Guardian: A Crowd-Powered Spoken Dialog System for Web APIs

Ting-Hao (Kenneth) Huang
Carnegie Mellon University
Pittsburgh, PA USA
tinghaoh@cs.cmu.edu

Walter S. Lasecki
University of Michigan
Ann Arbor, MI USA
wlasecki@umich.edu

Jeffrey P. Bigham
Carnegie Mellon University
Pittsburgh, PA USA
jbigham@cs.cmu.edu

Abstract

Natural language dialog is an important and intuitive way for people to access information and services. However, current dialog systems are limited in scope, brittle to the richness of natural language, and expensive to produce. This paper introduces *Guardian*, a crowd-powered framework that wraps existing Web APIs into immediately usable spoken dialog systems. Guardian takes as input the Web API and desired task, and the crowd determines the parameters necessary to complete it, how to ask for them, and interprets the responses from the API. The system is structured so that, over time, it can learn to take over for the crowd. This hybrid systems approach will help make dialog systems both more general and more robust going forward.

Introduction

Conversational interaction allows users to access computer systems and satisfy their information needs in an intuitive and fluid manner, especially in mobile environments. Recently, spoken dialog systems (SDSs) have made great strides in achieving that goal. It is now possible to speak to computers on the phone via conversational assistants on mobile devices, *e.g. Siri*, and, increasingly, from wearable devices on which non-speech interaction is limited. However, despite decades of research, existing spoken dialog systems are limited in scope, brittle to the complexity of language, and expensive to produce. While systems such as Apple's Siri integrate a core set of functionality for a specific device (e.g. basic phone functions), they are limited to a pre-defined set of interactions and do not scale to the huge number of applications available on today's smartphones, or web services available on the Internet.

Despite frameworks which have been proposed to reduce the engineering efforts of developing a dialog system (Bohus et al. 2007), constructing spoken language interfaces is still well-known as a costly endeavor. Moreover, this process must be repeated for each application since general-purpose conversational support is beyond the scope of existing dialog system approaches. Therefore, to tackle these challenges, we introduces *Guardian*, a framework that uses *Web APIs* (Application Programming Interfaces) combined with *crowdsourcing* to efficiently and cost-effectively enlarge the scope of existing spoken dialog systems. Furthermore, Guardian is structured so that, over time, an automated dialog system could be learned from the chat logs collected by our dialog system and gradually take over from the crowd.

Web-accessible APIs can be viewed as a gateway to the rich information stored on the Internet. The Web contains tens of thousands of APIs (many of which are free) that support access to myriad resources and services. As of April 2015, ProgrammableWeb[1] alone contains the description of more than 13,000 APIs in categories including travel (1,073), reference (1,342), news (1,277), weather (368), health (361), food (356), and many more. These Web APIs can encompass the common functions of popular existing SDSs, such as Siri, which is often used to send text messages, access weather reports, get directions, and find nearby restaurants. Therefore, if SDSs are able to exploit the rich information provided by the thousands of available APIs on the web, their scope would be significantly enlarged.

However, automatically incorporating Web APIs into an SDS is a non-trivial task. To be useful in an application like Siri, these APIs need to be manually wrapped into conversational templates. However, these templates are brittle because they only address a small subset of the many ways to ask for a particular piece of information. Even a topic as seemingly straightforward as weather can be tricky. For example, Siri has no trouble with the query "What is the weather in New Orleans?", but cannot handle "Will it be hot this weekend in the Big Easy?" The reason is that the seemingly simple latter question requires three steps: recognizing that hot refers to temperature, temporally resolving weekend, and recognizing "the Big Easy" as slang for "New Orleans." These are all difficult problems to solve automatically, but people can complete each fairly easily, thus Guardian uses crowdsourcing to disambiguate complex language. Though crowd-powered dialog systems suffer the drawback not being as fast as fully automated systems, we are optimistic that they can be developed and deployed much more quickly for new applications. While they might incur more cost on a per-interaction basis, they would avoid the huge overhead of an engineering team, and enable quickly prototyping dialog systems for new kinds of interactions.

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1] http://www.programmableweb.com

To this end, we propose a crowd-powered Web-API-based Spoken Dialog System called Guardian (of the Dialog). Guardian leverages the wealth of information in Web APIs to enlarge its scope. The crowd is employed to bridge the SDS with the Web APIs (**offline phase**), and a user with the SDS (**online phase**).

In the offline phase of Guardian , the main goal is to connect the useful parameters in the Web APIs with actual natural language questions which are used to understand the user's query. As there are certain parameters in each Web API which are more useful than others when performing an effective query on the API, it is crucial that we know which questions to ask the user to acquire the important parameters. There are three main steps in the offline phase, where the first two can be run concurrently. First, crowd-powered *QA pair collection* generates a set of questions (which includes follow-up questions) that will be useful in satisfying the information need of the user. Second, crowd-powered *parameter filtering* filters out "bad" parameters in the Web APIs, thus shrinking the number of candidate useful parameters for each Web API. Finally, crowd-powered *QA-parameter matching* not only matches each question with a parameter of the Web API, but also creates a ranking of which questions are more important is also acquired. This ranking enables Guardian to ask the more important questions first to faster satisfy the user's information need.

In the online phase of Guardian, the crowd is in charge of *Dialog Management*, *Parameter Filling*, and *Response Generation*. Dialog management focuses on deciding which questions to ask the user, and when to trigger the API given the current status of the dialog. The task of parameter filling is to associate the information acquired from the user's answers with the parameters in the API. For response generation, the crowd translates the results returned by the API (which is usually in JSON format) into a natural language sentence readable by the user.

To demonstrate the effectiveness of Web-API-based crowd-powered dialog systems, the Guardian system currently has 8 Web APIs incorporated, which cover topics including weather, movies, food, news, and flight information. We first show that our proposed method is effective in associating questions with important Web API parameters (QA-parameter matching). Then, we present real-world dialog experiments on 3 of the 8 Web APIs, and show that Guardian as able to achieve a task completion rate of 97%.

The contributions of this paper are two-fold:

- We propose a Web-API based, crowd-powered spoken dialog system which can significantly increase the coverage of dialog systems in a cost-effective manner, and also collect valuable training data to improve automatic dialog systems.

- We propose an effective workflow to combine expert and non-expert workers to translate Web APIs into a usable dialog system format. Our method has the potential to scale to thousands of APIs.

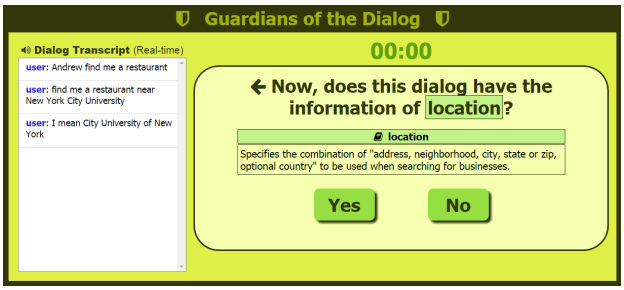

Figure 1: The user interface for crowd workers in Guardian. The left-hand side is a chat box that displays the running dialog. The right-hand side is the working panel displaying decision-making questions.

Background and Related Work

Our approach to generating dialog systems using the crowd builds on prior work on dialog system design, as well as interactive and offline human computation via crowdsourcing.

Dialog Systems

There is a considerable body of research on goal-oriented spoken dialog systems ranging in domain from travel planning (Xu and Rudnicky 2000) to tutoring students (Litman and Silliman 2004). Systems vary in their approach to dialog from simple slot-filling (Bobrow et al. 1977), to complex plan-based dialog management architectures (Ferguson, Allen, and others 1998; Horvitz and Paek 1999). A common strategy for simulating and prototyping is Wizard-of-Oz (WoZ) control (Maulsby, Greenberg, and Mander 1993). Crowd-powered dialog systems can be viewed as a natural extension of WoZ prototypes with several important characteristics. First, they have the potential to be deployed quickly, with easily-recruited workers powering the system as it learns to automate itself. Second, different groups of workers control different aspects of the system, resulting in an "assembly-line" of dialog system controllers, each of which can specialize in one specific aspect – for example, mapping the user's utterances into changes in dialog state, or guiding the dialog policy. This division of roles could enable more complex systems than those controlled by a single "wizard," and offers a path toward automation as computation takes over for controller as it is able to do so.

Prior work has also considered how the interfaces of Web applications implicitly define APIs (Hartmann et al. 2007), and how they can be used to create APIs for resources that do not otherwise expose one (Bigham et al. 2008).

Crowdsourced Question Answering

Our approach leverages prior work in human computation to bootstrap the creation of automated dialog systems. Human computation (Von Ahn 2009) has been shown to be useful in many areas, including writing and editing (Bernstein et al. 2010), image description and interpretation (Bigham et al. 2010; Von Ahn and Dabbish 2004a), and protein folding (Cooper et al. 2010).

The crowd can be quick. VizWiz was one of the first systems to elicit nearly real-time responses from the crowd by using a queueing model (Bigham et al. 2010). Other systems have shown that asking workers to wait while others join (Von Ahn and Dabbish 2004a; Chilton 2009) can be effectively used to have workers ready at a moment's notice (within a second or two). Combining this with continuous interaction has resulted in total per-response latencies under 5 seconds (Lasecki et al. 2012).

Chorus enables a real-time conversation with a crowd of workers as if they were a single conversational partner (Lasecki et al. 2013). Constructive, on-topic responses are elicited from workers using an incentive mechanism that encourages workers to balance response speed with accuracy to the best of their ability (Singh et al. 2012).

Framework of the Guardian System

The workflow we introduced consists of two phases: an "offline" phase and an "online" phase. The offline phase is a preparation process prior to the online phase. During the offline phase, necessary parameters are selected and questions are collected that will be used to query for those parameters during the online phase. The online phase is run in real-time through an interactive dialog. For each API, the offline phase only needs to be run once.

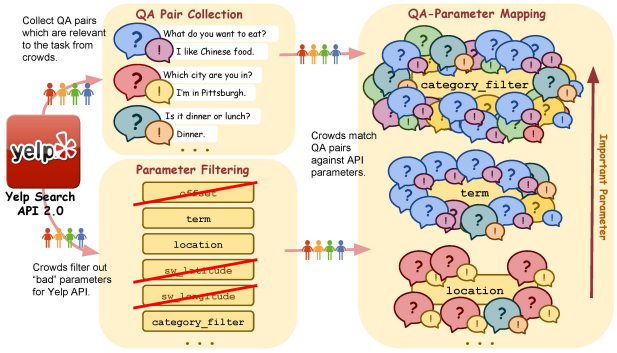

Figure 2: Offline Phase: A 3-stage Parameter Voting Workflow. Untrained crowd workers collect question and answer (QA) pairs related to the task, filter out unnatural parameters, and match each QA pair with the most relevant parameter.

Offline Phase: Translate a Web API to a Dialog System with the Crowd

As a preparation of the Guardian system, we propose a process powered by a non-expert crowd to select proper parameters that fit in the usages of dialog systems. As a byproduct, this process also generates a set of questions associated with parameters that can be used in the Guardian dialog management component as default follow-up questions.

The goal of this process is to significantly lower the threshold for programmers to contribute to our system, and thus make adding thousands of web APIs into the Guardian system possible. As shown in Figure 2, our process consists of 3 steps: First, given an API with a task, we collect various question and answer pairs related to the task. Second, to shrink the size of the parameters, we perform a filtering to prune out any "unnatural" parameters. Finally, we design a voting-like process where unskilled workers vote for the "best" parameters for each question.

Note that whether a parameter is optional or required is separate from their "applicability". For instance, in the Yelp API you need to specify the location by using one of the following three parameters: (1) city name, (2) latitude and longitude, or (3) geographical bounding box. The three parameters are "required parameters"; however, only the (1), city name, is likely to be mentioned in a natural dialog. We focus only on developing the workflow to enable unskilled crowd workers to rate the "applicability" of parameters. The "optional/required" status of the parameters is best realized when implementing the API wrapper.

Question-Answer (QA) Pair Collection The first stage is to collect various questions associated with the task. We ask crowd workers the following question: "A friend wants to [task description] and is calling you for help. Please enter the questions you would ask them back to help accomplish their task." We also ask the workers for the first, second, and third questions they would ask the other person, along with possible answers their conversational partner may reply with. This process is iteratively developed based on our experiments. We collect more multiple questions to increase the diversity of collected data. In our preliminary study, we found that for some tasks like finding food, the very first questions among different workers are quite similar (i.e., "What kind of food would you like?"). Moreover, instead of collecting only questions, we also collect corresponding answers, because question-answer pairs provide more clues to pick the best parameters in the next stage.

Parameter Filtering In the second stage, we perform a filtering process with an unskilled crowd to shrink the size of candidate parameters. Scalability is a practical challenge that often occurs when trying to apply general voting mechanisms to parameters of an API. For any API with N parameter and M QA pairs, there will be a total of $N * M$ decisions to make. For some more complicated APIs with large numbers of parameters, the cost would be considerable. Our solution is to adopt a filtering step before the actual voting stage. Based on the idea that humans are good at identifying outliers at a glance, we propose a method that simply shows all the parameters (with the names, types, and descriptions of the parameters) on the same web page to the workers, and ask them to select all the "unnatural" items that are unlikely to be mentioned in real-world conversations, or are obviously designed for computers and programmers.

QA-Parameter Matching In the third stage, we match the QA pairs collected from Stage 1 against the remaining parameters from Stage 2. We display one QA pair along with all the parameters at once, and ask crowd workers the following question: "In this conversation, which of the following piece of information is provided in the answer? The followings are parameters that used in a computer system. The descriptions could be confusing, or even none of them really fit. Please try your best to choose the best one." For

each represented QA pair, the workers are first required to pick one best parameter, and then rate their confidence level (low=1, medium=2, and high=3). This mechanism is developed empirically, and our experiments will demonstrate that this process could not only pick a good set of parameters for the dialog system application, but also pick good questions associated with each selected parameter. The workers' interface is shown in Figure 3.

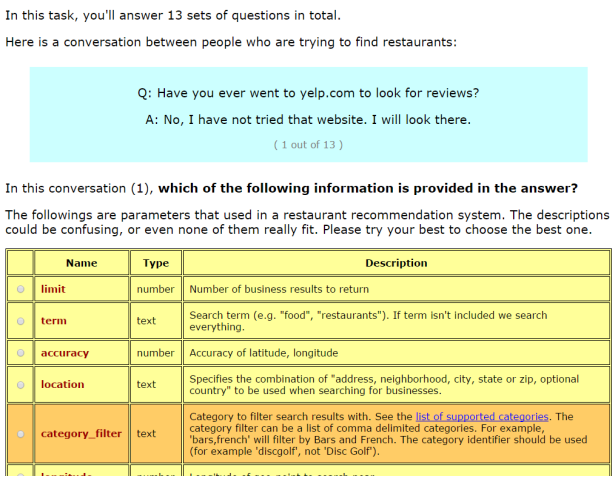

Figure 3: The interface for crowd workers to match of parameters to natural language questions.

Online Phase: Crowd-powered Spoken Dialog System for Web APIs

To utilize human computation to power a spoken dialog system, we address two main challenges: rapid information collection and response generation in real-time. Conceptually, a task-oriented dialog system performs a task by first acquiring the information of preference, requirements, and constrains from the user, and then applies the information to accomplish the task. Finally, the system reports the results back to the user in spoken language. Our system architecture is largely inspired by the solutions modern dialog systems use to simulate the process of human dialog which has been proven reasonably robust and fast on handling dialogs. To apply prior solutions which are developed originally with the assumption that the response time of each component is extremely short requires pushing the limits of crowd workers' speed to make the solution feasible. In Guardian, we apply ESP-game-like parameter filling, crowd-powered dialog management, and template-based response generation to tackle these challenges. The whole process is shown in Figure 4.

Parameter Filling via Output Agreement To encourage quality and speed of parameter extraction in Guardian, we designed a multi-player output agreement process to extract parameters from a running conversation. First, using a standard output agreement setup (von Ahn and Dabbish 2004b), crowd workers propose their own answers of the parameter value without communicating with each other. Guardian

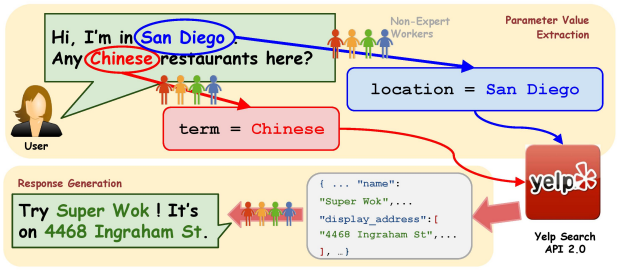

Figure 4: On-line Phase: crowd workers extract the required parameters and turn resulting JSON into responses.

automatically matches workers' answers to ensure the quality of extracted parameter value. To prevent the system from idling in the case that no answers match one another, a hard time constraint is also set. The system selects the first answer from workers when the the time constraint is reached.

Crowd-powered Dialog Management Second, we use the idea of dialog management to control the dialog status. Dialog management simulates a dialog as a process of collecting a set of information – namely, parameters in the context of web APIs. Based on which parameters are given, the current dialog state can be further decided (Figure 5). For most states, the dialog system's actions are pre-defined and can be executed automatically. Crowd workers are able to vote to decide the best action within a short amount of time. For example, in the dialog state where the query term ("term") is known but the location is unknown, a follow-up question (e.g., "Where are you?") can be pre-defined. Furthermore, the dialog management also controls when to call the web API. For instance, in Figure 5, if only one parameter is filled, the system would not reach to the state which is able to trigger the API.

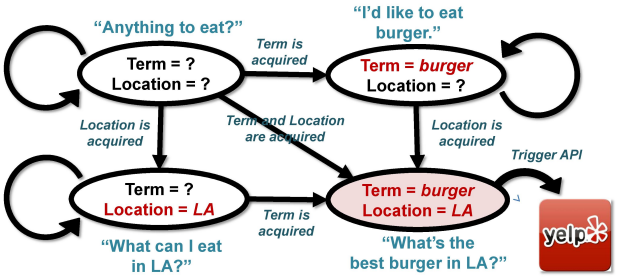

Figure 5: The State Diagram of dialog Management. In the context of crowd-powered systems, introducing a dialog manager reduces the time it takes the crowd to generate a response because most actions can be pre-defined and generated according to the dialog state.

Template-based Response Generation Finally, when we get the query results from the web API, the response object is usually in JSON format. To shorten the response time, we propose to use a prepared template to convert a given JSON file into a response to the user. In Guardian, we aim

	Web API	Task	#Parameter Filtered/Origin	Top Parameter	Top Question of the Top Parameter
1	Cat Fact	Search for random cat facts	1 / 1	number	tell my specificity what you what to know ?
2	Eventful	Look for events	14 / 16	include	Is it local?
3	Flight Status	Check flight status	8 / 9	flight	What is your exact flight number?
4	RottenTomatoes	Find information of movies	3 / 3	q	Okay no problem, is that all?
5	Weather Underground	Find the current weather	1 / 5	query	Time?
6	Wikipedia	Search for Wikipedia pages	7 / 15	action	Do you have any topic in mind?
7	Yahoo BOSS News Search	Search for news	5 / 6	sites	What information [sic] you want?
8	Yelp Search API	Find restaurants	10 / 13	location	Where?

Table 1: Selected Web APIs for parameter voting experiments. All of the 8 web APIs are used in the parameter voting experiments (Experiment 1).

to develop a system that gradually increases the capability to be automated. Therefore, instead of creating a separate data annotation step, we visualize the JSON object which contains the query results as an interactive web page, displays it to the crowd in real-time, and asks the crowd to answer the user's question based on the information in the JSON file. The JSON visualization interface implemented with JSON Visualizer[2] is shown in Figure 6. When doing this, Guardian records two types of data: The answer produced by the crowd, and the mouse clicks workers make when exploring the JSON object visualization. By combining these two types of data, we are able to identify the important fields in the response JSON object that have frequently been clicked, and also create natural-language templates mentioning these fields.

Note that in Guardian we focus on developing a task-oriented dialog system, and assuming all the input utterance are in-domain queries.

Figure 6: Interactive web UI to present the JSON data to non-expert crowd workers. With this user-friendly interface, unskilled workers can explore and understand the information generated by the APIs.

[2] http://visualizer.json2html.com/

Retainer Model and Time Constraints To support real-time applications with Guardian, we apply a retainer model and enforce time constraints on most actions in the system. The retainer model maintains a pool of waiting workers, and then signals them when tasks arrive. Prior work has shown that the retainer model is able to recall 75% of workers within 3 seconds (Bernstein et al. 2011). Furthermore, for most actions that workers can perform in the Guardian system, time constraints are enforced. For instance, in the ESP-game-like parameter filling stage, we set 30-second time constraints for all workers. If a worker fails to submit an answer within 30 seconds more than 5 times, the worker will be logged out of the system.

Experiment 1: Translate Web API to Dialog Systems with the Crowd

To examine the effectiveness of our proposed parameter ranking workflow, we explore the ProgrammableWeb website and select 8 popular web APIs for our experiment. To focus on real-world human conversation, we select only the text-based service rather than image or multimedia services, and also avoid heavy weight APIs like social network APIs or map APIs. We also define a task that is supported by the API. The full list of the selected APIs is shown in Table 1. Based on the task, we perform our Parameter Ranking process mentioned above on all possible parameters of the API. The Question-Answer Collection and Parameter Filtering stages are performed on the CrowdFlower (CF) platform. The Question-Parameter Matching is performed on Amazon Mechanical Turk (MTurk) with our own implemented user interface. The detailed experimental setting is as follows: First, the question-answer collection experiment was run on the CF platform. In our experiments, we use the following scenario: a friend of the worker's wants to know some information but is not able to use the Internet, so the friend has called them for help. We ask workers to input up to three questions that they would ask this friend to clarify what information is needed. We also ask workers to provide the pos-

	MAP				MRR			
	Parameter Voting	Not Unnatural	Ask Siri	Ask a Friend	**Parameter Voting**	Not Unnatural	Ask Siri	Ask a Friend
Cat Fact	1	1	1	1	1	1	1	1
Eventful	0.626	0.401	0.456	0.408	0.500	0.500	0.250	0.500
Flight Status	0.864	1	0.889	0.528	1	1	1	0.333
RottenTomatoes	1	0.333	0.333	0.333	1	0.333	0.333	0.333
Weather Underground	1	1	0.333	0.200	1	1	0.333	0.200
Wikipedia	0.756	0.810	0.250	0.331	1	1	0.250	0.333
Yahoo BOSS News Search API	0.756	0.917	0.867	0.917	1	1	1	1
Yelp Search API	0.867	0.458	0.500	0.578	1	0.333	0.500	1
Average	**0.858**	0.740	0.578	0.537	**0.938**	0.771	0.583	0.588

Table 2: Evaluation of Parameter Ranking. Both the MAP and MRR indicates that our approach is a better way to rank the parameters.

sible answers this friend may reply with. For each task listed in Table 1, we post 20 jobs on CF and collect 60 question-answer pairs from 20 different workers. Second, the experiment of parameter filtering is also conducted on CF. As mentioned in the previous section, for each parameter, we ask 10 workers to judge if this parameter is "unnatural". We filter out the parameters that at least 70% of workers judge as "unnatural". The remaining parameters after filtering are shown in Table 1. Finally, for each task, we take all collected QA pairs and asked 10 unique workers to select the most relevant parameters with a confidence score. We then summed up all of the confidence scores (1, 2, or 3) that each parameter received as the rating score. In total, 77 unique workers participated in the QA collection experiments. 23 unique workers participated in the parameter filtering experiments, and 26 unique workers participated in the QA-parameter matching experiments.

Our parameter rating process essentially performs a ranking task on all parameters. Therefore, we measure our proposed approach by utilizing two common evaluation metrics in the field of information retrieval, i.e., the mean average precision (MAP) and mean reciprocal rank (MRR). In our evaluation, each API is treated as a query, and the parameters are ranked by the rating score produced by our QA-parameter matching process. Similar to the process of annotating the relevant documents in the field of information retrieval, we hire a domain expert to annotate all the parameters that are appropriate for a dialog system as our gold-standard labels.

We implemented three baselines and asked crowd workers to rate parameters based on 3 different instructions. We first explained the overview of dialog systems and our project goal to workers, and then showed the following instructions, respectively:

- **Ask Siri**: Imagine you are using Siri. Please rate how likely you are to include a value for this parameter in your question?
- **Ask a Friend**: Imagine that you were not able to use the Internet and call a friend for help. How likely are you say include this information when asking your friend?
- **Not Unnatural**: This baseline directly takes the results from the "parameter filtering" stage, and calculates the percentage of workers who rate the parameter as "not unnatural".

10 unique workers were recruited on CrowdFlower to rate each parameter on a 5-star rating scale. Parameters were ranked using their average scores. The detailed evaluation results are shown in Table 2. Our QA-parameter matching approach largely outperforms all three baselines. Furthermore, both the high score of MAP and MRR strongly suggest that the unskilled crowd is able to produce a ranking list of API parameters that are very similar to that of domain expert's.

Note that we do not consider Siri a directly comparable system to Guardian. With the help of the crowd, Guardian acts quite differently from Siri, and is capable of working with the user to refine their initial query through a multi-turn dialog, while Siri focuses only on single-turn queries. Guardian works reasonably well in arbitrary domains (APIs) without using knowledge bases or training data, and can also handle the out-of-domain tasks that Siri cannot handle. More importantly, for any arbitrary web APIs, Guardian can collect conversational data annotated with filled parameters to generate response templates for automated dialog systems like Siri.

Experiment 2: Real-time Crowd-Powered Dialog System

Based on the results of Experiment 1, we implement and evaluate Guardian on top of 3 web APIs: the Yelp Search

Web API	Parameter Name	Parameter Desc	Avg. Parameter Filled Time(s) (mean / stdev)		JSON Filled Time(s) (mean / stdev)		#Turn	JSON Valid Rate	TCR	Domain Referenced TCR
Yelp Search API	term, location	query term, location	48.35	21.69	61.70	27.41	2.80	0.90	1.00	0.96 (Tomko 2005)
Rotten Tomatoes	q	query term	23.70	30.18	24.90	30.45	1.80	0.60	1.00	0.88 (Tomko 2005)
Weather Underground	query	zip code of location	69.50	136.04	70.60	135.99	2.60	0.90	0.90	0.94 (Lee et al. 2009)

Table 3: Dialog System Evaluation Result. This results demonstrates the average performance of Guardian out of 10 trials. **Parameter filled time** indicates the average time the system spends to fill one given parameter since the end of the user's first dialog turn. **JSON filled time** indicates the time Guardian spends to acquire the JSON string from the web API since the end of the user's first utterance input. **Number of turns** reflects the number of times that the user talks to the system to complete a task. **JSON valid rate** indicates the percentage of times that the JSON string returned by the API call contains useful information to complete the task. **Task completion rate (TCR)** indicates percent completion of the task. For reference, TCRs of the automated systems reported in literature are also listed (but note that the numbers are not directly comparable).

API 2.0[3] for finding restaurants, the Rotten Tomatoes API for finding movies[4], and the Weather Underground API[5] for obtaining weather reports.

Implementation

Guardian was implemented as a spoken dialog system that takes speech input and generates text chats as responses. The input speech was firstly transcribed by using Google Chrome's implementation of the Web Speech API in HTML5. The speech transcript was then displayed in real-time on both user's and crowd workers' interfaces.

All the functionalities mentioned in this paper were implemented. We utilized a game-like task design and interfaces (as shown in Figure 1) to incorporate all the features. From the perspective of a worker, the workflow are as follows: Once a worker accepts the task, the dialog management system first asks the worker the existences of one or more particular parameters. If the worker determines a parameter occurs in the current conversation, the system will further ask the worker to provide the value of this parameter. Behind the scene, Guardian adopts an ESP-game-like mechanism to find the matched answer among all workers, and uses the matched answers as parameter values. As shown in Figure 5, the dialog management system keeps track on current dialog state based on parameter status, and automatically ask the user corresponding questions.

Once all the required parameters are filled, Guardian will attempt to call the Web API with the filled parameters. If an JSON object is successfully returned by the Web API, the worker will then be shown with an interactive visualization of the JSON object (Figure 6) so that the results can be used by the worker to answer the user's questions.

Guardian uses a voting system to achieve consents among all workers. If a worker proposes a response, this request will be immediately sent to all other active workers of the same task. Only the responses that most workers agree with will be shown to the end user.

Currently, Guardian is fully running on Amazon Mechanical Turk. 10 workers were recruited to hold each conversation together.

Experimental Result

To test Guardian, we follow an evaluation method similar to the one used to evaluate Chorus (Lasecki et al. 2013): using scripted end-user questions and tasks. We first generated a script and task for each API before the experiments, which researchers followed as closely as possible during trials, while still allowing the conversation to flow naturally. The tasks and scripts for each API are as follows:

- **Yelp Search API**: Search for Chinese restaurants in Pittsburgh. Ask names, phone numbers, and the addresses of the restaurants.
- **Rotten Tomatoes API**: Look for the year of the movie "Titanic" and also ask for the rating of this movie.
- **Weather Underground API**: Look for current weather, and only use zip code to specify the location. Ask for the temperature and if it is raining now.

For each condition, we conducted 10 trials in a lab setting. We manually examined the effectiveness of the information in the resulting JSON object and the response created by the crowd. We defined *task completion* as either the obtained JSON string containing information that answers users' questions correctly, or crowd workers respond to the user with effective information despite of the status of the web API. The performance of Guardian is shown in Table 3.

In terms of the task completion rate (TCR), Guardian performed well on all three APIs with an average TCR of 0.97. The crowd workers were able to fill the parameters for the web APIs and generate responses based on the API query results. The TCR reported by the automated SDS of the same domain was also listed for comparison. Note that the TCR and SDS values were not directly compatible.

[3] `http://www.yelp.com/developers/documentation/v2/search_api`

[4] `http://developer.rottentomatoes.com/`

[5] `http://www.wunderground.com/weather/api/`

Case Study

In this section, we demonstrate some example chats in the experiments to show the characteristics of our system.

Parameter Extraction In our experiments, the crowd demonstrated the ability to extract parameters with a multi-player ESP-game-like setting. For instance, in the following chat, the crowd identified the query term (q) as "Titanic" right after the first line. With the correct parameter value, the RottenTomatoes API then correctly returned useful information to assist the crowd.

USER: hello I like to know some information about the movie Titanic
[PARAMETER EXTRACTED]: q = "TITANIC"
USER: the movie
USER: Titanic
GUARDIAN: < *URL of IMDB* >
USER: < *ASR error* > is the movie
GUARDIAN: < *URL of Rotten Tomatoes* >
USER: I like to know the year of the movie
USER: and the rating of this movie
GUARDIAN: 1997
GUARDIAN: 7.7

Dialog Management In the experiment, our dialog management system is capable of asking questions that require missing information. For example, in the following chat, the system asks a question for acquiring "term" from the user:

USER: < *ASR error* > can I find some food
[PARAMETER STATUS]: term = null, location = null
GUARDIAN [AUTOMATED RESPONSE]: What do you want to eat?

In the following example, the crowd first agreed on the query term (Chinese), but still needs to determine the location. Therefore, the system asks the follow-up question for location.

USER: < *ASR error* > can I get Chinese restaurant in Pittsburgh
USER: please tell me the phone number
[PARAMETER STATUS]: term = Chinese, location = (pending)
GUARDIAN [AUTOMATED RESPONSE]: Where are you?
USER: I am in Pittsburgh

The Crowd Recovers Invalid JSON In Guardian, the crowd has two ways to complete a task. First, workers can fill in API parameters and choose a response from the JSON that is returned. Second, workers can propose responses through a propose-and-vote mechanism. As a result, the API does not need to return a valid response for Guardian to respond correctly. In our experiments, most tasks were completed using the API response. The crowd generated their own messages when the API returned an error message within the JSON response, or the crowd found that the returned information was incorrect. In other words, the crowd in our system is able to recover from the errors that occurred in previous stages. Therefore, the TCR in Table 3 is higher than JSON valid rate.

The following are partial chats where the crowd overcame the null API results. In this example, all parameter values provided by the crowd were unmatched, so the API was not triggered at all. On one hand, despite of the absence of the API, the crowd was still able to hold a conversation with the user and complete the task. On the other hand, compared to the average number of turns as shown in Table 3, the crowd used more conversational turns to complete this task. Moreover, when the API's result was absent, some crowd workers could be confused and provided noisy responses, e.g., asking the user to look outside.

USER: is it raining now
USER: < *ASR error* >
GUARDIAN: location please
USER: sorry I only know the zip code
USER: < *ASR error* > 15232 < *ASR error* >
GUARDIAN: Where, which zip code?
USER: my location is < *ASR error* > zip code 15232
GUARDIAN: What is the weather in your location?
USER: sorry I only know the zip code
USER: the zip code here is
GUARDIAN: hello user, Pittsburg PA ! Let me look.
USER: sorry 15232
[PARAMETER STATUS]: location = (no matched answer found)
GUARDIAN: Look outside and tell me the weather please.
GUARDIAN: http://www.weather.com/weather/hourbyhour/l/Pittsburgh+PA+15232:4:US

Template Generation

We also analyzed the click data collected in the experiments to demonstrate the feasibility of generating a response template. As mentioned above, Guardian records two types of data when generating the response: the proposed response text, and the click data. When the crowd workers explore the interactive visualization of the JSON object, we keep track of all filed names and values that the crowd clicked through. From our experiments, a total of 273 unique clicks were collected, and 77 were from the Yelp Search API. We manually annotated the distribution of the category of the fields (Table 4). After filtering out the URLs and the clicks that occurred in the first layer of the JSON object, this result suggests a promising future of capturing important fields.

Field Category	#	%
Number of Business Retrieved (1st entry of the top layer of JSON)	27	35.1%
URL	17	22.1%
Name	12	15.6%
Phone Number	9	11.7%
Neiborhood or Address	3	3.9%
Review Count	3	3.9%
Rating	2	2.6%
Snippet Text	2	2.6%
Latitude and Longitude	1	1.3%
Menu Date Updated	1	1.3%
Sum	**77**	**100.0%**

Table 4: Distribution of the crowd worker's mouse clicks when exploring the Yelp Search API's JSON result. This distribution reflects the important fields in the JSON object.

Discussion

In this section, we discuss some practical issues when implementing the system, as well as some additional insights from creating Guardian.

Portability and Generalizability

On one hand, the Guardian framework has a great portability. It is worth mentioning that we ported our original Guardian system based on the Yelp Search Yelp to two other web APIs performed in the on-line phase experiments in less than one day. It only requires the implementation of a wrapper of a given web API that the system is able to send the filled parameters to the API. All other remaining work can be performed by the crowd. The system's great portability makes it possible to convert hundreds of more web APIs to dialog systems.

On the other hand, some challenges do exists when we plan to generalize this framework. In our experiment, the Weather Underground API has a more strict standard about the format of the input parameter value than other two APIs. As a consequence, the "JSON valid rate" significantly drops, mainly due to the incorrect input format. Although this problem can be easily fixed by adding an input validator, it raises two important questions about generalizability: First, we could domain-specific knowledge – such as adding an input validator for a specific API – be this would be the main bottleneck in integrating hundreds or thousands of APIs into Guardian? (If yes, how do we overcome this?) Second, not all web APIs are created equal – some are more easily translated into a spoken dialog system than others. Additionally, as mentioned in the Introduction section, there are more than 13,000 web APIs, so how do we correctly choose which one to use for a given query?

Connections to Modern Dialog System Research

Our work is largely inspired by the research of modern dialog systems, *e.g.*, slot filling and dialog management. To assess our work, we compare our selected parameters for Yelp Search API to the slots suggested by the modern research of dialog systems on a similar task, i.e., restaurant queries. "Cambridge University SLU corpus" (Henderson et al. 2012) is a dialog corpus of a real-world restaurant information system. It suggests 10 slots for a restaurant query task: "addr"(address), "area", "food", "name", "phone", "postcode", "price range", "signature", "task", and "type". By comparing these slots against the selected parameters of Yelp API in our work, the "location" parameter can be mapped to the "addr" and "area" slots, and our "term" and "category_filter" can be mapped to the "food" slot. From the perspective of dialog system research, this comparison suggests that the offline phase of the Guardian framework can also be viewed as a crowd-powered slot induction process, and it is able to produce a compatible output with expert-suggested (Henderson et al. 2012) or automatic induced slots (Chen, Wang, and Rudnicky 2013).

Conclusion and Future Work

In this paper, we have introduced a crowd-powered web-API-based spoken dialog system (SDS) called Guardian. Guardian leverages the wealth of information in web APIs to enlarge the scope of the information that can be automatically found. The crowd is then employed to bridge the SDS with the web APIs (offline phase), and a user with the SDS (online phase). Our experiments demonstrated that Guardian is effective in associating questions with important web API parameters (QA-parameter matching), and can achieve a task completion rate of 97% in real-world dialog experiments on three different tasks. In the future, these dialog systems could be generated dynamically, as the need for them arises, making automation a gradual process that occurs based on user interests. Intent recognition can also aid this lazy-loading process by determining a user's goal and drawing on prior interactions, even by others, to collaboratively create these systems.

Acknowledgments

This work was supported by Yahoo! InMind and National Science Foundation award #IIS-1149709.

We thank our anonymous reviewers for their comments. We are also grateful to Shoou-I Yu, Shihyun Lo, Chih-Yi Lin, Lingpeng Kong, Yun-Nung Chen, and Chu-Cheng Lin for their help.

References

Bernstein, M. S.; Little, G.; Miller, R. C.; Hartmann, B.; Ackerman, M. S.; Karger, D. R.; Crowell, D.; and Panovich, K. 2010. Soylent: a word processor with a crowd inside. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, 313–322. ACM.

Bernstein, M. S.; Brandt, J.; Miller, R. C.; and Karger, D. R. 2011. Crowds in two seconds: Enabling realtime crowd-powered interfaces. In *Proceedings of the 24th annual ACM*

symposium on User interface software and technology, 33–42. ACM.

Bigham, J. P.; Cavender, A. C.; Kaminsky, R. S.; Prince, C. M.; and Robison, T. S. 2008. Transcendence: Enabling a personal view of the deep web. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, IUI '08, 169–178. New York, NY, USA: ACM.

Bigham, J. P.; Jayant, C.; Ji, H.; Little, G.; Miller, A.; Miller, R. C.; Miller, R.; Tatarowicz, A.; White, B.; White, S.; et al. 2010. Vizwiz: nearly real-time answers to visual questions. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, 333–342. ACM.

Bobrow, D. G.; Kaplan, R. M.; Kay, M.; Norman, D. A.; Thompson, H.; and Winograd, T. 1977. Gus, a frame-driven dialog system. *Artificial intelligence* 8(2):155–173.

Bohus, D.; Puerto, S. G.; Huggins-Daines, D.; Keri, V.; Krishna, G.; Kumar, R.; Raux, A.; and Tomko, S. 2007. Conquest: an open-source dialog system for conferences. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Companion Volume, Short Papers*, 9–12. Association for Computational Linguistics.

Chen, Y.-N.; Wang, W. Y.; and Rudnicky, A. I. 2013. Unsupervised induction and filling of semantic slots for spoken dialogue systems using frame-semantic parsing. In *Proceedings of 2013 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU 2013)*, 120–125. Olomouc, Czech: IEEE.

Chilton, L. B. 2009. Seaweed: A web application for designing economic games. In *Master's Thesis, MIT*.

Cooper, S.; Khatib, F.; Treuille, A.; Barbero, J.; Lee, J.; Beenen, M.; Leaver-Fay, A.; Baker, D.; Popović, Z.; et al. 2010. Predicting protein structures with a multiplayer online game. *Nature* 466(7307):756–760.

Ferguson, G.; Allen, J. F.; et al. 1998. Trips: An integrated intelligent problem-solving assistant. In *AAAI/IAAI*, 567–572.

Hartmann, B.; Wu, L.; Collins, K.; and Klemmer, S. R. 2007. Programming by a sample: Rapidly creating web applications with d.mix. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, 241–250. New York, NY, USA: ACM.

Henderson, M.; Gašić, M.; Thomson, B.; Tsiakoulis, P.; Yu, K.; and Young, S. 2012. Discriminative Spoken Language Understanding Using Word Confusion Networks. In *Spoken Language Technology Workshop, 2012. IEEE*.

Horvitz, E., and Paek, T. 1999. A computational architecture for conversation. *COURSES AND LECTURES-INTERNATIONAL CENTRE FOR MECHANICAL SCIENCES* 201–210.

Lasecki, W.; Miller, C.; Sadilek, A.; Abumoussa, A.; Borrello, D.; Kushalnagar, R.; and Bigham, J. 2012. Real-time captioning by groups of non-experts. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, 23–34. ACM.

Lasecki, W. S.; Wesley, R.; Nichols, J.; Kulkarni, A.; Allen, J. F.; and Bigham, J. P. 2013. Chorus: a crowd-powered conversational assistant. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, 151–162. ACM.

Lee, C.; Jung, S.; Kim, S.; and Lee, G. G. 2009. Example-based dialog modeling for practical multi-domain dialog system. *Speech Communication* 51(5):466–484.

Litman, D. J., and Silliman, S. 2004. Itspoke: An intelligent tutoring spoken dialogue system. In *Demonstration Papers at HLT-NAACL 2004*, 5–8. Association for Computational Linguistics.

Maulsby, D.; Greenberg, S.; and Mander, R. 1993. Prototyping an intelligent agent through wizard of oz. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, 277–284. ACM.

Singh, P.; Lasecki, W. S.; Barelli, P.; and Bigham, J. P. 2012. Hivemind: A framework for optimizing open-ended responses from the crowd. Technical report, URCS Technical Report.

Tomko, S. 2005. Improving user interaction with spoken dialog systems via shaping. In *CHI'05 Extended Abstracts on Human Factors in Computing Systems*, 1130–1131. ACM.

Von Ahn, L., and Dabbish, L. 2004a. Labeling images with a computer game. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 319–326. ACM.

von Ahn, L., and Dabbish, L. 2004b. Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, 319–326. New York, NY, USA: ACM.

Von Ahn, L. 2009. Human computation. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, 418–419. IEEE.

Xu, W., and Rudnicky, A. I. 2000. Language modeling for dialog system.