API Usage Dialogues with a Simulated Virtual Assistant

Zachary Eberhart, Aakash Bansal, and Collin McMillan

Abstract—Virtual Assistant technology is rapidly proliferating to improve productivity in a variety of tasks. While several virtual assistants for everyday tasks are well-known (e.g. Siri, Cortana, Alexa), assistants for specialty tasks such as software engineering are rarer. One key reason software engineering assistants are rare is that very few experimental datasets are available and suitable for training the AI that is the bedrock of current virtual assistants. In this paper, we present a set of Wizard of Oz experiments that we designed to build a dataset for creating a virtual assistant. Our target is a hypothetical virtual assistant for helping programmers use APIs. In our experiments, we recruited 30 professional programmers to complete programming tasks using two APIs. The programmers interacted with a simulated virtual assistant for help – the programmers were not aware that the assistant was actually operated by human experts. We then annotated the dialogue acts in the corpus along four dimensions: illocutionary intent, API information type(s), backward-facing function, and traceability to specific API components.

Index Terms—Software Agents, Virtual Assistants, Software Engineering, Wizard of Oz (WoZ).

1 Introduction

Virtual Assistants (VAs) are software systems that interact with human users via natural language and perform tasks at the request of those users. VAs for everyday tasks (e.g., Cortana, Alexa, Siri) are proliferating after a period of heavy investment – a confluence of sufficient training data, advancements in artificial intelligence, and consumer demand have fed rapid growth [1].

Many of the achievements of VAs for everyday tasks are beginning to be brought to specialty applications such as medicine [2] and education [3]. However, a key observation is that these applications are quite specific: e.g., not education in general, but a specific type of geography for a specific age group of students. The reason is that data collected for one application is difficult to generalize to other applications – a virtual assistant must master the both the language and the strategies humans use to move through conversations, and the complexity is simply "too much" for existing AI technologies to learn without detailed, specific training data [4]. A relevant dataset must be collected and annotated for every type of conversation in which a virtual assistant needs to converse. For example, a VA for everyday tasks would require different training data to recommend a restaurant and to reserve a table at that restaurant [5], [6].

VAs for software engineering tasks suffer from the same hunger for data. Despite long-recognized demand for virtual assistants to help programmers [7], [8], working relevant VA technology remains something of a "holy grail." Several research prototypes have made significant advances (see Section 2), but a major barrier to progress is a lack of well-understood, annotated datasets that are specific to software engineering tasks. A survey in 2015 by Serban *et al.* [9] found none related to SE tasks, and since that time only one has been published to our knowledge, targeting the task of bug repair [10].

One reason for the lack of suitable datasets is the investment cost necessary for experiments in numerous target tasks, and a perceived disincentive in terms of publication versus data and software artifacts [11]. A recent book by Reiser and Lemon [12] provides clear guidance for how to build dialogue systems, focusing especially on the design of experiments for data collection. A major theme of the book is that, despite a perception that data collection experiments yield few immediate research outcomes, in fact the experiments provide answers to research questions about how people seek knowledge to perform tasks. These answers are critical to later design of virtual assistants, in addition to the data produced. Towards this end, Reiser and Lemon establish two first steps towards building a VA: 1) conduct "Wizard of Oz" (WoZ) experiments to collect simulated conversation data, and 2) annotate every utterance in the conversations with Dialogue Act (DA) types.

A WoZ experiment is one in which a virtual agent is simulated. Participants interact with a virtual assistant to complete a task, but they are unaware that the VA is actually operated by a human "wizard". The deception is necessary because people communicate differently with machines than they do with other humans [13] and our objective is to create data for a machine to learn strategies to converse with humans. The key element of these strategies are "dialogue acts": a dialogue act is a spoken or written utterance that accomplishes a goal in a conversation [14]. A conversation is composed of a series of utterances taken by different

- *Manuscript received —– —– ——. This work is supported in part by the NSF CCF-1452959 and CCF-1717607 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.*
- *The authors are with the Department of Computer Science and Engineering, University of Notre Dame, IN 46556.*
  E-mail: zeberhar, abansal1, cmc@nd.edu
- *This paper has supplementary downloadable multimedia material available at https://github.com/ApizaCorpus/ApizaCorpus provided by the authors. This includes materials related to the experimental design, experimental results, and dialogue act annotations. This material is 13.8 MB in size.*

speakers, and each utterance functions as a dialogue act. For example, the utterance "tell me how to find my classroom" is a dialogue act explicitly requesting information, whereas "this is the wrong classroom" is a statement that, depending on context, may imply a request for information. A VA must learn to recognize when a human is e.g. requesting information, and to do that it relies on training data in which humans have annotated the dialogue acts in conversations.

At the same time, in software engineering, one task that cries out for help from virtual assistants is API usage: programmers trying to use an unfamiliar API to build a new software program. As Robillard *et al.* [8] point out, API usage is a high value target for VAs due to the high complexity of the task, and a tendency to need the same information about different APIs [15], [16]. In other words, programmers often have the similar kinds of questions about different APIs (e.g. "Is there an API type that provides a given functionality?", or "How do I determine the outcome of a method call?"), even if the tasks performed by the APIs are not similar – the similarity of questions makes API usage a good target for VAs, since the VAs are likely to be able to learn what programmers need to know.

There are a wide variety of scenarios in which programmers may benefit having a VA to assist with API usage. The authors of APIs are often not available to answer questions (e.g. for free APIs found online), and web support (e.g. StackOverflow) is neither a guarantee nor immediately available, which makes the rapid help a VA can provide more valuable. In some cases, programmers may have limited access to traditional documentation. For instance, blind programmers rely heavily on API documentation to understand the structure of code [17], but are limited to existing screen-reading tools. Another example would be programmers looking to incorporate smartphones or other devices with limited screen size into their workflows. These programmers may prefer to query a VA for specific information, rather than navigate large documents. Children or novice programmers may also find a VA more approachable than traditional documentation.

In this paper, we conduct WoZ experiments designed to lay a foundation for the creation of virtual assistants for API usage. We hired 30 professional programmers to complete programming tasks with the help of a "virtual assistant," which was operated by a human wizard. The programming tasks involved building a program that met specified objectives using an API (see Section 3.2 for details). The programmers conversed with the virtual assistant, though they were not aware that it was operated by a human. Each programming session lasted approximately 90 minutes.

We then annotated the dialogue acts in all 30 conversations. First, we labeled the illocutionary dialogue act types by adapting the DA annotation scheme from the AMI conversation corpus [18]. The AMI dialogue act labels are 14 coarse-grained illocutionary types (e.g. INFORM, ELICIT-SUGGESTION) extracted from simulated business meetings. The advantage to annotation with these dialogue act types is that comparison is possible with several other datasets that use the same types, but the disadvantage is that they are so coarse-grained that they alone do not capture the nuances of SE conversations. Therefore, we also annotated the API dialogue act types content of each utterance. These types, adapted from Mallej and Robillard [19], are a set of 12 labels (e.g. CONTROL-FLOW, FUNCTIONALITY) corresponding to domains of API knowledge. Next, we annotated the backward-facing function of each dialogue act. The AMI scheme provides a small set of labels to describe the relationship of an utterance (e.g. POSITIVE, NEGATIVE, PARTIAL) to a previous utterance. These relationships allow us to identify and track conversational threads. Finally, we annotated specific API components (e.g. specific method or variable names) that were referenced in each utterance in order to observe the traceability between specific concepts and the language used to discuss them.

2 Background and Related Work

This section discusses background on the problem we target, supporting technologies, and related work.

2.1 Problem, Significance, Scope

The problem we target in this paper is that the composition and patterns of dialogue acts are not known for conversations between programmers and virtual assistants during API usage tasks. This problem is significant because information about these dialogue acts must be known in order to create lifelike virtual assistant systems. The situation is a "chicken or egg" question because in order to obtain the dialogue act structure, one must have conversations between programmers and VAs, but to have a VA one must know the dialogue act structure. Software engineering literature does not describe dialogue acts for API usage conversations, which impedes development of usable VAs.

Reiser and Lemon provide an excellent summary of the state-of-the-art in VA development in their recent book [12]. In short, they explain that a highly-effective method to kick start development of VAs is to conduct Wizard of Oz (WoZ) experiments to collect conversations between humans and (simulated) VAs. The data from those experiments can then be used to design a VA prototype via reinforcement learning. Later, as more people interact with the VA, real-world data can be collected. As mentioned in the previous section, the process boils down to collecting WoZ data and annotating the dialogue acts in that data.

There is a temptation to jump as soon as possible to training machine learning algorithms based on whatever data are available. This temptation should be resisted because the data collected to train a VA to do one task are generally not usable for training the VA to do other tasks. While one hopes that it would be possible to annotate a dataset once and train a VA for many different tasks, in fact it is necessary to re-annotate data based on task-specific criteria. For example, in API usage, different tasks could be obtaining usage code examples, or explaining rationale behind code, or determining if an API function has changed since last used; task-specific annotations must be obtained for each of these tasks. Nevertheless, as Tenbrink *et al.* [20] point out, different tasks often share similar high-level dialogue act types, from which low-level, more detailed dialogue acts can be derived.

The scope of this paper is to lay groundwork for building VAs for API usage in the future. To that end, we 1) conduct

WoZ experiments, 2) annotate the conversations with high-level dialogue act types, semantic content, and utterance relationships that are likely to be useful for many tasks, and 3) discuss the key tasks within API usage that occur in our data to suggest what detailed training should be conducted in future work, as well as other considerations for future researchers. This scope is already quite extensive, so we note that we do not yet attempt automatic classification of dialogue acts, training of VAs for strategy, or natural language generation.

2.2 Wizard of Oz Experiments

A "Wizard of Oz" (WoZ) experiment is one in which a human (the user) interacts with a computer interface that the human believes is automated, but is in fact operated by another person (the wizard) [13]. The purpose of a WoZ experiment is to collect conversation data unbiased by the niceties of human interaction: people interact with machines differently than they do with other people [21]. These unbiased conversation data are invaluable for kickstarting the process of building an interactive dialogue system. We direct readers to a comprehensive survey by Riek *et al.* [22] for further justification and examples of WoZ experiments.

In a WoZ experiment, researchers must provide the user and the wizard with some specific *scenario*. A scenario is "a task to solve whose solution requires the use of the system, but where there does not exist one single correct answer" [13]. A well-designed scenario promotes interaction relevant to the simulated system [23]. To that end, it is important for the scenario to place constraints on both the user and the wizard. Such constraints may include limitations on the resources available to the user, or the types of responses the wizard can generate. Researchers often provide the wizard with an interface that simplifies and expedites the process of generating a response [24].

The WoZ technique enables researchers to rapidly prototype and evaluate different system features or characteristics. Kelley [25] describes how researchers can use these simulations in an iterative design process, in which the wizard is gradually phased out as the simulated functionality is added in. Reiser and Lemon [12] examine subjective feedback solicited from users to determine if different wizard "strategies" (see *Dialogue Acts* below) are more or less effective. This feedback also serves as a baseline against which future iterations of a system can be compared.

2.3 Dialogue Acts

A "dialogue act" (DA) is a spoken or written act that accomplishes a specific purpose in a conversation, as introduced above. DAs are typically viewed as classes into which utterances can be categorized [26], such as greeting or information-elicitation.

A single dialogue act can perform multiple functions in a conversation simultaneously: for instance, the utterance "I have to work tonight" in response to an invitation functions as an informative act and as a rejection of the previous utterance. Dialogue act annotators can choose to provide separate labels for each relevant *dimension* of a DA, or they can use a one-dimensional annotation scheme that accounts for all relevant functional combinations [27]. Most popular DA annotation schemes provide anywhere from 3 to 10 different functional dimensions [28], [29], [30]. In practice, researchers typically select or create an annotation scheme to suit to their particular research problem [31].

A key observation is that there is a difference between the *strategy* involved in a conversation and the *language* used by people to implement that strategy [4] (even though a recent trend has been to train ML algorithms to capture both strategy and language at the same time [32], [33]). The language is represented as the actual words used to render an utterance, while the strategy is represented as the sequence of DAs used. E.g., the language "A: Hello. B: Hi. A: Where should we eat? B: At Joe's." versus the conversation flow/strategy: greeting, greeting, suggestion-elicitation, suggestion.

To create a dialogue system, both language and DA types must be known for utterances in example conversations [12]. A VA must learn to mimic good strategies in terms of DA flow (e.g. it must recognize that it should respond to a suggestion-elicitation with a suggestion). Once it knows that it should respond with a particular DA type (e.g. a suggestion), it must then collect the information to portray in an utterance (e.g. a restaurant to recommend) and then convert the information into an understandable utterance in natural language. As Serban *et. al* [9] point out in a recent survey, several datasets, especially involving WoZ experiments, have been created for a variety of domains to serve as starting points for training VAs.

2.4 Reinforcement Learning for Dialogue Agents

Reinforcement learning (RL) is one of three fundamental machine learning paradigms. Unlike supervised machine learning, which involves leaning from labeled data, and unsupervised machine learning, which involves identifying patterns in unlabeled data, reinforcement learning does not involve learning from a dataset; rather, it involves learning by directly interacting with an environment over a discreet number of time steps. Reinforcement learning is commonly used to kickstart autonomous agents, as it requires minimal pre-existing training data while enabling agents to learn long-term strategies. We refer readers to Sutton and Barto [34] for a comprehensive overview of RL theory.

In brief, we note that many RL problems are formulated as Markov Decision Processes (MDP). An MDP is a mathematical framework for decision-making that is defined by the tuple $\langle S, A, \mathscr{T}, \mathscr{R} \rangle$. S refers to a *state space*, which is the set of all reachable states for an agent in the MDP; this constitutes an agent's beliefs and view of its environment. A refers to an *action set*, which is the set of all actions that are available to the agent. $\mathscr{T}$ refers to a *state transition function*, which defines the likelihood of the agent transitioning to a new state s' from a previous state s after performing some action a. $\mathscr{R}$ refers to the *reward function*, which defines the agent's expected reward for transitioning from state s to state s' via action a. When implementing any reinforcement learning solution, the state space, action set, state transition function, and reward function must be clearly defined.

Reiser and Lemon [12] provide detailed guidance on training dialogue agents via RL by using WoZ data to select an action set and state space and determine appropriate

state transitions and rewards. The first step is to annotate both the wizards' and the users' dialogue acts. The annotations of the wizards' DAs are used to define the agent's action set, while the users' DAs are used to build a user simulator that defines the state transitions. The state space is determined by identifying the dialogue features (e.g. dialogue length, previous dialogue act, number of search results relevant to a query) that strongly influence agent actions. Finally, the reward function can be created by examining how certain positive and negative dialogue outcomes (e.g. the system successfully assisting the user with a task, or the user having to repeat a question several times) correlate with task outcomes, as measured by task completion and subjective user feedback scores.

Reiser and Lemon use a modular dialogue system architecture, in which dialogue strategy (or dialogue management) is trained separately from the other components of a dialogue agent, such as natural language understanding and generation. There is a large body of work supporting the use of reinforcement learning for dialogue strategy in a modular architecture; early work (e.g. by Singh *et al.* [35] and Scheffler *et al.* [36]) helped formalize the problem and demonstrated preliminary results, while more recent work (e.g. by Dhingra *et al.* [37] and Peng *et al.* [38]) have applied deep reinforcement learning to tackle complex problems with multiple subtasks and ambiguous states. Recent work (e.g. [39] and [32]) has also attempted to use end-to-end deep RL to train multiple dialogue components (e.g. language understanding and dialogue management) simultaneously; while these approaches eliminate or reduce the need for annotated data, they struggle with some task-oriented and domain-specific applications.

2.5 Related Work in Software Engineering

Related work in the software engineering (SE) literature can be broadly categorized as either supporting experimentation or prototype virtual agents. In terms of supporting experimentation, recent work at FSE'18 [10] is the most similar to this paper. In that work, the authors conducted WoZ experiments for debugging tasks and built an automated classifier for DA types. However, the one-dimensional DA annotations in that study were rather general, such as "statement" or "apiQuestion." This paper is different in that we have entirely new WoZ experiments for API usage and provide more thorough analysis by annotating additional DA dimensions. Work by Maalej *et. al* [19] is also key supporting experimentation, in that it explores what types of information programmers are provided in API documentation. Many studies exploring the use of language in and the contents of API documentation have been published in recent years, including [40], [41], [42], [43], [44], [45].

Prototype virtual agents are less common, but include APIBot [46] (a QA system for API documentation), WhyLine [47] (a natural language debugging tool), TiQi [48] (a natural language interface to query software projects), and Devy [49] (a VA that performs Git operations). A comprehensive survey was recently conducted by Arnaoudova *et. al* [50]. Our work most closely resembles APIBot [46], in that our ultimate goal is to build a system to help developers use APIs. However, there are several key differences: for instance, APIBot is not designed for multi-turn dialogues, and can not request additional information from the user or consider dialogue history. Furthermore, the authors of APIBot did not investigate what kinds of questions users would ask a VA in practice. Therefore, we view our work as complementing and enhancing existing work on VAs in SE.

This work follows a history of empirical studies in software engineering [51]. In their "roadmap" to empirical studies in SE, Perry *et. al* [52] emphasize that "empirical studies can be used not only retrospectively to validate ideas after they've been created, but also proactively to direct our research." Indeed, "exploratory studies" play an important role in motivating, guiding, and informing future work [53], such as an exploratory study on feature location processes by Wang et al. [54] that directly inspired subsequent improvements [55]. It is our hope that the present study will similarly facilitate the development of VA technology for SE.

TABLE 1
Programmer/wizard pairings in the WoZ experiments. Participants either worked locally (in an on-campus office) or remotely.

Wizard	Programmer	Scenario	# Sessions
Author	Local	libssh	2
Author	Remote	libssh	4
Local	Remote	libssh	2
Remote	Remote	libssh	7
Author	Local	Allegro	0
Author	Remote	Allegro	4
Local	Remote	Allegro	1
Remote	Remote	Allegro	10

3 Wizard of Oz Experiments

This section describes the Wizard of Oz experiments we designed to simulate the experience of programming with the help of a virtual assistant. We designed two scenarios in which programmers were asked to complete programming tasks using an API that was unfamiliar to them. The first scenario used the libssh networking library, while the second used the Allegro multimedia library. In lieu of documentation, we introduced the programmers to an "experimental virtual assistant," which we named *Apiza*. Unbeknownst to the programmers, Apiza was controlled by a human "wizard."

3.1 Participants

We distinguish between two participant roles in the experiments: the "programmers" and the "wizards." No participant served as both a wizard and a programmer. All participants filled out an entry survey describing their background and programming experience. Each participant either worked "locally" in a controlled environment or "remotely" from the location of his or her choice. Table 1 shows the number of experimental sessions conducted with local or remote participants. It also shows the number of sessions in which the first author participated. For example, there were 4 sessions using the libssh scenario in which the first author served as the Wizard and the Programmer worked remotely.

3.1.1 Programmers

We recruited 30 participants to serve as programmers. We recruited 2 locally through our university's Computer Science graduate program, 7 through various freelancer Subreddits, and the remaining 21 through the freelancing website Upwork. All programmers had experience using C in an academic or professional software engineering context and had not previously been exposed to the APIs used in our experiments.

Each programmer participated in a single session. We gave half of the programmers the libssh scenario, and the other half the Allegro scenario. The two local programmers participated in an on-campus office room using a laptop we provided. The rest of the programmers worked remotely in the environment of their choice, using their own computers. All programmers worked in a virtual machine running Ubuntu 16.04. We asked programmers to work from this virtual environment for two reasons: to ensure that all of the necessary libraries and compilation tools were properly installed and configured, and to reduce the number of potential distractions on the user's screen.

3.1.2 Wizards

We recruited 6 participants to serve as wizards. The first author served as the wizard for ten sessions. The other wizards – 2 computer science graduate students and 3 professional software engineers – served for between one and six sessions each. Qualifications for wizards were identical to the programmers: they had used C in an academic or professional software engineering context, and had not previously been exposed to the APIs in our experiment. One wizard only participated in the libssh scenario, one only participated in the allegro scenario, and four participated in both the libssh and Allegro scenarios. Two wizards worked locally in an on-campus office; the rest worked remotely. All wizards used their own computers.

In line with other WoZ experiments, we provided all wizards with a custom tool to draft messages and navigate documentation (shown in Figure 1). The tool allowed wizards to search for API components by keyword and category. Wizards could click on labeled sections of function documentation (e.g. Description, Returns, Parameters) to copy individual sections, or click on a function name to copy the entire documentation for that function. The tool also provided searchable links to the header files. All wizards were asked to familiarize themselves with the tool before their first session.

A feature of our study, differing from Wood *et al.* [10] but similar to Uller *et al.* [56] and Kruijff-Korbayová *et al.* [57], is that we hired wizards as experimental participants in addition to the programmers. The decision to hire multiple wizards allowed us to collect a more diverse set of wizard strategies, to help researchers determine optimal strategies based on the experimental outcome (since optimal strategies for wizards are not described in the literature). Similarly, by hiring wizards who were unfamiliar with the APIs and allow them to participate in multiple sessions, we intended to produce a learning effect; we anticipated that wizards would adopt new strategies as they became more familiar with the API upon completing successive sessions.

3.2 Scenarios

We created two scenarios, which consisted of sets of tasks based on two different APIs: the libssh API, and the Allegro API. We chose APIs from two different domains (networking and multimedia) in order to enhance the generalizability of this study. We chose the Allegro and libssh APIs in particular because they are both fairly large (with over 200 public functions and structs) and well-documented. Summaries of both scenarios are shown in Table 2.

Prior to the experiment, the first author completed the tasks in each scenario in order to ensure that the tasks were possible to complete and of reasonable difficulty. As described in Section 2.2, the goal was to design tasks that would increase the amount of meaningful interaction between the programmer and the wizard.

We prepared both scenarios ahead of time in the virtual machine used by the programmers. For each scenario, a programmer only needed to edit a single file containing some skeleton code. After making changes, the programmer was responsible for running a premade `make` file to observe the program's behavior and evaluate his or her progress. We didn't specify any IDE or text editor – the programmer was free to use any editor inside the VM. While this decision introduced some variability, it meant that programmers did not need to waste time adapting to a totally unfamiliar development environment.

3.2.1 Scenario 1: libssh

The first scenario involved using the libssh API to programmatically create and employ SSH network connections. It consisted of five tasks, presented in order of increasing difficulty. The programmer was instructed to complete these tasks in order.

The first task simply directed the programmer to compile the program and observe its behavior. This task did not require use of the API, but we included it to ensure the programmer understood how to evaluate his or her progress. The second task directed the programmer to create a new `ssh_session` object. We intended for this task to be simple, requiring only a single API call (`ssh_new`), in order for the programmer to become acquainted with Apiza.

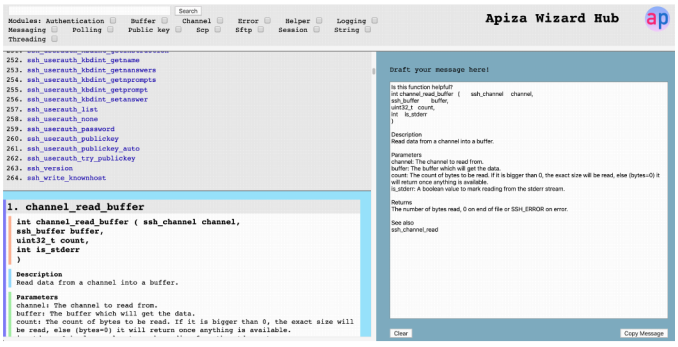

Fig. 1. Tool used by wizards in the WoZ experiments. The header bar allowed wizards to search for components by keyword and filter by categories defined in the API documentation. The left half of the interface displayed a list of the names of all components that satisfied the search, followed by the complete documentation for each of those components. The right half contained a text box used to draft messages.

TABLE 2
Summaries of the two experimental scenarios.

API	Domain	# Participants	# Tasks	Task Detail	Completed in Order	API Examples in Starter File
libssh	Networking	15	5	Direct instructions	True	False
Allegro	Multimedia	15	7	Open-ended	False	True

We divided each of the remaining tasks into a series of subtasks. The third task directed the programmer to set up the `ssh_session` created in the previous task by connecting to a server (the localhost), authenticating the server, and disconnecting from the server. (e.g. connecting to the server could be completed using the function `ssh_connect`). The fourth task directed the programmer to complete an empty method called `show_remote_user` by creating and opening an `ssh_channel`, executing the `who` command on the channel, reading the response, and shutting down the channel. This task used many methods similar to those previously used (e.g. `ssh_channel_new`), as well as new methods to execute on and read from a channel. The final task directed users to fill out an empty method called `sftp_operations` by creating and initializing an `sftp_session`, creating a new file in a new directory, writing a string to the file, and finally, and closing the file.

3.2.2 Scenario 2: Allegro

The second scenario involved using the Allegro multimedia library to add features to a simple video game. Additionally, there were a few key differences between the two scenarios.

Whereas programmers in the libssh scenario were provided a nearly empty source file to work with, we provided Allegro programmers a template program with several features already implemented (such as the display and core game loop). These features needed to be correctly implemented before any other interesting tasks could be completed, but they were too complicated and used too few API functions to serve as good tasks themselves.

Because we observed some programmers in the libssh scenario finish all five tasks with time to spare, we included seven tasks in this scenario. Additionally, we did not include the initial compilation and observation of program behavior as a separate task, as in the libssh scenario. We instructed the programmer to compile and observe the program before beginning the session, to allow more time to work on the tasks that actually involved the API.

Unlike the libssh scenario, we allowed the programmers to work through tasks in any order to prevent them from getting stuck on any one problem (in practice, nearly all programmers worked through them in the order provided). Also unlike the libssh tasks, these tasks provided only high-level descriptions of the features that were to be incorporated and 2-3 details or hints. As such, these problems were a bit more open-ended than those in the libssh set. This diversity in tasks was intended to procure a wider range of programmer-wizard interactions.

The first task directed the programmer to add keyboard functionality to the game. This required installing the keyboard subsystem, registering it as an event source, and checking for keyboard events. Though this task required the use of at least 4 API methods, they were analogous to methods already implemented in the template program (e.g. the `al_get_display_event_source` method may have hinted at the existence of the `al_get_keyboard_event_source` method).

The remaining tasks directed the programmer to add a "game over" sound effect, show a score on the display, draw images on the display, rotate the images appropriately, pause the game when the player clicked on the display, and make the display resizable. These tasks generally required similar steps, such as identifying the correct subsystems and handling events. The later tasks required the programmer to understand more complex aspects of the API.

3.3 Methodology

At the start of each session, we instructed the programmer to open the virtual machine testing environment and login to a Slack channel for communication. At the same time, we had the wizard participant login to the same Slack channel using an account named "Apiza".

In Slack, we provided the programmer with a full description of the scenario, consisting of the list of specific tasks to complete with the unfamiliar API (as described in the previous section). We asked that all questions relating to the API be directed via Slack text messages to our "experimental virtual assistant" called Apiza. We explained that Apiza was an "advanced AI," able to carry out "organic, natural-language conversation" and discuss "both high-level and low-level functionality."

Once the programmer confirmed that he or she understood the description and the tasks, we started a timer and instructed the programmer to begin. For the next 90 minutes, the programmer worked through as many of the tasks as he or she could. Throughout, the programmer sent messages to the wizard, who answered them as quickly and correctly as he or she could. We instructed the wizard that his or her responses didn't need to seem "robotic," but at no point was the wizard to reveal that Apiza was a human.

During the session, we did not allow the programmer access to the API's documentation. While this does not necessarily represent the most-likely use case for a VA for API usage, it is a constraint necessary in the vast majority of WoZ experiments to force programmers out of their habits and into using the experimental tool [22]. However, we did permit the programmer to search the internet for general programming questions (e.g. related to C syntax) in order to narrow the scope of the dialogue.

Unlike the programmer, the wizard had access to the API documentation and header files via the tool described in Section 3.1.2. Additionally, we permitted the wizard to search for API-related information on the internet. The only factor limiting the wizard's access to information was the pressure to generate a timely response.

When the time ran out or the programmer finished all of the tasks, we instructed the programmer to stop working. We then asked the programmer to send us his or her code and the list of all URLs visited in the course of the study. We also asked them to complete an exit survey; as recommended by Reiser and Lemon [12], the exit survey included the PARADISE [58] questions, which rated user satisfaction on a 5-point Likert scale.

3.4 Data Collection

We collected six key data items for every experimental session:

1) The programmer's entry survey.
2) The wizard's entry survey.
3) The dialogue between the programmer and the wizard.
4) The source code written by the programmer.
5) A record of any websites the programmers visited during the session for general C syntax questions.
6) The programmer's exit survey.

Additionally, some programmers elected to provide additional feedback about their experience with Apiza.

3.5 Threats to Validity

As in any experimental study, our experimental design carries a number of threats to validity, including human factors, the participant selection process, the details of the experimental scenarios, and the design of the communication interface. Here, we will briefly address these threats, and explain the steps we took to mitigate them.

Human factors. Human factors, such as fatigue, distraction, and failure to follow the task guidelines, had the potential to impact each individual participant's performance. These threats were especially present for the majority of trials that were conducted remotely. To reduce the effects of fatigue and distraction, we limited each experimental session to 90 minutes. We also attempted to curb distraction by providing distraction-free virtual environments to both the programmers (the virtual machine testing environment) and the wizards (the custom tool). To encourage programmer participants to follow the task guidelines, we included all pertinent information and restrictions in the document detailing the experimental tasks. Before beginning sessions, we asked them to read through the document and gave them the opportunity to ask any questions. Still, despite our instructions to direct all API-related questions to Apiza and only use online resources for general C syntax questions, we observed a small number or instances in which programmers searched for API information on the internet. These were infrequent occurrences, and they were documented in the internet history sent at the end of the session.

Participant selection. We invited programmers from a wide range of backgrounds to participate, in order to collect a diverse array of programmer dialogue strategies. However, this leniency means that it is possible that our results and conclusions may have been different with a different set of programmers. We attempted to mitigate this risk by recruiting a fairly large number of programmers (30) and having them complete entry surveys to document their backgrounds. Similarly, the wizard strategies that we observed may have changed had a different set of wizards participated, or if the same wizards had completed more or fewer sessions. We recruited a fairly large number of wizards (6) for a WoZ study, and we had many of them participate multiple times in order to observe possible learning effects. We believe that our participant selection process struck a reasonable balance between experimental control, generalizability, and practicality.

Experimental Scenarios. The choice of APIs and the design of the particular tasks we asked participants to complete in each scenario may have also had an effect on the types and quantities of interactions that occurred. The domain of each API and the quality of its documentation likely affected the wizard's ability to find relevant components and make recommendations, as well as the programmer's ability to determine which components were appropriate and how to use them. Similarly, the specificity and wording of the tasks may have influenced the types of questions programmers asked. However, there is no single API or task set can generalize to all APIs and tasks. Therefore we chose two APIs from different domains and made one task set more open-ended than the other in order to broadly gauge whether there would be any impact on the dialogues. Still, the many differences between the two scenarios (such as the different number and quality of tasks) means that any differences observed between them in the results cannot clearly be attributed to any individual variable.

Communication interface. Finally, the chosen communication interface may have impacted the results of the study. Being limited to text-based communication over Slack may have affected the amount of information wizards decided to include in individual messages (e.g. they may have suggested fewer API components in spoken messages, or more components if they had been able to link directly to documentation). At the same time, the interface may have affected the frequency and wording of programmer questions. As analyzing the differences between communication modalities was not in the scope of this study, we felt that text-based communication via Slack would suffice, as Slack is a fairly well-known messaging platform with few restrictions on message length or frequency, and one for which many developers have created actual bots.

4 Experiment Results

In this section, we present the key experimental results of our WoZ experiments. We outline the basic structure and statistics of the collected dialogues, and briefly examine the programmers' task performance and satisfaction with the simulated system.

4.1 Dialogues

We collected 30 API usage dialogues. In general, the programmer and the wizard sent messages in turn. Most frequently, these were question-answer pairs, with the programmer querying some functionality of the API and the wizard providing the answer. Other types of interactions occurred as well, such as greetings, assessments, and side-exchanges – these are explored in greater detail in Section 6. The following excerpt typifies the interactions that

TABLE 3
Comparison of our corpus to other WoZ corpora. The numbers of words and unique words in each corpus are shown where available.

Task Domain	# of Dialogues	# of Turns	# of Words	# Unique Words
APIs (this paper)	30	1947	47928	3190
Debugging [10]	30	2243	50514	4162
Design [59]	31	3606	27459	–
General QA [60]	33	2534	125534	–
Audio player [61]	72	1772	17076	–
Tutoring [62]	37	1917	12346	–
Mission planning [63]	22	1738	–	–

occurred in the dialogues. Messages are labeled with "PRO" or "WIZ", denoting the speaker as a "Programmer" or "Wizard," respectively.

PRO: allegro keyboard input

WIZ: You can save the state of the keyboard specified at the time the function is called into the structure pointed to by `ret_state`, using `al_get_keyboard_state`

PRO: Whats the function signature for `al_get_keyboard_state`

WIZ: `void al_get_keyboard_state( ALLEGRO_KEYBOARD_STATE *ret_state)`

Across all dialogues, participants collectively generated 1927 Slack messages (also referred to as "turns"). Wizards and programmers sent similar quantities of messages, averaging to 31.8 messages/dialogue sent by programmers and 33.1 messages/dialogue sent by wizards. The frequency was also similar across the two tasks; participants sent an average of 68.5 messages in th libssh scenario, compared to 61.3 sent in the Allegro scenario.

The dialogues contain a total of 47928 word tokens[1] with a vocabulary size of 3190 words. Wizards used considerably more words (41185) and drew from a larger vocabulary (2988) than programmers, who used 6743 words and 880 unique words. This disparity in word usage is to be expected; programmers manually wrote the majority of their messages' content, and had access to only limited information. By contrast, wizards frequently copied and pasted large chunks of pre-written documentation to send to the user. This corpus is similar in size to published WoZ corpora across a broad range of domains, as shown in Table 3.

4.2 Task Completion

Not every programmer completed every task. This fact is valuable because the task completion rate can be used as a metric to judge the efficacy of different wizard strategies. Recall that we hired 6 wizards, who each completed between 1 and 10 sessions, providing us with a diverse variety of wizard strategies. As Reiser and Lemon [12] describe, identifying which strategies lead to better or worse outcomes can help enables VA designers to determine which behaviors are desirable in a VA. While this extensive analysis is beyond the scope of this paper, we present the programmers' task completion rates in order to characterize the corpus.

[1]Word tokens were generated using the `word_tokenize` method from Python's `nltk.tokenize.punkt` module.

TABLE 4
Programmers' performance on the tasks. "Attempt rate" refers to the percentage of all programmers that wrote at least one line code directly related to the task. "Success rate" refers to the percentage of those programmers that successfully completed the task.

Scenario	Task	Attempt Rate (%)	Completion Rate (%)
libssh[2]	A	100.0	100.0
	B	100.0	78.6
	C	100.0	21.4
	D	71.4	40.1
	E	51.1	28.0
Allegro	A	100.0	73.3
	B	93.3	78.6
	C	80.0	33.4
	D	40.0	66.8
	E	6.7	100.0
	F	6.7	100.0
	G	0.0	N/A

Specifically, we present two metrics: the task *attempt rate* (that is, the proportion of programmers that attempted a task) and the task *completion rate* (of the programmers who attempted a task, the proportion who completed it). We considered a programmer to have "attempted" a task if he or she wrote at least one line of code directly related to the task. If it was ambiguous whether a line of code related to a task, we referred to the dialogue for additional context. We considered a programmer to have "completed" a task if, in our judgment, he or she wrote all of the code to satisfy all of the task's requirements, even if the code did not compile or execute. Table 4 shows programmers' attempt and completion rates for each task. For instance, 71.4% of all programmers attempted Task D of the libssh scenario; of those programmers, 40.1% completed the task.

Attempt and completion rates for different tasks varied between 0% and 100%. We generally observed lower attempt and completion rates for the later tasks, which were more difficult and for which the programmers may have had less time, depending on their performance on earlier tasks. Tasks E and F of the Allegro scenario each have a success rate of 100% because only one programmer attempted and completed those tasks. Programmers generally finished one task before moving to the next; however, they occasionally moved on from a task without successfully completing it.

4.3 User Satisfaction

In addition to observing the programmers' objective performance on the tasks, we also asked programmers to fill out surveys subjectively rating their satisfaction with the "virtual assistant" system on a 5-point Likert scale. These questions serve as another metric to evaluate different wizard strategies. The survey questions were taken from the `PARADISE` [58] framework for automatic dialogue evaluation. To measure different dimensions of user satisfaction,

[2]Only 14 sessions are considered here, as one participant in the libssh scenario did not submit a source code file.

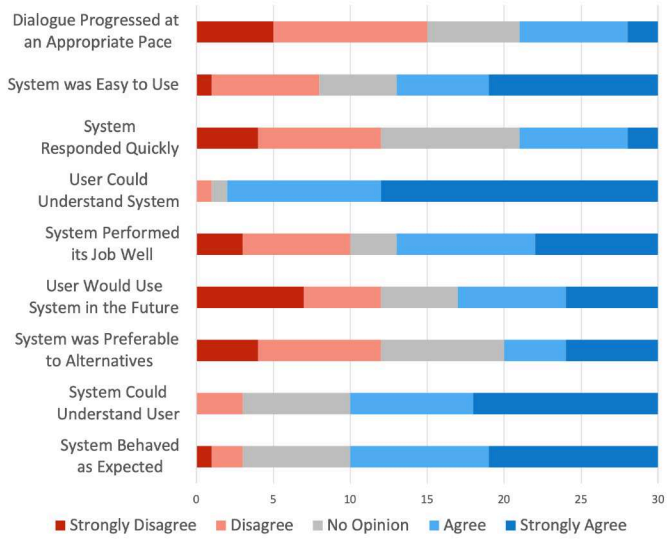

Fig. 2. Programmers' PARADISE ratings for the Apiza system.

the authors of the PARADISE framework created a set of 9 survey questions corresponding to features such as "Interaction Pace" and "Task Ease." These questions have been widely adapted and used in subsequent dialogue studies [12]. We include the full list of survey questions and programmer responses in our online appendix.

The features corresponding to the system's ability to understand user utterances and the understandability of the system's utterances were the most highly-rated, scoring around 4 points on average. The features related to the pace of the conversation and the speed of the system's response received the lowest ratings, around 2.8 points on average. Scores varied greatly across raters; the harshest rater gave an average score of 1.5, while the most generous rater awarded an average score of 4.9.

That the features relating to interaction pace generally scored relatively low and those related to system understanding scored high is not surprising given the nature of the Wizard-of-Oz deception. For users accustomed to real virtual assistants, the system's high intelligence came as a pleasant surprise, while its sluggishness was a source of frustration. As one programmer wrote in a comment, "Most of the time the first answer given by Apiza directly answered the question [...] The wait was a bit annoying though. I'm used to getting answers (well, search results) within a few seconds of entering a query."

4.4 Other Observations

We made several additional observations regarding the dialogues. First, we noted that many programmers enjoyed interacting with Apiza and testing the system's capabilities, going so far as to ask questions like "How's it going?" and "How is the weather where you are?" It would not be crucial that a real VA for API usage be able to handle these types of unrelated questions, but users may use similar questions to explore the system's functionality.

Despite being allowed to search the internet for general C syntax questions, several programmers chose to direct these questions to Apiza. Conversely, a small number of programmers grew frustrated by Apiza's inability to provide certain information and decided to search for API documentation online. In total, 16 of the 30 programmers searched for information on the internet; of those, 5 wound up accessing resources directly related to the API.

In the dialogues, we observed habits that programmers may have learned by interacting with other virtual assistants and tools. Several programmers avoided the use of pronouns to refer to recently mentioned API components. They may have wanted to be as clear as possible in the event that Apiza would be unable to resolve the antecedent. Some programmers treated Apiza like a search engine, simply querying vagues terms without specifying the type of information desired.

As noted in Section 4.1, the interactions in the dialogues typically followed a simple question-answer structure, in which the programmer sought information and the wizard provided it. This was not always the case, however. Sometimes wizards sought information from the programmer (e.g. clarification questions); other times, programmers proactively offered information. Occasionally, a speaker would send multiple messages in a row, as demonstrated in the following excerpt:

> PRO: if I have an ALLEGRO_EVENT called ev, should ev.type == ALLEGRO_EVENT_KEY_DOWN return true when I press a key?
>
> WIZ: I'm not sure I understand
>
> WIZ: Are you interested in al_get_keyboard_state or al_key_down?
>
> PRO: no
>
> PRO: tell me about al_get_keyboard_state

In total, 15.7% of the wizards' messages and 18.3% of the programmers' messages were followed by one or more messages from that speaker. Furthermore, speakers often expressed multiple, separate ideas and intentions within a single message. Identifying these utterances and parsing their meanings are key challenges a virtual assistant needs to overcome. The next section describes the first steps we've taken towards tackling these challenges.

5 Corpus Annotation

We annotated the dialogue acts in the WoZ corpus along four dimensions. As described in Section 2, dialogue act annotations are needed to train virtual assistants to perform specific tasks. Different annotation schemes are needed to facilitate different functionality in a VA – to attempt to generate a comprehensive list of all dimensions for all functionality is not in line with findings from related literature [64]. Instead, we provide a first round of high-level annotations as a foundation on which future VA designers can build.

5.1 Research Questions

We investigated four specific research questions:

RQ_1 What is the composition of the corpus in terms of **illocutionary** dialogue act types?

RQ_2 What is the composition of the corpus in terms of **API** dialogue act types?

RQ_3 What is the composition of the corpus in terms of **backward-facing** dialogue act types?

RQ_4 What is the **traceability** of specific API components in the corpus?

We addressed each of these research questions by annotating the conversation corpus along a different dimension. The annotation schemes are described in detail in 5.2.

The rationale for RQ_1 is to discover the conversational "flow" of the API dialogues. An "illocutionary" dialogue act type describes the illocutionary intent behind an utterance e.g. distinguishing between a statement intend to inform and a statement intended to suggest future action. Annotating illocutionary DA types enables us to train a VA to identify the intent behind a user's utterance and to predict the appropriate type of response (e.g. whether to respond to a question with an answer or a follow-up question).

The rationale for RQ_2 is to evaluate the domain-specific content of the API dialogues. Whereas illocutionary dialogue act types label utterances as "questions" or "statements," API dialogue act types describe what types of API knowledge (such as functionality, usage patterns, or examples) are addressed in each utterance. As Wood *et al.* [10] point out, a virtual assistant must be able to identify the type of domain-specific knowledge a programmer asks about in order to respond with a relevant answer.

The rationale for RQ_3 is to evaluate relationships among the utterances in the API dialogues. We observed that the programmer and the wizard did not always engage in simple question-answer turn-taking. Rather, dialogues sometimes became complex, with multiple conversational "threads" potentially progressing in parallel. A virtual assistant should be able to track separate threads in order to draft contextually appropriate responses.

The rationale behind RQ_4 is to identify and track the specific API components that are addressed throughout the dialogues. This identification is related to the concept assignment problem, as it involves connecting specific software components to their relevant, natural-language concepts. We refer to this identification as "traceability," as it relates to the concept of traceability in software engineering [65]. A virtual assistant for APIs must be able to identify the API elements relevant to a user's utterance (even when those elements are not mentioned by name).

5.2 Methodology

This section details the annotation scheme used to investigate the research questions detailed above. See Table 5 for summaries of each scheme and example labels. The full list of labels for each dimension can be found can be found in that dimension's source.

5.2.1 Segmentation

Before assigning any labels, we first had to segment the Slack messages into individual segments, or utterances. McCowan *et al.* [18] emphasize that dialogue acts reflect speaker intention, and in their guide, they recommend that "each time a new intention is expressed, you should mark a new segment." The length of an utterance is variable; it may consist of a single word or entire paragraphs, depending on the speaker's intention. In our experiments, programmers and wizards often expressed multiple intentions, responded to multiple utterances, or referenced multiple API components in the course of a single message. Therefore, it was important to segment the messages before labeling.

Our corpus presented a unique segmenting challenge: wizards often shared verbatim chunks of documentation to the programmer. These chunks could contain several paragraphs worth of information. To segment each individual utterance of the verbatim documentation would not quite be appropriate, as they do not necessarily represent separate intentions on the part of the wizard. On the other hand, to group an entire chunk of documentation into a single segment would be too coarse grained, as wizards had the ability to purposefully include and exclude certain types of information using the documentation tool described in Section 3.1. Given that fact, we chose to segment chunks of documentation by topic (e.g. discussion of all parameters was included in a a segment, separate from discussion of the return values).

5.2.2 Illocutionary Dialogue Act Types

We labeled illocutionary dialogue act types using the so-called "AMI labels." The AMI corpus, presented by McCowan *et al.* [18], provides a coarse-grained set of labels for illocutionary DA types that are applicable to many types of conversations. As Gangadharaiah *et al.* [70] point out, a useful place to start annotation is with a set of 10-20 coarse-grain labels to provide a common comparison point with other datasets, even though these annotations alone are not sufficient for industrial use.

Our methodology for annotation was straightforward: McCowan *et al.* [18] provide an annotation guide with detailed instructions for every DA type. We followed this guide for every conversation, labeling each utterance with the most appropriate label. Note that we annotated both sides of the conversations, wizard and programmer, even though a virtual assistant would only need to classify the dialogue acts of the programmer – it would know the wizard's dialogue act types because it would have generated them. However, we annotate both sides anyway, since we are interested in the wizards' conversation strategies and providing guidance to designers of virtual assistants. This decision is in line with recommendations by Wood *et al.* in a study of software engineering VAs [10].

To analyze the distribution of illocutionary DA types, we compared our corpus to two others: the AMI meeting corpus [18] and the WoZ debugging corpus [10]. The AMI corpus consists of spoken multi-party discourse in simulated meetings, while the debugging corpus consists of written dialogues between programmers and wizards completing a series of bugfixing tasks. We compared the frequencies of each illocutionary DA type in our corpus to each of these corpora. The frequencies didn't account for three of the original AMI labels (STALL, FRAGMENT, and BACKCHANNEL) that were primarily relevant to spoken

TABLE 5
The four dimensions along which the corpus was annotated.

RQ	Dialogue Act Dimension	Source	Summary	Examples
1	Illocutionary Dialogue Act Types	[66]	14 labels describing the forward-facing illocutionary force of an utterance.	`INFORM, ELICIT-INFORM, SUGGEST, ...`
2	API Dialogue Act Types	[67]	11 labels describing the API information referenced in an utterance.	`FUNCTIONALITY, PATTERNS, EXAMPLES, CONTROL FLOW, ...`
3	Backward-Facing Dialogue Act Types	[66]	7 labels describing the relationship of an utterance to a previous utterance.	`POSITIVE, NEGATIVE, PARTIAL, ...`
4	Traceability	[68], [69]	All relevant components in an API.	`ssh_session, ALLEGRO_KEY_-DOWN, ssh_disconnect(ssh_-session session)...`

modalities, to allow for a more direct comparison between the AMI corpus and the WoZ corpora.

We include this comparison for two reasons. First, we want to establish some context for the distributions we observe in our dataset. Qualitatively analyzing whether certain DA types see particularly high or low usage in API usage dialogues requires us to know how frequently they are used in other dialogues in similar and dissimilar domains. Second, if the distribution of illocutionary DA types in API usage dialogues is evidently similar to that in other domains, it may encourage researchers to consider using certain transfer learning techniques for tasks such as dialogue act classification.

5.2.3 API Dialogue Act Types

The API dialogue act types in each utterance were labeled according to the taxonomy proposed by Mallej and Robillard [19]. In their work, they generated a set of 12 broad categories that may be used to classify the information types present in API documentation. Tian *et al.* [46] applied these label not only to API documentation, but to questions about documentation as well. By training a model on API question/answer pairs associated with these labels, they were able to achieve high performance on an API information type retrieval task.

We followed the annotation guide provided by Mallej and Robillard, which describes in detail every knowledge type and provides suggestions to resolve uncertainties. We labeled API DA types in utterances by both the wizards and the programmers, but we only labeled utterances that actually contained API information. Unlike the illocutionary DA annotation scheme, Mallej and Robillard explicitly allowed for multiple labels to be applied to a single unit.

We found that the knowledge type that Mallej and Robillard referred to as "NON-INFORMATION" actually encompassed valuable information in the context of these dialogues, such as the names of functions, their parameters, and their return values. As such, we decided to rename the "NON-INFORMATION" class to "BASIC."

We compared the distribution of API DA types to the type distribution found in the documentation for the JDK 6 Java platform, which was one of two systems that Maalej and Robillard originally coded with API knowledge types [19]. There are several substantive differences between the API usage corpus and the documentation for JDK 6: they involve different APIs using different programming languages and communicate information for different purposes. Furthermore, the distribution of API information types in the JDK documentation is not necessarily representative of the distribution in all API documentation. As such, the comparison between these two sources was intended to identify overall trends, rather than fine-grained distinctions.

5.2.4 Backward-Facing Dialogue Act Types

We used another layer from the AMI annotation scheme [18] to capture the relationships among utterances. This layer consists of only four backward-facing dialogue act types: "POSITIVE," "NEGATIVE," "PARTIAL," and "UNCERTAIN." Each of these covers a wide range of relationship types; for instance, "POSITIVE" can imply agreement with a previous utterance, understanding of a previous utterance, or an attempt to provide something that the previous utterance requested.

We again followed the AMI guidelines, marking relationships among utterances. In the AMI scheme, an utterance can only relate directly to a single prior utterance, so an annotation consisted of a single label and the ID of the related utterance. Any utterance could relate to any prior utterance by either speaker. Any utterance without a backward-facing function (e.g. an utterance starting new lines of questioning) was given a placeholder "[NONE]" label.

In the course of annotating relationships, we found that the four backward-facing dialogue act labels did not provide enough granularity to satisfactorily capture all of the observed relationships between utterances. We noted that other dialogue act labeling schemes (such as DAMSL [28]) provide more distinction between different types of relationships (e.g. DAMSL distinguishes among "Accept," "Acknowledge," and "Answer" relationships, all of which AMI would classify as "POSITIVE"). Therefore, we chose to add three additional labels to the original AMI set of backward-facing dialogue act type labels in order to more meaningfully evaluate the relationships that occur in our specific corpus.

The three labels that we added were "REPEAT," "FOLLOW-UP," and "CONTINUE." The "REPEAT" label was used to mark repetitions, repairs, or rephrasings of a previous utterance. The "FOLLOW-UP" label was used to mark acts that followed up on a previous utterance, either by asking a question predicated on the utterance, or providing unprompted information or suggestions based upon the previous utterance. The "CONTINUE" label was intended to link contiguous utterances by one speaker that

form a single question or response. For instance, Apiza often responded to requests for documentation by sending information about parameters, return values, functionality, etc. These responses each comprised several utterances – it would be inaccurate to label each utterance as having a totally separate relationship to the original query.

5.2.5 Traceability

The "traceability" annotation of specific API components was inspired by topic- and entity-labeling methodologies in NLP and SE [71], [72], [73]. This annotation layer did not have a predefined label set per se – rather, every component of the API could potentially be used as a label. We went through every utterance and labeled any API components that were the "topic" of the utterance. An API component did not need to be referenced by name to constitute a topic, and direct references to a component did not necessarily make them a topic. Rather, the decision was heavily based on context, the intention of the speaker, and the retroactive role of the component in the conversation.

For instance, the utterance "How do I ensure a session was successfully created?" does not explicitly name any component of the libssh API – however, the "session" is in reference to an `ssh_session` struct, and the question is asked directly after a discussion of the `ssh_new()` method. Therefore, both the `ssh_session` struct and the `ssh_new()` method are labeled for that utterance.

5.2.6 Note on Reliability

In any annotation process, it is important to consider the *reliability* of the annotations, or "the extent to which different methods, research results, or people arrive at the same interpretations or facts" [74]. Bias, fatigue, and other factors may cause an individual annotator to produce inconsistent or unreliable results [75], [76].

It is common for researchers to gauge the reliability of their annotations by asking multiple, independent annotators to annotate data and then calculating an agreement score using e.g. Cohen's kappa or Krippendorff's alpha [77]. Establishing reliability is especially important in "conventional" qualitative analysis, or "open-coding" processes in which annotators do not use predetermined sets of labels. In those cases, agreement among independent annotators does not just indicate the reliability of a particular set of annotations, but rather, the reliability of the annotation scheme as a whole.

However, the act of calculating agreement does not itself improve reliability. Furthermore, agreement scores are notoriously difficult to interpret (e.g. while an agreement score of .8 is generally considered to indicate high reliability, it is not sufficient for applications that are "unwilling to rely on imperfect data" [78]).

By contrast, we performed "directed" qualitative analysis; that is, we annotated the corpus using preexisting sets of labels. Our priority was not to measure the reliability of the existing annotation schemes, but to ensure the correct application of those schemes. To achieve unbiased results in this sort of analysis, Hsieh and Shannon [75] suggest using an "auditing" process, in which experts discuss the application of label sets and resolve any ambiguities. This type of procedure has frequently been used in the social sciences [79], [80], and more recently, in software engineering [10]. This process allows for the creation of a single, higher-quality set of annotations for use in applications that are less willing to rely on imperfect data [10].

We followed this procedure: the first author annotated the corpus following the guidelines for existing annotation schemes for each relevant dimension. Whenever there was some ambiguity as to the correct application of an annotation scheme, the first and third authors discussed the situation and decided on a correct implementation (or, in some cases, modifications to the scheme). Although this auditing process does not allow us to calculate an agreement score, it allowed us to generate a single set of "more accurate" annotations [75].

Still, our intention is for these high-level annotations to guide the creation of a API usage VA; as described in Section 2.3 and Section 2.4, more task-specific annotations will be needed to enable particular functionalities. We acknowledge that future researchers who wish to apply these annotations directly may be wary of the fact that a reliability metric cannot be calculated, and choose to independently reannotate the corpus to address those concerns.

6 Annotation Results

In this section, we discuss the results of the annotation process described above. Before assigning any labels, we segmented the programmers' and the wizards' messages into discreet utterances, as described in Section 5.2.1. We ultimately segmented the 1947 messages in the corpus into 3183 utterances. Programmers' messages contained 1.1 utterances on average, with 6% containing more than one utterance. Wizards' messages contained 2.2 utterances on average, with 57% containing more than one utterance.

6.1 RQ1: Illocutionary Dialogue Act Types

Figure 3 shows the composition of the corpus in terms of illocutionary dialogue act labels. Programmers most frequently used dialogue acts of the ELICIT-INFORM and ELICIT-OFFER-OR-SUGGESTION illocutionary types, collectively accounting for approximately 80% of all programmers' dialogue acts. Wizards primarily used dialogue acts of the INFORM type, accounting for nearly 74% of their dialogue acts. The next most common label for the wizards was SUGGEST, accounting for 13% of their labels. These preferences seem to reflect the task goals and the participants' different roles in the API dialogues: programmers sought to learn how to use the API, and wizards provided the desired information.

We compared the distribution of AMI labels in our corpus to two others: the AMI meeting corpus [18] and the WoZ debugging corpus [10]. As shown in Figure 3, the three corpora shared a few traits: the relatively high frequency of INFORM acts and the relative rarity of the BE-NEGATIVE, COMMENT-ABOUT-UNDERSTANDING, and ELICIT-COMMENT-ABOUT-UNDERSTANDING acts.

Beyond those similarities, the distributions varied substantially among the three corpora. The participant role (wizard or programmer) affected the frequencies of illocutionary types in both of the WoZ corpora. Compared to the

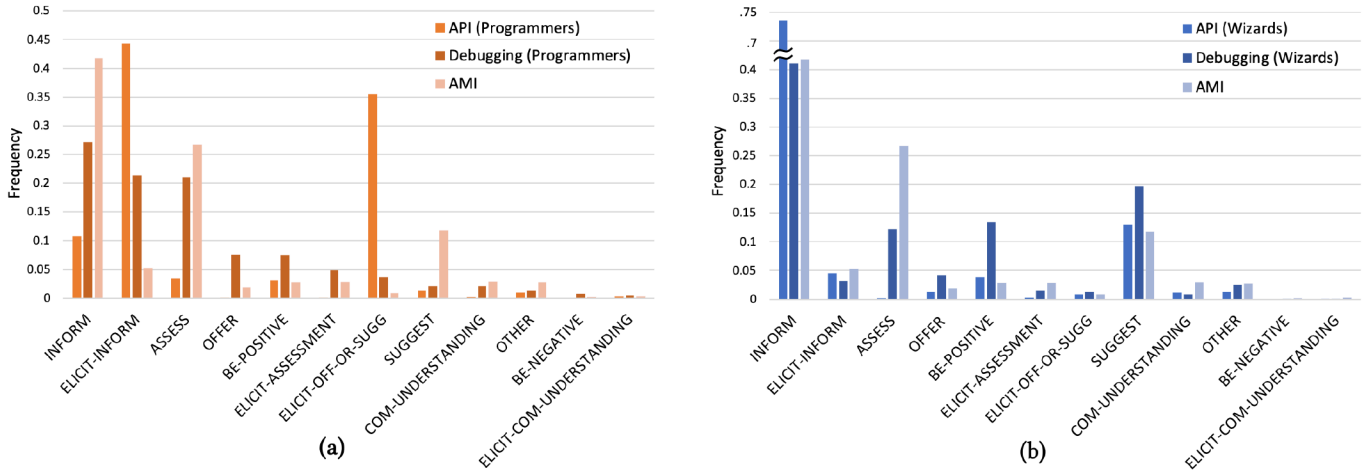

Fig. 3. Frequencies of illocutionary dialogue act types in our API corpus, the debugging corpus by Wood *et al.* [10], and the AMI meeting corpus [18]. (a) shows programmer frequencies, while (b) shows wizard frequencies.

speakers in the AMI corpus, programmers in both WoZ studies used more dialogue acts of the ELICIT-INFORM and ELICIT-OFFER-OR-SUGGESTION types and fewer of the INFORM and SUGGEST types, while the opposite was true of the wizards. These tendencies were much more pronounced in the API study than the debugging study; wizards in the API study used INFORM acts nearly twice as frequently as wizards in the debugging study, and programmers used ELICIT-INFORM acts more than twice as frequently. Other notable differences include the relative lack of ASSESS and OFFER act types in the API corpus compared to the other corpora.

We observe that the distribution of illocutionary dialogue act types in the API corpus is highly imbalanced, even when compared to that of another WoZ corpus in the software engineering domain. This imbalance has a few implications for researchers looking to build an intelligent VA for API usage. On one hand, it is more difficult to train dialogue act models on skewed datasets [81]. E.g. a classifier trained on this corpus would be unlikely to correctly identify an utterance of the OFFER type, as it would have been exposed to very few training instances of that type. Researchers have previously used techniques such as downsampling [81], oversampling, and SMOTE [10] to counteract the effects of imbalanced data in dialogue act modeling, but these solutions are typically poor replacements for real data.

On the other hand, this imbalance means that VA designers can narrow the range of interaction types supported by a VA for API usage while still providing the desired functionality. Because a few key illocutionary DA types account for the vast majority of turns, VA designers may choose to focus on those types and group the rest into the OTHER category. Doing so would inevitably reduce the expressiveness of the dialogue system, but it would ultimately be easier to train due to the constrained state and action space [12].

6.2 RQ2: API Dialogue Act Types

We annotated 2826 utterances with API dialogue act types across 1790 messages. 94% of programmers' messages and 91% of wizards' messages were labeled with at least one API DA type. Figure 4 shows the composition of the corpus in terms of API dialogue act labels.

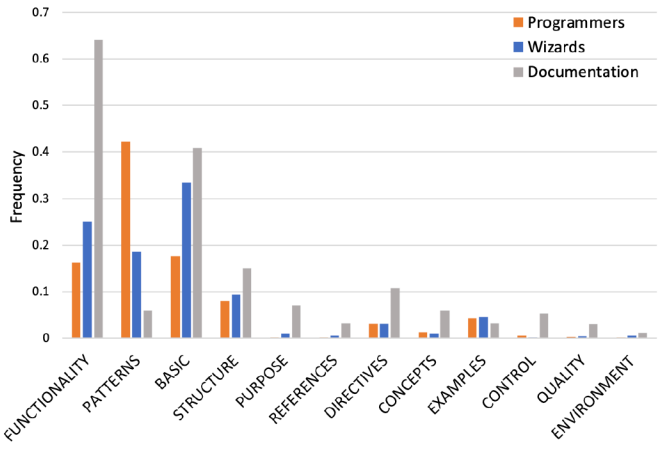

Fig. 4. Frequencies of API dialogue act types in our API corpus and the documentation for JDK 6 [19]

Programmers most frequently asked for information about PATTERNS (accounting for nearly half of their utterances), describing *how* to accomplish a specific objective with the API, followed by queries about BASIC information and FUNCTIONALITY details. Wizards referenced the same three API types most frequently, but the distribution differed: BASIC information was present in about 30% of the wizards' utterances, FUNCTIONALITY in about 25%, and PATTERNS in about 20%.

Overall, the distribution of API dialogue act types was somewhat similar to that of the JDK documentation [19], with a few notable exceptions. Compared to documentation, the dialogues contained a much higher proportion of PATTERN information, and a much smaller proportion of PURPOSE and CONTROL information. The rest of the labels followed a few trends in both the documentation and the corpus: FUNCTIONALITY, BASIC, and STRUCTURE information appeared fairly frequently, while the QUALITY, REFERENCES, and ENVIRONMENT types were rare.

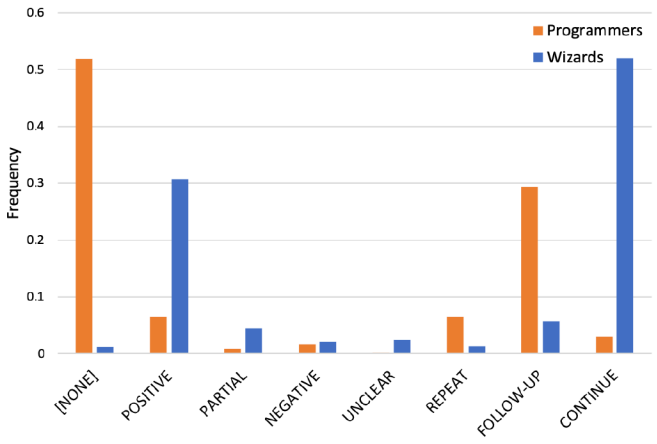

Fig. 5. Frequencies of backward-facing dialogue act types in the corpus.

Again, we observe an imbalanced distribution of API DA types. This imbalance presents researchers with challenges and opportunities similar to those discussed in RQ1. Specifically, while it would be difficult to train some ML models on this skewed dataset, VA designers may choose to provide functionality for only the most common API DA types. We discuss these implications in greater detail in Section 7.

6.3 RQ3: Backward-Facing Dialogue Act Types

Figure 5 shows the the distribution of backwards-facing dialogue act types used by the wizard and the programmer. The majority of the programmers' dialogue acts were assigned the NONE label (meaning they did not have a direct relationship to any previous utterance) or the FOLLOW-UP label. The PARTIAL, NEGATIVE, and UNCLEAR tags saw minimal use by the programmers. The majority of the wizards' dialogue acts were assigned the CONTINUE label, primarily due to the segmentation method used for verbatim chunks of documentation. The next most frequent label was POSITIVE, and the least frequent labels were START and REPAIR. These distributions demonstrate that many programmer-wizard interactions took place in the context of previous utterances. Oftentimes, the wizard relied on this context to produce an appropriate response. The following excerpt demonstrates one such situation:

> PRO: Apiza, what is the command to create a new ssh_session in the libssh API?
>
> WIZ: The command to create a new ssh session is: ssh_session ssh_new(void) [...] Returns A new ssh_session pointer, NULL on error.
>
> PRO: Just to confirm, is the ssh_session type a pointer type?
>
> WIZ: ssh_session is not a pointer type.
>
> PRO: But when you gave me the command to create a new ssh session, you said it "Returns" "A new ssh_session pointer...". I'm confused.
>
> WIZ: ssh_new returns a pointer to type ssh_session

In that exchange, the wizard needed to recall the context of the method `ssh_new` as well as the question "is the ssh_session type a pointer type" in order to rectify the programmer's confusion. Furthermore, programmers did not necessarily respond to the wizard's most recent message; 16% of programmers' backward-facing dialogue acts were related older to messages in the conversation.

6.4 RQ4: Traceability

In total, we identified 157 specific API components across 2430 utterances. The wizards referred to specific API components more often than the programmers, and a slightly broader range of API components were identified in the Allegro scenario than in the libssh scenario. The total number of API component labels and number of unique component labels annotated are shown in Table 6.

7 Discussion

This section discusses several observations from our study and provides recommendations for future VA designers.

7.1 Reflections on the Wizard of Oz Experiment.

We support the use of the WoZ methodology to collect dialogues in which programmers genuinely interact with another human as they would with a VA. Most WoZ experiments do not include a "control" condition where users are aware that the system is actually operated by another human, as it is already well-established in the literature that humans interact with machines differently than they do with other humans. However, as VAs become increasingly intelligent and humans become more accustomed to interacting with them, these differences may begin to shrink, tempering the need for the WoZ deception.

The dialogues themselves were fruitful; over 90% of programmer's messages were directly related to some aspect of the API. The rest were either related to dialogue control (e.g. salutations, stalling, expressing gratitude) or off-topic turns to "test" the system's capabilities (e.g. "It was hot here today. How is the weather where you are?"). But as we suggested in Section 4.4, even these interactions are valuable to observe, as they indicate the types of irrelevant input users might give a VA in practice.

While the WoZ experimental methodology produced high-quality data, we observed several bottlenecks in the experimental design that slowed our actual data collection. First, designing each experimental scenario required us to create a set of tasks for the programmers to complete. This

TABLE 6
Number of references to specific API components in the corpus.

Library	Variable	Programmer	Wizard	Both
libssh	# Components	383	1053	1436
	# Distinct labels	41	74	76
Allegro	# Labels	343	922	1265
	# Distinct labels	47	80	81
Both	# Labels	726	1975	2701
	# Distinct labels	88	154	157

process was slow, as it involved cross-referencing API documentation and online resources and manually attempting each iteration of the task set. We found that official tutorials published by API authors often provided good frameworks for tasks; they were generally modularized, with each tutorial introducing new API components to solve a different problem. Developing a system to automatically or semi-automatically generate tasks from API tutorials may be an interesting direction for future researchers.

Second, the process of recruiting and coordinating participants was particularly time-consuming. We found that unexperienced programmers were more eager to inquire about the study than expert programmers, many of whom were uninterested in accepting a job with such a short duration. This required us to spend more time targeting potential participants and extending them offers. Once a programmer was interested, we then had to identify a time when both the programmer and a wizard would be able to participate, which was often several days in the future. Future researchers conducting WoZ experiments with expert programmers may want to investigate other avenues for recruiting and coordinating participants.

Third, the actual execution of the experiments was a necessary bottleneck, as each experimental session took 90 minutes to complete, with additional time for the programmer to set up the testing environment, read through the tasks, and complete an exit survey. We only ever conducted one session at a time, to ensure that the first author could be available to help with any problems that arose. Future researchers may want design their experiments in such a way as to allow sessions to be executed in parallel.

7.2 Potential Role for a VA in a Developer's Workflow.

Although there are certainly specific use cases in which a VA may serve as a developer's only means to access documentation, our results suggest that a general purpose VA for API usage should function alongside, rather than in lieu of, traditional documentation. By and large, the developers in this study expressed frustration that they were forbidden from accessing typical documentation. While some programmers acknowledged that they were just "used to" traditional documentation, they also cited several specific grievances, including the slow response speed, the inability to easily follow links to other components or obtain a high-level view of the API structure, and the fact that responses were sometimes irrelevant, incorrect, or inadequate.

Still, many programmers identified specific "intelligent" features that they admired in Apiza. In particular, many were impressed by Apiza's ability to provide context-specific recommendations and synthesize information when the documentation was incomplete. While the programmers were frustrated with *how* Apiza communicated information, they were impressed by its ability to determine *what* to communicate. As one programmer expressed, "the only times that I [the programmer] could see myself using it would be times that I usually try to ask a person."

Therefore, we propose that a VA for API usage should help users to navigate and understand, but not necessarily replace, traditional documentation. The system should be capable of engaging in dialogue to determine what a user wants, directing the user to relevant pieces of documentation, and answering questions about the API that may not be clear from the documentation alone.

7.3 Considerations for Specific VA Features.

We present four considerations regarding features that API VA designers may wish to incorporate into their system.

First, our study revealed that programmers interacting with a VA primarily ask questions about five types of API information: patterns (e.g. "what functions should I use to bring about a specific outcome?"), basic (e.g. "what is this function's return type?"), functionality (e.g. "what does this function do?"), structure ("what functions are related to this one?"), and examples (e.g. "show me an example invocation of this function."). While there may be a temptation to design a VA capable of answering any arbitrary API-related question, it may be more efficient to focus on training it to answer some subset of these question types.

In particular, while the wizards in our study performed several different API-related tasks, we found that the majority of the interactions revolved around identifying the appropriate API components to bring about specific outcomes. We noted five distinct phases in this process: the programmer explaining his/her requirements, the wizard verifying those requirements, the wizard providing one or more suggestions, the user asking follow-up questions about those suggestions, and the wizard answering those questions. The preponderance of this interaction pattern is due, at least in part, to the type of tasks in this study; had programmers been performing maintenance tasks, there may have been fewer questions about implementing new patterns and more about understanding existing code. Still, based on our studies, we believe that identifying useful API components would be a high-value application for a VA.

Another consideration relates to the potential use of online resources. In our studies, wizards occasionally directed programmers to websites related to the programmers' queries. We also frequently observed wizards struggle to determine how to help the programmers implement certain design patterns; in many of those cases, the wizards could have easily found the answer by searching the internet. A VA could take advantage of the huge amount of API information available on question-answering websites like StackOverflow, in projects hosted on sites like GitHub, and in tutorials around the web.

Finally, many of the programmers in our study shared a key complaint about the simulated virtual assistant: it was too slow. A real VA already has an advantage over a human wizard, in that it can rapidly search through large amounts of data; however, features that involve accessing online resources or running neural models may cause severe delays. VA designers will have to balance the desire to include sophisticated features against the need for the system to generate a speedy response.

7.4 Implementing a User Interface.

VA designers will have to make a number of important decisions when implementing a user interface. In particular, the input and output modalities of a VA for API usage are likely to have a strong impact on the overall

usability of the system. The programmers and wizards in our experiments communicated solely through a text-based interface, but there are several alternative forms of input (such as voice and mouse/touchscreen input) and output (such as computer speech and automatic navigation to relevant documentation/resources) that we did not explore. The text-based system was ideal for this study, as it did not restrict the types of questions or information that either party could share. However, it did limit the quantity of information a wizard could reasonably share in a single turn, and many users found it inconvenient compared to standard documentation navigation.

We speculate that a speech-based interface may be more convenient to users, to some extent, as they would not have to divert their attention from the task at hand to interact with it. However, it would also introduce a number of limitations. For starters, although state-of-the-art voice-recognition technology is quite advanced, an API usage scenario would present several unique challenges. The VA would need to recognize vocabulary and syntax specific to the programming domain (e.g. uncommon tokens like "int" or "struct" and explicit punctuation) and identifiers specific to a given API (e.g. it would need to identify function names consisting of multiple concatenated words). Furthermore, if the system produced speech as output, it would need to be limited to certain types of utterances; for instance, it would be impractical for the VA to audibly recite code examples or long passages of documentation.

Other possibilities include touchscreen or mouse-based interfaces. As discussed in Section 2.3, utterances can categorized into a finite number of dialogue act types; in some cases, a VA may be able to identify the most-likely options for a user's upcoming dialogue act (e.g. a request for more information about the function, or a request for similar functions) and allow the user to select one, rather than require the user to generate an utterance. Of course, there are many more cases where a user will need to provide the VA with new information, or specify an API component; in these cases, the user would need a less limited form of input.

We suggest that VA designers consider all potential UI options in light of their particular use case. The most user-friendly interface for a particular application may involve mixing different modalities; for instance, when a programmer asks the VA to suggest a function, it may be appropriate for the system to say the name of the function out loud and simultaneously navigate to the function's documentation. An example of a multimodal UI integrated with an IDE could allow a user to highlight a function invocation, and then verbally ask a question about that function's behavior.

7.5 Formulating a Reinforcement Learning Problem.

Following the framework laid out in Section 2.4, VA designers may use the data from this study to train the dialogue management module of a VA for API usage via reinforcement learning. Here, we discuss the next steps in this process.

1) *Choose the task(s).* VA designers must first decide on the particular task(s) they would like their system to complete. Recall that four components must be clearly defined in any reinforcement learning problem: the system's action set, the state space, the state transition function, and the reward function. Defining each of these components requires a concrete understanding of an agent's goals. The wizards in our study performed a number of distinct activities: they helped users identify relevant API components, they answered questions about those components, they fetched code examples from the source code and online resources, and more. Choosing the particular task(s) a VA should be able to complete will guide the rest of its development.

2) *Structure the knowledge base.* Depending on the particular task(s), a VA for API usage may require different types of knowledge from different sources: it may use a single API or multiple APIs; it may or may not access online resources; it may have access to well-structured documentation or poorly structured header comments in source files. No matter what knowledge base(s) the VA needs to access, it needs some structured way to access and query them in order to make decisions and provide information to the user.

3) *Identify the relevant system and user actions.* While we provide a first round of high-level annotations of annotations for the dialogues, their granularity will likely not suffice for many tasks. For instance, to design a VA that suggests relevant functions, it may be important to differentiate between actions in which the user elicits *all relevant functions* and those in which the user elicits *the most relevant function*; in our annotation scheme, both of these would simply be labeled with the "ELICIT-OFFER-OR-SUGGESTION" illocutionary DA type. To this end, VA designers may want to reannotate the dataset with task-specific labels, either following an open-coding process or using another predefined annotation scheme, to identify the system and user actions that are relevant to their particular task(s).

4) *Identify the state space.* Recall that the "state" refers to an RL agent's beliefs and view of its environment, and the "state space" is the set of all possible states. Choosing an appropriate, narrow state space facilitates the learning process. VA designers can use the labeled dialogues to identify which task- and dialogue-specific features actually influenced the wizards' dialogue strategies, in order to determine which ones to include in the state space.

Potential features to be included in the state space include dialogue length, how many questions the user has asked, whether the user has repeated the current question, the previous user and system actions, whether a specific API component has been identified, the number of components in the API that appear to be relevant to a user's query and how relevant those components appear to be, the number of web resources that appear to be relevant to a user's query and how relevant those appear to be, a belief about the user's ongoing programming activity (bugfixing, refactoring, implementing a new method, etc.), whether the user's request has been satisfied, and many more. It is up to the VA designer to consider all features that may be relevant to the particular task(s) at hand.

5) *Build a "user simulator" and design state transitions.* Once the action sets and state space are established, the next step is to create the model that represents the simulated environment and dictates how different actions lead to different states. In the context of reinforcement learning for dialogue strategy, a primary component of this model is referred to

as a "user simulator." The user simulator emulates how a user would react to the system's action at a given point in a dialogue. In many ways, the user simulator can be just as complex as the agent it is used to train; it has its own set of actions, goals, and constraints that can change over the course of a dialogue. It is common for the user simulator to use manually crafted heuristics to update its goals and constraints and to use supervised machine learning to choose an appropriate action. More rules are then needed to define how the agent's state changes in response to the simulated user's action.

Reiser and Lemon [12] demonstrate a fairly naive user simulator that keeps track of whether the agent has completed certain tasks and uses a simple bigram model (trained on labeled WoZ data) to generate user actions. Once again, a VA designer must consider the interactions that took place in the real WoZ dialogues and determine how sophisticated a user simulator needs to be for a particular application. For instance, a simple user simulator designed to train a VA to identify relevant API components may choose a target function and then ask questions and provide information related to that function. But the designer may also want the VA to account for scenarios in which the user asks for a component that does not exist, or scenarios in which the user initially wants a certain component, but is satisfied by a similar component. Any desired functionality in the VA must be reflected in the user simulator.

6) *Determine the reward function.* The reward function enables the RL agent to learn optimal strategies by assigning point values to specific actions and outcomes to encourage and discourage certain behaviors. For example, agents are usually penalized a small amount for each turn in a task-oriented dialogue to encourage them to complete the task as quickly as possible. They are also generally rewarded or penalized depending on whether they complete their task within a specified amount of time steps. However, in the context of task-oriented dialogues, there are other behaviors that a VA designer may wish to discourage; for instance, users may not like it if the VA shares too much information in a single dialogue turn, even though doing so might increase the likelihood of the agent completing its task. Therefore, to determine which actions and behaviors to include in the reward function and how to weigh the impact of each individual factor, API VA designers should consider how those factors correlated with user task completion and subjective user feedback in the WoZ dialogues.

Once these steps are completed, VA designers can determine an appropriate RL algorithm and implement it to train the VA's dialogue management system. From there, they will be able to incorporate natural language understanding and generation, and create a suitable user interface.

8 Conclusion

Virtual assistants for programmers have not been widely researched, despite recent advancements in VA technology and calls for more intelligent tools in software engineering. This is largely due to the lack of publicly-available datasets that can be used to understand which programming tasks would be high-value targets for VAs and to train task-specific dialogue systems.

In this paper we laid the groundwork for a VA for API usage. First, we presented the methodology and results of Wizard of Oz experiments designed to simulate interactions between a programmer and a virtual assistant for API usage. Then, we annotated the dialogue acts in the programmer-wizard interactions along four dimensions: illocutionary DA type, API DA type, backward-facing DA type, and traceability. Finally, we discussed the implications of our study on future VA development.

In doing so, we have made the following specific contributions to the field of software engineering:

1) A corpus of 30 Wizard-of-Oz dialogues, comprising 44 hours of programming activity and including two separate APIs.
2) The results of those programming sessions, including programmers' comments, ratings of the simulated virtual assistant, and performance on the task sets.
3) Corpus annotations along 4 dimensions.
4) Several recommendations and considerations for future VA designers.

We have made all data related to the experimental design, experimental results, and dialogue act annotations available via an online appendix:

https://github.com/ApizaCorpus/ApizaCorpus

Acknowledgments

The authors would like to sincerely thank the participants in the Wizard of Oz experiments, as well as the anonymous reviewers whose reccomendations have greatly improved the manuscript.

References

[1] R. W. White, "Skill discovery in virtual assistants," *Communications of the ACM*, vol. 61, no. 11, pp. 106–113, 2018.
[2] F. A. Brown, M. G. Lawrence, and V. O. Morrison, "Conversational virtual healthcare assistant," Nov. 21 2017, uS Patent 9,824,188.
[3] Z. Lv and X. Li, "Virtual reality assistant technology for learning primary geography," in *International Conference on Web-Based Learning*. Springer, 2015, pp. 31–40.
[4] H. He, D. Chen, A. Balakrishnan, and P. Liang, "Decoupling strategy and generation in negotiation dialogues," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018, pp. 2333–2343.
[5] S. Whittaker, M. A. Walker, and J. D. Moore, "Fish or fowl: A wizard of oz evaluation of dialogue strategies in the restaurant domain." in *LREC*, 2002.
[6] B. Schmidt, R. Borrison, A. Cohen, M. Dix, M. Gärtler, M. Hollender, B. Klöpper, S. Maczey, and S. Siddharthan, "Industrial virtual assistants: Challenges and opportunities," in *Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers*. ACM, 2018, pp. 794–801.
[7] B. Boehm, "A view of 20th and 21st century software engineering," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 12–29.
[8] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez *et al.*, "On-demand developer documentation," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 479–483.
[9] I. V. Serban, R. Lowe, P. Henderson, L. Charlin, and J. Pineau, "A survey of available corpora for building data-driven dialogue systems," *arXiv preprint arXiv:1512.05742*, 2015.

[10] A. Wood, P. Rodeghero, A. Armaly, and C. McMillan, "Detecting speech act types in developer question/answer conversations during bug repair," in *Proc. of the 26th ACM Symposium on the Foundations of Software Engineering*, 2018.

[11] J. Howison and J. D. Herbsleb, "Scientific software production: Incentives and collaboration," in *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, ser. CSCW '11. New York, NY, USA: ACM, 2011, pp. 513–522. [Online]. Available: http://doi.acm.org/10.1145/1958824.1958904

[12] V. Rieser and O. Lemon, *Reinforcement learning for adaptive dialogue systems: a data-driven methodology for dialogue management and natural language generation*. Springer Science & Business Media, 2011.

[13] N. Dahlbäck, A. Jönsson, and L. Ahrenberg, "Wizard of oz studies—why and how," *Knowledge-based systems*, vol. 6, no. 4, pp. 258–266, 1993.

[14] K. Bach and R. Harnish, "Linguistic communication and speech acts," 1979.

[15] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[16] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar apis: An exploratory study," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 266–276.

[17] S. Mealin and E. Murphy-Hill, "An exploratory study of blind software developers," in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2012, pp. 71–74.

[18] I. McCowan, J. Carletta, W. Kraaij, S. Ashby, S. Bourban, M. Flynn, M. Guillemot, T. Hain, J. Kadlec, V. Karaiskos *et al.*, "The ami meeting corpus," in *Proceedings of the 5th International Conference on Methods and Techniques in Behavioral Research*, vol. 88, 2005, p. 100.

[19] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013.

[20] T. Tenbrink, K. Eberhard, H. Shi, S. Kuebler, and M. Scheutz, "Annotation of negotiation processes in joint-action dialogues," *Dialogue & Discourse*, vol. 4, no. 2, pp. 185–214, 2013.

[21] J. Schatzmann, B. Thomson, K. Weilhammer, H. Ye, and S. Young, "Agenda-based user simulation for bootstrapping a pomdp dialogue system," in *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Companion Volume, Short Papers*. Association for Computational Linguistics, 2007, pp. 149–152.

[22] L. D. Riek, "Wizard of oz studies in hri: a systematic review and new reporting guidelines," *Journal of Human-Robot Interaction*, vol. 1, no. 1, pp. 119–136, 2012.

[23] A. Green, H. Huttenrauch, and K. S. Eklundh, "Applying the wizard-of-oz framework to cooperative service discovery and configuration," in *RO-MAN 2004. 13th IEEE International Workshop on Robot and Human Interactive Communication (IEEE Catalog No.04TH8759)*, Sep. 2004, pp. 575–580.

[24] S. Dow, B. MacIntyre, J. Lee, C. Oezbek, J. D. Bolter, and M. Gandy, "Wizard of oz support throughout an iterative design process," *IEEE Pervasive Computing*, vol. 4, no. 4, pp. 18–26, 2005.

[25] J. F. Kelley, "An iterative design methodology for user-friendly natural language office information applications," *ACM Transactions on Information Systems (TOIS)*, vol. 2, no. 1, pp. 26–41, 1984.

[26] N. Reithinger and M. Klesen, "Dialogue act classification using language models," in *Fifth European Conference on Speech Communication and Technology*, 1997.

[27] A. Popescu-Belis, "Dialogue acts: One or more dimensions," *ISSCO WorkingPaper*, vol. 62, 2005.

[28] M. G. Core and J. F. Allen, "Coding dialogs with the damsl annotation scheme," 1997.

[29] H. Bunt, V. Petukhova, D. Traum, and J. Alexandersson, *Dialogue Act Annotation with the ISO 24617-2 Standard*, 11 2017, pp. 109–135.

[30] H. Bunt, "The dit++ taxanomy for functional dialogue markup," *Journal of Philosophical Logic*, 01 2009.

[31] A. Clark and A. Popescu-Belis, "Multi-level dialogue act tags," in *Proceedings of the 5th SIGdial Workshop on Discourse and Dialogue at HLT-NAACL 2004*, 2004, pp. 163–170.

[32] J. Li, W. Monroe, A. Ritter, M. Galley, J. Gao, and D. Jurafsky, "Deep reinforcement learning for dialogue generation," *arXiv preprint arXiv:1606.01541*, 2016.

[33] M. Lewis, D. Yarats, Y. Dauphin, D. Parikh, and D. Batra, "Deal or no deal? end-to-end learning of negotiation dialogues," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017, pp. 2443–2453.

[34] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[35] S. P. Singh, M. J. Kearns, D. J. Litman, and M. A. Walker, "Reinforcement learning for spoken dialogue systems," in *Advances in Neural Information Processing Systems*, 2000, pp. 956–962.

[36] K. Scheffler and S. Young, "Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning," in *Proceedings of the second international conference on Human Language Technology Research*. Citeseer, 2002, pp. 12–19.

[37] B. Dhingra, L. Li, X. Li, J. Gao, Y.-N. Chen, F. Ahmed, and L. Deng, "Towards end-to-end reinforcement learning of dialogue agents for information access," *arXiv preprint arXiv:1609.00777*, 2016.

[38] B. Peng, X. Li, L. Li, J. Gao, A. Celikyilmaz, S. Lee, and K.-F. Wong, "Composite task-completion dialogue policy learning via hierarchical deep reinforcement learning," *arXiv preprint arXiv:1704.03084*, 2017.

[39] T. Zhao and M. Eskenazi, "Towards end-to-end learning for dialog state tracking and management using deep reinforcement learning," *arXiv preprint arXiv:1606.02560*, 2016.

[40] M. Meng, S. Steinhardt, and A. Schubert, "Application programming interface documentation: what do software developers want?" *Journal of Technical Writing and Communication*, vol. 48, no. 3, pp. 295–330, 2018.

[41] E. Aghajani, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on linguistic antipatterns affecting apis," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 25–35.

[42] A. Head, C. Sadowski, E. Murphy-Hill, and A. Knight, "When not to comment: questions and tradeoffs with api documentation for c++ projects," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 643–653.

[43] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 31, 2014.

[44] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? an empirical study on the directives of api documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2012.

[45] M. P. Robillard and R. Deline, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[46] Y. Tian, F. Thung, A. Sharma, and D. Lo, "Apibot: Question answering bot for api documentation," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 153–158.

[47] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2004, pp. 151–158.

[48] P. Pruski, S. Lohar, W. Goss, A. Rasin, and J. Cleland-Huang, "Tiqi: answering unstructured natural language trace queries," *Requirements Engineering*, vol. 20, no. 3, pp. 215–232, 2015.

[49] N. C. Bradley, T. Fritz, and R. Holmes, "Context-aware conversational developer assistants," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 993–1003.

[50] V. Arnaoudova, S. Haiduc, A. Marcus, and G. Antoniol, "The use of text retrieval and natural language processing in software engineering," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 949–950.

[51] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, July 1999.

[52] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 345–355. [Online]. Available: http://doi.acm.org/10.1145/336512.336586

[53] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721–734, Aug 2002.

[54] J. Wang, X. Peng, Z. Xing, and W. Zhao, "An exploratory study of feature location process: Distinct phases, recurring patterns, and

elementary actions," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2011, pp. 213–222.

[55] J. Wang, X. Peng, Z. Xing, and W. Zhao, "Improving feature location practice with multi-faceted interactive exploration," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 762–771.

[56] C. B. ULLER, A. FIEDLER, M. GABSDIL, H. HORACEK, M. PINKAL, J. SIEKMANN, D. TSOVALTZI, V. Bao Quoc, and M. WOLSKA, "A wizard-of-oz experiment for tutorial dialogues in mathematics."

[57] I. Kruijff-Korbayová, T. Becker, N. Blaylock, C. Gerstenberger, M. Kaisser, P. Poller, V. Rieser, and J. Schehl, "The sammie corpus of multimodal dialogues with an mp3 player." in *LREC*, 2006, pp. 2018–2023.

[58] M. Hajdinjak and F. Mihelič, "The paradise evaluation framework: Issues and findings," *Computational Linguistics*, vol. 32, no. 2, pp. 263–272, 2006.

[59] L. A. Pineda, A. Massé, I. Meza, M. Salas, E. Schwarz, E. Uraga, and L. Villaseñor, "The dime project," in *Mexican International Conference on Artificial Intelligence*. Springer, 2002, pp. 166–175.

[60] N. Bertomeu, H. Uszkoreit, A. Frank, H.-U. Krieger, and B. Jörg, "Contextual phenomena and thematic relations in database qa dialogues: results from a wizard-of-oz experiment," in *Proceedings of the Interactive Question Answering Workshop at HLT-NAACL 2006*, 2006, pp. 1–8.

[61] V. Rieser, I. Kruijff-Korbayová, and O. Lemon, "A corpus collection and annotation framework for learning multimodal clarification strategies," in *6th SIGdial Workshop on DISCOURSE and DIALOGUE*, 2005.

[62] C. Benzmüller, H. Horacek, H. Lesourd, I. Kruijff-Korbayová, M. Schiller, and M. Wolska, "A corpus of tutorial dialogs on theorem proving; the influence of the presentation of the study-material," in *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, 2006.

[63] L. E. Asri, H. Schulz, S. Sharma, J. Zumer, J. Harris, E. Fine, R. Mehrotra, and K. Suleman, "Frames: A corpus for adding memory to goal-oriented dialogue systems," *CoRR*, vol. abs/1704.00057, 2017. [Online]. Available: http://arxiv.org/abs/1704.00057

[64] Y. Kang, Y. Zhang, J. K. Kummerfeld, L. Tang, and J. Mars, "Data collection for dialogue system: A startup perspective," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*, vol. 3, 2018, pp. 33–40.

[65] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, Oct 2002.

[66] J. Carletta, S. Ashby, S. Bourban, M. Flynn, M. Guillemot, T. Hain, J. Kadlec, V. Karaiskos, W. Kraaij, M. Kronenthal *et al.*, *Guidelines for Dialogue Act and Addressee Annotation Version 1.0*, Oct 2005. [Online]. Available: http://groups.inf.ed.ac.uk/ami/corpus/Guidelines/dialogue_acts_manual_1.0.pdf

[67] W. Maalej and M. P. Robillard, *API Knowledge Coding Guide Version 7.2*. [Online]. Available: https://cado.informatik.uni-hamburg.de/coding-guide/

[68] (2018) libssh 0.7.3 documentation. [Online]. Available: https://api.libssh.org/stable/index.html

[69] (2018) Allegro 5 reference manual. [Online]. Available: https://liballeg.org/a5docs/5.2.4/

[70] R. Gangadharaiah, B. Narayanaswamy, and C. Elkan, "What we need to learn if we want to do and not just talk," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*, vol. 3, 2018, pp. 25–32.

[71] V. Stoyanov and C. Cardie, "Topic identification for fine-grained opinion analysis," in *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*. Association for Computational Linguistics, 2008, pp. 817–824.

[72] J. Wiebe and E. Riloff, "Creating subjective and objective sentence classifiers from unannotated texts," in *International conference on intelligent text processing and computational linguistics*. Springer, 2005, pp. 486–497.

[73] R. Witte, Q. Li, Y. Zhang, and J. Rilling, "Text mining and software engineering: an integrated source code and document analysis approach," *IET software*, vol. 2, no. 1, pp. 3–16, 2008.

[74] K. Krippendorff, "Agreement and information in the reliability of coding," *Communication Methods and Measures*, vol. 5, no. 2, pp. 93–112, 2011.

[75] H.-F. Hsieh and S. E. Shannon, "Three approaches to qualitative content analysis," *Qualitative health research*, vol. 15, no. 9, pp. 1277–1288, 2005.

[76] B. Downe-Wamboldt, "Content analysis: method, applications, and issues," *Health care for women international*, vol. 13, no. 3, pp. 313–321, 1992.

[77] K. Krippendorff, "Reliability in content analysis," *Human communication research*, vol. 30, no. 3, pp. 411–433, 2004.

[78] R. Craggs and M. M. Wood, "Evaluating discourse and dialogue coding schemes," *Computational Linguistics*, vol. 31, no. 3, pp. 289–296, 2005.

[79] M. Bengtsson, "How to plan and perform a qualitative study using content analysis," *NursingPlus Open*, vol. 2, pp. 8 – 14, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352900816000029

[80] W. Spooren and L. Degand, "Coding coherence relations: Reliability and validity," *Corpus linguistics and linguistic theory*, vol. 6, no. 2, pp. 241–266, 2010.

[81] A. Stolcke, K. Ries, N. Coccaro, E. Shriberg, R. Bates, D. Jurafsky, P. Taylor, R. Martin, C. V. Ess-Dykema, and M. Meteer, "Dialogue act modeling for automatic tagging and recognition of conversational speech," *Computational Linguistics*, vol. 26, no. 3, pp. 339–373, 2000. [Online]. Available: https://doi.org/10.1162/089120100561737