ReactGenie: An Object-Oriented State Abstraction for Complex Multimodal Interactions Using Large Language Models

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

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

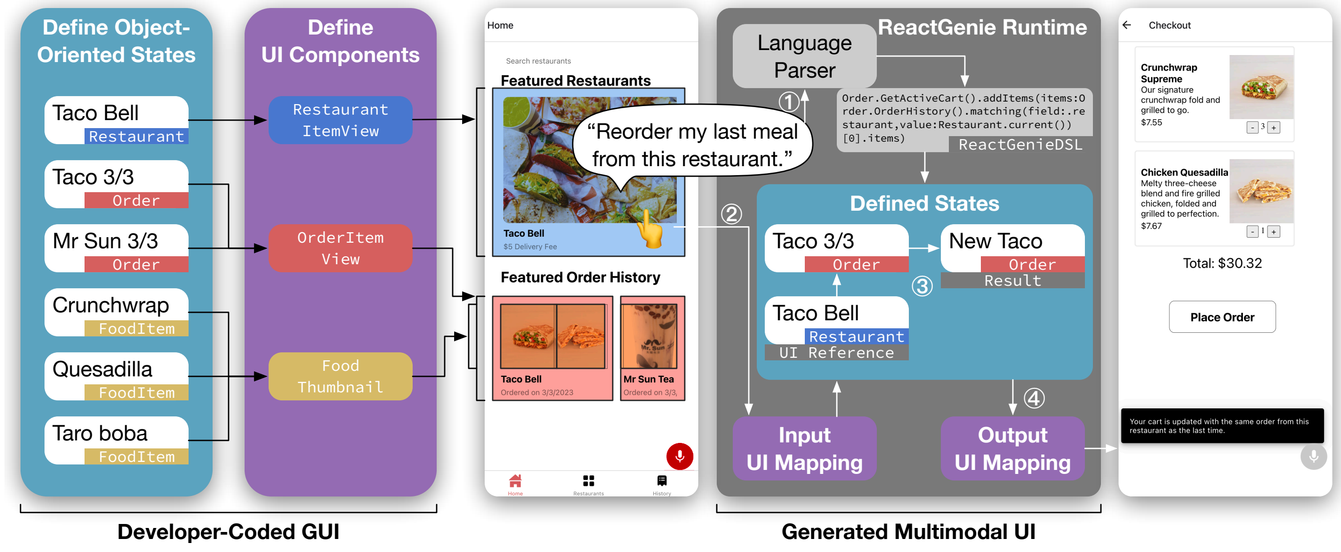
Daniel Wan Rosli
danwr@stanford.edu
Stanford University
Stanford, CA, USA

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

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

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

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



```
Order.GetActiveCart().addItems(items:Order.OrderHistory().matching(field:.restaurant,value:Restaurant.current())[0].items)
```

Figure 1: ReactGenie allows developers to easily build multimodal mobile applications by defining object-oriented states and UI components. (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 generates a natural language parser from developer-defined states and generates input and output UI mappings from developer-defined UI components. It can then execute complex multimodal commands by composing the methods and properties of states and presenting the results using existing UI components.

ABSTRACT

Multimodal interactions have been shown to be more flexible, efficient, and adaptable for diverse users and tasks than traditional graphical interfaces. However, existing multimodal development frameworks either do not handle the complexity and compositionality of multimodal commands well or require developers to write a substantial amount of code to support these multimodal interactions. In this paper, we present ReactGenie, a programming framework that uses a shared object-oriented state abstraction to support building complex multimodal mobile applications. Having different modalities share the same state abstraction allows developers using ReactGenie to seamlessly integrate and compose these modalities to deliver multimodal interaction.

ReactGenie is a natural extension to the existing workflow of building a graphical app, like the workflow with React-Redux. Developers only have to add a few annotations and examples to indicate how natural language is mapped to the user-accessible functions in

arXiv, ,
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

the program. ReactGenie automatically handles the complex problem of understanding natural language by generating a parser that leverages large language models.

We evaluated the ReactGenie framework by using it to build three demo apps. We evaluated the accuracy of the language parser using elicited commands from crowd workers and evaluated the usability of the generated multimodal app with 16 participants. Our results show that ReactGenie can be used to build versatile multimodal applications with highly accurate language parsers, and the multimodal app can lower users' cognitive load and task completion time.

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, Karina Li, Daniel Wan Rosli, Shuning Zhang, Yuhan Zhang, Monica S. Lam, and James A. Landay. 2023. ReactGenie: An Object-Oriented State Abstraction for Complex Multimodal Interactions Using Large Language Models. In *Proceedings of Uploaded to arXiv (arXiv)*. ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Multimodal interactions allow users to use a combination of input and output modalities, such as touch, voice, and graphical user interfaces (GUIs). They have been shown to offer more flexibility, efficiency, and adaptability for diverse users and tasks [47]. However, developing multimodal applications remains a challenging task for developers. Existing frameworks [10, 12, 28, 37, 38, 44, 45] often require developers to manually handle the low-level control logic for voice interactions and manage the complexity of multimodal commands. This manual process significantly increases development costs and time and often limits the expressiveness of voice and multimodal commands. There are systems [24, 36, 48] that 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 behavior of the app.

The goal of this research is to give developers a simple abstraction, while automatically handling the complexity of natural language understanding and composition of different modalities. We focus on the smartphone, the most popular device with millions of existing apps. We focus on letting users access off-screen content and actions, and complete tasks involving multiple taps in a single multimodal command, as illustrated in Figure 2. We aim to reuse the existing declarative GUI development workflow with little added workload to encourage the adoption of multimodal interactions and make multimodal interactions more accessible to end-users.

This paper presents ReactGenie[1], a declarative programming framework for developing multimodal applications. ReactGenie uses an object-oriented state abstraction to represent the data and logic of the app, and uses declarative UI components to represent the UI. Complex multimodal commands usually require multiple state changes. Existing declarative UI state management frameworks [5] use a monolithic object to represent the state of the UI. In other words, how the state changes for each step of a complex command is hidden within the monolithic object state, making it difficult for the framework to compose steps using different modalities. The object-oriented state abstraction in ReactGenie, on the other hand, encapsulates state changes in each step as methods with strictly typed parameters and return values. This allows easy and accurate composition of methods and properties of existing states for executing complex multimodal commands.

One example of a complex multimodal interaction is shown in the center of Figure 1: the user says, "Reorder my last meal from this restaurant" while touching the display of a restaurant on the screen. These commands are *intuitive* for human communication, but they actually involve multiple steps (retrieving the history of orders from the restaurant, creating an order, and adding food to the order) for the app and require combining inputs from both modalities, hence being "complex".

With ReactGenie, developers build graphical interfaces in a similar development workflow to a typical React + Redux application. To add multimodality, the developer supplies a few annotations and examples that indicate what methods/properties can be used in voice and how they can be used. By supplying the extracted class and function definitions from the developer's state code as well as few-shot examples, ReactGenie creates a parser that leverages large language models (LLM) [15] to translate the natural language to our new domain-specific language (DSL), *ReactGenieDSL*. Combined with our custom-designed interpreter, ReactGenie can seamlessly handle multimodal commands and present the result in the graphical UI that the developer built.

As shown in Figure 1 left, developers can build object-oriented state abstraction classes dealing with data changes and UI components containing an explicit mapping between the state and the UI. Similar to React, when the user interacts with the app, the state of the app will be updated and the UI will be re-rendered. What is unique about ReactGenie is its ability to support complex multimodal input (shown in Figure 1 right).

To process the example mentioned above:

(1) ReactGenie first translates the user's voice command to the ReactGenieDSL code. Here, the user referred to an element on the UI by voice ("this restaurant") and the language 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 both the parsed DSL and UI context, ReactGenie's interpreter can then execute the generated ReactGenieDSL 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.

[1]source code: https://jya.ng/reactgenie

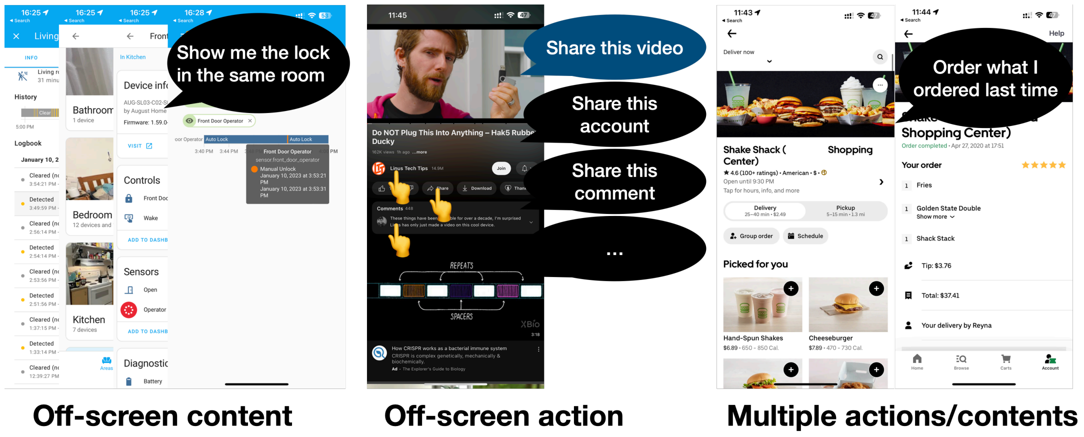


Figure 2: ReactGenie's targeted interaction scenarios.

(4) ReactGenie passes the return value of the ReactGenieDSL 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 ReactGenieDSL, and the return value: "Your cart is updated with the same order from this restaurant as the last time."

Our main contributions to this paper are as follows:

- ReactGenie, a new multimodal app development framework based on object-oriented state abstraction that is easy on developers and generates apps that support complex multimodal interactions.
- Automatic handling of natural language understanding in a multimodal app framework. This involves the design of the high-level annotation of user-accessible functions, the automatic generation of a natural language parser using LLMs that targets ReactGenieDSL, a new DSL for complex multimodal commands, and an interpreter that executes ReactGenieDSL.
- Evaluations of ReactGenie:
  - We demonstrated the expressiveness of the ReactGenie framework by building three demo apps.
  - We elicited commands from crowd workers and found the ReactGenie language parser can achieve 90% accuracy on supported tasks for one of the example apps.
  - We found in a user study (N=16) that the app generated with ReactGenie significantly reduced cognitive load and task completion time compared to a GUI-only app.

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 were developed in the 1980s [13]. They demonstrated that users can interact with a computer using a combination of voice and gestures. QuickSet [18] 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 [43, 46], photo editing [33], and digital virtual assistants [29]. Researchers have also explored different devices and environments for multimodal interaction, including augmented reality [54], virtual reality [35, 51], wearables [14], and the Internet of Things [21, 30, 50, 53].

These papers have demonstrated the great potential of multimodal interaction systems. However, multimodal systems still have limited adoption in the real world due to their added development complexity.

2.2 Graphical UI frameworks

ReactGenie is built on top of existing graphical UI frameworks to provide a familiar development experience.

Model–view–controller (MVC) [31] is a traditional approach to UI development, including frameworks such as Microsoft's Windows Forms [25], and Apple's UIKit [1]. The model stores data while the controller manages GUI input and updates the GUI 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 [40], 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 to abstract the GUI state from the presentation using designs like interactors [39]. While interactors allow the UI state to be rewired and thus to be updated from another modality like voice or gesture [32], they still lack the capability to enable manipulation of more abstract states (like an entire query) that are not directly mapped to a single UI control.

Declarative UI frameworks, such as React [2], Flutter [3], and SwiftUI [6], 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 [5], Flux [8], and Pinia [4], are often used. 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.

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 model can be compared to children (class instances) playing in a playground, with views (UI components) acting as cameras capturing different angles of the centralized state. In this way, ReactGenie enables complex action composition through type-checked functional 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, Dialogflow, and Rasa, are designed to handle natural language understanding and generation. They allow developers to define intents and entities, and then train the model to recognize the intents and entities from the user's input. These frameworks require complete redevelopment of the application to support voice input only. Frameworks such as Alexa Skills Kit and Google Actions allow developers to extend existing applications to support voice input. However, they still require manual work to build functions only for voice, and their UI updates are limited to simple text and a few pre-defined UI elements. Additionally, the intent-based nature limits the compositionality of the voice commands.

Research-focused voice/natural language frameworks, such as Genie [16, 49] and other semantic parsers [11, 41], are designed to support better compositionality of voice commands. However, given that 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 process 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. [17]. 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 [19, 20] and frameworks [10, 12, 28, 37, 38, 44, 45] 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 complex multimodal applications.

Recently, there have been research papers about generating voice commands by learning from demonstration [23, 42], extracting from graphical user interfaces with large language models [24, 36, 48], or building multimodal applications using existing voice skills [52]. 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 limited by what is provided by the voice skills and, traditionally, these have been very 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.

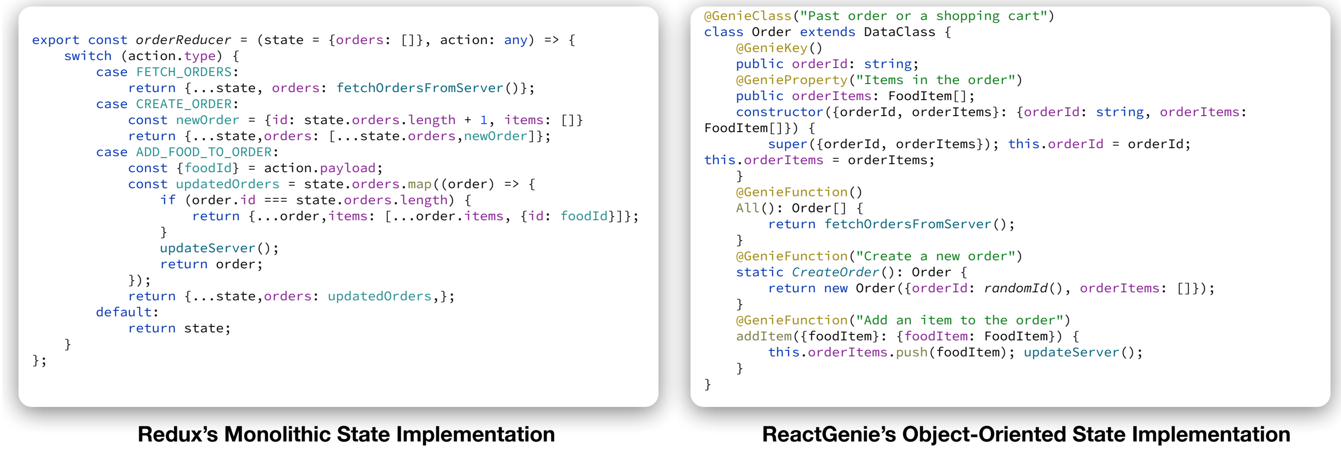
3 SYSTEM DESIGN

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

3.1.1 Interaction Design. ReactGenie is designed to enhance the interaction of mobile applications. 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. These interactions can be categorized into three interaction design goals (see Figure 2):

- **I1 Access off-screen content**: For example, the user noticed abnormal motion on their living room security camera footage. So they can say, "Show me the lock status history


Redux's Monolithic State Implementation
ReactGenie's Object-Oriented State Implementation

```
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: []}
            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;
    }
};
```

```
@GenieClass("Past order or a shopping cart")
class Order extends DataClass {
    @GenieKey()
    public orderId: string;
    @GenieProperty("Items in the order")
    public orderItems: FoodItem[];
    constructor({orderId, orderItems}: {orderId: string, orderItems:
FoodItem[]}) {
        super({orderId, orderItems}); this.orderId = orderId;
this.orderItems = orderItems;
    }
    @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();
    }
}
```

Figure 3: A comparison between state code in 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 do not have explicit types and they directly manipulate the state so no return values are defined. 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 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.

in the same room." This interaction would usually require multiple GUI navigation steps to access the content.

I2 **Access off-screen actions**: 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 on mobile devices.

I3 **Combine multiple actions/content**: 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 of these interactions is that they require the multimodal interaction framework to understand the content and actions available in the app. For the *content*, the framework needs to know what the content on the screen is, how to access it, and how to represent it to show the retrieved content. For the *actions*, the framework needs to know the list of available actions and how to reflect changes back to the user after the action is triggered. Finally, the framework needs to understand the *user's complex intent* and potentially represent it as a series of actions and content.

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 on it. The app will then parse the voice command and touch input and execute the corresponding actions to help the user with the types of scenarios described above.

3.1.2 Framework Design. To understand the content, the actions, and the user's complex intent, ReactGenie needs to seek information from the app's code. However, 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 needs to design mechanisms to handle voice, deal with the complexity of multimodal commands, and maintain control of the app's behavior. Instead, ReactGenie's framework design goals include handling these issues:

F1 **Ease the learning curve** by providing a similar programming experience to existing GUI frameworks.

F2 **Maximize the reuse of existing GUI code** and alleviate the developer from handling the complexity of multimodal commands.

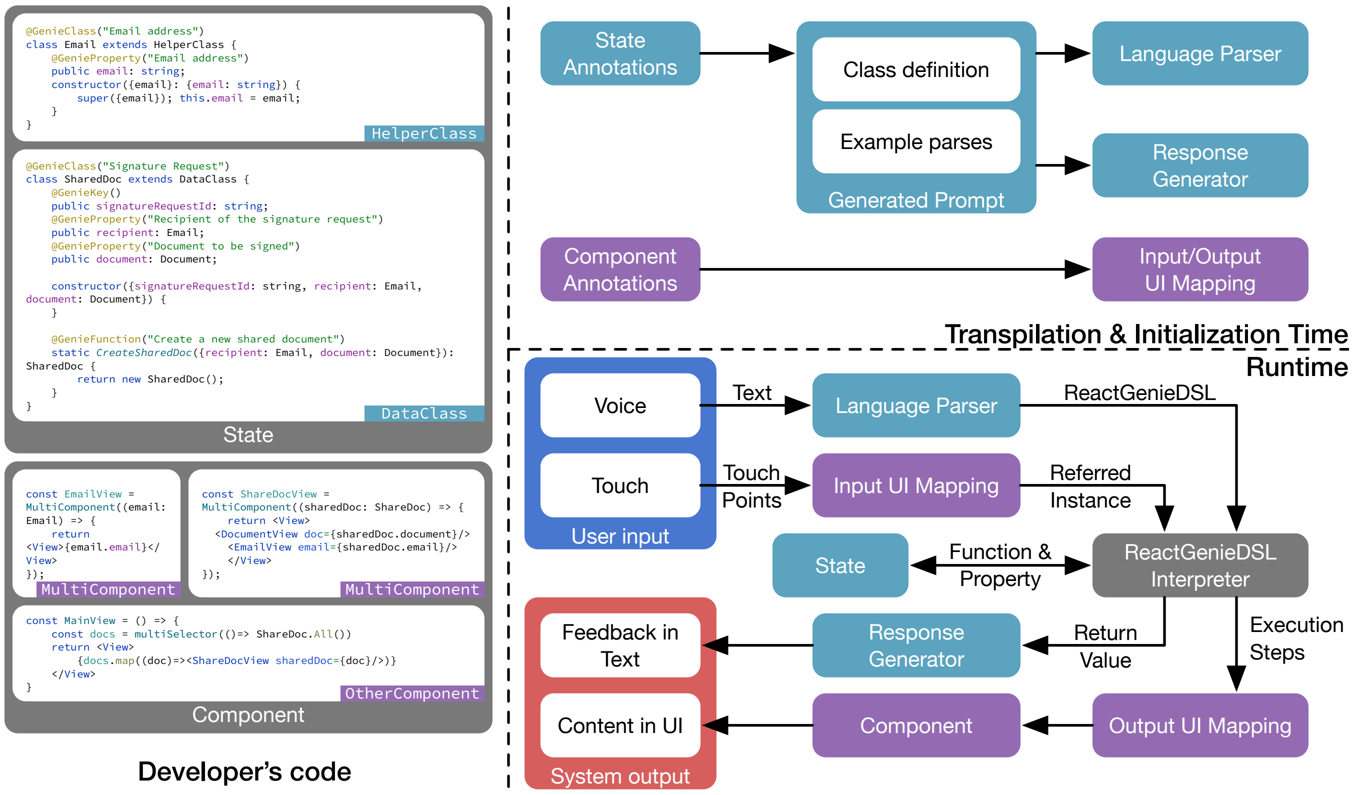
F3 **Allow developers to have full control** over the graphical UI appearance and app behavior.

3.2 Theory of Operation

ReactGenie presents an object-oriented state programming model to the developer. As mentioned in Section 2.2, in frameworks like React, UI development is moving towards separating the UI from the state, so that there is a unidirectional data flow from the state to the UI. This trend 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 frameworks, the state is simplified to a single data store with functional transforms (actions) on the data store. The monolithic state makes it difficult to compose multiple actions involving different modalities in one command (**I3**).

ReactGenie introduces the object-oriented programming model to the state of a declarative UI app. The object-oriented state model naturally separates the content and actions in the app. The content is usually defined as properties of the state objects, and the actions are defined as methods of the state objects.



```
@GenieClass("Email address")
class Email extends HelperClass {
    @GenieProperty("Email address")
    public email: string;
    constructor({email}: {email: string}) {
        super({email}); this.email = email;
    }
}
```

```
@GenieClass("Signature Request")
class SharedDoc extends DataClass {
    @GenieKey()
    public signatureRequestId: string;
    @GenieProperty("Recipient of the signature request")
    public recipient: Email;
    @GenieProperty("Document to be signed")
    public document: Document;

    constructor({signatureRequestId: string, recipient: Email,
document: Document}) {
    }

    @GenieFunction("Create a new shared document")
    static CreateSharedDoc({recipient: Email, document: Document}):
SharedDoc {
        return new SharedDoc();
    }
}
```

```
const EmailView =
MultiComponent((email:
Email) => {
    return
<View>{email.email}</
View>
});
```

```
const ShareDocView =
MultiComponent((sharedDoc: ShareDoc) => {
    return <View>
  <DocumentView doc={sharedDoc.document}/>
    <EmailView email={sharedDoc.email}/>
    </View>
});
```

```
const MainView = () => {
    const docs = multiSelector(()=> ShareDoc.All())
    return <View>
        {docs.map((doc)=><ShareDocView sharedDoc={doc}/>)}
    </View>
}
```

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

The developer identifies the user-accessible content and actions with the `GenieProperty` and `GenieFunction` annotations, along with an example of how it may be referred to in English as shown in Figure 3 right. The GUI is defined as cascading components that are rendered from the user-accessible state. *Components* are functions that developers define, which convert data in the state to HTML, UI elements, or a combination of other components. 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 (**F1**). ReactGenie automatically handles the retrieval of content off screen (**I1**), the execution of actions off screen (**I2**), and combinations of the two (**I3**).

We contrast ReactGenie's object-oriented approach with React in Figure 3. In React, if the user says "Add a food to a new order", this is implemented as a code sequence that calls the `CREATE_ORDER` action, indexes into the last of the orders in the state, and calls the `ADD_FOOD_TO_ORDER` action. Due to the nature of these actions, there is no explicit connection linking them together. Whereas in ReactGenie, each step corresponds to a method invocable by multimodal input. These methods are all strictly typed, which helps the natural language parser come up with the correct combination of methods. The entire natural language command can be represented succinctly with a single-line command: `Order.CreateOrder().addItem(foodItem:food)`. This succinctness improves the accuracy of the natural language parser that translates the natural language command to this code. Furthermore, as the same content and state are shared by the GUI and voice modality, this representation supports interchangeability in the modality of user input for each step.

Due to the complexity of the user's multimodal commands, we cannot use a simple intent classifier like that used by traditional voice assistants. Instead, we use a sequence-to-sequence neural semantic parser to translate the user's natural language command into ReactGenieDSL, which is executed using the developer's code. We do not generate JavaScript directly, because the expressiveness of JavaScript may cause unintended changes to the app's behavior (contradicting **F3**).

3.3 System Components

As shown in Figure 4, a ReactGenie app consists of developer-supplied code, MultiReact-supplied runtime modules, and modules generated from the developer's code.

3.3.1 Developer Modules. 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 includes the definition of classes and instances. Classes have methods and properties, which can be labeled as `GenieFunction` and `GenieProperty` using annotations to indicate that they are user-accessible via multimodal commands. 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.

Class Definitions. ReactGenie provides developer-accessible classes for developers to implement the state of their app: `DataClass` and `HelperClass`. The former stores data of the app, and the latter provides definitions to ease the user's interaction with the data. Examples of each type of class are shown in the top left of Figure 4.

All `DataClass` instances need to implement two methods and one property:

(1) `constructor` method: The constructor of the class initializes the instance with all the required data.
(2) `All` method: A static method that either returns a list of instances of the class or returns a `Query` object that can be used to retrieve the instances. Should be annotated with `GenieFunction`.
(3) `id` property: A unique identifier of the instance.

With this information, ReactGenie can automatically generate two methods for each `DataClass`:

(1) `Get` method: A static method that takes an `id` as input and returns the instance in memory with the corresponding `id`.
(2) `Current` property: A static method that returns the instance that is being referred to by the user. This is automatically annotated with `GenieProperty`.

A `DataClass` is like a table in a database. The `constructor`, `All`, `id`, and `Get` are basically insertion, selection, primary key, and retrieval of a row using its primary key. ReactGenie automatically maintains the instances of `DataClass` in memory to form the state of the app. Similar to many of the common state management frameworks, when any of the properties of a `DataClass` instance is changed, UI components that refer to that property will be automatically updated. This ensures that the UI is always in sync with what is being represented in the state.

`DataClass` can be backed by a remote database, which is common in modern app development. This database is automatically managed by ReactGenie. When the data is backed by a remote database, `All` can be implemented as a query to the original database, and it can instantiate the `DataClass` with the data retrieved from the database when needed.

The `HelperClass` allows developers to define new types that can be used in the `DataClass`. For example, ReactGenie already provides a basic `DateTime HelperClass` that can be used to represent a date. It can support operations like offsetting the time by a certain amount of time or setting the year/month/day/hour/minute/second/day of the week to a certain value. It allows commands such as "last Thursday", which can be translated to "`DateTime.Current.offset(week:-1).set (weekOfTheDay:4)`". Other `HelperClass` instances can be defined by the developer to support more complex operations such as length unit conversion and so on. The `HelperClass` is only required to have 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 properties of the instance.

Natural language annotations. ReactGenie calls an LLM with the functions and properties labeled `GenieFunction` and `GenieProperty` in a format resembling a class definition to teach the LLM the app's functionality. If the design of the classes is unusual, the developer needs to define a few example commands and the corresponding ReactGenieDSL to teach the LLM the syntax (see Section 3.3.2).

For example, when building an example food ordering app, we represent both the current shopping cart and previously placed orders using the same `Order` object and distinguishing them with the `orderPlaced` property. We found that the LLM cannot parse some of the commands correctly, and we need to provide two examples to fix the majority of parsing errors.

Graphical user interface declarations. The ReactGenie developer needs to define the GUI (as shown in the bottom left of Figure 4) as a set of functional components[3] similar to 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. Therefore, ReactGenie introduces a special component called `GenieComponent` for that purpose. Instead of the arbitrary parameters of a normal functional component, `GenieComponent` takes a `DataClass` or `HelperClass` instance as input. `GenieComponent` allows the ReactGenie runtime to understand which component is mapped to which instance in memory. 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.

3.3.2 ReactGenie Modules. The most important component of ReactGenie is the *DSL interpreter* module. It is responsible for running the generated ReactGenieDSL code and calling the corresponding methods in the developer's state code. The ReactGenieDSL is designed to meet these language design goals:

L1 *Easy to generate*: LLMs can generate syntactically correct ReactGenieDSL code.

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

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

L2 *Robust to generation errors*: LLMs can generate semantically correct ReactGenieDSL code.

L3 *Able to express multimodal commands*: ReactGenieDSL can express diverse multimodal commands.

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

To reduce complexity (also helps with **L2**), we decided not to use any lambda functions in ReactGenieDSL. To maintain expressiveness **L3**, we added a few array functions to aid basic filtering (`matching`, `between`), sorting (`sort`), and summarization (`sum`, `average`, `count`). The full grammar of ReactGenieDSL is listed in Appendix A. We implemented the *DSL interpreter* module using peggy[4].

The ReactGenie framework ships with one `Helperclass` called `DateTime` and a set of `GenieComponent` that can be used to represent the `DateTime` instance, like one that shows the data in a convenient "*n* seconds/minutes/hours/days ago" format. In the future, ReactGenie can provide more `HelperClass` and `GenieComponent` to support common operations. Libraries can also be built on top of ReactGenie to provide more `HelperClass` and `GenieComponent` for specific domains.

3.4 System Workflow

Typically, a TypeScript app is transpiled to JavaScript so that it can be run in a mobile app or in a browser. *Transpilation* is a source-to-source translation process from the TypeScript that the developer writes to Javascript that the machine executes. 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 understand the developer's code. Therefore, ReactGenie works by generating modules during transpilation and at initialization time when information about the developer's code is still available. ReactGenie calls the generated modules during runtime (as shown in the top right of Figure 4).

3.4.1 Transpilation and Initialization for States. We built a custom transpiler plugin that generates extra metadata for `@GenieProperty`, and `@GenieFunction` of `DataClass` and `HelperClass`. During initialization of the app, ReactGenie will load injected metadata from *state classes* to generate a prompt for the LLM. LLMs work by generating text continuations given a paragraph of previous text. The provided previous text is often referred to as the *prompt*. By controlling the prompt, we change the information that the LLM has access to and guide the LLM to do what we want (generate the corresponding ReactGenieDSL of the user's command). ReactGenie's *generated prompt* contains two parts: 1) The *class definitions* contain all the `DataClass` and `HelperClass` methods and properties 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 *language parser* adds the user input to the generated prompt and presents that to the LLM to translate into ReactGenieDSL. The *response generator* prompts the LLM with the generated prompt, the user input, the parsed ReactGenieDSL, and the `description` of the return value from the execution of ReactGenieDSL to produce a short text response.

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

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

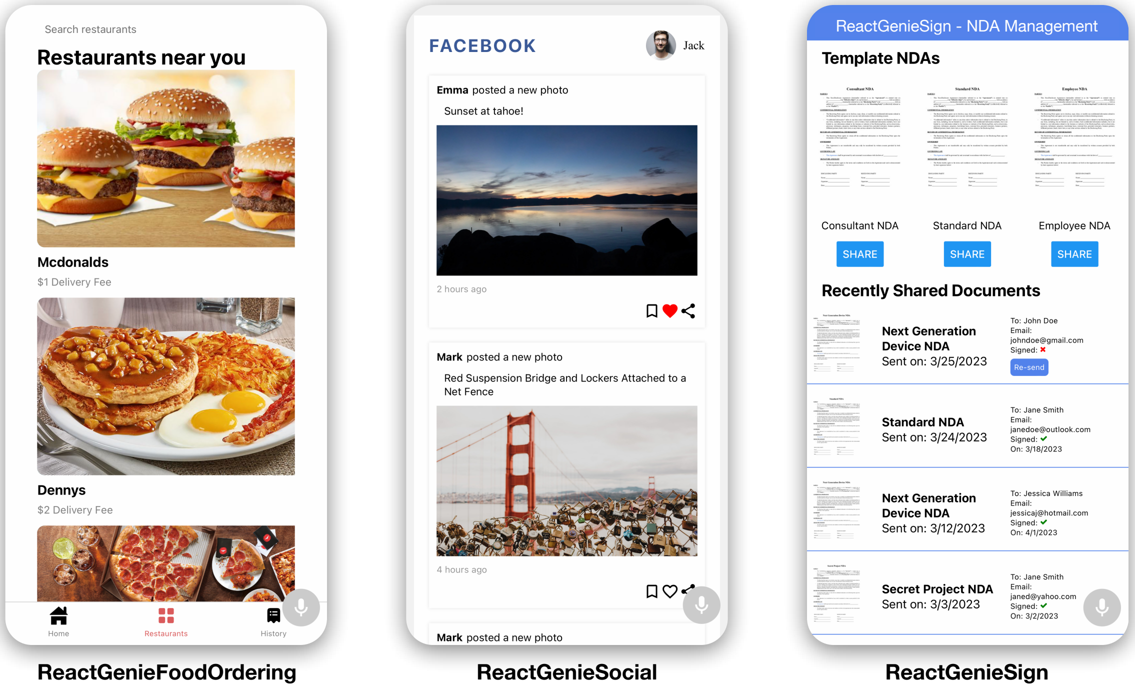
It is common for multiple UI components to cover the area where the user taps on the screen. For example, in Figure 1, all the `FoodThumbnail` components overlap with the `OrderItemView` components. ReactGenie allows the user to use their voice to disambiguate the reference: If the user mentions food, 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 the tapped area is covered by multiple components of the same type, 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 are representing. When the result of the executed ReactGenieDSL 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 that the translated ReactGenieDSL 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 "create an empty cart". In that case, it would be intuitive to render the result on the screen directly. So, when the return value can be represented by a `GenieComponent`, ReactGenie will always find the highest priority `GenieComponent` and render it.

[4]https://peggyjs.org/



Emma posted a new photo
Sunset at tahoe!
2 hours ago

Mark posted a new photo
Red Suspension Bridge and Lockers Attached to a Net Fence
4 hours ago

Mark posted a new photo

Next Generation Device NDA
Sent on: 3/25/2023
To: John Doe
Email: johndoe@gmail.com
Signed: ✗

Standard NDA
Sent on: 3/24/2023
To: Jane Smith
Email: janedoe@outlook.com
Signed: ✓
On: 3/18/2023

Next Generation Device NDA
Sent on: 3/12/2023
To: Jessica Williams
Email: jessicaj@hotmail.com
Signed: ✓
On: 4/1/2023

Secret Project NDA
Sent on: 3/3/2023
To: Jane Smith
Email: janed@yahoo.com
Signed: ✓
On: 3/2/2023

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

The second type is that the translated ReactGenieDSL returns a value that cannot be rendered by a `GenieComponent`. For example, if the user asks to "add a hamburger to the cart" (ReactGenieDSL: `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 effect of the action, so we can provide confirmation via text feedback. In the case when the user is already on a restaurant page where they can see an indicator of the number of items in the cart (the counter as a component would also represent 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 make sure the user understands the action being performed. In that case, ReactGenie will automatically check the execution history and find the last readable result is `Order.GetActiveCart()`. ReactGenie will also check all the currently shown components (in a similar way as how the *input UI mapping* works) to see if there is any component representing the same instance. ReactGenie would only render this result if the current page does not have any component representing the same instance.

3.4.3 Runtime. Similar to normal React or React-Native apps, when users interact with buttons and visual controls in the app, 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 3.3.1), the ReactGenie modules (Section 3.3.2), and the generated modules (Section 3.4.1) 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 then is passed to the *language parser* module to generate the ReactGenieDSL code. The touch point(s) are passed to the *input UI mapping* module to figure out which component and state instance the user can be referring to. Both pieces of information are then passed to the ReactGenieDSL interpreter to execute the ReactGenieDSL code with the corresponding relevant state instance. ReactGenie uses the methods and properties of the developer-provided *state classes* to execute the ReactGenieDSL 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 figure out 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.*

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 1: Implementation statistics of 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 EVALUATION

We first followed the guidelines of Ledo et al. [34] to evaluate ReactGenie as a toolkit through demonstration and technical performance. We demonstrate the expressiveness and usefulness of the ReactGenie framework by building three demo apps. We analyze the number of annotations and examples needed when building the demo apps to show that the cost of building multimodal support is low. We also analyze the technical performance of the ReactGenie system by measuring the accuracy of the language parser through an elicitation study. We then verified our demo apps are effective through a lab-based usability study with the demo food ordering app. Our results showed the multimodal version of the app significantly improved the user experience over the GUI-only version in various aspects.

4.1 Example Apps

We built three example apps across three major categories of apps: food & drink, social networking, and business, as shown in Figure 5. The implementation statistics are shown in Table 1.

4.1.1 ReactGenieFoodOrdering. ReactGenieFoodOrdering is a food ordering app that allows users to order food from a restaurant. It has the basic functions of browsing menus, shopping cart management, and checking order history. In total, the app is composed of around 2689 lines of code, only 88 (3%) of which are related to building multimodal UIs. 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 has the basic functionalities of browsing posts, interacting with posts, and sharing posts. In total, the app is composed of around 1034 lines of code, only 49 (5%) of which are related to building multimodal UIs.

4.1.3 ReactGenieSign. ReactGenieSign is a business app that manages NDAs and contracts. It has the basic functionalities of creating documents, sharing documents for signing, and user management. In total, the app is composed of around 867 lines of code, only 51 (6%) of which are related to building multimodal UIs.

4.1.4 Summary. These demo apps showed that when building a typical app with ReactGenie, only a small fraction (5% on average) of the code has to be written to handle multimodal interactions. This is particularly impressive since defining multimodal interaction can be intricate and typically requires a substantial amount of code to support. While building these demo apps, we also noticed that most UIs are naturally decomposed into components that represent different ReactGenie state instances, which made it easy to decompose the UI into `GenieComponents`.

4.2 Elicitation Study

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:

(1) RQ1: What percentage of the commands 1) are 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.
(2) RQ2: How accurate the parser is when parsing commands in the targeted interactions.

4.2.1 Elicitation. We would like to get multimodal commands that users may use in a real-world scenario. We adopted a similar method as described as Cloudlicit [9]. We provided the user with three screenshots (restaurant listing page, restaurant menu page, and past orders page) of the two most popular food ordering apps in the US: DoorDash and UberEats. In our pilot study, we found that many participants thoughts on what they can do were 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 in a random order, containing 4 videos for each of the three categories of interactions. Among these 12 videos, we also made sure half of them involve 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 to complete and we paid $2 for each participant.

From these 50 participants, we got 300 commands. We filtered out 12 responses that were unclear or not related to the survey. For example, one participant wrote "various good foods to order or view that can be good" as a command. After filtering, we have 288 commands in the dataset.

4.2.2 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 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 filled *"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 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. We 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 [26] to measure the inter-rater reliability. The AC1 score is 0.83, which means the labels are highly consistent. We 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 fall into the three targeted interactions, and 16 commands were out of the scope of ReactGenie.

This shows that users can come up with tasks that are 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.

4.2.3 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 ReactGenieDSL 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 is trying 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; 3) delivery time estimates for restaurants.

From the 101 commands that are supported by ReactGenieFoodOrdering, we found that 91 commands are parsed correctly by the parser, and 10 commands are not parsed correctly. Therefore, on this dataset, the parser has an accuracy of 90%.

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 ReactGenieDSL for the majority (38) of these commands. 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 ReactGenieDSL. 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 are 33 unsupported commands that are not parsed correctly by the parser. Some of them are due to the parser generating ungrammatical ReactGenieDSL 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 ReactGenieDSL 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 language parser by always adding the touch context even when it seems unnecessary.

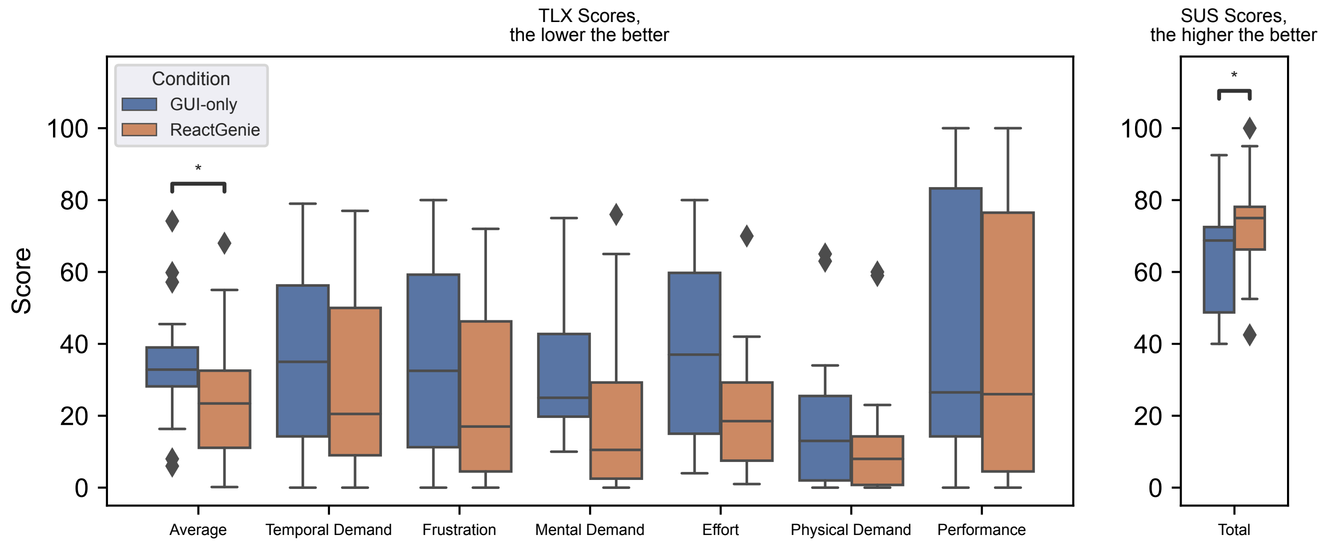


Figure 6: Cognitive load (left) and usability (right) of the GUI vs. the multimodal UI.

4.3 Usability Study with Prototype Applications

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.

4.3.1 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-subject design, where each participant completed the same tasks using both variants of the app. For each variant of the app, we first teach the participant how to use the app using one example task, then we ask 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 [27]) and the usability of the experience (using SUS [7]). 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 re-ordering 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 want them to do, and what's 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 pairs of tasks.

4.3.2 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 to 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 to 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.

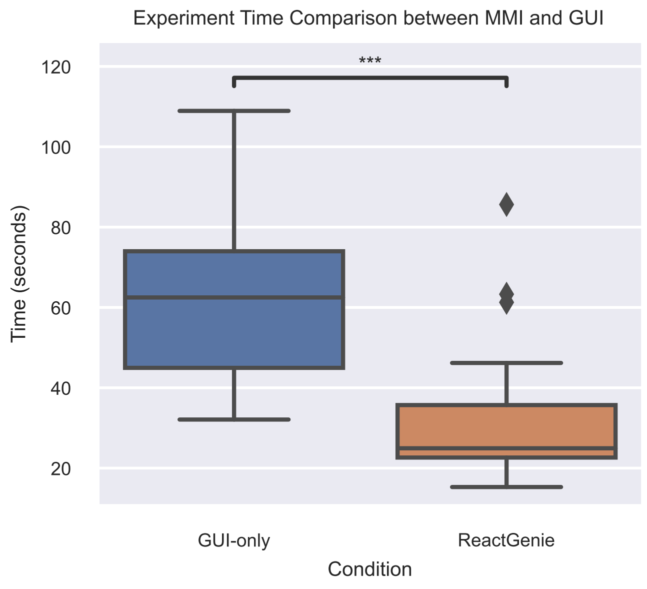


Figure 7: Time it takes to complete each task using the graphical UI and the multimodal UI (average across all participants).

4.3.3 *Result.*

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

We compared NASA-TLX average scores between the two conditions (see Figure 6 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 6 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 people 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 in the graphical UI. For people 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.

4.3.4 *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 graphical UIs. 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. 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.

5 DISCUSSION

In this section, we will discuss the limitations, future work, safety, and implications of ReactGenie.

5.1 Limitations and Future Work

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

5.1.1 *Better Voice Interfaces.*

ReactGenie accepts the user's voice 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, what can be an area of improvement is maintaining natural language context. For example, if the user says, "What 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 actually 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 asks, "Order one of that" (ReactGenieDSL`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 UI.

Another way to improve the reliability of the generated interfaces is to better leverage multimodal commands for disambiguation. As shown in Section 4.2.3, 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 language parser to further push the parser's accuracy closer to 100%.

5.1.2 *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 also increase the effectiveness of the examples. The majority of these examples are there for teaching the parser how to generate syntactically correct ReactGenieDSL 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 ReactGenieDSL code, similar to the method used in SEMPRE [11] or Genie [16]. Another route is to fine-tune the Codex model with the ReactGenieDSL code generated by the interpreter so that the interpreter can generate syntactically correct ReactGenieDSL code with fewer examples.

Future extensions to the ReactGenie framework can also help developers to 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.

5.2 Safety and Implications

ReactGenie uses a machine learning model to understand the users' commands. This may bring safety issues when the wrong command is interpreted and executed. This risk can be reduced by having a more accurate language parser, and ReactGenie already has relatively high accuracy.

Also, compared with an end-to-end natural language assistant like ChatGPT[5], ReactGenie allows more control over the *presented information* and *performed actions*. ReactGenie will only present information that exists in the app and will not hallucinate information. One particular case of error is when the user asks for delivery time but because the app does not support delivery time estimation, ReactGenie returns the delivery fee instead. In this case, the text feedback mechanism can be used to inform the user of the information that is actually returned. In the future, an error correction mechanism would be useful for the user to report the error and 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 that is 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 GUI where there is little room for error.

Another implication of ReactGenie is the possible social implications of multimodal interaction. ReactGenie encourages users to use voice and touch to quickly achieve their goals without the need to go 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. [22] that can be used in public spaces.

6 CONCLUSION

Commercial user interfaces have stagnated with the same mobile GUI for the past 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 complex 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 allowing developers to create multimodal UIs that allow for more expressiveness than traditional GUIs, with little additional programming effort. ReactGenie accomplishes this by introducing a new object-oriented state programming framework coupled with a powerful natural language understanding module that leverages the capabilities of LLMs. In this paper, we demonstrated the expressiveness, usefulness, and accuracy of the ReactGenie framework. In the future, through the introduction of developer tools based on frameworks like ReactGenie, and the research on multimodal interaction these tools enable, we hope to see humans communicating with computers more expressively and more easily.

ACKNOWLEDGMENTS

We thank the reviewers and the participants in our user studies for their feedback. We also thank Tianshi Li for her feedback on writing and building early prototypes of the system.

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

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.]. Redux - A predictable state container for JavaScript apps. | Redux. https://redux.js.org/. (Accessed on 04/04/2023).
[6] [n. d.]. SwiftUI. https://developer.apple.com/xcode/swiftui/. Accessed on 2023-04-05.
[7] 1996. SUS: A 'Quick and Dirty' Usability Scale. In *Usability Evaluation In Industry*. CRC Press, 207–212. https://doi.org/10.1201/9781498710411-35
[8] 2023. GitHub - facebookarchive/flux: Application Architecture for Building User Interfaces. https://github.com/facebookarchive/flux. (Accessed on 04/04/2023).
[9] 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
[10] 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
[11] 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/
[12] 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
[13] 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
[14] 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
[15] 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
[16] 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
[17] 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
[18] 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
[19] 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/
[20] 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
[21] 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
[22] 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

[23] 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

[24] Divyansh Garg. 2023. Multi on. https://multion.ai/

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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.

[32] 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

[33] 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

[34] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. Evaluation Strategies for HCI Toolkit Research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3173574.3173610

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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

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

[50] 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

[51] 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

[52] 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

[53] 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

[54] 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 REACTGENIEDSL

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