Dissertation

A Crowd-Powered Conversational Assistant That Automates Itself Over Time

Ting-Hao (Kenneth) Huang

June 12th, 2018

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

To Lavender

Abstract

Interaction in rich natural language enables people to exchange ideas efficiently and come to a shared understanding quickly. Modern personal intelligent assistants, such as Apple's Siri and Amazon's Echo, utilize conversation as their primary communication channels and illustrate a future in which conversing with computers would be as easy as talking to a friend. However, despite decades of research, modern conversational assistants are still limited in domain, expressiveness, and robustness. In this dissertation, we present a system that blends real-time human computation with artificial intelligence and is able to have real-world open conversations with users. Instead of bootstrapping automation from the bottom up with only automatic components, we took a top-down approach that started with a working crowd-powered system. We developed and deployed a crowd-powered conversational assistant, *Chorus*, and then created a framework, Evorus, that enables Chorus to automate itself over time. Evorus allowed external task-oriented chatbots and chattterbots to be added into Chorus to take over parts of conversations, reuse crowd-submitted responses to answer future similar questions, and gradually learn to select high-quality responses to reduce its reliance on crowd oversight. To make the deployment more robust, we invented a new recruiting method, the Ignition model, to hire workers quickly. We then created Guardian, a framework that easily converts Web APIs to task-oriented chatbots, to empower Chorus through Evorus framework. In order to prevent each of chatbots that Guardian created from preparing its own ad-hoc parameter extractors, we introduced Dialog ESP Game to have crowd workers reliably extract information from running dialogs in a few seconds. Finally, to augment Chorus' capability in controlling users' devices and environments, we created InstructableCrowd, a system that generates trigger-actions rules based on conversation. In our two-year-long deployment, more than 420 users have talked with Chorus during more than 2,200 conversation sessions. Our work demonstrates how a crowd-powered conversational assistant can be automated over time, and more importantly, how such a system can be deployed to talk with real users to help them with their everyday tasks.

Contents

Chapter 1

Introduction

Personal intelligent assistants, such as Amazon Echo, Google Home, and the millions of chatbots on the Facebook Messenger platform, illustrate a future where users can seek help from computers using natural language. Currently, however, these devices have not yet realized the functionality imagined by scientists, filmmakers, and science fiction novelists, where intelligent assistants communicate with humans using natural language fluently, ask questions to learn what users want, track recent interactions, and learn from experience to make interactions more fluid and efficient.

Researchers in the dialog system community have made significant progress in enriching the capability of these systems, yet creating a fully automated system that can hold a long, sophisticated, open conversation with users is still a challenge. Indeed, in 2018, Amazon offered a $1 million prize for a system capable of holding a high-quality 20-minute social conversation with users. So far, no team has achieved this goal [137].

Although automated dialog systems have not yet realized their promise, crowd-powered conversational assistants have been shown to be able to hold high-quality conversations with users. However, few studies have explored how to transition a crowd-powered system into an automated system. In our dissertation work, we demonstrate that a crowd-powered dialog system can be automated over time to support real-world open conversations. In this chapter, we describe the motivation, scope, goal, and overview of this dissertation.

1.1 Thesis Statement

The thesis of this dissertation is as follows:

> **By integrating new chatbots into a crowd-powered conversational assistant, reusing crowd answers, and gradually reducing the crowd's role in choosing high-quality responses, a crowd-powered dialog system can be automated over time to support real-world open conversations.**

1.2 What type of system can hold an open conversation?

The goal of my dissertation work was to automate a functional crowd-powered dialog system capable of holding open conversations. The Cambridge Dictionary defines "conversation" as a "talk between two or more people in which thoughts, feelings, and ideas are expressed, questions are asked and answered, or news and information is exchanged.[1]" In the context of dialog system research, we build on this definition to define an open conversation as inclusive of the following characteristics:

- **Multi-turn interaction:** Multiple, lengthy back-and-forth exchanges. Today's voice-enabled devices, such as Amazon's Echo, only support single-turn or short interactions (*e.g.*, one-shot voice commands).
- **Open domains:** Content from multiple domains rather than just one. For instance, when people talk about traveling, they often discuss related matters such as flights, food, weather, attractions, etc. Furthermore, human-to-human conversation offers unlimited possible conversational domains. While some domains (*e.g.*, weather, food) arise more frequently than others, open conversation requires at least some ability to understand and respond to communication in any arbitrary domain.
- **Personalized:** A recommender system that returns different results to the same query from different users based on each user's unique behavioral history. Conversations implicate underlying knowledge about the world, the speaking context, and the speakers involved. Consequently, people say different things to different people even about similar topics, and any automated system should be able to do the same.
- **Mixture of task-oriented and social dialog:** The dialog system community often deals with task-oriented dialog (where the goal is to complete a task) and social dialog (where the goal is to engage users) separately. Real-world conversation often contains a coherent combination of both types.

My dissertation work focused on text-based dialog system assistants. Text-based systems bypass the technical bottlenecks of understanding and generating multi-modal content, which allows researchers to engage the core challenges of conversational intelligence. Furthermore, text-based systems are generally easier to scale because they avoid the engineering burden raised by managing, storing, and streaming video and audio. Therefore, conversation is an efficient interface for communicating with dialog systems – whether fully automated, semi-automated, or operated entirely by people – with human-level intelligence. We aimed at building a text-based personal assistant that can answer user's questions in various domains, provide personalized suggestions, brainstorm ideas with users, discuss complex topics meaningfully – and be scalable and affordable at the same time.

1.3 Why conversation?

Conversation is a worthwhile modality for communicating with computers for two main reasons:

[1]The "conversation" entry in Cambridge Dictionary:
https://dictionary.cambridge.org/dictionary/english/conversation

Conversation is powerful. People from diverse backgrounds and with different knowledge use natural language to exchange ideas efficiently, come to a shared understanding quickly, and describe nearly anything in precise detail. Therefore, conversation is an ideal interface for systems with a human-level capability of understanding and responding.

Conversational interfaces are inevitable. While today's voice or natural language interfaces have limited capability to hold open conversations, they will be an inevitable part of future interaction between humans and computers. In a panel discussion at CHI 2017 [47], Pattie Maes described the three trends in today's technologies: the locations where interactions take place are getting closer to human bodies; the input and output modalities are increasingly based on human terms instead of computer terms; and people use computers in all aspects of their lives, not only for professional tasks. Conversational interfaces fit these trends well, and a conversational interface only requires a microphone or a keyboard, which nearly all smart devices already have, and with which most people are already familiar.

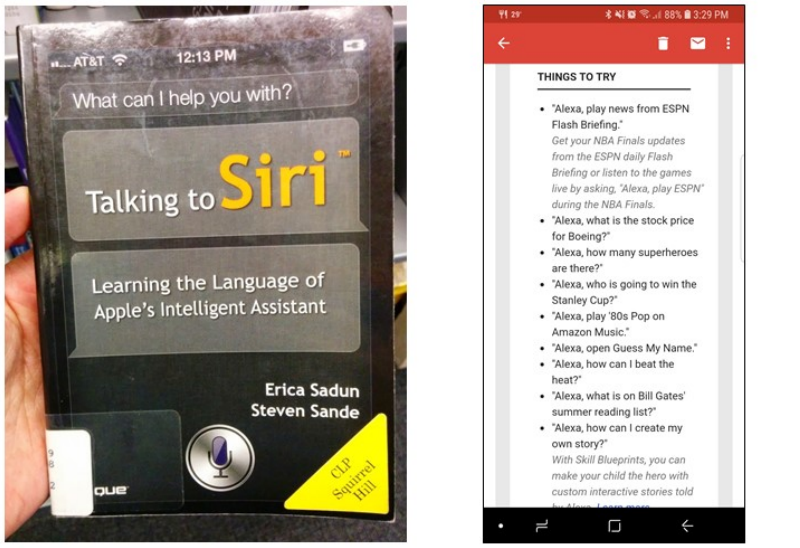
Moreover, conversational interfaces represent a natural evolution of existing interfaces. Natural language and voice interfaces have become a common part of modern digital life. Chatbots use text-based conversations to communicate with users; personal assistants on smartphones such as Google Assistant take direct speech commands from users; and speech-controlled devices such as Amazon Echo use voice as their only input mode. These clients, devices, and infrastructures have already been deployed to millions of workplaces, houses, and users.

1.4 Why have existing approaches failed?

While robust conversational assistance promises a future in which operating a computer is as easy as talking to a friend, today's conversational assistants are still far from that goal. Companies that create voice-enabled devices (*e.g.*, Apple's Siri, Amazon Echo, and Google Home) often instruct their customers to use a fixed input vocabulary or restricted phrasings when talking to their devices (see Figure 1.1).

To illustrate the status quo of conversational assistants, we defined a space where the X-axis indicates how close the systems are to holding open conversations, and the Y-axis indicates their level of automation (Figure 1.2). The classic phone menu (or phone tree) lands at the top-left corner of this space. These systems present an automated navigation menu to voice callers, use interactive voice response (IVR) with Dual-Tone Multi-Frequency codes (DTMF, touch tones) or simple voice recognition, and fully support customer service phone calls. They are automated, but can barely hold any conversation. Compared with older phone menus, today's voice-enabled devices better understand and respond to users' diverse languages and can generally cover broader topics.

The dialog system research community has strived to improve automated conversational assistants, pushing them closer to open conversation, as represented by the gray arrow in Figure 1.2. Researchers have tried to combine multiple dialog systems of different domains to form a single agent [180], to adapt a model trained in one domain to another [141, 158], and to build chit-chat systems for general conversation [7]. These dialog systems are generally better at handling open conversation than smart voice-enabled devices: they can hold longer dialogues, switch topics



"Talking to Siri" **"What's new with Alexa?"**

Figure 1.1: (Left) The book, "Talking to Siri," explains that despite advances in conversational assistants, users still must adapt to what these assistants can understand [126]. (Right) Amazon emails Echo users every week to instruct them how to talk to the device.

within the same conversation, or memorize personal details for each user. Yet, conversational assistants are still limited in the domains in which they work, the richness of expression they support, and their robustness in correctly handling variations in topic, domain, and user. Amazon has further incentivized work in this area through its Alexa Prize: As of November 2017, winning research teams produced bots that average over 10 minutes per conversation (judged as 3.17 on a 5-point scale). But the Grand Challenge prize – $1 million for a system able to hold conversations over 20 minutes with a score of 4.0 or higher – remains elusive [137].

Automated systems such as phone menu, voice-enabled devices, and AI-powered dialog systems come with all the advantages of automation: shorter response times, low run-time cost, and good scalability. However, currently, they are still far from being able to hold open conversations.

1.5 A new approach

Instead of bootstrapping automation from the bottom up with only automatic components, we started with the deployable crowd-powered conversational assistant Chorus [74, 90] and created a framework that enables it to automate itself over time. Crowd-powered conversational assistants are entirely operated by human workers and are known to be able to hold high-quality conversations [90]. An inherent advantage of starting with a working system is that users could talk to it naturally from day one, which allowed us to gather data that could not be collected otherwise. For example, the following is an actual conversation between one user and the deployed version of Chorus.

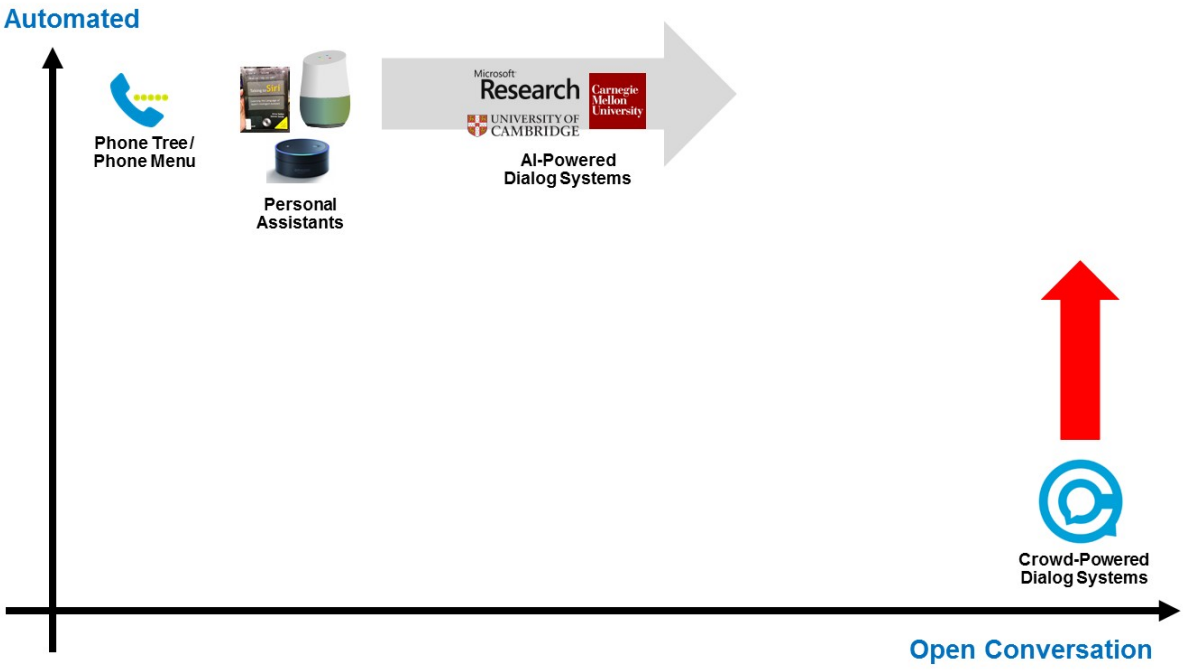


Figure 1.2: To illustrate the status quo of conversational assistants, we defined a space where the X-axis describes how close the systems are to holding open conversations, and the Y-axis indicates how automated the systems are.

user How many suitcases can I take on a flight from the US to Israel?

crowd **Let me check.**
Can I ask you from where are you planning to board the flight?

user Pittsburgh

crowd **with which company are you flying?**
and which air services are you using?

user Delta airlines

crowd **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

crowd **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?

crowd **`< 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....

crowd **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)

crowd **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!

Although these crowd-powered systems are capable of long, sophisticated conversations with users, their lack of automation results in higher latency and costs. The literature has little to say about how to practically transition a crowd-powered system into an automated (or semi-automated) system. Our dissertation work seeks to address this gap. In other words, the goal of our dissertation work is to push the systems in the bottom-right corner of (Figure 1.2) further up the Y-axis without sacrificing their ability to hold open conversations.

The thesis of this dissertation, as stated in Section 1.1, is that **by integrating new chatbots into a crowd-powered conversational assistant, reusing prior crowd answers, and gradually reducing the crowd's role in choosing high-quality responses, a crowd-powered dialog system can be automated over time to support real-world open conversations.**

This dissertation is structured as follows.

1.5.1 Part I: Developing and Deploying Chorus

Chorus Deployment (Chapter 3) Launched as a Google Hangouts chatbot in May 2016 [74], Chorus is a crowd-powered conversational assistant that can hold open conversations about nearly any topic [90]. Users can talk to Chorus anytime, anywhere, using any device–smartphone, smartwatch, or desktop. When a user initiates a conversation, a group of crowd workers is recruited from Amazon Mechanical Turk and directed to an interface where they propose responses, take notes on important facts, and vote on others' replies to identify optimal responses. Collectively, then, the crowd converses with the user as a single, consistent, conversational partner. To date, over 420 users have held more than 2,200 conversations with Chorus, about topics ranging from weather, travel, and birthday gifts to relationship consulting, politics, and shopping.

The system, though, was inadequate at identifying the natural end of conversations, dealing with malicious users and workers, and handling on-demand recruiting. Observations from this deployment will not only improve Chorus but also inform future deployments of low-latency crowd systems in general.

Ignition: A Hybrid Recruiting Methods for Low-Latency Crowdsourcing (Chapter 4) To work interactively, crowd-powered systems, such as Chorus, need access to on-demand labor. To meet this demand, workers can either be recruited as needed directly from the crowd marketplace or recruited in advance and asked to wait in a *retainer pool*. Most evaluations of these recruiting systems have been over a short period, even though we know that marketplaces change and adapt over time.

Ignition was invented to support the deployment of Chorus through a hybrid approach to fast worker recruitment. In this chapter, we describe the novel Ignition approach and the observed times required to recruit and retain workers from Amazon Mechanical Turk, along with the experimental results of a 10-month deployment [67]. Our results demonstrate that it is possible to recruit workers with low latency even for long timeframes. They also suggest a number of opportunities for future work on recruitment strategies and modeling that may further improve on-demand recruitment for deployed systems.

1.5.2 Part II: A Framework That Automates Chorus Over Time

Evorus: A Crowd-Powered Conversational Assistant That Automates Itself Over Time (Chapter 5) Crowd-AI architectures have long been proposed to reduce cost and latency for crowd-powered systems. Evorus demonstrates how automation can be introduced successfully in a deployed system. Its architecture allows researchers to improve on the underlying automated components in the context of a deployed, open-domain dialog system. The goal of my dissertation work is to build upon the deployed Chorus to create a framework that enables Chorus to gradually replace its crowd-powered components with automated approaches using the data it collects. This approach helps deploy robust conversational assistants while driving down costs and gradually reducing reliance on the crowd. This chapter introduces Evorus, a framework built

to integrate new chatbots into Chorus and thus automate more scenarios by reusing prior crowd answers and learning to automatically approve response candidates [71, 75]. Our five-month deployment with 80 participants and 281 conversations showed that Evorus could automate itself without compromising conversation quality.

1.5.3 Part III: Building Chatbots Efficiently For Empowering Chorus

Guardian: Transitioning Web APIs into Crowd-Powered Dialog Systems (Chapter 6) Evorus uses chatbots from different domains to support conversations, but struggles to create a sufficient number of chatbots. While frameworks have been proposed to reduce the engineering effort in developing a chatbot, constructing a usable bot is still a costly endeavor. Scalability is also a concern because of the thousands of chatbots needed to empower Evorus. This chapter describes Guardian, a crowd-powered framework that wraps existing web APIs into immediately usable conversational agents, providing computers with access to an incredible amount of information [68, 69]. Web-accessible APIs can be viewed as a gateway to the rich information stored on the Internet: as of July 2018, ProgrammableWeb.com alone contains the description of more than 19,900 APIs. If Chorus can exploit this robust information, its scope would be significantly enlarged. However, automatically incorporating web APIs into a conversational system is a non-trivial task. Guardian takes as input the web API and desired task. It then uses the crowd to determine the parameters necessary to complete that task, how to make the request, and how to interpret the responses from the API. The system is structured so that, over time, it can learn to take over from the crowd. This hybrid approach helps make dialog systems both more general and more robust.

Dialog ESP Game: Real-Time On-Demand Crowd-Powered Entity Extraction (Chapter 7) One of the key components of the Guardian framework is having crowd workers extract needed information on demand from a running dialog. The extracted information is then passed to a web API. When users interact with conversational assistants such as Chorus, they expect a response in 10-30 seconds. While this latency range allows real-time crowdsourcing techniques to function, the literature has little to say about speed-quality trade-offs when the time budget is only a few seconds. If workers have as much time as they want to annotate a sentence, most AI systems would assume that the annotation is trustworthy, but it was not clear that this assumption would hold when workers have only 20 seconds, for example. This chapter presents the experimental results of on-demand, crowd-powered entity extraction [72]. The solution, Dialog ESP Game, uses a similar mechanism as the ESP game for image labeling to have multiple workers extract key information from a running dialog. When multiple players agree, entities can reliably be extracted from a statement. The experiment demonstrated that this approach is robust in extracting unexpected input and can recognize new entities. This approach achieves better F1 scores than those of the automated baseline for complex queries, with latency under 10 seconds. The proposed method is also evaluated via Google Hangouts' text chat and demonstrates the feasibility of real-time, crowd-powered entity extraction.

1.5.4 Part IV: Expanding the Capabilities of Chorus

InstructableCrowd: Creating IF-THEN Rules for Smartphones via Conversations with the Crowd (Chapter 8) One limitation of Chorus is that it can only provide information to its users: it cannot perform other tasks or interact with its users' environments. This is an obvious shortcoming because smartphones contain a wealth of sensors and effectors: people living in colder climates, for instance, may want their phones to wake them up earlier than usual if it has snowed overnight. The crowd could be used to program such functionality, though this would require giving unknown persons access to personal devices. To address this problem, we created *InstructableCrowd*, a system that allows end users to create rich, multi-part IF-THEN rules via conversation with the crowd [73]. Users verbally express a problem to crowd workers, who collectively program relevant IF-THEN rules to help them via conversation. InstructableCrowd allows users to create rules on the go via voice and does not require a complicated interface. Our study with 12 non-programmers showed that InstructableCrowd achieves a similar average F1 score (0.93) in selecting sensors/effectors as users themselves (0.94), and accuracy of 90.7% in filling attributes for those sensor/effectors. Incremental editing on crowd-created rules resulted in even better performance. These results indicate that InstructableCrowd can let users converse with the crowd to personalize their increasingly powerful and complicated devices.

While most automated systems created from crowd work simply use the crowd for data, Chorus and Evorus demonstrated how a crowd-powered conversational assistant can be automated gradually. Our systems tightly integrated crowds and machine learning, and provided specific points where automated components can be introduced. Over time, more and more chatbots can be integrated into Evorus, more answers collected during conversations can be reused, and the machine-learning algorithm can better choose high-quality responses. In this dissertation work, we have demonstrated a crowd-powered dialog system can be automated over time to support real-world open conversations.

Chapter 2

Related Work

The work is related to *(i)* dialog systems, *(ii)* human computation, and *(iii)* real-time crowd-powered systems.

2.1 Automated Dialog Systems

The Spoken Dialog System (SDS) research community has long explored how automated dialog systems can be built to understand human language, hold conversations, and serve as personal assistants. Yun-Nung (Vivian) Chen *et al.* [29, 32] and Steve Young [178] offer, respectively, comprehensive overviews of the research in dialog systems. Although approaches to building a conversational assistant differ significantly among dialog system research and real-time crowd-sourcing research, our work was inspired by the work done in this field. In this section, we provide an overview of the status of modern dialog systems, and position our work in the context of dialog system research.

2.1.1 Task-Oriented Dialog Systems

Modern dialog system research has largely sought to build "task-oriented dialog systems," or systems that can help users with tasks. A task-oriented dialog system traditionally consists of three main stages: natural language understanding, dialog management, and natural language generation [16, 18]. In this dissertation, in order to empower Chorus, Guardian followed this three-stage architecture to create task-oriented chatbots quickly, but used crowd workers to operate natural language understanding and generation (Chapter 6) [69].

Natural Language Understanding: Natural language understanding aims to convert an input user utterance into a form that computers can process. Three common subtasks of natural language understanding are **domain identification**, **intent detection**, and **slot filling.** For example, when a user asks "Are there any action movies to see this weekend?"[1], the system first identifies the "domain" of this sentence. In our example, the domain should be "movie" rather

[1]This example is from the tutorial by Chen *et al.* [32].

than "weather" or "food." Next, the back-end system is designed to detect which action the user wants to perform, such as buying tickets or locating show times, in this case. Let's assume our example sentence maps to an action called `request_movie`, which looks up movies that satisfy a set of search criteria. Finally, when the system has decided which action to execute, it fills in a set of parameters (slots) according to the action: the action `request_movie` needs to pass the search criteria, where the requested movie `genre` is "action", and the `time` is "this weekend." At the end of language understanding procedure, the output is usually a semantic representation form of the input language, *e.g.*, `request_movie (genre = action, date = this_weekend)`.

Recent research has focused on using neural approaches to tackle language understanding tasks, including a recurrent neural network (RNN) for intent detection [122] and slot filling [107, 110, 173] and a joint model based on convolutional neural networks (CNN) for intent detection and slot filling [54, 170]. Though effective, these approaches often require a large amount of labeled training data, which does not exist for many domains. Furthermore, even when labeled training data could be collected, state-of-the-art supervised learning approaches can still be brittle in extracting unseen slot values [171]. In other words, complex sentences can be problematic for neural approaches. In our dissertation, Guardian managed the problems of training data and complex sentences by using crowd workers to extract slot values from a running conversation in nearly real-time (Chapter 7) [72].

Dialog Management: Based on the semantic representation compiled by the natural language understanding procedure, the dialog management component then decides what the system is going to *do*, such as asking confirmation or clarification questions, asking follow-up questions to collect more information, or querying the back-end system to get an answer. Classic dialog management frameworks have two main functions: tracking the **dialog state** and performing the **dialog policy**. Early dialog systems used a set of predefined "dialog states" to represent each possible situation (*e.g.,* which parameters have been filled). For instance, when the dialog manager receives the semantic form `request_movie (genre = action, date = this_weekend)`, it can track the current dialog state as `[genre, time]`, which indicates the parameters `genre` and `time` have been filled. In each dialog state, the "dialog policy" decides which action should the system take. If our system is built for one theater, these two parameters (`[genre, time]`) should be sufficient and the policy could specify the system to search the movie database. However, if the system supports multiple theaters, the dialog policy might have the system ask the user which theater is he/she looking for.

In modern dialog systems, both dialog states and dialog policies can be machined-learned from data. Williams *et al.* used a partially observable Markov decision process (POMDP) to learn dialog policy [168], which allowed models to consider the uncertainty introduced by upper-stream components, such as speech recognizer, and could therefore be more robust to errors. Researchers have also used deep-learning methods to track dialog states. The state-of-the-art dialog managers, such as the DNN-based approach [63] used in Dialog State Tracking Challenge in 2013 [167], employed neural models to monitor the dialog progress. One challenge of this type of systems is its generalizability: Since the errors propagated from upper-stream modules could significantly damage dialog manager performance, each of the components in the pipeline,

including speech recognizer and semantic decoder, needs to be fine-tuned. This makes it difficult to adapt a working dialog system from its original domain to a new domain.

Natural Language Generation: Finally, the language generation component took the output from dialog management component and created a response in natural language. Early dialog systems used rule-based or template-based methods to produce responses [1]. In 2000, Oh *et al.* introduced a corpus-based approach [116], and recently, researchers used Neural Network (NN) based to generate natural language for both task-oriented and social dialog systems [108, 139, 151].

It is noteworthy that speech synthesis is *not* usually considered part of natural language generation because speech-synthesis technologies often take natural-language text as input but do not involve the text generation process. Similarly, speech recognition is *not* usually considered part of natural language understanding, since most language-understanding technologies take the transcribed text or the probability estimates output by the speech recognizer as their input. Thus, although this dissertation focused only on text-based personal assistants, the technologies we developed can be easily transferred to support spoken dialog systems.

Traditional practices modularize a task-oriented dialog system into several sub-modules such as language understanding and dialog management. This approach makes it easier to focus on each sub-problem, yet also makes domain adaptation difficult because each component needs to perform well to collectively form a usable pipeline. In our work, Evorus does not require each automated component to work perfectly. Instead, our system has the crowd's oversight and allows chatbots to make mistakes. With each mistake, Evorus gradually learns to use the corresponding chatbot at the right point of a conversation.

In order to bypass the issues introduced by modularization of dialog systems, researchers have begun using deep neural networks to learn dialog end-to-end. For example, Wen *et al.* introduced a network-based end-to-end trainable task-oriented dialog system, which treated dialog as a mapping process from dialog histories to system responses [165]; Zhao *et al.* used an end-to-end reinforcement learning approach to jointly learn policies for both language understanding and dialog strategy [179]; and Li *et al.* presented an end-to-end neural dialog system for task completion [103]. However, again, these approaches require large training data, which is not always available.

2.1.2 Social Bots

Instead of completing tasks for users, the goal of social bots is to engage users in social conversations. These bots have gained more attention in recent years [48], especially after Amazon launched the first Alexa Prize in 2017 to develop bots that could mimic everyday conversations [133]. Early social bots (also known as "chatterbots"), such as Eliza [164] and ALICE [159], were powered by hand-crafted scripts and parsers. To reduce the time needed to develop adequate conversational responses, researchers turned to deep-learning methods. Li *et al.* used mutual information to promote neural models to produce more diverse responses [101], and this model was later improved by using reinforcement learning [102]. Other approaches to conversational responses have also shown promise: The "Sounding Board" from University of

Washington [46], the winning team of the 2018 Alexa Prize, focused on users' engagement and providing content that users are interested in.

In sum, researchers in the dialog system community have spent tremendous effort studying and developing systems that can hold conversations. However, modular dialog systems are still limited in their domains, expressiveness, and robustness; and while deep-learning methods have been shown to be useful in end-to-end training and can bypass some limitations introduced by modular systems, they often require large amount of data to train a usable system. In this dissertation, we introduced an alternative approach that starts with a crowd-powered system, and the system can automate itself over time. Our method does not require training data, and it allows automated chatbots, despite being imperfect, to contribute to an open conversation.

2.2 Human Computation

In his 2005 PhD dissertation, Luis von Ahn introduced the concept of "human computation" [152], or "harnessing human time and energy for addressing problems that computers cannot yet tackle on their own." My dissertation focused on combining human and computer intelligence to tackle the grand challenge of conversational assistants, which state-of-the-art dialog systems can barely achieve.

This dissertation builds upon two sets of research under the broader umbrella of human computation: *(i)* crowdsourced question answering (QA), and *(ii)* crowd-machine hybrid systems.

2.2.1 Crowdsourced Question Answering (QA)

The goal of personal assistants is to help users solve everyday problems, answer their questions, or provide information. Toward those ends, human computation researchers have developed various systems to answer user's questions or organize information for a given topic. For instance, Savenkov *et al.* created the Crowd-powered Real-time automatic Question Answering system (CRQA), which combined an automatic question answering system and human workers to answer questions posted on Yahoo! Answers [130], and ChaCha[2] used individual workers to respond to users' questions in nearly real time. Prior work has also looked at providing answers to uncommon web queries by having workers extract answers from automatically generated candidate web pages [12] or by asking crowd workers to answer visual questions sent from users who are blind or visually impaired [14]. These projects demonstrated that crowd workers are able to search for or generate short, focused answers to various questions, even within a short period of time. However, in order to answer real-world questions that users would ask, this one-shot, short interaction is often insufficient. A good personal assistants should be able to ask follow-up questions and determine the context to provide personalized answers.

Other projects aimed at creating longer, sophisticated answers to open questions. Knowledge Accelerator [53], for example, used a workflow to have a group of crowd workers collect information from the Internet and compose a Wikipedia-style article to answer open questions such as:

[2]ChaCha: https://en.wikipedia.org/wiki/ChaCha_(search_engine)

"What are the best attractions in Los Angeles for families with young children?" While Knowledge Accelerator is able to provide high-quality answers to open questions, it was not developed for assisting people in real-time and thus lacks users' synchronous feedback. Similar limitations are found in the community-based QA (CQA) services such as Quora[3] and Yahoo! Answers[4], where answers, by design, should be general enough to help anyone, not just the original poster. In a reality, a decision such as which attraction in Los Angeles to visit with family usually involves many small, subtle, or personal preferences and constraints. We believe an interactive conversation can better explore the problem space and provide better answers.

2.2.2 Crowd-Machine Hybrid Systems

In 2011, when Edith Law and Luis von Ahn echoed the original definition of human computation, they explicitly added "artificial intelligence," stating that human computation harnesses human intelligence "to solve computational problems that are beyond the scope of existing Artificial Intelligence (AI) algorithms" [98]. In the same year, Quinn *et al.* surveyed a set of work that self-identified as "human computation" and concluded one of the key factors shared among these projects is that the problems fit the general paradigm of computation, and "as such might someday be solvable by computers" [119]. These definitions suggest the possibility of transitioning a system from being entirely operated by human workers to a certain level of automation. However, literature has very little to say about how this type of transition would practically happen.

Researchers explored hybrid systems that combine human and machine to solve a wide range of tasks. In the spirit of human computation, most prior work used human workers as a complementary computing power to augment existing algorithms or systems. For example, Alloy used crowd workers to revise item clusters automatically generated by clustering algorithms [25]; Flock asked humans to suggest predictive features and their importance for machine-learning models [33]; CrowdDB [49] used human input to complement the missing information in the database and thus enabled functions that were not possible before; and JellyBean [129] combines human and machine to count objects in photos. While these projects showed that human insights can improve system output, most of these projects did not study how to gradually reduce reliance on the crowd while retaining good performance. Among the few exceptions was the work of Kamar *et al.*, which demonstrated the human assessment label collected at execution time by a crowd-machine hybrid system can teach the system to predict satisfactory system output, and thus potentially require fewer workers to evaluate the system in the long run [81]. In this dissertation, we introduced a general framework using not only quality-assessment labels (upvotes and downvotes), but also the crowd-generated content to improve a hybrid crowd-machine system over time.

Some projects, on the other hand, attempted to use automated technologies to assist human tasks. For example, Kamar *et al.* dispatched easier annotation tasks via a decision algorithm to an automated classifier to make crowdsourced data-annotation more salable [82]; and Foundry used automated algorithms to find an appropriate combination of experts in order to form a

[3]Quora: https://www.quora.com/

[4]Yahoo! Answers: https://answers.yahoo.com/

flash team [125]. Most of these projects can clearly specify which part of human labor can be automated by algorithms, but they did not study how automated models might improve by having more data. Zensors is one of the few examples that focus on using the crowd to bootstrap automated models [85]: a video feed initially monitored by the crowd was gradually bootstrapped to automated computer-vision models to take over the surveillance tasks. To the best of our knowledge, none of prior crowd-machine systems have focused on creating and deploying a conversational personal assistant.

2.3 Real-Time Crowd-Powered Systems

Early crowdsourcing systems leveraged human intelligence through batches of tasks completed over hours or days. For example, while the ESP Game paired workers synchronously to allow them to play an interactive image-label guessing game [153], it did not provide low-latency response for any individual label. However, for a usable personal assistant, quick response time is essential. In this dissertation, we focused on personal assistants in a text-based instant messaging setting (*e.g.,* Google Hangouts). According to literature, the average response time in instant messaging is 24 seconds [79]. 24.5% of instant messaging chats get a response within 11-30 seconds, and 8.2% of the messages have longer response times [8]. Another study focusing on small groups also showed that, on average (Least-Square Means), students respond to an instant message in 32 seconds, and people in startups respond in 105 seconds [4]. In order to build a crowd-powered conversational assistant that can respond this quickly, we used technologies of real-time crowdsourcing, which enable systems to utilize human intelligence within a few seconds. In this section, we overview the motivations, techniques, and applications of real-time crowdsourcing and real-time crowd-powered systems.

2.3.1 How Can We Recruit Workers Quickly?

Because crowdsourcing can be slow, a primary challenge for real-time crowd-powered systems is decreasing latency. At a high level, there are at least three sources of latency for such systems: *(i)* time required to assign workers to the task; *(ii)* time required for the workers to do the job; and *(iii)* time for the system to integrate the work they did into the output for users. While *(ii)* and *(iii)* are domain-specific, all crowd-powered systems share the challenge of recruiting workers quickly.

Two main approaches have been used to recruit workers quickly from crowd marketplaces:

1. **On-demand recruiting:** Workers are recruited when they are needed, often when the task starts [14], usually by posting HITs on the MTurk marketplace with some Search Engine Optimization (SEO) tricks, such as over-posting, to increase recruiting speed.
2. **Retainer:** Workers are recruited ahead of time into a retainer pool from which they can be called upon quickly [10]. Further work has used queuing theory to show that this latency can be reduced to under one second and has also established reliability bounds on using the crowd in this manner [11].

These approaches have trade-offs. Recruiting by posting HITs is inexpensive but slow, while a full-time retainer is fast but expensive.

On-demand recruiting is known to be robust, as it has been used in deployed systems such as VizWiz, but its recruiting time is reportedly longer than a minute, which may be too long for many interactive applications [14]. In the VizWiz system, Bigham *et al.* used pre-recruiting to reduce the experienced response time of users – workers are recruited when users begin using the application, which gives the system a lead time of approximately one minute. However, pre-recruiting is not always possible. For conversational assistants, the time between users opening the application and sending an initial message is very short, which makes pre-recruiting less effective. Furthermore, the SEO tricks used in VizWiz's recruiting process became less economical when Amazon decided to increase the fee from 20% to 40% in 2015.

The retainer model provides fast response time (less than ten seconds) but can be expensive for real-world deployments – especially small and medium-sized deployments, where most of the money is used for waiting time. In this dissertation, in order to deploy a crowd-powered conversational assistant, we introduced a new hybrid approach, the Ignition model (Chapter 4) [67], which combines these two methods to make recruiting workers quick and affordable.

Prior works have aimed to improve the general speed of crowdsourcing by optimizing the crowd component's response time [172], addressing the sources of crowdsourcing latency [52], or inventing new mechanisms for humans to quickly label data [84]. However, none of these technologies have been thoroughly tested in a deployed crowd-powered system for longer than a couple of weeks.

2.3.2 Why Use Real-Time Crowdsourcing?

In this subsection, we overview real-time crowdsourcing systems and technologies that inspired our work, categorizing them under the three common purposes that motivate the uses of real-time crowdsourcing: synchronizing with users, other workers, or automated systems. These usages are not mutually exclusive: complex crowd-powered systems such as Evorus [71, 75], for example, could involve crowd workers interacting with all of these at the same time.

Synchronizing with Users The exploration of real-time crowdsourcing started with the goal of providing fast responses directly to end users. VizWiz, for example, utilized crowd workers to answer visual questions quickly for blind people [14]; Scribe asked non-experts to caption speech for deaf and hard of hearing audiences [88]; and Adrenaline used the crowd to pick "the best moment" from a short video in a second after the film was shot [10].

Whereas the examples cited above focused on discrete, one-shot crowdsourcing jobs, Lasecki *et al.* introduced continuous real-time crowdsourcing in Legion [87], which showed that a dynamic crowd can be recruited to support continuous tasks. This finding is critical to our work because holding a conversation is, by its nature, a continuous and long task. Enabling instant and continuous responses opened the era of *interactive* crowd-powered systems, where users can synchronously give and receive feedback with crowd workers over the course of multiple rounds of interaction. For example, Soylent had workers edit and proofread text in real time inside the user's text editor [9]; Chorus recruited a group of workers to collectively hold a synchronous

conversation with the end user [90]; and IdeaGens enabled an expert to provide real-time guidance to crowd workers (ideators) to generate new ideas [24]. Lasecki *et al.* further introduced the concept of a real-time "crowd agent", saying a dynamic crowd of workers can collectively form a single agent to interact with users and perform tasks [92]. This concept has been shown useful across many domains, including personal assistance.

A variation of this type of system is the real-time "polling" systems, where workers or target audiences are asked to provide their opinions about a given question or item instantly (*e.g.,* the design of a poster). For example, the APP "1Q"[5] uses smartphone push notifications to ask poll questions and collect responses from target audiences within a few minutes. These systems focus on answering a single opinion question, rather than having deeper, interactive conversations.

Synchronizing with Other Workers Prior work has shown that multiple workers can be recruited for collaboration by having workers wait until a sufficient number of workers have arrived [34]. While this approach does not provide low-latency responses for any individual labels, nor to the user, workers are often expected to respond quickly to *other workers.* For example, the ESP Game paired workers synchronously to allow them to play an interactive image-label guessing game [153] and Revolt coordinated crowd workers to collaboratively identify ambiguous items in the data [26]. Similar mechanisms have been adopted by researchers who recruited groups of Amazon Mechanical Turk workers to study collaborative learning [163] and intelligent agent behaviors inside human groups [5].

In our version of Chorus, workers can communicate with each other synchronously using the side memory board on the interface. Sometimes workers talk with each other to clarify users' questions, or the experienced workers can teach new workers how to use the interface properly. Furthermore, we also introduced "Dialog ESP Game" (Chapter 7) [72], which used the mechanism similar to the ESP Game for image labeling [153], but adding a tight time constraint such as 10 or 15 seconds, to extract information from a running dialog within a few seconds.

Synchronizing with Fast-Paced Automated Systems One of the earliest examples to demonstrate the power of a real-time continuous crowd was [87], in which a group of crowd workers collectively controlled a running toy robot. Using Legion tools, the control latency was typically under one second. This example demonstrated that real-time crowdsourcing can also be used to interact with fast-paced automated systems. Following this paradigm, a real-time crowd has been used to control or interact with various automated systems that require a short response time. For example, CrowdDrone used a real-time crowd to orient unmanned aerial vehicles in an unknown environment [128], and CrowdAR used the crowd to identify and track targets in a live video feed [127]. CRQA system used crowd workers to quickly select good answers from a set of answer candidates automatically generated by computers, and could provide high-quality answers within 60 seconds [130, 131]. Many projects in this dissertation have explored the collaboration between crowd workers and fast-paced automated components. Evorus recruited a group of crowd workers to collaborate with automated bots to hold a conversation (Chapter 5) [71, 75]; Guardian had crowd workers extract key information from a running dialog (Chapter 7) [72],

[5] 1Q: https://1q.com/

pass the extracted information to a web API, and then convert the API responses back into natural language (Chapter 6) [69].

One variation of this type of system is the crowd-powered "surveillance" systems, which do not need instantaneous guidance all the time, just fast responses when target incidents occur. For example, Zensors utilized real-time crowdsourcing to monitor a live surveillance video feed and notify end users when a target event (*e.g.,* it starts raining) occurs [85]. Similar concepts have also been applied to recruit individuals in the local community to report air pollution via a mobile app in nearly real time [65]. However, the continuously back-and-forth nature of human conversations make it hard to apply these crowd-powered surveillance technologies in our work.

2.3.3 Deployed Real-Time Crowd-Powered Systems

Deploying a real-time crowd-powered system that is fully run on Mechanical Turk without developer monitoring is challenging, and nearly none of the crowd-powered systems introduced in academia has been brought out of the lab. VizWiz is a rare example of a deployed real-time crowd-powered system [14]. The system has already helped answer over 100,000 questions for thousands of people with visual impairments[6]. This deployment demonstrated some of the realistic trade-offs that need to be addressed. For example, to make the system cost effective, latency was higher and fewer answers were collected per question.

In this dissertation, we deployed a crowd-powered conversational assistant, Chorus, as a Google Hangouts chatbot (Chapter 3) [74]. This is the first crowd-powered conversational assistant that was deployed using Mechanical Turk. During our deployment, we discovered many new challenges and introduced various new technologies to tackle them. For example, we invented a new recruiting method, Ignition model [67], to hire workers quickly; and we also introduced InstructableCrowd so that users can use conversational assistants to customize their devices.

This dissertation is built upon prior real-time crowd-powered systems, and influenced by the work of crowdsourced QA and crowd-machine systems. We deployed the first crowd-powered conversational assistant that is run on Mechanical Turk, and developed our work on top of this system. We are also aware of the strength and limitations of modern dialog system research, and aim to provide an alternative approach to tackle on the grand challenge of open conversation.

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

Part I

Developing and Deploying Chorus

Chapter 3

Chorus Deployment

Over the past few years, crowd-powered systems have been developed for various tasks, from document editing [9] and behavioral video coding [91], to speech recognition [86], question answering [131], and conversational assistance [90]. 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 chapter, 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 [69, 89, 90]. 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[74]. While prior work has shown that crowd-powered conversational systems were possible to create, and have been shown to be effective in lab settings [69, 73, 90], 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.

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.

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

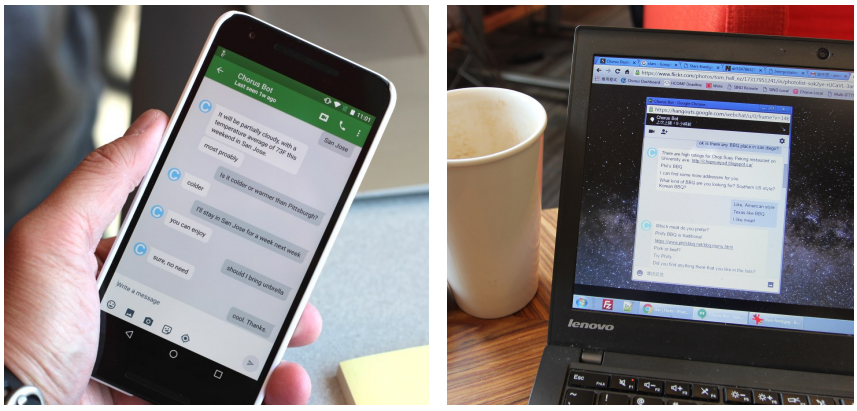


Figure 3.1: Chorus is a crowd-powered conversational assistant deployed via Google Hangouts, which lets users access it from their computers, phones and smartwatches.

3.1 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 [90], and 2) the bot that bridges the crowd component and Google Hangouts' clients. An overview of Chorus is shown in Figure 3.2. 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.

3.1.1 Worker Interface

Almost all core functions of the crowd component have a corresponding visible part on the worker interface (as shown in Figure 3.2). 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.

Proposing & Voting on Responses: Similar to Lasecki *et al.*'s proposal [90], 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 3.2, 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.

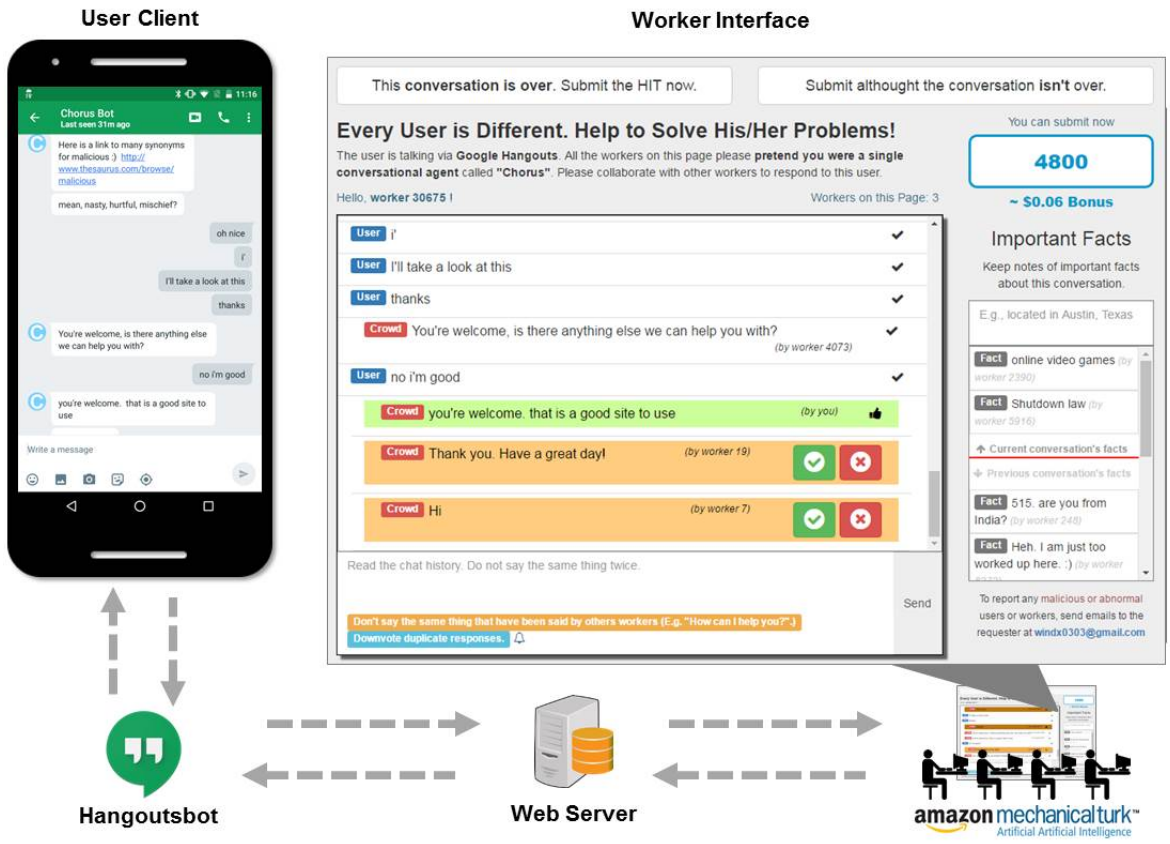


Figure 3.2: The Chorus UI is formed of existing Google Hangouts clients for desktop, mobile or smartwatch. Users can converse with the agent via Google Hangouts on mobile or desktop clients. Workers converse with the user via the web interface and vote on the messages suggested by other workers. Important facts can be listed so that they will be available to future workers.

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 3.2) 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 3.2), 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.

3.1.2 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[2]. 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 [10], 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."

3.2 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[3], 59 users participated in a total of 320 conversational sessions (researchers in this

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

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

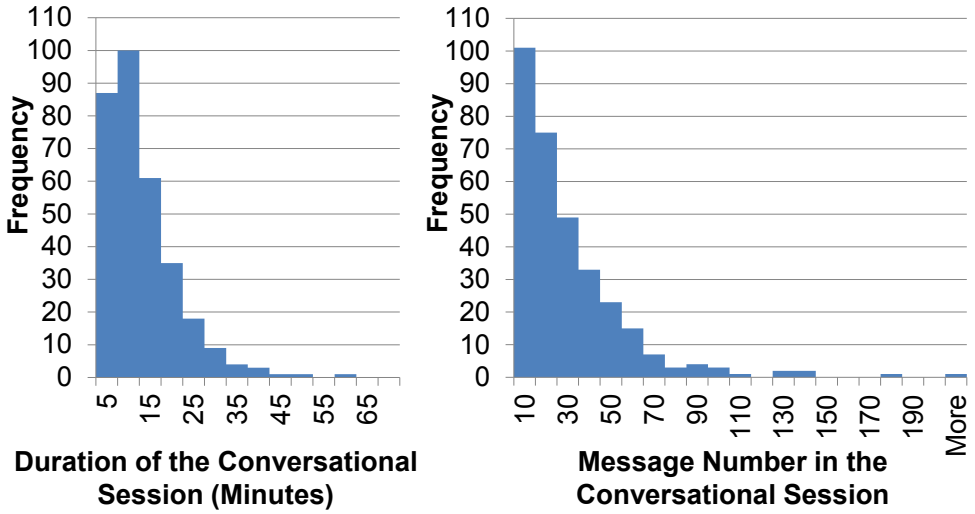


Figure 3.3: The distribution of durations and number of messages of conversational sessions. 58.44% of conversational sessions are no longer than 10 minutes; 55.00% of sessions have no more than 20 messages.

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 3.3. 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 [90]. In Section 1.5, we showed 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.

3.3 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.

3.3.1 "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**

3.3.2 The Dynamics of User Intent

Identifying users' intent is difficult [143]. 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

3.3.3 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.

3.4 Challenge 2: Malicious Workers & Users

Malicious workers are long known to exist [43, 155]. Many crowdsourcing workflows were proposed to avoid workers' malicious actions or spammers from influencing the system's performance [78]. The threats of workers' attack on crowdsourcing platforms have also been well studied [93]. 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.

3.4.1 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?**

3.4.2 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**

3.4.3 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 [155]. 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.

3.4.4 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 [4]. 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.

3.5 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

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

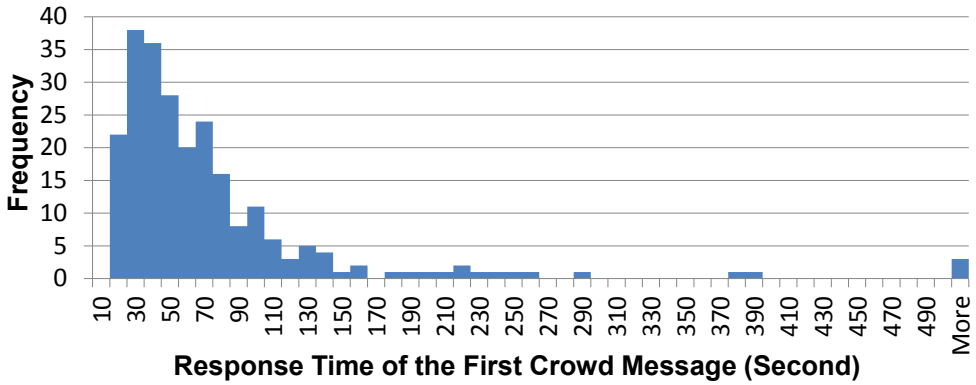


Figure 3.4: Distribution of the response time of the first crowd message. 25.0% of conversations received a first response in 30 seconds, and 88.3% of conversations received a first response in 2 minutes.

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 [79]. 24.5% of instant messaging chats get a response within 11-30 seconds, and 8.2% of the messages have longer response times [8]. 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 MTurk), in which $2.80 is the base rate[5], 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[6]. 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

[5]$0.20 per assignment and 10 assignments per HIT. MTurk charges a 40% fee for HITs with 10 or more assignments.

[6]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.

is shown in Figure 3.4. 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 3.4, 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.

3.6 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.

3.6.1 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**

3.6.2 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**

3.6.3 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?**

3.6.4 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.

3.7 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.

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 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 chapter. We also observed that several users discussed their personal problems such as relationship-related issues. These uses of Chorus are all quite 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.

3.7.1 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[7]) is that workers overall found our tasks 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.

3.8 Summary

In this chapter, we describe 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 crowd-powered systems, and thus represent a rich source of problems for future research to address. Some of these problems will be addressed or resolved in the following chapter, *e.g.,* the new recruiting methods we invented to support Chorus deployment (Chapter 4); and we will discuss more challenges and limitations of the Chorus and its updated version that contained automated components in the Discussion chapter (Chapter 9).

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

Chapter 4

Ignition: A Hybrid Recruiting Methods for Low-Latency Crowdsourcing

A number of interactive crowd-powered systems have been developed to solve difficult problems out of reach for automated solutions. For instance, Soylent used the crowd to edit and proofread text [9], and Legion allowed a crowd of workers to interact with an UI-control task [87]. A primary challenge for such interactive systems is to decrease latency because crowdsourcing can be slow. At a high level, there are at least three sources of latency for such systems: *(i)* time required to get workers to show up to the task, *(ii)* time required for the workers to do the work, and *(iii)* time for the system to integrate the work that they did into the output given to users. While *(ii)* and *(iii)* are domain-specific, all crowd-powered systems share the challenge of recruiting workers quickly.

As mentioned in the Related Work chapter (Section 2.3), two main approaches have been used to recruit workers from crowd marketplaces, such as Amazon Mechanical Turk (MTurk), quickly:

1. **On-demand recruiting:** Workers are recruited when they are needed (task starts) [14], usually by simply posting HITs (Human Intelligence Tasks) on MTurk marketplace.
2. **Retainers:** Workers are recruited into a retainer pool and can be called up on quickly [10].

However, recruiting by posting HITs is inexpensive but slower; whereas, a full-time retainer is expensive but faster. On-demand recruiting is known to be robust, as it has been used in deployed systems such as VizWiz [14], however, its recruiting time is reportedly longer than a minute, which may be too long for many interactive applications. In the VizWiz system [14], Bigham *et al.* use pre-recruiting to reduce the experienced response time of users – workers are recruited when users begin using the application, which gives the system a lead time of approximately one minute. However, pre-recruiting is not always possible. For applications such as crowd-powered conversational agents [74], the time between users opening the application and sending their first message is short, which makes pre-recruting less effective. On the other hand, the retainer model, which holds workers in a waiting pool and calls workers back when tasks come in, has a fast response time (less than 10 seconds). However, the retainer model can be expensive for real-world deployments, especially for small or medium size of deployment, in which most of the money is used for waiting time.

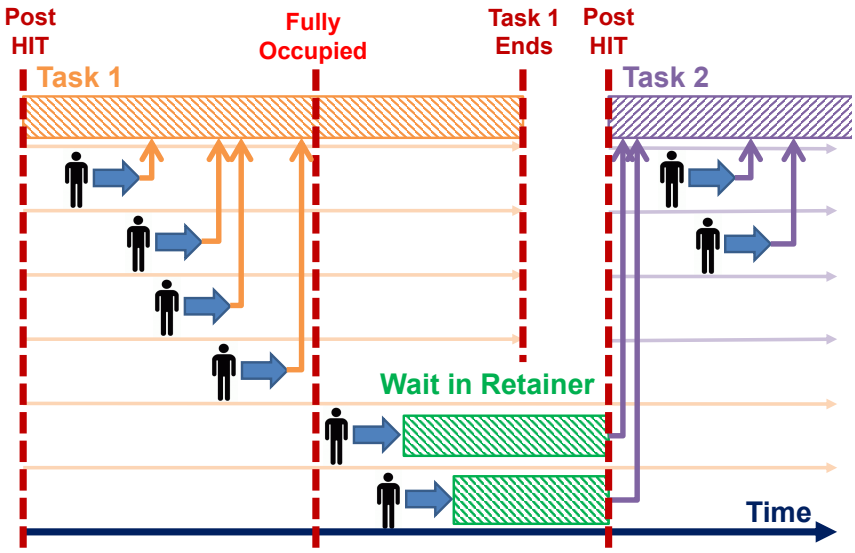


Figure 4.1: Ignition combines on-demand recruiting of workers with a retainer to affordably recruit workers quickly to power deployed crowd systems.

To tackle this cost-latency trade-off, we proposed *Ignition*, which makes low-latency crowd-powered systems feasible over long deployments by balancing two recruiting methods: *(i)* simply posting HITs, and *(ii)* maintaining a worker waiting pool (*i.e.,* a retainer.) As shown in Figure 4.1, Ignition starts recruiting workers when a task begins, and the trick is that the model recruits *slightly more* workers than each task needs. The workers who arrive after a task is fully occupied are retained in a waiting page, and then be directed to the next available task.

Ignition is particularly useful for supporting the following types of tasks:

- Applications with an expected response time between **30 seconds to 2 minutes.**
- Tasks with **dynamic length.**
- Tasks that are better with **multiple workers,** but can start when the first worker arrives.
- Systems that are deployed at small or medium scale.

In this chapter, we report on our experience developing Ignition and our experience with a 10-month deployment with 648 workers collaboratively completing 745 tasks. We deployed Ignition in support of our on-going deployment of a crowd-powered conversational assistant, Chorus [74]. Each Chorus task can start with 1 worker, and could be collectively operated by a maximum of 5 workers. We report on its response time (both of workers arriving to Ignition, and to the supported task), its stability over time, and worker response rates. Our experience may inform future efforts to deploy low-latency crowd-powered systems and develop the underlying infrastructure supporting them.

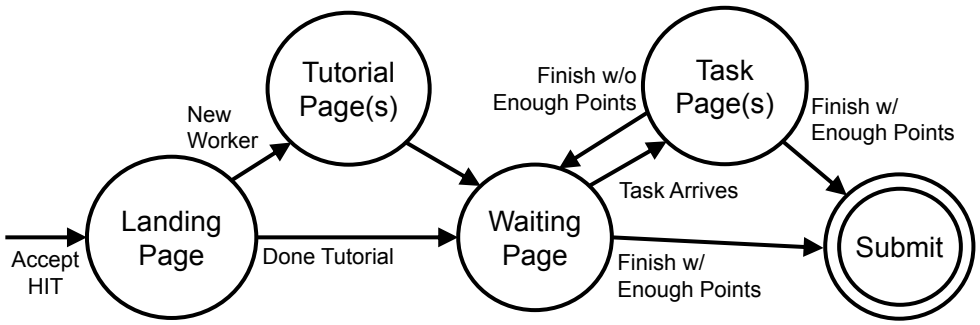


Figure 4.2: Transition graph for workers in Ignition. A worker first reaches the landing page for the introduction and tutorial, and then goes to a waiting page (also known as retainer page) to wait for tasks. When a task arrives for the worker, the worker is called back to perform the task using a pop-up alert and a sound notification.

4.1 Ignition Framework

In this section we describe our implementation of Ignition. This implementation has been deployed for supporting Chorus in-the-wild since June 2016. We iteratively improved the design of the system through multiple empirical evaluations and feedback from workers and our collaborators.

4.1.1 Worker's Workflow

From the workers' perspective, Ignition is composed of a sequence of web pages. As shown in Figure 4.2, the workflow of workers are as follows: First, a worker reaches to the landing page. The landing page uses five slides to briefly introduce the task, and also have the worker sign the consent form when necessary. Each new worker who has never submitted our HIT before is also required to finish the one-minute interactive tutorial. Second, the worker then clicks a button to enter to the waiting page. The interface of the waiting page is shown in Figure 4.3. The worker is instructed to keep the browser tab open to wait for the task. The system grants the worker with an retainer reward (2 points) per second for his/her waiting time. Reward points are later converted to bonus pay for workers. The accumulated reward points are displayed in the middle of the page, with the remaining waiting time and estimated bonus amount listed below.

When a task arrives, the waiting page uses a pop-up alert and a bird sound notification to call the worker back, and the worker is required to respond within 20 seconds. If the worker responds in time, he/she will be then directed to the task page to perform the task; If the worker reaches to 4000 points (estimatedly 33 minutes) without any tasks, the waiting page will also call the worker back to confirm that he/she is still available, and then automatically submits the HIT if the worker responds within 20 seconds. The workers who wait in the retainer pool promise to respond within a specific amount of time. We recognize these promises and the time spent by the workers as valuable contributions to keep a deployed crowd-powered system 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. On the other hand, if a worker does not respond in

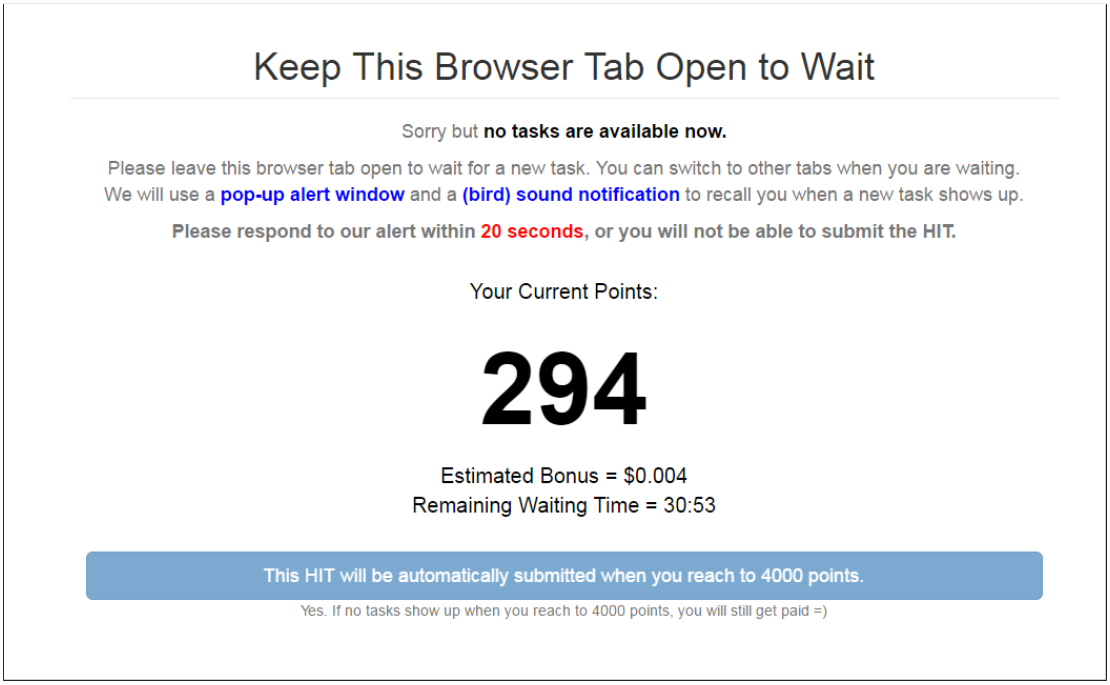


Figure 4.3: The worker interface. Workers are instructed to keep the browser tab open to wait for tasks. Their earned reward points are displayed in the middle of the page, with the remaining time and estimated bonus amount listed below.

time, the interface will be wiped out and become unable to submit. An instruction will further be displayed to ask the worker to return this HIT as a penalty.

Furthermore, as shown in Figure 4.2, if the task utilizes a comparable reward point system as that of Ignition, the worker can be sent back to the waiting page to continue accumulating points if he/she did not make sufficient contribution to the task (R_{task}.) This is particularly useful for the tasks that have dynamic length such as conversation tasks [74]. If a worker enters a conversation task right before it ends and thus earns insufficient reward, Ignition allows the worker to go back to retainer with his/her accumulated reward points to continue waiting. This design also allows workers to focus only on the amount of reward points they have earned, instead of keep track of both waiting time and reward points at the same time.

4.1.2 Recruiting Strategy & Worker Routing

When a task arrives, Ignition recruits slightly more workers than needed, and uses a retainer to hold the extra workers for future tasks. In our deployed system, the task can start with 1 worker, but is better with 4 to 6 workers. Therefore, for each incoming task, we aim to recruit 8 workers in total. The recruiting process can be break down into two parts: Ignition first recalls $C_{retainer}$ workers from retainer, and then aims to recruit C_{market} workers from Amazon Mechanical Turk marketplace. Namely, $C_{retainer}$ and C_{market} sums up to 8, as Equation (4.1) shows.

$$C_{retainer} + C_{market} = 8 \tag{4.1}$$

The underlying assumption of Ignition is that a larger C_{market} has a faster recruiting time from the market, and thus compensates the speed decrease when no (or few) workers are waiting in the retainer ($C_{retainer} = 0$.) The detailed process is as follows.

1. Upon the arrival of a new task, Ignition checks if any workers are currently waiting in the retainer If yes, the system greedily calls at most 6 workers back from retainer to do the task ($0 \leqslant C_{retainer} \leqslant 6$).
2. Ignition then tries to recruit C_{market} workers from mturk marketplace by posting 1 HIT with an average of C_{market} assignments. We introduced randomness into the system that Ignition has a 30% chance to add one assignment ($C_{market}+1$), 30% chance to subtract one assignment ($C_{market} - 1$), and 10% to post 10 assignments. Randomness was added here to allow us to collect data about different numbers of assignments, in case worker response rate or time is dependent on the number of workers already recruited (which may be true if we are pulling from an especially small pool), and to explore latency effects potentially introduced by the platform itself (anecdotally, HITs with different numbers of assignments seem to appear with different latencies on MTurk).

When a task has 5 or more workers (either from retainer or marketplace,) the task is labeled as "fully-occupied" and stops taking more workers, and the workers recruited via the same HIT who arrives later will start waiting in the retainer. However, if some workers left before the task ends and thus the task has less than 5 workers, the task will be open to workers again.

4.1.3 Instant, Retained, and No Tasks

When a worker reaches to the waiting page and starts waiting, one of the following events will occur.

- **[Instant Task]** The task has been created and remains open when the worker arrives to the waiting page. In this case the worker does not need to wait and will be called immediately when reaching to the waiting page.
- **[Retained Task]** When the worker arrives to the waiting page, the original task is fully-occupied or over. The worker opens the browser tab to wait, and then the next task arrives.
- **[No Task]** Similar to the Retained Task, worker arrives to the waiting page and starts waiting. However, no tasks appear till the end of his/her waiting time. The workers can submit the HIT at the end and just get paid with mturk price and waiting bonus.

It is noteworthy that workers are not allowed to "double waiting." If a worker is currently waiting in the retainer and reaches to the waiting page again via a different HIT, the system will block him/her on the second browser tab.

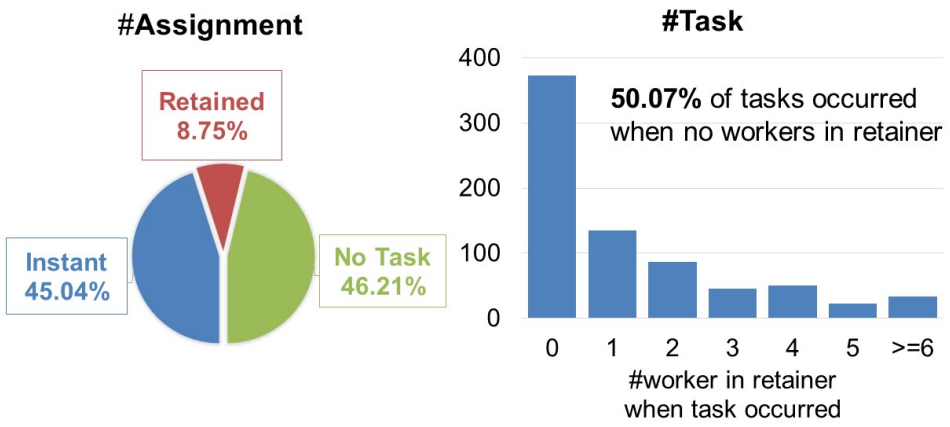


Figure 4.4: The distribution of assignments and tasks. 50.07% of HITs were posted when no workers were in the retainer. Amongst totally 6,823 assignments, 45.04% of assignments were of Instant tasks, 8.75% were of Retained tasks, and 46.21% were of no tasks.

4.2 Long-term Deployment Study

The current version of Ignition was initially launched in June, 2016 for supporting the on-demand crowd-powered conversational agent, Chorus[1], that was deployed to public [74][2]. To date[3], 122 users used the conversational agent during 745 conversational sessions. Each session was one task, which lasted an average of 10.87 minutes (SD = 15.26.) The assignment and task distribution is shown in Figure 4.4. Totally 6,823 assignments by 648 workers were recorded, in which 45.04% were of Instant tasks, 8.75% were of Retained tasks, and 46.21% were of no tasks. As for the task distribution, 50.07% of HITs were posted when no workers were in the retainer, 18.12% of HITs were posted when 1 worker was in the retainer, and 11.54% of HITs were posted when 2 workers were in the retainer. This distribution suggests that in Ignition, the speed of recruiting from marketplace and of recalling workers from the retainer is equally important. We will analyze the performance of these two parts in the follosing subsections.

4.2.1 Recruiting from the Marketplace

In the deployed Ignition, majority of assignments were of the [Instant Task] case, in which workers do not need to wait and are directed to tasks immediately after they reach to the retainer. In the case of [Instant Task], most of the recruiting time are spent in waiting for workers to find the HIT, accept the HIT, start doing the HIT, and finish the tutorial. To the best of our knowledge, no prior works reported how fast a HIT will be taken on Amazon Mechanical Turk marketplace. Therefore, we calculated the probability of a posted HIT that have at least 1 (and at least 3) workers in the retainer at x seconds after the HIT was posted. The results are shown in Figure 4.7.

[1]Chorus: http://talkingtothecrowd.org/

[2]Three built-in qualifications of MTurk were used to ensure task quality: HIT Approval Rate ($> 90\%$), Number of Approved HITs (> 200), and Adult Content Qualification.

[3]All results presented in this chapter are based on data recorded from July 1st, 2016 to April 27th, 2017.

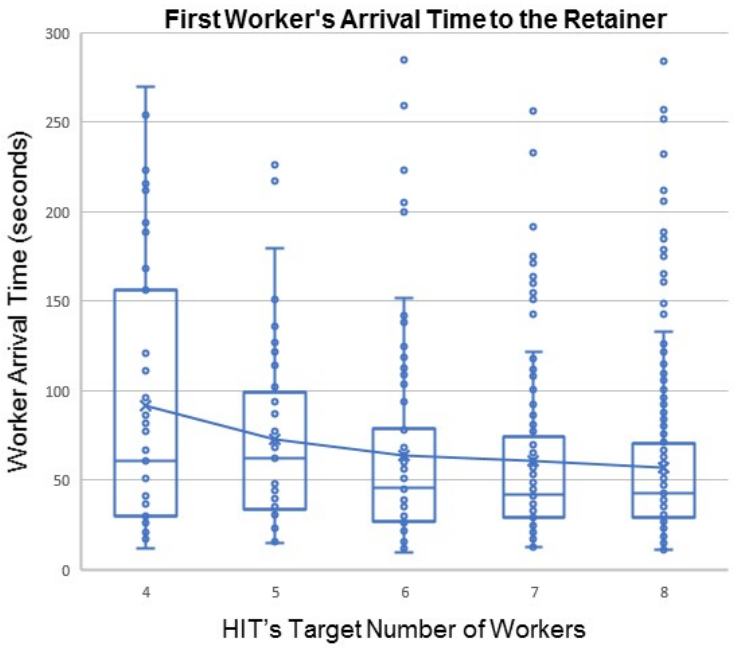


Figure 4.5: The distribution of the first workers' arrival times when the system aimed to recruit various number of workers from mturk marketplace. A larger C_{market} results in not only a smaller mean of arrival time, but also a smaller standard deviation.

The x axis is the cut-off time (x seconds,) and the y axis is the probability at x seconds.

The results suggest that when a HIT posted to mturk marketplace with more assignments, it is more likely workers will arrive to the task earlier. When Ignition posted a HIT with an average of 5 assignments, 40% of the time the first worker will reach to the retainer under 2 minutes; when Ignition posted a HIT with an average of 8 assignments, 79% of the time the first worker will reach to the retainer under 80 seconds. With a larger C_{market}, Ignition's recruiting strategy can get workers faster. This results confirm the underlying assumption of Ignition that a larger C_{market} results in a faster recruiting time from market, which can compensate the speed decrease when no workers waiting in the retainer. Furthermore, the distribution of the first workers' arrival times when the system aimed to recruit various number of workers from mturk marketplace is shown in Figure 4.5 (workers who arrived after 5 minutes are excluded.) A larger C_{market} results in not only a smaller mean of arrival time, but also a smaller standard deviation.

The positive correlation between C_{market} and recruiting speed could be caused by several factors: First, a HIT with more assignments has a longer lifetime on Mturk marketplace before all assignments are taken, and thus has better visibility. Second, a HIT with more assignments is more robust to the workers who hold the accepted tasks for a while instead of doing the task immediately. Finally, given that we had a relatively small group of active workers who took most of our HITs, a smaller $C_{retainer}$ could indicate that more of these active workers are still on the marketplace, and are thus easier to be recruited from the market. Workers use web browser extensions to alert them when new HITs are posted by favored requesters, and therefore some of the results we have seen could be influenced by the use of these tools.

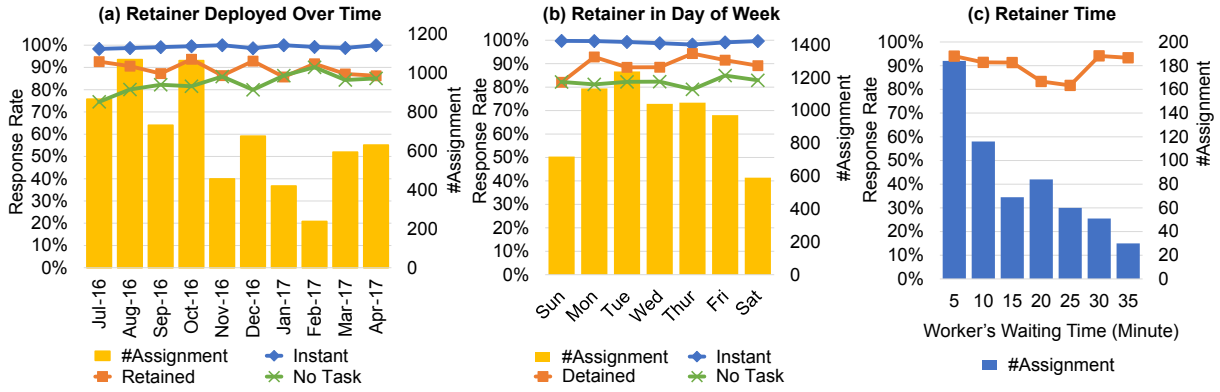


Figure 4.6: Worker recall rate from the retainer *(a)* by months, *(b)* by the day of the week, and *(c)* by time spent waiting. Overall, results demonstrate that the recall rate was reasonably stable during our deployment, although this rate did vary.

4.2.2 Recruiting from the Retainer

The retainer model has shown to be able to recall 75% of workers within a couple of seconds [10] Our deployment study echoes this findings, and further demonstrates the long-term dynamics of a deployed retainer. We calculated the response rate (*i.e.*, the probability that a worker responded to the task within 20 seconds) of [Instant Task], [Retained Task], and [No Task][4] during each month of our deployment. As Figure 4.6(a) shows, the response rate of [Instant Task] cases were nearly perfect. We believe that it is because workers do not need to wait. Except for the first two months, the response rates of [Retained Task] and [No Task] were all higher than 80% during the entire deployment. Interestingly, we found that the response rate of [Retained Task] cases slightly dropped on Sunday (Figure 4.6(b).) Within the [Retained Task] cases, we also calculated the response rate with respect to each worker's waiting time in the retainer (Figure 4.6(c).) We found that workers have the lowest response rate when waited in the retainer between 15 to 25 minutes. It is possible that workers learned to pay more attention at the end of their waiting time (*i.e.*, 33 minutes), perhaps because they were aware of their chances of getting paid without actually doing the task.

It is noteworthy that our implementation has a more relaxed response time constraint (20 seconds) for workers than that of Bernstein *et. al*'s work ($\leqslant$ 5 seconds.) Therefore the response rate reported in this section is higher. During our deployment, the average response time from the retainer is 7.703 seconds (SD = 4.679, all cases included.)

In sum, during our deployment, the retainer was shown to be able to stably recall 80% to 90% workers when tasks comes in, which suggests its mechanism is reliable to real-world use (although potentially expensive).

[4]For [No Task], workers also need to respond at the end of their waiting time to confirm that they are still available.

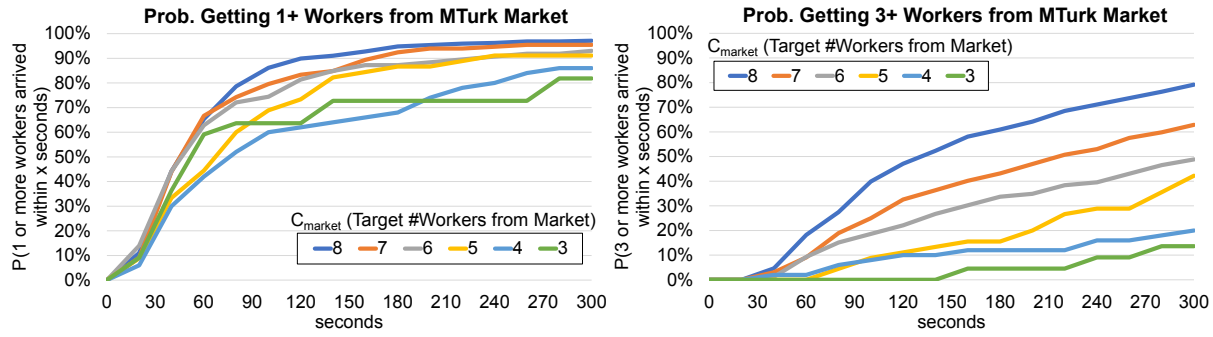


Figure 4.7: The probability of a posted HIT that have at least 1 (and at least 3) workers in the retainer at x seconds after the HIT was posted. When Ignition posted a HIT with an average of 5 assignments, 40% of the time the first worker will reach to the retainer under 2 minutes. ($N = 346, 132, 86, 45, 50, 22$)

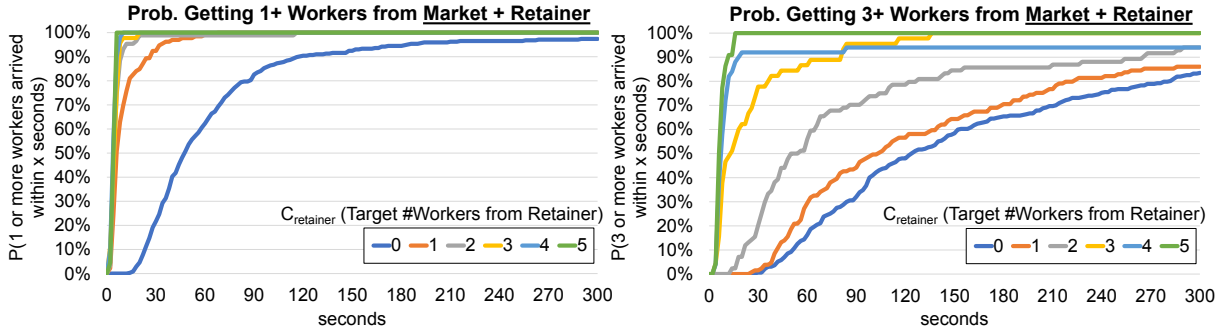


Figure 4.8: The probability that a task will have at least 1 (or 3) new worker(s) reach to the task over time, regardless of the sources of workers, given the number of workers that need to be recruited.

4.2.3 Recruiting by Ignition (Retainer & Marketplace)

Finally, for understanding the end-to-end performance of Ignition, we analyzed the probability of a started task that have at least 1 (and at least 3) workers reaching to the task (*not* retainer), regardless of the sources of workers (either from marketplace or from retainer.) The result is shown in Figure 4.8. The x-axis is the cut-off time (x seconds) after the HIT was posted, and the y-axis is the probability at x seconds. Figure 4.7 and Figure 4.8 demonstrates how our hybrid approach works inside a real-world deployed system. It is noteworthy that the lines of "$C_{market} = 8$" in Figure 4.7 (navy blue color) and the lines of "$C_{retainer} = 0$" in Figure 4.8 (navy blue color) are nearly the same (because $C_{market} + C_{retainer} = 8$.) When no workers are waiting in the retainer pool, Ignition posts more assignments to mturk marketplace to recruit workers to have a better recruiting speed (Figure 4.5); when some workers are waiting in the retainer pool, Ignition recalls workers back from retainer and thus results in a much shorter response time (Figure 4.8.)

As shown in Figure 4.4, half of tasks occurred when no workers were waiting in the retainer pool. Ignition dynamically decides the number of assignments to post based on the number of

workers in retainer, and thus helps to balance monetary cost and response speed.

4.3 Worker Survey

We believe that understanding workers' opinion and behaviors could help us further improve the design of future low-latency crowd-powered systems. While low-latency crowdsourcing has been proposed and developed for many years, literatures had little to say about workers' perspective of these technologies, nor how they work with these tasks. Therefore, we designed a questionnaire to collect opinions and self-reported behavior for 156 workers who have completed at least 10 HITs in the Ignition system. We posted the survey as a $2.0 HIT on Amazon Mechnical Turk for two weeks, and contacted these workers to participate in the survey. A total of 101 workers finished the survey, *i.e.*, the response rates is 64.74%. Each worker on average took 10 minutes 20 seconds to finish to the questionnaire.

4.3.1 Workers' Opinion about Retainer HITs

In the survey, we asked workers to rate if they like *(i)* "doing HITs on MTurk in general" and *(ii)* "doing retainer HITs,"[5] which we referred to as *likability* score. Responses were collected on a five-point Likert scale (strongly disagree = 1, and strongly agree = 5). As a result, the average likability rating of general HITs is significantly higher than that of the retainer HITs ($p = 0.0058$). Workers reported an average of 4.47 points (SD = 0.81) for general HITs, and 4.12 points (SD = 1.00) for retainer HITs. The distributions of rating points are shown in Figure 4.9.

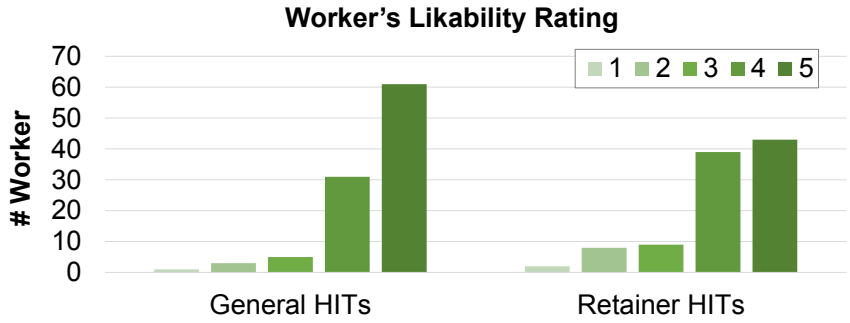


Figure 4.9: Answers to Likert scale questions on our survey indicating that workers like doing *(i)* HITs on MTurk in general and *(ii)* retainer HITs. The average likability rating of general HITs is significantly higher ($p = 0.0058$).

[5]In the questionnaire we defined a "retainer HIT" as follows: *Unlike a typical HIT that rewards a worker for completing a task, a HIT served by a retainer system pays a worker for "waiting for a task to appear" in addition to completing the actual task.*

What Workers Disliked

Based on our experience of deploying Ignition, we expected that workers would prefer regular HITs over retainer HITs. In the survey, we further asked workers to rate the extent to which they *disliked* the four main aspects that workers have complained about: *(i)* needing to wait until work is available, *(ii)* committing their next 30 minutes, *(iii)* responding to the recall alert with 20 seconds, and *(iv)* believing the payment for waiting time is too low. Workers reported feeling most disliking the payment rate for their waiting time, for which the average dislikability rating is 3.64 (SD = 1.22). The next factor was the requirement to respond quickly, in which the average rating was 3.09 (SD = 1.41). The third is the requirement that workers need to commit 30 minutes, for which the average rating was 2.76 (SD = 1.31). The least dislikable factor was needing to wait, in which the average rating was 2.52 (SD = 1.24). The rating distributions are shown in Figure 4.10. We also asked workers to rate on a five-point Likert scale how *easy* it is for them to commit 30 minutes and to respond within 20 seconds. Workers on average find it slightly easier to commit their time (3.75, SD = 1.13) than to respond to the recall alert quickly (3.70, SD = 1.24).

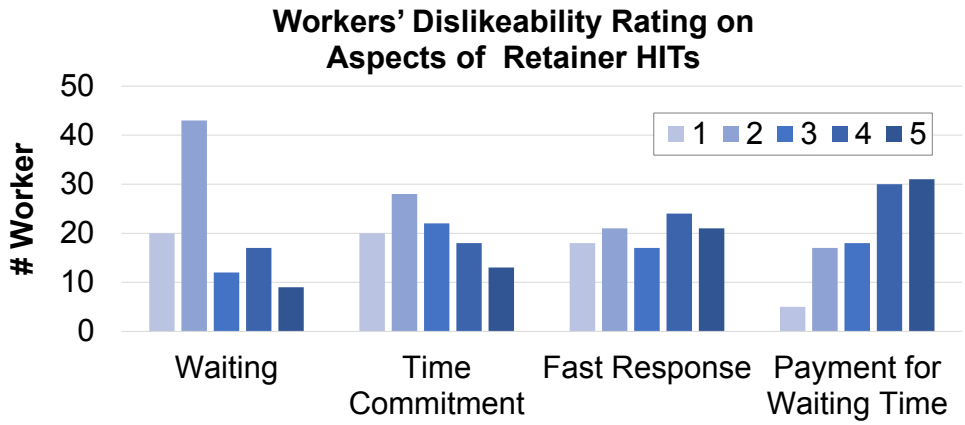


Figure 4.10: Answers to Likert scale questions on our survey indicating that workers dislike *(i)* waiting, *(ii)* committing time, *(iii)* responding to the recall alert fast, and *(iv)* getting paid lower for their waiting time of retianer HITs. As a result, workers feel most unsatisfied about the lower payment for their waiting time, and least concern about the fact that they need to wait.

Furthermore, we asked workers to provide some other dislikable factors which were not covered in these four items. 5 workers mentioned that the retainer HITs are likely "buggier" than regular HITs. One worker said that while our HITs were fine, "I've seen the issue with other retainer hits." Another worker also said "Maybe it didn't always operate smoothly." Meanwhile, most workers echoed their thoughts on these four aspects, especially about the lower wage for their waiting time, instead of mentioning new dislikable factors. Workers suggested the ideal wage they expected for waiting. One worker said, "it's hardly \$0.20 + 0.05 to wait. It should be at least \$0.40 +0.10"; a worker mentioned another requester provided \$0.50 for 10-minute waiting time.

What Workers Liked

We asked workers to answer in free text what they *like* about the retainer HITs. The following three themes emerged in the collected responses: *(i)* getting paid for simply waiting, *(ii)* the waiting page have a large clear timer to show the amount of accumulated bonus and remaining time, and *(iii)* being able to work on other HITs or do other things in parallel.

34.7% of workers mentioned they like to get paid for just waiting. As one worker said, "...(Your HITs) give a sense of 'not waiting in the dark' since they pay you to wait (not even having focus to your tab). This is a form of respect to the turkers and predisposes me to do my best to complete your projects." Workers also pointed out that some other requesters did not pay for their waiting time if they did not encounter any tasks. For example, one worker said, "I liked that even if there was no work i was still getting paid something just to wait, most other requesters don't give that courtesy." 29.7% of workers mentioned that they like the design of our waiting page, especially it has a big timer showing the bonus and remaining time. As one worker put it, "...the way the system shows points increasing with the time and the minimum amount of time one needs to wait to submit the hit is useful information and helps keep ones attention to the task." Another workers said, "I like that the countdown and the bonus totals change in real time and I can see how much time is left." One other worker also said "I think it is really great that you have a waiting page, I wish more requesters did." 25.7% of workers mentioned that they like the fact that they are free to work on other HITs or do other things during waiting. For instance, one worker commented, "We can work or do other things as well if there is no task assigned, so we can utilize our time effectively because of the recall alert and waiting page feature."

4.3.2 How Do Workers Work with Retainer HITs

We asked workers the browsers[6] and tools (*e.g.,* browser extensions) they used to keep track of our HITs. 33.7% of worker said they used browser extensions to subscribe to our HITs on Amazon Mechanical Turk. We then asked these workers which tools or extensions they have been using. The following are all the browser extensions mentioned, along with the number of workers mentioned it: HIT Scraper (12), Turkmaster (6), JR Mturk Panda Crazy (5), Mturk Suite (3), Hit Finder (2), HIT Notifier (2), Openturk (1), Overwatch for worker.mturk (1), HIT Monitor (1), "Greasemonkey scripts" (1), and "a auto reload tool" (1).

We also asked workers "Is there any forum or community you use to keep track of our HITs?" 28.7% of workers said yes and reported the forum they use. The followings are the on-line communities worker reported using, along with the number of workers who mentioned using it: MturkCrowd.com (10), "worker forums" (4), TurkerNation (4), "Hits Worth Turking For" Reddit group (3), TurkerHub (3), TurkOpticon (3), mturkgrin (1), Turkalert (1), and "a whatsapp group" (1).

Furthermore, we asked workers "What do you usually do when waiting on the countdown timer page?". 79.2% of workers usually do other HITs in parallel; 15.8% of workers usually do something else on their computers in parallel instead of doing other HITs; and 2% of workers do not use computer in parallel, but do something else (*e.g.,* watching TV) instead. While the majority of workers take other HITs when they are waiting, around 20% of workers do something

[6] 87.2% of workers used Google Chrome, and 18.8% of workers used Mozilla Firefox.

else instead, even not in front of their computers. Allowing them to choose a louder or more aggressive notification is likely helpful to recall them back.

We also asked workers "If something unexpected happen during your waiting time and you have to leave, what do you usually do?". 46.5% of workers said they usually leave the browser open, just in case he/she could be back soon; 44.6% of workers usually return the HIT and leave; and 5.9% of workers just close the browser directly. Since many workers are willing to come back to continue waiting after they were interrupted, enabling them to pause and resume on the waiting page could be helpful. However, since the task distribution over retainer time is not uniformly distributed (Figure 4.6(c)), a more sophisticated mechanism might be needed for preventing workers from abusing a pausing feature (*e.g.*, disallowing workers to pause during the first 5 minutes, or setting a maximum pause time.)

4.4 Discussion

Delay-Cost Trade-offs As expected, there are trade-offs between response speed and cost when recruiting workers on MTurk. Maintaining a retainer pool can result in a very short response time, however, is also expensive. Given our current rate, which is $0.25 per 33 minutes, a base rate of running a full-time retainer can be calculated as follows. If we maintain a 5-worker retainer for 24 hours, it would cost $65.45 per day (including MTurk's 20% fee), $458.18 per week, or approximately $2,000 per month. This price does not consider the possible refillings of retainer when Chorus requires more than 5 workers to support multiple conversations at the same time. Ignition' cost is basically a function of task numbers, getting rid of the basic rate that needed to maintain a retainer pool.

It may be possible to learn optimal policies for recruiting based on a budget. Our results suggest that variables such as the number of workers waiting, the number of workers already recruited, the time workers have already spent in the retainer, each worker's prior response rate/-time, etc. could be inputs to such a model. Furthermore, given workers generally expect a higher wage for their waiting time, the hourly wage of the retainer and task also likely play roles. For systems that will be deployed over long periods, it may be useful to model the observed latency recruiting workers from the marketplace – some applications may not even need to use a retainer, relying instead on the natural latency afforded by the marketplace itself.

Task-Dependent Factors It is noteworthy that in this project we only measure workers' *arrivals* to the waiting page and tasks, but not their completions nor performances on tasks. In other words, even if Ignition is able to recruit workers quickly with reasonable financial cost, workers can still return the HIT or not complete the HIT until it expires, even when they reach the HIT quickly. In the early stage of deploying Chorus [74], many workers return our HITs not because of the Ignition recruiting system, but the unfamiliarity of Chorus tasks.

4.5 Summary

In this chapter, we have introduced *Ignition*, an approach that combines both on-demand recruiting and the retainer model to bring workers to tasks from Amazon Mechanical Turk. We have explored deployment of Ignition over 10 months to support a medium-sized crowd-powered system deployment, Chorus, finding that it reasonably balanced the cost and latency of recruiting workers. The chapter discusses the observed stability, timing, and retention of workers using the model, demonstrating the feasibility of on-demand recruitment over time.

In the future, it would be interesting to explore how our findings are affected by increased load on the crowd marketplace, and how these results might change if we instead consider a large, active deployment in which many more workers are involved. Furthermore, prior workers have explored using old tasks [14] or micro diversion breaks [38] to engage workers longer, which could be incorporated in future Ignition frameworks. Future research may also consider the effect of continuity on worker response time and recruitment, as prior work has found that this can affect quality [95]. For instance, it may be that workers are more easily brought back to work on another task after they have finished a prior one. It is also likely that the variables we measure change over time, as workers learn of the task and adapt to it, or as the underlying market changes. It would thus be interesting to compare the evolution of more than one system, on more than one platform, over a long period of time.

Part II

A Framework That Automates Chorus Over Time

Chapter 5

Evorus: A Crowd-powered Conversational Assistant That Automates Itself Over Time

Conversational assistants, such as Apple's Siri, Amazon's Echo, and Microsoft's Cortana, are becoming increasingly popular, but are currently limited to specific speech commands that have been coded for pre-determined domains. As a result, substantial effort has been placed on teaching people how to talk to these assistants, *e.g.*, via books to teach Siri's language [126], and frequent emails from Amazon advertising Alexa's new skills [2]. To address the problem of users not knowing what scenarios are supported, in 2017, AI2 built an Alexa skill designed to help people find skills they could use, only to have it rejected by Amazon [35].

Crowd-powered assistants such as Chorus are more robust to diverse domains, and are able to engage users in rich, multi-turn conversation. Despite their advantages, crowd-powered agents remain largely impractical for deployment at large scale because of their monetary cost and response latency [14, 74]. On the other hand, crowd-powered systems are often touted as a path to fully automated systems, but transitioning from the crowd to automation has been limited in practice. The most straightforward approach is to use data from prior conversations to train an automated replacement. This can work in specific domains [168], or on so-called "chit-chat" systems [7]. Fully automating a general conversational assistant this way can be difficult because of the wide range of domains to cover and the large amount of data needed within each to train automated replacements. Such automated systems only become useful once they can completely take over from the crowd-powered system. Such abrupt transition points mean substantial upfront costs must be paid for collecting training examples before any automation can be tested in an online system.

In this chapter, we explore an alternative approach of a crowd-powered system architecture that supports gradual automation over time. In our approach, the crowd works with automated components as they continue to improve, and the architecture provides narrowly scoped points where automation can be introduced successfully. For instance, instead of waiting until an automated dialog system is able to respond completely on its own, one component that we developed recommends responders from a large set of possible responders that might be relevant based on the on-going conversation. Those responses are then among the options available to the crowd to choose. Another component learns to help select high-quality responses. Each problem is tightly scoped, and thus potentially easier for machine learning algorithms to automate.

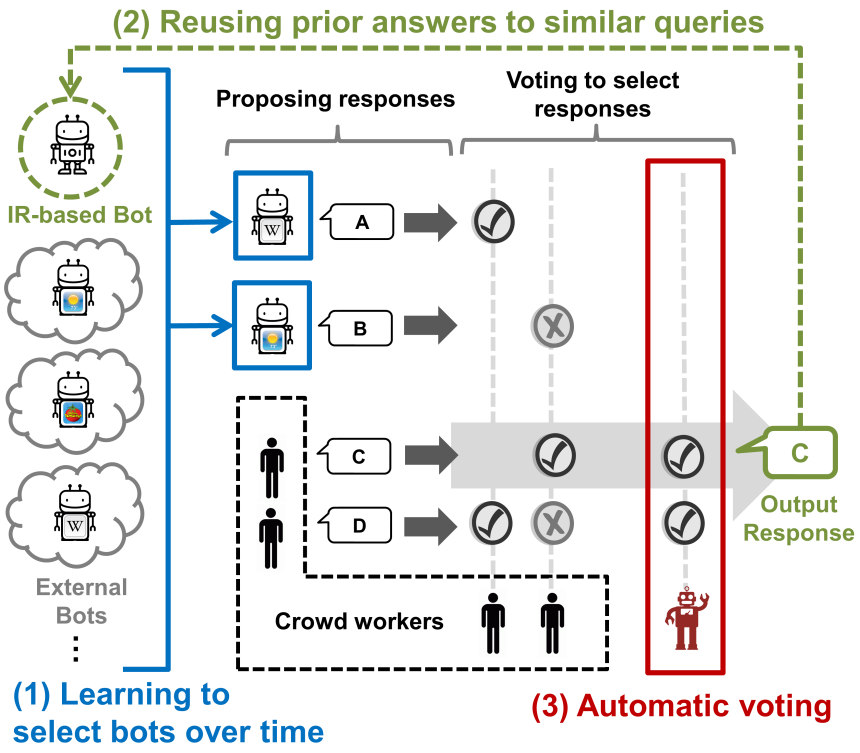


Figure 5.1: Evorus is a crowd-powered conversational assistant that automates itself over time by *(i)* learning to include responses from chatterbots and task-oriented dialog systems over time, *(ii)* reusing past responses, and *(iii)* gradually reducing the crowd's role in choosing high-quality responses by partially automating voting.

We introduces Evorus, a crowd-powered conversational agent that provides a well-scoped path from crowd-powered robustness to automated speed and frugality. Users can converse with Evorus in open domains, and the responses are chosen from suggestions offered by crowd workers and *any number* of automated systems that have been added to Evorus. Evorus supports increased automation over time in three ways (Figure 5.1): *(i)* allowing third-party developers to easily integrate automated chatterbots or task-oriented dialog systems to propose response candidates, *(ii)* reusing crowd-generated responses from previous conversations as response candidates, and *(iii)* learning to automatically select high-quality response candidates to reduce crowd oversight over time.

In Evorus, existing dialog systems can be incorporated via simple REST (REpresentational State Transfer) interfaces that take in the current conversation context, and respond with a response candidate. Over time, Evorus learns to select a subset of the automated components that are most likely to generate high-quality responses for different context. The responses are then forwarded to crowd workers as candidates. Workers then choose which of the responses to present to the users. Evorus sees workers selecting responses from candidates as signals that enable it to learn to select both automated components and response candidates in the future. It is important to note that while Evorus is a functioning and deployed system, we do not see the current version and its constituent components to be final. Rather, its architecture is designed to

allow future researchers to improve on its performance and the extent to which it is automated, by working on constituent problems, which are each challenging in their own right. The structure of Evorus provides distinct learning points that can be bettered by other researchers. Others may include additional dialog systems or chatterbots, and improve upon its learning components, driven by the collected data and its modular architecture.

We deployed the current version of Evorus over time to better understand how well it works. During our deployment, automated response were chose 12.44% of time, Evorus reduced the crowd voting by 13.81%, and the cost of each non-user message reduced by 32.76%. In this project, we explore when the system was best able to automate itself, and present clear opportunities for future research to improve on these areas.

This project makes four primary contributions:

- **Evorus Architecture:** a crowd-powered conversational assistant that is designed to gradually automate itself over time by including more responses from existent chatbots and reduce the oversight needed from the crowd;
- **Learning to Choose Chatbots Over Time:** we introduced a learning framework that uses crowd votes and prior accepted message to estimate the likelihood of each chatbots when receiving a user message;
- **Automatic Voting:** we implemented a machine learning model for automatically reducing the amount of crowd oversight needed, evaluated its performance on a dataset of real conversations, and developed a mathematical framework to estimate the expected reward of using the model; and
- **Deployment:** we deployed Evorus for over 5 months with 80 participants and 281 conversations to understand how the automatic components we developed could gradually take over from the crowd in a real setting.

5.1 Evorus Framework: Chorus Part

Evorus' uses Chorus' architecture to hold conversations, which obtains multiple responses from multiple sources, including crowd workers and chatbots, and uses a voting mechanism to decide which responses to send to the end-user. In this section, we describe Evorus' conversational assistant framework that basically follows that of Chorus.

Worker Interface Evorus' worker interface contains two major parts (Figure 5.2): the *chat box* in the middle and the *fact board* on the side. Chat box's layout is similar to an online chat room. Crowd workers can see the messages sent by the user and the responses candidates proposed by workers and bots. The role label on each message indicates it was sent by the user (blue label,) a worker (red label,) or a bot (green label.) Workers can click on the check mark (✔) to *upvote* on the good responses, click on the cross mark (✖) to *downvote* on the bad responses, or type text to propose their own responses. Beside the chat box, workers can use the fact board to keep track of important information of the current conversation. To provide context, chat logs and the recorded facts from previous conversations with the same user were also shown to workers.

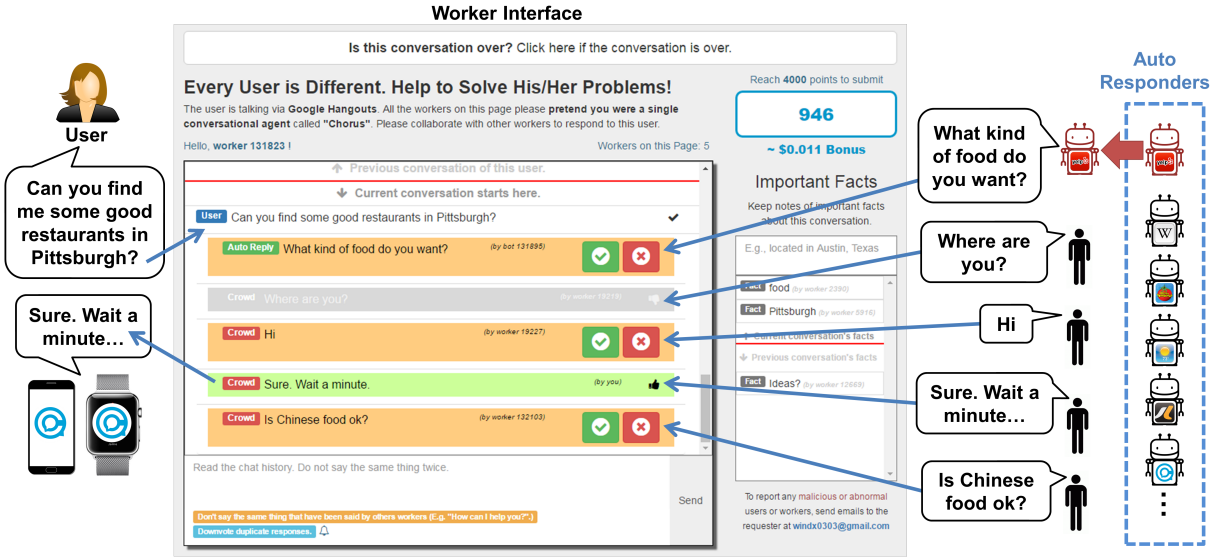


Figure 5.2: The Evorus worker interface allows workers to propose responses and up/down vote candidate responses. The up/down votes give Evorus labels to use to train its machine-learning system to automatically gauge the quality of responses. Evorus automatically expires response candidates upon acceptance of another in order to prevent workers from voting through candidate responses that are good but no longer relevant. Workers can tell each message is sent by the end-user (blue label), a worker (red label), or a chatbot (green label) by the colored labels.

The score board on the upper right corner displays the current reward points the worker have earned in this conversation. If the conversation is over, the worker can click the long button on the top of the interface to leave and submit this task.

Selecting Responses using Upvotes and Downvotes Crowd workers and bots can upvote or downvote on a response candidate. As shown in Figure 5.2, on the interface, the upvoted responses turned to light green, and the downvoted responses turned to gray. Crowd workers automatically upvote their own candidates whenever they propose new responses. Upon calculating the voting results, we assigned a negative weight to a downvote while an upvote have a positive weights. We empirically set the upvote weight at 1 and downvote's weight at 0.5, which encourages the system to send more responses to the user. We inherited the already-working voting threshold from deployed Chorus [74], which accepts a response candidate when it accumulates a vote weight that is larger or equal to 0.4 times number of active workers in this conversation. Namely, Evorus accepts a response candidate and sends it to the user when Equation (5.1) holds:

$$\begin{aligned} &\#upvote \times W_{upvote} - \#donwvote \times W_{downvote} \\ &\geqslant \#active\_workers \times threshold \\ W_{upvote} = 1.0, \quad &W_{downvote} = 0.5, \quad threshold = 0.4 \end{aligned} \tag{5.1}$$

We formally defined the *#active_workers* in the later subsection of real-time recruiting. Evorus does not reject a message, so it does not have a threshold for negative vote weight.

Expiring Unselected Messages to Refresh Context When Evorus accepts a response, the system turns the accepted message to a white background, and also *expires* all other response candidates that have not been accepted by removing them from the chat box in the worker interface. This feature ensures all response candidates displayed on the interface were proposed based on the latest context. We also created a "proposed chat history" box on the left side of worker interface, which automatically records the worker's latest five responses. Workers can copy his/her previously-proposed response and send it again if the message expired too fast.

A Proposed, Accepted, or Expired Message In Evorus, non-user messages are in one of three states: [Proposed], [Accepted], or [Expired]. [Proposed] messages are open to be up/downvoted. These messages were proposed by either a worker or a bot, has not yet received sufficient votes to be accepted, and has not yet expired; [Accepted] messages received sufficient votes before they expired and were sent to the user; and [Expired] messages did not receive sufficient votes before they expired. These messages were not sent to the users, and were removed from the worker interface. A [Rejected] state does not exist since Evorus does not reject a message proactively.

Worker's Reward Point System To incentivize workers, Evorus grants reward points to workers for their individual actions such as upvoting on a message or proposing a response candidate, and also for their collective decisions such as agreeing on accepting a message or proposing a message that were accepted. The score box on the right top corner of the interface shows the current reward points to the worker in real-time. Reward points are later converted to bonus pay for workers. Without compromising output quality, if some of these crowd actions can be successfully replaced by automated algorithms, the cost of each conversation can be reduced. Evorus' reward point schema was extended from the Chorus reward schema, which was previously used during its year-long deployment [74]. This schema encodes the importance of each action, and thus provides a good guide for algorithms to estimate the benefit and risk when automating a crowd action. Moreover, this reward schema will be used to estimate the expected reward points (and corresponding costs) an automatic voting bot can save, which we describe later.

Real-time Recruiting & Connecting to Google Hangouts When a conversation starts, Evorus uses the Ignition model [67] (Chapter 4) to recruit workers quickly and economically from Amazon Mechanical Turk, and uses the Hangoutsbot [57] library to connect with the Google Hangout servers so that users can use its clients to talk with Evorus on computers or mobile devices. Each conversation starts with 1 worker and incorporates 5 workers at most. Workers may reach a conversation at different times, but typically stay to the end of the conversation (average duration $\simeq$ 10 minutes). The *#active_workers* in Equation 5.1 is defined as "the number of crowd workers who were working on this conversation when the message was proposed," which varies as workers arrive (or drop out) at different times. In our deployment, the average *#active_workers* of all crowd messages is 3.56 (SD=1.29) and 77.56% of the crowd messages had *#active_workers* $>=$ 3.

5.2 Evorus Framework: Automation and Learning Part

Evorus is a conversational assistant that is collaboratively run by real-time crowdsourcing and artificial intelligence. The core concept of Evorus is to have crowd workers work with automated virtual agents (referred to as "bots") on the fly to hold sophisticated conversations with users. To test this, we developed two types of bots: the automatic response generators, *i.e.*, **chatbots**, and the automatic voting algorithms, *i.e.*, **vote bots**. Evorus monitored all ongoing conversations, and periodically called chatbots and vote bots to participate in active conversations. Both chatbots and vote bots take the entire chat log as input, and based on the chat log to generate responses or votes. To coordinate with human workers' speed, Evorus often needs to set constraints on the frequency or capability (*e.g.*, in which condition can a chatbot propose responses) of bots. More importantly, Evorus can learn from the crowd feedback to automate itself over time via three primary mechanisms: *(i)* the chatbot selector, *(ii)* the retrieval-based chatbot that can reuse old responses, and *(iii)* the automatic voting bot.

5.2.1 Part I: Learning to Choose Chatbots Over Time

In Evorus, existing dialog systems or chatbots can be incorporated by defining a simple REST interface on top of them that accepts information about the current conversation state, and responses with a suggested response. When a sufficient amount of chatbots are included in the bot pool, selecting the most appropriate chatbots to answer different questions becomes critical. For instance, a simple "ping-pong" chatbot that always responds with what it was told can be selected to reply echo questions such as "Hi" or "How are you?"; a restaurant recommendation bot can be selected when the user is looking for food; and a chatbot that was built on a friend's chat log can be selected when the user feels lonely [114]. In this project, we introduce a learning framework that uses crowd votes and prior accepted messages to estimate the likelihood of each chatbot when receiving a user message. The learning framework naturally assigns a slightly higher likelihood to newly-added chatbots to collect more data. The beauty of this design is that any chatbot can contribute, as long as it can effectively respond to – even a small – set of user messages.

5.2.2 Part II: Reusing Prior Answers

Upon receiving a message from the user, Evorus uses a retrieval-based approach to find the most similar message in prior conversations and populates its prior response for the crowd to choose from. By doing so, Evorus is capable to reuse the answer of prior similar questions to respond to users. The advantage of using a retrieval-based method is that it naturally increases its capability of answering questions with the growth of the collected conversations, without the need of recreation or retraining of machine-learning models. With the oversight of the crowd, the retrieval-based approaches also do not need to be perfect to contribute. As long as it find good responses to a portion of user conversations, the learning framework described in Part I can gradually learn when to use it.

5.2.3 Part III: Automatic Voting

Closing the loop of automating the entire system, the last piece is to automate the oversight of the crowd that are necessary for quality control, *i.e.*, the voting process in Evorus. We formulated response voting as a classification task and tackled it with a supervised machine-learning approach. A set of features based on literature, including the word, the speaker, and the time of the proposed messages are used to develop a machine-learning model, and the prior collected crowd votes are used as gold-standard labels. While the overall classifier performance is efficient in the dataset, a misfired vote (a false-positive) that mistakenly accepts a low-quality response will not only disturb the conversation, but also waste extra bonus money to crowd workers who proposed and voted for it. In this project we propose a mathematical framework to estimate expected benefits of using an automatic voting classifier.

In Evorus, both workers and the vote bot can upvote suggested responses. When a new suggestion is offered, the vote bot is called. It first calculates its *confidence* score, and, if the confidence is greater than a threshold, which is estimated by our proposed mathematical framework, the vote bot automatically upvotes the message. Evorus monitors the latest down/upvotes and calculates voting results in real-time. When a crowd message collects sufficient vote weight, Evorus *(i)* accepts it and sends it to the user, and *(ii)* removes all other candidate messages from the worker interface to refresh context.

In the following three sections, we describe in detail the three main parts of the Evorus framework.

5.3 Part I: Learning to Choose Chatbots Over Time

Evorus' chatbot selector learns over time from the crowd's feedback to choose the right chatbots to respond to user messages. Evorus also **regularly populates lower-ranking chatbots** to allow the model to learn about new chatbots and tp keep the model up-to-date.

5.3.1 Ranking and Sampling Chatbots

Upon receiving a message from a user, Evorus uses both the text and prior collected data to estimate how likely each chatbot is capable of responding the user (*i.e.,* $P(bot|message)$). We used a conditional probability, as shown in Equation 5.2, to characterize the likelihood of selecting a chatbot (bot) after receiving a user $message$.

$$\begin{aligned} &P(bot|message) = P(bot) \times P(message|bot) \\ &\approx P(bot) \times similarity(message,\ history_{bot}) \end{aligned} \tag{5.2}$$

$P(bot)$ is the prior probability of the chatbot, and $P(message|bot)$ is the likelihood of the user message given the chatbot's history (*i.e.*, previous user messages that the bot has successfully responded). While training an n-gram language model using previous messages to estimate $P(message|bot)$ is intuitive [56], sufficient data for building such model is often unavailable for newly-added bots. To generalize, we used a similarity measure based on distance between word

vectors ($similarity(message,\ history_{bot})$) to approximate this likelihood. We will explain how we calculate these two components in the following subsection.

Equipped with the estimates, Evorus ranks all the chatbots based on the likelihood values, and always calls the first-ranking chatbot to provide its response. More interestingly, in addition to the top chatbot, Evorus also randomly selects a lower-ranking chatbot to provide responses. By doing so, Evorus is capable to gradually update its estimates of each bot based on the crowd feedback and learn over time the best scenario to call each chatbot. Similar strategies, such as the *epsilon-greedy strategy* that yields a small portion of probability for random outcomes and collects feedback, have been used in models that learn to select crowd workers [145] and dialogue actions [132]. For the new chatbot, Evorus initially assigns a starting probability to it to allow the system to collect data about it, which we describe in the following subsection.

5.3.2 Estimating Likelihood of a Chatbot

We aimed at designing a learning framework that is *(i)* inexpensive to update, since we want the model to be updated every single time when the system receives a new label, and *(ii)* allows new bots to be added easily.

Prior Probability of Chatbots: To generate more reliable prior estimation for newly-added bots with limited histories, we used a beta distribution with two shape parameters α and β (Equation 5.3) to model the prior probability of each chatbot.

$$P(bot) \approx \frac{(\text{\#accepted messages from bot}) + \alpha}{(\text{\#user messages since bot online}) + \alpha + \beta} \tag{5.3}$$

$P(bot)$ can be interpreted as the *overall acceptance rate* of the chatbot without conversation contexts. The two shaping parameters α and β can be viewed as the number of accepted (positive) and not-accepted (negative) messages that will be assigned to each new chatbot to begin with, respectively. Namely, any new chatbot's prior probability $P(bot)$ will be initially assigned as $\alpha/(\alpha+\beta)$, and then later be updated over time. The beta distribution's α and β are both functions of the mean (μ) and variance (σ^2) of the distribution. In our pilot study, in which each automatic response requires only one vote to be accepted, four chatterbots had an average message acceptance rate of 0.407 (SD=0.028.) Since we increased the required vote count from 1 to 2 in the final deployment, a lower acceptance rate is expected. We used $\mu = 0.3$ and $\sigma = 0.05$ to estimate the shape parameters, where $\alpha = 24.9$ and $\beta = 58.1$.

Similarity between Messages and Chatbots To estimate $similarity(message,\ bot)$, we first used the pre-trained 200-dimension GloVe word vector representation trained on Wikipedia and Gigaword [117] to calculate the average word vector of each message. We then used previous user messages that were successfully responded by the chatbot as the bot vector $\vec{w}_{bot}$ that represents the chatbot in the word-vector space. We also calculated the centroid vector of *all* user messages, $\vec{w}_{overall}$, to represent general user messages. Finally, as shown in Equation 5.4, the

similarity between a message and a chatbot is defined as the distance ratio between the vectors.

$$similarity(message,\ history_{bot}) := \frac{dist(\vec{w}_{message}, \vec{w}_{overall})}{dist(\vec{w}_{message}, \vec{w}_{bot}) + dist(\vec{w}_{message}, \vec{w}_{overall})} \tag{5.4}$$

While $\vec{w}_{bot}$ can be calculated as the centroid vector of prior user messages that were successfully responded to by the chatbot, in cold-start scenarios, a chatbot will not have sufficient accepted messages to calculate the vector. We provide two solutions for chatbot developers: First, the developer can provide a small set of **example messages** where their chatbots should be called. For instance, the developer of an Yelp chatbot can list "*Find me a sushi restaurant in Seattle!*" as an example. Evorus will treat these example messages as the user messages that the chatbot successfully responded to, and use their centroid vector as the initial $\vec{w}_{bot}$. When more messages are accepted, they will be added into this set and update the vector. Second, for some chatbots, especially non-task chatterbots, it could be difficult to provide a set of examples. Therefore, if the developer decided not to provide any example messages, we set the initial $dist(\vec{w}_{message}, \vec{w}_{bot}) = 0$ for new chatbots.

5.4 Part II: Reusing Prior Responses

Evorus uses an information-retrieval-based (IR-based) method to find answers to similar queries in prior conversations to suggest as responses to new queries. To do so, Evorus first extracts query-response pairs from all the old conversations, and then performs a similarity-based sorting over these pairs.

5.4.1 Extracting Query-Response Pairs

One advantage Evorus has is that each accepted response has crowd votes, which can be used as a direct indicator of the response's quality. For each turn of a conversation between one user and Evorus, we extracted the accepted crowd *response* which did not receive any downvotes, along with the user message *(query)* it responded to, as a *(query, response)* pair. Since the deployed Evorus has not had any prior conversation with users yet, we obtained conversation data that were collected by the deployed Chorus, which also used crowd voting to select responses, during May, 2016 to March, 2017 to start with. We further removed the messages from known malicious workers and users, also removed all conversations where the users are co-authors or collaborators of Chorus [74]. At the beginning of the Evorus deployment, 3,814 user messages were included, and each of these user message is paired with 1 to 5 crowd responses.

5.4.2 Searching for the Most Similar Query

For the *query* message in each *query-response* pair, we calculated its average word vector by using the pre-trained 200-dimension GloVe word vector representation based on Wikipedia and Gigaword [117] and stores the vector in the database. When Evorus receives a user message,

the system first calculates its average word vector $\vec{w}_{message}$ using the same GloVe representation, and then searches in the database to look for the top k responses that their corresponding queries' word vectors had the shortest Euclidean distances with $\vec{w}_{message}$. Finally, for increasing answer's diversity, the system randomly selects one from top k responses to send back to Evorus for the crowd to choose from. We empirically set $k = 2$ in our deployment.

5.5 Part III: Automatic Voting

Evorus uses supervised learning to vote on responses.

5.5.1 Data Preparation

The voting mechanism has been proven to be useful in selecting good responses and holding conversations in the lab prototype [90] and deployed system [74]. The final status (*i.e.*, accepted or not) of a messages is a strong signal to indicate its quality. However, when we used this data to develop an AI-powered automated voting algorithm, it is noteworthy that expired messages were not all of lower quality. In some cases the proposed response was good and fitted in the old context well, but the context changed shortly after the message was sent; Some messages were automatically accepted and bypassed the voting process because Evorus does not have enough active workers, *i.e.*, when $0.4 \times \#active\_workers < 1$ in Equation 4.1); Furthermore, since downvotes can cancel out upvotes, the voting results in Evorus could be influenced by race conditions among workers. For instance, when two upvotes of a message has been sent to the server, Evorus might decide that this message' vote weight is sufficient and sent it to the user, in which a belated downvote would deduct its vote weight to lower than the threshold. Therefore, the training data needs to be carefully developed.

Similar to Part II, we used voting data collected during the Chorus deployment [74] to train the initial machine learning model for voting. We first extracted the expired messages with one or more downvote(s) as examples of "downvote", and extracted the accepted crowd messages with both one or more upvote(s) and zero downvote as examples of "upvote." We excluded the automatically-accepted messages that were sent when the task did not have sufficient number of active workers ($0.4 \times \#active\_workers < 1$) From all the messages collected by the deployed Chorus during September 2016 to March 2017, 1,682 "upvote" messages 674 "downvote" messages were extracted to from the dataset.

5.5.2 Model & Performance

We then used this dataset to train a LibLinear [45] classifier. For each message, Evorus extracted features in message, turn, and conversation levels to capture the dialogic characteristics [121], and also used GloVe word vector, which is identical as that of our retrieval-based response generation, to represent the content. Our approach reached to a precision of 0.740 and a recall of 0.982 (F1-score = 0.844) on the "upvote" class in a 10-fold cross-validation experiment. On the other hand, the performance of the "downvote" class is less effective. Its precision is 0.714 but

ID	Feature Description
	Message Level
1	Vocabulary size of this message
2	String length of this message
3	Worker ID of proposer of this message
	Turn Level
4	#Already-accepted crowd message in this turn
5	Duration from the latest user message (sec)
6	#Already-accepted (and Not-accepted) crowd message of this worker in this turn
7	Word vector of accepted (and non-accepted) crowd messages in this turn
	Conversation Level
8	# Accepted (and Not-accepted) messages of this worker in this conversation
9	Duration from the first accepted crowd message in this conversation
10	Duration from the latest crowd message of this worker in this conversation (sec)
11	# Already-accepted messages in this conversation
12	Message's acceptance rate of this worker in this conversation
13	# Turns in this conversation
	Word Vector (GloVe)
14	Average word vector of this message
15	Average word vector of all accepted crowd messages in this turn
16	Average word vector of all not-accepted crowd messages in this turn
17	Average word vector of all the user messages in this turn
18	Difference vectors between any two vectors in {14, 15, 16, 17}
19	Element-wise product vectors between any two vectors in {14, 15, 16, 17}

Table 5.1: Features used in automatic voting. In the feature analysis, the top three features (feature 12, 8, and 6) were all related to the performance of the worker who proposed this message according to history.

recall is only 0.134. This could be caused by the insufficient amount of training data, since Chorus promoted upvote more than downvote by design. According to this result, in the deployed Evorus, the system only automatically upvoted when the classifier ouput "upvote," but did not downvote otherwise.

5.5.3 Feature Analysis

We analyzed the features by using the 10-fold cross-validation feature selection function of the Weka toolkit [169]. The correlation attribute evaluator was used. As a result, the top three features were all related the **performance of the worker** who proposed this message according to history. The top feature was feature 12: the message acceptance rate in the current conversation of the worker who proposed this message; the second feature was the number of not-accepted

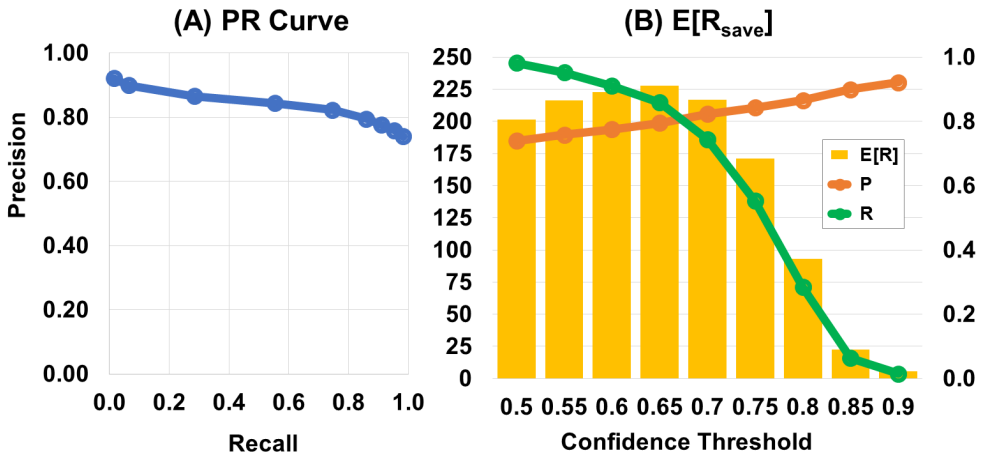


Figure 5.3: (A) The precision-recall curve of the LibLinear classifier for automatic upvoting. (B) Using our model (Equation (5.5)) to estimate the precision and recall at different thresholds and their corresponding expected reward amount saved.

messages in the current conversation of the worker (part of feature 8.) The third feature was the number of not-accepted messages in the current turn (part of feature 6.) These features indicates that the performance of the worker is a key information to predict the quality of the proposed messages. Among the top 20 features, 13 features were of particular dimensions of the **Word Vector** features in Table 5.1, including feature set 18 and 19. These suggest that the content of user's messages and crowd's messages (expired, accepted, proposed) and the dynamics between them were also an useful information. Top 20 features also contained the vocabulary size (feature 1) and string length (feature 2) of the message, the number of turns in the current conversation (feature 13), and the number of accepted messages in this conversation (feature 11).

5.5.4 Optimizing Automatic Voting

Automatic voting directly participates in the process of deciding which messages to send. While our machine-learning model resulted in good performance on our dataset, we would like to use Evorus' worker reward point schema to find the right **confidence threshold** for the automatic voting classifier. If the threshold is set too low, the classifier would vote frequently even when it is not confident about the prediction, and thus many low-quality responses would be accepted and disturb the conversation; if the threshold is set too high, the classifier would rarely vote, and the system will not gain much from using it. Liblinear can output the probability estimates of each class when performing prediction, which we used as the notion of confidence of the classifier. Thresholding out the predictions with lower confidences increased the precision but reduced the recall of the classifier. Figure 5.3(A) shows the Precision-Recall curve of Evorus' upvoting classifier.

Possible Outcomes When the Classifier Upvotes To find confidence thresholds for Evorus, we introduced the following heuristics to estimate reward points saved per message by using the

upvoting classifier. Consider the following cases:

1. **[Good Vote]** The classifier upvoted on a message that would originally be selected by the crowd. It saves 1 upvote reward ($R_{upvote} \times 1$) and 1 agreement reward ($R_{agreement} \times 1$) that would originally be granted to one human worker.
2. **[Bad Vote]** The classifier upvoted on a message that would originally *not* be selected by the crowd. In this case, one of the two following consequences will occur: *(i)* **[Misfire]** The message is **sent** to the user. The system grants agreement rewards to all human workers who upvoted on this message ($R_{agreement} \times \#upvoted\_workers$) and 1 successful proposal reward to the worker who proposed the message ($R_{proposal} \times 1$); and *(ii)*: **[No Difference]** The message remains **not sent**. Even with one extra upvote, this message's vote count was still insufficient to get accepted.

Estimating System's Expected Gain Given these setups, the expected reward points $E[R_{save}]$ saved per message by using the classifier can be estimated as follows:

$$E[R_{save}] = TPR \times E[Good] - FPR \times E[Bad] \tag{5.5}$$

TPR is the **true positive rate** and FPR is the **false positive rate** of the classifier. $E[Good]$ is the expected reward points saved per [Good Vote] event, and $E[Bad]$ is the expected reward points wasted per [Bad Vote] event.

In Evorus, $E[Good]$ is a constant ($R_{upvote} + R_{agreement}$). Meanwhile, [Bad Vote] event costs reward points only when the upvoted message is sent ([Misfire]). Therefore, $E[Bad]$ is decided by *(i)* how often one mistaken upvote triggers a misfire, and *(ii)* how expensive is one misfire, as follows:

$$E[Bad] = P(\textit{Misfire}|Bad) \times E[R_{\textit{Misfire}}]$$

$P(\textit{Misfire}|Bad)$ is the conditional probability of a message being sent to the user given the classifier has mistakenly upvoted on it. $E[R_{\textit{Misfire}}]$ is the expected reward points that were granted to workers in a single [Misfire] event.

Our training dataset only used the not-accepted messages with at least one downvote to form the "Downvote" class, which were less likely to be misfired after adding one extra automatic vote. For better estimating $P(\textit{Misfire}|Bad)$, we first ran the classifier on all messages that were *not* included in our training set, and within all the messages that the classifier decided to upvote, we then calculated the proportion of messages that would be sent to the user if one extra upvote were added. The rate was 0.692, which we used to approximate $P(\textit{Misfire}|Bad)$. Furthermore, based on Evorus' mechanism, $E[R_{\textit{Misfire}}]$ can be calculated as follows :

$$E[R_{\textit{Misfire}}] = R_{agreement} \times E[\#upvoted\_workers] + R_{proposal}$$

$E[\#upvoted\_workers]$ is the expected number of human workers who upvoted on the message in an [Misfire] event. Similarly, we ran the classifier on the unlabelled data, and calculated the average number of workers who upvoted on the messages that the classifier decided to upvote on. The number is 0.569 (SD = 0.731), which we used to approximate $E[\#upvoted\_workers]$.

Finally, $E[Hit] = 100+500 = 600$, and $E[R_{Bad}] = 0.692 \times (500 \times 0.569 + 1000) = 888.874$. Using Equation (5.5), we can estimate the precision and recall at different thresholds and their

corresponding reward amount (Figure 5.3(B)). According to the estimation, the best confidence threshold is at 0.65. In the deployed Evorus we selected a slightly higher precision and set the threshold at 0.7 ($P = 0.823$ and $R = 0.745$.)

5.6 Deployment Study and Results

Evorus was launched in the back end of the deployed Chorus, a crowd-powered Google Hangouts chatbot, in March 2017. While the end-users were not aware of the changes of the system from the client side, behind the scenes, our deployment had 3 phases: *(i)* Phase 1, *(ii)* Control Phase, and *(iii)* Phase 2. Phase-1 deployment started in March, 2017. We launched the system with only four chatterbots and one vote bot, without the learning component described in Part I, to understand the basics of having virtual bots working with human workers on the fly. For comparison, in May 2017 we then temporarily turned off all automation components and had the system solely run by the crowd till late August 2017, which we referred to as the Control Phase. Finally, for testing the capability of learning to select chatbots, we started Phase 2 deployment in early September. The Phase-2 deployment included several significant changes: *(i)* increasing the frequency of calling chatbots for responses, *(ii)* increasing the vote count needed to accept a response from 1 to 2, and *(iii)* incorporating the Part I learning.

To recruit users, we periodically sent emails to mailing lists at several universities and posted on social media sites, such as Facebook and Twitter. Participants who volunteered to use our system were asked to sign a consent form first, and no compensation was offered. After the participants submitted the consent form, a confirmation email was automatically sent to them to instruct them how to send messages to Evorus via Google Hangouts. The users can use Evorus as many times as they want to, for anything, via any devices that are available to them. Eighty users total talked with Evorus during 281 conversations. The Phase-1 deployment had 34 users talked to Evorus during 113 conversations, and Phase-2 deployment (till 17th September, 2017) had 26 users with 39 conversations. The Control Phase had 42 users with 129 conversations.

5.6.1 Phase 1: Chatterbots & Vote bot

Our Phase-1 deployment explored how chatbots and our vote bot could work together synchronously with crowd workers. We implemented four chatterbots (including the IR-based chatterbot using Chorus conversation data described in Part II) and a vote bot. During the Phase-1 deployment, the system only randomly selects one of four chatterbots to respond every half a minute, where the learning component described in Part I will be later included in Phase-2 deployment.

Implementing Four Chatterbots In our Phase-1 deployment, we implemented the following four chatterbots.

1. **Chorus Bot (shown as Part II):** A chatterbot that is powered by a retrieval-based method to reuse prior conversations to respond to users, which was described in Part II.
2. **Filler Bot:** A chatterbot that randomly selects one response from a set of candidates, regardless of context. We manually selected 13 common "conversation filler" in the Chorus

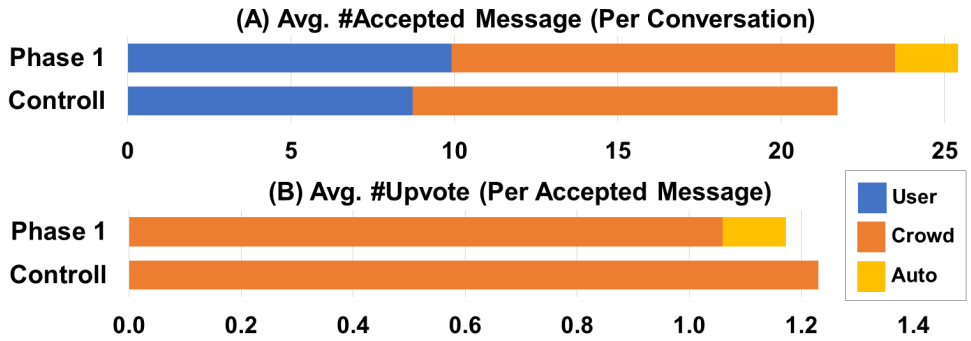


Figure 5.4: Phase-1 Deployment: (A) Average number of accepted messages per conversation. Automated responses were chosen 12.44% of the time. (B) Average number of upvotes per accepted non-user message. Human upvotes were reduced by 13.81% by using automatic voting.

dataset (*e.g.*, "Is there anything else I can help you with?", or "Thanks") to form the candidate pool.

3. **Interview Bot:** A chatterbot that uses a retrieval-based method, which is identical to Chorus Bot, to find the best response from 27,894 query-response pairs extracted from 767 transcripts of TV celebrity interview [113].
4. **Cleverbot:** Cleverbot is a third-party AI-powered chatbot which reuses more than 200 million conversations it had with users to generate responses [23, 166].

Vote Bot Setup Currrently, The vote bot only votes on human-proposed messages, but not messages proposed by chatbots. In Phase 1, Evorus requires each automatic vote to have at least one extra human upvote to be accepted. Vote bots also skipped messages if the same worker proposed identical content earlier in the conversation but did not get accepted. We believe that worker's re-sending is a strong signal of the poor quality of the message. Vote bots can decide not to vote if the confidence is too low (Part III.)

Automating Human Labors During Phase-1 deployment, a conversation on average contained 9.90 user messages (SD = 11.69), 13.6 accepted messages proposed by the crowd workers (SD = 10.44), and 13.58 accepted messages proposed by automatic chatterbots (SD = 2.81). Thus, **automated responses were chosen 12.44% of the time**. As a comparison (Figure 5.4(A)), in the Control Phase (42 users and 129 conversations), a conversation on average contained 8.73 user messages (SD = 10.05) and 12.98 accepted crowd messages (SD = 11.39). In terms of upvotes, each accepted non-user message received 1.06 human upvotes (SD = 0.73) and 0.11 automatic upvotes (SD = 0.18) In comparison, in the Control Phase, each accepted non-user message received 1.23 human upvotes (SD = 0.70.) **Crowd voting was thus reduced by 13.81%**. The comparison is shown in Figure 5.4(B). Moreover, an accepted non-user message sent by Evorus costed $0.142 in Phase-1 deployment on average, while it costed $0.211 during the Control Phase. Namely, with automated chatbots and the vote bot, **the cost of each message is reduced by 32.76%**.

We also calculated the acceptance rate of messages proposed by each chatbot. The Filler

Figure 5.5: (A) An actual conversation of Evorus. Conversations in Evorus tend to combine multiple chatbots and workers together. (B) User questionnaire used in Phase-2 deployment. The average user satisfaction rating of automated and non-automated conversations had no significant difference.

Bot, which ignores context and proposes responses randomly, had the highest acceptance rate, 41.67%. The Chorus Bot's acceptance rate was 30.99%, that of the Interview Bot was 33.33%, and that of the Cleverbot was 30.99%. This might be because Filler Bot's commonplace responses (*e.g.*, "I don't know") were often considered acceptable by human raters. In Phase 2, where an automatic response needed 2 human votes to go through, the Chorus Bot had the highest acceptance rate among all four chatterbots (Figure 5.6). While human workers, whose acceptance rate was 72.04% during Phase 1, still outperformed all chatbots by a large margin, chatbots with low accuracy can still contribute to the conversation. For instance, the Filler Bot, while being very simple and ignoring any context, nevertheless, often produces reasonable responses:

[*The user asked information about the wildfire and smoke in Emory university campus.*]

user Do you know where they are happening exactly? (The wildfires I mean)

auto-reply Can you provide some more details?

Compared with Filler Bot, Chorus Bot better targets its responses because it chooses messages based on similarity with previous human responses:

user Hey how many people like bubble tea here?

auto-reply Ask for their feedback when you talk with them

Conversation Quality We sampled conversations with accepted automatic responses and a matching set without automated contributions. For each, 8 MTurk workers rated [Satisfaction,

Clarity, Responsiveness, Comfort], which was based on the PARADISE's objectives for evaluating dialogue performance [157] and the Quality of Communication Experience metric [106], on a 5-point Likert scale (5 is best.) The original conversations (N=46) had an average rating of [3.47, 4.04, 3.88, 3.56], while those with automatic responses (N=54) had [3.57, 3.74, 3.52, 3.66]. The similar results suggest that the automatic components did not make conversations worse.

5.6.2 Phase 2: Learning to Select Chatbots

Our Phase-2 deployment explored how the learning component, described in Part I, can select the right chatbots in context, and how integrating additional chatbots affects performance. We implemented two additional *utility bots*, a Yelp Bot and a Weather Bot, that can perform information inquiry tasks for different contexts, in addition to the four chatterbots and one vote bot from Phase 1. We first launched the learning system with four chatterbots for two days, and then added the two utility bots for observing the changes of the model. Furthermore, in order to directly compare user satisfaction levels, all the automated components (including the chatbots, the vote bot, and the learning component) were only applied to 50% of the conversations randomly, and the other half of the conversations were solely run by the crowd as a baseline.

To efficiently collect crowd feedbacks to update our model, we increased the frequency Evorus called chatbots from randomly calling one chatbot and one vote bot every 30 seconds (Phase 1) to calling two chatbots (top-1 ranked plus random) and one vote bot every 10 seconds (Phase 2.) To compensate for the possible drop in quality caused by higher calling frequency and randomly selecting one of the two bots, we increased the required upvote count for accepting an automatic response from 1 vote to 2 votes. Namely, while Evorus obtained more automatic responses with a much higher frequency in Phase 2, it also required more human upvotes to approve each automatic response at the same time. As a result, among the conversations that had automation in Phase 2, automated responses were chosen 13.25% of the time, in which a conversation on average contained 10.68 user messages (SD = 8.90), 15.74 accepted messages proposed by the crowd workers (SD = 11.92), and 2.40 accepted messages proposed by automatic chatterbots (SD = 2.40). Each accepted non-user message received 1.90 human upvotes (SD = 1.13) and 0.30 automatic upvotes (SD = 0.22). We did not compare Phase 2's results in detail to that of the Control Phase because the high-frequency setup of Phase 2 is primarily for experimental exploration.

Implementing Two Utility Bots In addition to the four chatterbots in the Phase-1 deployment, we implemented the following two task-oriented utility bots:

1. **Yelp Bot:** A chatbot that suggests restaurants near the location mentioned by the user powered by the Yelp API [175]. If the user did not mention any location, it replies with "You're looking for a restaurant. What city are you in?".
2. **Weather Bot:** A chatbot that reports the current weather of a mentioned city powered by the WeatherUnderground API [148]. If the user did not mention any city names, it replies, "Which city's weather would you like to know?"

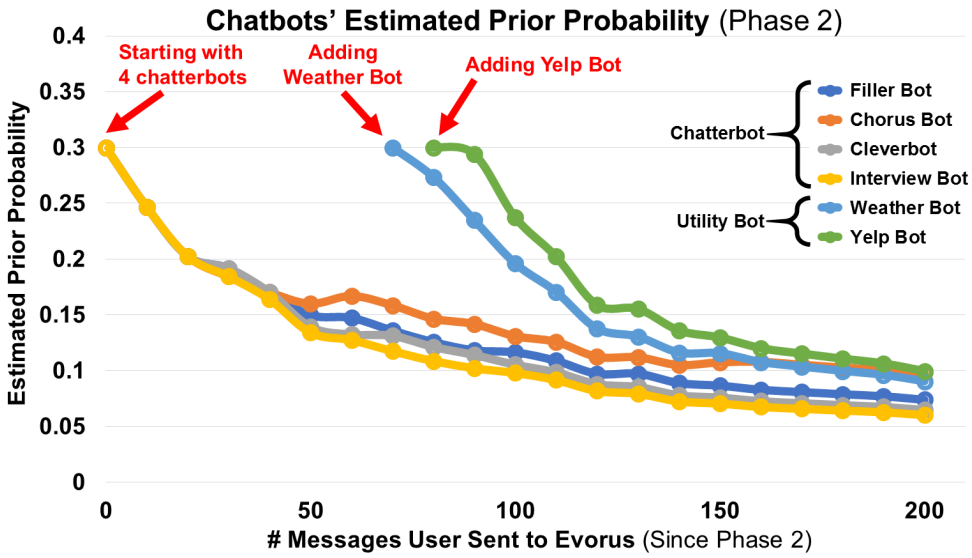


Figure 5.6: The estimated prior probability (Equation 5.3) of each chatbot was continuously updated with the growth of user messages.

The chatbots' developer (the first author) also provided three example chat messages for each that should trigger the corresponding chatbot, to initiate the learning process. For example, "Any restaurant recommendations in NYC?" for the Yelp Bot.

Similar User Satisfaction Level In Phase 2 we implemented an exit survey to measure end user satisfaction (Figure 5.5(B)). At the end of each conversation, the user had 10 minutes to report their satisfaction using a Likert-scale ranging from 1 (Very Dissatisfied) to 5 (Very Satisfied). 13 of 30 users provided feedback (response rate = 43%). The automated conversations' average user satisfaction rating was 4.50 (SD=0.5, N=4); the crowd conversations' average user satisfaction rating was 4.00 (SD=0.47, N=9), a difference that was not significant.

Updating Estimates of Chatbot's Prior Over Time While our deployment is of a medium scale, the dynamics of our likelihood model can still be observed. For instance, the estimated prior probability described in Equation 5.3 was continuously updated with the growth of conversation that Evorus had. Our model assigned a starting probability of 0.3 to each chatbots (Figure 5.6). When users started talking with Evorus, crowd workers provided their feedback by upvoting and downvoting, and thus changed the estimation over time. When new chatbots were added, Evorus intentionally assigned them a higher prior probabilities to allow quicker crowd feedback.

Utility Bots in Cold-Start Scenarios We would like to understand if Evorus can select appropriate chatbots to obtain responses in corresponding context. Since non-task chatterbots such as the Cleverbot could be difficult for humans to judge if it should be called given a message, we focused only on task-oriented utility bots in the evaluation. For each user message in the automated conversations in Phase 2, the researchers manually annotated if it is relevant to the topic

of "weather" and "restaurant," respectively. With the assumption that each utility bot should be called when its topic comes up, we compared the human-labelled topic against the top chatbot that were suggested by Evorus and calculated the precision, recall, and F1-score. It is noteworthy that we only evaluated when the appropriate chatbot was called, regardless of the quality of the responses it generated. As a result, two newly-added utility bots both had a high precision and a lower recall. The Weather Bot's precision was 1.00 and the recall was 0.47 (F1=0.64); and the Yelp Bot's precision was 0.67 and the recall was 0.20 (F1=0.31.) This result shows the nature of new bots in our learning framework: Evorus uses the expert-generated small set of examples to calculate the initial vector of each chatbot, which could result in precise predictions but with lower coverage. Over time, when Evorus collects more examples that each bot has successfully responded to and the recall increases. Moreover, conversations in Evorus tend to combine multiple chatbots and workers together. For instance, in the following conversation, the crowd had the user to narrow down the query, and then the Weather Bot was able to answer it. An additional detailed example is shown in Figure 5.5(A).

[*The user asked about the weather in Afghanistan.*]

crowd **What city in Afghanistan?)**

[*The crowd sends a website about Afghanistan weather.*]

user Kabul

auto-reply Friday's weather forecast for [Kabul, Afghanistan]: Cloudy with a few showers. High 79F. Winds NW at 5 to 10 mph. Chance of rain 30%.

Error Cases Our bot-selection algorithm starts with high precision and low recall, and increases recall as it gradually gathers examples. Therefore, most errors we observed were false-negatives, where a chatbot should have been triggered but was not. Other errors came from the chatbot, where the bot was correctly triggered but its response was invalid. Workers usually downvoted or ignored these suggestions. In the rare cases where invalid automatic responses were mistakenly sent to the user, the crowd often tried to explain how the response was automatically generated to the user afterward.

user hi! Can you summarize the features of the new iPhone for me?

[*Multiple messages list the features of the iPhone X.*]

auto-reply There is no iPhone 7.

[*The crowd lists more features, and the user says thank you.*]

crowd **No problem!**

crowd **Some Auto replies don't even make sense**

5.7 Discussion

We imagine a future where thousands of online service providers can develop their own chatbots, not only to serve their own users in a task-specific context, but also to dynamically integrate their services into Evorus with the help of the crowd, allowing users to interact freely with thousands

of online services via a universal portal. Supporting this scale offers opportunities for future research. For example, one direction is to improve the learning framework to support third-party chatbots that also improves overtime, or to better balance between the exploitation and exploration phases (like in a multi-armed bandit problem). Evorus could also be used to collect valuable fail cases to enable third-party developers to improve their bots (*i.e.*, when a bot was triggered, but its proposed response was rejected).

Most automated systems created from crowd work simply use the crowd for data; Evorus tightly integrates crowds and machine learning, and provides specific points where automated components can be introduced. This architecture allows each component to be improved, providing a common research harness on which researchers specializing in different areas may innovate and compete. For instance, "response generation" has long been developed in the NLP community; Evorus provides a natural evaluate it within a larger conversational system. The flexibility of the Evorus framework potentially allows for low cost integration between many online service providers and fluid collaboration between chatbots and human workers to form a single user-facing identity. Given the complexity of conversational assistance, Evorus is likely to be crowd-powered in part for some time, but we expect it to continue to increasingly rely on automation.

5.8 Summary

In this chapter, we introduced Evorus, a framework that enables Chorus to automate itself over time. Informed by two phases of public field deployment and testing with real users, we iteratively designed and refined its flexible framework for open-domain dialog. Evorus has three main advantages as compared to previous approaches. First, it is a working system that can serve as a scaffold for automation over time. A core advantage of starting with a working system is that users can talk to Evorus naturally from day one, ensuring conversation quality while collecting training data for automation. Second, given the oversight of the crowd, Evorus has a high tolerance for errors from its automated components. Even an imperfect automation component (*e.g.,* chatbots) can contribute to a conversation without hurting quality, which yields more space for algorithms to "explore" different actions (*e.g.,* selecting a chatbot with medium confidence.) Finally, Evorus allows a mixed group of humans and bots to collaboratively hold open conversations.

We will discussion more high-level issues in the Discussion chapter (Chapter 9), and also illustrate our blueprint for future work in the Conclusion chapter (Chapter 10.)

Part III

Building Chatbots Efficiently For Empowering Chorus

Chapter 6

Guardian: Transitioning Web APIs into Crowd-Powered Dialog Systems

In Chapter 5, we introduced Evorus, a framework that enables Chorus to uses external task-oriented chatbots in different domains and chatterbots to support part of open conversations, and thus automate itself over time. However, creating chatbots at scale is difficult. Despite frameworks which have been proposed to reduce the engineering efforts of developing a dialog system [17], constructing 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 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 Augsut 2018, ProgrammableWeb[1] alone contains the description of more than 19,924 APIs in categories including travel (1,512), reference (1,535), news services (1,541), weather (751), health (631), food (440), and many more. These Web APIs can encompass the common functions of popular existing dialog systems, such as Siri, which is often used to send text messages, access weather reports, get directions, and find nearby restaurants. Therefore, if dialog systems 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 dialog system 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?"

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

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 dialog system called Guardian (of the Dialog) [68, 69]. Guardian leverages the wealth of information in Web APIs to enlarge its scope. The crowd is employed to bridge the dialog system with the Web APIs (**offline phase**), and a user with the dialog system (**online phase**).

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

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

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

The contributions of this work are two-fold.

- We propose a Web-API based, crowd-powered 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

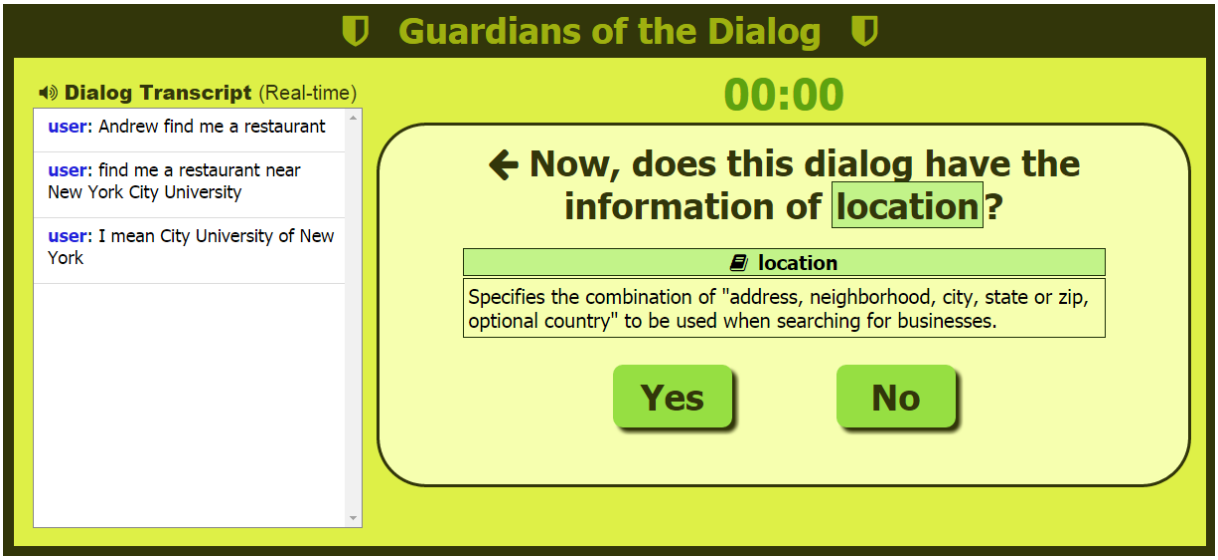


Figure 6.1: The UI for crowd workers in Guardian. The left-hand side is a chat box that displays the running dialog. The right-hand side is the working panel displaying decision-making questions.

Web APIs into a usable dialog system format. Our method has the potential to scale to thousands of APIs.

6.1 Guardian Framework

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.

6.1.1 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

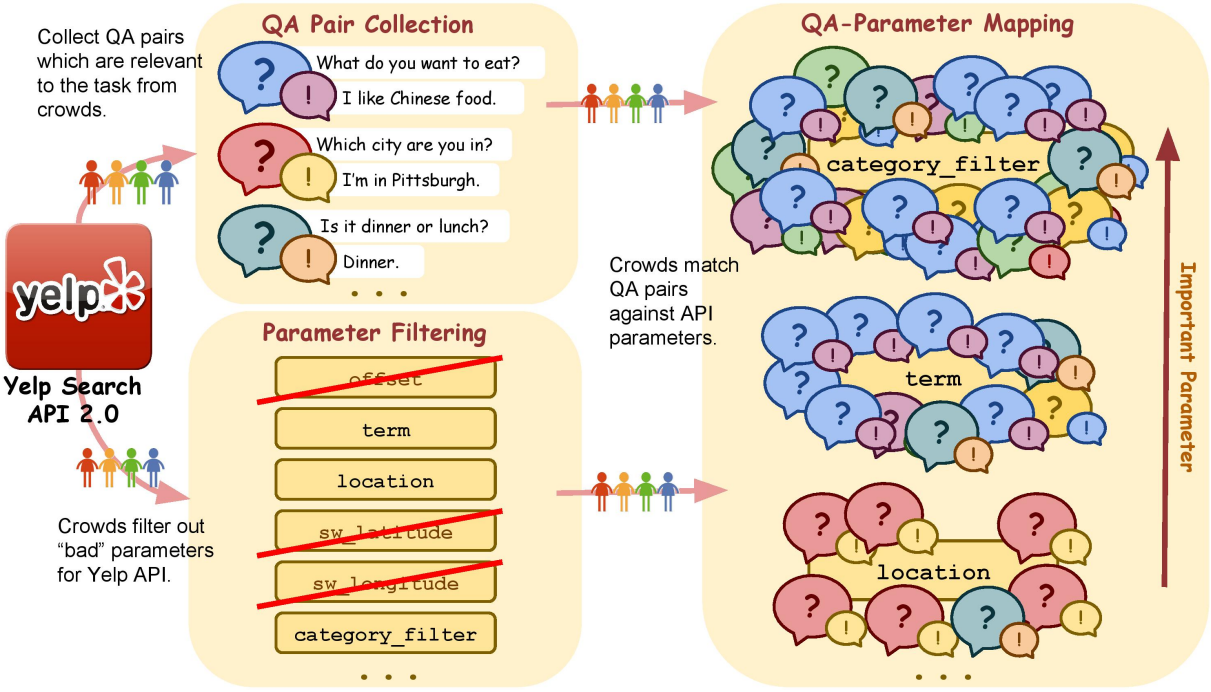


Figure 6.2: Offline Phase: A 3-stage Parameter Voting Workflow. Untrained crowd workers collect question and answer (QA) pairs related to the task, filter out unnatural parameters, and match each QA pair with the most relevant parameter.

voting-like process where unskilled workers vote for the "best" parameters for each question.

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

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

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

QA-Parameter Matching In the third stage, we match the QA pairs collected from Stage 1 against the remaining parameters from Stage 2. We display one QA pair along with all the parameters at once, and ask crowd workers the following question: "In this conversation, which of the following piece of information is provided in the answer? The followings are parameters that used in a computer system. The descriptions could be confusing, or even none of them really fit. Please try your best to choose the best one." For each represented QA pair, the workers are first required to pick one best parameter, and then rate their confidence level (low=1, medium=2, and high=3). This mechanism is developed empirically, and our experiments will demonstrate that this process could not only pick a good set of parameters for the dialog system application, but also pick good questions associated with each selected parameter. The workers' interface is shown in Figure 6.3.

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

To utilize human computation to power a 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 natural 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 [153], 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

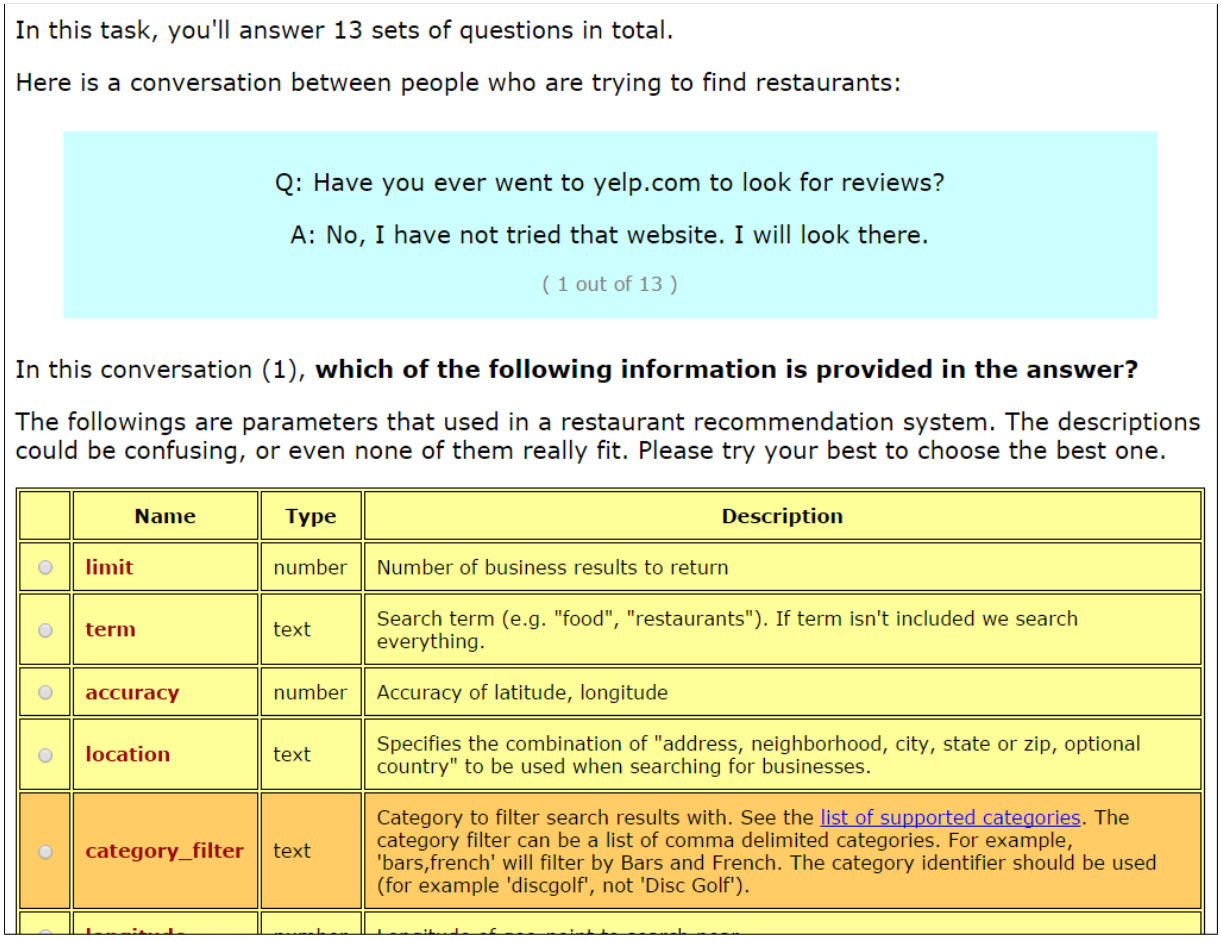


Figure 6.3: The interface for crowd workers to match of parameters to natural language questions.

value. To prevent the system from idling in the case that no answers match one another, a hard time constraint is also set. The system selects the first answer from workers when the the time constraint is reached.

Crowd-powered Dialog Management Second, we use the idea of dialog management to control the dialog status. Dialog management simulates a dialog as a process of collecting a set of information – namely, parameters in the context of web APIs. Based on which parameters are given, the current dialog state can be further decided (Figure 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.

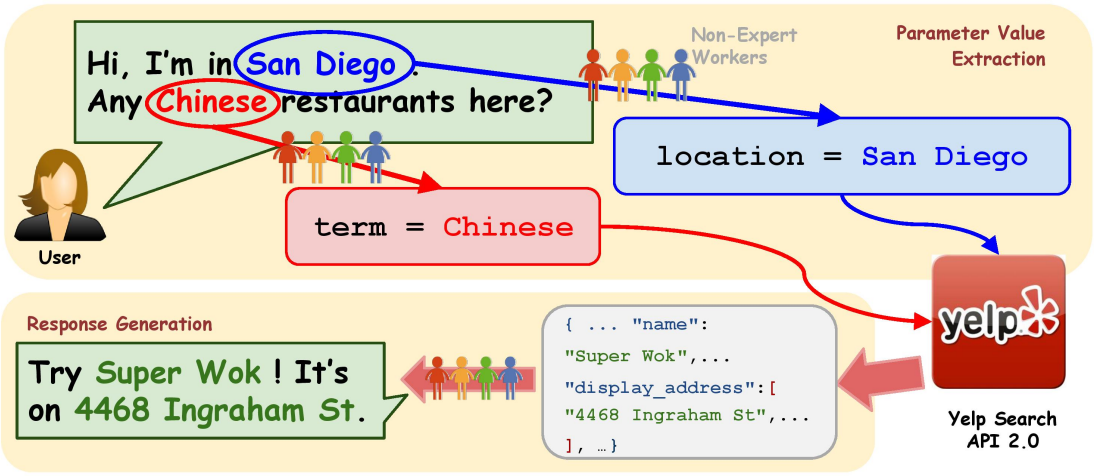


Figure 6.4: On-line Phase: crowd workers extract the required parameters and turn resulting JSON into responses.

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 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 [10]. 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/

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).

6.2 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

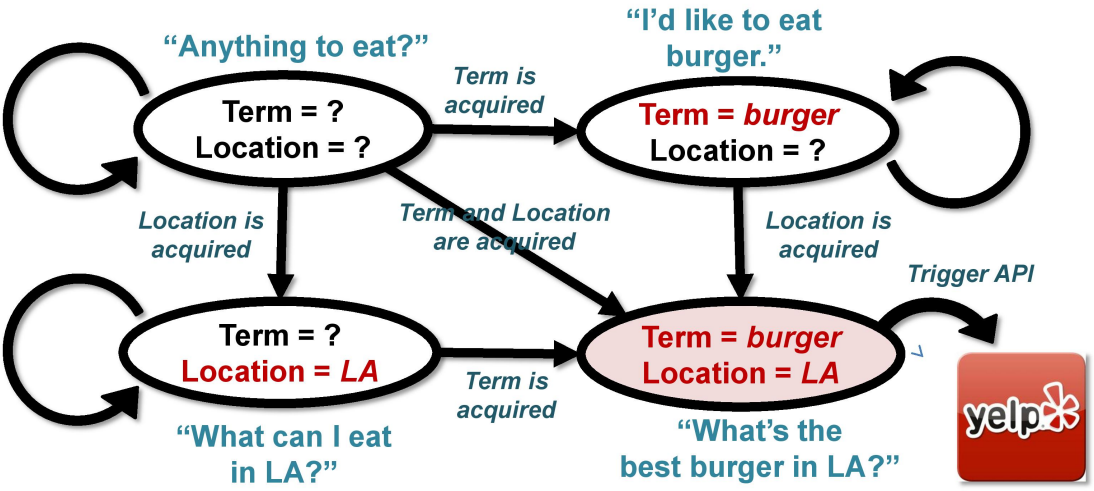


Figure 6.5: The State Diagram of dialog Management. In the context of crowd-powered systems, introducing a dialog manager reduces the time it takes the crowd to generate a response because most actions can be pre-defined and generated according to the dialog state.

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, 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

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.

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.

6.3 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.

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

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.

6.3.1 Implementation

Guardian was implemented as a 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.

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

Web API	Parameter Used		Time (sec) [Avg (Stdev)]		Avg. #Turn per Conv.	Task Completion Rate (TCR)		
	Name	Desc.	Fill Each Parameter	Obtain API's Result		API Only	API +Crowd Recover	Other System
Yelp Search	`term`	query term (words)	48.35 (21.69)	61.70 (27.41)	2.80	9/10	10/10	0.96 [144]
	`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 [144]
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 [99]

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.

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.

6.3.2 Experimental Result

To test Guardian, we follow an evaluation method similar to the one used to evaluate Chorus [90]: 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 dialog system of the same domain was also listed for comparison. Note that the TCR and dialog system values were not directly compatible.

6.3.3 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

user the zip code here is

crowd **hello user, Pittsburg PA ! Let me look.**

user sorry 15232

[*Parameter Status: location = (no matched answer found)*]

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.

crowd Look outside and tell me the weather please.

crowd http://www.weather.com/weather/ hourbyhour/l/Pittsburgh+PA+15232:4:US

6.3.4 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.

6.4 Discussion

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

6.4.1 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 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?

6.4.2 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" [62] 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 [62] or automatic induced slots [30].

6.5 Summary

We have introduced a crowd-powered web-API-based dialog system 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 dialog system with the web APIs (offline phase), and a user with the dialog system (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 this dissertation work, Evorus uses a set of external dialog systems to automate Chorus. Guardian enables generating a large set of external task-oriented dialog systems efficiently using Web APIs. Equipped with Guardian, Chorus can automate more and more conversations with users, as the need for them arises, making automation a gradual process that occurs based on user interests.

Chapter 7

Dialog ESP Game: Real-Time On-Demand Crowd-Powered Entity Extraction

Guardian uses crowd workers to extract parameter values from a running dialog (Chapter 6) using an output-agreement mechanism similar to the ESP Game for image labeling [153]. However, the literature has little to say about speed-quality trade-offs when the time budget is only few seconds, which is a common response time required for conversational assistants. If workers have as long as they want to annotate a sentence, most AI systems would assume the annotation is trustworthy and use it as the gold-standard label; but it was not clear that this assumption would hold when workers have only 20 seconds. To bridge this gap, explore quality-speed trade-offs of using on-demand real-time crowdsourcing to extract entities from a running dialog.

As mentioned in Related Work chapter (Chapter 2.1), modern dialog system frameworks such as Olympus [18] rely heavily on entity extraction, known as the core task of *slot filling* to understand user utterances. The goal of slot filling is to identify from a running dialog different *slots*, which correspond to different parameters of the user's query. 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) [123, 171] and Recurrent Neural Networks [110], 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 [177] or state tracking [167]. 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 brittle in extracting unseen entities. [171] 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 [160]; Unsupervised slot-filling approaches have also been developed in recent years [31, 61]. However, these methods are still underdeveloped.

To address these challenges, we propose to use real-time crowdsourcing as an entity extrac-

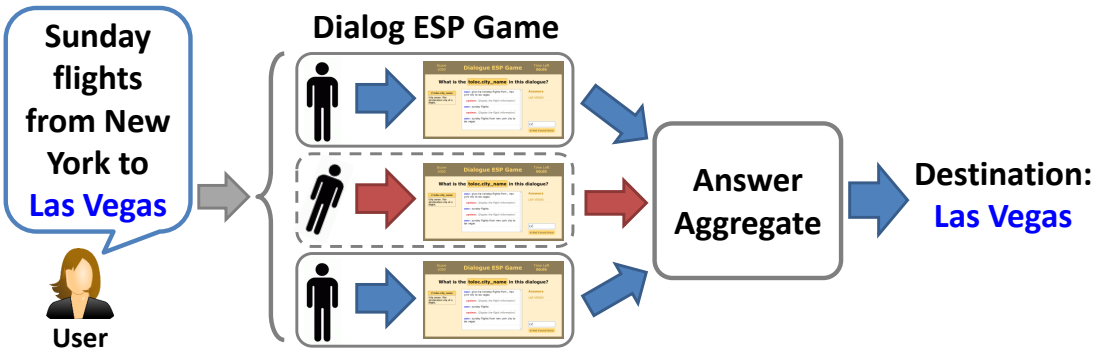


Figure 7.1: The crowd-powered entity extraction with a multi-player Dialog ESP Game. By aggregating input answers from all players, our approach is able to provide good quality results in seconds.

tor in dialog systems. To the best of our knowledge, few previous works have attempted to use crowdsourcing to extract entities from a running conversation. [162], for example, studied various methods to acquire natural language sentences for a given semantic form by the crowd. [89] utilized crowdsourcing to collect dialog data, and illustrated CrowdParse, a system that uses the crowd to parse dialogs into semantic frames. However, none of these works conducted formal studies on crowd-powered entity extraction in real-time.

Inspired by the ESP game for image labeling [153], 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. 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 this project 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.
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.

7.1 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 [153]. The original game randomly pairs two players and

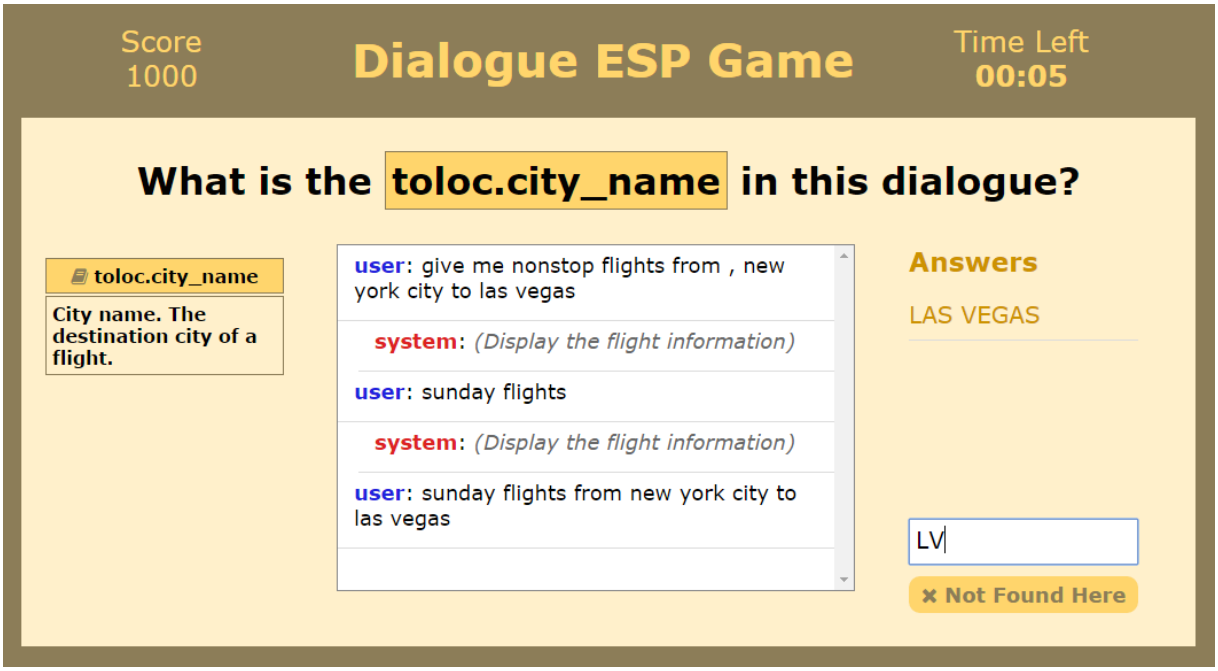


Figure 7.2: The Dialog ESP Game interface is designed to encourage quick and correct entity identification by crowd workers. Workers are shown the complete dialog and a description of the entity they should identify.

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, as introduced in the Related Work chapter (Section 2.3,) many approaches have been used in real-time crowd-powered systems. 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.

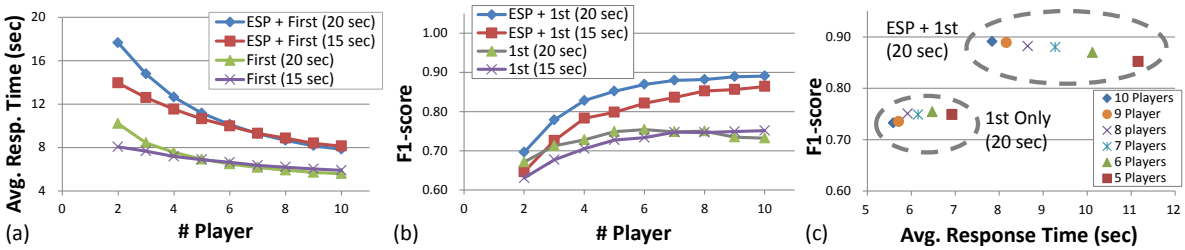


Figure 7.3: Trade-off curves between accuracy, average response time and number of players.

7.2 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.

7.2.1 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) [64]. 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.

7.2.2 Data Pre-processing & Experiment Setting

For Class A, we obtain the preprocessed data used in many slot filling works [59, 110, 123, 147, 171], 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 [36], 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`** 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.

7.2.3 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.

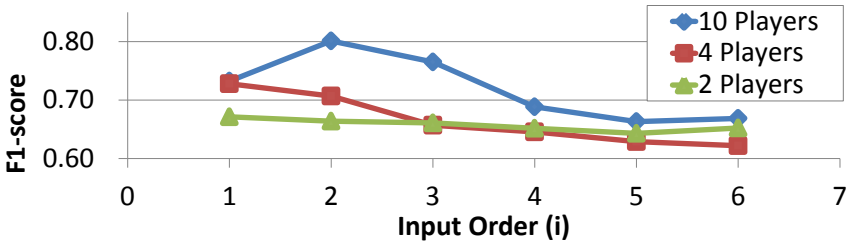


Figure 7.4: F1-score of the "ith Only" setting. Earlier input answers are generally of better quality (unless #players $\geqslant$ 10, where almost all ESP games have at least one matched answer and the ith answer might not be solely used.)

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.

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

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.

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.

7.2.4 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.

Experimental results are shown in Table 7.2. An automated CRF model is implemented as

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.

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.

a baseline.[1] 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 [77] and word frequency influence speed of text comprehension [60, 124]. 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 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);

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

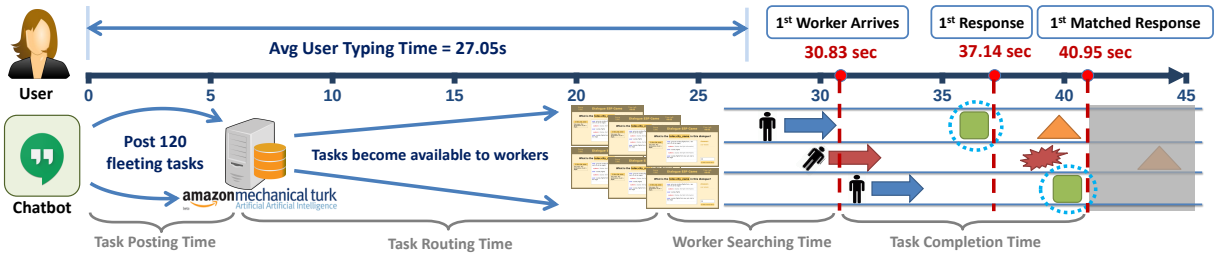


Figure 7.5: Timeline of the Real-time Crowd-powered Entity Extraction System. On average, the first worker takes 30.83 seconds to reach, the first answer is received at 37.14 seconds, and the first matched answer occurs at 40.95 seconds. A user on average spends 27.05 seconds to type a chat line, i.e.,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

"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.

7.3 Experiments 2: User Study 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 [79].

7.3.1 System Implementation

We implemented a Google Hangouts chatbot by using the Hangupsbot[2] 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.

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

To recruit crowd workers, we introduce *fleeting task*, a recruiting practice inspired by *quikturkit* [14]. This approach achieves low latency by posting hundreds of short lifetime tasks, 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 [10, 14, 90]. In a system deployed at scale, a retainer or push model is likely to work as well.

7.3.2 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. 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.

7.3.3 Experimental Results

Data Collected The followings are the lists of 9 food and 3 drinks created by 10 participants in our user study.

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

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

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%.[3] A trade-

[3]We only consider the answers submitted within 60 seconds.

	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[4]	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.

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.

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[5], 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 enti-

[4]Including the results from Food, Cook, and Chat scenarios.

[5] 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.)

ties 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 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 [79]. [8] 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.

7.4 Discussion

Incorporating domain-specific knowledge is a major obstacle in generalization of crowdsourcing technologies [69]. 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. For example, "Washington" and "Washington DC" can be mapped together if a list of city names is available. 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.

7.5 Summary

In this chapter, we have explored using real-time crowdsourcing to extract entities for dialog systems, which is the key component of the Guardian framework (Chpater 6.) 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. Equipped with this result, we are confident that real-time crowdsourcing is robust and fast enough to be used on real-world deployed systems such as Chorus and Guardian.

Part IV

Expanding the Capabilities of Chorus

Chapter 8

InstructableCrowd: Creating IF-THEN Rules for Smartphones via Conversations with the Crowd

Intelligent personal computing devices—such as smartphones, smartwatches, digital assistants (*e.g.,* Amazon's Echo) and wearables (*e.g.,* Google Glass), have become ubiquitous in society due to the power and convenience they offer. These devices are useful as shipped, but getting the most out of them requires tailoring them to their owner's preferences and needs. For example, after buying a smartphone, the user will usually first spend time customizing it by changing the wallpaper or adjusting the home screen layout. The same behavior is seen with nearly all other electronic devices, including personal assistants, tablets, laptops, and digital cameras. A great deal of customization takes place when the device is new, but the tuning process also usually continues at a slower pace over time as users adjust their devices in response to changing needs, the availability of new software or functionality, or shifts in personal circumstances. For example, a new security threat may lead to installing better firewall software; a near-miss with severe weather may prompt the user to change local weather alert preferences; and moving to a new city may lead to changing the parameters on travel or map software to reflect the user's new location. Users manually adjust the *long-term behavior* of their devices in order to better fit their own behavior.

As important as customization is, however, it is often held back by a variety of difficulties for users. One is that devices are becoming ever more complicated: new features and capabilities provide power and flexibility, but at the cost of complexity. Customizing a device often requires the user wading through complex, multi-layer menus, searching for the right app, or experimenting with poorly explained settings. All of this can be confusing and intimidating. Furthermore, getting the most from a device usually requires programming it to react intelligently to events and automate responses, and many users find programming to be difficult and even frightening. The complexity of devices also means that even a small adjustment to a system's long-term behavior through programming could result in unintended consequences to the user's experience (when compared with simple, one-time interactions such as setting an alarm). Thus, the more complex the interaction with the device, the more important are high accuracy and robustness in understanding the user's needs.

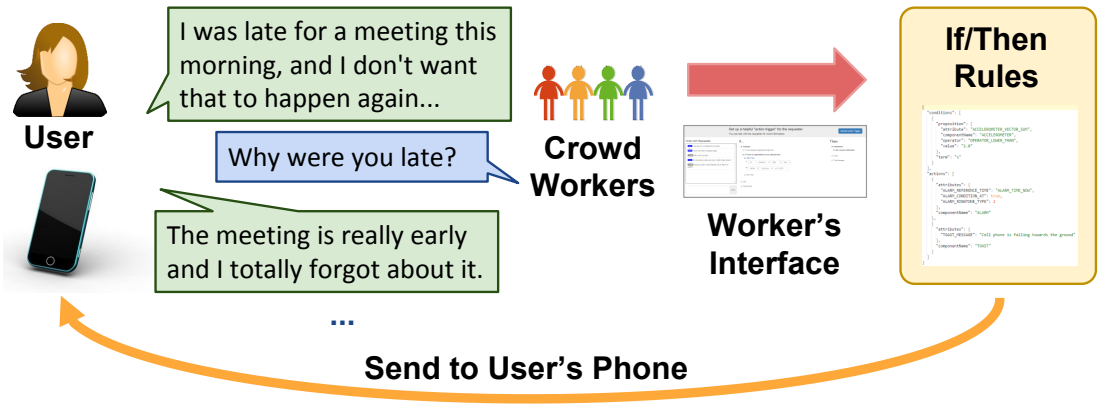


Figure 8.1: Users have a conversation with InstructableCrowd to create If/Then rules that then run on their phone to solve problems. The backend system is run by synchronous crowd workers who respond to the user, ask follow-up questions, and create rules. Users can then review the rules on their phone to make sure they were what they wanted.

One technology that has significant potential for addressing these problems is *natural language interface*. Users could much more easily customize and even automate their devices if they could simply speak to them rather than wading through instruction manuals, menu trees, and tutorials. And in fact, natural language interfaces have become a common part of modern digital life already. Chatbots utilize text-based conversations to communicate with users; personal assistants on smartphones such as Google Assistant take direct speech commands from their users; and speech-controlled devices such as Amazon Echo use voice as their only input mode.

In this exploratory project, we introduce *InstructableCrowd*, a crowd-powered system that allows users to *program* their devices and thus change their longer-term behavior via a natural language interface. InstructableCrowd is based around two key design decisions that address the main problems with device customization and automation which are outlined above. First, we have focused on creating relatively simple programs that are easy to use. Second, we make use of crowd workers to operate the natural language interface instead of using automated systems, since humans are much better at understanding and interpreting complex user requirements than current electronic systems.

Our programming system is oriented around relatively simple *IF-THEN rules*, also known as *trigger-action* rules. Modern smart devices, especially smartphones, contain a wealth of *sensors* and *effectors* that can be combined to perform useful customized tasks for their users. For example, they could be used to go beyond simple, static programming (such as setting a wake-up alarm to go off at a specific time every weekday) to customizations that are based on inputs and status information (like adjusting a wake-up alarm based on traffic conditions).

A prominent example of this type of rule-based system is the mobile application IFTTT (If This Then That, ifttt.com). The service enables users to author simple trigger-action rules that contain only *one* trigger (*e.g.*, a post on Twitter) and *one* action (*e.g.*, synchronizing the latest Twitter post to Facebook). The service is obviously useful – it has millions of users [76] – and its simplicity makes it easy to use. However, that same simplicity also means that the system fails

to cover many real-world scenarios [39, 66, 149]. Research has shown that 22% of behaviors that people came up with require more than one sensor or effector [149]. The complexity of rules people would like to create is likely to only increase as services like IFTTT continue to be integrated with other services and more devices. Therefore, in this project, we focus on an extended version of IFTTT-style rules, in which the IF and THEN can each contain more than one sensor/effector.

With the awareness of the limitation of automated dialog systems, we developed a *crowd-powered* conversational agent. *InstructableCrowd* allows end users to create rich, multi-part IF-THEN rules via conversation with the crowd (Figure 8.1). A group of crowd workers is recruited on demand to talk with a user and create rules based on the conversation. With intelligent workers on a rich desktop interface supporting users, the interface can be simplified into a familiar speech or text chat app, 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. Users may know their problems but not know what solutions would best resolve them. The crowd can help users identify possible solutions that the user did not even know existed and then create the rules needed to implement them. InstructableCrowd then lets users edit and improve the created rules. 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 with the wealth of sensors and effectors on their smartphones and other devices, which may have broader implications for the future of programming with speech.

8.1 Related Work

In addition to crowd-powered systems and real-time crowdsourcing, which has been described in the Related Work chapter (Section 2.3,) InstructableCrowd is also related to prior work on *(i)* end-user programming and *(ii)* automatic IF-THEN rules generation.

8.1.1 End-User Programming

InstructableCrowd builds upon the long history of research and products in end-user programming [104], which aims at enabling non-programmers to author or compose their own applications. Early works in this field started from database [58] and email management [109], and later gradually became more common as more and more senors and effectors became available for general users [19, 21, 22, 37]. For instance, CoScripter allowed end-users to program scripts by demonstration [13, 100]. CoScripter used its corpus of scripts to allow easier creation of new actions from mobile devices [97]; Sikuli is another famous end-user programming project [174], which 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.*, click the button.)

Trigger-action programming is one simple model of end-user programming where the user forms a new functionality by combining pre-defined triggers (sensors of "IF") with pre-defined actions (effectors of "THEN"). Many solutions were proposed to realize the trigger-action programming, such as using existing notations of business processes modeling (BPM) to represent

rules [20], adopt an effective workflow to create rules [80, 83], or create solutions for domain-specific applications [39]. The IFTTT project has had a 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 domains, such as smart home applications [41, 149], cross-device interactions [50], and the Internet of Things [146].

IFTTT only allows rules to be composed of a single trigger and action. Several frameworks were proposed to support multiple triggers (IFs) and actions (THENs). Dey et al. created an interface where users can drag and drop multiple sensors and effectors on a sheet to create new rules [42]. Huang et al. [66] and Ur et al. [149] both extended IFTTT's interface to allow users select more than one triggers or actions. Ghiani et al. used interactive composition and natural-language feedback to assist non-programmers to compose arbitrary trigger-action rules, which could be more complex than IFTTT rules [51]. However, most of these works focused on the challenges in designing interfaces or workflows 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 [55]. 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, 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. [39] 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. [66] 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 that uses crowdsourcing to build software [96].

8.1.2 Automatic IF-THEN Rules Generation

Automatically translating a natural-language utterance into the form that computers can execute is a well-known task in natural language processing, which is referred to as *language understanding* or *semantic parsing*. For instance, Artzi *et al.* used a grounded CCG (Combinatory Categorial Grammar) semantic parsing approach to map instructions such as "at the corner, turn left to face the blue hall" to actions that the agent (virtual robot) can execute [3]; and NaturalJava aimed to use a natural-language interface for creating, modifying, and examining Java programs [118].

Particularly for IFTTT rules, Quirk *et al.* collected 114,408 IF-THEN rules and their natural-language descriptions from the IFTTT website and demonstrated the possibility of producing IF-THEN rules based on corresponding descriptive text [120]. Several follow-up works that proposed different approaches such as attention-enhanced encoder-decoder model [44], using latent attention [105], or syntactic neural model [176] further improved the accuracy of IFTTT rule generation. Under the context of conversational assistance, Chaurasia *et al.* created an

automated dialog system that generates IFTTT rules by having a conversation with users [27]. With a "Free User-Initiative" setting (which the authors referred to as "a more realistic setting"), Chaurasia's system achieved an accuracy of 81.45% in generating IFTTT rules. However, this performance is still not sufficient for practical use, and none of prior works attempted to produce multi-part rules that are more complex than that of IFTTT.

8.2 InstructableCrowd

InstructableCrowd is implemented as an Android mobile application (Figure 8.3) for supporting 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 instance, 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. Furthermore, InstructableCrowd is also able to merge multiple rules sent by different crowd workers to form a more reliable final rule. We describe the system architecture and implementation details in this section.

8.2.1 Rules, Sensors, and Effectors

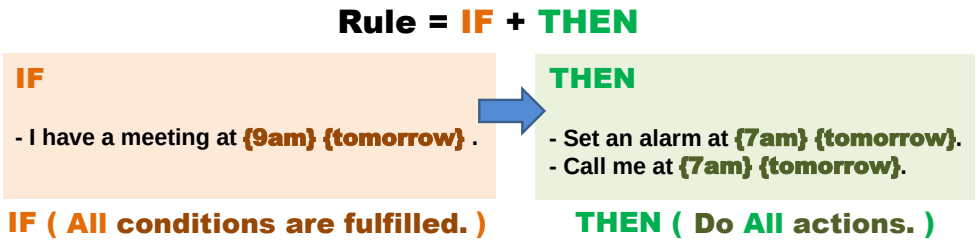


Figure 8.2: Example of a rule in InstructableCrowd. A rule is defined as a tuple that contains an IF part and a THEN part. The IF part contains a set of sensors that describe aspects of the user's life and context, and the THEN part contains a set of effectors that can be performed.

In this work, a **rule** is defined as a tuple that contains an **IF** part and a **THEN** part. The IF part contains a set of **sensors** (also referred to as **IFs**) that describe aspects of the user's life and context. For instance, the "Calendar" application 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 can be sensors in the IF part. The THEN part contains a set of **effectors** (also referred to as **THENs**) that can be performed, such as push a notification, set an alarm, and send a text message, etc. It is noteworthy that InstructableCrowd allows more than one sensors/effectors in each part, while IFTTT only allows one. An overview of an example rule is shown in Figure 8.2.

Each sensor has one or more triggers that can be selected. For instance, the "calendar" sensor could have three different triggers that reflect the status of 1) currently ongoing events, 2) future

Sensor	Trigger	Trigger Description	Attributes (Input Type)
Bus	Current location	The bus is currently at a certain stop:	Bus Number (Text) Bus Stop (Text)
	Future location	The bus will arrive at a certain stop in minutes:	Bus Number (Text) Will Arrive at Stop (Text) In How Many Minutes (Text)
Calendar	Current event	If I am having an event right now that:	Event Type (Select)
	Future event (absolute time)	If I will have an event that (absolute time):	Day (Select) Start Time (Time) End Time (Time) Event Type (Select)
	Future event (relative time)	If I will have an event that (relative time):	In How Many Minutes (Text) Event Type (Select)
Call	Receive a call	If I receive a phone call that:	From (Text)
Clock	Current time	The current time is:	At/Before/After (Select) Time (Time)
Email	Receive an email	If I receive an email that:	Sent By (Text)
GPS	Current location	I am currently located at:	Location Name (Text)
	Distance to a location	If my distance to a certain location that:	To (Text) Is Greater/Less Than/Equals To (Select) Distance (Text)
Message	Receive a message	If I receive a text message that:	Sent By (Text) Contains the word(s) (Text)
News	Receive a news	If I receive a breaking news that:	Title contains the word(s) (Text)
Phone Body	Phone falls	If my phone is falling.	N/A
	Drive	If I am driving.	N/A
Weather	Weather forecast	If the weather forecast that:	Day (Select) Forecast (Select)

Table 8.1: Sensors (IFs) with their Triggers and Attributes as implemented in InstructableCrowd.

Effector	Action	Action Description	Attributes (Input Type)
Alarm	Set an alarm	Set an Alarm that:	Day (Select) Time (Time)
Calendar	Add an event	Add an Event on my Calendar that:	Day (Select) Start Time (Time) End Time (Time) Event Type (Text) Event Title (Text)
Call	Dial a call	Call:	To (Text) What to Say (Text)
Email	Send an email	Send Email(s) that:	To (Text) Email Title (Text) Email Content (Text)
Message	Send a message	Send Message(s) that:	To (Text) Message Content (Text)
Notification	Send a notification	Push me a Notification that:	Notification Content (Text)

Table 8.2: Effectors (THENs) with their Actions and Attributes implemented in InstructableCrowd.

events at an absolute time (*e.g.*, 9am today), or 3) future events at a relative time (*e.g.*, in 30 minutes.) Similarly, one effector can also have one or more actions to perform. Each trigger and action is composed of a set of **Attributes** to specify the details of the condition. For instance, for configuring "Calendar" sensor to tell if the user has any events in 30 minutes with the "Future Event (Relative Time)" Trigger, the "In How Many Minutes" attribute needs to be filled with "30," and the "Event Type" attribute needs to be filled with "Any." In this project, we focused on observing end-user and workers behavior in **selecting Sensors/Effectors** and **filling Attributes**.

The full list of sensors and effectors with their triggers/actions and attributes used in our study are listed in Table 8.1 and Table 8.2.

8.2.2 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 8.3), 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 adopts the LIA framework [6], which uses a CCG parser to parse the input text into a logical form and execute the corresponding commands, to recognize user's voice input. Once the user give verbal commands such as *"create a rule,"* LIA connects to InstructableCrowd and initiates 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 workers converse by text, and the user may either use text or voice).

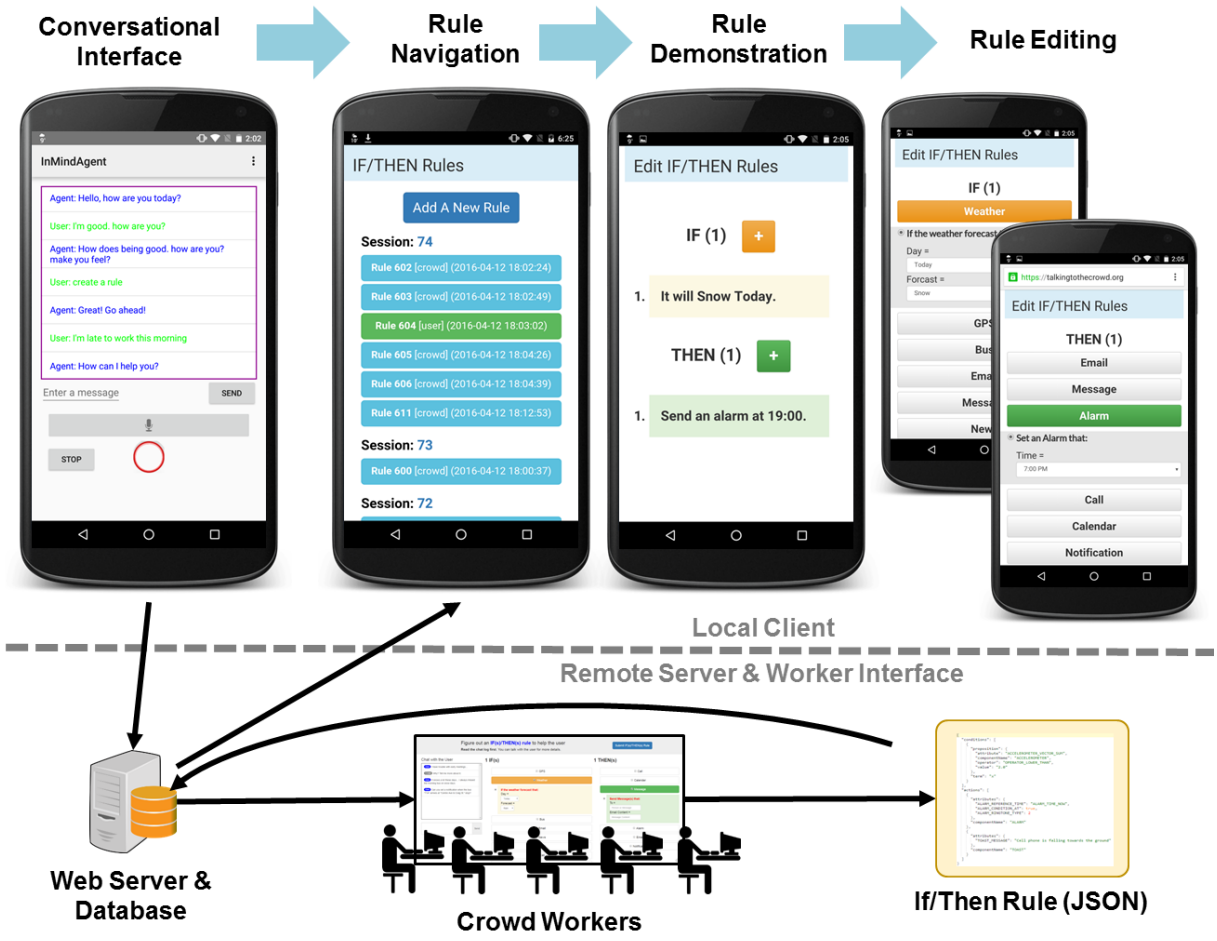


Figure 8.3: InstructableCrowd users have a conversation with crowd workers about a problem they are having. Crowd workers collectively create IF-THEN rules that may help the end user solve their problem using sensors and effectors available on the smartphone platform. The rules are then sent back to the user's phone for review, editing, and approval. The rules then run on the smartphone.

Once the rule is created, it is sent back to the user's phone, where a Decision Rule Engine component will store, validate, and process that rule. Currently, the system is implemented and tested on the Android OS 6.0.1, and the server is implemented in Java.

8.2.3 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 8.3, 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 trigger. For

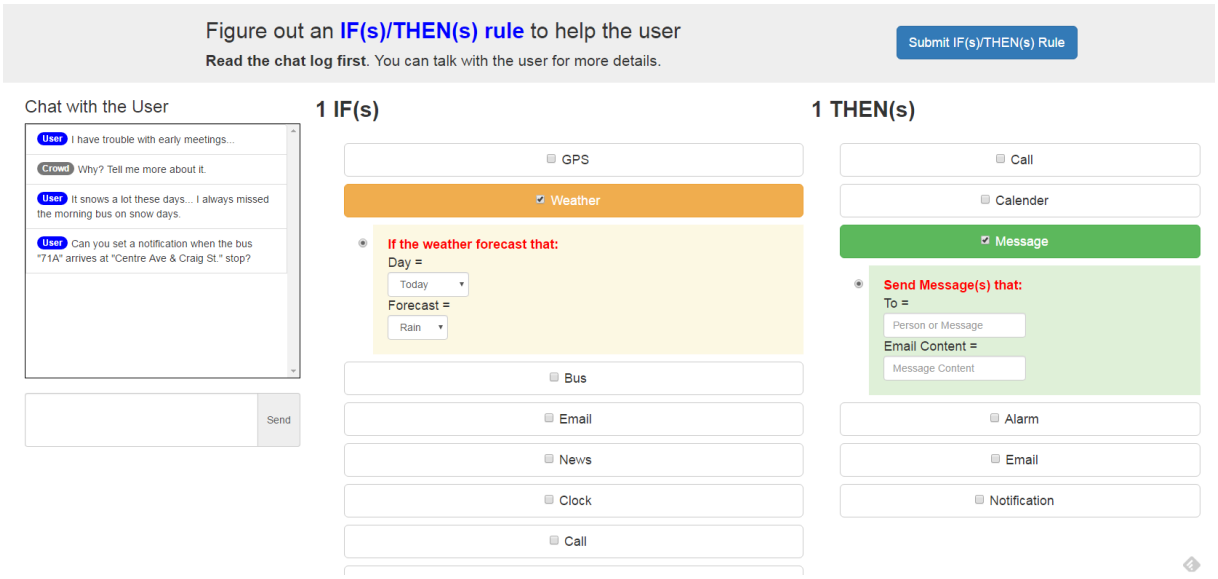


Figure 8.4: Worker interface. A chat interface (left) allows workers to talk to the end user to discuss the problem. The IF section (middle) allows the worker to specify sensors, along with triggers (in red text) and their attributes; the THEN (right) section allows them to specify effectors, along with actions (in red text) and their attributes.

instance, the description template of "Weather" sensor's forecast trigger is "It will weather day." If "Weather" sensor's this trigger 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 editing interface is inspired by the IFTTT mobile APP. However, it enables the user to combine multiple IFs and THENs while IFTTT focuses on one-to-one APP compositions.

8.2.4 Worker Interface

The worker interface allows crowd workers to select sensors (IFs) and effectors (THENs) easily. The interface contains three main parts (Figure 8.4). 1) The web-based chat interface allows workers to discuss the problem with the end-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. Workers first select appropriate sensors (*e.g.*, Calendar) in the IF conditions, and then select triggers under the sensors (*e.g.*, Future Event (Relative Time)), and finally, they fill in the appropriate attribute values (*e.g.*, In How Many Minutes = 30.) 3) The THEN section allows workers to select effectors and corresponding actions and fill in attribute values. By selecting IFs and THENs, the worker is able to create rules that trigger certain actions based on specific conditions.

8.2.5 Merge Multiple Crowd-Created Rules by Voting

InstructableCrowd recruits multiple workers for each conversation; therefore, multiple rules are received, respectively, from each conversation. End users are free to pick any rules submitted by the crowd or wait the rules are merged together automatically into a *final rule*. Our automated rule-merging process uses output agreement to identify the best components to use. Output-agreement mechanisms such as ESP Game for collecting image labels [153] have been widely used in human computation to obtain reliable human-generated labels from multiple workers [154]. First, any sensors and effectors that are selected by more than two workers (our current threshold) are included in the final rule. Second, for each Sensor/effector picked in the first step, its trigger/action that is selected by most workers will be chosen. Finally, for each selected trigger/action, InstructableCrowd fills each attribute with the value that was proposed by the most workers. If two values were proposed by an identical number of workers, InstructableCrowd selects the value which was proposed earliest.

8.2.6 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 Weather sensor's JSON file representing that "it will snow tomorrow" (Trigger = Weather forecast).

```
{
  "name": "if-weather",
  "condition": "if-weather-forecast",
  "attributes": [
    {
      "name": "if-weather-forecast-day",
      "value": "Tomorrow",
      "type": "select"
    },
    {
      "name": "if-weather-forecast-condition",
      "value": "Snow",
      "type": "select"
    }
  ]
}
```

The following is the JSON representation of the Alarm effector for "set the alarm at 7am tomorrow" (Action = Set an alarm.)

```
{
```

```
  "name": "then-alarm",
  "condition": "then-alarm-send",
  "attributes": [
    {
      "name": "then-alarm-send-day",
      "value": "tomorrow",
      "type": "text"
    },
    {
      "name": "then-alarm-send-time",
      "value": "07:00",
      "type": "text"
    }
  ]
}
```

The following is the JSON representation for an IF-THEN rule, "IF it will snow, and I have a meeting at 9am tomorrow, THEN set alarm at 7am," which includes two sensors (Weather and Calendar) and one effector (Alarm.)

```
{
  "if": [
    {
      "name": "if-weather",
      "condition": "if-weather-forecast",
      "attributes": [
        {
          "name": "if-weather-forecast-day",
          "value": "Tomorrow",
          "type": "select"
        },
        {
          "name": "if-weather-forecast-condition",
          "value": "Snow",
          "type": "select"
        }
      ]
    },
    {
      "name": "if-calendar",
      "condition": "if-calendar-future",
      "attributes": [
        {
          "name": "if-calendar-future-day",
```

```
            "value": "Tomorrow",
            "type": "select"
          },
          {
            "name": "if-calendar-future-type",
            "value": "Meeting",
            "type": "select"
          },
          {
            "name": "if-calendar-future-start",
            "value": "09:00",
            "type": "time"
          }
        ]
      }
    ],
    "then": [
      {
        "name": "then-alarm",
        "condition": "then-alarm-send",
        "attributes": [
          {
            "name": "then-alarm-send-day",
            "value": "tomorrow",
            "type": "text"
          },
          {
            "name": "then-alarm-send-time",
            "value": "07:00",
            "type": "text"
          }
        ]
      }
    ]
}
```

New sensors and effectors can be added easily once they are implemented in our middleware, by simply adding new JSON entries for them. Currently, we implemented 10 sensors and six effectors in InstructableCrowd (Table 8.1 and Table 8.2.) As we go forward, we plan to continue to expand the set of available sensors/effectors.

8.2.7 Decision Rule Engine

The Decision Rule Engine is in charge of validating, storing, processing, and executing rules created by either a crowd-worker or the user. Decision Rule Engine is composed of multiple modules that interact with each other in order to execute an action given a set of specific conditions that are true. These modules are interconnected as shown in Figure 8.5. The following outlines the work flow (in steps), message passing, and how Decision Rule Engine components cooperate over time to manage rules created by user or crowd-workers.

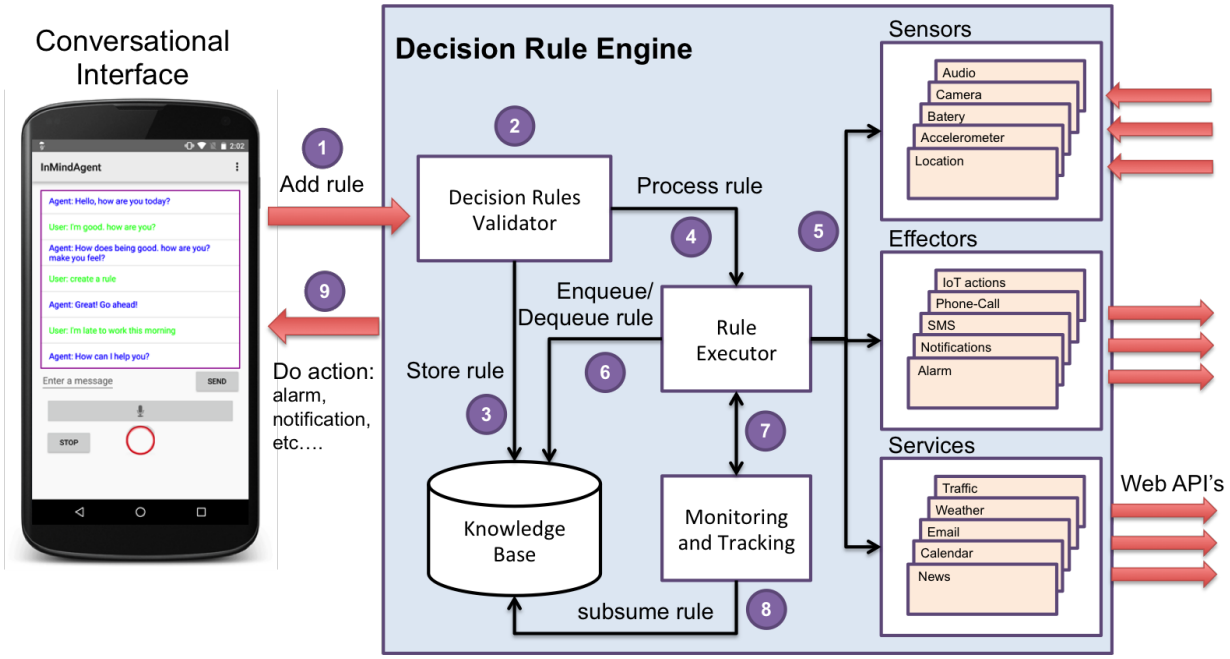


Figure 8.5: The architecture of InstructableCrowd's Decision Rule Engine. The step (1) to step (9) outlines the work flow, message passing, and how Decision Rule Engine components cooperate over time to manage rules created by user and crowd-workers.

- **Decision Rule Validator:** After the user or crowd worker has defined a new rule to be added (Step 1 in Figure 8.5), this component validates the syntax of that rule according to the sensors' and the effectors' attributes and constraints (Step 2). For instance, if the rule has a condition that refers to attribute `<CALENDAR_START_TIME>`, the validator will parse this condition and check whether there actually exists a sensor called "Calendar" that has an attribute called *startTime*, which must be of type *Date* and whose value must be a date/time that occurs later than current date/time.

- **Knowledge Base:** Once the rule is parsed and validated, it is stored in a knowledge base where it can be accessed anytime by any component (Step 3). These rules are stored locally for performance and privacy reasons, so potentially sensitive information contained within the rule is protected.

- **Rule Executor:** After validation, the rule is immediately processed in order to determine

whether it should be executed in that moment (Step 4). If so, it invokes actions from the appropriate effectors (Step 5). If not, it adds the rule to a queue so it can be executed later when all its conditions are true. Enqueued rules are validated periodically for execution, and this validation may occur at different time periods (Step 6).

- **Monitoring & Tracking:** This module is responsible for monitoring the rule execution process (Step 7) by checking if there are rules that are either never triggered or conflicting with each other (*e.g.*, one rule intends to turn the GPS on while the other one intents to turn it off). When conflicts occur, the Monitoring/Tracking module temporarily subsumes the less relevant rule (*i.e.*, the one that has been activated less frequently) and then user is asked to confirm this subsumption decision (Step 8).

- **Built-in & External Sensors/Effectors:** In addition to built-in sensors and effectors that are part of the operating system, such as GPS and SMS Messages, some virtual sensors/-effectors are based on external services, such as the Weather forecast and News feeds. In our implementation, we use a RESTFUL API to upload, extract, and collect information from web servers. Finally, user is always aware of action execution through notifications, text messages, alarms, etc. (Step 9).

8.3 User Study

For evaluating the performance of InstructableCrowd, we conducted a set of in-lab user study. Our goal was to understand if creating IF-THEN rules using conversation would sacrifice rule quality, compared with using a graphic user interface (GUI.) Furthermore, 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. Participants created rules using a mobile application in a control condition to allow us to compare with how users currently create rules using applications such as IFTTT.

8.3.1 Scenario Design

We designed the following six scenarios (S1 to S6) inspired by Huang *et. al* [66], along with a gold-standard set of sensors and effectors for each that we consider to be ground truth for assessing the performance.[1] 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 (one sensor and one effector), S3, S4, and S5 are **intermediate** scenarios (two sensors and one effector), and S6 is **hard** scenario (two sensors and two 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. (Easy scenario.)

[1] The attributes that were not specified in a gold-standard rule indicate that the user or worker should leave these attributes blank. In the evaluation, the textual attributes such as message content or email content will be examined manually. It is also noteworthy that in this section, we only listed one common gold-standard rule, while more than one rule (*e.g.,* adding or alternating notifications) could be considered valid for a scenario. We describe the details of evaluation in Section 8.4.

- ***IF**: News (Receive a news: Title contains the word(s) = "Steelers")*
- ***THEN**: Notification (Send a notification: Notification Content = "News of Steelers!")*

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. (Easy scenario.)
   - ***IF**: Message (Receive a message: Sent By = Mom, Contains the word(s) = "grandfather")*
   - ***THEN**: Notification (Send a notification: Notification Content = "Mom just texted you about grandfather!")*

3. **[S3] Snow & Meeting:** It snowed last night. I was late for work this morning and missed an important meeting at 9 am because I had to take care of all the snow. My boss was quite upset and warned me this can not happen again. (Intermediate scenario.)
   - ***IF**: Weather (Weather forecast: Day = today, Forecast = snow) + Calendar (Future event [absolute time]: Day = tomorrow, Event Type = meeting, Start Time = 09:00)*
   - ***THEN**: Alarm (Set an alarm: Day = tomorrow, Time = 07:00)*

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 reply to him/her with a message saying "Sorry I'm driving." (Intermediate scenario.)
   - ***IF**: Phone Body (Driving) + Call (Receive a call: From = Anyone)*
   - ***THEN**: Message (Send a message: To = People mentioned in "IF(s)", Message Content = "Sorry, I am driving.")*

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. (Intermediate scenario.)
   - ***IF**: Bus (Current location: Bus Number = 53, Bus Stop = "Washington St") + Clock (Current time: At/After/Before = After, Time = 17:00)*
   - ***THEN**: Notification (Send a notification: Notification Content = "Bus 53 will be arriving at Washington St. stop soon!")*

6. **[S6] Late for Dinner:** My wife, Amy, does not like me to be late at home when we have a big dinner scheduled. 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

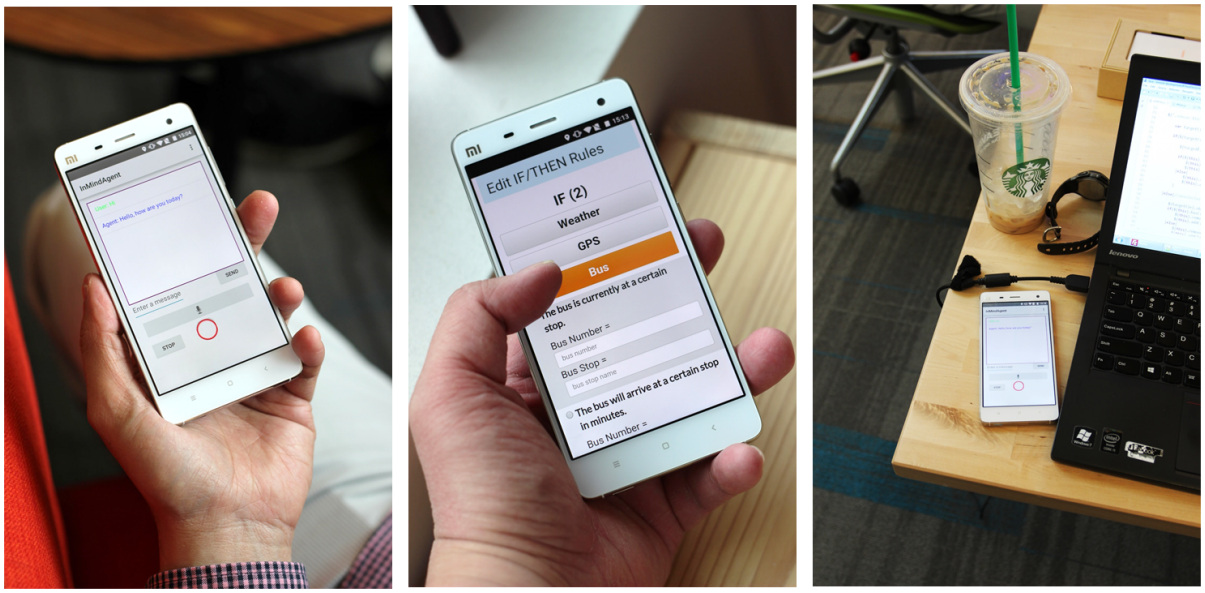


Figure 8.6: User study setting. While waiting for responses from the crowd, participants used their own laptops or mobile devices to simulate the likely context of use in the real world.

be home late." Also, give a phone call to "Ben's Flower Shop" and tell them to "Prepare a small surprise bouquet." (Hard scenario.)

- ***IF**: GPS (Distance to a location: Is Greater/Less Than/Equals To = Is Greater Than, To = Home, Distance = 30) + Calendar (Future event [relative time]: Event Type = Dinning, In How Many Minutes = 30)*
- ***THEN**: Message (Send a message: To = Amy, Message Content = "I might be home late.") + Call (Dial a call: To = Ben Flower Shop, What to Say = "Prepare a small surprise bouquet for me.")*

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

8.3.2 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, depending on the participant's preference) 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 8.6) to manually create a rule.

In condition (1), three 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 10 crowd-created rules (**Crowd Voting**) using the process described in Section 8.2.5 (the threshold for including a sensor/effector was two.) We refer to condition (2) as **User Only**.

For recruiting participants, we posted the information on social media sites such as Facebook and Twitter. We also posted flyers on the campus of Carnegie Mellon University (Pittsburgh campus) and University of Pittsburgh. 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 who had very limited experience in programming, or none at all. People who volunteered to participate in our study were directed to a web form for signing up, in which we asked people to self-report their programming skill level ("How good are you at programming?"), from 1 ("I don't know anything about programming.") to 7 ("I'm an expert programmer."). We selected the first 14 participants who signed up with a self-reported programming skill level of 1 or 2. The first two participants were recruited for the pilot study, in which we tested and refined our study protocol and the system, and the remaining 12 participants were recruited for the formal user study. All the results reported in this chapter were based on the formal user study with these 12 participants, who were aged from 26 to 36 years (mean = 29.42, SD = 3.48); there were eight female and four male participants; and 11 participants rated their own programming skill level as 1 (out of 7), and only one participant self-rated as 2 (out of 7). It is noteworthy that the goal of this project is to examine the feasibility of using a natural-language interface to create IF-THEN rules. While our participants were of a younger population, we believe that a user study with 12 participants is sufficient to show the idea of InstructableCrowd works, and that InstructableCrowd can be helpful to some users.

In our user study, we scheduled an one-hour time slot with each participant and brought them in the lab, one at a time. Each participant was requested to create an IF-THEN rule, which would resolve each of the six 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 condition they were associated with. That is, each scenario was given to six subjects as condition 1 and to the other six subjects as condition 2. In addition, the scenarios were controlled for the order in which they appeared; that is, each scenario was given in each possible order (first, second, third, fourth, fifth, and last) exactly once for each condition. This was done in order to reduce the learning-effect. Participants were instructed to follow the scenarios as closely as possible but were allowed to propose minor changes during the conversation, *e.g.*, changing "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 (as shown in Figure 8.6,) 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. The compensation for each participant was $20.

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.

As listed in Table 8.1 and Table 8.2, in the user study, crowd workers and end-users had 10 sensors to choose from: Email, Bus, Message, GPS, Weather, Call, Clock, Calender, News, and Phone Body (for driving and phone falling); and six effectors: Message, Email, Alarm, Call, Notification, and Calendar (for adding an event).

8.4 Rule Quality Evaluation

In this section, we evaluated the quality of resulting rules in each setting. In order to assess the quality of a composed IF-THEN rule, we focused on two subtasks: **sensor/effector selection** and **attribute filling**. Composing an IF-THEN rule contains three sub-tasks: sensor/effector selection, trigger/action selection, and attribute filling. For instance, to know that you have an early meeting tomorrow, the "Calendar" sensor firstly needs to be selected, and then its "Future Event (Absolute Time)" trigger needs to be selected, and finally the "Start Time" attribute needs to be filled with "Before 8 am." Since each sensor used in our study on average only has 1.5 triggers (SD=0.71) and each effector only has 1 action, we did not evaluate the performance of trigger/action selection separately but merged it as a part of attribute filling. Namely, in the case that the triggers/actions selected by users or the crowd were incorrect, we noted the accuracy of attribute filling as zero in the sensor/effector.

In this section, we describe the evaluation results of InstructableCrowd and demonstrate how the system is able to produce high-quality IF-THEN rules via conversation.

8.4.1 Evaluation of Sensor/Effector Selection

The evaluation process was as follows: First, we expanded the set of our original gold-standard rules to include participant-created rules, which were useful, 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 setting a push 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 wherever appropriate. For instance, some participants thought that setting a notification is not enough and decided to send an email or set an alarm. We considered these alternative rules to be effective as well. Finally, a piece of software was created to perform an automated evaluation on all recorded rules.

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

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

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

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

When a rule was 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 8.3. 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 observed 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 noticed that participants actually corrected errors in the crowd-created rules, as both the average precision and recalls were higher in "Crowd+User" than "Crowd Only". For instance, in the "Late for Dinner" scenario (S6), one common mistake was that the crowd selected only one of Calender or GPS sensor, 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 that the current time was after 5 pm. One participant fixed this by adding the Clock sensor back to the IF.

	IF			**THEN**			**Avg**
	Precision	Recall	F1 score	Precision	Recall	F1 score	F1 score
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 8.3: Sensor/effector selection overall performance. Both "Crowd+User" and "Crowd Voting" settings achieved comparable performances to that of the "Crowd Only" setting in both IF and THEN parts.

We also evaluated the performance based on the scenarios' difficulty level. The dynamics of F1-scores are shown in Figure 8.7. 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. These 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 [149].

8.4.2 Evaluation of Attribute Filling

The evaluation process of attribute filling is similar to that of sensor/effector selection. Any value for an attribute that seemed appropriate was considered to be correct. For instance, the content of the sent messages or emails could vary, so we manually labeled the effectiveness of each "content" attribute in effectors; the "Day" attribute (Table 8.1) in the Weather sensor of S3 could be set to either "Today" or "Tomorrow;" however, it would only be judged as correct if

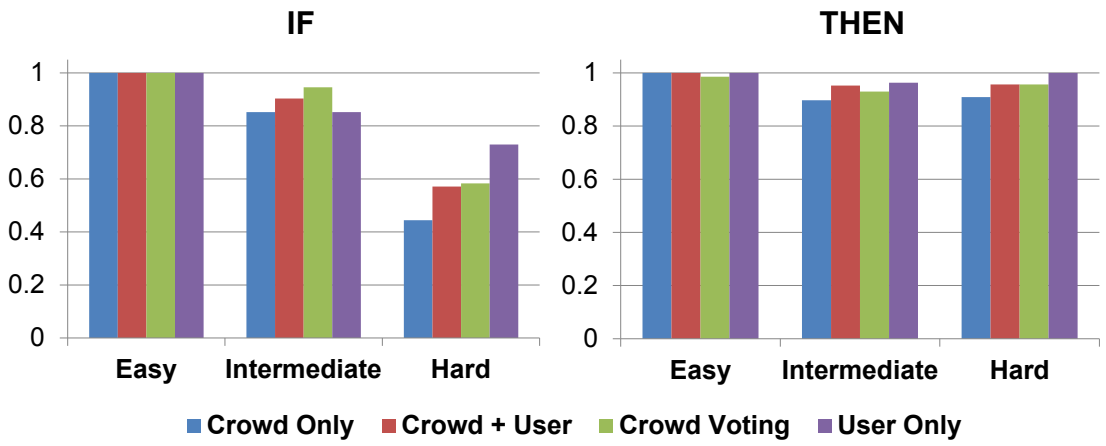


Figure 8.7: Average F1-score of sensor/effector selection in easy, intermediate, and hard scenarios. "Crowd Voting" performed similarly or slightly better than "User Only" in easy and intermediate rules but worse in hard rules.

the Alarm's "Day" attribute (Table 8.2) was set to the same value. A software was created to evaluate these attributes automatically.

For a given sensor/effector S that is correctly selected, we calculated the accuracy of its attribute values as follows.

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

$$\text{If trigger/action of } S \text{ is incorrect, Accuracy} = 0.$$

The overall evaluation results of attribute filling are shown in Table 8.4. 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 8.8. 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 the meanings of triggers**. 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 their "dad" often sent them messages (instead of "mom"), and the crowd therefore filled "dad" in the "Sent By" attribute; in S5, one participant did not mention to the crowd that it usually takes five minutes to walk to the bus stop, so the crowd arbitrarily filled the "In How Many Minutes" attribute of Bus sensor with two minutes (trigger = "Future location"). Another source of error is the misunderstanding the meanings of triggers. In S6, we found that some crowd workers confused the "Future Event (absolute time)" trigger with "Future Event (relative time)" trigger of the Calendar sensor. In addition, both users and crowd workers have **typos** in their attributes. For instance, a worker misspelled "Steelers" as "Stelers" in S1, and

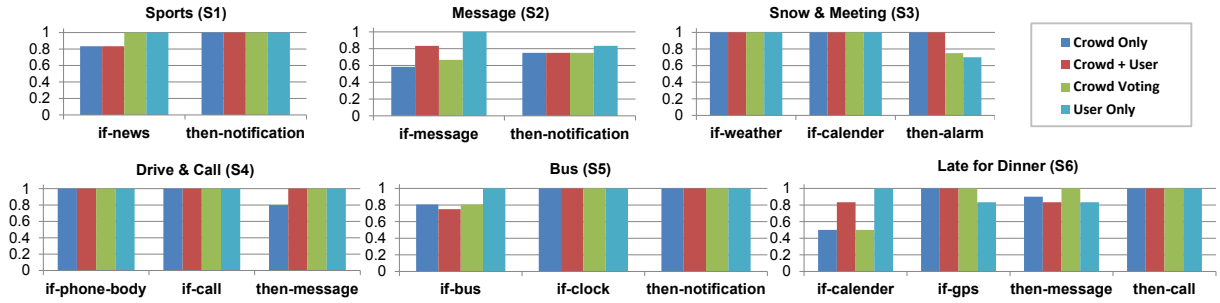


Figure 8.8: Average accuracy of attribute filling of correctly-selected sensors/effectors. "Crowd Voting" performed similarly as "User Only" in most cases. We analyzed S2, S5, and S6 and found that crowd errors are mainly caused by communication gap and misunderstanding of attributes.

another worker entered "19:00" as the "start time of the meeting" in S3, while the expected answer was "07:00."

	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 8.4: Attribute filling overall performance. 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.

8.5 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 a rule. In our study, participants spent an average of two 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 three 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 calculate the final rule ("Crowd Voting"). The complete timeline is shown in Figure 8.9. We also plot the task completion time of three settings in Figure 8.10.

On average, the "Crowd Voting" setting took a user one more minute to create a rule than

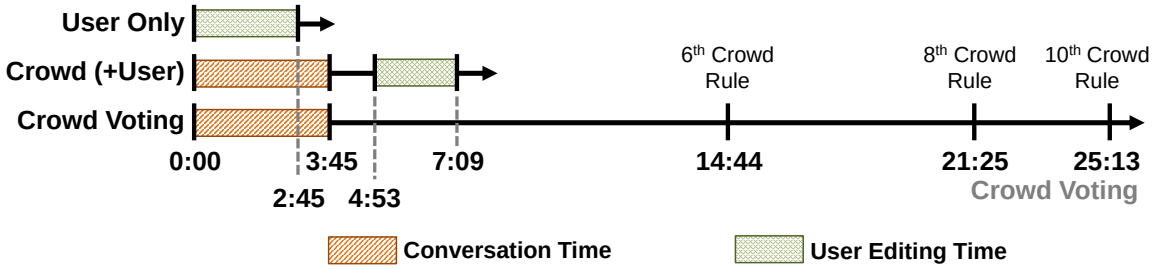


Figure 8.9: The complete timeline of InstructableCrowd. With the cost of a slightly longer user active time, InstructableCrowd is able to generate rules with comparable quality user-created rules. Furthermore, in our post-study survey (Section 8.6.1), the participants who preferred using InstructableCrowd over rule editor claimed that InstructableCrowd is "faster" or "quicker", while their user active time of using InstructableCrowd is actually longer.

that of the "User Only" setting. That is to say, with the cost of a slightly longer user active time, InstructableCrowd opens up a hand-free manner of creating IF-THEN rule via conversations with the crowd. We believe this is reasonable because an advantage of a speech interface is that it can be hands-free, so users can intersperse other activities while conversing to create their rules. According to our technical evaluation, the resulting rules from InstructableCrowd is as high quality as user-created rules. It is also noteworthy that user's cognitive load when editing a rule manually and when talking with a conversational partner 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. In our post-study survey, which we will describe in Section 8.6.1, the participants who preferred using InstructableCrowd over rule editor claimed that InstructableCrowd is "faster" or "quicker", while their user active time of using InstructableCrowd is actually longer.

8.6 Qualitative Results

In addition to the technical evaluation, we also collected qualitative feedback about InstructableCrowd from participants. This result suggests that InstructableCrowd provides an easier way to compose applications for the users who have difficulty creating complex rules manually on their phones.

8.6.1 Feedback from Participants

We collected participants' subjective feedback immediately after they finished the lab-based study. We asked participants what method they preferred, *i.e.*, InstructableCrowd ("Crowd+User") or rule editor ("User Only"), and grouped them into two groups according to their preference. The feedback we received was that four participants preferred InstructableCrowd, seven participants preferred the rule editor, and one participant had no preference. We also asked participants to rate the difficulty of using InstructableCrowd versus using the rule editor themselves, on a Likert scale, where 1 corresponds to very easy, 2 to easy, 3 to slightly easy, 4 to neither easy nor hard,

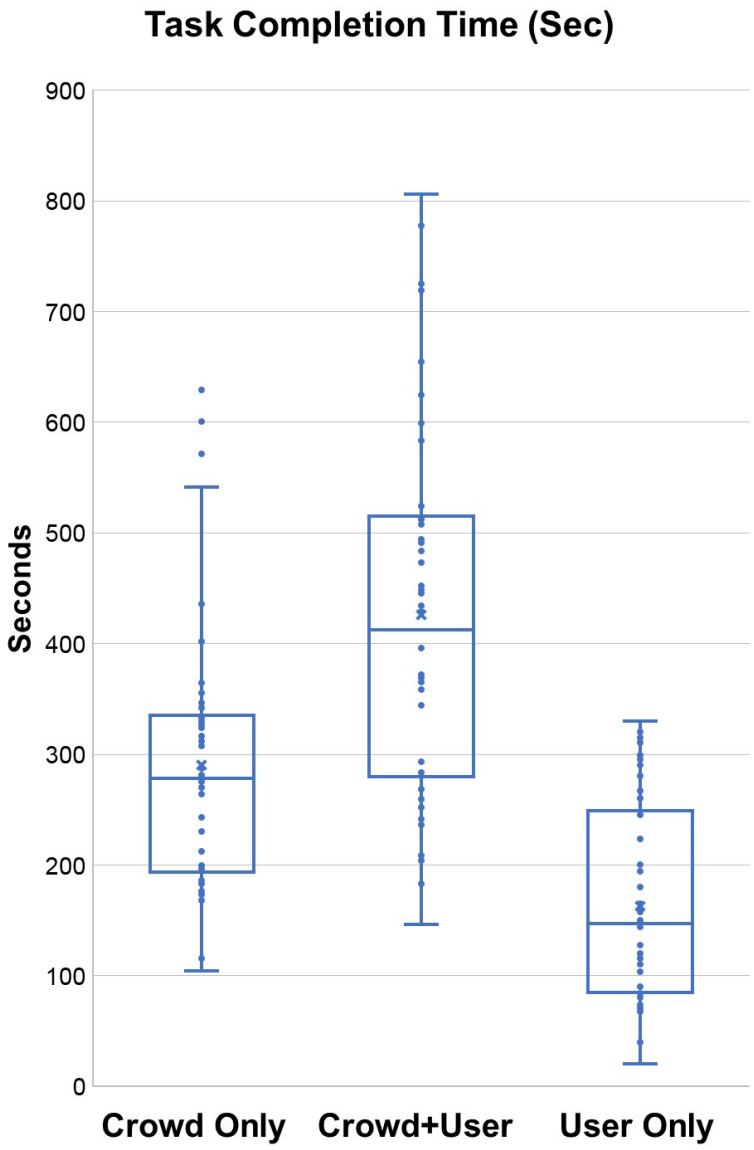


Figure 8.10: The box and whisker diagram of task completion time of *(i)* Crowd Only, *(ii)* Crowd + User, and *(iii)* User Only settings. Although the "Crowd Only" and "Crowd + User" settings often took longer than the "User Only" setting, InstructableCrowd opens up a hand-free manner of creating IF-THEN rule via conversations with the crowd. We believe this is reasonable because an advantage of a speech interface is that it can be hands-free, so users can intersperse other activities while conversing to create their rules.

		Participants Grouped by Preference	
		Prefer InstructableCrowd	**Prefer Rule Editor**
No. of participants		4	7
Avg. Difficulty Rating	**InstructableCrowd (Crowd+User)**	**3.25** (SD=1.50)	3.57 (SD=1.62)
	Rule Editor (User Only)	4.25 (SD=1.71)	**2.29** (SD=0.76)
Avg. Time to Create a Rule Manually (User Only) (mm:ss)		03:15 (SD=01:20)	02:30 (SD=0:45)

Table 8.5: The average difficulty ratings and rule composing time of participants that prefer InstructableCrowd v.s. rule editor. Difficulty rating ranged from 1 (very easy) to 7 (very hard). The participants who preferred InstructableCrowd had a higher difficulty rating for using the rule editor, and also took longer to manually compose a rule.

5 to slightly hard, 6 to hard, and 7 corresponds to very hard. As shown in Table 8.5, 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. 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. Namely, the users who had a hard time using the rule editor prefer to use InstructableCrowd. A similar relation was not found between "user's difficulty rating on InstructableCrowd" and "preferring the rule editor" (correlation coefficient = 0.06). Table 8.5 also shows that the participants who preferred InstructableCrowd also took longer than the other group to manually compose an IF-THEN rule on average. This result suggests that InstructableCrowd provides an easier ways to create IF-THEN rules for the users who have difficulty creating complex rules manually on mobile phones. The one participant who had no preference between using the rule editor and using InstructableCrowd 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, three out of these four participants said that InstructableCrowd is "faster" or "quicker", while they actually spent longer time to create a rule via InstructableCrowd when comparing to the time it took them when using the rule editor. This could be so 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 that is sent from multiple crowd workers. One participant who chose to use speech input said it is "faster" because he/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. Six 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 relay the conversation to ask help from a server."; Three out of 12 participants said they would choose InstructableCrowd when they are busy, e.g., "I would use it when I am busy."

8.6.2 Information Inquiry, Confirmation, and Suggestions in Conversations

We analyzed the conversations between the participants and the crowd and found that the responses from the crowd were often requests for more information or explicit confirmations of user's intent. Both are known to be common dialogue acts of conversational agents [156].

Most of the conversations between users and the crowd happens for collecting information. For instance, in the following conversation of S3, crowd workers ask for the information that 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 conversation of S6, 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.

In the following conversation, with a different user for the same scenario, S6, a different crowd worker asked similar follow-up questions:

user I don't want to be late for home too often, otherwise my wife would get angry at me

crowd **So how may I help you**

crowd **when do you want to get an alert?**

user can you send Amy a message saying I might be home late

user yes

crowd **what time do you want this to be sent?**

user if I'm going to be late

crowd **what time is late?**

user for our scheduled dinner on my calendar

Crowd workers sometimes confirmed with the users information which was conveyed previously. For example, in the following conversation of S5, a worker asked a confirmation question about the time.

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

Furthermore, an open conversation can lead to solutions that the user did not think of. For example, in the following conversation of S2, the crowd worker suggested to send a message back or to use an alarm/notification instead of setting a phone call. The alternatives that the crowd came up with demonstrates their potential to be creative and think of solutions that the user might not have.

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!

8.6.3 Alternative Solutions for the Same Scenario

We observed that participants and workers could come up with different rules in response to the same scenario, for four main reasons: First, people have their own preferred ways to be notified under different circumstances and thus, sometimes, chose different effectors than we intended in their rules. For instance, more than one participant tried to add extra effectors, such as an alarm in the "Message" scenario (S2.) because they believed missing a message about the hospitalized grandfather can be quite serious. Second, similarly, users also have their own preferences for sensors. For example, in the "Snow & Meeting" scenario (S3,) one participant selected "News" in addition to the gold-standard sensors and argued that s/he would only wake up for heavy snow, which is likely to be mentioned in the news. Third, some alternative rules created by crowd workers may be caused by the ambiguities in user's instruction. For instance, in the following conversation of S4, 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.

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**

Finally, sometimes, two different rules can behave similarly or even identically in the real world. 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," since the Bus 53 usually takes 5 minutes to drive from Hamilton St. stop to Washington St. Both rules occurred in our study.

8.7 Discussion

In this chapter, we introduced InstructableCrowd, a system that allows users to create IF-THEN rules for smartphones via conversation with the crowd. This work provides a potential route toward more interesting conversations with intelligent agents than is currently possible. In this section, we discuss some of the issues and reflections that came from the development and study of InstructableCrowd.

8.7.1 Assessing Performance and Goal Achievement

The study showed that the performance of the crowd system is nearly the same as that of a typical GUI in terms of the *quality* of the generated rules. This might lead some to question whether users would want to use InstructableCrowd if it is not *better* than other options at creating accurate IF-THEN rules. The motivation of InstructableCrowd is to challenge the traditional methods of manually composing an IF-THEN rule within the context of performing complex tasks via alternative interfaces. The key question we wanted to answer is: "Can the system perform as well as users themselves, while employing a new method of doing it?" Outsourcing complex tasks to the crowd is not always about whether or not the system can do it "better." Often it is about opening up an opportunity to achieve the same goal using a different technology or method, which in this case is via natural-language interface. In this respect, being "better" really depends on how one is assessing achievement of the goal. In prior projects within this theme, crowd-powered systems have not always performed *better* than users. In WearWrite [112], Chorus [90], and Knowledge Accelerator [25, 53], the proposed solutions did not necessarily produce results that were faster or of higher quality than traditional methods. The value of these projects was opening up new possibilities of completing tasks in ways that were not possible before, especially with respect to flexibility. Creating a blog post by talking to a smartwatch with WearWrite will not necessarily result in higher article quality than typing it on a laptop, but the system lets users create content on the fly nearly anywhere. Searching via Chorus crowd workers might not provide better results than just using a search engine, but it is much more convenient. Similarly, Knowledge-Accelerator's use of crowd workers allows a user to ask an open-ended question and get a sophisticated answer in few hours, and open-ended questions are something that computers do not deal with very well.

8.7.2 Challenges in Producing High-Quality Rules

Creating a multi-part IF-THEN rule is difficult because computer-executable rules (like all programs) have little tolerance for mistakes. If we break down an IF-THEN rule to a composition of sensors and effectors with attribute values, experiments have shown that humans are reasonably good at composing sensors/effectors and filling their attributes, respectively. However, when we add up all the work, any mistakes will make the resulting IF-THEN rule ineffective. A natural response to this issue would be to enforce stricter validation for human input in rule creation. However, a strict input validation on the interface would increase the time it takes to create a rule for both users and the crowd and frustrate users more easily. It would also increase the engineering effort required to add a new sensor or effector to the system, which often comes with arbitrary constraints. IFTTT, as a successful rule-creation product, avoids multiple sensors and effectors and uses a user-friendly workflow to balance possible user frustration. Our project suggests using conversation and iterative editing to permit robust rule creation.

8.7.3 Rule Validation

One of the most common issues faced by the Decision-Rule Engine is the *rule conflict resolution* issue, *i.e.*, deciding which rule should be triggered when there are multiple with the same set of conditions (IFs) but a different set of actions (THENs). If a user receives multiple rules during the same conversational session, it is reasonable to assume that they are redundant and to allow the user to pick only a single rule from this set. However, if the user creates many rules in many different sessions, he/she may forget about a created rule and attempt to create the same rule again. Furthermore, the user may at first create a very specific rule (e.g., IF I have a meeting at 9 am, THEN notify me the night before) and later try to generalize it (e.g., IF I have a meeting at 10 am or earlier, THEN notify me the night before). If the Decision-Rule Engine were to follow these rules regardless of conflicts, the same action might be executed more than once, which is not likely the user's intent. Currently, the monitoring/tracking module may detect such conflicts and automatically subsume the less-used rules, but further research is required into identifying these cases and alerting the user in advance. One approach could be keeping these conflicting rules and defining some heuristics that would determine when a rule should subsume or inhibit others, or when they should be executed sequentially, etc. Some technologies such as TrigGen [111] have also been developed for automatically detecting missing triggers in a trigger-action rule. Another approach could be defining a mechanism that removes those rules that are redundant or conflicting and less relevant than others (with the user's approval).

8.7.4 Timing of Executing Triggers and Actions

Different sensors and effectors may require very different frequencies. For example, while a weather-related sensor trigger such as "IF it is snowing early in the morning" can be checked once every 24 hours, a "Phone Body" sensor trigger connected to the phone's accelerometer (e.g., "IF the phone is dropping towards the floor") might need to be checked every 100 milliseconds. Other sensor conditions, such as calendar events (*e.g.* "IF I have a meeting tomorrow before 10am") may be validated immediately after the rule is created, and then checked again

every hour (in case new meetings have been added). Similarly, effectors also have different execution timing requirements. Some actions can be executed immediately after the conditions are met, while others must be scheduled for later execution. For instance, the action "THEN show me a notification right now" is executed right after the conditions are fulfilled, whereas the action "THEN send me a reminder tonight at 10 pm" would be scheduled for execution at the appropriate time. Currently, the Rule Validator in InstructableCrowd's middleware uses different timing validation mechanisms for different sensors and effectors. To scale up to a larger number of sensors and effectors, a more systematic manner for categorizing the frequency ranges of sensors and effectors is likely required.

8.7.5 User Privacy

One participant in our study expressed a concern about user privacy. In the current prototype, a limited view of a user's personal information (*e.g.*, a contact list created for the purpose of the study) was exposed to crowd workers. In the future, we may use aliases that are either automatically assigned or created by the user to prevent true names or other information from being disseminated to crowd workers. For instance, instead of an actual address, the user could provide an alias such as "Home" or "Office" when talking to the crowd. Aliases can also be used to protect information about people or time (*e.g.*, using "Wife" instead of "Amy," or "Birthday" instead of the actual date). However, the use of aliases cannot completely prevent the user from providing personal information in a conversation. While several privacy-preserving human computation workflows have been proposed for annotating videos [94] and accessing users' personal information [142], privacy is still a well-known issue in the field of crowdsourcing, since the data is processed by human workers. A future direction is to further explore privacy issues that may arise with conversational interfaces.

8.7.6 Limitations

One natural limitation of the architecture of InstructableCrowd is that all the sensors and effectors must be *comprehensible* to the majority of crowd workers. For example, despite being one of the most common built-in sensors in smartphones, the accelerometer sensor's raw output is very difficult to use directly by non-experts to interpret certain movements of the phone (*e.g.*, falling or being in motion while driving or walking). Future systems may find value in explicitly recruiting to their crowds people with programming expertise who can provide abstractions of raw sensor values that could be shared and reused by others. Using current sensors to express high-level semantics (*e.g.*, determining when the user is sleeping) requires specialized knowledge that most crowd workers likely do not have. IF-THEN rules have low tolerance for mistakes, and 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 whether or not to do it.

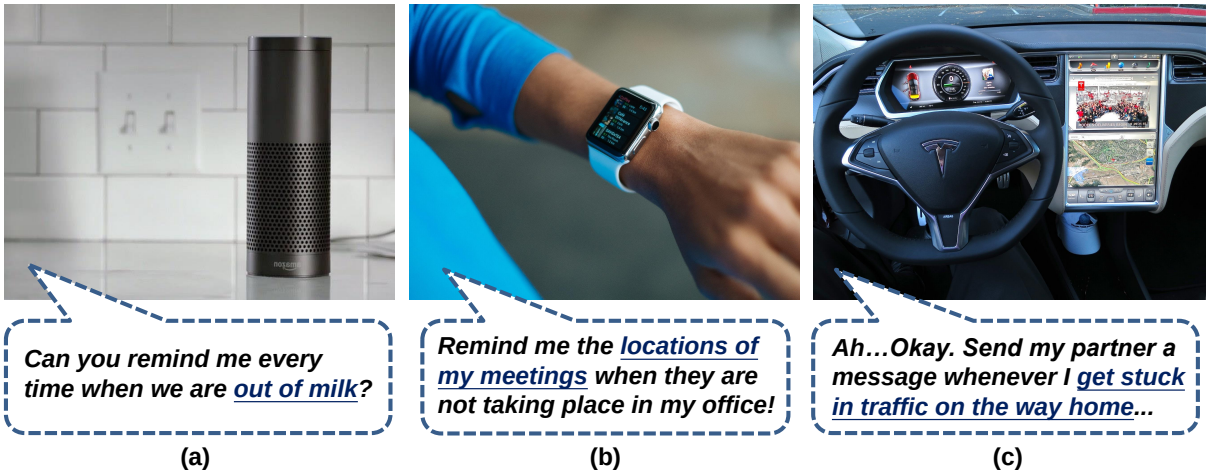


Figure 8.11: Scenarios of future conversational assistants that allow end-users to verbally create IF-THEN rules to control smart devices. When end-users experience a problem such as *(a)* being out of milk, *(b)* forgetting the meeting room, or *(c)* getting stuck in traffic when driving home, they can verbally instruct their assistants at the scene to set up IF-THEN rules to prevent the problems from happening again. The framework of InstructableCrowd can not only be implemented on the mobile phone, but also smart watch and voice-enabled devices such as Amazon's Echo.

8.8 Future Work

InstructableCrowd suggests a number of opportunities for future work. With the help of crowd workers, InstructableCrowd is able to convert a natural-language conversation to an IF-THEN rule. Human workers are known to be able to perform various tasks that automated systems still can not do; however, they often do it with the cost of longer latency and higher operating budget. One natural follow-up step is to explore the potential of automating the process of InstructableCrowd. While the automated approach did not perform as well as humans in prior work, a better performance can be expected when the system is able to collect larger amount of training data. Furthermore, the attribute filling task in creating IF-THEN rules is similar to the "slot filling" task in dialogue systems, in which we can take advantage of existent approaches such as Conditional Random Fields (CRF) [123] or Recurrent Neural Networks (RNN) [110]. Creating multi-part IF-THEN rules is a challenging task, for both human and machines. We imagine a future where automated components can work with human workers to make such systems more robust and scalable.

Furthermore, InstructableCrowd introduced a new interaction paradigm of conversational agents, which can not only be implemented in smartphones, but also be applied to smart homes, smart watches, voice-enabled devices such as Amazon's Echo, or smart cars in hand-free scenarios. End-users can freely record the problems they are experiencing and create an IF-THEN rule to solve it via any devices that are available at the spot. Figure 8.11 illustrates potential user scenarios of future InstructableCrowd on different devices. In a smart home setting, when the user opens a smart refrigerator and finds that they are out of milk, he/she can tell their Echo in the

kitchen to create a rule that reminds them when they do not have much milk left in the refrigerator (Figure 8.11 (a)); when a professor realizes that the next meeting will not be held in his/her own office but can not remember the room, he/she can set up a rule by talking to the smartwatch to set up a push notification about the room if the incoming meeting is in a different room; and when users get stuck in traffic when driving home, they can talk to the smart car panel and set up an automatic message informing their family whenever they will be late home (Figure 8.11 (c)). Voice interface opens up many possibilities for end-users to keep track of their behavior and improve life quality in the moment, and we believe that enabling users to create IF-THEN rules by talking to their smartphones is a promising start.

8.9 Summary

In this chapter, we introduced InstructableCrowd, a system that allows 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 built support for crowd workers to have a conversation with the users and allow them to suggest rules for the users. The user study shows that non-programmers can effectively create rules via conversation and suggests that collaboration between the user and the crowd while creating IF-THEN rules could be a fruitful area for future research. As we collect examples of IF-THEN rules, we will look for ways to use them to automate the creation of common IF-THEN patterns.

One limitation of Chorus was that it can only provide information to its users: it can not *do* anything or interact with their environments. Equipped with InstructableCrowd, users now can work with remote crowd workers to bring about powerful functionality despite the constraints of mobile and wearable devices.

Chapter 9

Discussion

In this chapter, we discuss our observations about users, limitations of our approach, ethical implications of a deployed crowd-powered systems, and trade-offs of our design decisions.

9.1 User Behavior

The deployment of Chorus was one of the few examples of a multi-turn interactive crowd-powered system being brought out of lab and tested in the wild. In this section, we describe our qualitative observations and thoughts about users' motivations for using Chorus and their interactions when the system was wrong.

9.1.1 Why Did People Use Chorus?

We are interested in why users chose to use Chorus to get information instead of other options such as searching on the Internet. While we did not conduct a survey particularly on *why* they signing up to use Chorus, we had the following mandatory question in our sign-up form:

> **Assume you have a "perfect" personal assistant, which you could contact via an instant messenger, please list at least one scenario in which you would use this personal assistant for.**

Users answered in free text. To understand users' motivations and their imaginations about a "perfect" personal assistant, we conducted open coding on these responses. After removing answers from our collaborators and research team members, 433 users responded to this question. 52.42% (227) of these users were students, 7.85% (34) were engineers, and 7.29% (32) were researchers, professors, or scientists. The average self-reported age of these users was 28.41 (SD = 9.65, Median = 25). 70.67% (306) of these users were male, 24.25% (105) were female, 3.93% (17) preferred not to answer, and 1.15% (5) were others.

We filtered out 29 users' responses because they were meaningless (*e.g.,* "gfasdadsagasadsa"), too vague (*e.g.,* "tasks"), or simply "I don't know." We then conducted open coding on the remaining 404 answers. Most of the responses contained only one scenario. Only 48 responses contained more than one scenario. In order to prevent user bias, for each user, we included a maximum of three scenarios mentioned earliest in the answer. The results of open coding are

shown in Table 9.1, sorted by how many users (number: #, percentages: %) mentioned this task type in their answers.

It turned out the top scenario mentioned by most users was managing their schedule or time; the second scenario was to ask for recommendations; and the third was to ask questions. Some tasks on this list were not supported by the deployed version of Chorus. For example, Chorus does not have access to user's personal calendar, to-do list, or emails; Chorus can not proactively initiate a conversation so can not be used as reminder or alarm; Chorus also can not make purchase or book tickets on the user's behalf; and Chorus can not help with any physical tasks. In the following, we elaborate several top scenarios mentioned in this study that Chorus supported.

Ask For Recommendations One of the most common uses of Chorus was asking for suggestions or recommendations, such as "what movie should I see tonight?" The following is an example:

user Is there any good movie in the cinemas for this weekend?

crowd **Hi**
Dead Pool, Jungle Book

user I have already seen those, any other suggestion?

crowd **Suicide Squad**

Recommendation tasks happened in various domain. For example, asking for cooking suggestions:

user What should I cook for dinner?

crowd **steak**
What are you in the mood for?

user I'm vegetarian.

crowd **pasta**
How about some grilled eggplant?

Or book recommendations:

user so i am looking for a book recommendation

crowd **Alright, is there a particular book or genre?**

And restaurant recommendations:

user What are some good places to eat in Pittsburgh?

crowd **What is your zip code?**

user [User's Zipcode]

crowd **How much would you like to spend?**
And do you have any preferences?

user $20
Sushi

	Task Type	#	%
1	Manage my personal time, schedule, calendar, or todo-list.	74	18.32%
2	Ask for recommendations based on some criteria (e.g., fashion, shopping, music, recipe.)	56	13.86%
3	Ask questions (one-shot question, or ask iteratively.)	50	12.38%
4	Search and collect information for a topic, or investigate a topic.	49	12.13%
5	Remind or notify me of my personal schedules; or set alarms.	35	8.66%
6	Others	29	7.18%
7	Make reservation (e.g., restaurants, doctor appoints), book tickets (e.g., flight), or purchase something on my behalf.	25	6.19%
8	Social or emotional support (e.g., motivate me, chitchat, etc.)	22	5.45%
9	Update me with external information sources (e.g., weather, traffic, news, fashion trends, etc.)	22	5.45%
10	Ask directions, or help me navigate.	16	3.96%
11	Read, reply, summarize, or organize my emails.	13	3.22%
12	Travel planning.	12	2.97%
13	Mundane tasks (e.g., laundry, fill forms), or being my substitute for boring tasks.	12	2.97%
14	Mentor or advice me; being my consultant.	11	2.72%
15	Physical tasks (e.g., meet up, pick up items, etc.)	9	2.23%
16	Make phone calls or send text on my behalf.	8	1.98%
17	Content creation (e.g., translate, write an essay, etc.)	7	1.73%
18	Learn new things (e.g., learn a new language.)	6	1.49%
19	Get feedback; review or evaluate something for me.	4	0.99%
20	Monitor my personal behavior (e.g., health, sleep, diet, etc.)	4	0.99%
21	Creative activities (e.g., brainstorming)	1	0.25%

Table 9.1: The answers to the question: **Assume you have a "perfect" personal assistant, which you could contact through an instant messenger, please list at least one scenario in which you would use this personal assistant for.** We conduced open coding on answers collected from 404 users who signed up to Chorus. Most of the responses contained only one scenario, and we only included a maximum of 3 scenarios from each answer. The task types are sorted by how many users (number: #, percentages: %) mentioned them in their answers.

crowd **Alright, perfect. One second!**
Check out this list: [The URL of the web page "Top 10 Restaurants in Pittsburgh"]

Another classic example was asking for gift suggestions; commonly, users had some vague gift ideas and iterated them with workers to narrow down to a few concrete options. Some users did not have any good gift ideas and started the conversation by describing the person for whom the gift was intended. Workers usually proposed a variety of gift ideas based on the description provided, refining the suggestions based on the user's feedback.

The following is a long discussion between a user and Chorus:

user can you help me figure out what to get my brother for Christmas?

crowd **How old is he?**

user I need ideas for a gift

crowd **Sure. What are your current ideas?**

user he just turned 30

crowd **What are his hobbies?**

user I'm not sure. He really likes skiing, camping, hiking
he also really likes football
and surfing
he lives in LA

crowd **Does he have a GoPro?**

user he does.

crowd **I would highly recommend an activity tracker, like this one [A link to Fitbit Fitness Wristband]**
What's your budget?

user around $100 is good

crowd **A good choice also might be tickets to his favorite football team.**

user he also just got engaged

crowd **original jersey of team he like should be good**

user so I could get him something for him and his fiance

crowd **Ok, this hi-tech wrist band is within your budget. Are you satisfied?**

user tickets to college football would be hard
but a jersey is not a bad idea
I'm not sure if a tracker is his thing

crowd **That's true**
Ok, let's move on to Jerseys
Check this out! ***[A link of an Amazon web page of a Football Jersey]***

user any ideas on a joint gift?
for him and his woman?

crowd **Do they have kids?**

user no
they are planning their wedding now

crowd **Pitty. A nice joint gift around $100 would be a new coffee machine.**
or some live performance for 2 ticket

user a coffee machine is a good idea

crowd **[A link to a Cuisinart Coffeemaker]**

user but I don't think she drinks coffee

crowd **This is one has the best reviews.** ***[A link of an Amazon web page of a Mr.Coffee Espresso Machine]***
Oh, she doesn't drink coffee. Good for her nerves, not for ours!!

user you are funny
chorus where are you?

crowd **Ok, what about this Steamer?? Good for health!!!** ***[A link of an Amazon web page of a steamer]***
How about this gift for a couples? [A link to a mug]
Using a steamer they can both live a healthy life!

user I like the mugs

crowd **Great**
Mugs are a good choice too.
Seriously? You're gonna get them mugs for a present???

user the steamer is good too
should I not?

crowd **Would you like more information on this steamer?**
How about buying an instant pot? It's a 7 in 1 multicooking device.

user I have heard great things about the instant pot
do you have a good link?

crowd **Yes, I actually just bought one myself.**
I bought one for my grandmother. She doesn't drink coffee too!
[A link of an Amazon web page of an instant pot]

user do you drink coffee?

crowd **Yes, quite a bit actually.**
:-) Have you made up your mind yet or would you like some more help?

user great! Thanks! I think this is a great gift idea!
I'll get the instant pot! I also need help with more gifts

crowd **Okay, come back when you have more questions!**

We noticed that many users had done some research about potential gift options and just wanted workers to recommend more options for them. We speculate users' motivations of asking Chorus to recommend things is to have crowd workers to brainstorm ideas. This process is similar to the crowd-powered system IdeaGens [24], where the expert can provide real-time guidance to crowd workers to generate new ideas.

Ask Questions Users also asked Chorus many questions. From questions as short as "what time is it":

user what's the time in LA?

crowd **Hello, it's 12:12PM in LA :)**

Or the weather of a specific place:

user What's the weather in mexico city?

crowd **It's cloudy.**
67 degrees and cloudy. There's a 40% chance of rain.

For simple questions such as time and weather, we speculate that the main motivation might be curiosity. Many users, especially at the beginning of the deployment, were very curious about the system and crowdsourcing in general. We even saw some journalists send interview questions to workers via Chorus, for example:

[*User asks about weather in Seattle*]

user gotcha. how'd you get into it?

crowd **through mechanical turk**

user ah, good old mturk.

crowd **yes**
its pretty good platform

user so, no pressure whatsoever, but i'm a journalist and i'm thinking about doing a story about this type of crowdsourced work (especially stuff like chatbots). if you're at all willing to be interviewed, feel free to send me an email at *[email address]* - i'd love to know more about what it's like to do mturk work
anyway, that's it for me today, just curious how this worked and wanted to see if it could help me plan my weekend. you were super helpful - thank you!

crowd **Would this pay?!**
lol

Users also asked some questions that would take some more research to answer. For instance, comparing the differences between multiple items:

user What is the difference between tequila and mexcal?

crowd **One moment while we research this**
Mezcal (traditionally spelled mescal) is a Mexican distilled spirit that is made from the agave plant. *[A long paragraph explaining the differences.]*

Or asking about information that is not trivial to find:

user What is the average salary of a Bloomberg software engineer

crowd **$124,107 per year**
Anything else I can help you with?

user oh sorry I meant a software engineer intern

crowd **please let me check that info for you**
$71450

For these questions, we speculate the motivation of using Chorus is to delegate the searching task to workers. Some information might require more than one or two clicks to find on the Internet, and it would be easier to ask Chorus to get the answer. The following is another example:

user how many people are in USC University

crowd **Hello, including staff?**
University of Southern California?
Or just students?

user students and staff included

crowd **43000 students and 25211**
Undergraduates 19,000
Graduate and professional students 24,000
Total 43,000
25211 staff

Have Workers to Collect or Search Information Echoing the previous point, some users even explicitly assigned a topic to workers to collect information. The following example is a conversation about the strike in France in 2017:

user can you help me find info about on-going strikes in France?

crowd **What an I help you with?**
Sure!
Give me a minute, please
Air France is on strike: *[A link to the news of the strike]*
Do you need anything else?

user can you summarize?

crowd **Sure, give me a minute**
The airline Air France continues the second day of the strike of flight attendants. Sunday, March 19, joined the campaign about 38% of the stewards and stewardesses of the company, but the airline says that the flight schedule is almost affected, and promises to cancel 10% of flights. It is reported by RFI.
Stade Francais players strike over Racing 92 merger as France stars hold meeting against Guy Noves's wishes

user what are the causes of this strike?
what is the chance of succeeding?
what's the impact of an airline strike?

crowd **The unions represent just under half of the airline's cabin crew, who say they're unhappy over general working conditions.**
Air France has said that "some flights may be cancelled" and that the reduced crew means that passenger numbers may be limited, so the company "may not be able to honour all reservations".

user why now?
(the strike)

crowd **The employees are protesting against a new labor contract, as well as management's plans to create a new branch – a low-budget airline Boost.**
pilots expressed their opposition to the airline's plan to shift focus to a lower-cost (and lower-paying) subsidiary

Users also asked Chorus "how to do X" fairly often, *e.g.,* how to buy train tickets in Europe, how to change window curtains, or as follows, how to fix a broken phone screen:

user how can i fix my phone broken screen?

crowd **What type of phone do you have?**

user iphone 6s

crowd **How is the screen broken? Is it physically cracked, or is it a software issue?**
It won't be easy to do it yourself I am afraid.

user i droppped *[sic]*

crowd **Is it still under warranty? If so, you can send to Apple.**
Otherwise, there are many repair shops. Where do you live?
This video shows how.

user no
there is not warranty

crowd ***[A link to the YouTube video: "iPhone 6 Screen Replacement done in 5 minutes"]***
If you have the tools you might be able to fix it..
If you follow the video tutorial.

user so ı will buy screen and than i can fix
right?

crowd **Yes that is right.**
There are also many places that will fix it for you

user okey *[sic]* thank u

These questions often require some research on the Internet, and Chorus can also talk back and forth with the user to figure out all the customized details, for example, which phone is it, is it still under warranty, etc.

Social and Emotional Support Chorus can hold any types of conversations with users, including social chats. We observes that occasionally users expressed their feelings to Chorus. Sometimes, the user was simply bored:

user Hiii

crowd **Hi, how can i help you?**

user How are you
Heeeeeyy
Are you there?

crowd **I am there! How are you?**
Do you need help with something?

user I am bored. What about you?

Sometimes, the emotion can be strong. In the following conversation, the user talked to Chorus about having a fight with family during the New Year vacation, and workers tried to ask more details:

user i am depressed
i don't want to talk about it

crowd **ok i understand**
take you time
here when you are ready

user thx

In summary, Chorus provided numerous advantages in solving difficult, complex tasks over other methods. Users could use crowd workers to delegate jobs, help to brainstorm gift or cooking ideas, or collect information. Beyond these simple functions, users could have meaningful discussions on open-ended topics or even seeking some emotional support.

9.1.2 What Happened When the System Was Wrong?

Systems are not perfect and even with human oversight Chorus and Evorus made mistakes. We found that users could often recognize incorrect answers sent back by Chorus. While this dissertation did not focus on dialog error-handling strategies [136], we observed several common user reactions. We also saw workers come up with some strategies to recover from bad answers.

Users frequently ignored irrelevant responses. During our deployment, we found that users frequently ignored irrelevant or incorrect answers. Since Chorus and Evorus both prompted workers to provide more responses rather than fewer, when a bad or incorrect answer occurred, there were often other correct answers around near the same conversational turn. Users often chose to focus on these correct answers rather than reacting to the incorrect ones. Because the errors were usually obvious, we believe that in most cases the users could tell the correctness of the answers provided. For instance:

user Cool, so suggestions about tonight
EDM, Norwegian death metal, and EDM all work for me

crowd **There are many crowdworkers websites. Certainly more than two.**
Can you help me out?

user Are any of them here tonight in Philly?

crowd **Start an evening with drinks at one of the city's many rooftop bars, outdoor beer gardens or cocktail destinations, and then move on to dinner at any number of destination**
Doesnt seem like the trocadero has anything going on

Kimmel Center for the Performing Arts
Mann Center for the Performing Arts.
are you know kimmel center

user I'm looking for live music tonight ... Norwegian death metal, EDM , rock
That kinda thing

The responses in red were proposed by automatic chatbots and obviously irrelevant. The user simply ignored it and continued the conversation. We also noticed that human workers rarely returned responses that were totally "wrong" – some were just less useful than others or did not satisfy all the criteria specified. Automated chatbots proposed irrelevant responses or obviously incorrect answers much more often.

Some users who suspected a wrong answer asked follow-up questions for clarification. Another common reaction of users was to point out that an answer was wrong. For instance, users sometime provided concrete reasons to explain why the answers were invalid, or explicitly stated that they knew the answer was wrong. The following is a good example:

user i am hungry. How long do you think you can suggest a restaurant?

crowd **i don't tell my name but i help you find a good restaurant**
in which location are you

user walnut st.

crowd **Talula's Garden is good**

user never heard of it
are you sure it is on walnut st?
I could not find it.

crowd **Wawa Food Market, Barra Rossa, Le Pain Quotidien, Morimoto all are near walnut st.**
wt food you prefer?

user I am now pretty sure we are talking about two different places

While some users became annoyed based on their tone, Chorus users rarely blamed the system or workers for performing poorly.

Workers attempted to recover from bad answers. We observed that workers attempted to recover from bad or incorrect answers. One common strategy was to apologize to the user and then explain the reason for the error. If the bad response was proposed by chatbots, workers often referred to it as a system error or a "bug." If the low-quality response was written by other workers, we found that workers tended to provide more vague reasons such as "a system issue" rather than pointing fingers at other people. For example, in the following conversation, the irelevant response was written by a human worker, but the other worker just explained it was a "mistake":

[*User asked about the difference between shield volcano and composite volcano.*]

crowd **A shield is more explosive when water enters into its vent.**

user ok

crowd **Create an account to start this course today**
Fact: Mauna Loa is the highest shield volcano

user what account?

crowd **Sorry. It was a mistake.**

user ok

This might be because we explicitly instructed workers to collectively act as a single personal assistant, so blaming other workers could break the illusion of being a single agent. Workers also used similar explanations when the error was caused by a chatbot. In the following conversation, the user asked about NBA games, and a chatbot proposed incorrect answer:

user Who are the top team in the east?

crowd **Fillon, Le Pen or Macron**
The toronto raptors

user Fillon, Le Pen or Macron?

crowd **sorry that was an error**
toronto raptors top team

user Hmmm

Sometimes the error was caused by miscommunication or ambiguous instructions from users. We observed that workers rarely blamed users or explicitly reacted to user errors, typically continuing the conversations naturally.

9.2 Limitations

In this section we talk about the limitations and potential drawbacks of our approach.

9.2.1 Coverage of "Open Conversation"

While having human workers in the loop enables the system to hold most dialogs, we observed some limitations of our approach in terms of handling or automating certain types of conversations.

When Consensus is Not Enough As briefly mentioned in Chapter 3, Chorus and Evorus used worker consensus to choose high-quality responses, with the underlying assumption that the majority of workers likely agree on the same high-quality responses. However, sometimes this assumption did not hold. There were certain types of questions, *e.g.,* subjective or opinion-based ones, where workers did not agree with each other easily or quickly. Chorus currently cannot hold these types of conversations without breaking the single-agent illusion. One potential solution would be automatically detecting such topics and altering the worker interface or voting mechanism correspondingly.

Less Effective in Automating Domain-Independent Tasks and Social Conversations The Evorus framework focused only on automating domain-specific tasks. However, some real-world tasks are cross-domain or even independent of any domains. It would be difficult for Evorus to automate these tasks because its bot-selection algorithm used topic similarity to choose chatbots. For instance, negotiating (salary or price) is a common task in the real world, and negotiation can happen in many different domains, including real estate, shopping, or trading. If Evorus included a "negotiation bot" that could negotiate with users, the bot-selection algorithm would have a hard time using domain similarity to decide when to use it.

Furthermore, so far our work has focused more on tasks and less on social conversations. While the current version of Evorus included four chatterbots to provide general responses, we did not incorporate any existing approaches or knowledge about social conversations in our system. As in non-domain tasks, the bot-selection algorithm is also inefficient in selecting social chatterbots because they do not represent a clear, single domain.

Limited Support for Workers to Create Structures for Information Another limitation is that Chorus did not help workers organize information in a conversation in a structured way. The system only had a "memory board" where workers could write down a *list* of important facts. While this approach was sufficient for most conversations, for open-ended, long discussions such as "which school should I go to?" or "what's the meaning of working?" a list is not enough for organizing all the arguments and information. For example, in the following conversation about the politics of South Korea, it would be better if workers can maintain and display collected information in a more structural manner, rather than simply listing all the text at once.

user Hello
can you summarize politics in South Korea?

crowd **Politics of South Korea. Politics of the Republic of Korea takes place in the framework of a presidential representative democratic republic, whereby the President is the head of state, and of a multi-party system. Executive power is exercised by the gov...**
Is there anything else I can help you with?

user can you explain in more details?
also summarize a recent impeachment

crowd **outh Korea's President Park Geun-hye has become the country's first democratically elected leader to be forced from office**

user what are the causes?

crowd **as the permafrost melts, the world's biggest "megaslump" is expanding rapidly. Already about a kilometre long and 90m deep, it is widening by up to 20m a year,**
President Park was found guilty of corruption.

user how is permafrost related?

crowd **she take govt decision in favour of her friends**
Several news media including JTBC and Chosun Broadcasting Company reported that Choi, who has no official government position, had access to con-

fidential documents and information for the president, and acted as a very close confidant for the president. ...
she provide benefit and access to govt files etc to her friend . which got bust in public so people protested and court find her guilty
On December 5, 2016, three opposition parties agreed to introduce a joint impeachment motion against President Park Geun-hye. The motion, which was signed by 171 of 300 lawmakers, was put to a vote on Friday, 9 December 2016, and passed with 234 out of 3...
But she must now leave office - and her official residence - and a presidential election will be held within the next 60 days.

Multimodal Manners Finally, all our systems were text-based, which ignored any multimodal signals in human conversation. We are aware of a significant body of prior work, *e.g.,* embodied agents, that has used multimodal manners to interact with users. While speech recognition and synthesis can be easily added to Chorus, the multimodal signals were not involved in any components of our systems.

9.2.2 Being An "Agent"

Chorus, Evorus, and the concept of a "crowd agent" strongly imply the interaction metaphor of an "agent." We are aware of the historical debate about the pros and cons of the "agent" metaphor for human-computer interaction. Ben Shneiderman is one of the best-known researchers who opposes this model. As early as 1986, he argued that people are different from computers, and "human-human interaction is not necessarily an appropriate model for human operation of computers." [134] Ben Shneiderman and Pattie Maes had a landmark debate about the agent metaphor in 1997 [135]. In this debate, they discussed the pros and cons of "direct manipulation" and "software agents." Shneiderman argued for the power of good user interfaces and visualizations and stated that human-human interaction should not be the model for human operation of computers. Maes argued that having full control of unlimited actionable items and overly complex tasks is impractical, and thus future users will need "extra eyes and extra ears" to which to delegate tasks. In 2017, they revisited this topic and debated again. In this latest round, Maes argued for the concept of "human-computer integration," while Shneiderman countered that computers are tools rather than equal partners of humans, and the goal of interactive technology should be "ensuring human control, while increasing the level of automation." [47]

We would like to start our discussion with an example of a real-world complex task: tax preparation. Two widely used solutions for this complicated process are hiring human experts and using tax software. These reflect the two sides of the arguments around the "agent" metaphor: Human experts such as accountants or tax professionals utilize conversations and natural languages (*e.g.,* email) as primary means of communication. On the other hand, tax programs such as TurboTax use a nicely designed workflow and user interface to transform the complex tax preparation process into a task that most users can accomplish.

Each solution has its pros and cons. Human experts are more intelligent at understanding user needs and preferences, more flexible in communication, better at acknowledging context in order to provide customized suggestions, and can save time and effort for users. However,

tax professionals are expensive to hire, and an expert's time is not scalable. Tax software is more affordable (some entry-level versions of TurboTax are even free) and users have more direct control and instant feedback on the interface. However, software is less flexible, harder to customize for complex tax situations, and most importantly, requires much more time and effort from users to input all the data manually.

Similar trade-offs occur in many other real-world complex tasks. For planning travel, people use travel planning websites or talk to travel agencies; when house-hunting, they use online real estate sites or hire real estate agents. In our work, we took the path of the agent model because we saw great potential in it, and we also agreed with Maes's insights that the software agent is inevitable and powerful (Section 1.3.) However, we are aware that the agent model has some drawbacks. In this section, we discussed the limitations caused by being an "agent."

Miscommunication Users need to communicate problems and context to an agent to delegate tasks, which means miscommunication can happen. Chorus took advantage of the flexibility and expressiveness of natural language and enabled open conversation with users, but ambiguity is inevitable in human languages. When a user specified something verbally, it sometimes could be interpreted in different ways. In the following example, the workers misunderstood the word "Georgia" in the request:

user Could u tell me whats the weather in Georgia now?

crowd **Sure, give me one second to look it up for you**
It is 50 degrees F right now
and it is sunny

user Is there day or night?

crowd **It is daytime in georgia right now**
There is also a 17 mph wind and 37% humidity

user R u talking bout the Country or The state?

crowd **The state**
Would you like the country instead?

user Yes

crowd **Okay, sorry about that. I will check for Georgia the country**
Tbilisi, Georgia - 8C at the moment

Another example occurred when a Pittsburgh user was trying to specify a location in Oakland. This is the name of a neighborhood in Pittsburgh, but workers confused it with Oakland, California:

user Hi. Can you tell me the best tea shop in Oakland

crowd **Hello, absolutely, just a moment.**
Sure
Give me a minute
How about the Golden Tea Shop in Chinatown?
It has excellent reviews on Yelp

user Oakland in pittsburgh *[sic]*

crowd **Oh, OK. Let me check again, just a second.**
Try Rally House North Shore
You can give Spice Island Tea House a try
You can go to Earthly Tea and Coffee
Fuku Tea has good reviews as well

Language was not the only cause of miscommunication. Sometimes the user was not aware of certain pieces of information that were required and only realized this upon seeing the answer, or the user simply forgot to pass all the contextual information to the agent. For example, in the following conversation, the user realized workers were not aware of the fact that his/her mom did not speak English after they suggested an English book:

user can you suggest a gift for my mom's birthday?

crowd **Sure let me look something for you**
Any ideas on her preferences? Like Music etc
How old is she?

user she likes dancing music, she is at her 50's
but i don't want to buy her music

[*The user discuss different options with the crowd.*]

crowd **Here's the "Imported books" section, you could try to find a book in English: *[The link of the book]***
Does she like flowers? this might be a good option: *[The link of the book about flower]*

user She does not speak english

crowd **She speaks chinese?**

user yes

crowd **Here's the site for chinese books: *[The link to a Chinese book site]***

user sorry, i should have told you that

crowd **it's okay!**

It is impossible that the agent is aware of all the context, preferences, and constraints that the user knows, and thereforem, miscommunication is inevitable.

Potentially Longer Task Completion Time Another limitation of outsourcing a task to an agent is that the end-to-end task completion time could be longer than that required if the users did the task themselves, especially for tasks where users were proficient. As mentioned in Section 9.1.1, users did not always use Chorus because the task was difficult for them. For tasks that were already familiar or easy, it could take Chorus longer because of the need to communicate with the agent, iterate back and forth to narrow down results, and finally obtain a result. In our experiments with InstructableCrowd (Chapter 8), in some scenarios, the time the system took to produce IF-THEN rules was longer than that required by users on a mobile interface. While the users do not need to commit any time or effort while waiting, we acknowledge that communicating with Chorus takes time and the end-to-end completion time could be longer.

Hard or Expensive for Hands-on Controls Authorizing an agent to do tasks means users do not have hands-on controls of all task details. The agent often needs to make some small decisions without checking constantly with the user, which could be a problem for tasks that need to be precisely customized (*e.g.,* the gift for mom) or when the user strongly prefers hands-on controls of the minutiae. This is one of the fundamental limitations when hiring any agency.

Higher User Expectation Finally, another fundamental limitation is that anthropomorphic agents create higher expectations of users that can harm the user experience or create greater frustration in case of problems. For instance, if a user verbally asked an Echo device to "turn off the light" and it failed to understand, the user might experience more frustration than if a manual switch did not work. However, we also argue that higher user expectations will motivate users to explore new tasks that they did not think were possible and thus push the boundaries of the capability of conversational assistants. Users can adapt to technologies, even it is frustrating at the beginning. When Apple first introduced Siri, it was with substantial enthusiasm among users. While the system ended up disappointing some users [28], it indeed opened up many new research problems and new use cases, influencing how today's users interact with personal assistants. Over the long term, higher expectations are not necessarily a limitation but an opportunity.

9.2.3 Mechanism Vulnerability

There are two known vulnerabilities of the current mechanisms of Chorus and Evorus: collaborative spammers and workers talking to themselves via Chorus.

Collaborative Spammers Our system is vulnerable to collaborative spammers. One core underlying assumption of Chorus is that the majority of workers in each conversation are *not* spammers, and therefore the voting process can make sure that the few malicious workers do not influence the output. However, since the system has been deployed for two years and a significant number of workers have used the system, we have observed that some collaborative spammers started targeting our HITs on Mechanical Turk. They worked together to enter the same conversation, and always upvoted each other's spamming responses. Currently, Chorus uses a maximum of five workers in each conversation, which means three collaborative spammers can control all the outcomes of the majority voting process. We learned about this vulnerability because there were three spamming workers sent responses that were similar to each other, and always upvoted each other's spamming responses, in the same conversation. In the following example, totally four workers proposed responses. We highlighted the three spamming workers' responses in red, blue, and orange, respectively:

user Hi chorus, how are you today?

crowd **I am doing good, how are you?**
How may I help you today?

user I'm doing great
I'm in Sunnyvale California and looking for something to do tonight

crowd **Let me look that up for you.**
Your name
hello
doing pla any games
woww

user These aren't helpful

crowd **superr**
nice
ANY GAMES
yeahh
What are your hobbies?
NIGHT PARTY ANYBODY HERE

user i asked for something to do in Sunnyvale, CA

crowd **nice**
[A Yelp link of night life in Sunnyvale]
what link
what your country?
please

user hi folks, these HITs will be rejected because you're spamming
your IDs will be blocked

We later learned that these three workers *always* upvoted each others' spamming response so these low-quality answers can be accepted and be sent back to the user. It is noteworthy that in this conversation, the only one "good" worker, who reported this incident to us, was the minority and thus can not change the outcome of majority voting even he/she tried to downvote the spamming responses.

We investigated this problem by checking the chat log of these worker IDs and discussing with some workers we have been collaborating with. We speculate that either these worker accounts were owned by the same person – some workers told us buying multiple MTurk worker accounts is possible – or they were in the same household. A worker told us via email, "I wouldn't be surprised if they are (collaborative). I know few workers who are from the same household or many who are rl (real-life) friends, and for them exploiting this isn't really difficult." This problem could potentially be reduced by verifying each worker's IP address or adding some random factors when assigning workers to conversations.

When Workers Talking to Themselves Since we cannot verify the true identities of either users or MTurk workers, we cannot prevent workers from talking to themselves by using Chorus as a user. We discovered this situation because one worker contacted us using a personal email address to ask why we blocked the individual. We found that this worker's email address was the same as that of the user the worker constantly talked with in Chorus. We initially blocked this worker because of responses that did not make much sense, even though the user seemed to be fine with them.

During the Chorus deployment, we did not try to avoid having workers act as users on our system. In fact, we encouraged some workers to try the system out to understand it better.

However, when workers talk to themselves using Chorus constantly, the purpose is not satisfying curiosity or understanding the system. The main interest at this point has become financial gain. By using Chorus, the worker can trigger the system to post HITs on Amazon Mechnical Turk; they can accept the task and start spamming conversations without hurting any real users' experiences.

This problem could be reduced if we hired in-house workers to operate the system, allowing us to identify the workers' true identities.

9.3 Ethical Implications

As one of the few deployed crowd-powered systems, Chorus and Evorus provoked some discussions about ethical implications, especially privacy concerns. In this section, we discuss the ethical issues that were brought up during our deployment.

9.3.1 User's Privacy

Chorus used human workers to collectively serve as personal assistants to whom users passed personal information. One apparent concern here was user privacy. It is noteworthy that to get help with personal tasks, a certain level of privacy compromise is often needed. For instance, if a user wants to hire a butler to take care of his or her everyday needs, the butler probably needs some personal information. Even fully automated intelligent personal assistants such as Amazon Echo keep track of users' personal information, *e.g.,* name, address (to schedule services or provide a weather forecast), items the user ordered, etc. Arguably, fully automated systems do not *need* to expose personal information to human workers, but it is unclear whether or not companies such as Amazon [140, 181] or Google [40] pass collected information to their employees. Even without human intervention, passing processed user data that only machines can understand (*e.g.,* word vectors) to fully automated components, such as advertising systems, could still compromise some of a user's privacy [150]. In other words, many of the privacy concerns related to Chorus are general concerns shared among most personal assistants.

Workers have direct access of the raw conversation data. The main privacy issue where Chorus differs from Echo or Google Home is that human workers have *direct access* to the raw conversation data and the user. In Chorus, the raw conversations are, by design, exposed to human workers in their entirety. During the conversation, workers can see the entire chat log of the same session, and there are no effective ways to prevent them from keeping a copy of the conversation. The impact of this issue can be reduced by having workers sign explicit agreements that disallow keeping a copy or sharing dialog content with others. One future direction is to alter or hide part of conversation from workers, but it would take further research to understand this obfuscation's influence on conversation quality.

Workers have direct access to the users. Workers can directly talk with users in Chorus. As mentioned in Chapter 3, workers could ask questions about users' private information not

relevant to the assistance task. In addition to explicit questions, workers could also use social engineering to collect sensitive data such as birthdates or the answers to common security question, *e.g.,* mother's maiden name.

These two concerns suggest that a thorough protocol for workers should be introduced in deployed crowd-powered systems, which should include a standard operating procedure (SOP) for handling sensitive and personal information from users, and general guidelines for conversing with users. It would take more research to study how these details of the protocol could be enforced among workers hired from Amazon Mechanical Turk. An alternative solution would be to use in-house workers to operate the system, where stronger training and monitoring would be possible.

9.3.2 Workers' Privacy

Workers can ask questions of users in Chorus, but users can ask questions of the workers as well. During our study, some users tried to convince workers to disclose their personal information. Some workers provided information, like their names or where they were from, mostly because the user was asking and they might feel this made the conversation more friendly or social. While users cannot verify the truthfulness of such statements, workers' privacy should also be considered when deploying a crowd-powered system. As with worker training and protocols, a set of rules should also be explicitly communicated to users. In our system, we had a set of rules for users, such as prohibiting malicious language and not asking Chorus to perform physical tasks like delivering items or meeting in person. For future deployed crowd-powered system, we strongly suggest that developers and researchers design instructions for users to protect workers.

9.3.3 Reusing Crowd Responses

Evorus reused responses that were generated by crowd workers to respond to later similar questions. In our study, most responses could be effectively reused in many other similar conversations, but we noticed that a few responses contained a user's personal information, such as an address. One common scenario was that crowd workers repeated the information that the user just said to confirm the request, and that repeated message was later reused by Evorus. This situation was problematic because Evorus could unintentionally share a user's private information with other users. One solution to this is to use text-processing technologies, *e.g.,* text duplication detection or entity recognition, to remove sensitive data before reusing it. Alternatively, the system could be structured to have crowd workers filter out responses that contain personal information. A potential issue with these practices is that cleaning data manually is expensive and automated approaches are not always reliable.

9.3.4 Malicious Language

Finally, as mentioned in Chapter 3, malicious language occurred in Chorus: Both users and workers said improper things to each other during our deployment. While we explicitly informed

both users and workers that there were real humans on each side of Chorus and not to send any malicious content, inappropriate conversations still happened.

One solution to this problem would be to introduce text-processing technologies such as profanity detection [138] or abusive language detection [115]. Having workers police and report on each other is also a reasonable way to reduce malicious behavior, and, in fact, most online platforms have this function: on Facebook, Twitter, or YouTube, users are allowed to report problematic content and users. However, when we discussed with workers about making "reporting malicious workers" an explicit function on the worker interface, most did not think it would be a good idea because it would be likely to be abused by spammers against "good workers." We would like to encourage future developers and researchers to create systems that focus on incentivizing positive behavior among workers instead of punishing malicious acts. Amazon Mechanical Turk is an ecosystem, and we believe that encouraging good workers could result in better long-term results.

9.4 Technical Decisions

We hope that Evorus will encourage more people to explore the potential of crowd-AI architectures, especially using them to transit crowd-powered systems to (semi-)automated systems. One key feature of Evorus was that the system showed both *(i)* suggestions generated by a small group of human workers, and *(ii)* a few top suggestions generated by (or selected from) a larger group of automated bots at the same time; human workers could then select from them. If this selection process can be learned by machines over time, more and more tasks can be dispatched to bots gradually. Crowd-AI frameworks similar to Evorus could be applied to many other applications, especially lengthy tasks that can be formulated as a sequence of smaller decisions, *e.g.,* audio transcription, translation, or writing.

To make it easier for other researchers to apply crowd-AI architectures to different tasks, in this subsection we explicitly describe our higher-level technical decisions and discuss their pros and cons.

9.4.1 Retrieval vs. Generation

One high-level technical decision we made was to formulate "holding a conversation" as a sequence of "retrieval" tasks, which assumes valid responses have existed in a much larger set of resources or knowledge bases and the system's primary goal is to find them. In contrast, one other popular formulation of conversation is "generation," in which the system takes the chat history as input and the system's goal is to generate good follow-up responses based on that input. The differences between these two formulations can be subtle, and in practice, systems can mix the two. Automated question-answering systems often use a retrieval-based architecture, which retrieves short answer candidates (known as "nuggets") from a large text corpus and then ranks them based on usefulness or other measurements. Machine translation systems often use a generation-based architecture that tries to automatically learn the mappings between source and target languages.

In our dissertation work, we took an approach that is more like retrieval-based methods. At a high level, Evorus believed that many questions could be answered by existing chatbots or prior answers. This belief was motivated by the observation that although chatbots cannot hold long conversations, most of them are good at one or two things. For example, Yelp Bot can answer restaurant questions, Weather Bot can answer weather questions, and general chitchat bots such as Cleverbot can hold short social conversations to a certain extent. This assumption breaks down a conversation into smaller, easier tasks that each can be reasonably solved by today's chatbots. Focusing on selection also enabled Evorus to gradually learn to include more and more response generators. The drawbacks of this approach are that Evorus can only select from responses that modern AI systems can generate, and it is harder to push the boundaries of automatic text generation technologies. Furthermore, as mentioned in Related Work (Section 2.1), modular pipeline systems tend to propagate errors more easily than end-to-end generative systems, while end-to-end training usually requires a large amount of data.

In addition to conversational assistants, the architecture we proposed might work better for applications in which generating a large set of candidates is easy, *e.g.,* speech recognition or question answering. As an example, the CRQA system also used a generate-and-select model to have human workers select good answers from a set of machine-generated candidates [130, 131].

9.4.2 Language Understanding

Evorus used a general language-understanding component that runs in the background to extract context (such as location, time, names, and more) from the dialog and make the extracted information available to all the chatbots. While we believe that this is a good approximation of context, current technologies are relatively simple and cannot express complex semantics. To scale up, a more sophisticated semantic parser and a richer knowledge base will likely be needed to increase the system's ability to express context.

Furthermore, the extracted semantic information was not used by Evorus's learning framework to select chatbots. For instance, if a user mentioned "beer" and "steak," if the semantic parser recognized these items as "food," the Yelp Bot would likely be more useful than a weather bot or movie bot. This level of understanding requires not only good semantic parsing but also good topic similarity modeling. Evorus used Glove [117] word vector to model topic similarity, which did not include any knowledge linking entities and domains, *e.g.,* that food is relevant to the topic of "restaurant."

In the future, we might use more advanced technologies, such as entity linking or more advanced semantic parsers, to extract context from a conversation and use context to better learn to retrieve chatbots. As for workers, currently they can take notes to keep important facts on the interface, but in this dissertation work, we did not attempt to automatically generate notes or provide extracted context to workers. There is still much more we could do to improve the language-understanding capability of our current system.

To summarize, Chorus and Evorus came with several limitations and concerns. Some of these limitations were caused by the restrictions of technologies or system designs; some were more fundamental, such as ambiguity in natural languages, or the inevitable communication cost when

delegating tasks to an agent. Our systems have also raised concerns about user privacy and verbal abuse. Furthermore, we also learned some vulnerabilities of the systems' current mechanisms.

All of these limitations, concerns, vulnerabilities have introduced new research questions and challenges. We are aware of potential solutions to address or resolve these concerns. We would also like to encourage researchers and developers to look into these challenges, and come up with effective solutions that can make crowd-AI systems more secure, effective, and easy to use.

Chapter 10

Conclusions and Future Work

In this dissertation, we showed how a deployed crowd-powered conversational assistant could be automated over time by integrating new chatbots, reusing prior crowd answers, and gradually reducing the crowd's role in choosing high-quality responses. We began with a lab prototype system, Chorus, which we reimplemented and deployed as a Google Hangouts chatbot. To recruit workers quickly and economically in Chorus, we invented a new recruiting method: the Ignition model. Since its launch, over 420 users have talked to Chorus across more than 2,200 conversations. We also introduced Evorus, a framework on top of Chorus. Evorus allowed external chatterbots and task-oriented dialog systems to be added to Chorus to automate conversations, allow Chorus to reuse prior answers proposed by crowd workers for future similar questions, and implement an automatic voting bot to help workers select high-quality responses. To make the most of Evorus, we also created Guardian, a crowd-powered framework that converts Web APIs into chatbots, letting us quickly and efficiently create many chatbots intended for various domains. One key component of Guardian is the Dialog ESP Game, which uses multiple crowd workers to collectively extract necessary information from a running dialog within a few seconds so the chatbot does not need to prepare a pretrained automatic extractor before being deployed. Finally, to augment the ability of Chorus to "do" things for users rather than only "saying" things, we created InstructableCrowd, a crowd-powered system that generates IFTTT-style trigger-action rules based on the user's needs as communicated through a conversation. A two-year deployment showed the effectiveness of our approach and also demonstrated that such a system can be used as a conversational personal assistant to help people in their everyday lives.

10.1 Future Work

This dissertation focused on crowd-powered conversational systems, which we look forward to studying further with greater depth and at a larger scale. In the future, we will also explore other crowd-AI systems.

Deploy Chorus as an Open Research Platform In the future, Chorus will continue to serve as an open research platform with its growing user base and active crowd workers to operate

it, allowing the system to conduct live experiments on various technologies. We can use the deployed Chorus platform to explore the following possibilities:

- **Chorus Dialog System Challenge:** Testing dialog systems with a live system setup is difficult, so most have been evaluated offline. With Chorus, hosting a shared task that invites different dialog systems to contribute responses and compete with each other (*e.g.*, by response acceptance rate) becomes possible, breaking new ground and providing fresh perspective to the community.
- **Coordinating 1,000+ Chatbots:** Our vision of the future of Chorus is a system that fuses 1,000+ response contributors. Developing a sophisticated and robust crowd-AI framework that supports this level of scaling will certainly be exciting.
- **Open Chorus API:** It takes a tremendous amount of effort to build a real-time crowd-powered system. The *Open Chorus API* will make such technologies easier for other researchers to use. Making Chorus available to the community will also encourage more researchers to explore various types of crowd-powered systems, benefiting the research community as a whole.

Crowd-Powered Systems on Smart Devices We imagine a future where smart devices can perform complex, custom tasks. To do so, human-in-the-loop architectures are essential. Interactive crowd-powered systems can not only be implemented in smartphones or as web applications, but also be integrated into smart homes, smart watches [142], voice-enabled devices, or even smart cars. These uses will open up many exciting possibilities. They will also present new challenges, such as the extremely short response time that users expect when talking to a voice-enabled device, applying the system to different hardware architectures, and dealing with user privacy concerns.

Future Crowd-AI Systems One theme that emerged from this dissertation is using real-time collaboration between the crowd and AI to create better systems. In the future, we will apply the knowledge that we obtained from these projects to develop crowd-AI systems especially for tasks that are useful to people but still challenging for AI alone. One example would be visual storytelling [70], which was just introduced to the AI community two years ago. Another would be identifying subtle emotions in messages, which can be helpful under many circumstances but hard to automate [161]. Recognizing deaf speech could be another good application, as today's speech recognition systems perform poorly here but humans can generally understand it [15].

At a higher level, crowd-powered approaches are robust, intelligent, and do not require training data, but they are slow and costly. In contrast, automated approaches are fast, scalable, and more affordable, but they require large amounts of training data and have many more capability limitations. Most projects in this dissertation attempted to answer this critical question: How do we combine crowd-powered and automated components wisely to leverage the advantages of each while minimizing the disadvantages?

Our take was slightly different from many prior attempts, which added pieces of human work into a larger automated system architecture. Instead, we introduced pieces of automation into human-powered systems: Evorus used chatbots and vote bots to help workers more quickly and

easily hold conversations; Guardian used the crowd to understand language and talk to users and employed Web APIs in the back end to perform queries and obtain information; the Dialog ESP Game used the crowd to extract information and pass it to downstream automated components; and InstructableCrowd used crowd workers to create IF-THEN rules that computers can then execute. One lesson of our work is the effectiveness of the "top-down" research approach. We started with a working, deployed system, learned to improve it, and created a framework that allows the system to automate itself over time. One core advantage of this top-down approach is that users can start using the system from day one, providing realistic data to guide future automation. Furthermore, thanks to the oversight of crowd workers, such human-in-the-loop frameworks allow automated components to make more mistakes, opening more spaces for algorithms to try different strategies.

We believe that – even in this era of AI, machine learning, and deep learning – combining human intelligence with automated approaches will result in systems that are more robust, usable, and scalable, thereby creating a better world.

Bibliography

[1] James F Allen, Bradford W Miller, Eric K Ringger, and Teresa Sikorski. A robust system for natural spoken dialogue. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 62–70. Association for Computational Linguistics, 1996. 2.1.1

[2] Amazon. Meet alexa, 2017. URL `https://www.amazon.com/meet-alexa/b?ie=UTF8&node=16067214011`. 5

[3] Yoav Artzi and Luke Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62, 2013. 8.1.2

[4] Daniel Avrahami, Susan R. Fussell, and Scott E. Hudson. Im waiting: Timing and responsiveness in semi-synchronous communication. In *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work*, CSCW '08, pages 285–294, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-007-4. doi: 10.1145/1460563.1460610. URL `http://doi.acm.org/10.1145/1460563.1460610`. 2.3

[5] Amos Azaria, Ariella Richardson, and Sarit Kraus. An agent for deception detection in discussion based environments. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 218–227. ACM, 2015. 2.3.2

[6] Amos Azaria, Jayant Krishnamurthy, and Tom M Mitchell. Instructable intelligent personal agent. In *Proc. AAAI '16*, AAAI '16, 2016. 8.2.2

[7] 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. 1.4, 5

[8] Naomi S Baron. Discourse structures in instant messaging: The case of utterance breaks. *Language Internet*, 7(4):1–32, 2010. 2.3, 3.5, 7.3.3

[9] 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`. 2.3.2, 3, 4

[10] 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, 2.3.2, 3.1.2, 2, 4.2.2, 6.1.2, 7.3.1

[11] 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

[12] 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. 2.2.1

[13] 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. 8.1.1

[14] 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. 2.2.1, 1, 2.3.1, 2.3.2, 2.3.3, 1, 4, 4.5, 5, 7.3.1

[15] Jeffrey P. Bigham, Raja Kushalnagar, Ting-Hao Kenneth Huang, Juan Pablo Flores, and Saiph Savage. On how deaf people might use speech to control devices. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '17, pages 383–384, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4926-0. doi: 10.1145/3132525.3134821. URL http://doi.acm.org/10.1145/3132525.3134821. 10.1

[16] 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. 2.1.1

[17] 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

[18] 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. 2.1.1, 7

[19] 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. 8.1.1

[20] 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. 8.1.1

[21] Jeppe Bronsted, Klaus Marius Hansen, and Mads Ingstrup. Service composition issues in pervasive computing. *IEEE Pervasive Computing*, 9(1):62–70, 2010. 8.1.1

[22] 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. 8.1.1

[23] Rollo Carpenter. Cleverbot, 2006. URL https://www.cleverbot.com/. [Online; accessed 08-March-2017]. 4

[24] Joel Chan, Steven Dang, and Steven P Dow. Improving crowd innovation with expert facilitation. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, pages 1223–1235. ACM, 2016. 2.3.2, 9.1.1

[25] Joseph Chee Chang, Aniket Kittur, and Nathan Hahn. Alloy: Clustering with crowds and computation. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 3180–3191, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858411. URL http://doi.acm.org/10.1145/2858036.2858411. 2.2.2, 8.7.1

[26] Joseph Chee Chang, Saleema Amershi, and Ece Kamar. Revolt: Collaborative crowdsourcing for labeling machine learning datasets. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 2334–2346. ACM, 2017. 2.3.2

[27] Shobhit Chaurasia and Raymond J Mooney. Dialog for language to code. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, volume 2, pages 175–180, 2017. 8.1.2

[28] 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. 9.2.2

[29] Yun-Nung Chen and Jianfeng Gao. Open-domain neural dialogue systems. *Proceedings*

of the IJCNLP 2017, Tutorial Abstracts, pages 6–10, 2017. 2.1

[30] 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. 6.4.2

[31] Yun-Nung Chen, Dilek Hakkani-Tür, and Gokhan Tur. Deriving local relational surface forms from dependency-based entity embeddings for unsupervised spoken language understanding. *Proceedings of SLT*, 2014. 7

[32] Yun-Nung Chen, Asli Celikyilmaz, and Dilek Hakkani-Tür. Deep learning for dialogue systems. *Proceedings of ACL 2017, Tutorial Abstracts*, pages 8–14, 2017. 2.1, 1

[33] Justin Cheng and Michael S Bernstein. Flock: Hybrid crowd-machine learning classifiers. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 600–611. ACM, 2015. 2.2.2

[34] Lydia Chilton. Seaweed: A web application for designing economic games. Master's thesis, MIT, 2009. 2.3.2

[35] Josh Constine. Amazon rejects ai2's alexa skill voice-search engine. will it build one?, May 2017. URL `https://techcrunch.com/2017/05/31/amazon-skill-search-engine/`. 5

[36] 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. 7.2.2

[37] 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. 8.1.1

[38] Peng Dai, Jeffrey M Rzeszotarski, Praveen Paritosh, and Ed H Chi. And now for something completely different: Improving crowdsourcing workflows with micro-diversions. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 628–638. ACM, 2015. 4.5

[39] 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. 8, 8.1.1

[40] Barb Darrow. Eavesdropping google home mini units are igniting privacy concerns, 2017. URL `http://fortune.com/2017/10/11/google-home-mini-data-privacy/`. 9.3.1

[41] 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`. 8.1.1

[42] 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. 8.1.1

[43] 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. 3.4

[44] Li Dong and Mirella Lapata. Language to logical form with neural attention. *arXiv preprint arXiv:1601.01280*, 2016. 8.1.2

[45] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear - a library for large linear classification, 2008. URL `http://www.csie.ntu.edu.tw/~cjlin/liblinear/`. The Weka classifier works with version 1.33 of LIBLINEAR. 5.5.2

[46] Hao Fang, Hao Cheng, Maarten Sap, Elizabeth Clark, Ari Holtzman, Yejin Choi, Noah A Smith, and Mari Ostendorf. Sounding board: A user-centric and content-driven social chatbot. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 96–100, 2018. 2.1.2

[47] Umer Farooq, Jonathan Grudin, Ben Shneiderman, Pattie Maes, and Xiangshi Ren. Human computer integration versus powerful tools. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '17, pages 1277–1282, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4656-6. doi: 10.1145/3027063.3051137. URL `http://doi.acm.org/10.1145/3027063.3051137`. 1.3, 9.2.2

[48] Emilio Ferrara, Onur Varol, Clayton Davis, Filippo Menczer, and Alessandro Flammini. The rise of social bots. *Communications of the ACM*, 59(7):96–104, 2016. 2.1.2

[49] Michael J. Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. Crowddb: Answering queries with crowdsourcing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 61–72, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989331. URL `http://doi.acm.org/10.1145/1989323.1989331`. 2.2.2

[50] 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`. 8.1.1

[51] Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. Personalization of context-dependent applications through trigger-action rules. *ACM Trans. Comput.-Hum. Interact.*, 24(2):14:1–14:33, April 2017. ISSN 1073-0516. doi: 10.1145/3057861. URL `http://doi.acm.org/10.1145/3057861`. 8.1.1

[52] Daniel Haas, Jiannan Wang, Eugene Wu, and Michael J. Franklin. Clamshell: Speeding up crowds for low-latency data labeling. *Proc. VLDB Endow.*, 9(4):372–383, December 2015. ISSN 2150-8097. doi: 10.14778/2856318.2856331. URL `http://dx.doi.`

org/10.14778/2856318.2856331. 2.3.1

[53] 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. 2.2.1, 8.7.1

[54] Dilek Hakkani-Tür, Gökhan Tür, Asli Celikyilmaz, Yun-Nung Chen, Jianfeng Gao, Li Deng, and Ye-Yi Wang. Multi-domain joint semantic frame parsing using bi-directional rnn-lstm. In *Interspeech*, pages 715–719, 2016. 2.1.1

[55] 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. 8.1.1

[56] Bo Han and Timothy Baldwin. Lexical normalisation of short text messages: Makn sens a #twitter. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, HLT '11, pages 368–378, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics. ISBN 978-1-932432-87-9. URL http://dl.acm.org/citation.cfm?id=2002472.2002520. 5.3.1

[57] Hangoutsbot. hangoutsbot/hangoutsbot, Apr 2017. URL https://github.com/hangoutsbot/hangoutsbot. 5.1

[58] Eric N Hanson and Jennifer Widom. An overview of production rules in database systems. *The Knowledge Engineering Review*, 8(02):121–143, 1993. 8.1.1

[59] Yulan He and Steve Young. A data-driven spoken language understanding system. In *ASRU'03*, pages 583–588. IEEE, 2003. 7.2.2

[60] 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. 7.2.4

[61] 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

[62] 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. 6.4.2

[63] Matthew Henderson, Blaise Thomson, and Steve Young. Deep neural network approach for the dialog state tracking challenge. In *Proceedings of the SIGDIAL 2013 Conference*, pages 467–471, 2013. 2.1.1

[64] 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, Stroudsburg, 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. 7.2.1

[65] Yen-Chia Hsu, Paul Dille, Jennifer Cross, Beatrice Dias, Randy Sargent, and Illah Nourbakhsh. Community-empowered air quality monitoring system. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 1607–1619, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4655-9. doi: 10.1145/3025453.3025853. URL http://doi.acm.org/10.1145/3025453.3025853. 2.3.2

[66] 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. 8, 8.1.1, 8.3.1

[67] Ting-Hao K. Huang and Jeffrey P. Bigham. A 10-month-long deployment study of on-demand recruiting for low-latency crowdsourcing. In *In Proceedings of The fifth AAAI Conference on Human Computation and Crowdsourcing (HCOMP 2017)*. AAAI, AAAI, oct 2017. 1.5.1, 2.3.1, 2.3.3, 5.1

[68] 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. 1.5.3, 6

[69] 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. 1.5.3, 2.1.1, 2.3.2, 3, 3, 6, 7.4

[70] 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. 10.1

[71] Ting-Hao K. Huang, Joseph Chee Chang, Saiganesh Swaminathan, and Jeffrey Bigham. Evorus: A crowd-powered conversational assistant that automates itself over time. In *Poster track of the 20th ACM Symposium on User Interface Software and Technology (UIST Poster 2017)*, oct 2017. 1.5.2, 2.3.2, 2.3.2

[72] Ting-Hao K. Huang, Yun-Nung Chen, and Jeffrey P. Bigham. Real-time on-demand crowd-powered entity extraction. In *Proceedings of the 5th Edition Of The Collective Intelligence Conference (CI 2017)*, New York University, NY, USA, jun 2017. Oral presentation. 1.5.3, 2.1.1, 2.3.2, 2.3.2

[73] 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. 1.5.4, 3

[74] 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. 1.5, 1.5.1, 2.3.3, 3, 4, 4.1.1, 4.2, 4.4, 5, 5.1, 5.1, 5.4.1, 5.5.1

[75] Ting-Hao (Kenneth) Huang, Joseph Chee Chang, and Jeffrey P. Bigham. Evorus: A crowd-powered conversational assistant built to automate itself over time. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 295:1–295:13, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5620-6. doi: 10.1145/3173574.3173869. URL `http://doi.acm.org/10.1145/3173574.3173869`. 1.5.2, 2.3.2, 2.3.2

[76] 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`. 8

[77] 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. 7.2.4

[78] Panagiotis G. Ipeirotis, Foster Provost, and Jing Wang. Quality management on amazon mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, 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`. 3.4

[79] 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. 2.3, 3.5, 7.3, 7.3.3

[80] 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. 8.1.1

[81] Ece Kamar and Lydia Manikonda. Complementing the execution of ai systems with human computation. In *AAAI Workshop on Crowdsourcing, Deep Learning and Artificial Intelligence Agents 2017*. AAAI, 2017. 2.2.2

[82] Ece Kamar, Severin Hacker, and Eric Horvitz. Combining human and machine intelligence in large-scale crowdsourcing. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 467–474. International Foundation for Autonomous Agents and Multiagent Systems, 2012. 2.2.2

[83] 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. 8.1.1

[84] Ranjay A. Krishna, Kenji Hata, Stephanie Chen, Joshua Kravitz, David A. Shamma, Li Fei-Fei, and Michael S. Bernstein. Embracing error to enable rapid crowdsourcing.

In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 3167–3179, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858115. URL http://doi.acm.org/10.1145/2858036.2858115. 2.3.1

[85] G. Laput, W. S. Lasecki, J. Wiese, R. Xiao, J. P. Bigham, and C. Harrison. Zensors: Adaptive, rapidly deployable, human-intelligent sensor feeds. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '15, New York, NY, USA, 2015. ACM. URL http://www.cs.cmu.edu/~jbigham/pubs/pdfs/2015/zensors.pdf. 2.2.2, 2.3.2

[86] 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. 3

[87] 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. 2.3.2, 2.3.2, 4

[88] Walter S. Lasecki, Chris D. Miller, Adam Sadilek, Andrew Abumoussa, Donato Borrello, Raja Kushalnagar, and Jeffrey P. Bigham. Real-time captioning by groups of non-experts. In *In Proceedings of the Symposium on User Interface Software and Technology (UIST 2012)*, pages 23–34, 2012. URL http://hci.cs.rochester.edu/pubs/pdfs/scribe.pdf. 2.3.2

[89] Walter S. Lasecki, Ece Kamar, and Dan Bohus. Conversations in the crowd: Collecting data for task-oriented dialog learning. In *HCOMP*, 2013. 3, 7

[90] 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. 1.5, 1.5.1, 2.3.2, 3, 3, 3.1, 3.1.1, 3.2, 5.5.1, 6.3.2, 7.3.1, 8.7.1

[91] 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. 3

[92] Walter S Lasecki, Christopher Homan, and Jeffrey P Bigham. Architecting real-time crowd-powered systems. *Human Computation*, 1(1), 2014. 2.3.2

[93] 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. 3.4

[94] 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. 8.7.5

[95] Walter S. Lasecki, Jeffrey M. Rzeszotarski, Adam Marcus, and Jeffrey P. Bigham. The effects of sequence and delay on crowd work. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1375–1378, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3145-6. doi: 10.1145/2702123.2702594. URL http://doi.acm.org/10.1145/2702123.2702594. 4.5

[96] Thomas D LaToza and Andre van der Hoek. Crowdsourcing in software engineering: Models, motivations, and challenges. *IEEE Software*, 33(1):74–80, 2016. 8.1.1

[97] 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. 8.1.1

[98] Edith Law and Luis von Ahn. Human computation. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 5(3):1–121, 2011. 2.2.2

[99] 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. 6.3.2

[100] 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. 8.1.1

[101] Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. A diversity-promoting objective function for neural conversation models. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 110–119, 2016. 2.1.2

[102] Jiwei Li, Will Monroe, Alan Ritter, Dan Jurafsky, Michel Galley, and Jianfeng Gao. Deep reinforcement learning for dialogue generation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1192–1202. Association for Computational Linguistics, 2016. doi: 10.18653/v1/D16-1127. URL http://www.aclweb.org/anthology/D16-1127. 2.1.2

[103] Xiujun Li, Yun-Nung Chen, Lihong Li, Jianfeng Gao, and Asli Celikyilmaz. End-to-end task-completion neural dialogue systems. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 733–743, 2017. 2.1.1

[104] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. *End-user development: An emerging paradigm*. Springer, 2006. 8.1.1

[105] Chang Liu, Xinyun Chen, Eui Chul Shin, Mingcheng Chen, and Dawn Song. Latent attention for if-then program synthesis. In *Advances in Neural Information Processing Systems*, pages 4574–4582, 2016. 8.1.2

[106] Leigh Anne Liu, Chei Hwee Chua, and Günter K Stahl. Quality of communication experience: definition, measurement, and implications for intercultural negotiations. *Journal of Applied Psychology*, 95(3):469, 2010. 5.6.1

[107] Yi Luan, Shinji Watanabe, and Bret Harsham. Efficient learning for spoken language understanding tasks with word embedding based pre-training. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015. 2.1.1

[108] Yi Luan, Chris Brockett, Bill Dolan, Jianfeng Gao, and Michel Galley. Multi-task learning for speaker-role adaptation in neural conversation models. *arXiv preprint arXiv:1710.07388*, 2017. 2.1.1

[109] 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`. 8.1.1

[110] 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. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23(3):530–539, 2015. 2.1.1, 7, 7.2.2, 8.8

[111] Chandrakana Nandi and Michael D. Ernst. Automatic trigger generation for rule-based smart homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 97–102, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4574-3. doi: 10.1145/2993600.2993601. URL `http://doi.acm.org/10.1145/2993600.2993601`. 8.7.3

[112] Michael Nebeling, Alexandra To, Anhong Guo, Adrian A. de Freitas, Jaime Teevan, Steven P. Dow, and Jeffrey P. Bigham. Wearwrite: Crowd-assisted writing from smartwatches. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 3834–3846, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858169. URL `http://doi.acm.org/10.1145/2858036.2858169`. 8.7.1

[113] Cable News Network, 2017. URL `http://transcripts.cnn.com/TRANSCRIPTS/`. 3

[114] 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`. 5.2.1

[115] Chikashi Nobata, Joel Tetreault, Achint Thomas, Yashar Mehdad, and Yi Chang. Abusive language detection in online user content. In *Proceedings of the 25th international con-*

ference on world wide web, pages 145–153. International World Wide Web Conferences Steering Committee, 2016. 9.3.4

[116] Alice H Oh and Alexander I Rudnicky. Stochastic language generation for spoken dialogue systems. In *Proceedings of the 2000 ANLP/NAACL Workshop on Conversational systems-Volume 3*, pages 27–32. Association for Computational Linguistics, 2000. 2.1.1

[117] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL `http://www.aclweb.org/anthology/D14-1162`. 5.3.2, 5.4.2, 9.4.2

[118] David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. Naturaljava: a natural language interface for programming in java. In *Proceedings of the 5th international conference on Intelligent user interfaces*, pages 207–211. ACM, 2000. 8.1.2

[119] Alexander J Quinn and Benjamin B Bederson. Human computation: a survey and taxonomy of a growing field. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 1403–1412. ACM, 2011. 2.2.2

[120] Chris Quirk, Raymond J Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *ACL (1)*, pages 878–888, 2015. 8.1.2

[121] Antoine Raux and Maxine Eskenazi. Optimizing endpointing thresholds using dialogue features in a spoken dialogue system. In *Proceedings of the 9th SIGdial Workshop on Discourse and Dialogue*, pages 1–10. Association for Computational Linguistics, 2008. 5.5.2

[122] Suman Ravuri and Andreas Stolcke. Recurrent neural network and lstm models for lexical utterance classification. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015. 2.1.1

[123] Christian Raymond and Giuseppe Riccardi. Generative and discriminative algorithms for spoken language understanding. In *INTERSPEECH*, pages 1605–1608, 2007. 7, 7.2.2, 8.8

[124] 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. 7.2.4

[125] Daniela Retelny, Sébastien Robaszkiewicz, Alexandra To, Walter S Lasecki, Jay Patel, Negar Rahmati, Tulsee Doshi, Melissa Valentine, and Michael S Bernstein. Expert crowdsourcing with flash teams. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 75–85. ACM, 2014. 2.2.2

[126] Erica Sadun and Steve Sande. *Talking to Siri: Mastering the Language of Apple's Intelligent Assistant*. Que Publishing, 2014. 1.1, 5

[127] Elliot Salisbury, Sebastian Stein, and Sarvapali Ramchurn. Crowdar: augmenting live video with a real-time crowd. In *Third AAAI Conference on Human Computation and Crowdsourcing*, 2015. 2.3.2

[128] 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. 2.3.2

[129] Akash Das Sarma, Ayush Jain, Arnab Nandi, Aditya Parameswaran, and Jennifer Widom. Surpassing humans and computers with jellybean: Crowd-vision-hybrid counting algorithms. In *Third AAAI Conference on Human Computation and Crowdsourcing*, 2015. 2.2.2

[130] 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. 2.2.1, 2.3.2, 9.4.1

[131] 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. 2.3.2, 3, 9.4.1

[132] Konrad Scheffler and Steve Young. Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning. In *Proceedings of the Second International Conference on Human Language Technology Research*, HLT '02, pages 12–19, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1289189.1289246`. 5.3.1

[133] Amazon Developer Services. The alexa prize, 2016. URL `https://developer.amazon.com/alexaprize`. 2.1.2

[134] Ben Shneiderman. *Designing the user interface: strategies for effective human-computer interaction*. Pearson Education India, 2010. 9.2.2

[135] Ben Shneiderman and Pattie Maes. Direct manipulation vs. interface agents. *interactions*, 4(6):42–61, November 1997. ISSN 1072-5520. doi: 10.1145/267505.267514. URL `http://doi.acm.org/10.1145/267505.267514`. 9.2.2

[136] Gabriel Skantze. Exploring human error handling strategies: Implications for spoken dialogue systems. In *ISCA Tutorial and Research Workshop on Error Handling in Spoken Dialogue Systems*, 2003. 9.1.2

[137] Jackie Snow. Amazon is trying to make alexa more chatty-but it's very, very difficult, Feb 2018. URL `https://www.technologyreview.com/the-download/610381/amazon-is-trying-to-make-alexa-more-chatty-but-its-very-very-difficult`. 1, 1.4

[138] Sara Owsley Sood, Judd Antin, and Elizabeth F Churchill. Using crowdsourcing to improve profanity detection. In *AAAI Spring Symposium: Wisdom of the Crowd*, volume 12, page 06, 2012. 9.3.4

[139] Alessandro Sordoni, Michel Galley, Michael Auli, Chris Brockett, Yangfeng Ji, Margaret Mitchell, Jian-Yun Nie, Jianfeng Gao, and Bill Dolan. A neural network approach to context-sensitive generation of conversational responses. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics:*

Human Language Technologies, pages 196–205, 2015. 2.1.1

[140] Ben Stegner. 7 ways alexa and amazon echo pose a privacy risk, Jan 2018. URL https://www.makeuseof.com/tag/alexa-amazon-echo-privacy-risk/. 9.3.1

[141] Ming Sun. *Adapting Spoken Dialog Systems Towards Domains and Users*. PhD thesis, YAHOO! Research, 2016. 1.4

[142] Saiganesh Swaminathan, Raymond Fok, Fanglin Chen, Ting-Hao (Kenneth) Huang, Irene Lin, Rohan Jadvani, Walter S. Lasecki, and Jeffrey P. Bigham. Wearmail: On-the-go access to information in your email with a privacy-preserving human computation workflow. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 807–815, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4981-9. doi: 10.1145/3126594.3126603. URL http://doi.acm.org/10.1145/3126594.3126603. 8.7.5, 10.1

[143] 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. 3.3.2

[144] 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. 6.3.2

[145] Long Tran-Thanh, Sebastian Stein, Alex Rogers, and Nicholas R Jennings. Efficient crowdsourcing of unknown experts using bounded multi-armed bandits. *Artificial Intelligence*, 214:89–111, 2014. 5.3.1

[146] 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. 8.1.1

[147] 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. 7.2.2

[148] Weather Underground. A weather api designed for developers, 2017. URL https://www.wunderground.com/weather/api/. 2

[149] 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. 8, 8.1.1, 8.4.1

[150] Giridhari Venkatadri, Athanasios Andreou, Yabing Liu, Alan Mislove, Krishna P Gummadi, Patrick Loiseau, and Oana Goga. Privacy risks with facebook's pii-based targeting: Auditing a data broker's advertising interface. In *IEEE Symposium on Security and Privacy (SP)*, pages 221–239, 2018. 9.3.1

[151] Oriol Vinyals and Quoc V. Le. A neural conversational model. In *ICML Deep Learning Workshop*, 2015. URL http://arxiv.org/pdf/1506.05869v3.pdf. 2.1.1

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

[153] 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. 2.3, 2.3.2, 6.1.2, 7, 7, 7.1, 8.2.5

[154] Luis von Ahn and Laura Dabbish. Designing games with a purpose. *Commun. ACM*, 51(8):58–67, August 2008. ISSN 0001-0782. doi: 10.1145/1378704.1378719. URL http://doi.acm.org/10.1145/1378704.1378719. 8.2.5

[155] 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. 3.4, 3.4.3

[156] Marilyn Walker and Rebecca Passonneau. Date: a dialogue act tagging scheme for evaluation of spoken dialogue systems. In *Proceedings of the first international conference on Human language technology research*, pages 1–8. Association for Computational Linguistics, 2001. 8.6.2

[157] Marilyn A. Walker, Diane J. Litman, Candace A. Kamm, and Alicia Abella. Paradise: a framework for evaluating spoken dialogue agents. In *Proceedings of the Association for Computational Linguistics*, ACL '98, pages 271–280, Stroudsburg, PA, USA, 1997. Association for Computational Linguistics. doi: 10.3115/976909.979652. URL http://dx.doi.org/10.3115/976909.979652. 5.6.1

[158] 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. 1.4

[159] Richard S Wallace. The anatomy of alice. In *Parsing the Turing Test*, pages 181–210. Springer, 2009. 2.1.2

[160] 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

[161] Shih-Ming Wang, Chun-Hui Scott Lee, Yu-Chun Lo, Ting-Hao Huang, and Lun-Wei Ku. Sensing emotions in text messages: An application and deployment study of emotionpush. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: System Demonstrations*, pages 141–145, 2016. 10.1

[162] 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

[163] Xu Wang, Miaomiao Wen, and Carolyn Rose. Contrasting explicit and implicit support for

transactive exchange in team oriented project based learning. In *Proc. 12th International Conference on Computer Supported Collaborative Learning (CSCL) 2017*. Philadelphia, PA: International Society of the Learning Sciences., 2017. 2.3.2

[164] Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966. 2.1.2

[165] Tsung-Hsien Wen, David Vandyke, Nikola Mrkšić, Milica Gasic, Lina M Rojas Barahona, Pei-Hao Su, Stefan Ultes, and Steve Young. A network-based end-to-end trainable task-oriented dialogue system. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, volume 1, pages 438–449, 2017. 2.1.1

[166] Wikipedia. Cleverbot — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Cleverbot&oldid=771836990`, 2017. [Online; accessed 02-April-2017]. 4

[167] 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. 2.1.1, 7

[168] Jason D Williams and Steve Young. Partially observable markov decision processes for spoken dialog systems. *Computer Speech & Language*, 21(2):393–422, 2007. 2.1.1, 5

[169] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016. 5.5.3

[170] Puyang Xu and Ruhi Sarikaya. Convolutional neural network based triangular crf for joint intent detection and slot filling. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 78–83. IEEE, 2013. 2.1.1

[171] 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. 2.1.1, 7, 7.2.2

[172] Tingxin Yan, Vikas Kumar, and Deepak Ganesan. Crowdsearch: Exploiting crowds for accurate real-time image search on mobile phones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 77–90, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-985-5. doi: 10.1145/1814433.1814443. URL `http://doi.acm.org/10.1145/1814433.1814443`. 2.3.1

[173] Kaisheng Yao, Geoffrey Zweig, Mei-Yuh Hwang, Yangyang Shi, and Dong Yu. Recurrent neural networks for language understanding. In *Interspeech*, pages 2524–2528, 2013. 2.1.1

[174] 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`. 8.1.1

[175] Yelp. Yelp api documentation, 2017. URL https://www.yelp.com/developers/documentation/v2/overview. 1

[176] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *The 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, Vancouver, Canada, July 2017. URL https://arxiv.org/abs/1704.01696. 8.1.2

[177] Steve Young. Using pomdps for dialog management. In *SLT*, pages 8–13, 2006. 7

[178] Steve Young. Statistical spoken dialogue systems and the challenges for machine learning. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 577–577. ACM, 2017. 2.1

[179] Tiancheng Zhao and Maxine Eskenazi. Towards end-to-end learning for dialog state tracking and management using deep reinforcement learning. In *17th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, page 1, 2016. 2.1.1

[180] Tiancheng Zhao, Kyusong Lee, and Maxine Eskenazi. Dialport: Connecting the spoken dialog research community to real user data. In *Spoken Language Technology Workshop (SLT), 2016 IEEE*, pages 83–90. IEEE, 2016. 1.4

[181] Youyou Zhou. An oregon family's encounter with amazon alexa exposes the privacy problem of smart home devices, May 2018. URL https://qz.com/1288743/amazon-alexa-echo-spying-on-users-raises-a-data-privacy-problem/. 9.3.1