that are not in the ReactGenieFoodOrdering app. To our surprise, the parser also generated sensible NLPL for the majority (38) of these commands. The ReactGenie parser approximates the request command with available features in the app for 24 of these commands. For example, ReactGenie parser generates `Restaurant.GetRestaurant(name:"pizzahut").getFoodItems().between(field:.price,from:0,to:5)` for the command *"What deals does pizza hut have?"*. In this case, the parser approximates deals with food items that are less than 5 USD. For 14 commands, the parser would generate function calls and property accesses that are not supported by the app. For example, the parser parsers *"What time does Chipotle open?"* to `Restaurant.GetRestaurant(name:"Chipotle").openingTime`. In this case, ReactGenieFoodOrdering does not have the property `openingTime` for restaurants, but the parser is still capable enough to generate a sensible NLPL. In the future, the ReactGenie runtime can leverage this information to inform the user of the missing property and potentially even suggest the developer add common missing features to the app.

There were 33 unsupported commands that were not parsed correctly by the parser. Some of them are due to the parser generating ungrammatical NLPL, and others use incorrect properties and methods. For example, the parser parses *"Find restaurants that deliver in less than 25 minutes."* to `Restaurant.All().matching(field:.deliveryFee,value:<25)`. In this case, ReactGenieFoodOrdering does not know the estimated delivery time of restaurants, but the correct parsing should be `Restaurant.All().between(field:.deliveryTime,from:0,to:25)`.

The results show that ReactGenie parser is a reasonably good implementation for parsing natural language commands to NLPL using only information extracted from the shared logic code and the few-shot examples provided by the developer.

Another interesting metric is that 104 of the 172 commands contain at least one touch point, but there are only 18 cases where these touch points are required to execute the command. In many of these commands, the user taps relevant objects, hoping that it would help the system understand. For example, they would tap on the "Restaurant" menu bar while saying "Show me a pizza restaurant near me." Another interesting observation is that when they referred to objects on screen, they often would not use a reference term like "this" or "that." Instead of saying *"Reorder this order"*, the participant would say *"Reorder my Mendocino Farms order from Thursday."* This shows a potential opportunity to improve the semantic parser by always adding the touch context even when it seems unnecessary.

5.2 Usability of Supported Interactions

We conducted a usability study with the ReactGenieFoodOrdering app to understand if the generated multimodal UIs are useful for end users. We measured the performance of the multimodal UIs in terms of the time it takes to complete a task, the cognitive load, and the usability of the experience when using the app compared to the same app limited to using only the GUI.

5.2.1 Study Design. In the study, we asked participants to complete a set of tasks using two variants of the ReactGenieFoodOrdering app, one generated by ReactGenie and one limited to only the GUI. We used a within-subjects design, where each participant completed the same tasks using both variants of the app. For each app variant, we first teach the participant how to use the app using one training task. Specifically, for the ReactGenie condition of the app, we explained that in addition to typical touch-only interaction, they can also tap the microphone button to initiate a speech + gesture command when they want to. We then asked them to complete two test tasks with the variant. After completing the two tasks, we asked them to complete a survey about their cognitive load using the system (using NASA-TLX [31]) and the usability of the experience (using SUS [8]). At the end of the study, we asked the participants about their subjective preferences between the two variants of the app and their reasons for their preferences.

We designed one training task and two test tasks for each variant of the app. The training tasks are to order the cheapest food item from the menu of two different restaurants. The test tasks are reordering an order from two different days (today or yesterday), and finding the most recent order containing two different items. When presenting these tasks, we described a scenario, what we wanted them to do, and the expected outcome (order placed screen or a certain screen showing a past history order). We counterbalanced the order of the two apps and the order of the three tasks.

5.2.2 Participants. We recruited 16 participants, aged 18–30, with a median age of 23. Eight of our participants are female, six are male, one stated other, and one prefers not to say. One of our participants uses food ordering daily, two use it weekly, five use it monthly, seven use it a few times per year, and one rarely or never uses it. All of our participants use graphical mobile interfaces daily. Two of our participants use voice interfaces daily, four use them weekly, two use them monthly, four use them a few times per year, and four rarely or never use them. The study took about 30 minutes to complete, and we compensated each participant with a 15 USD Amazon gift card for their time.

5.2.3 Results. We computed the time it took to complete each task using the graphical UI and the multimodal UI (see Figure 7). The average time it took to complete each task using the graphical UI was 63.6 seconds, while the average time it took to complete each task using the multimodal UI was 33.6 seconds. We used a paired t-test and found that the difference is statistically significant ($p = 0.0004, t = 3.955$).

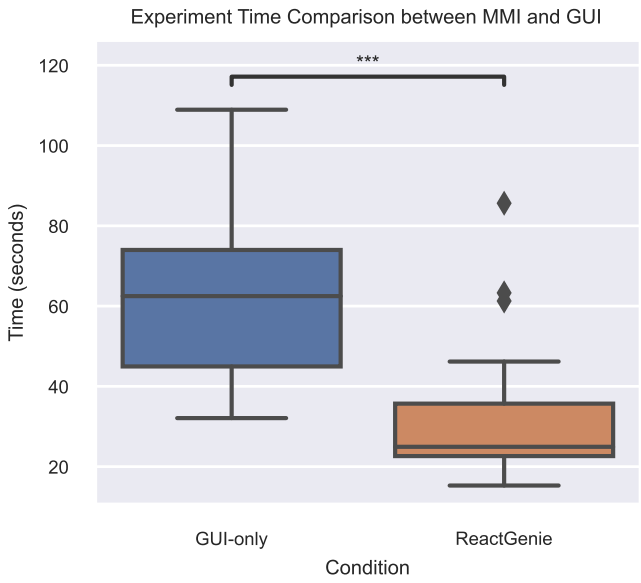

Figure 7: Users can complete common tasks faster ($p = 0.0004, t = 3.955$) with the multimodal interface (MMI) app built with ReactGenie compared with a baseline GUI app.

We compared NASA-TLX average scores between the two conditions (see Figure 8 left). The average NASA-TLX score for the graphical UI is 34.5, while the average NASA-TLX score for the multimodal UI is 24.6 (note: lower is better). We used a Wilcoxon test and found that the difference is statistically significant ($p = 0.013, z = 21$).

We compared the average SUS scores between the two conditions (see Figure 8 right). The average SUS score for the graphical UI is 63.3, while the average SUS score for the multimodal UI is 73.0 (note: higher is better). We used a Wilcoxon test and found that the difference is statistically significant ($p = 0.031, z = 22$).

11 out of 16 of our participants preferred the ReactGenie generated multimodal UI over the graphical UI. For participants who preferred the multimodal UI, the most common reason was that it was easier to use (P4, P8, P13, P16). P2 mentioned that they would prefer to use a mix of both in the real world, which is well supported by ReactGenie. P6 mentioned that the multimodal UI could be especially useful when they are unfamiliar with the app. P12 mentioned that the multimodal commands allowed them to do more complex tasks with a clear path rather than searching and finding out how to do so in the graphical UI. For participants who preferred the graphical UI, the most common reason was that the speech recognition was not accurate (P5, P7, P14). P9 and P11 mentioned that they generally do not use voice interfaces.

5.2.4 Discussion. The results of our usability study show that the multimodal UIs generated by ReactGenie are more efficient, have a lower cognitive load, and have higher usability compared to the corresponding graphical UI versions. These findings suggest that the ReactGenie system is successful in generating multimodal UIs that enhance the user experience, making it easier and more efficient for users to complete tasks. All participants, when using the ReactGenie variant of the app, used both typical touch interaction for simple navigation and browsing and touch (optional) + speech interaction for more complex inputs. The combination of graphical and voice interfaces allows users to take advantage of the strengths of each modality, resulting in a more streamlined and enjoyable experience.

6 DISCUSSION

In this section, we will characterize the properties of the multimodal interaction supported by ReactGenie, and discuss the limitations, future work, safety, and implications of ReactGenie.

6.1 Properties of Multimodalities in ReactGenie

We can characterize the properties of the ReactGenie interactions using the framework proposed by Coutaz et al. [22]. ReactGenie's voice + (optional) touch actions are implemented using the same functions used for graphical user interfaces. Therefore, all voice and touch actions are equivalent, and almost no actions belong to the assignment category, meaning that almost no actions can only be performed using a specific modality. Note that the functions only interact with the state. Commonly minute actions such as scrolling will not be included in part of the state, so actions like "Scroll to here" will not be supported by the voice + touch commands. This is intentional, as scrolling is likely more efficient using touch than voice + touch. However, in ReactGenie, developers can expose anything as part of the state. In an e-book or a map application, where the current read position/map position is a crucial part of the experience, developers can choose to expose the position as part of the API. The user's command "scroll to here" can be translated to `ReadingPosition.SetPosition(position :ReadingPosition.Current())`.

For redundancy, ReactGenie apps will only accept touch commands when the microphone button is not activated and will only accept voice + touch commands when the microphone button is activated to achieve partial redundancy. For complementarity, ReactGenie primarily supports voice for actions and touch for deictic gestures for reference. After the user clicks the microphone button, the order of touch and voice does not matter, the user can tap first then speak, speak first then tap, or tap while speaking. The voice and touch mode will end automatically after receiving no new words or touches for 0.5 seconds.

6.2 Limitations and Future Work

ReactGenie is the first attempt at integrating multimodal development into the modern declarative GUI development process. It provides a familiar workflow, allows the reuse of state code and UI, and can understand rich multimodal commands. However, it is far from perfect. There are three directions that future work can improve on 1) better voice interfaces, 2) better developer support, and 3) support for more modalities.

6.2.1 Better Voice Interfaces. ReactGenie accepts the user's voice and touch input and generates text and GUI output based on the result. We currently provide text but not voice feedback, which is easy to change by using a commercial text-to-speech module. However, a more significant area of improvement is in maintaining natural language context. For example, if the user says, "What

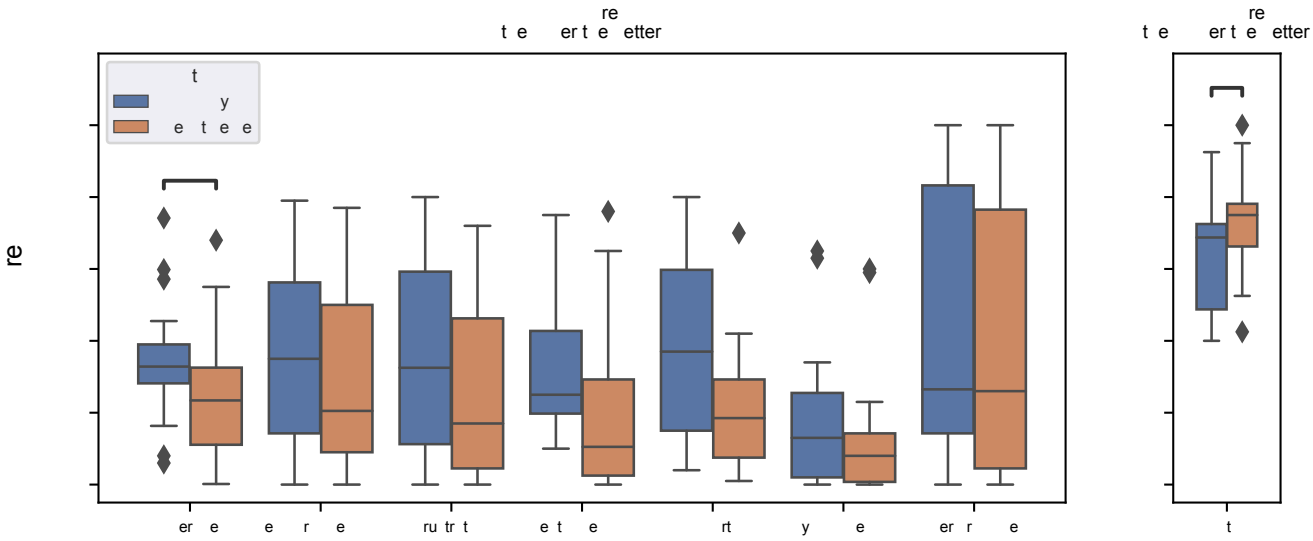

Figure 8: Cognitive load (left) and usability (right) of the GUI vs. the ReactGenie multimodal UI.

is the best pizza restaurant?" and then asks, "What about Chinese food?", the system should be able to understand that the user is asking about the *best* Chinese food instead of any Chinese food restaurant. Note that ReactGenie can handle some conversations gracefully by using the current UI context as the context for the next command. An example would be the user saying, "Find me the cheapest hamburger at McDonald's" and then asking, "Order one of that" (NLPL: `Order.GetActiveCart().addItems([FoodItem.Current()])`). ReactGenie would present the food item after the first command, and when the user says the second command, ReactGenie would know that the user is referring to the food item presented in the GUI.

Another way to improve the reliability of the generated interfaces is to better leverage multimodal commands for disambiguation. As shown in Section 5.1.3, many of our elicited commands include redundant information from voice and the GUI. Future work can leverage this redundancy and provide extra GUI context to the semantic parser to further push the parser's accuracy closer to 100%. For example, a potential path is to improve ReactGenie runtime to extra GUI context and enhance the neural semantic parser through prompt engineering to allow it to process the extra GUI context.

6.2.2 Better Developer Support. Although ReactGenie provides a customizable and easy way of programming multimodal apps, it can still be improved. One area that we see as a potential improvement is to reduce the number of examples necessary and to increase the effectiveness of the example parses. The majority of these examples are there for teaching the parser how to generate syntactically correct NLPL code. However, given we have the interpreter, we can potentially use it as an example generator to teach the parser how to generate syntactically correct NLPL code, similar to the method used in SEMPRE [13] or Genie [19]. Another route is to fine-tune the Codex model with the NLPL code generated by the interpreter so that the interpreter can generate syntactically correct NLPL code with fewer examples.

Future extensions to the ReactGenie framework can also help developers identify potential voice commands that the user may want to say. Using ReactGenieFoodOrdering as an example, its API only supports 59% of the commands that we elicited from crowd workers. Some top categories of unimplemented commands are about delivery time (mentioned in 8 commands), food customization options (8), discount/deal information (7), pickup/delivery support of restaurants (6), food types (e.g., vegetarian or vegan) (6), and calorie/health/allergy information (6). If we can implement these commands, we can potentially reduce the number of unsupported commands by more than 50%. Future work can consider embedding elicitation studies directly into the app development cycle, or the framework could record unsupported commands from actual users and use this data as feedback to the development team to help improve the system after the initial deployment.

6.2.3 Support More Modalities. As stated in Section 3.1, ReactGenie is targeted at gesture and voice interactions, and it is optimized for deictic gestures used for reference. Currently, ReactGenie can support complex gestures in the regular GUI provided other gesture interaction frameworks, but not simultaneous voice + complex gesture interactions. Future frameworks can improve on supporting more diverse gestures and modalities beyond these two categories.

Gestures and modalities other than voice can also be used to indicate the actions that the user wants to perform. This can be supported by summarizing what happened using other gestures in language and presenting them along with the user's speech commands. For example, suppose a user says, "Animate this box," and performs a move while rotating and gesturing on the screen. In that case, we can present `voice:animatethisbox` and `gesture :rotateandmovefromx1,y1,tox2,y2` to the large language model and ask the model to generate NLPL for this animation. (`x1,y1` is

the coordinate of the gesture start coordinate, and the `x2,y2` is the gesture end coordinate.)

Another way gestures can be integrated is through a new class called `Gesture` provided to the ReactGenie system to indicate the gesture trajectory that the user performed. For example, if the user says "Deploy soldiers along this path" in a QuickSet-like [21] system, ReactGenie can translate it to `Solders.DeployOnPath(path :Path.FromGesture(gesture:Gesture.Current()))`.

Through a combination of the aforementioned methods and utilization of the gesture recognition/modality understanding components of prior multimodal interaction libraries [32, 49], future work can combine the flexibility and expressiveness of ReactGenie with different modalities.

6.3 Takeaways for Future UI Frameworks

ReactGenie demonstrated a feature-first, rather than interaction-first, way of implementing multimodal apps. ReactGenie lets developers do what they know best: implement the features the users want, and the framework intelligently takes care of users' diverse interactions using their specific way of making commands. This is similar to modern graphical UI frameworks that, rather than directly passing the user's touch point and letting developers implement the rest, ask developers to provide functions and pages at a feature level. The graphical UI framework handles how to render and convert user touch points and typical gestures to the corresponding function calls. This was not easy to achieve previously for multimodal interactions. ReactGenie made it possible because of the power of LLM's language-parsing capability and the expressiveness of NLPL. Future frameworks can build on this idea to support more modalities and be more adaptive to the user's interaction context.

Other researchers in intelligent human interfaces can also take advantage of how ReactGenie integrates LLMs into the human interaction loop. Rather than asking developers to call LLMs in their app, which requires them to understand prompting techniques, ReactGenie automatically generates an interface to the LLM as a programming framework. This program-LLM interface provides the available functionality to the LLM and allows the LLM to call the compound functionality of the program. Other intelligent human interfaces, such as conversational agents or adaptive interfaces, can also learn from this technique. They can automatically generate an interface from the developer's code that exposes available features and relevant context to LLMs and allows LLMs to use a programming language (maybe NLPL) to perform their job.

6.4 Safety and Implications

ReactGenie uses a machine learning model to understand the users' commands. This may introduce safety issues when the wrong command is interpreted and executed. In our evaluation, a pleasant finding is that wrongly parsed commands are either not executable or still helpful towards reaching the user's intended goal. Further risks can be reduced by having a more accurate semantic parser.

Also, compared with an end-to-end natural language assistant like ChatGPT[7], ReactGenie allows more control over the *presented information* and *performed actions*. ReactGenie's framework only processes and shows information in the developer's provided state code and can reduce hallucinated information. One particular case of error in our testing was when the user asked for the delivery time, but because the app does not support delivery time estimation, ReactGenie returned the delivery fee instead. In this case, the text feedback mechanism can be used to inform the user of the information that is returned. In the future, an error correction mechanism would be useful for the user to report the error, and this may allow the developer to fix it.

For performed actions, ReactGenie gives text feedback and renders the related UI elements to ensure the user is aware of the command being executed, so when there is an error, the user can easily identify and recover from it. A design decision we made while creating the three demo apps is not to expose non-recoverable actions to voice. For example, in the ReactGenieFoodOrdering app, the user can browse items, add items to the cart, and go to the checkout page via voice, but placing the order will only present the checkout page and require the user to click the "Place Order" button to place the order. This way, the irreversible action is only triggered through the GUI, with little room for error. It should be strongly recommended to developers using ReactGenie to either not expose (via Genie annotations) functions that would create an irreversible effect, such as payment-related or account management functions, or add a confirmation stage via graphical user interfaces or explicit voice commands.

Another implication of ReactGenie is the possible negative social implications of noisy multimodal interaction. ReactGenie encourages users to use voice and touch to quickly achieve their goals without going through multiple UI actions and exploration steps. The benefit of ReactGenie comes from the expressiveness of voice and touch, but voice interfaces may not always be appropriate. One possibility is to explore silent voice interfaces like those presented by Denby et al. [26] that can be used in public spaces.

7 CONCLUSIONS

Commercial user interfaces have stagnated with the GUI for more than a decade. Although these GUIs work well for communicating exact information (e.g., from a menu) and binary actions (e.g., using a button), they are not expressive enough to communicate and collect rich multimodal information, such as the way a waiter or waitress can obtain a person's order from a restaurant menu. ReactGenie attempts to break that UI stagnation by enabling developers to create multimodal UIs that allow for more expressiveness than traditional GUIs, with little additional programming effort. ReactGenie accomplishes this first by introducing an interaction programming paradigm where the interaction logic is better separated from user interface implementation and second by using a powerful natural language understanding module that leverages the capabilities of LLMs to execute code in the interaction logic. In this paper, we demonstrated the easy adoption of the ReactGenie framework for developers and tested the expressiveness, usefulness, and accuracy of the resulting multimodal apps with end-users. In the future, by introducing developer tools based on frameworks like ReactGenie, and the research on the multimodal interactions these tools enable, we hope to see humans communicating with computers more expressively and more easily.

[7] https://openai.com/blog/chatgpt/

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 Alfred P. Sloan Foundation, and the Verdant Foundation 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] [n. d.]. About App Development with UIKit. https://developer.apple.com/documentation/uikit/about_app_development_with_uikit. Accessed on 2023-04-05.
[2] [n. d.]. Describing the UI. https://react.dev/learn/describing-the-ui. Accessed on 2023-04-05.
[3] [n. d.]. Introduction to declarative UI. https://docs.flutter.dev/get-started/flutter-for/declarative. Accessed on 2023-04-05.
[4] [n. d.]. Pinia | The intuitive store for Vue.js. https://pinia.vuejs.org/. (Accessed on 04/05/2023).
[5] [n. d.]. React Redux | React Redux. https://react-redux.js.org/. (Accessed on 02/27/2024).
[6] [n. d.]. Redux - A predictable state container for JavaScript apps. | Redux. https://redux.js.org/. (Accessed on 04/04/2023).
[7] [n. d.]. SwiftUI. https://developer.apple.com/xcode/swiftui/. Accessed on 2023-04-05.
[8] 1996. SUS: A 'Quick and Dirty' Usability Scale. In *Usability Evaluation In Industry*. CRC Press, 207–212. https://doi.org/10.1201/9781498710411-35
[9] 2009. Human-Computer Interaction. https://doi.org/10.1201/9781420088861
[10] 2023. GitHub - facebookarchive/flux: Application Architecture for Building User Interfaces. https://github.com/facebookarchive/flux. (Accessed on 04/04/2023).
[11] Abdullah X. Ali, Meredith Ringel Morris, and Jacob O. Wobbrock. 2019. Crowdlicit: A System for Conducting Distributed End-User Elicitation and Identification Studies. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3290605.3300485
[12] Sean Andrist, Dan Bohus, Ashley Feniello, and Nick Saw. 2022. Developing Mixed Reality Applications with Platform for Situated Intelligence. In *2022 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*. IEEE. https://doi.org/10.1109/vrw55335.2022.00018
[13] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic Parsing on Freebase from Question-Answer Pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, 18-21 October 2013, Grand Hyatt Seattle, Seattle, Washington, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*. ACL, 1533–1544. https://aclanthology.org/D13-1160/
[14] Dan Bohus, Sean Andrist, Ashley Feniello, Nick Saw, Mihai Jalobeanu, Patrick Sweeney, Anne Loomis Thompson, and Eric Horvitz. 2021. Platform for Situated Intelligence. https://doi.org/10.48550/ARXIV.2103.15975
[15] Richard A. Bolt. 1980. "Put-that-there": Voice and gesture at the graphics interface. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques - SIGGRAPH '80*. ACM Press. https://doi.org/10.1145/800250.807503
[16] Stephen Brewster, Joanna Lumsden, Marek Bell, Malcolm Hall, and Stuart Tasker. 2003. Multimodal 'eyes-free' interaction techniques for wearable devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/642611.642694
[17] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. https://doi.org/10.48550/ARXIV.2005.14165
[18] Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S. Lam. 2017. Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee. https://doi.org/10.1145/3038912.3052562
[19] Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S. Lam. 2019. Genie: a generator of natural language semantic parsers for virtual assistant commands. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. https://doi.org/10.1145/3314221.3314594
[20] P. Cohen, D. McGee, S. Oviatt, L. Wu, J. Clow, R. King, S. Julier, and L. Rosenblum. 1999. Multimodal interaction for 2D and 3D environments [virtual reality]. *IEEE Computer Graphics and Applications* 19, 4 (1999), 10–13. https://doi.org/10.1109/38.773958
[21] Philip R. Cohen, Michael Johnston, David McGee, Sharon Oviatt, Jay Pittman, Ira Smith, Liang Chen, and Josh Clow. 1997. QuickSet: multimodal interaction for distributed applications. In *Proceedings of the fifth ACM international conference on Multimedia - MULTIMEDIA '97*. ACM Press. https://doi.org/10.1145/266180.266328
[22] Joëlle Coutaz, Laurence Nigay, Daniel Salber, Ann Blandford, Jon May, and Richard M. Young. 1995. *Four Easy Pieces for Assessing the Usability of Multimodal Interaction: The Care Properties*. Springer US, 115–120. https://doi.org/10.1007/978-1-5041-2896-4_19
[23] Deborah Dahl, Paolo Baggia, and Ken Rehor. 2003. *Multimodal Architecture and Interfaces*. Technical Report NOTE-mmi-arch-20031020. W3C. https://www.w3.org/TR/mmi-arch/
[24] Deborah A. Dahl. 2013. The W3C multimodal architecture and interfaces standard. *Journal on Multimodal User Interfaces* 7, 3 (apr 2013), 171–182. https://doi.org/10.1007/s12193-013-0120-5
[25] Adrian A. de Freitas, Michael Nebeling, Xiang 'Anthony' Chen, Junrui Yang, Akshaye Shreenithi Kirupa Karthikeyan Ranithangam, and Anind K. Dey. 2016. Snap-To-It: A User-Inspired Platform for Opportunistic Device Interactions. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/2858036.2858177
[26] B. Denby, T. Schultz, K. Honda, T. Hueber, J.M. Gilbert, and J.S. Brumberg. 2010. Silent speech interfaces. *Speech Communication* 52, 4 (April 2010), 270–287. https://doi.org/10.1016/j.specom.2009.08.002
[27] Michael H. Fischer, Giovanni Campagna, Euirim Choi, and Monica S. Lam. 2021. DIY assistant: a multi-modal end-user programmable virtual assistant. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM. https://doi.org/10.1145/3453483.3454046
[28] Divyansh Garg. 2023. Multi on. https://multion.ai/
[29] Andy (Steve) De George and Alex Buck2. 2023. What is windows forms - windows forms .NET. https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-7.0
[30] Kilem Li Gwet. 2008. Computing inter-rater reliability and its variance in the presence of high agreement. *Brit. J. Math. Statist. Psych.* 61, 1 (may 2008), 29–48. https://doi.org/10.1348/000711006x126600
[31] 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 (oct 2006), 904–908. https://doi.org/10.1177/154193120605000909
[32] Lode Hoste, Bruno Dumas, and Beat Signer. 2011. Mudra: a unified multimodal interaction framework. In *Proceedings of the 13th international conference on multimodal interfaces*. ACM. https://doi.org/10.1145/2070481.2070500
[33] Michael Johnston, John Chen, Patrick Ehlen, Hyuckchul Jung, Jay Lieske, Aarthi Reddy, Ethan Selfridge, Svetlana Stoyanchev, Brant Vasilieff, and Jay Wilpon. 2014. MVA: The Multimodal Virtual Assistant. In *Proceedings of the 15th Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGDIAL)*. Association for Computational Linguistics. https://doi.org/10.3115/v1/w14-4335
[34] Runchang Kang, Anhong Guo, Gierad Laput, Yang Li, and Xiang 'Anthony' Chen. 2019. Minuet: Multimodal Interaction with an Internet of Things. In *Symposium on Spatial User Interaction*. ACM. https://doi.org/10.1145/3357251.3357581
[35] Glenn E. Krasner and Stephen T. Pope. 1988. A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. *J. Object Oriented Program.* 1, 3 (aug 1988), 26–49.
[36] Monica S. Lam, Giovanni Campagna, Mehrad Moradshahi, Sina J. Semnani, and Silei Xu. 2022. ThingTalk: An Extensible, Executable Representation Language for Task-Oriented Dialogues. https://doi.org/10.48550/ARXIV.2203.12751
[37] James A. Landay and Brad A. Myers. 1993. Extending an existing user interface toolkit to support gesture recognition. In *INTERACT '93 and CHI '93 conference companion on Human factors in computing systems - CHI '93*. ACM Press. https://doi.org/10.1145/259964.260123
[38] Gierad P. Laput, Mira Dontcheva, Gregg Wilensky, Walter Chang, Aseem Agarwala, Jason Linder, and Eytan Adar. 2013. PixelTone: a multimodal interface for image editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/2470654.2481301
[39] Minkyung Lee and Mark Billinghurst. 2008. A Wizard of Oz study for an AR multimodal interface. In *Proceedings of the 10th international conference on Multimodal interfaces*. ACM. https://doi.org/10.1145/1452392.1452444
[40] Toby Jia-Jun Li and Oriana Riva. 2018. Kite: Building Conversational Bots from Mobile Apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM. https://doi.org/10.1145/3210240.3210339
[41] David L. Martin, Adam J. Cheyer, and Douglas B. Moran. 1999. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence* 13, 1-2 (jan 1999), 91–128. https://doi.org/10.1080/088395199117504
[42] Marilyn Rose McGee-Lennon, Andrew Ramsay, David McGookin, and Philip Gray. 2009. User evaluation of OIDE: a rapid prototyping platform for multimodal interaction. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. ACM. https://doi.org/10.1145/1570433.1570476

[43] B.A. Myers, D.A. Giuse, R.B. Dannenberg, B.V. Zanden, D.S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. 1990. Garnet: comprehensive support for graphical, highly interactive user interfaces. *Computer* 23, 11 (Nov. 1990), 71–85. https://doi.org/10.1109/2.60882

[44] Brad A. Myers. 1990. A new model for handling input. *ACM Transactions on Information Systems* 8, 3 (July 1990), 289–320. https://doi.org/10.1145/98188.98204

[45] Brad A. Myers, Dario Giuse, Andrew Mickish, Brad Vander Zanden, David Kosbie, Richard McDaniel, James Landay, Matthews Golderg, and Rajan Pathasarathy. 1994. The garnet user interface development environment. In *Conference companion on Human factors in computing systems - CHI '94*. ACM Press. https://doi.org/10.1145/259963.260472

[46] Siva Reddy, Mirella Lapata, and Mark Steedman. 2014. Large-scale Semantic Parsing without Question-Answer Pairs. *Transactions of the Association for Computational Linguistics* 2 (dec 2014), 377–392. https://doi.org/10.1162/tacl_a_00190

[47] 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*. ACM. https://doi.org/10.1145/3379337.3415848

[48] Gianluca Schiavo, Ornella Mich, Michela Ferron, and Nadia Mana. 2020. Trade-offs in the design of multimodal interaction for older adults. *Behaviour & Information Technology* 41, 5 (dec 2020), 1035–1051. https://doi.org/10.1080/0144929x.2020.1851768

[49] Marcos Serrano, Laurence Nigay, Jean-Yves L. Lawson, Andrew Ramsay, Roderick Murray-Smith, and Sebastian Denef. 2008. The openinterface framework: a tool for multimodal interaction.. In *CHI '08 Extended Abstracts on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/1358628.1358881

[50] Wai Wa Tang, Kenneth W.K. Lo, Alvin T.S. Chan, Stephen Chan, Hong Va Leong, and Grace Ngai. 2011. i*Chameleon: a scalable and extensible framework for multimodal interaction. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/1979742.1979703

[51] Christiana Tsiourti, João Quintas, Maher Ben-Moussa, Sten Hanke, Niels Alexander Nijdam, and Dimitri Konstantas. 2017. The CaMeLi Framework—A Multimodal Virtual Companion for Older Adults. In *Studies in Computational Intelligence*. Springer International Publishing, 196–217. https://doi.org/10.1007/978-3-319-69266-1_10

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

[53] Bryan Wang, Gang Li, and Yang Li. 2022. Enabling Conversational Interaction with Mobile UI using Large Language Models. https://doi.org/10.48550/ARXIV.2209.08655

[54] Silei Xu, Giovanni Campagna, Jian Li, and Monica S. Lam. 2020. Schema2QA: High-Quality and Low-Cost Q&A Agents for the Structured Web. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. ACM. https://doi.org/10.1145/3340531.3411974

[55] Jackie (Junrui) Yang, Gaurab Banerjee, Vishesh Gupta, Monica S. Lam, and James A. Landay. 2020. Soundr: Head Position and Orientation Prediction Using a Microphone Array. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3313831.3376427

[56] Jackie (Junrui) Yang, Tuochao Chen, Fang Qin, Monica S. Lam, and James A. Landay. 2022. HybridTrak: Adding Full-Body Tracking to VR Using an Off-the-Shelf Webcam. In *CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3491102.3502045

[57] Jackie (Junrui) Yang, Monica S. Lam, and James A. Landay. 2020. DoThisHere: Multimodal Interaction to Improve Cross-Application Tasks on Mobile Devices. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM. https://doi.org/10.1145/3379337.3415841

[58] Jackie (Junrui) Yang and James A. Landay. 2019. InfoLED: Augmenting LED Indicator Lights for Device Positioning and Communication. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. ACM. https://doi.org/10.1145/3332165.3347954

[59] Chris Zimmerer, Erik Wolf, Sara Wolf, Martin Fischbach, Jean-Luc Lugrin, and Marc Erich Latoschik. 2020. Finally on Par?! Multimodal and Unimodal Interaction for Open Creative Design Tasks in Virtual Reality. In *Proceedings of the 2020 International Conference on Multimodal Interaction*. ACM. https://doi.org/10.1145/3382507.3418850

A GRAMMAR OF NLPL

$$
\begin{aligned}
top &::= value \mid all\_symbol \\
all\_symbol &::= index\_symbol(.all\_symbol)? \\
index\_symbol &::= function\_call \mid symbol([int\_literal])? \\
function\_call &::= symbol((parameter\_list?)) \\
parameter\_list &::= parameter\_pair(,parameter\_pair)* \\
parameter\_pair &::= symbol{:}value \\
value &::= \text{true} \mid \text{false} \mid int\_literal \mid float\_literal \mid all\_symbol \mid \\
&\quad accessor \mid \text{"}string\text{"} \mid [array\_value] \\
accessor &::= .value \\
array\_value &::= value(,value)* \\
symbol &::= [a-zA-Z\_][a-zA-Z0-9\_]* \\
int\_literal &::= (+ \mid -)?[0-9]+ \\
float\_literal &::= (+ \mid -)?[0-9]*.[0-9]+
\end{aligned}
$$

B EXAMPLE PARSES

Here is a code excerpt from the developer-provided example parses for ReactGenieFoodOrdering:

```
Order.Examples = [
    {
        user_utterance: "What is the total price of my
            order?",
        example_parsed:
            "Order.GetActiveCart().getTotalPrice()",
    },
    {
        user_utterance: "Order a burger and two fries.",
        example_parsed:
            "Order.GetActiveCart().addItems(items:
            [OrderItem.CreateOrderItem(foodItem:
            FoodItem.GetFoodItem(name: \"burger\")),
            OrderItem.CreateOrderItem(foodItem:
            FoodItem.GetFoodItem(name: \"fries\"),
            quantity: 2)])",
    },
    {
        user_utterance: "I would like to place an order for
            pick up",
        example_parsed:
            "Order.GetActiveCart().setPickUp(pickup: true)",
    },
    {
        user_utterance: "What's the cheapest item in my
            order?",
        example_parsed:
            "Order.GetActiveCart().items.sort(field:
            .foodItem.price(), ascending: true)[0]",
    },
    {
        user_utterance: "What have I ordered last time from
            mcDonalds?",
```

```
        example_parsed:
            "Order.OrderHistory().matching(field:
            .restaurant, value:
            Restaurant.GetRestaurant(name:
            \"mcDonalds\"))[0].items"
    }
]
```

C PROMPT FOR NLPL PARSER

Here is an example prompt for OpenAI Codex to convert the user command to NLPL and its expected response. Text snippets starting with // are sent to the LLM as part of the prompt. It resembles code comments to help the LLM understand the structure of the prompt. To help the reader understand where different data are generated, we added comments surrounded by <>, which are not part of the prompt sent to the LLM.

```
// Here are all the functions that we have
<developer's class skeletons>
class Restaurant {
    string name;
    string address;
    string cuisine;
    float rating;

    // All active restaurants
    static Restaurant[] All();

    // The current restaurants
    static Restaurant Current();

    // Get a list of foods representing the menu from a
        restaurant
    Food[] menu;

    // Book reservations on date
    Reservation get_reservation(date: DateTime)
}
...
// Examples:
<example parses>
user: get me the best restaurant in Palo Alto
parsed: Restaurant.all().matching(field: .address, value:
    "Palo Alto").sort(field: .rating, ascending: false)
...

// Current User Interaction
<current user command>
user: order the same burger that I ordered at McDonald's
    last time
parsed:
<expected LLM response>
Order.Current().addFoods(foods: Order.All().matching(field:
    .restaurant, value: Restaurant.All().matching(field:
    .name, value: "McDonald's")).sort(field: .orderTime,
    ascending: false)[0].foods)
```