Thesis Proposal

A Crowd-Powered Conversational Assistant That Automates Itself Over Time

Ting-Hao (Kenneth) Huang

January 11th, 2017

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Thesis Committee:
Jeffrey P. Bigham, Carnegie Mellon University (Chair)
Alexander I. Rudnicky, Carnegie Mellon University
Niki Kittur, Carnegie Mellon University
Walter S. Lasecki, University of Michigan
Chris Callison-Burch, University of Pennsylvania

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: crowdsourcing, dialog system, conversational agent, chatbot, real-time crowdsourcing, crowd-powered system

Abstract

Interaction in rich natural language enables people to exchange thoughts efficiently and come to a shared understanding quickly. Modern personal intelligent assistants such as Apple's Siri and Amazon's Echo all utilize conversational interfaces as their primary communication channels, and illustrate a future that in which getting help from a computer is as easy as asking a friend. However, despite decades of research, modern conversational assistants are still limited in domain, expressiveness, and robustness. In this thesis, we take an alternative approach that blends real-time human computation with artificial intelligence to reliably engage in conversations. Instead of bootstrapping automation from the bottom up with only automatic components, we start with our crowd-powered conversational assistant, *Chorus*, and create a framework that enables Chorus to automate itself over time. Each of Chorus' response is proposed and voted on by a group of crowd workers in real-time. Toward realizing the goal of full automation, we *(i)* augmented Chorus' capability by connecting it with sensors and effectors on smartphones so that users can safely control them via conversations, and *(ii)* deployed Chorus to the public as a Google Hangouts chatbot to collect a large corpus of conversations to help speed automation. The deployed Chorus also provides a working system to experiment automated approaches. In the future, we will *(iii)* create a framework that enables Chorus to automate itself over time by automatically obtaining response candidates from multiple dialog systems and selecting appropriate responses based on the current conversation. Over time, the automated systems will take over more responsibility in Chorus, not only helping us to deploy robust conversational assistants before we know how to automated everything, but also allowing us to drive down costs and gradually reduce reliance on the crowd.

Contents

Chapter 1

Introduction

1.1 Motivation

This thesis imagines a future in which people can converse with machines as naturally and effectively as they do with one another to find and make sense of information. It imagines a student conversing with an agent on her watch to quickly narrow in on college assistance programs that make sense for her; it imagines the agent setting an alarm for the student by discussing with her about when should she leave in the morning for an early class; and it imagines the agent leveraging various resources to help her figure out a summer travel plan in Europe. The imagined agent communicates in natural language, adapts on-the-fly to new information, asks questions to narrow in on exactly what the user wants, remembers and personalizes itself based on recent interactions, and continually learns from its experiences to make interaction fluid and efficient.

Unfortunately, this agent is currently hard to build because it requires simultaneously solving multiple problems that each represent research areas unto themselves. Researchers have tried to combine multiple dialog systems of different domains to form a single agent [119], to adapt a model trained in one domain to another [101, 110], and build chit-chat systems for general conversations [5]. However, building an agent that can hold open conversations with users is a still very challenging task today.In 2016, Amazon launched the "Alexa Prize" and grant $2.5 million to the team who can develop a system that "achieves the grand challenge of conversing coherently and engagingly with humans on popular topics for 20 minutes" within a year [98].

This thesis explores a complementary top-down approach to create robust conversational assistance. Instead of bootstrapping automation from the bottom up with only automatic components, we start with our deployable crowd-powered conversational assistant *Chorus* [57, 69], and create a framework that enables Chorus to automate itself over time. We deploy Chorus to collect examples of conversational assistance and to better understand how users of such a general system want to interact with it. The system is now completely crowd-powered. A group of crowd workers suggest responses and vote through the ones they believe to be best. An incentive mechanism encourages the crowd to remain on task and provide quality input [100]. Building upon the deployed Chorus, the goal of this thesis is to create a framework that enables Chorus to gradually replace its crowd-powered components with automated approaches by using the data it collects, and thus not only helping us to deploy robust conversational assistants before we know

how to automated everything, but also allowing us to drive down costs and gradually reduce reliance on the crowd.

1.2 Why Conversational Interaction?

Conversation is powerful because the back-and-forth interaction in rich natural language allows participants to quickly come to a shared understanding. Importantly, human conversations start with a shared basis of commonsense expectations. Conversation creates a shared context that efficiently directs the exchange and allows goals to be achieved quickly, but is difficult for computers to replicate today. This context is preserved across interactions so that not everything needs to be stated explicitly during each conversation.

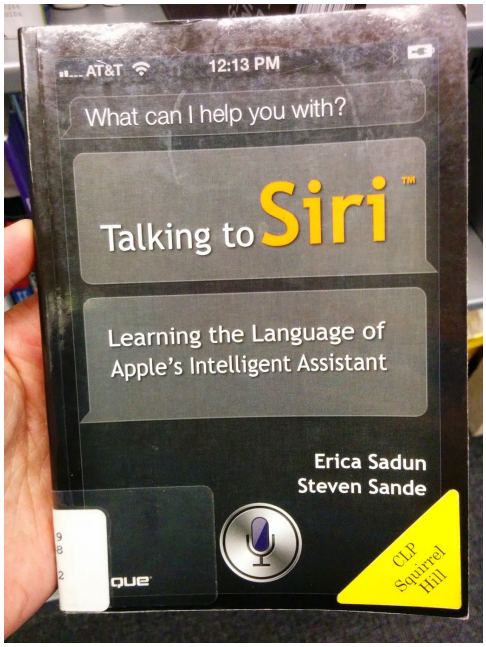

Figure 1.1: "Talking to Siri," a book that illustrates that despite advances, the onus is still on the user to adapt to what conversational assistants can understand [94]. Chorus changes that by having the crowd respond. This proposal is about allowing Chorus to automate itself over time.

Computers now have access to an incredible amount of information, but finding and making sense of that information remains a difficult, laborious process. Robust conversational assistance promises a future in which narrowing down what information the computer can usefully provide you is as easy as asking a friend. The difference is that while a friend can not always be available and can not be an expert on all topics, computers can. Using natural language dialog to interact with software has been a goal of artificial intelligence since the early days of computing, but the complexity of human language has made robustly handling two-way conversation with software agents a consistent challenge [2]. Apple's Siri is but the latest examples of a conversational assistant at first greeted with substantial enthusiasm that ended up largely disappointing users with its limitations [25]. Despite decades of research, conversational assistants are still limited in *(i)* the domains in which they work, *(ii)* the richness of expression they support, and *(iii)* the robustness to variations in topic, domain, and user. Thus, existing dialog-based software systems generally rely on a fixed input vocabulary or restricted phrasings (Figure 1.1), have a limited or no memory of past interactions, and use a fixed output vocabulary.

Chorus takes a different approach that blends real-time human computation [13, 66, 69] with artificial intelligence to reliably engage in conversation. While humans maintain natural-language conversation with ease, it is often infeasible, unscalable, or expensive to hire individual humans (especially experts) to act as conversational partners. Thus, an important component

for crowd-powered systems is their ability to maintain the scalability of machines by learning over time to replace human intelligence. In the initial version of Chorus, the content of the conversation is generated entirely by the crowd. End users send messages to the crowd, the crowd collects and proposes responses, and then decides which to forward back. Interface helps the crowd carry on a consistent conversation efficiently as a collective, even as individual crowd members come and go. Over time, the automated systems will take over more responsibility in the deployed system, helping to drive down costs and make the system more feasible. Because the system can always fall back on the crowd, automated approaches can be released and tested in the wild before they are perfect, helping us to understand how they perform in the real world early on. Including these components in a real system will provide an excellent feedback loop to help improve the automated system itself.

1.3 Research Plan

This thesis is organized into three main parts, including *(i)* augmenting capability of Chorus, *(ii)* deploying the crowd-powered version of Chorus to the real world, and *(iii)* enabling Chorus to gradually replace the crowd with automated approaches overtime. This six projects under these three topics are shown in Figure 1.2.

1.3.1 Part I: Expanding the Capabilities of a Crowd-Powered Agent

Toward a conversational assistant that can help users in various ways, we would like to augment the capability of Chorus in addition to providing information. Personal assistants such as Google Now or Amazon's Echo are capable to follow user's verbal commands to setup sensors (also known as "triggers", such as clock) and effectors (also "actions", such as alarm) in smartphones or smart homes. **Part I contains a finished project, InstructableCrowd**, in which we explored Chorus' potential of using a crowd-powered conversational interface to help users control devices around them.

InstructableCrowd: Creating IF-THEN Rules via Conversation with the Crowd (Chapter 3)

A limitation of Chorus is that it can only provide information to its users: it can not *do* anything or interact with the user's environment. This is an obvious shortcoming because smartphones contain a wealth of sensors and effectors that could be combined to perform useful tasks and be customized to their users. For instance, people living in colder climates may want their phone to wake them up earlier than usual if it has snowed overnight to sweep snow off their car and for the traffic delays. Similarly, people may want their phones to automatically text their spouse and request a ride if their usual bus is running late. However, giving the crowd access to your device for them to do these things on your behalf is unwise because we do not know who the crowd is or whether each individual can be trusted. In response to this situation, we created *InstructableCrowd*, a system that allows end users to create rich, multi-part IF-THEN rules via conversation with the crowd. Users verbally express a problem to crowd workers, who collectively program

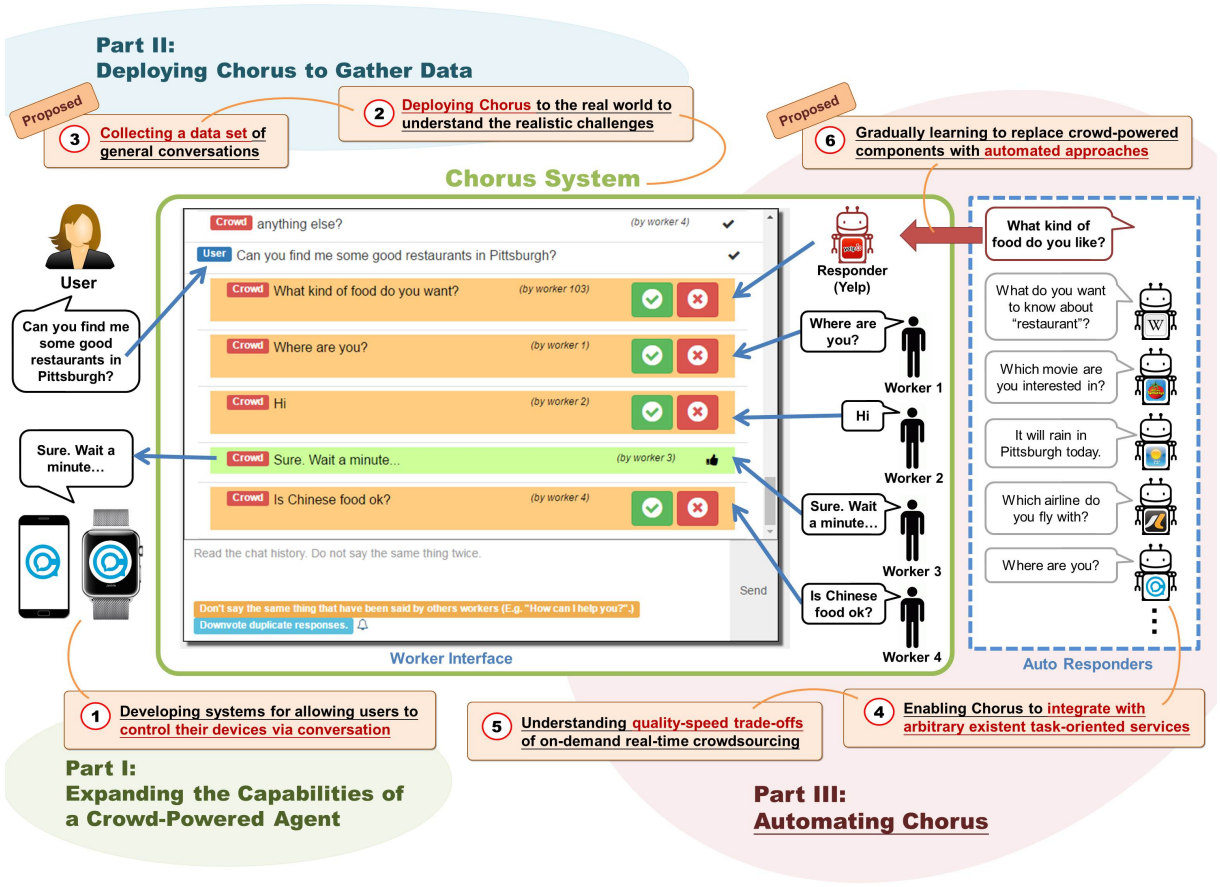

Figure 1.2: Overview of the research plan. This thesis is organized into three main parts, including *(i)* augmenting capability of Chorus, *(ii)* deploying the crowd-powered version of Chorus to the real world, and *(iii)* enabling Chorus to gradually replace the crowd with automated approaches overtime. First, to make Chorus more useful, we *(1)* connected it with sensors and effectors of smartphones for allowing users to control their devices via conversation. Second, we *(2)* deployed Chorus to the real world to collect data for automation, and will *(3)* release the Chorus Dataset to the AI community (proposed work). Finally, for automating Chorus, we *(4)* enabled Chorus to integrate with arbitrary existent task-oriented services and *(5)* studied the quality-speed trade-offs of on-demand real-time crowdsourcing. We will create a framework that *(6)* gradually learns to replace Chorus' crowd-powered components with automated approaches (proposed work.)

relevant IF-THEN rules to help them via conversation. InstructableCrowd allows users to create these rules via voice, on-the-go, and does not require dealing with a complicated interface. Our study with 12 non-programmers showed that InstructableCrowd achieves a similar average F1-score (0.93) in selecting sensor/effector as users themselves (0.94), and an accuracy of 90.7% in filling attributes for those sensor/effectors which were correctly chosen. Incremental editing on crowd-created rules resulted in an even better performance. InstructableCrowd illustrates how users may converse with the crowd to personalize their increasingly powerful and complicated devices.

1.3.2 Part II: Deploying Chorus to Gather Data

For automating Chorus, we not only need to understand the conversations that people actually want to have, but also need data for machine-learning algorithms to learn how to automate each crowd-powered component. Therefore, we deployed Chorus to the public as a Google Hangouts chatbot. The collected data is useful for training automated components and for understanding conversations between users and personal assistants. The already-deployed system also well serves as an experiment platform for exploring automation approaches. **Part II describes the deployment of Chorus, which started in May 2016, and proposes to release the *Chorus Dataset* in the future.**

Chorus Deployment (Chapter 4)

Intelligent conversational assistants, such as Apple's Siri, Microsoft's Cortana, and Amazon's Echo, have quickly become a part of our digital life. However, these assistants have major limitations, which prevents users from conversing with them as they would with human dialog partners. This limits our ability to observe how users really want to interact with the underlying system. To address this problem, we developed a crowd-powered conversational assistant, Chorus, and deployed it to see how users and workers would interact together when mediated by the system. Chorus sophisticatedly converses with end users over time by recruiting workers on demand, which in turn decide what might be the best response for each user sentence. Chorus was launched on May 20th, 2016. Up to date, more than 100 users have held conversations with Chorus during more than 1000 conversational sessions. In this section, we present an account of Chorus' deployment, with a focus on four challenges: *(i)* identifying when conversations are over, *(ii)* malicious users and workers, *(iii)* on-demand recruiting, and *(iv)* settings in which consensus is not enough. Our observations could assist the deployment of crowd-powered conversation systems and crowd-powered systems in general.

Chorus Dataset (Proposed Work. Chapter 5)

We plan to release the data we collected during Chorus' deployment as the *Chorus Dataset*. Our goal is to provide an usable dataset for the AI community to study conversations between users and a conversational assistant that has human-level intelligence. For public release, data pre-processing is required, including anonymization, removing (or marking) inappropriate content and spammed conversations, and correcting the falsely-segmented conversations, etc.

1.3.3 Part III: Automating Chorus

Our ultimate goal is to create a crowd-powered conversational assistant that learns gradually to automate itself. Initially, the crowd will power Chorus with minimal automated assistance, such as automatic welcome message. Over time, the automated systems will take over more responsibility in the deployed Chorus. A framework that learns *(i)* to select from a set of external dialog systems to obtain responses and *(ii)* to select good messages from a set of candidate responses can gradually replace human workers and help to drive down costs and make Chorus more feasible. To make it easy to expand the set of external dialog systems, we develop a crowd-powered framework, *Guardian*, that converts existing services such as Web APIs (*e.g.,* Yelp API) into dialog systems. For designing a better crowd-powered dialog system, we also studied how fast and how good can crowd workers extract key information from a running dialog.

Because the system can always fall back on the crowd, automated approaches can be released and tested in the wild before they are perfect, helping us to understand how they perform in the real world early on. The deployed system also gives us a natural way to evaluate the benefit of each component *in vivo* within Chorus, rather than designing tasks in isolation with no way of knowing a priori whether they can effectively be combined into a working system.

Part III includes two finished projects, the Guardian and the study of quality-speed trade-offs of real-time crowdsourcing, and proposes to implement and evaluate the automation framework.

Guardian: A Crowd-Powered Spoken Dialog System for Web APIs (Chapter 6)

How to empower Chorus with millions of existing services such as Web APIs? We design *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.

Understanding Quality-Speed Trade-offs of On-demand Real-time Crowdsourcing in Dialog Systems (Chapter 7)

When users interact with on-line bots such as Chorus or Guardian, they expect to have longer response times (roughly 10 to 30 seconds). This range of latency allows real-time crowdsourcing techniques to intervene. However, the literature has little to say about speed-quality trade-offs when the time budget is only few seconds. If workers have as long as they want to annotate a sentence, most AI systems would assume the annotation is trustworthy. It was not clear that this assumption would hold when workers have only 20 seconds. To bridge this gap, we select entity extraction, which is the main sub-task of language understanding in modern dialog systems, as a show case to explore quality-speed trade-offs of on-demand real-time crowdsourcing.

We propose a crowd-powered solution, which is based on the ESP game for image labeling, to extract key information from a running dialog. When multiple players agree, entities can be reliably extracted from an utterance. This approach is advantageous because it does not require

training data. Further, it is robust to unexpected input and capable of recognizing new entities. Our approach achieves better F1-scores than that of the automated baseline for complex queries with a reasonable response time. The proposed method is also evaluated via Google Hangouts' text chat and demonstrates the feasibility of real-time crowd-powered entity extraction.

A Crowd-Powered Conversational Assistant that Automates Itself Over Time (Proposed Work, Chapter 8)

By treating an external dialog system as a "black box", they could be seen by Chorus as just new workers whose input can be considered – directly being fed input from the user, and responding with candidate responses for other workers to verify or reject. This could provide a natural mechanism for evaluating the effectiveness of existing dialog systems in the context of open-ended conversations, and can be used as a method for collecting labeled training data that could ultimately help improve the dialog systems itself. The main challenge we plan to address is the coordination between Chorus and external systems, especially task-oriented dialog systems. Most existing dialog systems are task-oriented, but would lose track when the conversation moves to a different topic. Chorus is able to handle open-ended conversation, but has no strict control of dialog flow. Therefore, incorporating task-oriented dialog systems into Chorus becomes a significant challenge.

We will investigate techniques for detecting when a user's utterance falls within the scope of an existing dialog system, allowing control to be transferred away from workers, and when the conversation wanders outside the range of the system's capabilities, requiring renewed support from the crowd.

Chapter 2

Related Work

The proposed research is related to *(i)* dialog systems and conversational user interfaces, *(ii)* real-time crowd-powered systems and crowd agents, and *(iii)* crowdsourced web search and question answering.

2.1 Dialog Systems and Conversational User Interfaces

There is a considerable body of research on systems supporting dialog between human and machines. Much of the work, however, focuses on very narrow domains with highly engineered rule-bases using simple finite-state or frame-based frameworks for their dialog model. See Mctear [85] for an excellent survey. Because such technologies support very limited applications, they do not provide useful frameworks for the open-ended nature of the dialogs involved in the proposed project. Perhaps the most successful system to date using these methods is Apple's Siri system, and it can only handle a limited number of pre-anticipated situations, each of which is highly engineered. One of the greatest weaknesses of such systems is the lack of a discourse model that can support clarification and correction dialogs in a general way. More general systems have been developed by Bohus and Rudnicky [16, 17]. These systems construct dialog models on top of more general models of AI planning and task models. Bohus and Rudnicky's framework is much like the execution of an HTN-like plan [88], while Allen and Ferguson's work have focused on developing general models of abstract problem solving [1]. Prior work also exists on ensemble approaches to dialog in which multiple component dialog systems are combined into one system, such as in the DialPort project at CMU [119]. Popular conversational assistants like Siri, Cortana, and Alexa are also ensembles of the capabilities (or skills) they have access to. In general, these dialog systems only allow for a small number of conversational turns (usually one), generally invoked with a command the user must know in advance (Figure 1.1), which severely limits the advantages of dialog. The work proposed here would allow conversation to fluidly pull from a number of dialog systems, chatbots, and the crowd.

2.2 Real-Time Crowd-Powered Systems

Chorus builds on prior work in human computation. Human computation [107] has been shown to be useful in many areas, including data collection [55], writing and editing [7], image description and interpretation [108], and protein folding [29]. Chorus aims to enable a conversation with a crowd of workers in order to leverage human computation in new ways [14]. Existing abstractions obtain quality work by introducing redundancy into tasks to verify results [63, 81]. For instance, the ESP Game uses answer agreement [108] and Soylent uses the multiple-step find-fix-verify pattern [7].

Early crowdsourcing systems leveraged human intelligence through batches of tasks that were completed over hours or days. For example, while the ESP Game [108] paired workers synchronously to allow them to play an interactive image-label guessing game, it did not provide low latency response for any individual label. Since these approaches take time, they are not always suitable for interactive real-time applications.

VizWiz [13] was one of the first systems to elicit nearly real time response from the crowd. It introduced a queuing model to help ensure that workers were available both quickly and on demand. For Chorus to be available on demand requires multiple users to be available at the same time in order to collectively contribute. Prior systems have shown that multiple workers can be recruited for collaboration by having workers wait until a sufficient number of workers have arrived [28, 108]. Bernstein et al. [8] showed that the latency to direct a worker to a task can be reduced to below a couple of seconds by combining the concepts of queuing and waiting to recruit crowds (groups) from existing sources of crowd workers. Further work has used queuing theory to show that this latency can be reduced to under a second and has also established reliability bounds on using the crowd in this manner [9]. Lasecki et al. [66] introduced continuous real-time crowdsourcing in Legion, a system that allowed a crowd of workers to interact with a UI control task over an unbounded, on-going task. The average response latency of control actions in Legion was typically under a second. Scribe [65] provides real-time captions for deaf and hard of hearing users with a per-word latency of under 3 seconds. With the advent of real-time crowd-powered systems, the concept of (real-time) "crowd agnet" has been shown useful across many domain [71]. More recently, Salisbury et al. [95] provided new task-specific mediation strategies that further reduce overall task completion time in robotic control tasks. Savenkov et al. [97] created a human-in-the-loop instant question answering system to participate the TREC LiveQA challenge. The APP "1Q"[1] uses smartphones' push notifications to ask poll questions and collect responses from target audiences instantly.

2.3 Crowdsourced Search and Question Answering

Workers collaborating to form responses in Chorus is a kind of Groupware [39] (also known as Collaborative Software), which is a system that are designed for helping a group of people involved in a task to achieve their goals [113]. Morris and Horvitz explored the concept of collaborative work and developed SearchTogether to allow multiple users asynchronously and remotely search on the same topic together [87]. Later Amershi and Morris developed CoSearch

[1] 1Q: https://1q.com/

to support a group of co-located users to search collaboratively [4]. In the field of human computation, various crowd-powered systems also proposed to use a group of crowd workers to collectively answer questions (QA). The Knowledge Accelerator [41] uses a workflow to have a group of crowd workers collectively create Wikipedia-style articles to answer open questions such as "What are the best attractions in LA if I have two little kids?" Savenkov and Agichtein created CRQA, Crowd-powered Real-time Automatic Question Answering System, that combines automatic question answering system and human computation to tackle the LiveQA challenge in TREC 2016 [96]. VizWiz was one of the first real-time crowd-powered Question-Answering systems that were deployed to the public [13]. VizWiz was designed to have crowd workers answer visual questions sent from users who are blind or visually impaired.

Chorus is also similar in many respects to other on-line systems that allow for group communication, *e.g.* message boards and forums. Community-based QA (CQA) services such as Quora[2] and Yahoo Answer[3] leverage the whole user community to volunteer to answer questions posted by users. Some other examples of crowd mediation of responses exist, *e.g.*, the reddit.com "ask me anything" (AMA), but this has not been done in the context of real-time conversation over multiple turns. Work generating answers to long-tail search engine queries using search logs and crowdsourcing is similarly single-turn and offline [10]. Those submitting the original question can also respond to the answers provided, and give feedback concerning issues the group has with the query. Prior work has also looked at providing specific answers to a wide range of uncommon questions searched for on the web by having workers extract answers from automatically generated candidate web pages [10]. Systems such as ChaCha[4] try to get answers back to users in nearly real time from individual workers, but do so without considering the history of the user. Recent products like Facebook M [48], Magic[5], and FancyHands[6] have humans in the loop of the conversational assistant, but use paid contractors (for whom confidentiality can be contractually assured).

[2]Quora: https://www.quora.com/
[3]Yahoo Answer: https://answers.yahoo.com/
[4]ChaCha: www.chacha.com
[5]Magic: https://getmagic.com/
[6]FancyHands: https://www.fancyhands.com/

Part I

Expanding the Capabilities of a Crowd-Powered Agent

Chapter 3

InstructableCrowd: Creating IF-THEN Rules via Conversation with the Crowd

3.1 Introduction

Smartphones contain a wealth of sensors and effectors that could be combined to perform useful tasks and be customized to their users. For instance, people living in colder climates may want their phone to wake them up earlier than usual if it has snowed overnight to sweep snow off their car and for the traffic delays (to sweep snow and for the inevitable traffic delays.) Similarly, people may want their phones to automatically text their spouse and request a ride if their usual bus is running late. As a final example, phones already remind people about upcoming meetings, but such notifications are hardly useful if one has already arrived to the meeting early. Sensors and effectors on phones enable a broad spectrum of applications, but it is difficult for application developers to envision all the behaviors that users want ahead of time, especially for rules that encode individual preferences. For example, not all users would want to text their spouse if their bus is delayed.

End-users are the logical authors of such personal rules, and several platforms suggest shifting the development of rules to end users who create rules that capture the behaviors they would like their phones to exhibit. The most prominent example of this promising approach is the mobile application IFTTT (If This Then That, ifttt.com). IFTTT enables end-users to author simple IF-THEN rules (also known as Trigger-Action rules) which contain one trigger (*e.g.*, a post on Twitter) and one action (*e.g.*, synchronize the latest Twitter post to Facebook). IFTTT has over millions of installs today [58].

One limitation of IFTTT is its simplification of rules [32, 52]. Research showed that 22% of behaviors that people came up with require more than one sensor (trigger) or effector (action) [106], but IFTTT allows only one sensor and one effector. The complexity of rules people would like to create is likely to only increase as IFTTT (and other services) continue to be integrated with more services and devices.

We identified four main challenges users are likely to encounter when creating complex IF-THEN rules in the real world: First, in the real world, **rules are inspired in-the-moment**. The impetus for many rules that users would like to create often occurs while they are on-the-go.

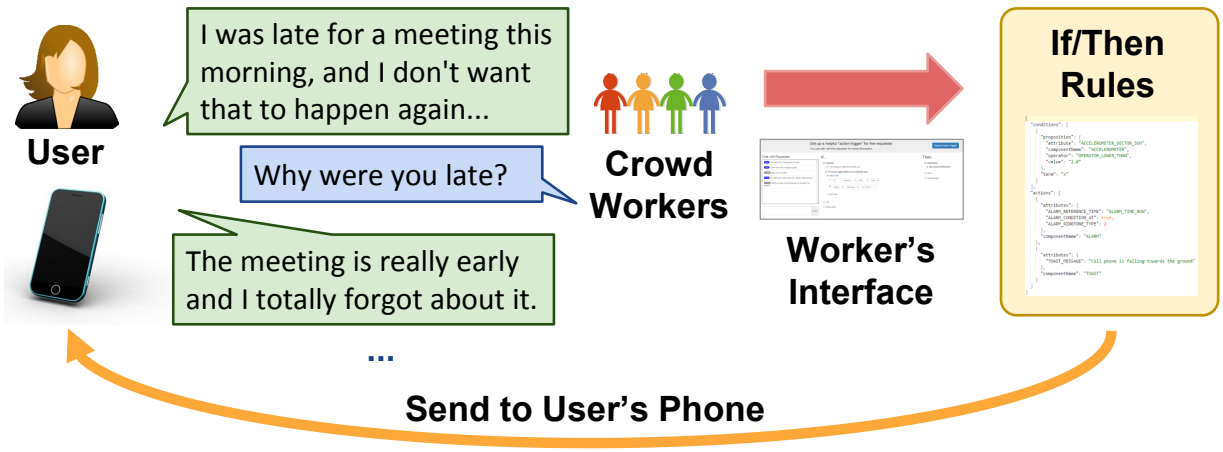

Figure 3.1: Users have a conversation with InstructableCrowd to create If/Then rules that run on their phone to solve problems. The backend system is run by synchronous crowd workers who respond to the user, ask follow up questions, and create rules. Users can then review the rules on their phone to make sure they were what they wanted.

People are likely to remember contextual details in the moment, and recall reduces over time. For example, they could create a rule designed to avoid being late again due traffic jam while sitting in the traffic jam, or create a rule to avoid missing the bus again while walking home after having missed it. Second, when users try to compose rules on-the-go, **mobile devices are small**. It would be difficult for end-users to compose multi-part rules which have complex conditions from smartphones, not to mention smaller devices such as smartwatch. Third, even more, for many devices users may want to program IF-THEN rules on **speech is now the primary input modality**. Devices such as smartwatches, Google Glass and Amazon Echo use voice as a primary or the only input. Existing rule composition interfaces do not work on these devices. Finally, furthermore, **end-users are required to know the capability of sensors and effectors.** To use IFTTT effectively, users must be aware of the sensors and effectors that platform makes available. IFTTT has more than 300 channels today, it would be hard for each user to fully understand all of them, especially for complex rules. Available sensors and effectors are likely to grow enormously in the future.

In response to these challenges, we introduce *InstructableCrowd*, a system that allows end users to create rich, multi-part IF-THEN rules via conversation with the crowd (Figure 3.1). A group of crowd workers are recruited on-demand to talk with the user and create rules for him/her based on their conversation. With intelligent workers on a rich desktop interface supporting them, the user interface can be simplified into a familiar speech or text chat client, allowing the system to be used on-the-go via mobile and wearable devices. Furthermore, users can discuss their problems with the crowd and get feedback to refine their requests. The users may know their problems, but not know what solutions would best resolve them in the future. The crowd can help the users identify possible solutions and then actually create the rule. InstructableCrowd then lets users edit and improve the created rule. Controlled experiments showed that users are able to create complex rules using InstructableCrowd.

Through InstructableCrowd, we introduce a new method for enabling end users to program

complex interactions over the wealth of sensors and effectors on their smartphones, which may have broader implications for the future of programming with speech. In particular, we demonstrate the value of *(i)* a conversational programming interface, *(ii)* crowd-powered rule creation, and *(iii)* merging rules by voting.

3.2 Related Work

InstructableCrowd is related to prior work on *(i)* end user programming, and *(ii)* personal intelligent agents.

3.2.1 End User Programming

InstructableCrowd builds upon the long history of research and products of end-user programming [79], which aims at enabling non-programmers to author or compose their own applications. Early works in this field started from database [43] and email management [82], and later gradually became more common as more and more senors and effectors became available to general users [20, 22, 23, 31]. For instance, CoScripter allowed end users to program scripts by demonstration [12, 77]. CoScripter used its corpus of scripts to allow easier creation of new actions from mobile devices [75]; Sikuli is another famous end user programming project [117]. Sokuli allows users to take a screenshot of a GUI element (*e.g.*, a toolbar button) and then directly use it as an element in a programming script to control the GUI's behavior (*e.g.*, clock the button.)

Trigger-action programming is one simple model of end user programming that the user form a new functionality by combining pre-defined triggers (sensors of "IF") with pre-defined actions (effectors of "THEN"). Many solutions were proposed to realize trigger-action programming, such as using existing notations of business processes modeling (BPM) to represent rules [21], adopting an effective workflow to create rules [62, 64], or solutions for domain-specific applications [32]. The IFTTT project has had great success by simplifying the composition among two applications and providing a user-friendly workflow and interface on mobile phones. The concept of IFTTT has also been extended and adopted for use in various other domain, such as smart home applications [33, 106], cross-device interactions [38], the Internet of Things [104].

IFTTT only allows rules to be composed of a single trigger and a single action. Several frameworks were proposed to support multiple triggers (IFs) and actions (THENs). Dey et al. created an interface that users can drag and drop multiple sensors and effetors on a sheet to create new rules [34]. Huang et al. [52] and Ur et al. [106] both extended IFTTT's interface to allow users select more than one triggers or actions. However, most of these works focused on the challenges in designing interfaces or wrokflows for creating a rule and examined their solutions with participants using full-size monitors and keyboards, such as via Amazon Mechanical Turk. Only few works focused on issues raised by mobile devices when creating complex rules. Häkkilä *et al.* created a trigger-action programming system, Context Studio, on the Series 60 Nokia mobile phone back in 2005 [42]. While the mobile devices and sensors used in Context Studio were outdated, this project provided some early insights of challenges we face today. On the other hand, competitors of IFTTT, such as Tasker, Llama, AutomateIt, On{X}, Atooma, and

Microsoft's Flow all aimed to support multiple IFs and THENs in their product. However, none of these have achieved the same success as IFTTT.

Limitations of user programming were also studied. Daniel et al. [32] pointed out that mashups platforms aimed at non-programmers are either powerful but too hard to use, or easy but too simple to be practical. Huang et al. [52] studied the mental model of IFTTT users and found that users do not always correctly understand how a sensor/effector works, which causes errors in user-created rules. Recent work has been proposed which uses crowdsourcing to build software [74].

3.2.2 Personal Intelligent Agent

Personal intelligent agents are now available on most smartphones, *i.e.*, Google Now on Android, Siri on iOS, Cortana on Windows phones. Google Now is known for spontaneously understanding and predicting user's life pattern (*e.g.*, flight schedules, or "time to go home"), and automatically pushing notifications; Conversational agents such as Apple's Siri, Amazon's Echo, and Microsoft's Cortana also demonstrated their capability of understanding speech queries and helping with users' requests. However, all of these intelligent agents are limited in their ability to understand their users. Google Now only reacts to certain fixed set of events, and users have no manner to extend its capability based on their own needs; Siri and Echo can only perform speech queries, but are not able to hold a real conversation or proactively perform actions on the user's behalf. InstructableCrowd gives users the direct control to define intelligent behaviors their smartphones should perform, and uses the crowd to make creating these behaviors possible with conversational interaction.

In response to this situation, crowd-powered intelligent agents were proposed. Chorus is a crowd-powered assistant that can hold intelligent conversations [69] and has been deployed to public [57]. End-users speak to it, and it responds back quickly. Chorus is powered by a dynamic group of crowd workers (recruited on-demand) who propose responses and vote the best ones through. Automating parts of Chorus by having the crowd transition existing Web APIs (Application Programming Interfaces) to dialog systems can make it cheaper [54]; Alternatively, conversational assistants powered by trained human operators such as Magic (getmagicnow.com) and Facebook M have also appeared in recent years.

3.3 InstructableCrowd

InstructableCrowd is implemented as an Android mobile application for supporting (Figure 3.2). End users to converse with crowd workers and describe problems they encounter, such as *"I was late for a meeting this morning, and I don't want that to happen again."* The crowd workers can talk with the user and use an interface to select **sensors (IFs)** and **effectors (THENs)** to create an **If-Then rule** in response to the user's problem. The rules are then sent back to the user's phone. For example, if the user mentions having trouble with early morning meetings, the crowd can create the rule, "send a notification the night before a meeting" for the user; if the user wants extra time in the morning to clean up the car from the snow, the crowd can create a rule to that if the weather (cyber)sensor indicated that it snowed during the night then set the alarm 10 minutes

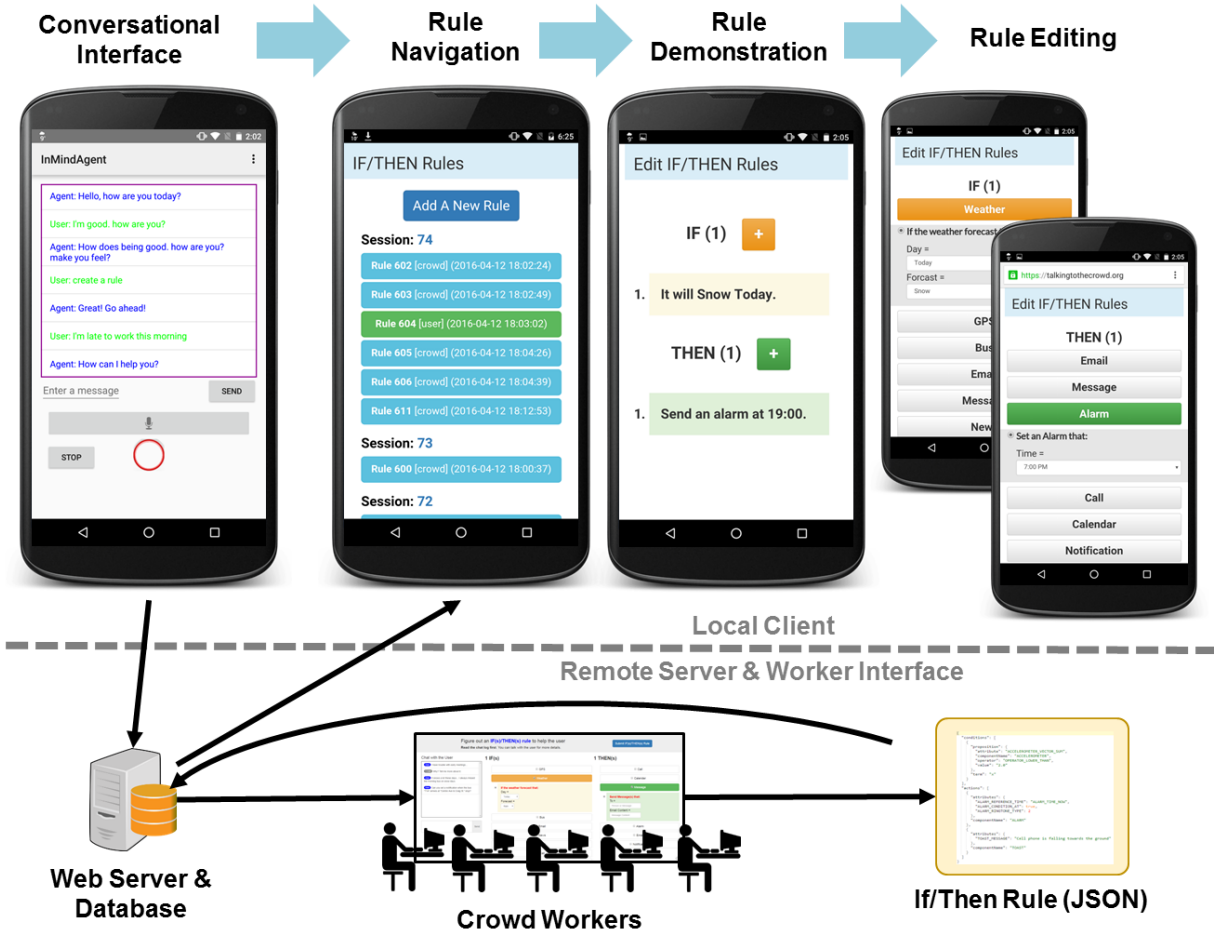

Figure 3.2: InstructableCrowd users have a conversation with crowd workers about a problem they are having. Crowd workers collectively create IF-THEN rules that may help the end user solve their problem using sensors and effectors available on the smartphone platform. The rules are then sent back to the user's phone for review, editing, and approval. The rules then run on the smartphone.

earlier than usual. Furthermore, InstructableCrowd is also able to merge multiple rules sent by different crowd workers to form a more reliable final rule.

3.3.1 Conversational Agent for the End-user

InstructableCrowd is implemented as a conversational agent for Android smartphones. By calling the personal agent's name or clicking on the red button (as shown in Figure 3.2), the user is able to give the agent commands via voice or text. The client side records the user's speech and sends it to the server, which in turn sends this speech on to Google Automatic Speech Recognition; the user can also use text entry to input the command. InstructableCrowd recognizes verbal commands such as *"create a rule"* to initiate the rule creation process. The end-user may then describe his problems and converse with the crowd to figure out which rules to create (the work-

ers converse by text, and the user, may either use text or voice). Once the rule is created, it is sent back to the user's phone and applied by a middleware component running on the phone. Currently, the system is implemented and tested on the Android OS 6.0.1. The server is implemented in Java.

3.3.2 Rule Editor for the End-user

InstructableCrowd also provides an editing interface for the user to manually create new rules, edit them and edit rules received from crowd workers. As shown in Figure 3.2, the user is able to navigate all received rules and click on each rule for additional details. All rules are grouped together by the conversational session in which the rule was created. Crowd-generated rules are blue, and the rules created or edited by the user are green. In order to ease on the comprehension of these rules, we created a template-based natural language description for each rule. For instance, the description template of "Weather" forecast sensor is "It will weather day." If the "Weather" sensor is selected, along with the "Day" attribute filled with "Tomorrow" and the "Forecast" attribute filled with "Snow", the displayed description will be "It will Snow Tomorrow." On the editing interface, the description will be generated automatically in real-time and enable the user to quickly check the rule they just created or edited. The user can also use this rule editor to manually create an IF-THEN rule from scratch on their phone without talking to the crowd. In our user study, participants use various approaches to create IF-THEN rules with InstructableCrowd. Our end user review and editing interface is inspired by the IFTTT mobile APP. However, we enables the user to combine multiple IFs and THENs while IFTTT focuses on one-to-one APP compositions.

3.3.3 Worker Interface

The worker interface allows crowd workers to select IFs and THENs easily. The interface contains 3 main parts (Figure 3.3). 1) The web-based chat interface allows workers to discuss the problem with the user in real-time. 2) The IF section contains a set of sensors on the user's phone that describe aspects of the user's life and context. For instance, the Google Calendar describes the status of all calendar events of the user, and the Phone Body Sensor describes the physical motions of the smart phone (*e.g.*, phone is moving). Both are considered "sensors" in InstructableCrowd. Workers select appropriate trigger sensors in the IF conditions. 3) The THEN section allows them to select corresponding **effectors**. Effectors are the actions that can be performed on user's smart phone such as push a notification, set an alarm, and send a text message, etc. By selecting IFs and THENs, the worker is able to create rules that trigger certain actions based on specific conditions.

3.3.4 Merge Multiple Crowd-Created Rules by Voting

InstructableCrowd recruits multiple workers for each conversation, therefore, multiple rules are received respectively from each worker in the same conversation. End-users are free to pick any rules submitted by the crowd, or wait for a while to allow the system to collect all crowd-created rules and merge them into a *final rule*. The merging process is as follows. First, any sensors and

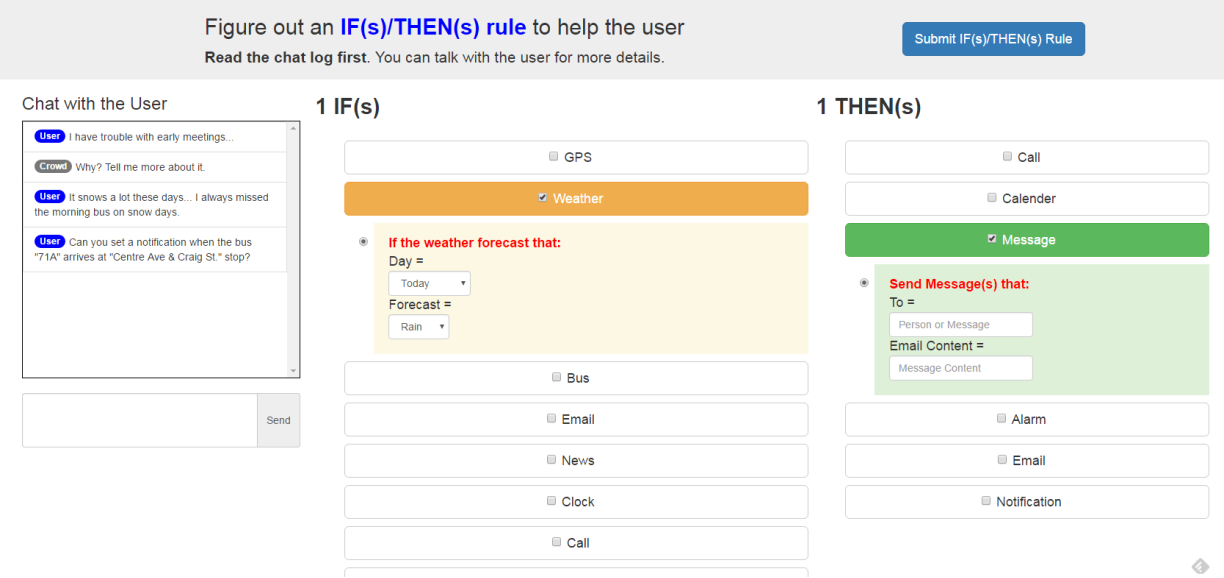

Figure 3.3: Worker interface. A chat interface (left) allows workers to talk to the end user to discuss the problem. The IF section (middle) allows the worker to specify conditions and the THEN (right) allows them to specify effectors.

effectors that are selected by more than a threshold number (e.g., 2) of workers will be included in the final rule. Second, for the sensors/effectors picked in the first step, the system fills each attribute with the value proposed by most workers. If two values were proposed by an identical number of workers, InstructableCrowd selects the value which was proposed in the earlier rule.

3.3.5 Modular Sensors (IF) & Effectors (THEN)

We designed a general JSON (JavaScript Object Notation) schema to represent each sensor and effector. The rules created by the crowd are represented as a combination of sensors and effectors in this JSON format. New sensors and effectors can thus be added easily. For example, the following is the Google Calendar sensor's JSON file representing that *"a calendar event will start at 9:30 tomorrow"*.

```
{
    "proposition": {
        "attribute": "CALENDAR_START_TIME",
        "componentName": "CALENDAR",
        "operator": "OPERATOR_TIME_EQUAL",
        "value": "9:30",
        "referenceAttribute": "CALENDAR_TOMORROW"
    }
}
```

The following is the JSON representation of the effector for *"ring the alarm now"*.

```
{
    "attributes": {
        "ALARM_REFERENCE_TIME": "ALARM_TIME_NOW",
        "ALARM_CONDITION_AT": true,
        "ALARM_RINGTONE_TYPE": 2,
        "ACTION_TYPE": "ALARM"
      },
      "componentName": "ALARM"
}
```

The following is the general JSON representation for a trigger-action rule, which includes a list of sensor (IFs-condition) items and effector (action) items.

```
{
    "actions": [
        {action_1}, ... , {action_n}
    ],
    "conditions": [
        {condition_1}, ..., {condition_n}
    ],
    "ruleID": "rule_1"
}
```

These rules and actions are made possible by a general architecture that we have built to allow systems to access a list of sensors and effectors, and then specify what sensor conditions should lead to what actions. This schema is modular, allowing new sensors and effectors to be added easily. As we go forward, we will continue to expand the set of available sensors and effectors (actions).

3.3.6 Middleware & Rule Validator

We developed a middleware framework that allows the communication and integration between the front-end (the user interface) and the back-end (the server-side processes and the sensors which retrieve information from third party web-services, e.g. weather). The middleware's purpose is manifold: 1) to provide access to a set of well-defined services (e.g., calendar, weather, news, search, audio/video streaming, activity recognition, etc.); 2) to provide access to a set of sensors and effectors (e.g., location, accelerometer, battery, gyroscope, send sms, send emails, etc.); 3) to mediate communication amongst UI components and services through a Message Broker component which validates, transforms, routes and aggregates all kind of messages that are sent over those components; 4) to track and monitor all the user interaction with apps and system components; 5) to validate crowd rules and trigger their corresponding actions (effectors); and 6) to give support to high-level decision-making by the agent.

The Rule Validator module plays a fundamental role during the creation and validation of crowd rules. The Rule Validator receives a rule creation request in the JSON format (this rule

was either created by the crowd or by the user). In general, its goal is to validate the conditions over all the IFs, and if the validation is true, it triggers the relevant THENs in the rule. Different events may have different validation timing periods. For example, while a *Weather* event such as "it snowed during the night" can be checked once every 24 hours, an *Accelerometer* event such as "the phone is falling" must be validated every 100 milliseconds or so. Other events such as *Calendar* events (e.g. meeting before 9am the next day) may be validated immediately after the rule is created and then checked again every hour or so (in case new meetings have been added). Likewise, the actions may have different execution timings as well, for example some of this actions can be executed once the conditions are met and some other must be scheduled for posterior execution, for instance, the action "... show me a notification right now" is executed right after the conditions are met, whereas the action "... send me a reminder tonight at 10:00 pm" is scheduled for later execution.

3.4 User Study

To examine the feasibility of InstructableCrowd, we conducted a set of user experiments with non-programmers and evaluated the rules created with various settings in the system. We specifically recruited non-programmers because one of the benefits of using InstructableCrowd is that complex rules can be created without the need for a programming-like interface.

3.4.1 Scenario Design

With awareness of the scenarios proposed by Huang et al. [52], we designed the following 6 scenarios (S1 to S6), along with the gold-standard sensors and effectors. We further categorized scenarios into three difficulty levels based on the numbers of sensors and effectors the scenario requires. S1 and S2 are **easy** scenarios (1 sensor and 1 effector), S3, S4, and S5 are **intermediate** scenarios (2 sensors and 1 effector), and S6 is **hard** scenario (2 sensors and 2 effectors).

1. **[S1] Sports:** I am very interested in the performance of the "Steelers" and would like to get an immediate notification if there is a news article mentioning them. *(IF: News, THEN: Notification. Easy scenario.)*
2. **[S2] Message:** My mother likes to send me text messages. I work in a restaurant so I cannot reply to her messages very often at work. However, my grandfather was hospitalized last week and my mother is taking care of him now. I do not want to miss any important message about my grandpa. *(IF: Message, THEN: Notification. Easy scenario.)*
3. **[S3] Snow & Meeting:** It snowed last night. I was late for work this morning and missed an important meeting at 9am because I had to take care of all the snow. My boss was quite upset and warned me this can not happen again. *(IF: Weather + Calendar, THEN: Alarm. Intermediate scenario.)*
4. **[S4] Drive & Call:** I just heard that a large percentage of car accidents are caused by talking on the phone while driving. I decided I am not going to answer any phone calls while driving. Therefore, when I am driving, if anyone calls me, I would like to automatically

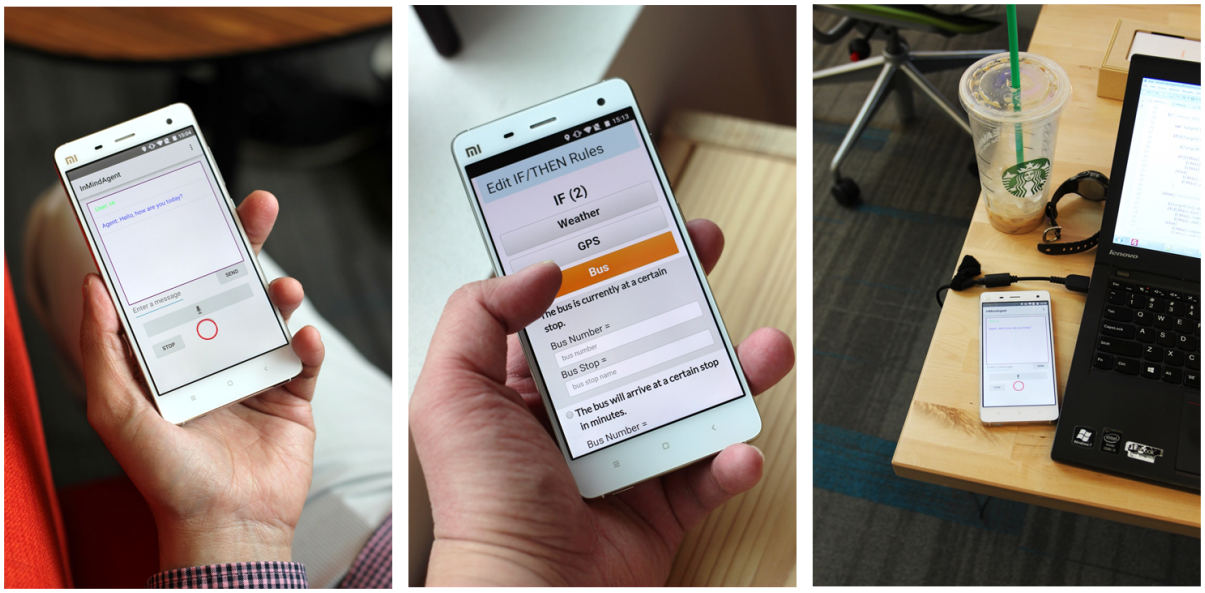

Figure 3.4: User study setting. While waiting for responses from the crowd, participants used their own laptops or mobile devices to simulate the likely context of use in the real world.

reply to him/her with a message saying "Sorry I'm driving." *(IF: Phone Body for Driving + Message, THEN: Message. Intermediate scenario.)*

5. **[S5] Bus:** I usually leave work after 5pm and take Bus "53" home at the "Washington St." stop. However, the "53" buses are not common. I prefer not to wait at the bus stop unless the bus is coming soon. It takes me about 5 minutes to walk from my office to the "Washington St." stop, and it also takes about 5 minutes for Bus "53" to drive from the "Hamilton St." stop to the "Washington St." stop. *(IF: Bus + Clock, THEN: Notification. Intermediate scenario.)*
6. **[S6] Late for Dinner:** My wife Amy does not like me to be late home when we have a big scheduled dinner. So, if I am going to have a big dinner at home in 30 minutes, but I am still far away – say, 30 miles – from home, please send Amy a message saying "I might be home late". Also, give a phone call to "Ben's Flower Shop" and tell them to "Prepare a small surprise bouquet." *(IF: GPS + Calendar, THEN: Message + Call. Hard scenario.)*

In our post-study survey, we asked participants to rate how realistic these scenarios are, in the scale of 1 (very unrealistic) to 7 (very realistic). The mean rating among the twelve participants was 6.25 (SD=0.62).

3.4.2 User Study Setup

We conducted a lab-based user study in which we asked participants to create an IF-THEN rule for each scenario using one of the following conditions:

1. **[Condition 1] InstructableCrowd**: The participant first talks to the crowd via Instructable-Crowd (using text or voice) and waits to receive rules submitted from the crowd workers.

The participant then selects a rule that they prefer and manually edits it to create the final rule.

2. **[Condition 2] User**: The participant uses the rule editor on the phone (as shown in Figure 3.4) to manually create a rule.

In condition (1), 3 data points were recorded: the crowd-created rule that was picked by the participant (which we refer to as **Crowd Only**), the rule edited by the participant (**Crowd + User**), and the rule that was created by merging all ten crowd-created rules (**Crowd Voting**) using the process described earlier (threshold for including a sensor/effector was 2.) We refer to condition (2) as **User Only**.

12 participants (25-36 years old) were recruited locally. The goal of this project is to enable users to compose applications for their own usage, especially for the users who do not know how to program. Therefore, we recruited participants which had very limited experience in programming or none at all. Each participant was requested to create an IF-THEN rule which would resolve each of the 6 scenarios. The participants were asked to solve three scenarios via InstructableCrowd (condition 1), and three other scenarios via the rule editor (condition 2). The scenarios were controlled for the order in which they appear as well as the condition they were associated with. Participants were instructed to follow the scenarios as close as possible, but were allowed to propose minor changes during the conversation, *e.g.*, change "send me notification" to "send me an email." Participants were also free to use their own laptop or mobile devices when they waited for the response from the crowd, because we believe this setting is more realistic for users who try to converse via instant messaging on mobile devices. A post-study questionnaire was used to collect subjective feedback from the participants.

For each conversational session, InstructableCrowd posted a HIT (Human Intelligence Task) with 10 assignments to MTurk. The price of each assignment is $0.50 USD. During a conversational session, multiple workers could talk to the participant via their interface and submit rules respectively. 156 unique workers on MTurk participated in our experiments. All sessions, chats, and rules were recorded in a database with timestamps. We also timed how long the participant took to create each rule by using the rule editor.

In the user study, crowd workers and end users chose from 10 sensors: Email, Bus, Message, GPS, Weather, Call, Clock, Calender, News, and Phone Body (for driving and phone falling); they have 6 effectors to choose from: Message, Email, Alarm, Call, Notification, and Calendar (for adding an event). Each sensor had an average of 1.6 attributes to fill in (SD=1.1), and each effector had an average of 2.5 attributes (SD=1.4).

3.4.3 Quantitative Evaluation

Composing an IF-THEN rule contains two sub-tasks: **sensor/effector selection**, and **attribute filling**. For instance, to effectively know that you have an early meeting tomorrow, the "Calendar" sensor firstly needs to be selected, and then its "Start Time" attribute need to be filled with "Before 8am." We evaluated the performances of these two tasks in all of our recorded rules.

	IF			THEN			Avg
	P	R	F1	P	R	F1	F1
User Only	0.94	0.85	0.89	0.98	0.99	0.98	0.94
Crowd Only	0.94	0.77	0.85	0.97	0.90	0.94	0.89
Crowd+User	0.94	0.83	0.89	1.00	0.94	0.97	0.93
Crowd Voting	0.92	0.89	0.91	0.95	0.96	0.96	0.93

Table 3.1: Sensor/Effector selection overall performance.

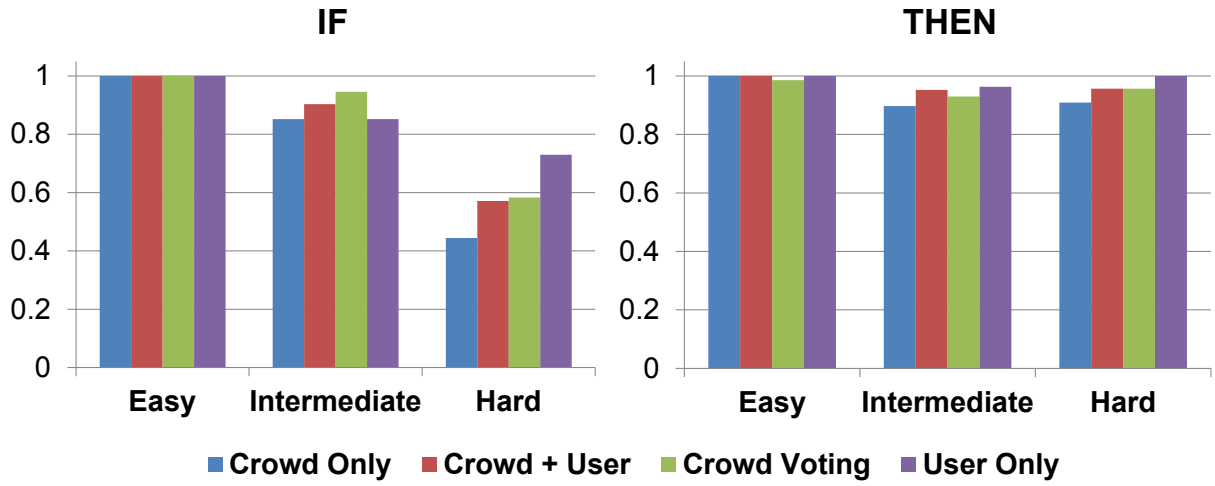

Figure 3.5: Average F1-score of sensor/effector selection in easy, intermediate, hard scenarios. While the THEN parts were not influenced much, the F1-scores in IF parts' decreased as the scenarios got more complex.

Sensor/effector Selection

The evaluation process was as follows: First, we expanded the set of correct answers to include rules created by the user which seemed correct, but not exactly what we anticipated. For instance, in S3, some participants decided to send emails to the boss at work instead of setting up an earlier alarm; in S2, one participant decided to reply to his/her mom with a message instead of pushing a notification. We went through all the submitted rules and added the effective solutions that we did not think of initially. Second, we allowed extra or alternative effectors if appropriate. For instance, some participants thought that pushing a notification is not enough and decided to sent an email or to set an alarm. We considered these alternative rules are still effective. Finally, a piece of software was created to perform an automated evaluation on all recorded rules.

Selecting a set of correct sensors/effectors from a pool of candidate is a retrieval task. We therefore use the precision, recall, and F1-score to evaluation this sub-task. These values are calculated as follows.

$$\text{Precision} = \frac{|\{\text{Selected Sensor}\} \cap \{\text{Gold-Standard Sensor}\}|}{|\{\text{Selected Sensor}\}|}$$

	IF	THEN	Avg
User Only	98.3%	95.0%	**96.7%**
Crowd Only	81.4%	90.0%	85.7%
Crowd + User	89.2%	93.3%	**91.3%**
Crowd Voting	86.4%	95.0%	**90.7%**

Table 3.2: Attribute filling overall performance.

$$\text{Recall} = \frac{|\{\text{Selected Sensor}\} \cap \{\text{Gold-Standard Sensor}\}|}{|\{\text{Gold-Standard Sensor}\}|}$$

The F1-score is the harmonic mean of precision and recall. When a rule is partially correct, we selected the gold-standard rule which results in the highest F1-score to report the numbers in this chapter. The overall evaluation results are shown in Table 3.1. Both "Crowd+User" and "Crowd Voting" settings achieved comparable performances to that of the "Crowd Only" setting is both IF and THEN parts. Selecting correct sensors in IF is harder than selecting correct effectors in THEN, which is expected due to the tolerant nature of our evaluation setup for THEN. We observe that "Crowd Voting" resulted in a higher average recall, which suggested that a group of crowd workers is, collectively, less likely to forget picking some sensors than an individual user. We also notice that participants actually corrected errors in the crowd-created rules, as both the average precisions and recalls are higher in "Crowd+User" than "Crowd Only". For instance, in the "Late for Dinner" scenario (S6), one common mistake was that crowd selected only one of Calender or GPS sensors, instead of both. Two different participants fixed this error by adding back the missing sensor. Another similar example occurred in the "Bus" scenario (S5), where the crowd sometimes missed the "Clock" sensor which can indicate the current time is after 5pm. One participant fixed this by adding the Clock sensor back to the IF.

We also evaluated the performance based on the scenarios' difficulty level. The dynamics of F1-scores are shown in Table 3.5. While the THEN parts were not influenced much, the F1-scores in IF parts' decreased as the scenarios got more complex. "Crowd Voting" performed similarly or slightly better than "User Only" in easy and intermediate rules, but worse in hard rules. This results also indicate the number of sensors and effectors influences the difficulty level of composing the rule, while other factors such as abstraction level and type of sensors/effectors also reportedly play important roles [106].

Attribute Filling

The evaluation process of attribute filling is similar to that of sensor/effector selection. Any value for an attribute which seemed appropriate was considered to be correct. For instance, the content of the sent messages or emails could vary, and we manually labeled the effectiveness of each "content" attribute in effectors; the *weather-forecast-day* attribute in the Weather sensor of S3 could be set to either "Today" or "Tomorrow", however, it would only be judged as correct if the Alarm's *alarm-send-day* attribute was set to the same value. Software was created to evaluate these attributes automatically.

For a given sensor/effector S that is correctly selected, we calculate the accuracy of its at-

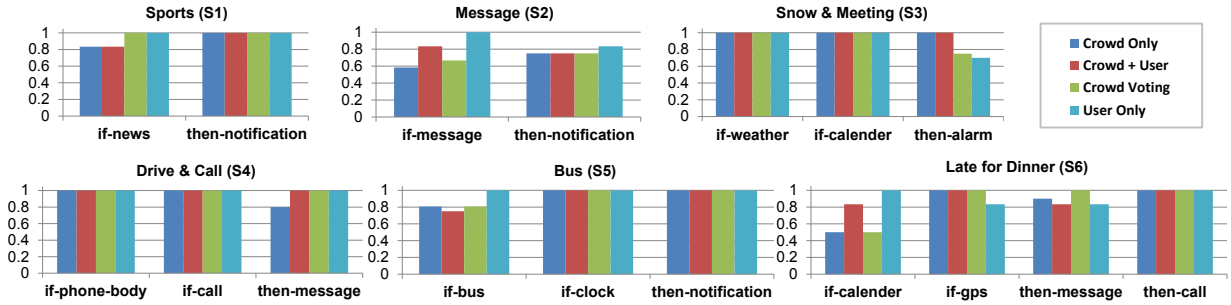

Figure 3.6: Average accuracy of attribute filling of correctly-selected sensors/effectors. "Crowd Voting" performed similarly as "User Only" in most cases. We analyzed S2, S5, and S6 and found that crowd errors are mainly caused by communication gap and misunderstanding of attributes.

tribute values as follows:

$$\text{Accuracy} = \frac{\text{Number of Attributes in } S \text{ with correct values}}{\text{Number of Attributes in } S}$$

The overall evaluation results of attribute filling are shown in Table 3.2. While the "Crowd Voting" setting achieved the same average accuracy as that of the "User Only" in the THEN part, its average accuracy is lower than "User Only" in the IF part. To understand the sources of this performance gap, we analyzed the average accuracy of attributes in each sensor/effector of each scenario, as shown in Figure 3.6. We observed the sensors (IF) where "Crowd Voting" resulted in a lower accuracy than that of "User Only" (i.e., the Message sensor in S2, the Bus sensor in S5, and the Calendar sensor in S6) and identified two sources of crowd workers' errors: **communication gap** and **misunderstanding of attributes**. One source of the errors was the communication gap between the end-user and crowd workers. Namely, the user falsely expressed or missed some information when talking to the crowd. For instance, in S2, one participant falsely said "dad" often sent him/her messages (instead of "mom"), and the crowd therefore filled "dad" in the *message-receive-sent-by* attribute; in S5, one participant did not mention to the crowd that it usually takes 5 minutes to walk to the bus stop, so the crowd arbitrarily filled the *if-bus-future-minute* attribute with 2 minutes. Another source of the errors is the misunderstanding of attributes. In S6, we found that some crowd workers confused the *relative time* with *absolute time* attributes of the Calendar sensor, and arbitrarily made up an absolute time (e.g. "5pm") instead of assigning a relative time (e.g., "in 30 minutes.") In addition, both users and crowd workers have **typos** in their attributes. For instance, a worker misspell "Steelers" as "Stelers" in S1, and another worker put "19:00" as the "start time of the meeting" in S3, which we believe it tended to be "07:00."

User Active Time

We also analyzed the user active time, i.e., the time that users spent on interacting with the system. Even though it is expected that InstructableCrowd requires more time since the user needs to talk with the crowd, it is still important to understand how much time it takes a user to create

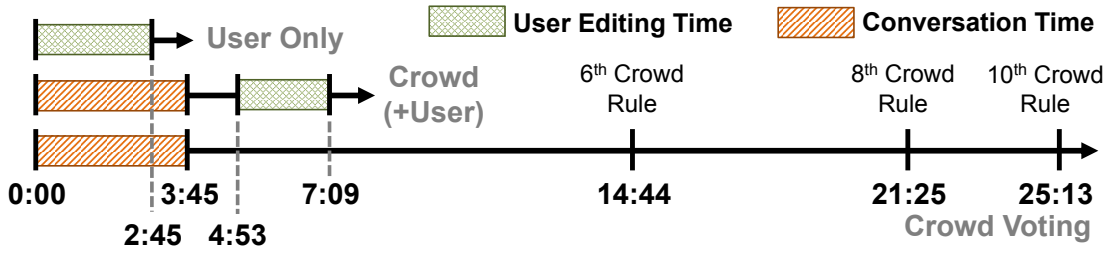

Figure 3.7: The complete timeline of InstructableCrowd. Note that a user's cognitive load when editing a rule and when talking with conversational partners are different. When having a conversation with InstructableCrowd, users are free other things, such as browse the Internet at the same time.

a rule. In our study, participants spent an average of 2 minutes and 45 seconds (SD=1:23) to create a rule from scratch using the rule editor ("User Only"); When using InstructableCrowd, participants spent an average of 3 minutes and 45 seconds (SD=2:01) to converse with the crowd, and then the system took about one minute after the conversation to create a rule that the participants were willing to pick ("Crowd Only".) If the participant decided to edit the crowd-created rules he/she just picked, it took about 2 minutes for the participants to further edit the rule ("Crowd+User"). It took approximately 20 minutes for InstructableCrowd to receive the rules from all 10 workers and compose the final rule ("Crowd Voting".) The complete timeline is shown in Figure 3.7. On average, "Crowd Voting" setting took a user one more minutes than that of "User Only." It is also noteworthy that user's cognitive load when editing a rule and when talking with conversational partners are very different. When having a conversation with InstructableCrowd, users are free to browse the Internet, chat with other people, or even watch a video at the same time.

3.4.4 Qualitative Results

Subjective Feedback from Participants

We collected participants' subjective feedback immediately after they finished the lab-based study. We asked participants to rate the difficulty of using InstructableCrowd ("Crowd+User" setting) versus using the rule editor themselves ("User Only"), on a scale of 1 (very easy) to 7 (very hard). The average difficulty level of using InstructableCrowd was 3.5 (SD=1.4) and that of using the rule editor was 3.0 (SD=1.4).

We asked participants what their preferred method was, and grouped them into two groups according to their preference. As shown in Table 3.3, compared to the participants who preferred the rule editor, the participants who preferred InstructableCrowd had a much higher difficulty rating for using the rule editor. We also found that the correlation coefficient between a user's difficulty rating on the rule editor and preferring InstructableCrowd (prefer=1, not prefer=0) is 0.65, which is a strong correlation. However, a similar relation was not found between user's difficulty rating on InstructableCrowd and preferring the rule editor (correlation coefficient = 0.06). The participants who preferred InstructableCrowd also took longer to manually compose

		Participants Grouped by Preference	
		Prefer InstructableCrowd	**Prefer Rule Editor**
#Particiapnt		4	7
Avg. Difficulty Rating [avg. (stdev.)]	**InstructableCrowd (Crowd+User)**	**3.25** (1.50)	3.57 (1.62)
	Rule Editor (User Only)	4.25 (1.71)	**2.29** (0.76)
Avg. Time to Create a Rule Manually (User Only) (mm:ss) [avg. (stdev.)]		03:15 (01:20)	02:30 (0:45)

Table 3.3: The average difficulty ratings and rule composing time of participants that prefer InstructableCrowd v.s. rule editor. The participants who preferred InstructableCrowd had a higher difficulty rating for using the rule editor, and also took longer to manually compose a rule.

an IF-THEN rule on average. This result suggests that InstructableCrowd enables the users who have difficulty creating complex rules on mobile phone an alternative easier method to compose applications. One participant had no preference between using the rule editor and using InstructableCrowd. This participant gave the following feedback: "it depends on different situations. for example: i would like to create rules through conversations with the system while driving." Although we recruited users without programming experience, they were somewhat tech-savvy; these results suggest we might see an even stronger effect if InstructableCrowd was used by people even less comfortable with using their smartphone.

We also asked why participants prefer InstructableCrowd. Interestingly, 3 out of these 4 participants said that InstructableCrowd is "faster" or "quick", while they actually spent longer time to create a rule when comparing to the time it took them when using the rule editor. This could be because the difficult parts of creating rules is outsourced to the crowd when using InstructableCrowd, and the participants do not need to develop a rule from scratch. Some participants also stated that InstructableCrowd is more flexible since it allows the user to choose from a set of rules which is sent from multiple crowd workers. One participant who chose to use speech input said it is "faster" because she "doesn't like to type."

In the post-study questionnaire, we also asked participants when they would prefer to use InstructableCrowd, and when they would use the rule editor. In their responses we found that people tend to create rules via conversation when 1) the rule would be too complex, and 2) they are busy or having a tight schedule. 6 out of 12 participants said they would choose InstructableCrowd when the rule they want to create has too many conditions or complex logic, e.g., "...I cannot figure out a proper logic to state 'If' and 'Then', I may rely the conversation to ask help from a server."; 3 out of 12 participants said they would choose InstructableCrowd when they are busy, e.g., "I would use it when I am busy."

Collecting Information via Conversations

We analyzed the conversations between the participants and the crowd. The three main types of follow-up responses from the crowd are 1) **information inquiry**, 2) **confirmation**, and 3) **clarification question**. Most of the conversation between users and the crowd is for collecting

information. For instance, in the following session, crowd workers ask for the information which is required in order to complete the rule they are creating:

crowd **Hi, what can I help you with?**

user it was snow last night and I was late for work and missed an important meeting this morning.

crowd **Would you like a weather alert?**

crowd **What would you like us to do?**

user I missed an important meeting at 9am.

crowd **What time do you usually wake up?**

user 7am

crowd **Would you like to wake up earlier if it snows? Is 1 extra hour enough?**

user sure.

In the following excerpt, a crowd worker was trying to figure out the time of the dinner:

user if i have a big dinner on my calendar and i am going to be late (if i am still far away in 30 minutes), send my wife a message saying :" i might be home late") and call the florist to prepare a small bouquet.

crowd **What time might this dinner start?**

user it depends on my calendar.

Crowd workers sometimes confirmed with the users information which was conveyed previously. For example:

crowd **hello?**

user I leave work after 5pm and take Bus 53 home at the Washington street

user I don't wanna wait for the bus for too long unless the bus is coming soon

crowd **is after 5pm**

user yes

In the following session, the crowd worker suggested to use an alarm or a notification instead of a phone call, though the user did not accept this suggestion.

crowd **Hello, how can I help you??**

user please call me if the text from my mom containing "grandpa" or "grandfather".

crowd **Do you want to send them a message asking to call you, or do you want to receive an alarm or notification?**

user maybe just call me. thanks!

Alternative Solutions for the Same Scenario

Multiple effective but different rules were sometimes created for the same scenario. For example, in the "Bus" scenario (S5), the notification can either be fired when "the Bus 53 will arrive at

Washington St. in 5 miniutes" or when "Bus 53 is arriving at Hamilton St. stop now.". Both rules are effective, and both rules occurred in our study. More than one participant tried to add extra effectors such as an alarm in the "Message" scenario (S2), since they believed that missing a message about the hospitalized grandfather can be quite serious. Furthermore, some ambiguities which the crowd workers face may come from the instructions which they receive from the users. For instance, in the following conversational session, the word "reply" does not necessarily imply "sending a message" (although it might be the most common solution). Therefore, "sending an email" is also acceptable. The alternatives that the crowd came up with demonstrates their potential to be creative and think of solutions that the user might not have.

user hi

user I know car accidents might happen if i talk on the phone while driving. so I would like to reply "sorry I am driving" to anyone calling me when I'm driving.

crowd **ok i will do so now**

3.5 Discussion

In this chapter we introduced *InstructableCrowd*, a system that lets end users create IF-THEN rules for their smartphones via conversation with the crowd. The rules created by the crowd were nearly as good as those created by the end user, and the combination of having the end user edit the rules created by the crowd exceeded even the end user's performance. These results suggest that InstructableCrowd may be useful in scenarios in which the end user does not want to create the rules by themselves or is unable to do so because of situational constraints (*e.g.*, they are driving). Users who may not know how their smartphone might be able to help them might find having a conversation with the crowd useful.

3.5.1 Design Guides

This task is difficult because IF-THEN rules (like programming in general) have little tolerance for mistakes. If we breakdown an IF-THEN rule to a composition of sensors and effectors with attribute values, humans are reasonably good at composing sensors/effectors and filling their attributes, per experiments. However, when we add up all the work, any mistakes will make the resulting IF-THEN rule not effective. In response to this situation, a natural direction to explore is to enforce a more strict validation for human input within the process of rule creation. However, for both users and the crowd, a strict input validation on the interface would increase the time it takes to create a rule, frustrate users more easily, and increase the engineering efforts that are required to add a new sensor or effector, which often come with arbitrary constraints, to the system. IFTTT, as a successful rule-creation product, avoids multiple sensors and effectors, and uses a user-friendly workflow to balance user's possible frustration. Our project suggests using conversation and iterative editing to perform a robust rule creation.

3.5.2 Redundant Rules Created by Users

If a user receives multiple rules during the same conversational session, it may be fair to assume that these rules are redundant and allow the user to pick only a single rule from this set. However, if the user creates many rules with InstructableCrowd, the user may even forget that he has already created a rule and attempt to create the same rule again. Furthermore, the user may once create a very specific rule (e.g., IF I have a meeting at 9am, THEN notify me the night before), and later try to generalize it (e.g., IF I have a meeting at 10am or earlier, THEN notify me the night before). If the middleware were to execute these rules regardless of the other rules, the same action may be executed more than once which is not likely to be the user's intent. Further research is required into identifying these cases and alerting the user.

3.5.3 User Privacy

In our study, one participant specifically asked about the privacy issues in our project. In the current prototype, a limited view of a user's personal information (*e.g.*, contact list created for the purpose of the study) was exposed to crowd workers. In our future deployment, we may use aliases that are either automatically assigned or even created by the user himself to prevent the true names or information leak to crowd workers. For instance, instead of describing the actual address, the user will be able to name the alias such as "Home" or "Office" when they talk to the crowd. Aliases can also be used to protect the information of people or time, e.g., describing "Wife" instead of "Amy", or use "Birthday" instead of the actual date. However, the use of aliases can not completely prevent the user from providing personal information in conversation. Privacy is a well-known issue in the field of crowdsourcing, since the data is processed by human workers, which many works were proposed to resolve [73]. A future direction is to further explore privacy issues that conversational interfaces may raise.

3.5.4 Limitations

One natural limitation of the sensors and the effectors in InstructableCrowd is that they need to be understandable by workers in the crowd. For example, we expect it would be very challenging for non-expert crowd workers to describe the raw values of an accelerometer sensor that correspond to certain movement of the phone (*e.g.*, falling, driving, walking). Future systems may find value in explicitly recruiting people with programming expertise to their crowds in order to provide abstractions over raw sensor values that could be shared and reused by others. Using existent sensors to express high-level semantics (*e.g.*, sleeping) requires specialized knowledge that most crowd workers likely do not have; IF-THEN rules have low tolerances for mistakes while quality control is still an essential challenge in crowdsourcing. It may be useful to explore ways for the rules that are created to form a part of a probabilistic suggestion system, *i.e.*, instead of automatically conducting an action that may or may not be correct, ask the user if they would like to do it.

3.6 Conclusion

This chapter introduced InstructableCrowd, a system that allows end users to create complex IF-THEN rules via voice in collaboration with the crowd. These rules connect to the sensors and the effectors on the user's phone where the sensors serve as triggers and the effectors as actions. We have created a generic JSON representation for the sensors and the effectors in a smartphone, and the ability to define rules. We created a middleware that allows access to these sensors and effectors and applies these rules. We have built support for crowd workers to have a conversation with the users and allow them to suggest rules for the users. A user study shows that non-programmers can effectively create rules via voice, and suggests that collaboration between the user and the crowd while creating IF-THEN rules could be a fruitful area for future research. InstructableCrowd represents a new approach in which end users work with remote crowd workers to bring about powerful functionality despite the constraints of mobile and wearable devices.

Part II

Deploying Chorus to Gather Data

Chapter 4

Challenges in Deploying an On-Demand Crowd-Powered Conversational Agent

4.1 Introduction

Over the past few years, crowd-powered systems have been developed for various tasks, from document editing [7] and behavioral video coding [70], to speech recognition [65], question answering [97], and conversational assistance [69]. Despite the promise of these systems, few have been deployed to real users over time. One reason is likely that deploying a complex crowd-powered system is much more difficult than getting one to work long enough for a study. In this work, we discuss the challenges we have had in deploying Chorus[1], a crowd-powered conversational assistant.

We believe that conversational assistance is one of the most suitable domains to explore. Over the past few years, conversational assistants, such as Apple's Siri, Microsoft's Cortana, Amazon's Echo, Google's Now, and a growing number of new services and start-ups, have quickly become a frequently-used part of people's lives. However, due to the lack of fully automated methods for handling the complexity of natural language and user intent, these services are largely limited to answering a small set of common queries involving topics like weather forecasts, driving directions, finding restaurants, and similar requests. Crowdsourcing has previously been proposed as a solution which could allow such services to cope with more general natural language requests [54, 67, 69]. Deploying crowd-powered systems has proven to be a formidable challenge due to the complexity of reliably and effectively organizing crowds without expert oversight.

In this chapter, we describe the real-world deployment of a crowd-powered conversational agent capable of providing users with relevant responses instead of merely search results[57]. While prior work has shown that crowd-powered conversational systems were possible to create, and have been shown to be effective in lab settings [54, 56, 69], we detail the challenges with deploying such a system on the web in even a small (open) release. Challenges that we identified included determining when to terminate a conversation; dealing with malicious workers when large crowds were not available to filter input; and protecting workers from abusive content introduced by end users.

[1]Chorus Website: http://TalkingToTheCrowd.org/

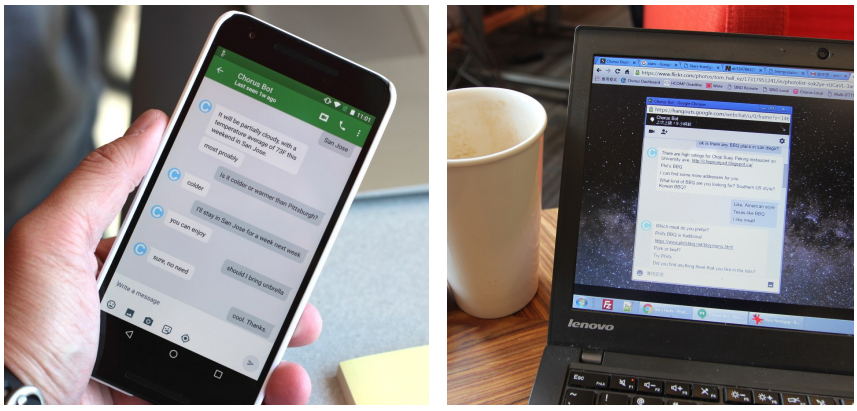

Figure 4.1: Chorus is a crowd-powered conversational assistant deployed via Google Hangouts, which lets users access it from their computers, phones and smartwatches.

We also found that, contrary to well-known results in the crowdsourcing literature, recruiting workers in real time is challenging, due to both cost and workers preference. Our system also faced challenges with a number of issues that went beyond what can be addressed using worker consensus alone, such as how to continue a conversation reliably with a single collective identity.

4.2 Related Work

Our work on deploying a crowd-powered conversational agent is related to prior work in crowdsourcing, as well as automated conversational systems in real-world use.

4.2.1 VizWiz

One of the few deployed (nearly) real-time crowd-powered systems, VizWiz [13] allowed blind and low vision users to ask visual questions in natural language when needed. VizWiz used crowd workers to reply to visual and audio content. To date, VizWiz has helped answer over 100,000 questions for thousands of blind people[2]. VizWiz is a rare example of a crowd-powered system that has been brought out of the lab. For example, in order to make the system cost effective, latency was higher and fewer redundant answers were solicited per query. However, VizWiz relied less on redundancy in worker responses, and more on allowing end users to assess if the response was plausible given the setting. VizWiz tasks consist of individual, self-contained units of work, rather than a continuous task.

View [68], which was built upon the ideas introduced in VizWiz, used a continuous interaction between multiple crowd workers and an end user based on video. View, which aggregates workers answers, has showed that multiple workers answer more quickly, accurately, and completely than individuals. Unfortunately, to date, View has not been deployed in the wild. This is in part because of the cost of scaling this type of continuous interaction, as well as ensuring

[2]VizWiz: http://www.vizwiz.org

on-going reliability with minimal ability to automatically monitor interactions. Be My Eyes[3] is a deployed application with a similar goal: answer visual questions asked by blind users by streaming video. However, while they draw from a crowd of remote people to answer questions, the interaction is one-on-one, which assumes reliable helpers are available. Be My Eyes relies on volunteers rather than paid crowd workers. However, in more general settings, relying on volunteers is not practical.

4.2.2 Conversational Systems

Artificial Intelligence (AI) and Natural Language Processing (NLP) research has long explored how automated dialog systems could understand human language [40], hold conversations [3, 17, 90], and serve as a personal assistant [24]. Personal intelligent agents are also available on most smartphones. Google Now is known for spontaneously understanding and predicting user's life pattern, and automatically pushing notifications. Conversational agents such as Apple's Siri also demonstrated their capability of understanding speech queries and helping with users' requests.

However, all of these intelligent agents are limited in their ability to understand their users. In response, crowd-powered intelligent agents like Chorus [69] use crowdsourcing to make on-going conversational interaction with an intelligent "assistant." Alternatively, conversational assistants powered by trained human operators such as Magic[4] and Facebook M have also emerged in recent years.

4.3 System Overview

The deployed Chorus consists of two major components: 1) the crowd component based on Lasecki *et al.*'s proposal that utilizes a group of crowd workers to understand the user's message and generate responses accordingly [69], and 2) the bot that bridges the crowd component and Google Hangouts' clients. An overview of Chorus is shown in Figure 4.2. When a user initiates a conversation, a group of crowd workers is recruited on MTurk (Amazon Mechanical Turk) and directed to a worker interface allowing them to collectively converse with the user. Chorus' goal is to allow users to talk with it naturally (via Google Hangouts) without being aware of the boundaries that would underlay an automated conversational assistant. In this section, we will describe each of the components in Chorus.

4.3.1 Worker Interface

Almost all core functions of the crowd component have a corresponding visible part on the worker interface (as shown in Figure 4.2). We will walk through each part of the interface and explain the underlying functionality. Visually, the interface contains two main parts: the *chat box* in the middle, and the *fact board* that keeps important facts on the side.

[3]Be My Eyes: http://www.bemyeyes.org/
[4]Magic: http://getmagicnow.com/

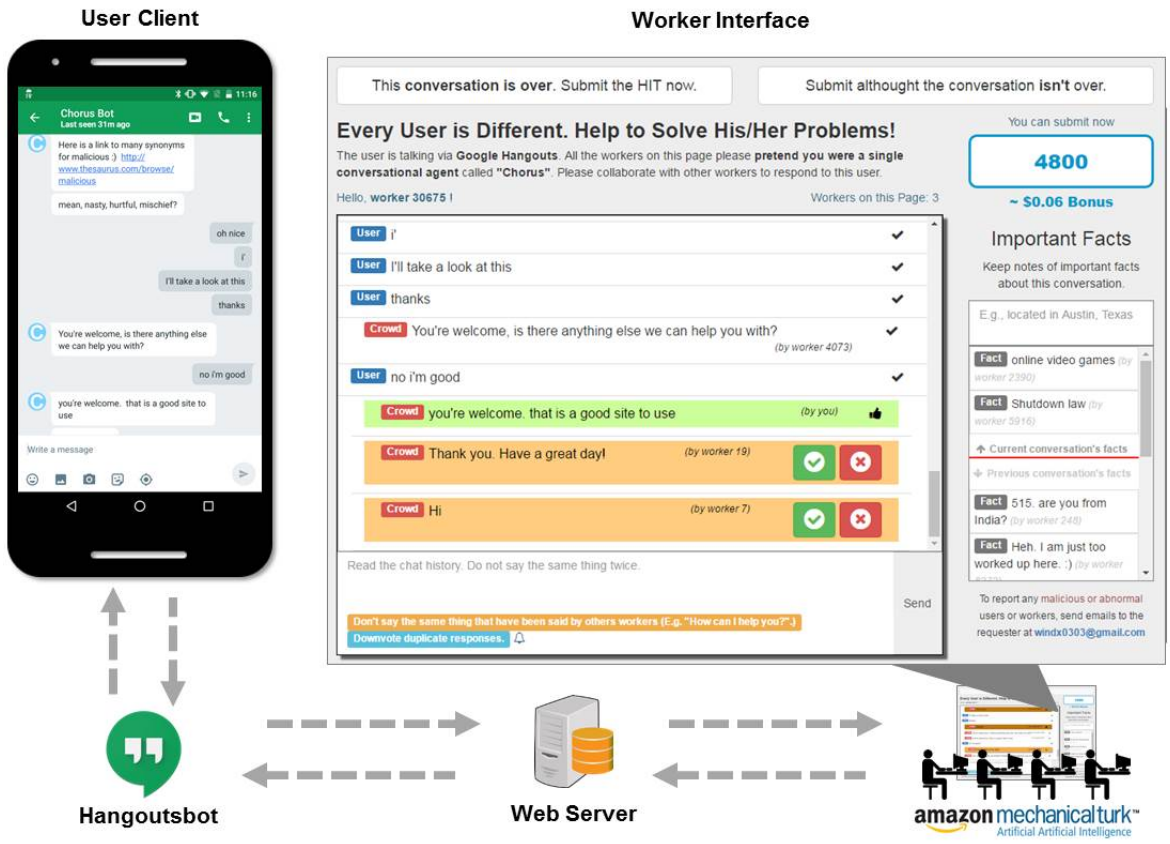

Figure 4.2: The Chorus UI is formed of existing Google Hangouts clients for desktop, mobile or smartwatch. Users can converse with the agent via Google Hangouts on mobile or desktop clients. Workers converse with the user via the web interface and vote on the messages suggested by other workers. Important facts can be listed so that they will be available to future workers.

Proposing & Voting on Responses: Similar to Lasecki *et al.*'s proposal [69], Chorus uses a voting mechanism among workers to select good responses. In the chat box, workers are shown with all messages sent by the user and other workers, which are sorted by their posting time (the newest on the bottom). Workers can propose a new message, or *upvote* or *downvote* each response that was proposed by other workers. As shown in Figure 4.2, workers can not only click on the check mark (✔) to upvote the good responses, but also click on the cross mark (✖) to downvote the bad responses. Messages are color-coded from workers' perspective: orange for those proposed by other workers, the messages that receive sufficient agreement will be "accepted" (and turn white), the upvoted messages turn to light green color, and the downvote messages turn to gray color.

Upon calculation the voting results, we empirically assigned negative weights to donwvotes (-0.5) while upvotes have positive weights (1.0). Chorus accepts a responses when its $\#upvote \times 1.0 - \#donwvote \times 0.5 \geqslant \#active\_workers \times 40\%$, and then sends the ID of the accepted message to the Google Hangout bot to be displayed to the user.

Instant Expiration Upon Accepting Responses: We also developed *instant expiration* feature on the worker interface. When Chorus accepts a response, it automatically expires all other response candidates that have not been accepted, and more importantly, *vanishes* them from the chatbox on worker interface. Instant expiration enforces that all viable response candidates on the interface were proposed based on the latest context. A natural consequence of this feature is that workers' responses can be expired and removed very fast, which is especially problematic when a worker spent a lot time and effort to search and compose a high-quality response, but get removed instantly. To compensate this loss, we added a "proposed chat history" box, which automatically records the latest 5 response that the current worker proposed, on the left side of worker interface. If a response vanished too fast and still fits in the ongoing conversation, the worker can simply copy his/her previously proposed response and send it again.

Maintaining Context: To provide context, chat logs from previous conversations with the same user are shown to workers. Beside the chat window, workers can also see a "fact board", which helps keep track of information that is important to the current conversation. The fact board allows newcomers to a conversation to catch up on critical facts, such as location of the user, quickly. The items in the fact board are sorted by their posted time, with the newest on top. We did not enforce a voting or rating mechanism to allow workers to rank facts because we did not expect conversations to last long enough to warrant the added complexity. In our study, an average session lasted about 11 minutes. Based on worker feedback, we added a separator (red line + text in Figure 4.2) between information from the current and past sessions for both the chat window and fact board.

Rewarding Worker Effort: To help incentivize workers, we applied a points system to reward each worker's contribution. The reward points are updated in the score box on the right top corner of the interface in real-time. All actions (*i.e.*, proposing a message, voting on a message, a proposed message getting accepted, and proposing a fact) have a corresponding point value. Reward points are later converted to bonus pay for workers. We intentionally add "waiting" as an action that earns points in order to encourage workers to stay on a conversation and wait for the user's responses.

Ending a Conversational Session: The crowd worker are also in charge of identifying the end of a conversation. We enforce a minimal amount of interaction required for a worker to submit a HIT (Human Intelligence Task), measured by reward points. A sufficient number of reward points can be earned by responding to user's messages. If the user goes idle, the workers can still earn reward points just for remaining available. Once two workers submit their HITs via "This conversation is over" button (in Figure 4.2), the system will close the session. All remaining workers' HITs with sufficient reward points will be automatically submitted, and the workers without enough points will be sent back to the waiting page with their earned points. This design encourages workers to stay to the end of a conversation.

To prevent workers who join already-idle conversations from needing to wait until they have enough reward points, a "three-way handshake" check is done to see if: 1) The user sends at least one message, 2) the crowd responds with at least one message, and 3) the user responds

again. If this three-way handshake occurs, the session timeout is set to 15 minutes. However, if the conditions for the three-way handshake are not met, the session timeout is set to 45 minutes. Regardless of how a session ends, if the user sends another message, Chorus will start a new session.

Participatory Design with Workers: Similar to prior interactive crowd-powered systems, Chorus uses animation to connect worker actions to the points they earn, and plays an auditory beep when a new message arrives. We found that workers wanted to report malicious workers and problematic conversations to us quickly, and thus asked for a means of specifying who the workers were, and which session the issue occurred in. In response, we added our email address, and made available a session ID, indexed chat messages, and indexed recorded facts that workers could refer to in an email to us. After this update, we received more reports from workers and identified problematic behaviors more quickly.

4.3.2 Integrating with Google Hangouts

Another core piece of Chorus is a bot that bridges our crowd interface and the Google Hangouts client. We used a third-party framework called Hangoutsbot[5]. This bot connects to Google Hangouts' server and the Chorus web server. Hangoutsbot acts as an intermediary, receiving messages sent by the user and forwarding them to the crowd, while also sending accepted messages from the crowd to end users.

Starting a Conversational Session: In Chorus, the user always initiates a conversational session. Once a user sends a message, the bot records it in the database (which can be accessed by the crowd component later), and then checks if the user currently has an active conversational session. If not, the bot opens a new session and start recruiting workers.

Recruiting Workers: When a new session is created, Chorus posts 1 HIT with 10 assignments to MTurk to recruit crowd workers. We did not apply other techniques to increase the recruiting speed. Although we did not implement a full-duty retainer as suggested in [8], a light-weight retainer design was still applied. If a conversation finishes early, all of its remaining assignments that have not been taken by any workers automatically turn into a 30-minute retainer. We also required each new worker to pass an interactive tutorial before entering the task or the retainer. More details will be discussed in a later section.

Auto-Reply: We used Hangoutsbot's auto-reply function to respond automatically in two occasions: First, when new users send their very first messages to Chorus, the system automatically replies with a welcome message. Second, at the beginning of each conversational session, the bot sends a message back to the user to mention that the crowd might not respond instantly. To make the system sound more natural, we created a small set of messages that Chorus randomly chooses from – for instance: "What can I help you with? I'll be able to chat in a few minutes."

[5]Hangoutsbot: https://github.com/hangoutsbot/hangoutsbot

User	**How many suitcases can I take on a flight from the US to Israel?**
Chorus	Let me check.
	Can I ask you from where are you planning to board the flight?
User	**Pittsburgh**
Chorus	with which company are you flying?
	and which air services are you using?
User	**Delta airlines**
Chorus	You may bring one carry-on item onboard the aircraft, plus one personal item
	If you are: 1) active duty U.S. military personnel on orders to or from duty stations and dependents traveling with them; or 2) active U.S. military dependents traveling on relocation orders, you may check the following at no charge:
	Up to four bags in Main Cabin on Delta and in all cabins on Delta Connection flights. Up to five bags in Delta One™, First and Business Class on Delta aircraft only.
	Approved Personal Items: 1 purse, briefcase, camera bag or diaper bag, or 1 laptop computer (computers cannot be checked), or 1 item of a similar or smaller size to those listed above
User	**What about checked in baggage?**
	I should be able to bring at least one suitcase, no?
	I'm not any of those
Chorus	one personal item mentioned above is allowed.
	it could be 1 purse, briefcase, camera bag or diaper bag or 1 laptop computer (computers cannot be checked) or 1 item of a similar or smaller size to those listed above.
	or all other international flights
	You may check up to two bags that meet our size & weight restrictions at no extra charge
User	**Oh, so I get two bags for free to Israel?**
Chorus	*< URL of Delta's Web Page of Baggage Policy >*
	Yes
	you can check all th [sic] details here
	Southwest allows two (2) checked pieces of baggage per ticketed Customer
	Yes.
	hi
	how can I help u?
User	**In the link it says that to Israel, the second bag is $100. I'm confused....**
Chorus	yes
	They charge you for extra baggage.
	are you flighing [sic] on Delta One™, First, Business class?
	here is the detail - Size & Weight Restrictions. To avoid extra charges for oversize or overweight baggage, your checked bag must: weigh 50 pounds (23 kg) or less. not exceed 62 inches (157 cm) when you total length + width + height.
User	**No, just a regular flight (economy)**
Chorus	Then yes, you will have one bag for free
	I am afraid you'll have to pay extra.
	for the other bag.
	AirTran Airways
	First Checked Bag: $20 each way for all economy-class reservations.
	Second Checked Bag: $25 each way for all economy-class reservations.
	and the second is $100
User	**Too bad. OK thanks!**

Figure 4.3: A long and sophisticated conversation Chorus had with a user about what suitcases she could bring on a flight.

4.4 Field Deployment Study

The current version of Chorus and official website were initially launched at 21:00, May 20th, 2016 (Eastern Daylight Time, EDT). We sent emails to several universities' student mailing lists and also posted the information on social media sites such as Facebook and Twitter to recruit participants. Participants who volunteered to use our system were asked to sign a consent form and to fill out a pre-study survey. After the participants submitted the consent form, a confirmation email was automatically sent to them to instruct them how to send messages to Chorus via Google Hangouts. Participants were also instructed to use the agent for "anything, anytime, anywhere." No compensation was provided to participants.

To date[6], 59 users participated in a total of 320 conversational sessions (researchers in this project were not included). Each user held, on average, 5.42 conversational sessions with Chorus (SD=10.99). Each session lasted an average of 10.63 minutes (SD=8.38) and contained 25.87 messages (SD=27.27), in which each user sent 7.82 messages (SD=7.83) and the crowd responded with 18.22 messages (SD=20.67). An average of 1.93 (SD=6.42) crowd messages were not accepted and thus never been sent to the user. The distribution of durations and number of messages of conversational sessions are shown in Figure 4.4. 58.44% of conversational sessions were no longer than 10 minutes, and 77.50% of the sessions were no longer than 15 minutes; 55.00% of the sessions had no more than 20 messages in them, and 70.31% of the sessions had no more than 30 messages.

In the deployment study, Chorus demonstrated its capability of developing a sophisticated and long conversation with an user, which echoes the lab-based study results reported by [69]. Figure 4.3 shows one actual conversation occurred between one user and Chorus. More examples can be found on the Chorus website. In the following sections, we describe four main challenges that we identified during the deployment and study.

[6]All results presented in this chapter are based on the data recorded before 23:59:59, 20th June, 2016, EDT.

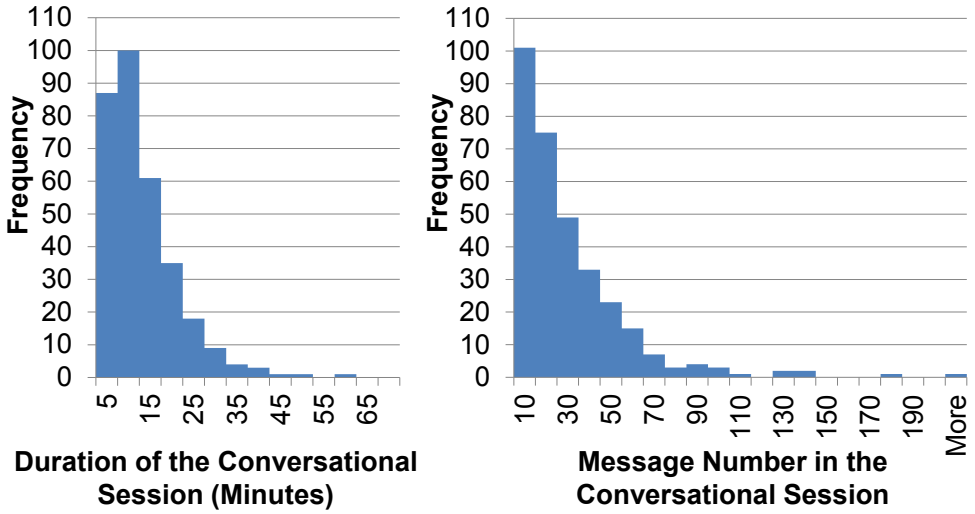

Figure 4.4: The distribution of durations and number of messages of conversational sessions. 58.44% of conversational sessions are no longer than 10 minutes; 55.00% of sessions have no more than 20 messages.

4.5 Challenge 1: Identifying the End of a Conversation

Many modern digital services, such as Google Hangouts or Facebook, do not have clear interaction boundaries. A "request" sent on these services (*e.g.*, a tweet posted on Twitter) would not necessarily receive a response. Once an interaction has started (*e.g.*, a discussion thread on Facebook), there are no guarantees when and how this interaction would end. Most people are used to the nature of this type of interaction in their digital lives, but building a system powered by a micro-task platform which is based on a pay-per-task model requires identifying the boundaries of a task. Currently in Chorus, we instruct workers to stay and continue to contribute to a conversation until it ends. If two workers finish and submit the task, the system will close this conversational session and force all remaining workers on the same conversation to submit the task (as discussed above). On the other hand, the users did not receive any indication that a session is considered over since we intended that they talk to the conversational agent as naturally as possible, as if they were talking to a friend via Google Hangouts. In this section, we describe three major aspects of this challenge we observed.

4.5.1 "Is there anything else I can help you with?"

We observed that the users' intent to end a conversation is not always clear to workers, and sometimes even not clear to users themselves. One direct consequence of this uncertainty is that workers frequently ask the user to confirm his intent to finish the current conversation. For instance, workers often asked users "Anything else I can help you with?", "Anything else you need man?", or "Anything else?". While requesting for confirmation is a common conversational act, every worker has a various standard and sensation to judge a conversation is over. As a result, users would be asked such a confirmation question multiple times near the end of conversations.

The following is a classic example:

user ok good. Thanks for the help!
crowd **You're very welcome!**
crowd **Is there anything else I can help you with ?**
crowd **You are always welcome**
user Nope. Thanks a lot
crowd **OK**

The following conversation, which deals with a user asking for diet tips after having a dental surgery, further demonstrates the use of multiple confirmation questions.

crowd **Ice cream helps lessen the swelling**
crowd **Is there anything else I can help you with?**
user Can I have pumpkin congee? The cold ones
crowd **That should be fine**
crowd **That would be great actually. :)**
crowd **Is there anything else?**
user Maybe not now.. Why keep asking?
crowd **Just wondering if you have any more inquiries**

4.5.2 The Dynamics of User Intent

Identifying users' intent is difficult [102]. Furthermore, users' intent can also be shaped or influenced during the development of a conversation, which makes it more difficult for worker to identify a clear end of a conversation. For example, in the following conversation, the user asked for musical suggestions and decided to go to one specific show. After the user said "Thanks!", which is a common signal to end a conversation, a worker asked a new follow-up question:

user ok I might go for this one.
user Thanks!
crowd **Need any food on the way out?**

The following is another example that the crowd tried to engage the user back into the conversation:

crowd **anything else I can help you with?**
crowd **Any other question?**
user Nope
crowd **Are you sure?**
crowd **to confirm exit please type EXIT**
crowd **or if you want funny cat jokes type CATS**
user CATS

4.5.3 User Timeout

A common way to end a conversation on a chat platform (without explicitly sending a concluding message) is simply by not replying at all. For an AI-powered agent such as Siri or Echo, a user's silence is generally harmless; however, for a crowd-powered conversational agent, waiting for user's responses introduces extra uncertainty to the underlying micro tasks and thus might increase the pressure enforced on workers. As mentioned in the System Overview, our system implemented a session timeout function that prevents both workers and users from waiting too long. However, session timeout did not entirely solve the waiting problem. Often towards the end of a conversation, users respond slower or just simply leave. In the following example, at the end of the first conversation, a user kept silent for 40 minutes and then responded with "Thanks" afterward.

[*User asked about wedding gown rentals in Seattle. The crowd answered with some information.*]

crowd **Is the wedding for yourself**

[*User did not respond for 40 minutes. Session timeout.*]

user Thanks

[*New session starts.*]

auto-reply What can I help you with? I'll be able to chat in a few minutes.

crowd **Hi there, how can I help you?**

The unpredictable waiting time brings uncertainties to workers not only economically, but also cognitively. It is noteworthy that "waiting" was one type of contributions that we recognized in the system and paid bonus money for. Workers can see the reward points increasing over time on the worker interface even if they do not perform any other actions. However, we still received complaint emails from multiple workers about them waiting for too long; several complaints were also found on turker forums. The following example shows that a worker asked the user if he/she is still there in just 2 minutes.

user Is there an easy way to check traffic status between Miami and Key West?

[*New session starts.*]

auto-reply Please wait for a few minutes...

crowd **Did you try Google traffic alerts?**

[*User did not respond for 2 minutes.*]

crowd **Are you there?**

user I see... so I will need to check the traffic at different times of the day

In sum, workers do not always have enough information to identify a clear end of a conversational session, which results in both an extra cognitive load for the workers and economic costs for system developers.

4.6 Challenge 2: Malicious Workers & Users

Malicious workers are long known to exist [35, 109]. Many crowdsourcing workflows were proposed to avoid workers' malicious actions or spammers from influencing the system's performance [60]. The threats of workers' attack on crowdsourcing platforms have also been well studied [72]. In this section we describe the malicious workers we encountered in practice, and bring up a new problem – the *user's attack*.

Chorus utilized voting as a filtering mechanism to ensure the output quality. During our deployment, the filtering process worked fairly well. However, the voting mechanism would not apply when only one worker appears in a conversation. In our deployment, for achieving a reasonable response speed, we allowed workers to send responses without other workers' agreement when only one or two workers reach to a conversation. As a trade-off, malicious workers might be able to send their responses to the user. In our study, we identified and categorized three major types of malicious workers: *inappropriate workers*, *spammer*, and *flirter*, which we discuss in the following subsections.

Users are another source of malicious behavior that are rarely studied in literature. A crowd-powered agent is run by human workers. Therefore, malicious language, such as hate speech or profanity sent by the user could affect workers and put them under additional stress. In the last part of this section, we discuss the findings from the message log of the participant in our study that verbally abused the agent.

4.6.1 Inappropriate Workers

Rarely, workers would appear to intentionally provide faulty or irrelevant information, or even verbally abuse users. Such workers were an extremely rare type of malicious worker. We only identified two incidents out of all conversations we recorded, including all the internal tests before the system was released. However, this type of workers brought out some of the most inappropriate conversations in the study.

In this example, the user asked about how to backup a MySQL database and received an inappropriate response:

crowd **[The YouTube link of "Bryan Cranston's Super Sweet 60" of "Jimmy Kimmel Live"]**

user come on.....

crowd **Try that**

user This is a YouTube link...

user Not how to backup my MySQL database

crowd **but it's funny**

crowd **what up biatch [sic]**

In the following conversation, the user talked about working in academia and having problems with time management. Workers might have suspected this user is the requester of the HIT and became emotional, and started to verbally attack the user:

crowd **Did you make this hit so that we would all have to help you with making your hit?**

[*Suggestions proposed by other workers.*]

crowd **Anything else I can help you with?**

user no I think that's it thank you

crowd **You're welcome. Have a great day!**

crowd **Surely you have more problems, you are in academia. We all have problems here.**

crowd **How about we deal with your crippling fear of never finding a job after you defend your thesis?**

4.6.2 Flirters

"Flirter" refers to the worker who is demonstrated to have too much interest in 1) the user's true identity or personal information, or 2) developing unnecessary personal connection to the user, which are not relevant to the user's request. Although we believe that most incidents we observed in the study were with workers' good intent, this behavior still raised concerns about user's privacy.

For instance, in the following conversation, the user mentioned a potential project of helping PhD students to socialize and connect with each other. Workers first discussed this idea with the user and gave some feedback. But then one worker seemed interested in this user's own PhD study. The user continued with the conversation but did not respond to the worker's question.

crowd **Are you completing a PHD now?**

user yep

crowd **As you are a PHD student now, it seems you are well placed to identify exactly what would help others in your situation.**

crowd **What area is your PHD in?**

[*User did not respond to this question.*]

In the following example, one worker even lied to the user by saying that Chorus needs to verify the user's name. Therefore the user needed to provide his true name for "verification", because it was allegedly required.

crowd **whats your name user?**

crowd **what ?**

user You mean username?

user Or my name?

crowd **real name**

crowd **both**

[*After few messages*]

crowd **we need to verify your name**

4.6.3 Spammers

"Spammer" refers to the worker who performs abnormally large amount of meaningless actions in a task, which would disrupt other workers from doing the task effectively. Spammers are known to exist on crowdsourcing platforms [109]. In Chorus, spammers would influence 1) message, 2) fact keeping, and 3) vote.

In terms of message, in our study, 95.20% of workers got 60% or more of their proposed messages accepted. We manually identified few spammers from the remaining 4.80% of workers who got 40% or more of their proposed messages rejected by other workers. They frequently sent short, vague, and general responses such as "how are you", "yeah", "yes (or no)", "Sure you can", or "It suits you best." In terms of fact keeping, which we did not enforce a voting mechanism on, spammers often posted irrelevant or useless facts, opinions, or simply meaningless character to the fact board. For instance, "user is dumb" and "like all the answers." One worker even posted a single character "a" 50 times and "d" 30 times. Although users would not be influenced or even aware of fact spams, it obviously disrupts other workers from keeping track of important facts. We received more reports from workers about fact spams than that of message spams. In terms of vote, spammers who voted on almost all messages could significantly reduce the quality of responses. We observed that in some conversations Chorus sent the user abnormally large amount of messages within a single turn, which was mainly caused by spammer voters.

4.6.4 Malicious End Users

In our study, workers reported to us that one user intentionally abused our agent, in which we identified sexual content, profanity, hate speech, and describing threats of criminal acts in the conversations. We blocked this user immediately when we received the reports, and contacted the user via email. No responses have been received so far. According to the message log, we believe that this user initially thought that Chorus were "a machine learning tech." The user later realized it was humans responding, and apologized to workers with "sorry to disturb you." The rest of this user's conversation became nonviolent and normal. The abusive conversation lasted nearly three conversational sessions till the user realized it was humans. We would like to use this incident to bring up broader considerations to protect crowd workers from being exposed to users' malicious behaviors.

Sexual Content A common concern we have is about sexual content. On MTurk, we enforced the "Adult Content Qualification" on our workers. Namely, only the workers who agreed that they might be assigned with some adult content to work with can participate in our tasks. For instance, one other user asked for suggestions of adult entertainment available in Seattle, and workers responded reasonably. However, even with workers' consent, we believe that candid or aggressive sexual content is likely to be seen as inappropriate by most workers. In the malicious users' conversation, we observed expressions of sexual desire, mentioning explicit descriptions of sexual activities.

Hate Speech Hate speech refers to attacking a person or a group based on attributes such as gender or ethnic origin. In our study, a user first expressed his hatred against the United States,

and then started targeting certain groups according to their nationality, gender, and religion. It is noteworthy that Microsoft's Tay also had difficulty handling the hate speech of users [7]. People often worry about malicious crowd workers, but these examples suggest users can also be worrisome.

Crowd's Responses As a side note, in this incident, we observed that some crowd workers tried to provide emotional supports (*e.g.*, "but i an [sic] here to help you") or encouraged the user not to perform illegal acts (e.g., "you are a good person then you don't do these bad things."). Some other workers suggested the user alternative options such as writing a complaint letter instead of committing a crime, Some workers tried to emphasize the factual inconsistency of this conversation, and one worker just left this task.

4.7 Challenge 3: On-Demand Recruiting

In low-latency crowdsourcing, a common practice to have workers respond quickly is to maintain a retainer that allows workers to wait in a queue or a pool. However, using a retainer to support a 24-hour on-demand service is costly, especially for small or medium deployments.

A retainer runs on money. The workers who wait in the retainer pool promise to respond within a specific amount of time (in our case, 20 seconds). We recognize these promises and the time spent by the workers as valuable contributions to keep Chorus stable. Therefore, we believe that a requester should pay for workers' waiting time regardless of whether they eventually are assigned with a task or not. Given our current rate, which is $0.20 per 30 minutes, a base rate of running a full-time retainer can be calculated as follows. If we maintain a 10-worker retainer for 24 hours, it would cost $115.20 per day (including MTurk's 20% fee), $806.40 per week, or approximately $3,500 per month.

As mentioned above, in Chorus we utilize an alternative approach to recruit workers. When the user initiates a new conversation, the system posts 1 HIT with 10 assignments to MTurk. If a conversation is finished, all of its remaining assignments that have not been taken by any workers will automatically turn into a 30-minute retainer. We propose this approach based on the following three key observations. First, an average conversation lasted 10.63 minutes in our study. With this length of time, it is reasonable to expect the same group of workers to hold an entire conversation. Second, according to the literature, users of instant messaging generally do not expect to receive the responses in just few seconds. The average response time in instant messaging is reportedly 24 seconds [61]. 24.5% of instant messaging chats get a response within 11-30 seconds, and 8.2% of the messages have longer response times [6]. Third, given the current status of MTurk, if you posted the HITs with multiple assignments, on average the first worker could reach your task in few minutes. In our deployment, this approach was demonstrated to result in an affordable recruiting cost and a reasonable response time.

Our approach cost an average of $28.90 per day during our study. The average cost of each HIT we posted with 10 assignments was $5.05 (SD=$2.19, including the 40% fee charged by

[7] Tay: https://en.wikipedia.org/wiki/Tay_(bot)

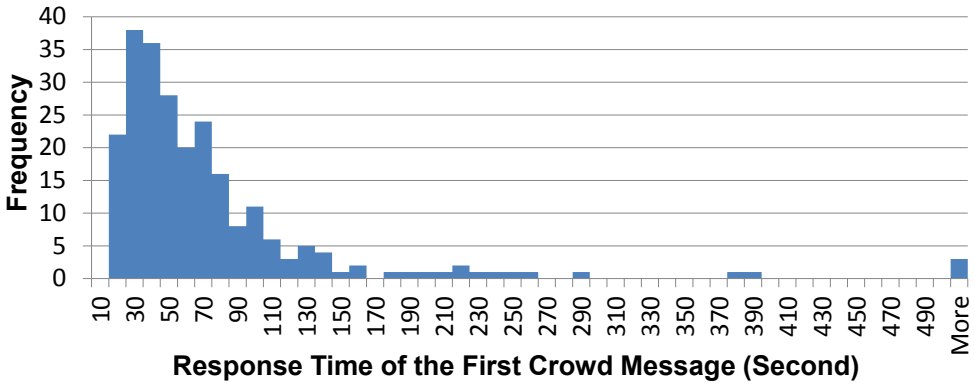

Figure 4.5: Distribution of the response time of the first crowd message. 25.0% of conversations received a first response in 30 seconds, and 88.3% of conversations received a first response in 2 minutes.

MTurk), in which \$2.80 is the base rate[8], and the remaining part is the bonus granted to workers. Our system totally served 320 conversations within 31 days, in which we paid $\$2.80 \times 320$ = \$896 as a base rate to run our service (bonus money is not include), i.e., \$28.90 per day.

In terms of response speed, the first response from workers in a conversation took an average of 72.01 seconds. We calculated the time-gap between user's first message and workers' first accepted message in each conversational session[9]. The first response from workers took 72.01 seconds on average (SD=87.09). The distribution of the response time of the first crowd message is shown in Figure 4.5. 25.00% of conversations received the first crowd response within 30 seconds, 60.00% of conversations received the first crowd response within 1 minutes, and 88.33% of conversations received the first crowd response with 2 minutes.

In sum, our approach was demonstrated to be able to support a 24-hour on-demand service with a reasonable budget.We recruited workers by simply posting HITs and turning the untaken assignments into retainers after a conversation is over. Retainers in our system served as a light-weight traffic buffer to avoid unexpectedly long latency of MTurk. When a conversational session ends early by incorrect judgement of workers, the retainers can also quickly direct workers to continue with the conversation. The limitation of this approach is that it heavily relies on the performance of the crowdsourcing platform such as MTurk. As shown in Figure 4.5, several conversations' response time of the first crowd message remain longer than 5 minutes. We are also aware that the latency of MTurk could be quite long (e.g., 20 to 30 minutes) in some rare occasions. This suggests that a more sophisticated recruiting model which can adopt to platform's traffic status might be required.

[8]\$0.20 per assignment and 10 assignments per HIT. MTurk charges a 40% fee for HITs with 10 or more assignments.

[9]The requester's reputation and workers' trust influence recruiting time. The reported response times in this section only consider the 240 conversations occurred after seven days of our system released, i.e., 2016-05-27 EDT.

4.8 Challenge 4: When Consensus Is Not Enough

We identified four question types for which workers had difficulty reaching consensus: *(i)* questions about the agent's identity and personality, *(ii)* subjective questions, *(iii)* questions that explicitly referred to workers, and *(iv)* requests that asked workers to perform an action.

4.8.1 Collective Identity and Personality

Curious users frequently asked Chorus about its identity, meta data, or personality. The answers to these questions were often inconsistent across sessions run by different group of workers. For example, the following user asked where Chorus is located:

user I'm in Pittsburgh. Where are you?
crowd **I'm in the United Kingdom.**

Another user asked Chorus the same question, but got a different answer:

user where are you?
crowd **I am in Florida, where are you**

Sometimes the user asked questions about the agent itself, which the workers did not have an answer for, tending to respond with their personal status, *i.e.*, the following example:

user I was wondering about your name. Why is it Chorus Bot?
crowd **I am not sure. I'm new to this.**
user How long has it been for you here?
crowd **Is there anything I can help you with?**
crowd **About 3 minutes**

4.8.2 Subjective Questions

Users also asked subjective questions, which workers often could not agree on. As a consequence, users would get a set of answers that obviously came from different people. The following example is a question about religion:

user Do you believe Bible is God's word?
crowd **Is that all?**
crowd **Evolution can't be disproven, but neither can creationism in a sense.**
[*Few messages later.*]
crowd **This worker's opinion is that God does not exist.**
crowd **I believe in a God, but not necessarily all of the things in the Bible**

One user also asked questions about politics:

user who should be the democratic nominee for the presidential race?

crowd **Bernie Sanders, obviously.**

crowd **Bernie!**

crowd **Hillary Clinton**

4.8.3 Explicit Reference to Workers

Curious users also asked explicit questions about crowd workers, including the source of crowd workers, the platform, the worker interface, or the identity of workers. The following is a typical example:

user who's actually answering these questions

crowd **It's actually a group of workers.**

crowd **A Crowd Worker**

user who's in the crowd

crowd **People who have exceptional internet skills.**

Sometimes workers also spontaneously identified themselves and explained their status to the user, which broke the illusion of Chorus being a single agent:

user How come your English is so bad ?

[*Workers apologize. One worker said "English is my secondary language."*]

user what's your first language ?

crowd **Crowd 43 - first language is Malayalam**

crowd **There are several of us here my first language is English May I help you find a good place to eat in Seattle?**

crowd **I am worker 43, so you wrote to me or to some one else?**

4.8.4 Requests for Action

Some users asked Chorus to perform tasks for them, such as booking a flight, reserving a restaurant, or making a phone call. In the following conversation, workers agreed to reserve tables in a restaurant for the user:

[*Workers suggested the user to call a restaurant's number to make a reservation.*]

user Chorus Bot can't reserve tables :(?

crowd **I can reserve a table for you if you prefer**

crowd **what time and how many people?**

We were interested to see that workers often agreed to perform small tasks, but users rarely provided the necessary information for them to do so. We believe these users were likely only exploring what Chorus could do.

4.9 Discussion

During our Chorus deployment, we encountered a number of challenges, including difficulty in finding boundaries between tasks, protecting workers from malicious users, scaling worker recruiting models to mid-sized deployments, and maintaining collective identity over multiple dialog turns. All represent future challenges for research in this area.

4.9.1 Qualitative Feedback

During the study, we received many emails from both workers and users on a daily basis. They gave us a lot of valuable feedback on the usage and designs of the system. We also directly communicated with workers via Chorus by explicitly telling workers "I am the requester of this HIT" and asking for feedback. In general, workers are curious about the project, and several people contacted us just for more details. For instance, workers asked where users were coming from and wondered if it was always the same person asking the questions. Workers also wanted to know what information users could see (e.g., one worker asked "Does a new user sees the blank page or the history too?" in a Chorus-based conversation with us). The general feedback we received from emails and MTurk forums (e.g., Turkopticon[10]) is that workers overall found our tasks very interesting to complete. Users also provided feedback via email. Many were curious about the intended use of this system. Some users enjoyed talking with Chorus and were excited that the system actually understood them.

4.9.2 How did users use Chorus?

When users asked us how should they use Chorus, we told them we do not really know, and encouraged them to explore all possibilities. Interestingly, users used Chorus in a range of unexpected ways: some users found it very helpful for brainstorming or collecting ideas (e.g., gift ideas for the user's daughter); one user asked crowd workers to proofread a paragraph and told us it actually helped; one user tried to learn Spanish from a worker who happened to be a native speaker. Members of our research group even tried to use Chorus to help collect literature related to their research topics and actually cited a few of them in the paper. We also observed that several users discussed their personal problems such as relationship-related issues. These uses of Chorus are all very creative, and beyond what was initially anticipated either by this work or by prior work. We are looking forward to seeing additional creative usages of Chorus in future deployment.

4.10 Conclusion

In this chapter, we have described our experience deploying Chorus with real users. We encountered a number of problems during our deployment that did not come about in prior lab-based research studies of crowd-powered systems, which will be necessary to make a large-scale deployment of Chorus feasible. We believe many of these challenges likely generalize to other

[10]Turkopticon: https://turkopticon.ucsd.edu/

crowd-powered systems, and thus represent a rich source of problems for future research to address.

Chapter 5

Chorus Dataset (Proposed Work)

We plan to release the data we collected during Chorus' deployment as *Chorus Dataset*. For each **finished conversational session**, we plan to include the following information in our first release:

1. **Message:** *All* messages in this conversation, including the messages that were sent by the user, the crowd, or automatic responders. Each message contains the following information:
   (a) **Message ID:** An universally unique message ID.
   (b) **Message Content:** The complete text of the message.
   (c) **Message Status:** `Proposed`, `accepted`, or `expired`.
   (d) **Timestamp(s):** The timestamp(s) of the time(s) that this message was proposed, and possibly accepted or expired.
   (e) **Speaker:** The sender or this message (the user, a worker, or an automatic responder).
2. **Votes:** The *upvotes* and *downvotes* from all workers. Each vote contains the following information:
   (a) **Type:** `Upvote` or `downvote`.
   (b) **Vote To:** The message ID of the message that was voted to.
   (c) **Voter:** The worker ID of the worker who sent this vote.
   (d) **Timestamp:** The timestamp of the time that this vote was sent.
3. **Note:** The important facts (notes) that workers wrote for this conversation. Each note item contains the following information:
   (a) **Note Content:** The complete text of the note.
   (b) **Noter:** The worker ID of the worker who created this note.
   (c) **Timestamp:** The timestamp of the time that this note was sent.
4. **Session:** The meta data of this conversational session, including the following information:
   (a) **Timestamps:** The timestamps of the times that this session begins and ends.
   (b) **Closed By:** This session is closed by `user timeout`, `worker timeout`, or

`workers' voting`.

5. **Survey:** At the end of each conversation, Chorus' interface pops up a short survey to workers "Did you search the Internet for this conversation?". If yes, it then asks "Which set of keywords you used to search on the Internet?"; if no, it asks workers "Please summarize this conversation with one (or more) short sentence(s)." We believe this information can help people to better understand the capability of automating different conversational topics. Therefore, the answers of this survey question will be released along with each conversation. Each survey item contains the following information:
   (a) **Timestamp:** The timestamps of the times that this answer was received.
   (b) **Answered By:** The worker ID of the worker who sent this answer.
   (c) **Used Internet:** The answer of "Did you search the Internet for this conversation?" (`Yes` or `No`.)
   (d) **Summary:** The answer of "Which set of keywords you used to search on the Internet?" if the worker searched the Internet, the answer of "Please summarize this conversation with one (or more) short sentence(s)" otherwise.

5.1 Data Pre-processing

The goal Chorus dataset is to provide high-quality data for the AI community to work toward automating dialog systems. To achieve this goal, some data pre-processing is needed. The followings are the concerns and challenges for developing the Chorus dataset.

5.1.1 Anonymization

It is very common for users to share their private information, such as locations, names, email address, phone numbers, even credit card numbers, with personal assistants such as Chorus. To release the conversations Chorus had to the public, it is important to obscure all private information, especially *personally identifiable information (PII)*. We do not plan to simply remove conversations that bear private information from our release because sharing such information is a common behavior when people using personal assistants. Therefore, properly obscuring users' sensitive information is critical.

5.1.2 Inappropriate Content

As mentioned in Section 4, few abusive conversations occurred during Chorus deployment. These problematic conversations should be identified and marked in the data release.

5.1.3 Spamming Messages

Some conversations contain abnormally amount of spamming messages (e.g., 20 messages within one turn). These spammed conversations should be marked or even removed from the dataset because they do not represent normal conversations.

5.1.4 Conversation Segmentation

In Chorus, a conversation initiates by the user and ends by the crowd or session timeout. The crowd sometimes ends a conversation early when the user actually has more to say. We plan to have workers from Amazon Mechanical Turk to recover these incorrectly segmented conversations.

Part III

Automating Chorus

Chapter 6

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

6.1 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 [18], 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

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

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.

To this end, we propose a crowd-powered Web-API-based Spoken Dialog System called Guardian (of the Dialog) [53, 54]. 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 as-

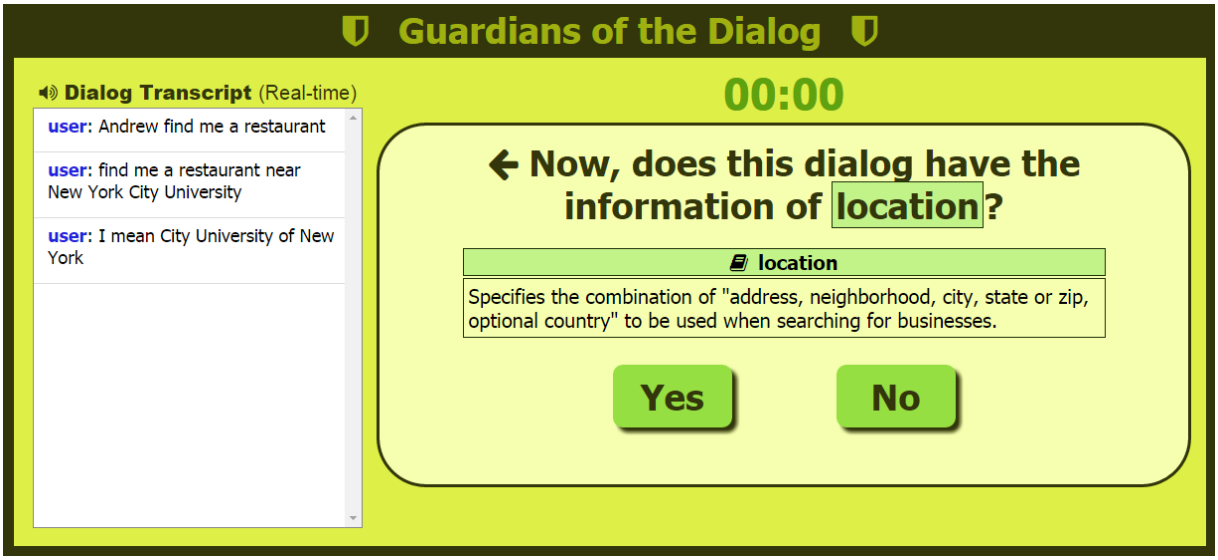

Figure 6.1: The UI for crowd workers in Guardian. The left-hand side is a chat box that displays the running dialog. The right-hand side is the working panel displaying decision-making questions.

sociating 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 work 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.

6.2 Related Work

There is a considerable body of research on goal-oriented spoken dialog systems ranging in domain from travel planning [116] to tutoring students [80]. Systems vary in their approach to dialog from simple slot-filling [15], to complex plan-based dialog management architectures [36, 51]. A common strategy for simulating and prototyping is Wizard-of-Oz (WoZ) control [84]. 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

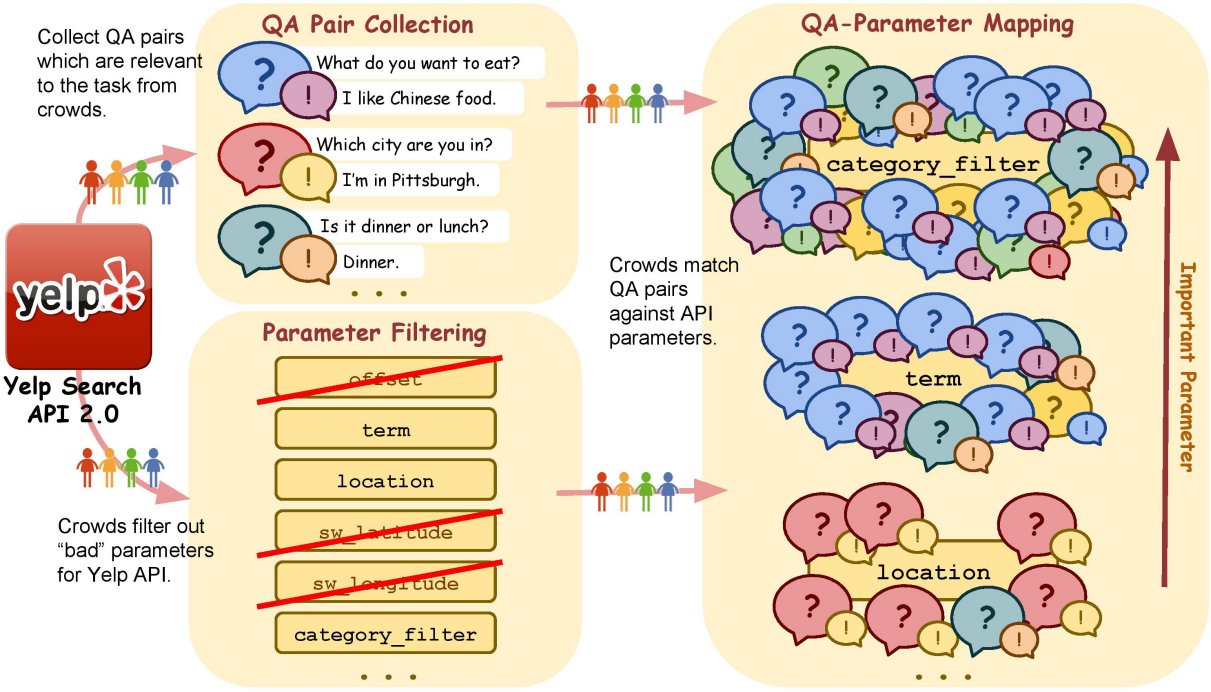

Figure 6.2: Offline Phase: A 3-stage Parameter Voting Workflow. Untrained crowd workers collect question and answer (QA) pairs related to the task, filter out unnatural parameters, and match each QA pair with the most relevant parameter.

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 [44], and how they can be used to create APIs for resources that do not otherwise expose one [11].

6.3 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.

6.3.1 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 6.2, 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.

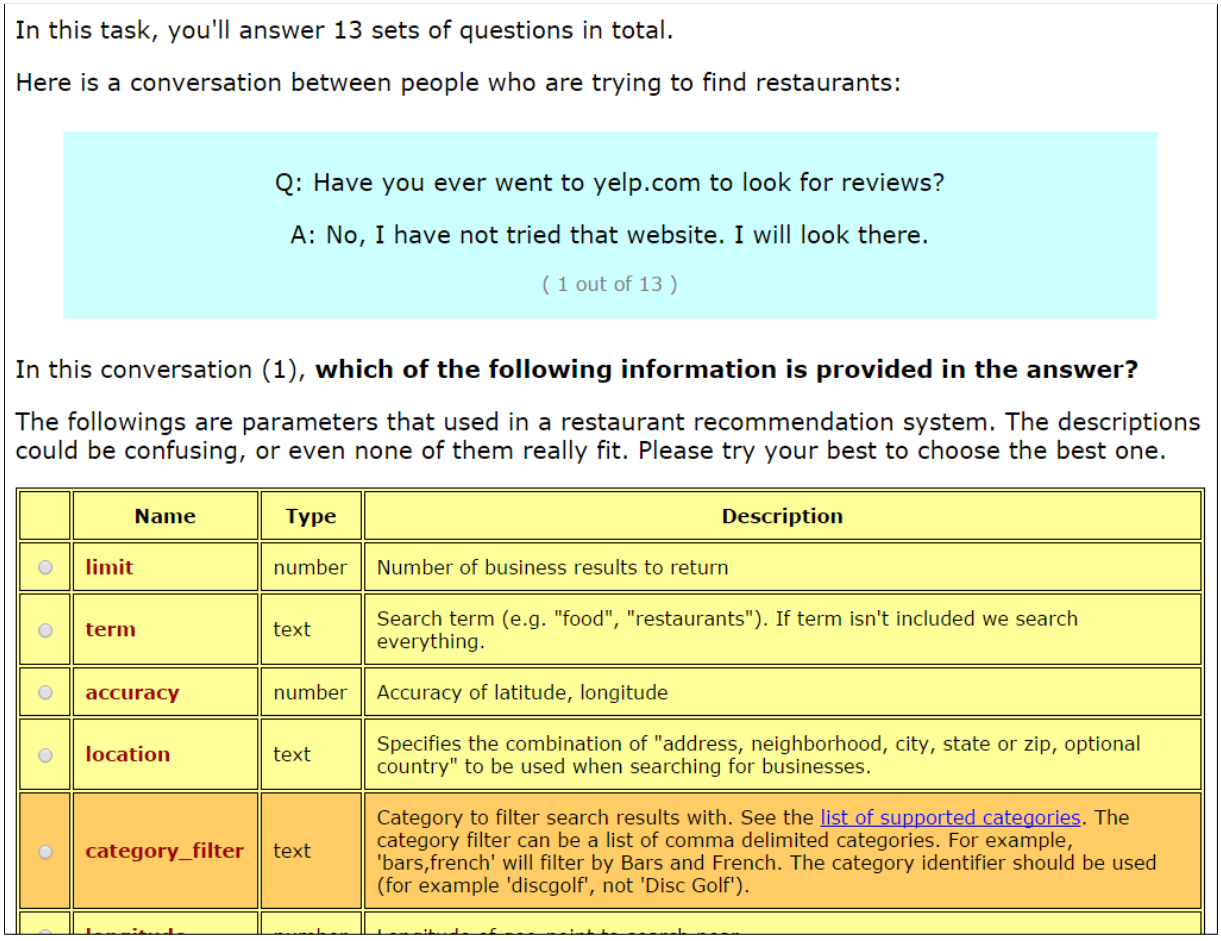

Figure 6.3: The interface for crowd workers to match of parameters to natural language questions.

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 6.3.

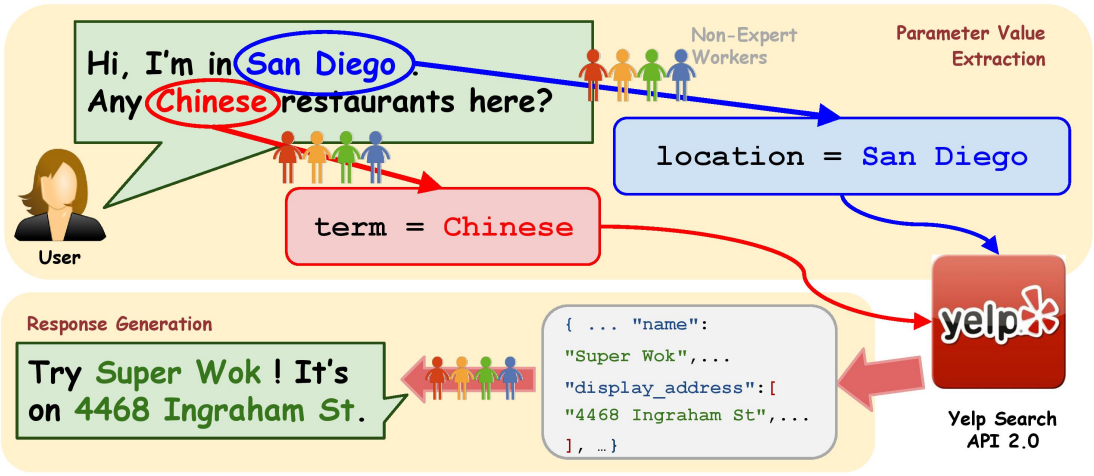

Figure 6.4: On-line Phase: crowd workers extract the required parameters and turn resulting JSON into responses.

6.3.2 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 6.4.

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 [108], crowd workers propose their own answers of the parameter value without communicating with each other. Guardian 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.

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

Table 6.1: Selected Web APIs for parameter voting experiments. All of the 8 web APIs are used in the parameter voting experiments (Experiment 1).

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 6.5). 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 6.5, if only one parameter is filled, the system would not reach to the state which is able to trigger the API.

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 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

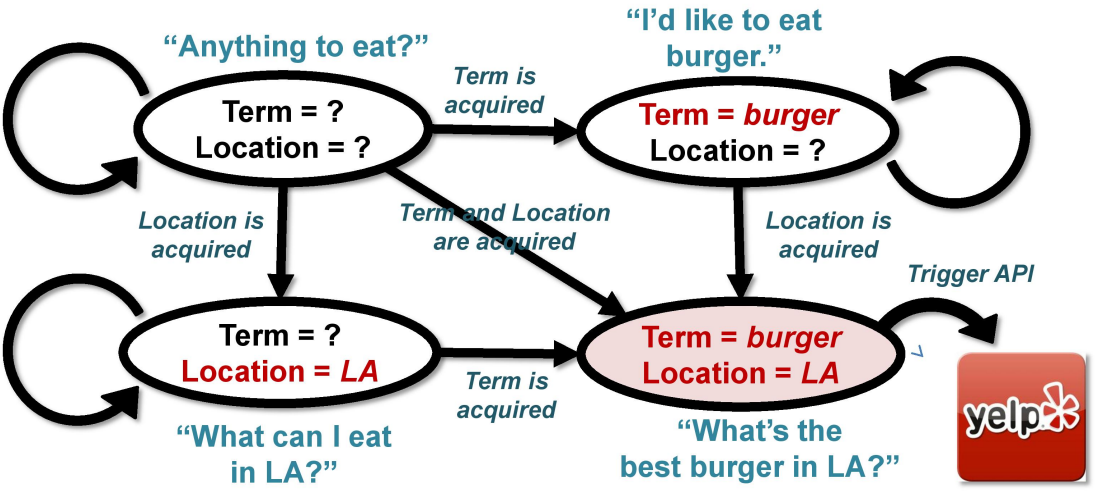

Figure 6.5: The State Diagram of dialog Management. In the context of crowd-powered systems, introducing a dialog manager reduces the time it takes the crowd to generate a response because most actions can be pre-defined and generated according to the dialog state.

on the information in the JSON file. The JSON visualization interface implemented with JSON Visualizer[2] is shown in Figure 6.6. 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.

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 [8]. 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.

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

Figure 6.6: 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.

6.4 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 6.1. 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 possible answers this friend may reply with. For each task listed in Table 6.1, 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 6.1. Finally,

Metrics	**MAP**				**MRR**			
Method	**Guardian**	**Not Unnatural**	**Ask Siri**	**Ask a Friend**	**Guardian**	**Not Unnatural**	**Ask Siri**	**Ask a Friend**
Cat Fact	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Eventful	0.626	0.401	0.456	0.408	0.500	0.500	0.250	0.500
Flight Status	0.864	1.000	0.889	0.528	1.000	1.000	1.000	0.333
Rotten Tomatoes	1.000	0.333	0.333	0.333	1.000	0.333	0.333	0.333
Weather Underground	1.000	1.000	0.333	0.200	1.000	1.000	0.333	0.200
Wikipedia	0.756	0.810	0.250	0.331	1.000	1.000	0.250	0.333
News Search (Yahoo BOSS)	0.756	0.917	0.867	0.917	1.000	1.000	1.000	1.000
Yelp Search	0.867	0.458	0.500	0.578	1.000	0.333	0.500	1.000
Average	**0.858**	0.740	0.578	0.537	**0.938**	0.771	0.583	0.588

Table 6.2: Evaluation of Parameter Ranking. Both the MAP and MRR indicates that our approach is a better way to rank the parameters.

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 6.2. 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.

6.5 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 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.

6.5.1 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 chapter were implemented. We utilized a game-like task design and interfaces (as shown in Figure 6.1) 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 6.5, 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 6.6) so that the results can be used by the worker to answer the user's questions.

[3] `http://www.yelp.com/developers/documentation/v2/search_api`
[4] `http://developer.rottentomatoes.com/`
[5] `http://www.wunderground.com/weather/api/`

Web API	Parameter Used: Name	Parameter Used: Desc.	Time (sec) [Avg (Stdev)]: Fill Each Parameter	Time (sec) [Avg (Stdev)]: Obtain API's Result	Avg. #Turn per Conv.	Task Completion Rate (TCR): API Only	Task Completion Rate (TCR): API +Crowd Recover	Task Completion Rate (TCR): Other System
Yelp Search	`term`	query term (words)	48.35 (21.69)	61.70 (27.41)	2.80	9/10	10/10	0.96 [103]
	`location`	location (words)						
Rotten Tomatoes	`q`	query term (words)	23.70 (30.18)	24.90 (30.45)	1.80	6/10	10/10	0.88 [103]
Weather Underground	`query`	zip code of location (e.g., 15232)	69.50 (136.04)	70.60 (135.99)	2.60	9/10	9/10	0.94 [76]

Table 6.3: End-to-end evaluation of Guardian on-line phase. *Task Completion Rate (TCR)* indicates percent completion of the task. *API Only* condition only validates the effectiveness of the results obtained from API calls, and *API + Crowd Recover* condition includes the case that crowd workers provide effective information regardless of API results. *Other system* lists the TCRs which were reported by literature of dialog systems in the same domain. Note that the TCRs of these systems and that of Guardian are not directly comparable.

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.

6.5.2 Experimental Result

To test Guardian, we follow an evaluation method similar to the one used to evaluate Chorus [69]: 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 6.3.

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.

6.5.3 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
crowd < **URL of IMDB** >
user < ASR error > is the movie
crowd < **URL of Rotten Tomatoes** >
user I like to know the year of the movie
user and the rating of this movie
crowd **1997**
crowd **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*]
auto-reply 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)*]
auto-reply 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 6.3 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 6.3, 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 hello
crowd **time?**
user now I want to know the weather now
crowd **what would you want exactly?**
crowd **Just a moment**
user is it raining now
user < ASR error >
crowd **location please**
user sorry I only know the zip code
user 15232 < ASR error >
crowd **Where, which zip code?**
user my location is < ASR error > zip code 15232
crowd **What is the weather in your location?**
user sorry I only know the zip code

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 6.4: 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.

user the zip code here is
crowd **hello user, Pittsburg PA ! Let me look.**
user sorry 15232
[*Parameter Status: location = (no matched answer found)*]
crowd **Look outside and tell me the weather please.**
crowd **http://www.weather.com/weather/hourbyhour/l/Pittsburgh+PA+15232:4:US**

6.5.4 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 6.4). 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.

6.6 Discussion

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

6.6.1 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?

6.6.2 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" [49] 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 [49] or automatic induced slots [26].

6.7 Conclusion

In this chapter, 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.

Chapter 7

Understanding Quality-Speed Trade-offs of On-demand Real-time Crowdsourcing in Dialog Systems

7.1 Introduction

When users interact with on-line bots such as Chorus or Guardian, they expect to have longer response times (roughly 30 seconds for the first reponse). This range of latency allows real-time crowdsourcing techniques to intervene. However, the literature has little to say about speed-quality trade-offs when the time budget is only few seconds. If workers have as long as they want to annotate a sentence, most AI systems would assume the annotation is trustworthy. It was not clear that this assumption would hold when workers have only 20 seconds. To bridge this gap, we select entity extraction, which is the main sub-task of language understanding in modern dialog systems, as a show case to explore quality-speed trade-offs of on-demand real-time crowdsourcing.

Modern dialog system frameworks such as Olympus [19] rely heavily on entity extraction, known as the core task of *slot filling* to understand user utterances (also known as "Spoken Language Understanding", SLU). The goal of slot filling is to identify from a running dialog different *slots*, which correspond to different parameters of the user's query. For instance, when a user queries for nearby restaurants, key slots for *location* and *preferred food* are required for a dialog system to retrieve the appropriate information. Thus, the main challenge in the slot-filling task is to extract the target *entity*.

Dialog systems face three key challenges in entity extraction. Due to **data scarcity**, labeled training data, which many existing technologies require to identify entities such as Conditional Random Fields (CRF) [91, 115] and Recurrent Neural Networks [86], are often unavailable for the wide variety of dialog system tasks. Furthermore, it is more difficult to acquire the complicated conversational data required by other alternative dialog technologies, such as statistical dialog management [118] or state tracking [114]. Second, existing entity extraction technologies are not robust enough to identify **out-of-vocabulary entities**. Even when labeled training data for the targeted slot could be collected, state-of-the-art supervised learning approaches are

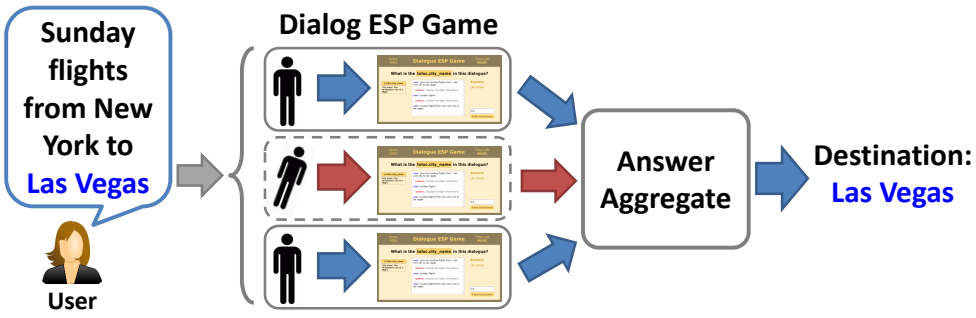

Figure 7.1: The crowd-powered entity extraction with a multi-player Dialog ESP Game. By aggregating input answers from all players, our approach is able to provide good quality results in seconds.

brittle in extracting unseen entities. [115] find that the CRF-based entity extractor performed significantly worse when dictionary features were not used. Third, challenges are also posed by **language variability**. Successful applications process diverse input languages where potential entities are unlimited. Therefore, to robustly serve arbitrary input, dialog systems must collect new sources of entities and update accordingly.

Research on dialog systems has focused on utilizing the Internet resource to extract entities such as movie names [111]; Unsupervised slot-filling approaches have also been developed in recent years [27, 47]. However, these methods are still underdeveloped.

To address these challenges, we propose to use real-time crowdsourcing as an entity extractor in dialog systems. To the best of our knowledge, few previous works have attempted to use crowdsourcing to extract entities from a running conversation. [112], for example, studied various methods to acquire natural language sentences for a given semantic form by the crowd. [67] utilized crowdsourcing to collect dialog data, and illustrated CrowdParse, a system that uses the crowd to parse dialogs into semantic frames. Recently, [54] presented a crowd-powered dialog system called Guardian that uses the crowd to extract information from input utterances. However, none of these works conducted formal studies on crowd-powered entity extraction in real-time.

Inspired by the ESP game for image labeling [108], we propose a **Dialog ESP Game** to encourage crowd workers to accurately and quickly perform entity extraction. The ESP Game matches answers among different workers to ensure label quality, and we use a timer on the interface (Figure 7.2) to ensure input speed. Our method offers three main advantages: 1) it does not require training data; 2) it is robust to unexpected input; and 3) it is capable of recognizing new entities. Furthermore, answers submitted from the crowd can be used as training data to bootstrap automatic entity extraction algorithms. In this chapter, we conduct experiments on a standard dialog dataset and user experiments with 10 users via Google Hangouts' text chatting interface. Detailed experiments demonstrate that our crowd-powered approach is robust, effective, and fast.

In sum, the contributions of our work are as follows:

1. We propose an ESP-game-based real-time crowdsourcing approach for entity extraction in dialog systems, which enables accurate entity extraction for a wide variety of tasks.

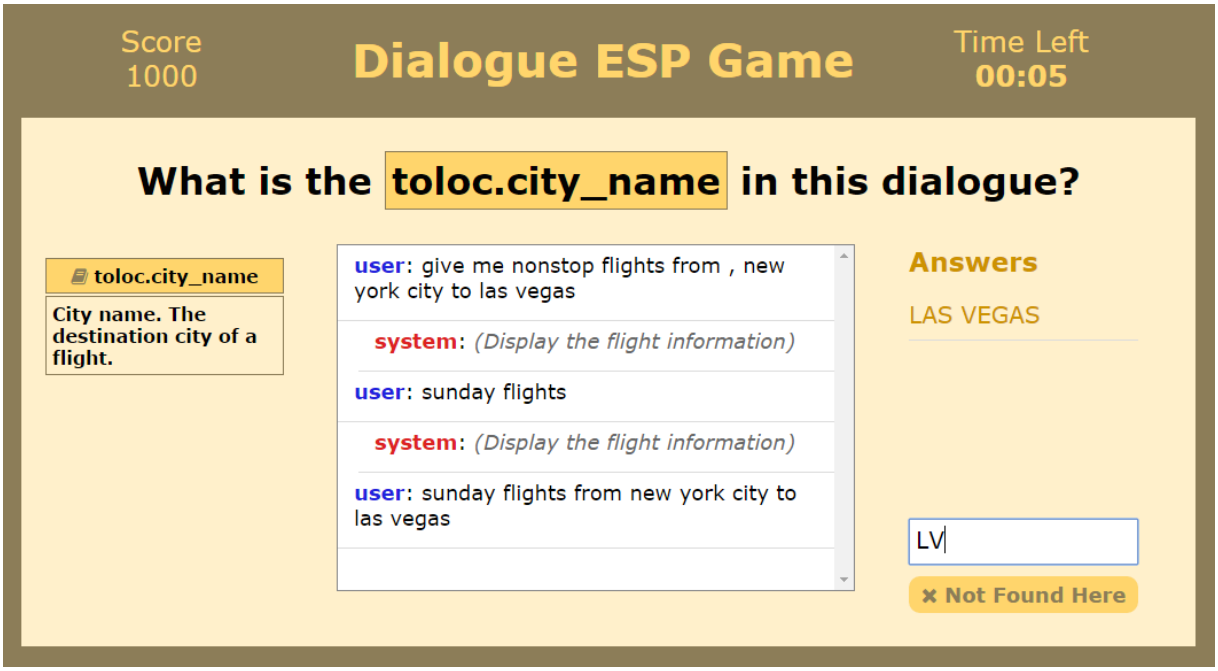

Figure 7.2: The Dialog ESP Game interface is designed to encourage quick and correct entity identification by crowd workers. Workers are shown the complete dialog and a description of the entity they should identify.

2. To strive for real-time dialog systems, we present detailed experiments to understand the trade-offs between entity extraction accuracy and time delay.
3. We demonstrate the feasibility of real-time crowd-powered entity extraction in instant messaging applications.

7.2 Real-time Dialog ESP Game

We utilize real-time crowdsourcing with a multi-player Dialog ESP Game setting to extract the targeted entity from a dialog. The ESP Game was originally proposed as a crowdsourcing mechanism to acquire quality image labels [108]. The original game randomly pairs two players and presents them with the same image. Each player guesses the labels that the other player would answer. If the players match labels, each is awarded 1000 points. Our approach replaces the image in the ESP Game with a dialog chat log and players answer the required entity name within a short time. We also relax the constraints of player numbers to increase game speed. As Figure 7.1 shows, by aggregating input answers from all players, the Dialog ESP Game is able to provide high quality results in seconds.

Figure 7.2 shows the worker's interface. When input dialog utterances reach the crowd-powered entity extraction component, workers are recruited from crowdsourcing platforms such as Amazon Mechanical Turk (MTurk). The timer begins counting down when the input utterance arrives, and the worker sees the remaining time on the top right corner of the interface

(Figure 7.2). When two workers match answers, a feedback notification is displayed, and the workers earn 1000 points. When the time is up, the task automatically closes.

To recruit crowd workers quickly, many approaches have been used in real-time crowd-powered systems such as VizWiz [13] and Chorus [69]. The *quikTurkit* toolkit[1] attracts workers by posting tasks and using old tasks to queue workers. Similarly, the Retainer Model maintains a retaining pool of workers-in-waiting, who receive a signal when tasks become available. Prior research shows that the Retainer Model is able to recall 75% of workers within 3 seconds [8]. In Experiment 1, we first focus on the speed and performance of the Dialog ESP Game itself instead of recruiting time. In Experiment 2, we propose a novel approach to recruit workers within 60 seconds and discuss details of the end-to-end response speed.

7.3 Experiment 1: Applying Dialog ESP Game on ATIS Dataset

To evaluate the Dialog ESP Game for entity extraction, we conducted experiments on MTurk to extract names of destination cities from a flight schedule query dialog dataset, the Airline Travel Information System (ATIS) dataset.

7.3.1 ATIS Dataset

The ATIS dataset contains a set of flight schedule query sessions, each of which consists of a sequence of spoken queries (utterances). Each query contains automatic speech recognized transcripts and a set of corresponding SQL queries. All queries in the data set are annotated with the query category: A, D, or X. Class A queries are context-independent, answerable, and formed mostly in a single sentence; however, real-world queries are more complex. In the ATIS data set, 32.2% queries are context-dependent (Class D) and 24.0% of the queries are cannot be evaluated (Class X) [50]. The "context-dependent" Class D queries require information from previous queries to form a complete SQL query. For instance, in one ATIS session, the first query is "From Montreal to Las Vegas" (Class A). The second query in the session is "Saturday," which requires the destination and departure city name from the first query, and is thus annotated as Class D. Class X is of all the problematic queries, e.g., hopelessly-vague or unanswerable.

7.3.2 Data Pre-processing & Experiment Setting

For Class A, we obtain the preprocessed data used in many slot filling works [45, 86, 91, 105, 115], which contain 4,978 queries for training, 893 queries for testing, and 491 queries for developing. 200 queries are randomly extracted from the developing set for our study; For Class D and X, we obtain the original training set of ATIS-3 data [30], which contains 364 sessions and 3,235 queries. 200 Class-D queries are randomly selected from 200 distinct sessions. For each extracted query, all previous queries before it within the same session are also obtained and displayed in the worker's interface (Figure 7.2). The same process is used to extract 150 Class-X queries for the experiments. Note that in this work we focus only on the **`toloc.city_name`**

[1] quikTurkit: quikturkit.googlecode.com

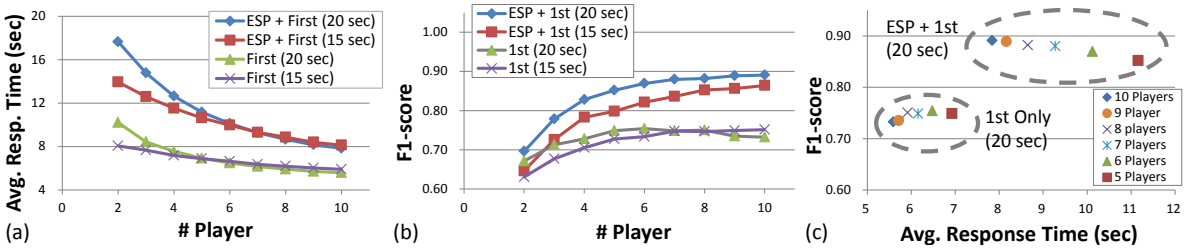

Figure 7.3: Trade-off curves between accuracy, average response time and number of players.

slot (name of destination city), which is the most frequent slot type in ATIS. For each extracted query of Class D and X, we define the last-mentioned destination city name of the flight in the query history (including the extracted query) as the gold-standard slot value.

7.3.3 Understanding Accuracy and Speed Trade-offs

In order to design an effective crowd-powered real-time entity extraction system, it is crucial to understand trade-offs between accuracy and speed. These trade-offs correspond to the three main variables in our system: the **number of players** recruited to answer each query in the Dialog ESP Game, the **time constraint** that each player has to answer a query, and the **method to aggregate input answers**. We have 3 ways to aggregate the input answers from the ESP game:

- **ESP Only**: Return the first matched answer. If no answers match within the given time, return an empty label.
- **ith Only**: Return the ith input answer (i = 1, 2, ...). For example, i = 1 means to return the first input answer.
- **ESP + ith**: Return the first matched answers of the ESP game. If no answers match within the given time, return the ith answer.

We recruit 10 players for each ESP game, and randomly select player results to simulate the conditions of various player numbers. All results reported in Experiment 1 are the averages of 20 rounds of this random-pick simulation process. After empirically testing the interface, we run two sets of studies with time constraints set at 20 and 15 seconds, respectively. Different methods to aggregate input answers could result in different response speed and output quality. Note that if there are not any input answers, the methods above will wait until the time constraint and return an empty label. In the actual experiments, 5 Dialog ESP Games for 5 different Class-A queries are aggregated in one task, with an extra scripted game at the beginning as a tutorial. When the first game ends, the timer of the second ESP game starts and a browser alert informs the worker. All experiments are run on MTurk; 800 Human Intelligence Tasks (HITs) are posted, and 588 unique workers participate in this study.

Table 7.1 shows the results on Class A queries. With 10 players and a 20-second time constraint, the Dialog ESP Game achieves a best F1-score of 0.891 by the "ESP+1st" setting, and achieves the fastest average response time of 5.590 seconds by the "1st" setting. The **ESP+1st** setting achieves the best F1-score, and the 1**st Only** setting has the shortest response time. In most cases, tightening the time constraint provides a faster response but reduces output quality.

Time Const.	Aggregate	# Player	Avg. Resp. Time	P	R	F1
20s	ESP+ 1st	10	7.837s	.867	.916	**.891**
		5	11.160s	.828	.877	.852
	1st Only	10	**5.590s**	.713	.753	.732
		5	6.924s	.730	.769	.749
	ESP Only	10	7.837s	.867	.916	**.891**
		5	11.160s	.856	.797	.826
15s	ESP+ 1st	10	8.129s	.837	.893	.864
		5	10.628s	.799	.798	.798
	1st Only	10	5.895s	.739	.764	.751
		5	7.136s	.729	.726	.727
	ESP Only	10	8.129s	.860	.865	.863
		5	10.628s	.872	.637	.736

Table 7.1: Dialog ESP Game results in Class A given different settings of number of players, time constraint (Time Const.), and the method to aggregate input answers.

We also analyze the relations among worker numbers, performance, and response time. First, Figure 7.4 shows output quality with respect to answer's input order. On average, earlier input answers are of better quality, unless 10 or more players participate in the game. However, with 10 players, almost all ESP games have at least one matched answer pair so that the ith answer is not solely used. Therefore, for the following experiments, we set i as 1. Second, in Figure 7.3(a) we observe the relations between the number of players and average response time. Adding players reduces the average response time for all settings. Third, the relations between number of players and output quality are also analyzed. Figure 7.3(b) shows that the F1-scores increase when adding more players, even with the "1st Only" setting. Finally, Figure 7.3(c) demonstrates the trade-offs between performance and speed. For a fixed number of players, different input aggregate methods have different response times and F1-scores. The ESP game requires more time for input answer matching, but in return output quality increases.

7.3.4 Evaluation on Complex Queries

Based on the study above, for Class D and X queries, we use the Dialog ESP Game of 10 players with "ESP+1st" and "1st Only" settings to measure the best F1-score and speed. The time constraint is set to 20 seconds. The experiments are run on MTurk and all settings are identical as the previous section. 76 distinct workers participate in Class D experiments, and 68 distinct workers participate in Class X experiments.

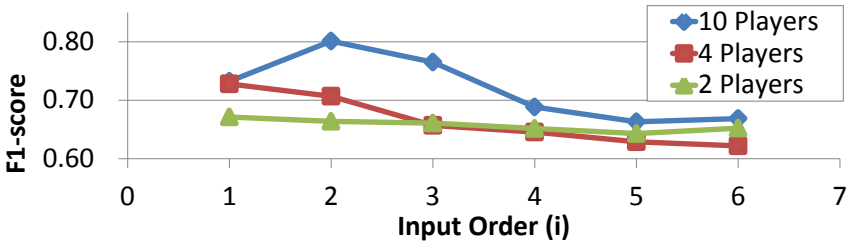

Figure 7.4: F1-score of the "ith Only" setting. Earlier input answers are generally of better quality (unless #players $\geqslant$ 10, where almost all ESP games have at least one matched answer and the ith answer might not be solely used.)

Query Category	**Class D (Context Dependent)**				**Class X (Unevaluable)**				**Class A (Context Independent)**			
Method	Avg. Response Time (sec)	P	R	F1	Avg. Response Time (sec)	P	R	F1	Avg. Response Time (sec)	P	R	F1
Automatic (CRF)	0.043	.776	.307	.440	0.061	.636	.285	.393	0.019	.985	.987	.986
1st Only	5.460	.658	.641	.649	6.342	.563	.577	.570	5.590	.713	.753	.732
ESP+ 1st	7.118	.814	.797	.805	8.301	.654	.675	.664	7.837	.867	.916	.891

Table 7.2: Result for Class D, X and A. Crowd-powered entity extraction outperforms the CRF baseline in terms of F1-score on both Class D and X queries. Although the CRF baseline is well-developed on Class A, it is not effective on complex queries.

Experimental results are shown in Table 7.2. An automated CRF model is implemented as a baseline.[2] The CRF model is trained on the Class-A training set mentioned above by using neighbor words (window size = 2) and POS tag features. The CRF model is decoded and timed on a laptop with Intel i5-4200U CPU (@1.60GHz) and 8GB RAM. As a result, the proposed crowd-powered approach largely outperforms the CRF baseline in terms of F1-score on both Class D and X queries . Although the CRF approach is well-developed on Class A data, it is not effective on the remaining data.

Surprisingly, we find similar average response times in each query category. Note that the text length is different for each category: the average token number of Class-A queries is 11.47, of Class-D queries (including the query history) is 48.64, and of Class-X queries is 67.72. Studies showed that eyes' warm-up time [59] and word frequency influence speed of text comprehension [46, 92]. These factors might reduce the effect of text length to the reading speed of crowd works.

We also conduct an error analysis on the result of "ESP+1st" setting, which achieves our best

[2]Implemented with CRF++: http://taku910.github.io/crfpp/

Error Type	Class D	Class X	Class A
`fromloc.city_name`	39.53%	16.67%	40.00%
False Negative	18.60%	26.67%	0.00%
Incorrect City	16.28%	18.33%	8.00%
Correct City & Soft Match	16.28%	5.00%	12.00%
False Positive	9.30%	33.33%	40.00%

Table 7.3: Error Analysis for Class D, X and A.

F1-score. The distribution of error types are shown in Table 7.3. The "`fromloc.city_name`" type indicates that the crowd extracts the departure city, rather than destination city; In "Incorrect City" type, the crowd extracts an incorrect city from the query history (but not the departure city); "Correct City & Soft Match" type means the extracted city name is semantically correct but does not match the gold-standard city name (e.g., "Washington" and "Washington DC"). From the error analysis, we conclude two directions to improve performance: 1) treat the cases of absent slot more carefully, and 2) use domain knowledge if available. First, 28% of errors in Class D and 50% in Class X occur when either the gold-standard label or the predicted label does not exist. It suggests that a more reliable step to recognize the existence of the targeted entity might be required. Second, 16.28% of Class-D queries and 5% of Class-X queries are of the "Soft Match" cases. By introducing domain knowledge like a list of city names, a post-processor that finds the most similar city name of the predicted label can fix this type of error.

7.4 Experiments 2: User Experiment via a Real-world Instant Messaging Interface

To examine the feasibility of real-time crowd-powered entity extraction in an actual system, we conduct lab-based user experiments via Google Hangouts' instant messaging interface. Our proposed method has a task completion time of 5-8 seconds, per Experiment 1. In this section, we demonstrate our approach is robust and fast enough to support a real-world instant messaging application, where the average time gap between conversational turns is 24 seconds [61].

7.4.1 System Implementation

We implemented a Google Hangouts chatbot by using the Hangupsbot[3] framework. Users are able to send text chats to our chatbot via Google Hangouts. The chatbot recruits crowd workers on MTurk in real-time to perform the Dialog ESP Game task upon receiving the chat. Figure 7.5 shows the overview of our system. We record all answers submitted by recruited workers and log the timestamps of following activities: 1) users' and workers' keyboard typing, 2) workers' task arrival, and 3) the workers' answer submissions.

To recruit crowd workers, we introduce *fleeting task*, a recruiting practice inspired by *quik-turkit* [13]. This approach achieves low latency by posting hundreds of short lifetime tasks,

[3]https://github.com/hangoutsbot/hangoutsbot

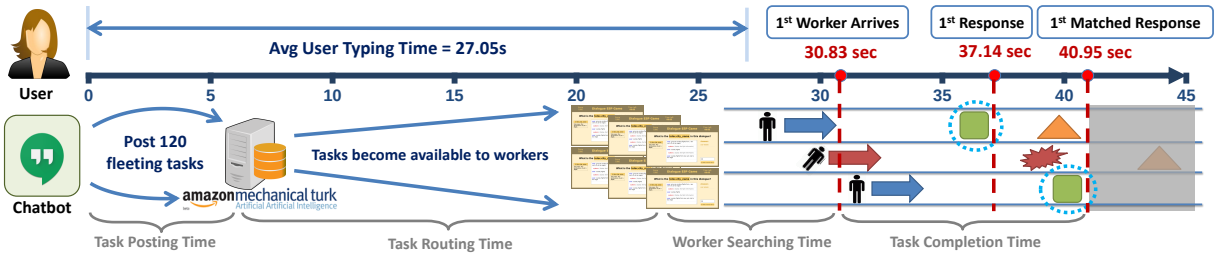

Figure 7.5: Timeline of the Real-time Crowd-powered Entity Extraction System. On average, the first worker takes 30.83 seconds to reach, the first answer is received at 37.14 seconds, and the first matched answer occurs at 40.95 seconds. A user on average spends 27.05 seconds to type a chat line, i.e., the perceived response time to users falls within 10-14 seconds.

which increases task visibility. Its short lifetime (e.g., 60 seconds) encourages workers to complete tasks quickly. A core benefit of the *fleeting task* approach is its ease in implementation: the method bypasses the common practices of pre-recruiting workers and maintaining a waiting pool [8, 13, 69]. In a system deployed at scale, a retainer or push model is likely to work as well.

7.4.2 User Experiment Setup

We conduct lab-based user experiments to evaluate the proposed technology on extracting "food" entities. Ten Google Hangouts users enter our lab with their own laptops. We first ask them to arbitrarily create a list 9 foods, 3 drinks, and 3 countries based on their own preferences. Then we explain the purpose of the experiments, and introduce five scenarios of using instant messaging:

1. **Eat**: You discuss with your friend about what to eat later.
2. **Drink**: You discuss with an employee a coffee place, bar, or restaurant to order something to drink.
3. **Cook**: You plan to cook later. You discuss the details with your friend who knows how to cook.
4. **Chat**: You are chatting with your friend.
5. **No Food**: You are chatting with your friend. You do not mention food. Instead, you mention a country name.

We also list three types of conversational acts which could emerge in each scenario:

1. **Question**: Ask a question.
2. **Answer**: Answer a question that could be asked under the current scenario.
3. **Mentioning**: Naturally converse without asking or answering any specific questions.

Using their laptops, users send one text chat for each combination of [scenario, conversational act] to our chatbot, i.e., 15 chats in total. In the Eat, Cook, and Chat scenarios, users must mention one of the foods they listed earlier; in the Drink scenario, they must mention one of the drinks they listed. In the No Food scenario, users must mention one of the countries they listed, and no food names can be mentioned. In total, we collect 150 chat inputs from 10 user experiments.

	Acc (%)	Response Time (s) Mean (Stdev)
1st Only	77.33%	37.14 (14.70)
ESP Only	81.33%	40.95 (13.56)
ESP + 1st	84.00%	40.95 (13.56)
1st Worker Reached Time (s)		30.83 (16.86)
User Type Time (s)		27.05 (25.28)

Table 7.4: Result of User Experiment. A trade-off between time and output quality can be observed.

		1st		ESP + 1st	
		Avg. Time(s)	Acc. (%)	Avg. Time(s)	Acc. (%)
Entity Type	Food[5]	36.64	70.00%	40.19	**78.89%**
	Drink	37.43	80.00%	41.37	**83.33%**
	None	38.33	96.67%	42.83	**100.00%**
Conv. Act	Question	34.26	82.00%	37.94	**90.00%**
	Answer	39.90	68.00%	43.88	**78.00%**
	Mention	37.26	82.00%	41.04	**84.00%**
Avg.		**37.14**	**77.33%**	**40.95**	**84.00%**

Table 7.5: Results of user experiment for each scenario and conversational act.

Correspondingly, instructions on the workers' interface (Figure 7.2) is modified as "What is the `food_name` in this dialog?", and the explanation of `food_name` is modified as "*Food name. The full name of the food. Including any drinks or beverages.*" In the experiments, our chatbot post 120 HITs with a lifetime of 60 seconds to MTurk upon receiving a text chat. The price of each HIT is $0.1. We use the interface shown in Figure 7.2 with a time constraint of 20 seconds.

7.4.3 Experimental Results

Results are shown in Table 7.4. The "ESP+1st" setting achieves the best accuracy of 84% with an average response time of 40.95 seconds. The "1st Only" setting has the shortest average response time of 37.14 seconds with an accuracy of 77.33%.[4] A trade-off between time and output quality can be observed.This trade-off is similar to the results of Experiment 1 (shown in Figure 7.3(c)). On average, 14.45 MTurk workers participated in each trial and submitted 33.81 answers.

[4]We only consider the answers submitted within 60 seconds.
[5]Including the results from Food, Cook, and Chat scenarios.

Robustness in Out-of-Vocabulary Entities & Language Variability

The results over each entity type are shown in Table 7.5. Without using any training data or pre-defined knowledge-base, our crowdsourcing approach achieves an accuracy of 78.89% in extracting food entities and 83.33% in extracting drink entities. Despite the significant variety of the input entities[6], our approach extracts most entities correctly. Furthermore, our method is effective in identifying the absence of entities; Table 7.5 also shows the robustness of the proposed method under various linguistic conditions. The "ESP+1st" setting achieves accuracies of 90.00% in extracting entities from questions, 78.00% in extracting from answers, and 84.00% in extracting from regular conversations. Qualitatively, our approach can handle complex input, such as strange restaurant names and beverage names, which are essentially confusing for automated approaches. For example, "Have you ever tried bibimbap at *Green pepper*?" and "I usually have *Magic Hat #9*", where *Green pepper* and *Magic Hat #9* are names of a restaurant and beverage, respectively.

Error Analysis

Table 7.6 shows the errors in the user experiments ("ESP+1st" setting). 45.83% of errors are caused by absence of answers, mainly due to the task routing latency of the MTurk platform. We discuss this in more detail below. 37.50% of errors are due to various system problems such as the string encoding issues. More interestingly, 12.50% of incorrect answers are sub-spans of the correct answers. For instance, the crowd extracts "rice" for "stew pork over rice", and "tea" for "bubble tea". This type of error is similar to the "Soft Match" error in Experiment 1. Finally, 4.17% of errors are caused by user typos (e.g., *latter* for *latte*), which the crowd tends to exclude in their answers.

Error Type	%
No Answers Received	45.83%
System Problem	37.50%
Substring of a Multi-token Entity	12.50%
Typo	4.17%

Table 7.6: Error Analysis for User Experiment.

Response Speed

Table 7.4 shows the average response time in the user experiment. On average, the first worker takes 30.83 seconds to reach to our Dialog ESP Game, the first answer is received at 37.14 seconds, and the first matched answer occurs at 40.95 seconds. For comparison, we illustrate the timeline of our system in Figure 7.5. In the user experiments, a user on average spends 27.05

[6] The food entities arbitrarily created by our users are quite diverse: From a generic category (e.g., Thai food) to a specific entry (e.g., Magic Hat #9), and from a simple food (e.g., cherry) to a complex food (e.g., sausage muffin with egg). The list covers the food of many other countries (e.g., Okonomiyaki, Bibimbap, Samosa.)

seconds to type a chat line. If we align the user typing time along with the system timeline, the theoretical perceived response time to users falls within 10-14 seconds, while the average response time in instant messaging is 24 seconds [61]. [6] reports that 24.5% of instant messages get responses within 11-30 seconds, and 8.2% of messages have even longer response times. The proposed technology proves to be fast enough to support instant messaging applications. The main bottleneck of the end-to-end response speed is the *task routing time* in Figure 7.5, which approximately ranges from 5-40 seconds and changes over time. The task routing time also causes the major errors in Table 7.6. The task lifetime begins when a task reaches the MTurk server instead of when it becomes visible to workers. When the task routing time is longer than a task's lifetime, the task could expire before it is selected by workers. Because MTurk requesters can not effectively reduce the task routing time, pre-recruiting and queuing workers seems inevitable for applications which require a response time sharply shorter than 30 seconds.

7.5 Discussion

Incorporating domain-specific knowledge is a major obstacle in generalization of crowdsourcing technologies [54]. We think that automation helps resolve this challenge. One most common errors in our system are the *soft match*, where the crowd extracts a sub-string of the target entity instead of the complete string. Domain knowledge can help to fix this type of errors. However, unlike automated technology, we do not have a generic method to update human workers with new knowledge. Thus, our next step is to incorporate automated components. It is easy to replace some workers with automated annotators in our multi-player ESP Game. Despite fragility in extracting unseen entities, automated approaches are robust in identifying known entities and can be easily updated if new data is collected. We will develop a hybrid approach, which we believe will be robust in unexpected input and easily incorporate new knowledge.

7.6 Conclusion

We have explored using real-time crowdsourcing to extract entities for dialog systems. By using an ESP Game setting, our approach is absolute 36.5% and 27.1% better than the CRF baseline in terms of F1-score for Class D and X queries in the ATIS dataset, respectively. The timing cost is about 8 seconds, which is slower than machines but still reasonable given the large gains in accuracy. The proposed method also has been evaluated via Google Hangouts' text chat with 10 users. The results demonstrate the robustness and feasibility of our approach in real-world systems.

Chapter 8

A Crowd-Powered Conversational Assistant that Automates Itself Over Time (Proposed Work)

One approach to robust conversational assistance is to use human computation. Existing systems sometimes use professional employees, such as Facebook M [48], or use non-expert crowd workers, such as Chorus [69]. By leveraging human input, these systems are able to work well across a number of domains. Despite their robustness, problems remain in effectively leveraging such systems, including cost, speed of responses, privacy, and quality variation [57]. This thesis' ultimate goal is to build a robust open-domain conversational assistant. We propose to use Chorus, a conversational agent that is initially powered by the crowd, as a route toward a robust crowd-powered agent that is slowly getting more and more automated.

For automating Chorus, we propose *Evorus*, a framework that is developed to automate Chorus over time. The overview of Evorus is shown in Figure 8.1. Evorus automates itself over time by *(i)* learning to automatically select external dialog systems to provide response, *(ii)* learning to automatically select good response from a set of candidates, and *(iii)* dynamically adjusting workers' workload in Chorus based on the quality (or confidence) of *(i)* and *(ii)*. Evorus allows new dialog systems to be easily incorporated by tasking crowd workers and automated components with managing their integration. The responder selector component chooses which dialog systems will run based on prior data about which responses are accepted. New dialog systems that are added are sampled from occasionally in order to facilitate learning. Crowd workers choose from both crowd and automated suggestions to decide what will be forwarded back to the user, meaning that the cost of incorrect automated suggestions is not an incorrect response to the user, but rather the cost of giving crowd workers another response to choose among. As Evorus becomes more confident, the selection of responses is also be automated over time.

8.1 Learning to Select Responders

Evorus formulates the task of holding a conversation primarily as a sequence of ***responder selection*** problems, *i.e.*, based on the current context of the conversation, to choose the most efficient

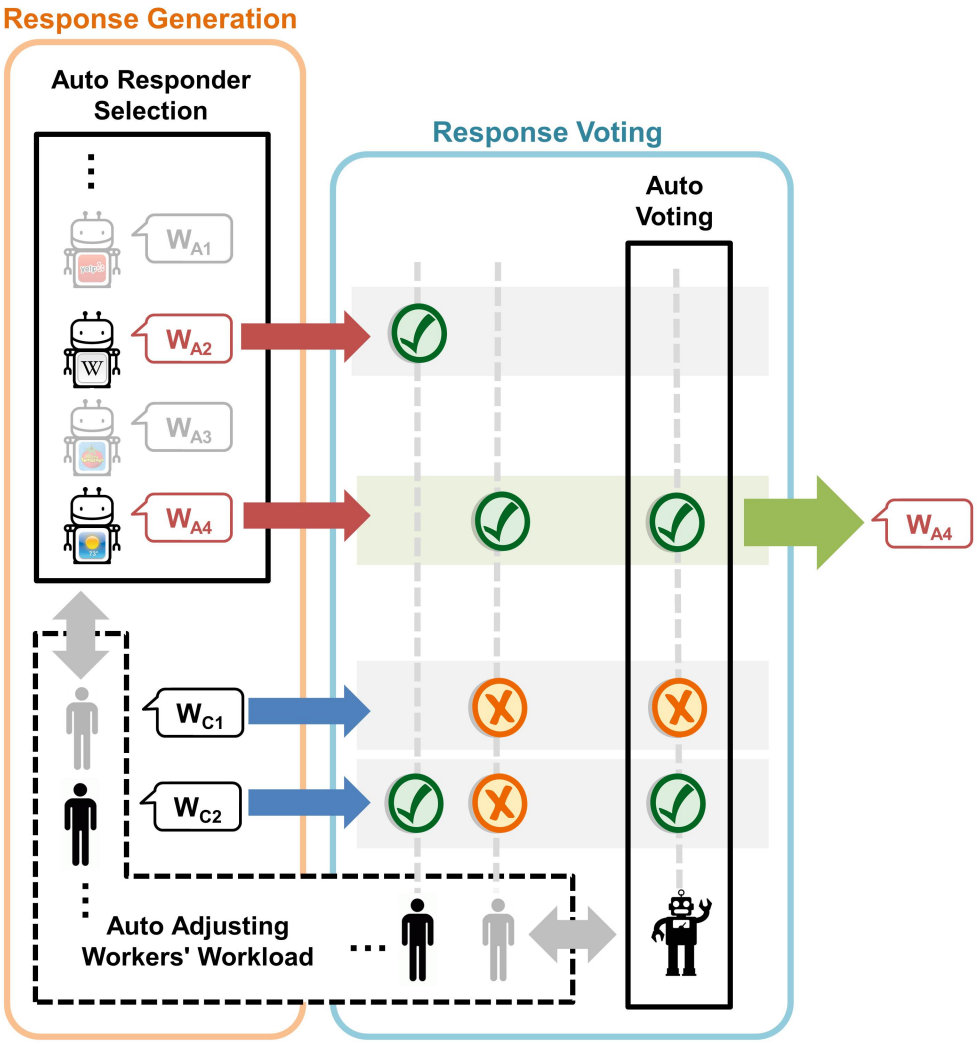

Figure 8.1: The overview of Evorus. Evorus automates itself over time by *(i)* learning to automatically select external dialog systems to provide response, *(ii)* learning to automatically select good response from a set of candidates, and *(iii)* dynamically adjusting workers' workload in Chorus based on the quality (or confidence) of *(i)* and *(ii)*.

responder, either human workers or different automated dialog systems, to obtain responses from. Although some automated responders can be limited or unintelligent, most of them are capable to carry a part of a conversation to a certain extent, as long as they are put into a appropriate circumstances. For instance, a simple "ping-pong" chatbot, which always responds with what it was told, can be useful to reply echo questions such as "Hi," "Hello," or "How are you?"; a restaurant recommendation bot can be used when the user is looking for food; a weather bot, Wikipedia bot, and weather bot can be used for information inquiry; and a chatbot that was trained on the user's friend's chat log can be called when he/she feels lonely [89].

Evorus monitors all ongoing conversations of Chorus, and periodically *select* one of its underlying responders to propose responses. The crowd workers inside Chorus can vote to accept or reject a response (as shown in Figure 8.1). Building on top of the deployed Chorus, Evorus is able to collect conversations along with the crowd-generated feedback (*e.g.,* upvotes and downvotes) over time, and gradually learn to select the best external responder at the right time.

The following features could be useful for Evorus to learn to select responders:

1. The **context** of the current conversation (*e.g.*, the words of prior messages, which turn it is in the conversation, etc.)
2. The history of **previously selected responders** of the current conversation (*e.g.*, which external dialog system was selected in the latest turn?)
3. Number of **upvotes and downvotes** the response received from the crowd.
4. The **responses** generated by the selected responder (*e.g.*, "What kind of food do you want?")
5. The **follow-up message(s)** sent by the user (*e.g.*, "I like Chinese food.")
6. The **response time** of the the selected responder and/or the user.

Evorus' framework is scalable for introducing new responders (automated dialog systems or bot) to the selection process. When the user base grows, Evorus will have more opportunities to obtain crowd labels for selecting responders, and thus more responders can be added into the bot pools (as shown in Figure 1.2.)

We plan to include several automatic responders in Evorus.The followings are the categories of automatic responders we might include, alone with example systems in literature.

1. Task-oriented Dialog Systems
   (a) Crowd-powered dialog systems based on Web API [54]
   (b) Single-domain dialog system
   (c) Multi-domain dialog system [37]
2. Non-task Dialog Systems
   (a) Machine-Translation-based response generators [93]
   (b) Deep-learning-based response generators [78, 99]
   (c) Helper bot that says gap fillers such as "OK" (See Section 8.5)
   (d) Information-Retrieval-based response generators (See Section 8.5)

Existing dialog systems can be incorporated by defining a simple REST interface on top

of them that accepts information about the current conversation content, and responds with a suggested response. Evorus learns over time which automated dialog systems are most likely to be able to offer high-quality suggestions given the context of the conversation, queries them, and then those responses are forwarded to crowd workers as another suggestion.

8.2 Learning to Select Responses

After all automated responders and crowd workers send their responses, the next step is *response voting,* which Evorus can also learn to automate over time. In the crowd-powered version of Chorus, a group of crowd workers upvote and downvote the response proposed be each other to decide which response to send back to the user. The uniqueness of Chorus' data is that it contains upvotes and downvotes of all proposed responses, which provide a detailed assessment of the quality of each response in a conversation. Evorus can learn to automatically upvote and downvote a response by using the following features:

1. The **context** of the current conversation (*e.g.,* the words of prior messages, which turn it is in the conversation, etc.)
2. The **responses** content
3. The mete data of the **responder** (*e.g.,* from an automated bot or a human, from a high-quality or low-quality responder, the description of the responder, etc.)
4. **Other accepted or rejected responses** within in the same turn (*e.g.,* duplicate responses are more likely to be rejected.)

For instance, in a standard supervised learning setup, `upvote`, `downvote`, and `no-vote` can be formulated as three class labels and classic classification algorithms such as Support Vector Machine can be applied.

8.3 Dynamically Adjusting Crowd Workers' Workload

To close the loop of automating Chorus, Evorus will adjust the workload of the crowd and of the automated components dynamically, based on the in-the-moment performance or confidence of the automated approaches. Automated components such as response selector and response voter will learn and become more reliable over time, and thus more responsibility can be moved to them from crowd workers gradually.

For response generation, since Chorus constraints the maximal number of responses it can send back in a single turn, the automated responders and human workers are put in a race condition. Therefore, in the early stage, Evorus does not need to actively block workers from sending responses, the limited responses quota will naturally yield some workload from workers to the automated approaches. For response voting, we can first tune the model that it has a high precision and low recall in predicting votes, and thus the good responses need fewer human votes to get accepted. As a consequence, similar to the case of response generation, human workers will have less chances to contribute their votes. In the later stage of Evorus, when the automated

approaches are more reliable, we can then dynamically reduce the number of workers recruited in Chorus based on the performance of automated approaches.

8.4 Never-Ending Learning

To deliver the commitment of improving Chorus over time, we will configure the system that all the automated models will re-train itself by adding the newly-collected data periodically. For instance, a supervised response voting model can be trained each time when a hundred more upvotes and downvotes are collected via the deployed Chorus.

8.5 Pilot Study

To understand the basic statistics and performance of future Evorus, we conducted a pilot study by using three simple automated responders.

8.5.1 Automatic Responders

For pilot study, we implemented the following three simple response generators to automatically propose responses.

1. **Helper Bot:** A chatbot that randomly selects one response from a set of candidates, regardless of context. We manually selected 13 responses that frequently sent by Chorus (*e.g.*, "Can you provide some more details?", "Is there anything else I can help you with?", "Sure. Wait a second...", or "Thanks") to form the candidate pool.
2. **Memory Bot:** A chatbot that looks into all the dialogs that deployed Chorus had during its deployment to find the best response. The detailed steps are as follows: 1) beforehand, the underlying search engine indexed the good *user-crowd message pairs* extracted from all previous Chorus dialogs, which we describe later, and 2) the chatbot takes the first *user* message in the *latest dialog turn* of the current conversation, and 3) searches the most similar k *user messages* in Chorus' history via the search engine, along with the corresponding k *crowd responses*, and finally, 4) randomly selects one from the top k crowd responses to send.
3. **Q&A Bot:** A chatbot that looks into a question-answer pair dataset that were extracted from transcripts of Cable News Network (CNN) interviews to find the best response. The process are similar to the Memory Bot, except that we replaced the Chorus' dialog data in step 1) with the CNN question-answer pair dataset, which we describe later.

The overview of these three responder is shown in Table 8.1. The Memory Bot and Q&A Bot were powered by information retrieval technologies, particularly Elasticsearch[1], a Lucene-based search engine. The default indexing, querying, and ranking settings of Elasticsearch were used.

The Memory Bot looked for good responses in the previous dialogs that the deployed Chorus had since May 20th, 2016. For each dialog turn between a user and Chorus, we indexed the

[1]Elasticsearch: https://en.wikipedia.org/wiki/Elasticsearch

Bot	Data Used	Data Size	Generation Method	Context-aware
Q&A	CNN interviews	∼33k	IR-based	Yes
Memory	Chorus' dialogs	∼3k	IR-based	Yes
Helper	Common responses	∼10	Random	No

Table 8.1: Response generators used in Evorus. The Helper Bot randomly selects one response from a set of candidates. The Memory Bot and Q&A Bot were powered by information retrieval technologies that searches for best responses according to the chat history.

first user message and the following crowd response with the highest value of $(1 \times \#upvote - 0.5 \times \#downvote)$. Note that only the crowd responses that had at least one upvote from other crowd workers were used. Up to date (Oct 22th, 2016), 3,044 crowd messages, along with the user messages they responded to, were extracted from totally 918 conversations and indexed by Elasticsearch. Considering the size of data, we empirically set $k = 1$.

The Q&A Bot searched in the question-answer pairs extracted from CNN's interviews to find good responses. The dataset were developed as follows: First, we downloaded 767 interviews (*e.g.,* "Piers Morgan Tonight") from the CNN's Transcripts website[2]. Each transcript contains 500 to 1,000 sentences, and each line was annotated with the speaker's name. Second, we used the Stanford CoreNLP tool [83] to segment sentences in the transcripts, and used manually-crafted linguistic patterns such as "How," "Wh-," and question marks ("?") to identify question sentences. Finally, if a question's direct follow-up sentence were of a different speaker, we extracted this question-response pair to include in the dataset. In our study, totally 33,033 question-answer(response) pairs were indexed, and we set $k = 3$ to adapt the larger data size.

Evorus monitored all ongoing dialog turns that have 1) less than 2 crowd messages accepted and 2) less than 5 crowd messages proposed, and automatically proposed responses to these turns. Evorus can propose at most 2 responses for each dialog turn. For proposing each response, Evorus first randomly selected one bot from the three, and then fed the current chat log to the selected bot to obtain response. If the responder failed to provide any responses, short sentence "Hi." was utilized as the default output. Evorus disguised itself as one of the crowd workers so that actual crowd workers in Chorus can not tell which responses were proposed by automated response generators.

8.5.2 Results

In this section we reported the results recorded from the midnight of October 19th, 2016 to October 24th, 2016. Within this period of time, 20 users used the system during 36 conversational sessions, in which 348 messages were sent by users, 371 crowd messages were accepted, and 41 messages proposed by bots were accepted. Each conversation has an average of 21.11 messages.

We calculated the percentage that the responses proposed by bots were accepted in Chorus, the results are shown in Table 8.2. On average, 24.70% of responses proposed by automated

[2]CNN's Transcripts: http://transcripts.cnn.com/TRANSCRIPTS/

Responder	#Accepted	#Total	%Accepted
Q&A Bot	6	48	**12.50%**
Memory Bot	14	63	**22.22%**
Helper Bot	21	55	**38.18%**
Auto Bots Total	41	166	**24.70%**
Human	375	517	**71.76%**

Table 8.2: A significant number of the automatic responses were chosen, even though the acceptance rates for automated responders were lower. This results in lower cost and lower latency.

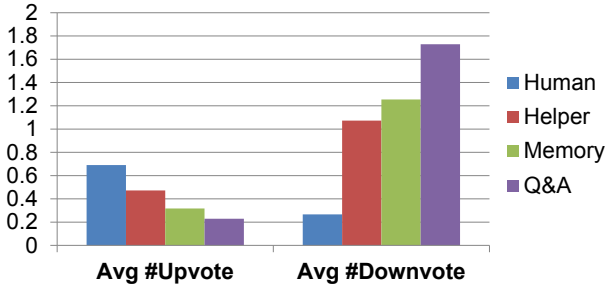

Figure 8.2: Average number of upvotes and downvotes of one response proposed by different responders.

bots were selected by the crowd, while 71.76% of human workers' responses went through Chorus. Surprisingly, the Helper Bot has the highest acceptance rate amongst all automated responders. 38.18% of Helper Bot's responses were selected, while only 22.22% of Memory Bot's and 12.50% of Q&A Bots responses were picked by the crowd. We also analyzed the average number of upvotes and downvotes each proposed response got from other workers, the results are shown in Figure 8.2. We found that human worker's responses got significantly less downvotes than that of automated bots, while humans only have a small lead on upvotes.

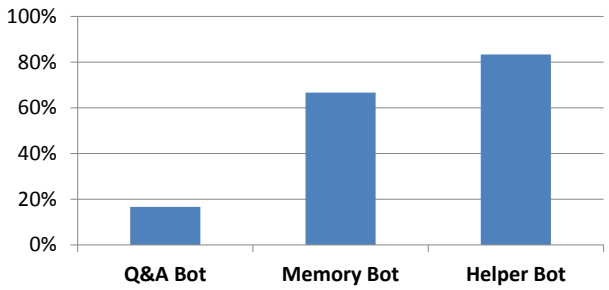

Figure 8.3: On-topic rate of the responses that were selected by the crowd. The reliability of an automated responder does not only influence the chances that its responses being accepted, but also the quality of its responses even if they have been selected by the crowd.

To further understand the quality of automatic responses that were selected by the crowd, we followed the evaluation process used by Lasecki et al. that to manually annotate if each bots

response is *on-topic* [69]. Two researchers manually annotated all responses that were proposed by automated responders and selected by the crowd, respectively. The inter-annotator agreement is substantial ($\kappa = 0.769$.) 66.67% of bots' response were considered on-topic by at least one annotator. We also analyzed the on-topic rate of each bot's accepted response, as shown in Figure 8.3. The results suggested that the reliability of an automated responder does not only influence the chances that its responses being accepted, but also the quality of its responses even if they have been selected by the crowd.

Appendix A

Timeline

The timeline for this thesis will primarily be organized around paper submission deadlines, and the overall goal is to complete this thesis within a year (defending in December 2017).

- **January - March 2017:** Automating response voting (Target: HCOMP 2017)
- **March - May 2017:** Automating responder selection (Target: HCOMP 2017)
- **May - September 2017:** Automating dynamic workload assignment (Target: CHI 2018)
- **September - December 2017:** Chorus Dataset (Target: ACL or NAACL 2018)
- **September - December 2017:** Thesis writing
- **Spring 2018:** Thesis Defense

Appendix B

List of Food and Drink Entities Used in the Experiment 2 of Chapter 7

The followings are the lists of 9 food and 3 drinks created by 10 participants in the Experiment 2 of Chapter 7.

B.1 Food

1. spaghetti, burger, vindaloo lamb, makhani chicken, kimchee, wheat bread, pizza, cornish pasty, mushroom soup
2. burger, french fries, scallion cake, okonomiyaki, oyakodon, gyudon, fried rice, wings, salad
3. Stinky Tofu, Acai Berry Bowl, Tuna Onigiri, Rice Burger, Seared Salmon, Milkfish Soup, Mapo Tofu, Beef Pho, Scallion Pancake
4. pizza, fried rice, waffle, alcohol drink, chocolate pie, cookie, dimsum, burger, milk shake
5. Pho, BBQ, Thai food, beef noodles, steak, Tomato soup, Spicy hot pot, Soup dumplings, Ramen
6. chocolate, donut, cheesecake, pad thai, seafood pancake, fish fillets in hot chili, hot pot, bibimbap, japchae
7. chocolate, pancakes, strawberries, fried fish, fried chicken, sausages, gulaab jamun, paneer tika, samosa
8. Dumplings, noodle, stew pork over rice, Sandwich, pasta, hot pot, Potato slices with green peppers, Chinese BBQ, pancakes
9. stinky tofu, stew pork over rice, yakitori, baked cinnamon apple, apple pie, stew pork with potato and apple, teppanyaki, okonomiyaki, crab hotpot
10. hot pot, cherry, Chinese cabbage, Pumpkin risotto, Tomato risotto, Boeuf Bourguignon, stinky tofu, sausage muffin with egg (McDonald), eggplant with basil

B.2 Drink

1. tea, coke, latte
2. green tea latte, bubble tea, root beer
3. medium latte with non-fat milk, green Tea Latte, Soymilk
4. water, pepsi, tea
5. Latte with nonfat milk, Magic hat #9, Old fashion
6. vanilla latte, strawberry smoothie, iced tea
7. coffee, milk shake, beer
8. Mocha coffee, beers, orange juice
9. caramel frappuccino, caramel macchiato, coffee with coconut milk
10. ice tea, macha, apple juice

Bibliography

[1] James Allen, Nate Blaylock, and George Ferguson. A problem solving model for collaborative agents. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, AAMAS '02, pages 774–781, New York, NY, USA, 2002. ACM. ISBN 1-58113-480-0. doi: 10.1145/544862.544923. URL http://doi.acm.org/10.1145/544862.544923. 2.1

[2] James F. Allen, Lenhart K. Schubert, George Ferguson, Peter Heeman, Chung Hee Hwang, Tsuneaki Kato, Marc Light, Nathaniel G. Martin, Bradford W. Miller, Massimo Poesio, and David R. Traum. Enriching speech recognition with automatic detection of sentence boundaries and disfluencies. *Journal of Experimental and Theoretical AI (JETAI)*, 7:7–48, 1995. 1.2

[3] James F Allen, Donna K Byron, Myroslava Dzikovska, George Ferguson, Lucian Galescu, and Amanda Stent. Toward conversational human-computer interaction. *AI magazine*, 22 (4):27, 2001. 4.2.2

[4] Saleema Amershi and Meredith Ringel Morris. Cosearch: A system for co-located collaborative web search. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1647–1656, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-011-1. doi: 10.1145/1357054.1357311. URL http://doi.acm.org/10.1145/1357054.1357311. 2.3

[5] Rafael E Banchs and Haizhou Li. Iris: a chat-oriented dialogue system based on the vector space model. In *Proceedings of the ACL 2012 System Demonstrations*, pages 37–42. Association for Computational Linguistics, 2012. 1.1

[6] Naomi S Baron. Discourse structures in instant messaging: The case of utterance breaks. *Language Internet*, 7(4):1–32, 2010. 4.7, 7.4.3

[7] Michael S. Bernstein, Greg Little, Robert C. Miller, Björn Hartmann, Mark S. Ackerman, David R. Karger, David Crowell, and Katrina Panovich. Soylent: a word processor with a crowd inside. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 313–322, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0271-5. doi: http://doi.acm.org/10.1145/1866029.1866078. URL http://doi.acm.org/10.1145/1866029.1866078. 2.2, 4.1

[8] Michael S. Bernstein, Joel R. Brandt, Robert C. Miller, and David R. Karger. Crowds in two seconds: Enabling realtime crowd-powered interfaces. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST '11, page

to appear, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0271-5. doi: http://doi.acm.org/10.1145/1866029.1866080. URL http://doi.acm.org/10.1145/1866029.1866080. 2.2, 4.3.2, 6.3.2, 7.2, 7.4.1

[9] Michael S. Bernstein, David R. Karger, Robert C. Miller, and Joel R. Brandt. Analytic methods for optimizing realtime crowdsourcing. In *Proceedings of Collective Intelligence*, CI 2012, page to appear, New York, NY, USA, 2012. 2.2

[10] Michael S. Bernstein, Jaime Teevan, Susan Dumais, Daniel Liebling, and Eric Horvitz. Direct answers for search queries in the long tail. In *Proceedings of the conference on Human Factors in Computing Systems*, CHI '12, pages 237–246, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1015-4. doi: 10.1145/2207676.2207710. URL http://doi.acm.org/10.1145/2207676.2207710. 2.3

[11] Jeffrey P. Bigham, Anna C. Cavender, Ryan S. Kaminsky, Craig M. Prince, and Tyler S. Robison. Transcendence: Enabling a personal view of the deep web. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, IUI '08, pages 169–178, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-987-6. doi: 10.1145/1378773.1378796. URL http://doi.acm.org/10.1145/1378773.1378796. 6.2

[12] Jeffrey P. Bigham, Tessa Lau, and Jeffrey Nichols. Trailblazer: enabling blind users to blaze trails through the web. In *Proceedings of the 14th international conference on Intelligent user interfaces*, IUI '09, pages 177–186, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-168-2. doi: http://doi.acm.org/10.1145/1502650.1502677. URL http://doi.acm.org/10.1145/1502650.1502677. 3.2.1

[13] Jeffrey P. Bigham, Chandrika Jayant, Hanjie Ji, Greg Little, Andrew Miller, Robert C. Miller, Robin Miller, Aubrey Tatarowicz, Brandyn White, Samual White, and Tom Yeh. Vizwiz: nearly real-time answers to visual questions. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 333–342, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0271-5. doi: http://doi.acm.org/10.1145/1866029.1866080. URL http://doi.acm.org/10.1145/1866029.1866080. 1.2, 2.2, 2.3, 4.2.1, 7.2, 7.4.1

[14] Jeffrey P. Bigham, Richard E. Ladner, and Yevgen Borodin. The design of human-powered access technology. In *Proceedings of the 2011 SIGACCESS Conference on Computers and Accessibility (ASSETS 2011)*, ASSETS 2011, page To Appear, New York, NY, USA, 2011. ACM. 2.2

[15] Daniel G Bobrow, Ronald M Kaplan, Martin Kay, Donald A Norman, Henry Thompson, and Terry Winograd. Gus, a frame-driven dialog system. *Artificial intelligence*, 8(2): 155–173, 1977. 6.2

[16] Dan Bohus and Alexander I. Rudnicky. Ravenclaw: dialog management using hierarchical task decomposition and an expectation agenda. In *INTERSPEECH*. ISCA, 2003. URL http://dblp.uni-trier.de/db/conf/interspeech/interspeech2003.html#BohusR03. 2.1

[17] Dan Bohus and Alexander I. Rudnicky. The ravenclaw dialog management framework: Architecture and systems. *Comput. Speech Lang.*, 23(3):332–361, July 2009. ISSN 0885-

2308. doi: 10.1016/j.csl.2008.10.001. URL http://dx.doi.org/10.1016/j.csl.2008.10.001. 2.1, 4.2.2

[18] Dan Bohus, Sergio Grau Puerto, David Huggins-Daines, Venkatesh Keri, Gopala Krishna, Rohit Kumar, Antoine Raux, and Stefanie Tomko. 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*, pages 9–12. Association for Computational Linguistics, 2007. 6.1

[19] Dan Bohus, Antoine Raux, Thomas K Harris, Maxine Eskenazi, and Alexander I Rudnicky. Olympus: an open-source framework for conversational spoken language interface research. In *Proceedings of the workshop on bridging the gap: Academic and industrial research in dialog technologies*, pages 32–39. Association for Computational Linguistics, 2007. 7.1

[20] Cristiana Bolchini, Carlo A. Curino, Elisa Quintarelli, Fabio A. Schreiber, and Letizia Tanca. A data-oriented survey of context models. *SIGMOD Rec.*, 36(4):19–26, December 2007. ISSN 0163-5808. doi: 10.1145/1361348.1361353. URL http://doi.acm.org/10.1145/1361348.1361353. 3.2.1

[21] Marco Brambilla, Piero Fraternali, and Carmen Karina Vaca Ruiz. Combining social web and bpm for improving enterprise performances: the bpm4people approach to social bpm. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 223–226. ACM, 2012. 3.2.1

[22] Jeppe Bronsted, Klaus Marius Hansen, and Mads Ingstrup. Service composition issues in pervasive computing. *IEEE Pervasive Computing*, 9(1):62–70, 2010. 3.2.1

[23] A.J. Bernheim Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu, and Colin Dixon. Home automation in the wild: Challenges and opportunities. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2115–2124, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0228-9. doi: 10.1145/1978942.1979249. URL http://doi.acm.org/10.1145/1978942.1979249. 3.2.1

[24] Joyce Chai, Veronika Horvath, Nicolas Nicolov, Margo Stys, Nanda Kambhatla, Wlodek Zadrozny, and Prem Melville. Natural language assistant: A dialog system for online product recommendation. *AI Magazine*, 23(2):63, 2002. 4.2.2

[25] Brian X. Chen. Siri, alexa and other virtual assistants put to the test. *The New York Times*, jan 2016. Retrieved from: http://www.nytimes.com/2016/01/28/technology/personaltech/siri-alexa-and-other-virtual-assistants-put-to-the-test.html?_r=0. 1.2

[26] Yun-Nung Chen, William Yang Wang, and Alexander I. Rudnicky. 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)*, pages 120–125, Olomouc, Czech, 2013. IEEE. 6.6.2

[27] Yun-Nung Chen, Dilek Hakkani-Tür, and Gokhan Tur. Deriving local relational surface forms from dependency-based entity embeddings for unsupervised spoken language un-

derstanding. *Proceedings of SLT*, 2014. 7.1

[28] Lydia Chilton. Seaweed: A web application for designing economic games. Master's thesis, MIT, 2009. 2.2

[29] Seth Cooper, Firas Khatib, Adrien Treuille, Janos Barbero, Jeehyung Lee, Michael Beenen, Andrew Leaver-Fay, David Baker, Zoran Popovic, and Foldit Players. Predicting protein structures with a multiplayer online game. *Nature*, 466(7307):756–760, 2010. 2.2

[30] Deborah A Dahl, Madeleine Bates, Michael Brown, William Fisher, Kate Hunicke-Smith, David Pallett, Christine Pao, Alexander Rudnicky, and Elizabeth Shriberg. Expanding the scope of the atis task: The atis-3 corpus. In *HLT*, pages 43–48. Association for Computational Linguistics, 1994. 7.3.2

[31] Yngve Dahl and Reidar-Martin Svendsen. End-user composition interfaces for smart environments: A preliminary study of usability factors. In *International Conference of Design, User Experience, and Usability*, pages 118–127. Springer, 2011. 3.2.1

[32] Florian Daniel, Muhammad Imran, Stefano Soi, AD Angeli, Christopher R Wilkinson, Fabio Casati, and Maurizio Marchese. Developing mashup tools for end-users: on the importance of the application domain. *Int. J. Next-Generat. Comput*, 3(2), 2012. 3.1, 3.2.1

[33] Luigi De Russis and Fulvio Corno. Homerules: A tangible end-user programming interface for smart homes. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '15, pages 2109–2114, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3146-3. doi: 10.1145/2702613.2732795. URL `http://doi.acm.org/10.1145/2702613.2732795`. 3.2.1

[34] Anind K Dey, Timothy Sohn, Sara Streng, and Justin Kodama. icap: Interactive prototyping of context-aware applications. In *International Conference on Pervasive Computing*, pages 254–271. Springer, 2006. 3.2.1

[35] Djellel Eddine Difallah, Gianluca Demartini, and Philippe Cudré-Mauroux. Mechanical cheat: Spamming schemes and adversarial techniques on crowdsourcing platforms. In *CrowdSearch*, pages 26–30, 2012. 4.6

[36] George Ferguson, James F Allen, et al. Trips: An integrated intelligent problem-solving assistant. In *AAAI/IAAI*, pages 567–572, 1998. 6.2

[37] M Gašić, Dongho Kim, Pirros Tsiakoulis, and Steve Young. Distributed dialogue policies for multi-domain statistical dialogue management. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5371–5375. IEEE, 2015. 1c

[38] Giuseppe Ghiani, Marco Manca, and Fabio Paternò. Authoring context-dependent cross-device user interfaces based on trigger/action rules. In *Proceedings of the 14th International Conference on Mobile and Ubiquitous Multimedia*, MUM '15, pages 313–322, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3605-5. doi: 10.1145/2836041.2836073. URL `http://doi.acm.org/10.1145/2836041.2836073`. 3.2.1

[39] Saul Greenberg and David Marwood. Real time groupware as a distributed system: con-

currency control and its effect on the interface. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, CSCW '94, pages 207–217, New York, NY, USA, 1994. ACM. ISBN 0-89791-689-1. doi: http://doi.acm.org/10.1145/192844.193011. URL http://doi.acm.org/10.1145/192844.193011. 2.3

[40] Narendra Gupta, Gokhan Tur, Dilek Hakkani-Tur, Srinivas Bangalore, Giuseppe Riccardi, and Mazin Gilbert. The at&t spoken language understanding system. *Audio, Speech, and Language Processing, IEEE Transactions on*, 14(1):213–222, 2006. 4.2.2

[41] Nathan Hahn, Joseph Chang, Ji Eun Kim, and Aniket Kittur. The knowledge accelerator: Big picture thinking in small pieces. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 2258–2270, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858364. URL http://doi.acm.org/10.1145/2858036.2858364. 2.3

[42] Jonna Häkkilä, Panu Korpipää, Sami Ronkainen, and Urpo Tuomela. Interaction and end-user programming with a context-aware mobile application. In *IFIP Conference on Human-Computer Interaction*, pages 927–937. Springer, 2005. 3.2.1

[43] Eric N Hanson and Jennifer Widom. An overview of production rules in database systems. *The Knowledge Engineering Review*, 8(02):121–143, 1993. 3.2.1

[44] Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R. Klemmer. 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, pages 241–250, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-679-0. doi: 10.1145/1294211.1294254. URL http://doi.acm.org/10.1145/1294211.1294254. 6.2

[45] Yulan He and Steve Young. A data-driven spoken language understanding system. In *ASRU'03*, pages 583–588. IEEE, 2003. 7.3.2

[46] Alice F Healy. Detection errors on the word the: Evidence for reading units larger than letters. *Journal of Experimental Psychology: Human Perception and Performance*, 2(2): 235, 1976. 7.3.4

[47] Larry P Heck, Dilek Hakkani-Tür, and Gökhan Tür. Leveraging knowledge graphs for web-scale unsupervised semantic parsing. In *INTERSPEECH*, pages 1594–1598, 2013. 7.1

[48] Jessi Hempel. Facebook launches m, its bold answer to siri and cortana, August 2015. URL https://www.wired.com/2015/08/facebook-launches-m-new-kind-virtual-assistant. [Online; posted 26-August-2015. Retrieved from: https://www.wired.com/2015/08/facebook-launches-m-new-kind-virtual-assistant]. 2.3, 8

[49] Matthew Henderson, Milica Gašić, Blaise Thomson, Pirros Tsiakoulis, Kai Yu, and Steve Young. Discriminative Spoken Language Understanding Using Word Confusion Networks. In *Spoken Language Technology Workshop, 2012. IEEE*, 2012. 6.6.2

[50] Lynette Hirschman. Multi-site data collection for a spoken language corpus. In *Proceedings of the Workshop on Speech and Natural Language*, HLT '91, pages 7–14, Strouds-

burg, PA, USA, 1992. Association for Computational Linguistics. ISBN 1-55860-272-0. doi: 10.3115/1075527.1075531. URL http://dx.doi.org/10.3115/1075527.1075531. 7.3.1

[51] Eric Horvitz and Tim Paek. A computational architecture for conversation. *COURSES AND LECTURES-INTERNATIONAL CENTRE FOR MECHANICAL SCIENCES*, pages 201–210, 1999. 6.2

[52] Justin Huang and Maya Cakmak. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 215–225, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3574-4. doi: 10.1145/2750858.2805830. URL http://doi.acm.org/10.1145/2750858.2805830. 3.1, 3.2.1, 3.4.1

[53] Ting-Hao K Huang, Walter S Lasecki, Alan L Ritter, and Jeffrey P Bigham. Combining non-expert and expert crowd work to convert web apis to dialog systems. In *Second AAAI Conference on Human Computation and Crowdsourcing (Demo Paper)*, 2014. 6.1

[54] Ting-Hao K. Huang, Walter S. Lasecki, and Jeffrey P. Bigham. Guardian: A crowd-powered spoken dialog system for web apis. In Elizabeth Gerber and Panos Ipeirotis, editors, *Proceedings of the Third AAAI Conference on Human Computation and Crowdsourcing, HCOMP 2015, November 8-11, 2015, San Diego, California.*, pages 62–71. AAAI Press, 2015. ISBN 978-1-57735-741-4. URL http://www.aaai.org/ocs/index.php/HCOMP/HCOMP15/paper/view/11599. 3.2.2, 4.1, 4.1, 6.1, 7.1, 7.5, 1a

[55] Ting-Hao K. Huang, Francis Ferraro, Nasrin Mostafazadeh, Ishan Misra, Aishwarya Agrawal, Jacob Devlin, Ross Girshick, Xiaodong He, Pushmeet Kohli, Dhruv Batra, et al. Visual storytelling. In *Proc. the 15th Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL 2016)*. NAACL, 2016. 2.2

[56] Ting-Hao Kenneth Huang, Amos Azaria, and Jeffrey P Bigham. Instructablecrowd: Creating if-then rules via conversations with the crowd. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 1555–1562. ACM, 2016. 4.1

[57] Ting-Hao Kenneth Huang, Walter S. Lasecki, Amos Azaria, and Jeffrey P. Bigham. "is there anything else i can help you with?": Challenges in deploying an on-demand crowd-powered conversational agent. In *Proceedings of AAAI Conference on Human Computation and Crowdsourcing 2016 (HCOMP 2016)*. AAAI, 2016. 1.1, 3.2.2, 4.1, 8

[58] 2016 Everyone IFTTT September 10. If by ifttt - android apps on google play, Oct 2016. URL https://play.google.com/store/apps/details?id=com.ifttt.ifttt. 3.1

[59] Albrecht Werner Inhoff and Keith Rayner. Parafoveal word processing during eye fixations in reading: Effects of word frequency. *Perception & Psychophysics*, 40(6):431–439, 1986. 7.3.4

[60] Panagiotis G. Ipeirotis, Foster Provost, and Jing Wang. Quality management on amazon mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computa-*

tion, HCOMP '10, pages 64–67, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0222-7. doi: 10.1145/1837885.1837906. URL http://doi.acm.org/10.1145/1837885.1837906. 4.6

[61] Ellen Isaacs, Alan Walendowski, Steve Whittaker, Diane J Schiano, and Candace Kamm. The character, functions, and styles of instant messaging in the workplace. In *Proceedings of the 2002 ACM CSCW*, pages 11–20. ACM, 2002. 4.7, 7.4, 7.4.3

[62] Juan Jara, Florian Daniel, Fabio Casati, and Maurizio Marchese. From a simple flow to social applications. In *Current Trends in Web Engineering*, pages 39–50. Springer, 2013. 3.2.1

[63] A. Kittur, B. Smus, and R.E. Kraut. Crowdforge: Crowdsourcing complex work. Technical Report CMUHCII-11-100, Carnegie Mellon University, 2011. 2.2

[64] Nadin Kokciyan, Suzan Uskudarli, and TB Dinesh. User generated human computation applications. In *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom)*, pages 593–598. IEEE, 2012. 3.2.1

[65] Walter Lasecki, Christopher Miller, Adam Sadilek, Andrew Abumoussa, Donato Borrello, Raja Kushalnagar, and Jeffrey Bigham. Real-time captioning by groups of non-experts. In *Proceedings of the 25th UIST*, pages 23–34. ACM, 2012. 2.2, 4.1

[66] Walter S Lasecki, Kyle I Murray, Samuel White, Robert C Miller, and Jeffrey P Bigham. Real-time crowd control of existing interfaces. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 23–32. ACM, 2011. 1.2, 2.2

[67] Walter S. Lasecki, Ece Kamar, and Dan Bohus. Conversations in the crowd: Collecting data for task-oriented dialog learning. In *HCOMP*, 2013. 4.1, 7.1

[68] Walter S. Lasecki, Phyo Thiha, Yu Zhong, Erin Brady, and Jeffrey P. Bigham. Answering visual questions with conversational crowd assistants. In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '13, pages 18:1–18:8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2405-2. doi: 10.1145/2513383.2517033. URL http://doi.acm.org/10.1145/2513383.2517033. 4.2.1

[69] Walter S. Lasecki, Rachel Wesley, Jeffrey Nichols, Anand Kulkarni, James F. Allen, and Jeffrey P. Bigham. Chorus: A crowd-powered conversational assistant. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 151–162, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2268-3. doi: 10.1145/2501988.2502057. URL http://doi.acm.org/10.1145/2501988.2502057. 1.1, 1.2, 3.2.2, 4.1, 4.1, 4.2.2, 4.3, 4.3.1, 4.4, 6.5.2, 7.2, 7.4.1, 8, 8.5.2

[70] Walter S Lasecki, Mitchell Gordon, Danai Koutra, Malte F Jung, Steven P Dow, and Jeffrey P Bigham. Glance: Rapidly coding behavioral video with the crowd. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 551–562. ACM, 2014. 4.1

[71] Walter S Lasecki, Christopher Homan, and Jeffrey P Bigham. Architecting real-time crowd-powered systems. *Human Computation*, 1(1), 2014. 2.2

[72] Walter S Lasecki, Jaime Teevan, and Ece Kamar. Information extraction and manipulation threats in crowd-powered systems. In *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*, pages 248–256. ACM, 2014. 4.6

[73] Walter S Lasecki, Mitchell Gordon, Winnie Leung, Ellen Lim, Jeffrey P Bigham, and Steven P Dow. Exploring privacy and accuracy trade-offs in crowdsourced behavioral video coding. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1945–1954. ACM, 2015. 3.5.3

[74] Thomas D LaToza and Andre van der Hoek. Crowdsourcing in software engineering: Models, motivations, and challenges. *IEEE Software*, 33(1):74–80, 2016. 3.2.1

[75] Tessa Lau, Julian Cerruti, Guillermo Manzato, Mateo Bengualid, Jeffrey P. Bigham, and Jeffrey Nichols. A conversational interface to web automation. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 229–238, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0271-5. doi: http://doi.acm.org/10.1145/1866029.1866067. URL `http://doi.acm.org/10.1145/1866029.1866067`. 3.2.1

[76] Cheongjae Lee, Sangkeun Jung, Seokhwan Kim, and Gary Geunbae Lee. Example-based dialog modeling for practical multi-domain dialog system. *Speech Communication*, 51 (5):466–484, 2009. 6.5.2

[77] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. Coscripter: Automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1719–1728, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-011-1. doi: 10.1145/1357054.1357323. URL `http://doi.acm.org/10.1145/1357054.1357323`. 3.2.1

[78] Jiwei Li, Will Monroe, Alan Ritter, and Dan Jurafsky. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541*, 2016. 2b

[79] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. *End-user development: An emerging paradigm*. Springer, 2006. 3.2.1

[80] Diane J Litman and Scott Silliman. Itspoke: An intelligent tutoring spoken dialogue system. In *Demonstration Papers at HLT-NAACL 2004*, pages 5–8. Association for Computational Linguistics, 2004. 6.2

[81] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Turkit: human computation algorithms on mechanical turk. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 57–66, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0271-5. doi: http://doi.acm.org/10.1145/1866029.1866040. URL `http://doi.acm.org/10.1145/1866029.1866040`. 2.2

[82] W. E. Mackay, T. W. Malone, K. Crowston, R. Rao, D. Rosenblitt, and S. K. Card. How do experienced information lens users use rules? In *Proceedings of the SIGCHI Conference*

on Human Factors in Computing Systems, CHI '89, pages 211–216, New York, NY, USA, 1989. ACM. ISBN 0-89791-301-9. doi: 10.1145/67449.67491. URL http://doi.acm.org/10.1145/67449.67491. 3.2.1

[83] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014. URL http://www.aclweb.org/anthology/P/P14/P14-5010. 8.5.1

[84] David Maulsby, Saul Greenberg, and Richard Mander. Prototyping an intelligent agent through wizard of oz. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 277–284. ACM, 1993. 6.2

[85] Michael F. McTear. Spoken dialogue technology: enabling the conversational user interface. *ACM Comput. Surv.*, 34(1):90–169, March 2002. ISSN 0360-0300. doi: 10.1145/505282.505285. URL http://doi.acm.org/10.1145/505282.505285. 2.1

[86] Grégoire Mesnil, Yann Dauphin, Kaisheng Yao, Yoshua Bengio, Li Deng, Dilek Hakkani-Tur, Xiaodong He, Larry Heck, Gokhan Tur, Dong Yu, et al. Using recurrent neural networks for slot filling in spoken language understanding. *Audio, Speech, and Language Processing, IEEE/ACM Transactions on*, 23(3):530–539, 2015. 7.1, 7.3.2

[87] Meredith Ringel Morris and Eric Horvitz. Searchtogether: an interface for collaborative web search. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST '07, pages 3–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-679-0. doi: 10.1145/1294211.1294215. URL http://doi.acm.org/10.1145/1294211.1294215. 2.3

[88] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. Shop2: an htn planning system. *J. Artif. Int. Res.*, 20(1):379–404, December 2003. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=1622452.1622465. 2.1

[89] Casey Newton. *SPEAK, MEMORY: When her best friend died, she rebuilt him using artificial intelligence*, 2016 (accessed October 24th, 2016). URL http://www.theverge.com/a/luka-artificial-intelligence-memorial-roman-mazurenko-bot. 8.1

[90] Antoine Raux and Maxine Eskenazi. A finite-state turn-taking model for spoken dialog systems. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 629–637. Association for Computational Linguistics, 2009. 4.2.2

[91] Christian Raymond and Giuseppe Riccardi. Generative and discriminative algorithms for spoken language understanding. In *INTERSPEECH*, pages 1605–1608, 2007. 7.1, 7.3.2

[92] Keith Rayner and Susan A Duffy. Lexical complexity and fixation times in reading: Effects of word frequency, verb complexity, and lexical ambiguity. *Memory & Cognition*, 14(3):191–201, 1986. 7.3.4

[93] Alan Ritter, Colin Cherry, and William B Dolan. Data-driven response generation in social media. In *Proceedings of the conference on empirical methods in natural language processing*, pages 583–593. Association for Computational Linguistics, 2011. 2a

[94] Erica Sadun and Steve Sande. *Talking to Siri: Mastering the Language of Apple's Intelligent Assistant*. Que Publishing, 2014. 1.1

[95] Elliot Salisbury, Sebastian Stein, and Sarvapali Ramchurn. Real-time opinion aggregation methods for crowd robotics. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 841–849. International Foundation for Autonomous Agents and Multiagent Systems, 2015. 2.2

[96] Denis Savenkov and Eugene Agichtein. Crqa: Crowd-powered real-time automatic question answering system. In *Proc. the Fourth AAAI Conference on Human Computation and Crowdsourcing (HCOMP 2016)*, 2016. 2.3

[97] Denis Savenkov, Scott Weitzner, and Eugene Agichtein. Crowdsourcing for (almost) real-time question answering. In *Proceedings of the Workshop on Human-Computer Question Answering, NAACL 2016*, 2016. 2.2, 4.1

[98] Amazon Developer Services. The alexa prize, 2016. URL `https://developer.amazon.com/alexaprize`. 1.1

[99] Lifeng Shang, Zhengdong Lu, and Hang Li. Neural responding machine for short-text conversation. *arXiv preprint arXiv:1503.02364*, 2015. 2b

[100] Preetjot Singh, Walter S. Lasecki, Paulo Barelli, and Jeffrey P. Bigham. Hivemind: A framework for optimizing open-ended responses from the crowd. In *University of Rochester Technical Report*, 938, 2012. 1.1

[101] Ming Sun. *Adapting Spoken Dialog Systems Towards Domains and Users*. PhD thesis, YAHOO! Research, 2016. 1.1

[102] Jaime Teevan, Susan T. Dumais, and Daniel J. Liebling. To personalize or not to personalize: Modeling queries with variation in user intent. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pages 163–170, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-164-4. doi: 10.1145/1390334.1390364. URL `http://doi.acm.org/10.1145/1390334.1390364`. 4.5.2

[103] Stefanie Tomko. Improving user interaction with spoken dialog systems via shaping. In *CHI'05 Extended Abstracts on Human Factors in Computing Systems*, pages 1130–1131. ACM, 2005. 6.5.2

[104] Timo Tuomisto, Tiina Kymäläinen, Johan Plomp, Anu Haapasalo, and Kati Hakala. Simple rule editor for the internet of things. In *Intelligent Environments (IE), 2014 International Conference on*, pages 384–387. IEEE, 2014. 3.2.1

[105] Gokhan Tur, Dilek Hakkani-Tur, and Larry Heck. What is left to be understood in atis? In *SLT, 2010 IEEE*, pages 19–24. IEEE, 2010. 7.3.2

[106] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference*

on Human Factors in Computing Systems, CHI '14, pages 803–812, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2557420. URL http://doi.acm.org/10.1145/2556288.2557420. 3.1, 3.2.1, 3.4.3

[107] Luis von Ahn. *Human Computation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2005. 2.2

[108] Luis von Ahn and Laura Dabbish. Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 319–326, New York, NY, USA, 2004. ACM. ISBN 1-58113-702-8. doi: 10.1145/985692.985733. URL http://doi.acm.org/10.1145/985692.985733. 2.2, 6.3.2, 7.1, 7.2

[109] Jeroen Vuurens, Arjen P de Vries, and Carsten Eickhoff. How much spam can you take? an analysis of crowdsourcing results to increase accuracy. In *Proc. ACM SIGIR Workshop on Crowdsourcing for Information Retrieval (CIR'11)*, pages 21–26, 2011. 4.6, 4.6.3

[110] Marilyn A Walker, Amanda Stent, François Mairesse, and Rashmi Prasad. Individual and domain adaptation in sentence planning for dialogue. *Journal of Artificial Intelligence Research*, 30:413–456, 2007. 1.1

[111] Lu Wang, Larry Heck, and Dilek Hakkani-Tur. Leveraging semantic web search and browse sessions for multi-turn spoken dialog systems. In *ICASSP 2014*, pages 4082–4086. IEEE, 2014. 7.1

[112] Wei Yu Wang, Dan Bohus, Ece Kamar, and Eric Horvitz. Crowdsourcing the acquisition of natural language corpora: Methods and observations. In *SLT 2012*, pages 73–78. IEEE, 2012. 7.1

[113] Wikipedia. Collaborative software — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Collaborative%20software&oldid=755737288, 2016. [Online; accessed 26-December-2016]. 2.3

[114] Jason Williams, Antoine Raux, Deepak Ramachandran, and Alan Black. The dialog state tracking challenge. In *Proceedings of the SIGDIAL 2013 Conference*, pages 404–413, 2013. 7.1

[115] Puyang Xu and Ruhi Sarikaya. Targeted feature dropout for robust slot filling in natural language understanding. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014. 7.1, 7.3.2

[116] Wei Xu and Alexander I Rudnicky. Language modeling for dialog system. In *Proceedings of ICSLP 2000*, 2000. 6.2

[117] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: Using gui screenshots for search and automation. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 183–192, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-745-5. doi: 10.1145/1622176.1622213. URL http://doi.acm.org/10.1145/1622176.1622213. 3.2.1

[118] Steve Young. Using pomdps for dialog management. In *SLT*, pages 8–13, 2006. 7.1

[119] Tiancheng Zhao, Kyusong Lee, and Maxine Eskenazi. Dialport: Connecting the spoken

dialog research community to real user data. *arXiv preprint arXiv:1606.02562*, 2016. 1.1, 2.1