Spring 1-1-2016

The Crowdrouter Framework: Addressing Issues of Software Design in Support of Crowdwork

Mario Bernardo Barrenechea
University of Colorado Boulder, mbarrenecheajr@gmail.com

Follow this and additional works at: https://scholar.colorado.edu/csci_gradetds
Part of the Databases and Information Systems Commons, and the Software Engineering Commons

Recommended Citation
https://scholar.colorado.edu/csci_gradetds/116

This Dissertation is brought to you for free and open access by Computer Science at CU Scholar. It has been accepted for inclusion in Computer Science Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.
The Crowdrouter Framework: Addressing Issues of Software Design in Support of Crowdwork

by

Mario Bernardo Barrenechea

B.S., University of Massachusetts Amherst, 2011
M.S., University of Colorado Boulder, 2014

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirement for the degree of
Doctor of Philosophy
Department of Computer Science

2016
This thesis entitled:
The Crowdrouter Framework: Addressing Issues of Software Design in Support of Crowdwork
written by Mario Bernardo Barrenechea
has been approved for the Department of Computer Science

(Kenneth M. Anderson)

(Judy Stafford)

(Leysia Palen)

(Tom Yeh)

Date______________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

IRB protocol # 16-0069
Barrenechea, Mario Bernardo (Ph.D., Computer Science (Department of Computer Science))

The Crowdrouter Framework: Addressing Issues of Software Design in Support of Crowdwork
Thesis directed by Associate Professor Kenneth M. Anderson

Crowdwork is a recent technique for mobilizing a large group of people to perform meaningful tasks for society on-line and to provide them with the tools they need to communicate, coordinate, and make progress. It is, however, an unexplored frontier in software development contexts in terms of having frameworks and tools that ease the development of crowdwork systems. This dissertation explores crowdwork systems in depth, surfacing the challenges that surround the creation of these systems via concerns raised by developers who have used and created crowdwork systems in the past. We use these concerns to identify a set of requirements for software frameworks that directly support the creation of crowdwork systems and then present a new framework—Crowdrouter—that meets these requirements. We evaluate the Crowdrouter framework by examining its versatility, i.e., with respect to its ability to support a wide variety of crowdwork-related workflows and tasks.
ACKNOWLEDGMENTS

I would like to thank a multitude of people without whom this dissertation work, as well as all of the work I have conducted during my time in Colorado, could not possibly have been carried out. First, I would like to thank my advisor Ken Anderson, whose guidance has been critical to my growth as a young scholar looking into the research world with curious and critical eyes, but reinforced by the humorous pragmatism with which his advice was given. From starting with pets in disaster, to data analytics, and now to crowdwork, Ken has been instrumental in shaping my thinking, writing, and confidence in conducting and performing scholarly work in computer science. Thank you, Ken, for your guidance throughout this process.

My thanks also go out to my “second” advisor—Leysia Palen—who, with the innate ability to shape computing to think not of people as users but as humans, has also supported my growth during the graduate school program, from IRB protocol writing to HCI and social computing classes. It has forever changed my perspective on the responsibility that computing must look ‘up the stack’ and include the social constructions of humans’ role with technology. Thank you, Leysia, for all that you do to inspire and influence others.

My thanks and appreciations also go out to the other graduate students at the Project EPIC lab, in both the ‘Ken’ group and ‘Leysia’ group. These colleagues are some of the world’s finest—Mazin, Ahmet, Sahar, Rsha, Reem, Jennings, Marina, Melissa, Robert, Joanne, Lise, Amanda, Kate, Ali, Aaron, Sarah, the list goes on and on. Thank you all for being there and in the trenches with me.
I’d like to acknowledge my friends that I have met and established in Colorado—Halley, Anshul, Mike, Jackie, Abhishek, Adam, etc., the list is very long here—for their continued support and guidance through the graduate school process. Acquiring deep perspectives of the world’s social, political and technological issues, as well as how the world views us in academia, extends far beyond campus. I have and will continue to value all of the conversations and connections I have made here with this collectively intelligent people of substance and style.

I would like to acknowledge the unwavering, loving support of my brother Dan, Erica, my parents, and friends back home in Massachusetts (and elsewhere). They make me up, opening up doors and opportunities, enabling me to step away from ‘the problem’ and tackle it later, and, most important of all, believing in what I can do, unconditionally. Thank you so much for what you have done to enable me to succeed in the life I want to live.

Cognizant of the privileged world of which I am a part, I have come to believe that the true value of an education, as David Foster Wallace articulates in his notorious 2005 Kenyon College commencement speech, is not wholly embedded in the cliché of ‘teaching you how to think’, but rather choosing how and what to think about. As I pursue my ambitions in a world different than when I came in—more brightly lit and saturated with information and technology—I submit that the most important lesson that I have learned from my graduate school experience is honing the will to take ownership of an intellectual idea. I take with this a full sense of responsibility
and the capacity to exercise it to optimize local and global conditions—for myself, colleagues, and the communities I participate in and affect.

Finally, I would like to dedicate the dissertation to my two grandparents Maria Elena Urrunaga de Kabrick and Mayor General F.A.P. Isaac Barrenechea Gonzales, who both passed away during the academic year of 2015-2016. They each had a phrase with which they used to associate me and all of the headwind I faced in becoming an adult and (now) a scholar: “YOU WILL GET THERE!” and, “el científico!”, respectively. Therefore, I end my acknowledgments by putting them together, and thanking them, one more time, for the unconditional support given to me, and for which I am now responsible for carrying out and providing to others: “YOU WILL GET THERE, científico!”
3.2.2.2 Focusing on the Crowd.................................48
3.2.2.3 Progress ..................................................50
3.2.2.4 Task Sequencing ...............................51
3.2.2.5 Gamification .........................................51
3.2.2.6 Summary ..................................................52
3.2.3 RQ3: What features are desirable for developing
crowdwork systems? What kinds of abstractions/
techniques or frameworks/tools are used by developers
to create crowdwork systems? ..........................53
3.2.3.1 No Framework ...........................................53
3.2.3.2 Task Sequencing ........................................55
3.2.3.3 Human-machine Computation .....................55
3.2.3.4 API Control ...............................................56
3.2.3.5 Quality of Work .........................................57
3.2.3.6 Versatility ...............................................58
3.2.3.7 Summary ..................................................60
3.3 Discussion ....................................................61
3.3.1 The Study Participants .................................62
3.3.2 Crowdwork Design Issues ..............................63
3.3.3 Crowdwork Framework Design Implications ......66
3.4 Summary .....................................................67

4. CROWDROUTER ..............................................69
4.1 Architecture ..................................................70
4.1.1 Core Entities ..............................................70
4.1.2 CrowdRequests and CrowdResponses ..........75
4.2 Features .....................................................77
4.2.1 Effective Routing .......................................77
4.2.2 Pipelining...................................................80
4.2.3 Branching ..................................................85
4.2.4 A/B Testing ................................................86
4.2.5 Authentication ..........................................87
4.2.6 Crowd Statistics .......................................88
4.2.7 Other Features .........................................89
4.3 Future Work ................................................92
4.3.1 WorkFlow Semantics ..................................92
4.3.1.1 Looping .................................................92
4.3.1.2 Task Shuffling .......................................93
4.3.1.3 Nesting ................................................93
4.3.2 Maintaining Progress ...............................94
4.3.3 Service-oriented Capabilities .......................95
4.3.4 Motivation ...............................................97
4.3.5 End-Crowd Programming .........................99
4.4 Prototype Implementation ..........................100
4.5 Framework vs. Abstraction ........................................... 101

5. EVALUATION ........................................................................ 103
  5.1 Versatility ........................................................................ 104
  5.2 Impact ............................................................................... 106
    5.2.1 EmergencyPetMatcher ............................................... 106
      5.2.1.1 Architecture ..................................................... 109
      5.2.1.2 Crowdwork Architecture ..................................... 111
    5.2.2 SimpleCrowdRouter .................................................. 112
    5.2.3 Evaluating Impact .................................................... 115
      5.2.3.1 SimpleCrowdRouter ............................................. 119
      5.2.3.2 EmergencyPetMatcher ......................................... 124
      5.2.3.3 Summary .......................................................... 129
  5.3 Range .............................................................................. 130
    5.3.1 GalaxyZoo ............................................................... 131
    5.3.2 MTurk .................................................................... 133
    5.3.3 Crowdcrafting .......................................................... 136
    5.3.4 Crowdwork System Comparisons ............................... 139
  5.4 Crowdrouter Versatility ................................................... 144

6. CONCLUSIONS ..................................................................... 146

BIBLIOGRAPHY ........................................................................... 150

APPENDIX

A. SEMI-STRUCTURED INTERVIEW QUESTIONS .......................... 156
TABLES

Table

1. SimpleCrowdRouter and EPM WorkFlow Additions and Changes.......................................................... 117
2. Crowdwork Systems and their Crowdwork Properties............. 140
FIGURES

Figure

1. Group by Average Programming Experience in Years ..................38
2. Example whiteboard sessions with physically-present participants (both CROWD-USE) ..........48
3. Crowdrouter Entity-Relationship Diagram ...........................................71
4. Example Crowdrouter Entities ...........................................................72
5. Example CrowdRouter route call ......................................................73
6. Traditional Routing vs. Crowd Routing ............................................78
7. An example implementation of the WorkFlow run method ..........81
8. Crowdrouter Pipelined WorkFlow Algorithm ................................84
9. A concrete AbstractCrowdChoice class to express task branching ...........................................86
10. Task Count Distribution for SimpleCrowdRouter .....................89
11. Crowdrouter UML Diagram ............................................................90
12. The Crowdrouter website ..........................................................101
15. EmergencyPetMatcher 3-tier Architecture ...................................110
17. The SimpleCrowdRouter Example Web Application ..............113
18. SimpleCrowdRouter with/without Crowdrouter .........................121
19. EmergencyPetMatcher with/without Crowdrouter ....................126
CHAPTER 1

INTRODUCTION

In his seminal piece, Notes on the Synthesis of Form (1964), Christopher Alexander weaves a discussion on designing for the natural world from an architectural perspective. He defines two design elements—form and context—in his pointed summarization of achieving good design:

...Every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. In other words, when we speak of design, the real object of discussion is not the form alone, but the ensemble comprising the form and its context. Good fit is a desired property of this ensemble which relates to some particular division of the ensemble into form and context.

The division that Alexander writes here represents a conscious commitment by the designer to the qualities of a physical product, e.g., its shape, color, and texture. It represents the array of decisions made to distinguish between how the product was designed and how its environment shaped that design. Alexander emphasizes that cultural, social, and political contexts (among others) shape what is known as good fit, and the designer, cognizant of her ability to architect a product that transforms the world from its current state to a preferred one, must also be cognizant of these contexts. In what Alexander calls a “self-conscious society,” Alexander asserts that it is a designer’s responsibility to reflect upon our designed world and have it influence their subsequent designs.
In his book, Alexander did not write about the design of software systems, but we can imagine that this balance between form and context translates to designing software just as it does to designing houses, villages, or cities. This is perhaps most evident in the way Alexander’s work has inspired the software design pattern movement (Gamma, et al., 1994; Fowler, 2002). In building software for a particular context, much of the complexity does not stem from the construction of one system; it belongs to the interdependencies between people, information, and other systems. For example, a new system may face challenges of adoption due to it replacing older systems that had familiar styles of interaction, having expectations of improved productivity, and the need to integrate with existing technologies, both physical and digital. Furthermore, a new system must also mesh with the cultural and organizational values held by the system’s constituents. A system designed for collaboration, for example, can fail when introduced to an ecosystem that values competition (Orlikowski, 1992), or when automated office systems do not accurately reflect dynamic work processes (Suchman, 1983). Truly, as has been demonstrated in diverse application domains, context (the environment) and form (the designed system) must reach mutual agreement to be a success.

Throughout society, the impact of software systems is clear: software is the fabric that weaves together an information infrastructure relied upon for the 21st century and beyond. One does not have to look far to recognize the variety of use cases that are supported by some form of software—automatic collision detection, auto-pilot mode at 30,000 feet, and—relevant to this discussion—handling thousands of users
and their interactions on social media services. As pervasive computing enables the public to connect via the Internet, a wide array of consumer devices, designed and developed for diverse use cases, are increasingly being adopted into the workplace, the home, and in civil and recreational spaces. Both the way technology is designed for the public, as well as how our social conventions, structures, and dynamics shape the technology, suggests that the developers of a successfully adopted system have carefully considered the social environment within which their system is placed—what the expectations are of the system, how the system is trusted based on use, and the extent to which it is easy to operate.

As producers and consumers of information found online, we are actively participating in the construction of a large-scale socio-technical ecosystem. Users rank, remix, and reshape resources that are of interest or importance to the crowd, and filter or ignore those that are old and outdated. Software is no longer viewed as an alienating force as satirized by Nelson (1977) but as a mechanism to construct sites of bonding, networking, communication, sharing, and problem-solving. Social networking sites broadcast these actions so that users can witness the sheer volume of data being transformed, enabling new ways of interacting, communicating, and sharing. Consequently, these online “public spaces” as William Whyte perhaps would call it (1980), are rich sites for viewing the relations among people and information, understanding how form affects context. In other words, a system designer might be interested in building a software system with the intention of having people use it. Yet in order to do so, knowledge of why people visit sites like the one proposed, what
they do, as well as how and why they do it, are vital questions to raise during the engineering design process.

These observations have had a transformative effect in how systems can be designed, especially those systems that host problems and allow users to perform large-scale collective and online work to solve those problems. This arrangement, based upon the relationships among systems, problems, and workers, as well as their specifications, needs, and expectations, are investigated by the field of crowdfwork. Also known as crowdsourcing, crowdfwork reframes the focus from a one-to-one point of interaction—between a system and a user—to one-to-many, devising techniques to decompose problems into small chunks that each user from a crowd is able to perform. This reframing offers opportunities for solving scalable problems that computers cannot solve on their own, allowing humans to be in the mix. Many exemplary systems have been designed this way in order to enhance web security measures for websites while digitizing physical book text (von Ahn, et al., 2008), map buildings, roads, and landmarks on a community-led map of the world during disaster events (Palen, et al., 2015), and create on-demand microtasks for text proofreading within word-processors (Bernstein, et al., 2010).

From top-down microtasking to bottom-up ad-hoc mobilization, crowdfwork can be viewed in many different contexts, such as that of crisis response, where a large convergence of disaster victims, both remotely and on-the-ground, are using technology to solve problems related to a disaster event (Palen, et al., 2010). Using technology through the use of personal computing devices, people are combating the
information dearth that is common in situations of mass emergency and uncertainty. They are logging onto social media platforms to help each other find essential resources such as food, water, shelter, and emotional support. They are participating online, becoming digital volunteers through the process of collectively making sense of information, and then producing and sharing more information to help with the disaster response (Starbird & Palen, 2011). This unprecedented wave of digital volunteerism has sparked innovation for software design and development as new software systems are assisting the crowd in performing meaningful and visible tasks, drilling down into large sets of data, gaining situational awareness, re-establishing strong and weak social ties, and collaborating with each other during a disaster event. Systems that do this well are seeing high rates of adoption because of their commitment to form and context; those that fail to understand how they are actually being used and do not respond to the needs of their users are doomed to fail.

Indeed, this is the critical objective of crowdwork systems. Not only must the system be designed properly to reliably host problems for the crowd at scale, but it must also be designed such that they understand its motivations. Digital volunteer-based crowdwork systems find motivation in altruism, and a will to become a Samaritan even in online public spaces, but other systems that do not recognize why the crowd wants to perform its work will risk workers feeling undervalued and misrepresented. These issues are common across many existing crowdwork systems (Martin, et al., 2014), and if left unaddressed, they can contribute to a vision of a Tayloristic future of crowdwork systems (Kittur, et al., 2013).
1.1 Objective

The objective of this dissertation is to design a new versatile framework to help developers address socio-technical challenges related to crowdwork system development. I achieve this objective in the following manner: first, I conduct a study with self-identified developers who have either used or developed crowdwork systems. Participants were asked to answer a range of development and crowdwork-related questions, including brainstorming solutions for a real-world crowdwork problem. As a result, this study generated insights into the socio-technical challenges that developers face when constructing crowdwork systems and using crowdwork frameworks.

In the context of this dissertation, I focus on the problem of *versatility*—the capacity with which a tool is used to satisfy a multitude of use cases—to inform the design of a new framework I have created called the Crowdrouter. The Crowdrouter enables first-class treatment of workflows and tasks so that the developer can more productively handle crowdwork complexities when building a system. It is also designed to minimize the large number of dependencies/requirements and high learning curves that are evident in other crowdwork frameworks, further increasing its flexibility to be integrated into crowdwork development projects. As a result, the Crowdrouter framework is lightweight, available in the popular Python programming language as an open-source library, and provides capabilities that map to desirable features for crowdwork frameworks identified by participants in my study.
In order to show that the Crowdrouter is a step in the right direction for the pursuit of better software tools for crowdwork system development, I evaluate the framework’s versatility as a function of both the impact it produces on software systems and the range with which it can address a diverse set of crowdwork use cases. I reference two crowdwork systems that I have constructed in the past as cases against which the Crowdrouter can be evaluated with respect to impact. They include a small web application that demonstrates Crowdrouter features called the SimpleCrowdRouter (SCR), as well as a pet reporting and matching system called EmergencyPetMatcher (EPM). The Crowdrouter is integrated within these systems to demonstrate impact, and a survey of popular crowdwork systems are tabulated to demonstrate the range with which it can satisfy diverse crowdwork configurations. By making these assessments, I show the level of versatility that is afforded by the Crowdrouter framework.

Chapter 2 grounds the dissertation work in the background literature. Chapter 3 illustrates the crowdwork study used to derive insights about crowdwork system development challenges and desired features for crowdwork frameworks. Chapter 4 summarizes the Crowdrouter tool and all of its features, as well as features that will be implemented in the future. Chapter 5 discusses the evaluation approach I took to show that the Crowdrouter is a versatile tool that can be used in a variety of crowdwork systems. Finally, chapter 6 concludes the dissertation with reflections and avenues for future work.
CHAPTER 2

BACKGROUND

This chapter grounds the dissertation in the literature, situating it along crossroads between research in software engineering, software architecture, human-computer interaction, social computing, crowdwork, and crisis informatics. The overarching objective is to place the Crowdrouter framework within this intersection, and that requires attention to the rich fields that have contributed to our understanding of the forms (software architectures) and contexts (crisis informatics, crowdwork) relevant to this dissertation.

In this dissertation, I introduce a crowdwork framework called the Crowdrouter. To properly situate it among the literature, I will discuss the following fields that have influenced its design and the challenges that I seeks to address: crisis informatics, crowdwork, and software architecture. Additionally, I survey the available tools for developers in constructing crowdsourcing applications from academic studies, open source, and industrial settings, and I properly place the Crowdrouter in context around these tools.

2.1 Crisis Informatics

Crisis informatics is an emerging field of study that examines socio-technical relationships among people, information, and technology during disaster events. While Hagar has coined this term to refer to the interconnectedness of people,
organizations, information, and technology during crisis events (Hagar, 2006), work in human-computer interaction and computer-supported cooperative work led by Palen and others have further supported the definition by providing a re-orientation towards these fields: “Crisis Informatics is the study of information and communication technology (ICT) in relation to actual or potential mass emergencies, with a particular focus on the role of social computing in such situations (Palen, 2014).” Based on these definitions, crisis informatics can be described as an interdisciplinary field that studies the interconnections of the people who experience disasters on-the-ground and from afar and perform work together in the technical landscape brought forth by ICT, but not excluding other non-computational means of producing and disseminating safety and time-critical information. Also crucial to this description is the information divide that separates the public’s perception of the disaster event from the perception pushed by federal agencies, local emergency management, and other formal response organizations. With pervasive computing in the hands of the public, the expectations placed on information—and the velocity, veracity, and volume with which it is produced and shared—are now needed by both sides to gain a situational awareness of the disaster event as it unfolds.

During disaster events, it has been observed that victims do not exhibit what popular media outlets have illustrated as panic and knee-jerk fleeing behaviors from the disaster scene. Rather, they have been seen to be converging onto the disaster scene to assist and help others. In fact, contrary to established notions about formal organizations responding to the event, members of the public are the true first
responders to the scene, coordinating with each other to pull victims out of the debris, drive the injured to shelters, perform impromptu rescue missions, provide supplies, and give emotional support (Fritz & Mathewson, 1957). In the 2001 World Trade Center attacks, people were ferried away from the fringe zone by spontaneous and coordinated work by ship captains (Kendra & Wachtendorf, 2007). During the 2010 Haiti earthquake, people converged onto the disaster scene to help find missing people, establish shelters for temporary relief, and other important relief tasks (Soden & Palen, 2014). These behaviors are not new; disaster sociologists have studied them for a century. That we are now seeing them online in software development contexts for humanitarian efforts and through social media, however, has generated a wave of large-scale public participation not seen previously.

Socio-behavioral work in this space uses a wide range of empirical research methods to discover and report on the convergence that is witnessed on social computing platforms such as Facebook (Palen & Vieweg, 2008; Palen, et al., 2009; White, et al., 2014), Twitter (Starbird & Palen, 2011; Starbird, et al., 2010), Skype (Starbird & Palen, 2013), and OpenStreetMap (Palen, et al., 2015), but it equally investigates “offline” cooperation and collaboration work with animals (White & Palen, 2015), maps (Soden & Palen, 2014), and other representations for humanitarian aid and response. Technical aspects of the field, on the other hand, examine the way in which software technologies are used by diverse stakeholders in the context of disaster events. Work in this area is sparse, given that research work
requires attention to a wide range of functional and non-functional factors (Barrenechea, 2015a), including:

**Commitment to the domain**: researchers need to have a deep contextual understanding of the targeted user base, relationships among those users, as well as have a research duty to protect private information that such users provide.

**Prototype to Production**: prototypes developed for intervention must be designed for scalable and reliable use, since such prototypes will potentially be used by many users.

**Usability**: They must be engineered with usability in mind, or else they won’t be used. This requires great attention to usability methods for interface design.

Systems such as Ushahidi (Okolloh, 2009), EmergencyPetMatcher (Barrenechea, et al. 2012; Barrenechea, et al., 2015a), and CrisisTracker (Rogstadius, et al., 2013) are examples of public technical interventions for disaster response. Community systems like Ushahidi have seen greater use than systems such as EPM and CrisisTracker. The latter systems were built to support crisis informatics research and thus have a smaller set of stakeholders. Still, the most popular systems used during disaster events are social networking services such as Twitter and Facebook due to the sheer scale and reliability that they achieve in connecting hundreds of thousands of people together, disaster or not.
Additionally, in the area of software infrastructure to support social media analysis of disaster events, systems must adhere to the above principles and perform information collection, synthesis, analysis, and dissemination (Anderson & Schram, 2011; Schram & Anderson, 2012; Oussalah, et al., 2013; Anderson, et al., 2013; Anderson, et al., 2015; Barrenechea, et al., 2015b). These systems identify a diverse set of challenges for the proper handling of large-scale data generated during major disaster events, as well as smaller-scale ones that produce large spikes of data from the convergence of those who produce and share information about the event. Ephemerality of the data is also an issue: systems must be timely enough to produce representative samples of event-based data, since research examining online behaviors depends on it (Anderson, et al., 2013). Thus, the use of big-data analytics to perform these large-scale functions on data produced during times of mass emergency has by-and-large become a requirement for representative analysis.

An important aspect of crisis informatics research is seeking to understand how disaster-abetted behaviors of emergency management and members of the public inform the design and use of systems during such events. In the forefront, this aspect speaks to the actions that are crucial in performing empirical software engineering research in this domain. What is important in this frame is the connectivity that software engineering has within the interdisciplinary nature of crisis informatics. If it were not for the plethora of software serving as social media, blogs, case management tools, and other important systems, researchers in information science, natural language processing, and other fields could not examine online behaviors
exhibited by the formal and public “informal” response to the event. Moreover, little work has investigated the extent to which the software itself has shaped those behaviors. The objective for software engineering research in this domain, therefore, is to empirically engage in design, development, and evaluation of future software systems built within the socio-technical environment and determine the ways in which it has enabled safety and time-critical online and offline work to be done.

2.2 Crowdwork

In the mid 2000s, Howe (2006) wrote about a new paradigm called “crowdsourcing” for performing distributed work by a large collection of workers. Since then, crowdsourcing has become a broad term to encompass the nature of managing work done by “the crowd” as well as the work itself, from volunteer crowdsourcing to business-driven “outsourcing” of labor (Starbird, 2012b).

Crowdwork, in its broadest sense, describes an approach to mobilize people in performing a recognized decomposable problem.\(^1\) It is a technique that has been developed and used to provide tasks to those willing to perform them, and it has seen success in various application domains, such as solving geo-spatial computational problems in humanitarian affairs (Meier, 2015; Palen, et al., 2015), discovering new algorithms for protein folding (Khatib, et al., 2011), and finding red balloons across

---

\(^1\) In this dissertation, I treat the word “crowdwork” as synonymous with “crowdsourcing”, but in reality “crowdwork” is the official term used in social computing to refer to a body of literature that reveals insights into how crowds form and perform work. “Crowdsourcing”, on the other hand, is a more casual term used to denote the outsourcing of labor to a crowd. Because the distinction isn’t important in the context of presenting and evaluating a tool built for developers, they are interchangeable in the text. However, for consistency, I will use the term “crowdwork”. 
the United States in the DARPA red balloon challenge (Tang, et al., 2011). Crowdwork systems make use of two important entities—the task and the workflow. A task is a single unit of work that the crowd performs; a workflow is a set of tasks that the crowd typically works through to perform those tasks. Most crowdwork systems rely on these entities to allow the crowd to perform work.

For work to be completed efficiently, traditional crowdwork systems rely on independent, serial, and single-user workflows because the work is decomposable and thus parallelizable. Kittur et al. (2011, 2013) draw parallels between such systems and distributed system theory because the style of work is similar to a MapReduce job, where work is splintered into independent equal-sized chunks, allowing the crowd to also splinter itself and perform the work needed to get the job done. This is often the case in such systems like Amazon Mechanical Turk (AMT), where each piece of work—a human-intelligence task—has a price that is awarded to workers who perform it, but the platform rarely hosts tasks that include multiple people working collaboratively and simultaneously.

But not all work is composed this way. Digital volunteer systems, as I will define, supports the crowd in defining and performing their own tasks, a sharp contrast to what is known about crowdwork. Human-machine computation, on the other hand, prescribes tight feedback coupling between crowd work and machine work. Finally, citizen reporting treats members of the crowd as sensors for information gathering, reporting, and sharing. Below I delineate these perspectives
under which crowdwork is defined, reinforcing the foundation that Starbird (2012b) writes in unpacking this often conflated term.

2.2.1 Microtasking. Microtasking is a crowdwork paradigm that assigns work from the top-down, or created by requesters (taskers) so that workers can perform them. Typically, these systems do not actively change their services based upon the results produced by the crowd. They combine aggregate work performed by the crowd to provide “big-picture” results to inform other systems or people who need such information for enhanced decision-making, triage, and aid. Microtasking systems can also provide rewards to workers in the form of monetary compensation (e.g. AMT) or reputation (Micromappers, CrowdCode) to keep working.

Systems such as the Soylent document editing system (Bernstein, et al., 2010), the Umati crowdsourced vending machine (Heimerl, et al., 2012), and AMT are examples of microtasking systems. These systems can even be used for more specialized tasking, such as CrowdCode, where requesters can assign function and testing tasks to programmers in order to crowdsource a software library (LaToza, et al., 2014). Microtasking systems during disaster events are commonplace as well, because their ability to use an online human workforce provides scalability in performing work that machines have trouble with performing alone (Meier, 2015; Palen, et al., 2015).

2.2.2 Digital Volunteerism. In contrast, disasters have proven to be sites of innovation between and among members of the public (Starbird, et al., 2010; Starbird & Palen, 2011; Palen, et al., 2008; Palen, et al., 2009; Palen, et al., 2010), emergency
response personnel (Hughes, et al., 2014), and other non-human resources such as animals (White, et al., 2014; White & Palen, 2015) to perform digital volunteerism, a bottom-up tasking paradigm. Digital volunteerism as a reference to a particular style of crowdwork is in stark contrast to its parent term in that the work performed by online volunteers is not thought about beforehand but rather constructed as a result of the collaborations among them, a bottom-up style of work (White, et al., 2014). There is no one to answer to, no one person for which to perform work, but instead there exists a diverse yet similarly-minded community of people who work together to establish cooperative work practices to solve recognized and meaningful problems. Much work aligned with this perspective of crowdwork has been conducted in the crisis informatics arena. In this dissertation, I reference digital volunteer systems as crowdwork systems that mobilize online communities that reflect this bottom-up tasking paradigm.

Digital volunteer systems play a subtly different role than microtasking systems. To support the behaviors of communities that engage in mutually meaningful and highly collaborative work, digital volunteer systems are designed with collaboration channels, allowing communication among workers, as well as support for improving upon work practices as the values of a community evolve. Their organizational structures vary as well; most are flat with no human-based coordination mechanisms to assign work, as in the case with activations of the Humanitarian OpenStreetMap team (Palen, et al., 2015), yet there are some that host volunteers that dedicate time and energy to onboard, coordinate, and verify work,
such as the Stand-by Task Force (Starbird & Palen, 2013). Digital volunteer crowds are also typically not as large as microtasking crowds, though their commitment to the online work in concert with similar-minded members, as well as their shared expertise, accommodates for their size.

2.2.3 Human Computation & Human-Machine Computation. Yet another perspective of crowdwork hones in on the configurations of work between humans and machines. Human-machine computation wields together processing of human and machine workflows within systems that neither workforce can perform alone. This is sometimes known as the “artificial artificial intelligence”, one that holds both computational entities on equal footing and allows one’s processing powers to affect the other’s, and vice versa. This class of systems derives from the notion of human computation, one that illustrates how humans perform work that is facilitated by the machine that could not perform it alone (von Ahn, 2005). The difference between a human-computation and a human-machine computation system, therefore, is in their workflows. On one hand, human computation systems relegate the machine to facilitate and rely upon work done by humans, constantly filtering, ranking, searching, and otherwise organizing that work, but the machine never performs tasks itself, nor does it improve upon or learn from the crowdwork; on the other hand, human-machine computation systems such as those envisioned by Quinn and Bederson (2011) include the machine that excels in performing mass computations and benefits from human work done to improve upon its processing capabilities. Both classes of crowdwork systems mobilize the crowd to perform work,
but they are distinguished between facilitating the crowd versus providing feedback to each other as they perform computations symbiotically.

Examples of human-computation systems include the reCAPTCHA system designed to use the crowd to digitize old written text more accurately than optical character recognition systems, while also maintaining web security measures (von Ahn, et al., 2008). FoldIt (https://fold.it/portal/info/about), a protein-folding experiment with a human-computation twist, makes it fun for users to predict protein structures while also contributing to scientific processing. In this dissertation, I discuss EPM as a digital volunteer and human-computation system, one that brings elements of mutually meaningful participation as well as machine-to-human support to an important societal problem—pet-to-family reunification after a disaster. I discuss the challenges highlighted by these characteristics of crowdwork and how they have architectural implications for future system design.

2.2.4 Citizen Reporting. Citizen reporting refers to the gathering of humans-as-sensors to report on collective wisdom regarding an issue or situation. Howe demonstrated in his early article that gave rise to the concept of crowdsourcing (2006) that such a framework for soliciting information from the crowd can be witnessed in many forms, from choosing wedding photos to highlighting graffiti, to reporting missing persons, etc.

Citizen reporting is very much like crowdsourcing in that the scale of work completed is dependent on the size of the crowd, but what is teased apart is the assignment of work, as well as the motivations for actually performing the work. An
unprecedented wave of citizen reporting during disaster events has shown how the collective wisdom of the crowd acts as a distributed network to altruistically document, archive, share, and verify information coming from the ground and from online sources. Ironically, systems like Facebook and Twitter are homes to these networks and their functions, yet they do not afford features that are specialized for the work that they perform. Nevertheless, the allowance for humans to share insights from their environment onto a platform appropriated for their needs is characteristic of citizen sensing and reporting for the collective action by the crowd.

Crowdwork systems vary in work, style, audience, and transparency. Literature in the fields of social computing and computer-supported cooperative work describe crowd work systems like AMT as a technical but Tayloristic approach to managing micro-labor for online solicited tasks (Kittur, et al., 2013; Martin, et al., 2014; Kittur, et al., 2008). As documented in these studies, it is easy to construct micro-tasks that do not engage the crowd in performing high quality work; not all work gets rewarded fairly, workers are treated poorly, and the visibility of work is menial or opaque, leading to lower quality work. As a reaction both in the form of criticism against the affordances of the AMT platform and the experiences reported by workers and taskers, systems such as Turker Nation (Martin, et al., 2014) have sprung up to collectively address issues of transparency and fairness in crowdsourcing. However, an alternate future paved by systems like Zooniverse (https://www.zooniverse.org/) and GalaxyZoo (http://www.galaxyzoo.org/) aims to embody such values within their framework, either offering a highly collaborative,
visible workspace for individuals with similar interests or a mutually beneficial and verifiable one for crowdwork in-the-large. Regardless of whether systems are designed to accommodate microtasking, digital volunteerism, human-machine computation, or citizen reporting, they should address the socio-technical issues inherent in the nature of crowdwork, and they should frame the available work as meaningful, visible, and fair for both workers and taskers. Otherwise, issues related to system use may arise.

With regards to crisis informatics, this dissertation supports and extends the research work done by Starbird (2012b) by empirically examining how crowdwork can be viewed as an architectural abstraction for future software design and development. The framework under study—Crowdrouter—is a system that provides software abstractions for routing and encapsulating crowd workflows and tasks. This can aid application developers in organizing crowdwork while adapting to the crowd’s demands within the system.

2.3 Software Architecture

Software architecture is an active research field that began in the early 1990s (Kruchten, et al., 2006). It emerged as a separate branch of software design, one that focused on the difficulties of creating large-scale software systems (Perry & Wolf, 1992; Garlan, 2000). Since its inception, research directions span from architectural description languages for designing architectures (Medvidovic & Taylor, 2000) to socio-technical insights in how architects collaborate (Bang, et al., 2013). Its history
has been articulated by the concerns of software engineering practitioners and researchers who found that large-scale software development poses greater challenges than programming-in-the-small (Deremer & Kron, 1976) and have since been committed to codifying architectural styles and patterns (Garlan & Shaw, 1993).

But simply describing research efforts in software architecture does not tell the whole story. Software engineering industry leaders such as Martin Fowler (2002) and the Gang of Four (Gamma, et al., 1994) have proposed that such codification can indeed support a common vocabulary to communicate implementations from one developer to another. The area of software design patterns has taken off considerably since these seminal works started drawing attention, allowing developers to see gains in extensibility, flexibility, scalability, and maintainability of software because such patterns have been tried and tested in practice.

There is no doubt that each pattern within a pattern language is an easily understandable “nugget” of knowledge, creating a higher-order vocabulary to discuss tradeoffs and implementation details of how to construct software systems. Shaw notes (2001) that software engineering paradigms have yet to fully appreciate, understand, and apply methods of inquiry within the field, yet it is evident that patterns can be an effective way of connecting research to practice.

One aim of this dissertation is to identify patterns that can be incorporated into crowdwork software frameworks that, in turn, enable crowdwork systems to be developed more efficiently and reliably in the future. A goal of the dissertation is then
to show that this framework can be shown to satisfy a wide array of use cases discovered through the study conducted on crowdwork use and development.

2.4 Crowdwork Frameworks

Crowdwork is an active research field that investigates a wide range of socio-technical issues surrounding crowd workflow and task design, reward systems, motivation, progress, and quality of work, to name a few. But it is critical that software development tools and techniques are brought to bear on the design and development of these systems and how they commit to solving these issues. I now turn to these tools and describe their use, the issues they solve, and how they compare to the framework of study—the Crowdrouter. This analysis is crucial in situating the Crowdrouter among a breadth of existing tools to aid developers in constructing crowdwork systems.

2.4.1 Crowdforge. Crowdforge (Kittur, et al., 2011) is a framework for designing crowdsourcing tasks using the MapReduce distributed computing paradigm. It targets users (not developers) to configure a problem that they want to solve. The system breaks that problem down into partition, map, and reduce tasks that each call upon the MTurk marketplace via an API to have the crowd perform pieces—instituted as human-intelligence tasks, or HITs—of the problem. Specifically, a partition task breaks up the problem into HITs, a map task applies crowd processing to a HIT, and a reduce task aggregates results from HITs.
Using the Crowdforge framework, users define a problem that they want to solve, define HIT templates (partition, map, reduce tasks) that are referenced by that problem, and then launch the problem, which the system responds to by connecting to the MTurk platform and creating HIT tasks for the crowd to perform based on the problem definition. Part of this crowd processing is dynamic in the sense that the results from one HIT will cause future HITs to be created, which depends on the results generated from the previous HIT. Crowdforge is built as a fully-fledged web application using Python, Django, an API connector to Amazon’s MTurk platform, and a database for storing results.

In comparison, the Crowdrouter framework is a tool to help construct and manage tasks and workflows, much like the HIT templates and flow instances managed by Crowdforge. However, while Crowdforge relies upon a MapReduce specification for generating tasks, the Crowdrouter does not restrict itself to a parallel processing model in which partition, map, and reduce tasks must be specified. Instead, the Crowdrouter takes a more general-purpose stance and provides the programming facilities to create concrete task classes that are brought together by a workflow class specification. These specifications are then used at runtime to properly route the crowd through these tasks. Therefore, the Crowdrouter is designed primarily as a lightweight framework for programmers to create crowdwork apps instead of users who want to perform crowd-based batch processing jobs via MTurk.

2.4.2 TurKit. TurKit (Little, et al., 2010) is a programming tool built on top of Javascript to crowdsource tasks—using HITs from MTurk—from an imperative
program. TurKit essentially provides a “crash-and-rerun” programming model in using familiar programming constructs (e.g. if-conditions, while-loops) to process crowdwork coming from MTurk. This means that the program will make an API call to the crowd and will wait for the results, but if they do not come back in some specified time, then the program will crash, but upon executing the program again once the results are ready, the program will store them in a database so that subsequent program re-runs will not make the same (costly) call again. Developers can run simple crowdwork programs this way, such as calling upon the crowd five times to vote on three defined options and aggregating the result within the program, even when the results don’t all come back in time. A simple re-run of the program will finish the work.

Compared to the Crowdrouter, TurKit gravitates to the other end of the spectrum where it caters mainly to developers and provides a familiar interface—the imperative programming model. It achieves effective crowdwork programming via this interface, using persistence in the background to store results coming from the crowd. In stark contrast, however, the Crowdrouter provides the abstractions necessary to specify the crowdwork and relies upon the Model-View-Controller paradigm to handle executions of the crowdwork tasks on the web. Because of its environment, it does not focus on becoming a fully-fledged crowdwork programming language, but rather a framework that developers use to build crowdwork systems.

2.4.3 iCrowd. iCrowd (Fan, et al., 2015) is a framework that seeks to increase MTurk microtask execution accuracies by using graph-based estimation models to
predict worker-to-task optimization. The iCrowd system formalizes this as an optimization problem and uses greedy algorithms to assign HIT tasks to workers that have previously qualified for similar HIT tasks. Though the framework helps provide an increase in 10-20% accuracy in HIT tasking compared to other solutions, iCrowd does not focus on how users may be able to leverage it for their own crowdwork applications. This is in contrast to the Crowdrouter, which may not provide estimation models for serving specific tasks to specific users, but it provides the flexibility with which these estimations can be implemented on top of a workflow specification. In other words, the implementation details of the iCrowd framework could in fact be placed within a Crowdrouter workflow.

2.4.4 PyBossa. PyBossa is arguably the most popular open-source crowdsourcing framework available today. It enables anyone to set up a crowdsourcing website with a few steps, including setting up a database, web server, cache server, and basic functionality for defining and serving crowd tasks. Authentication can be configured to permit anonymous usage or restrict (registered) usage. Users can create tasks using a built-in task interface and upload various media types such as Google spreadsheets, CSV, JSON, etc. It also provides an administrative dashboard, where privileged users can check on the status of tasks and visualize task run performances over time. CrowdCrafting is a citizen science platform that uses the PyBossa framework for managing and serving citizen science projects. The MicroMappers platform is also implemented using the PyBossa framework, customized so it can not only support geo-clicking and image-clicking
tasks but also send results to other systems such as the AIDR platform (Imran, et al., 2014).

Compared to the Crowdrouter framework, PyBossa is a much more mature system, stemming from the BOSSA framework (originally implemented in PHP). Because of this, it boasts many more features and is thus a more heavyweight framework; it provides the scaffold for most of the technology stack and leaves users with one job which is to define the tasks that need to be performed for the target crowdwork system. In contrast, the Crowdrouter boasts a relatively small set of features but does not constrain the project in which technologies or services to use, achieving flexibility through minimal requirements such as extra python packages and enabled web sessions.

On data modeling, PyBossa recognizes and encapsulates Projects, Tasks, and Task Runs, where a Project can have multiple Tasks, and Tasks can have multiple Task Runs. A Task Run represents the results of an execution of a Task instance. The Crowdrouter contains three models as well—the CrowdRouter manager class, the WorkFlow specification class, and the Task class. However, there are stark differences between these model sets. For one, there is no equivalent Task Run class; no results are stored by the Crowdrouter because it leaves it up to the application developer to store them. Secondly, the Crowdrouter relies upon the WorkFlow class to specify the implementations for how Tasks are executed, whereas in PyBossa, they are specified through the project web interface in any order desired. Third, the
Crowdrouter models are not persistence models, and so therefore creating extra database tables in the application is not needed.

While the PyBossa framework enables users to quickly build crowdwork web apps with ease, it creates major assumptions about the project environment and constrains the users to use the technologies it requires in order to function properly. The Crowdrouter is much more lightweight in this way; because it can be implemented in any language, the Crowdrouter can support heterogeneous environments where crowdwork is a vital concern but does not mandate technologies to use. It achieves the crowd-routing functionality using a familiar interface while PyBossa uses web interfaces to properly configure those workflows.

2.4.5 Hive. Hive (2014) is an open-source crowdsourcing framework from the Research and Development department at the New York Times that uses the popular search engine Elasticsearch to serve users, assets (media), task, assignments—a mapping between a user and a task—, and projects. Application developers need not worry about how the framework integrates within the application; the framework sits within a thin layer of middleware that is deployed along with an Elasticsearch server. In this manner, developers can concentrate on building front-facing crowdsourcing applications and making simple API calls to the Elasticsearch server to manage tasks, users, and projects while having the framework collect data on assignments and other crowd statistics.

Hive is an attractive option for developers who desire a compact solution for crowdsourcing asset, user, and task management. However, it does not offer
mechanisms to control task sequencing, branching, or grouping, leaving those responsibilities to the developer to implement. In contrast, the Crowdrouter focuses on workflow specification, but lacks the cross-platform API flexibility that is afforded through Hive’s middleware. Again, while Hive requires several aspects of the application technology stack, the Crowdrouter integrates well within any Python web framework to provide workflow and task specification and management.

In summary, these existing crowdwork frameworks offer features for controlling the execution of tasks to the crowd in various ways. To further filter down on the crowdwork frameworks to examine alongside the Crowdrouter, it is critical to classify them with respect to the crowds they help serve. For instance, the iCrowd, CrowdForge, and TurKit frameworks are all designed to invoke crowdwork to a specific crowd—the labor marketplace of MTurk. In contrast, frameworks like Hive, PyBossa, and the Crowdrouter are crowd-agnostic, enabling the developer to concentrate on designing crowdwork solutions for a target crowd of her choosing. Because they are on equal footing, these crowd-agnostic frameworks are important to discuss in further detail as to how they serve important features to the developer.

With respect to task management within crowdwork frameworks, Hive, PyBossa, and the Crowdrouter all have support for treating tasks as first-class entities. However, between these frameworks, the treatment of tasks differs. For example, with Hive, developers can define tasks as instances of the task model, which

---

2 This is not to imply that these frameworks and their techniques cannot be used in other crowd contexts; however, with the way that they explicitly invoke APIs of MTurk, they frame the crowd as a processor of work, rather than a heterogeneous set of people with varying skillsets and motivations. Therefore, it is safe to make the distinction that they are programmed to serve a specific crowd and not all crowds.
is defined by the framework. Similarly, PyBossa offers a GUI-based interface for creating/uploading tasks to its task creator module, requiring parameters that are used to render to various templates offered by the framework. However, in stark contrast, while the Crowdrouter also offers task management, it is implemented as a class definition, which provides a couple of benefits: for one, developers can construct tasks directly within their applications without needing to externally configure them through GUI interfaces, API calls, or model definitions. Secondly, within their own applications, developers can define however many attributes or methods that are important for the application to serve such tasks, without being constrained by the framework to manage these tasks.

Additionally, with respect to workflow management, which is an important concern in crowdwork systems (Kittur, et al., 2013; Lasecki, et al., 2015), these frameworks offer varying degrees of support to the developer. Because Hive only provides a RESTful API for crowdwork-related entities such as tasks, assignments, and users, there is no support for workflow definitions, task sequencing, parallelization, dependency management, or shuffling. PyBossa offers some workflow features like scheduling tasks based upon their task-run history, as well as prioritization of those tasks, but it does not offer support for workflows as an entity that can be managed via developers, nor does it offer such features like sequencing tasks or creating dependencies between them. The Crowdrouter, on the other hand, treats workflows just like it treats tasks—as a first-class programmable entity. In this way, developers can use these workflow classes to provide criteria that determine
which task is to be served to members of the crowd. Crowdrouter workflows can string together tasks into a logical flow, allowing the crowd to execute tasks without interruption. To the best of my knowledge, the Crowdrouter is the first of its kind to offer workflow features within a crowdwork development framework.

These are some of the critical differences between existing crowdwork frameworks and the Crowdrouter framework. The Crowdrouter operates in the smaller space of crowdwork frameworks that are crowd-agnostic, meaning that they do not constrain the developer in choosing which crowd to serve via the application being constructed. More details on the design and features of the Crowdrouter are discussed in Chapter 4.

2.5 Summary

My thesis lies at the intersection of the research fields of software engineering, software architecture, crisis informatics, and crowdwork. Software engineering and architecture help to ground the research to understand patterns of software design and how it affects system development for various application domains. In the domain of crisis informatics, I have discovered two important insights: one, the field makes contributions in understanding the instantiations of crowdwork pertinent across a wide landscape of tools and services to aid disaster preparation, response, and recovery. Secondly, the area of crowdwork is the context of the research work I carry out in this dissertation, but crisis informatics teaches us how to critically engage in not just technical but social issues relevant to the human-to-computer interactions
that underpin the contextual uses of technology during disaster events. I use these lens to identify crowdwork as a tool—not just a host of issues—that aid developers in constructing future crowdwork systems.

In the next Chapter, I describe a study that I performed to gain insight into the requirements needed for a crowdwork software framework; then, in Chapter 4, I describe my contribution to the crowdwork development space—a new crowdwork framework for designing crowdwork systems called the Crowdrouter.
CHAPTER 3

CROWDWORK STUDY

This chapter presents a study to investigate issues related to crowdwork and the development of crowdwork systems; the purpose of the study is to identify the challenges that are faced by developers who have used or created crowdwork programs in the past. The objective of the study is to surface the social and technical issues faced by crowdwork users and developers and translate these challenges into requirements for any software framework that aims to help developers create crowdwork systems. I intend to use these requirements to design my own software framework for crowdwork development that I call the Crowdrouter. The study also serves a second purpose: my conversations with developers have identified a wide range of features and workflows that crowdwork systems need to support and I can use this information to evaluate the utility and versatility of the Crowdrouter framework. The more of these features and workflows that Crowdrouter supports, the better situated it is to serve as a framework for crowdwork software development.

I begin by describing the design of my study, including information about the participants, the organization of the study itself, and any threats to validity that the study exposes. I then distill the results of the study and synthesize what crowdwork issues have been revealed by the interviews. I then translate these results into requirements for crowdwork development frameworks, such as the Crowdrouter. In the Evaluation chapter, I will then compare the Crowdrouter design and its prototype
implementation and evaluate how it satisfies these requirements and whether any limitations or gaps in its support exist.

3.1 Study Design

3.1.1 Research Questions. The study was conducted to derive insights into the development challenges faced by programmers with varying levels of crowdwork experience when using, designing, or implementing crowdwork systems. Specifically, the study asks three research questions, outlined here:

**RQ1**: Do developers design crowdwork systems differently than non-crowdwork systems?

**RQ2**: How do developers design crowdwork systems? What challenges do they address?

**RQ3**: What kinds of abstractions/techniques or frameworks/tools are used by developers to create crowdwork systems?

The first research question is a probe for understanding the difference in programming style or approach in constructing crowdwork systems. Participants in the study were asked this question during the interview and asked to think about what those differences are, if any. These insights prepared them to answer the next research question, which is to describe how they attempt to design crowdwork systems. Their strategies are revealed via a brainstorming session where they are asked to solve a real crowdwork problem. Finally, a more open question that was asked of the participants was about the tools/techniques that they have used when
using or building crowdwork systems, which generally also led to other questions about what features are important to the design of crowdwork systems. Together, these research questions directed the participants to think about crowdwork as a unique paradigm for programming and to help identify its unique challenges.

3.1.2 Structure. The study was conducted as a one-hour semi-structured interview consisting of a series of experience-related questions, crowdwork-related questions, and then a brainstorming session that walks the participant through a real-world crowdwork problem. The brainstorming problem closely resembled one that the MapMill application was trying to solve following the 2012 Hurricane Sandy event (2013). This problem was chosen because of its ease with which to create solutions—no one answer was necessarily the correct one because it can be solved in many different ways combining both crowd and machine processing. A semi-structured interview format was chosen because of the diverse background from the participants and the ease with which crowdwork topics raised during the interview could diverge into other meaningful topics related to experience. The Appendix contains the interview questions, as well as the brainstorming starting points that were used to guide the interviewee through a session.

After the questions and brainstorming sessions were complete, some interviewees were curious as to what research was being done and asked about the purpose of the crowdwork study. As the interviewer, I responded by introducing them to the Crowdrouter framework that I am developing and releasing under an open source license. However, to prevent bias and to keep the focus of the study on
crowdwork challenges and design, the Crowdrouter was not discussed until after an interview was over.

3.1.3 Data Collection. Interview data was collected by note-taking as a primary data collection method. No visual or audio recordings were performed. During the brainstorming session, a whiteboard was used in all of the co-located interviews. In non-co-located settings, a Google doc was used to synchronize the brainstorming discussion; the Google doc was also used by the interviewer to take notes during the interview. These methods were sufficient to capture enough data to draw scientific conclusions from the interviews.

3.1.4 Participants. There were 12 self-declared programmers/developers who participated in the study. All of them have used crowdwork applications in the past, while 7 of them said that they have been involved in building them. It was important to include developers who have not previously built crowdwork applications because their experience in using them is still valuable in discussing the challenges associated with the development of crowdwork systems; furthermore, their comments provide an interesting contrast to experienced crowdwork developers.

Participants were recruited by local and remote collegiate connections. Their affiliations and positions range from graduate student to software engineer to director at various research institutions. The amount of professional experience ranged from 0 to 21 years (mean: 6.1 years); the range of total experience, including non-professional, amateur, and casual experience, was higher from 4 to 32 (mean: 12.5 years). Their specialty also ranged from software architectural design, to
scientific/numerical programming and methods, although a large percentage reported software architectural design and backend development.

3.1.5 Crowdwork Systems Used and Built. Participants reported using a variety of crowdwork systems, including OpenStreetMap, Wikipedia, Kickstarter/IndieGoGo, Mechanical Turk, and Crowdflower. Some participants argued for other consumer/commodity services such as Uber, GrubHub, and GitHub to be regarded as crowdwork systems.

The developers that previously built crowdwork systems provided a diverse range of information in response to the interview questions. One developer reported building a citizen-reporting web application called CoCoRaHS (Community Collaborative Rain, Hail and Snow Network), with networked participation across the United States and 19,000 active users (40,000 users across its over-15-year lifespan). Members of the network use low-cost measurement tools to predict precipitation across the country and report those predictions in the system. In a similar vein, a second developer built and maintained a citizen science web application called CitSci. CitSci allows scientists to create citizen-science projects from a multitude of scientific fields on its platform and helps to recruit interested parties in providing observations for them. According to the website, CitSci has hosted almost 300 projects by 2,000 members and recorded over 570,000 measurements across 61,000 locations.

Another participant—a researcher and a developer—is building a research crowdwork prototype in the form of an integrated development environment that uses
a specialized crowd to provide programming-related answers to developers. On the crisis computing side, there are three participants, each with varying levels of experience and background, who have collaborated with each other in the past on two systems—a crowdwork system for identifying, categorizing, and verifying crisis-related social media called MicroMappers, and a machine classification system called AIDR that uses results from MicroMappers to refine its processing. Finally, another (professional) developer had built a marketing and analytics platform for crowdsourcing reports and assessments of consumer and product goods found in supermarkets, restaurants, and other venues.

For the rest of the analysis, I place the participants in two disjoint groups—developers who have used but have not built crowdwork applications will be referred to as the CROWD-USE group, while ones who have used but also built crowdwork applications will be in the CROWD-BUILD group. All participants in CROWD-BUILD have used crowdwork apps in the past. I create these groups so that the analysis can speak to the insights generated from both. Figure 1 illustrates the amount of professional and total programming experience from both CROWD-USE and CROWD-BUILD participants in the study.
3.2 Interview Results

Participants answered an array of questions designed to elicit their background in crowdwork use and development. Below I synthesize the results from the interviews and brainstorming sessions to answer my research questions.

3.2.1 RQ1: Do developers design crowdwork systems differently than non-crowdwork systems? Participants were asked to answer the following question: “Do developers design crowdwork systems differently than non-crowdwork systems? If so, how?” I first directed this question to participants, and then I engaged in a brainstorming session with them to determine how they actually approached a crowdwork problem. The data from both points of interaction within the study, as well as follow-up questions from their background, revealed their perspective on whether crowdwork system development was indeed different from traditional (non-crowdwork) system development.
3.2.1 No Difference: When I asked participants this question, some of them responded by saying that crowdwork and non-crowdwork system development were not different at all, arguing that “everything goes through design—database, UI, all normal parts...” (CROWD-USE). Another participant from CROWD-USE echoed these opinions by saying, “they are the same — it’s a basic web application.” These comments suggest that participants not involved in crowdwork projects do not see how crowdwork may change or alter system development. However, their claims about what is common between these systems also partially validates their understanding of how to build crowdwork systems, because generally speaking, the ‘normal parts’—database, UI, web architecture—for building both kinds of systems are in fact the same. This speaks to the technical factors inherent in both types of systems. While it may be true that crowdwork and non-crowdwork systems address different audiences, tasks, domains, and issues, their implementations use common tools and techniques in diverse software engineering communities of practice.

3.2.1.2 Social and Technical Differences: What is particularly striking in some of the responses from this question was that the differences noted, if any, were not technical in nature but rather social, or concerning the relationship between humans and the system. One participant who helped construct the MicroMappers application agreed with commonalities between crowdwork and non-crowdwork systems, saying that the “coding is the same with a different theme”, and it’s “how you are using it (crowdwork). For me as an SE (software engineer), I don’t see much difference. For MM (Micromappers), it is for social good.” (CROWD-BUILD) A participant focusing
on big data research from CROWD-USE discussed differences: “workflow ‘A’ could be
done similarly with a crowdsourcing app...Infrastructure-wise, there might be some
similarities, but with crowdsourcing, the philosophy might be different.” Mentions of
“themes” or “philosophies” were not uncommon; they hint at varying perspectives on
constructing crowdwork systems that are non-technical in nature. These wider
concerns extend questions of technology choice to those that speak to the objective
that the crowd is trying to achieve. Though they may view these system development
efforts as similar, they also recognize that there are larger concerns at play.

To provide contrast in opinion, other participants in both CROWD-USE and
CROWD-BUILD were vocal about the non-functional differences between crowdwork
and non-crowdwork system development. They generally referred to specific
scenarios to demonstrate notions of visibility, progress, and system adoption. When
asked about these differences, one participant focusing in crisis research said:

Totally – users are going to have a different level of interaction than non-
crowdsourcing apps. Example: logging into a bank site, I don’t want to know
anyone else logged into their banking account. The level of transparency is
different. In crowdsourcing, I want to be aware of what has happened, things
that come before and after when I came in, and how they’re going to change
how I interact with the app. (CROWD-USE)

Crowdwork systems indicate a level of visibility and progress not seen in other
types of systems. There are, of course, features inherent within domains like banking
that are important to have from a user’s perspective, such as the opaqueness of
viewing personal bank accounts. This sense of awareness, however, seems to be a
first-class concern in crowdwork systems, where collective action is made visible by
the system to perform work on tasks and show progress of that work, which may
motivate how much work one worker does. This invokes work from theories of social translucence (Erickson & Kellogg, 2000), where transparency of activity provides accountability and trust between workers in the system, even if they may not be working collaboratively.

3.2.1.3 Non-Functional Concerns: Other participants spoke about other non-functional concerns, such as adoption, reward, and trust. One developer from CROWD-BUILD is responsible for managing a large, multi-institutional weather-reporting based crowdwork system that is largely based around a network of 40,000 volunteer-based observers across the United States. He spoke about the “chicken-and-egg problem” inherent in crowdwork systems, which refers to the irony that systems can only be useful when people use it, but people will not use it if there is not already an established user-base using it: “now that we’ve been around long enough for people to trust us...volunteers now donate to operating costs... As long as we focus on the data and the observers, we’ll keep getting funded.” This indicates that adoption and trust go hand-in-hand. The sense of reliability that the system brings to the crowd over time promotes deeper connections between workers and taskers (people who create tasks for the crowd to perform). The same participant also remarks that “having that thing (weather-reporting) to do at 7am to start the day is meaningful for observers,” suggesting a deeper relationship that is constructed as a result of maintaining the system and responding to the needs of users for a long period of time. This commitment, the participant believes, ultimately leads to adoption, a solution to
the chicken-and-egg problem, and in the context of crowdwork systems development, an issue that developers must consider (Anderson, 2015).

Participants also expressed concern about crowd motivation and how it differs from that of non-crowdwork systems. A CROWD-BUILD participant involved in the MicroMappers and AIDR crisis projects described the problem of motivation: “besides altruism, there is no real reward for the crowd. The user is donating spare cycles to help with these problems.” Essentially, this problem hinges upon the problem that the developer wants to solve using crowdwork, as well as the underlying domain for which the system will be developed. If altruism is absent, then there must be other means for which the crowd will perform tasking work. Kittur et al. distill the importance of motivation with approaches for incentivizing work, but they also call upon future work to match task design with appropriate crowd demographics and incentive structures so that the crowd is more willing to spend time on the crowdwork system (2013).

Sometimes motivation can be indicative of the organizational identity of the crowd. One participant asks, “Why would people want to volunteer?...This goes beyond altruism...it goes to the sense of community and identity.” (CROWD-BUILD) He believes that the success of the Micromappers crowdwork system is due in part to that intrinsic urge to be called for work, instilling a sense of responsibility to generate accurate data that can be used to save lives during a disaster event. Indeed, this is a core component of crisis crowdwork systems, but it may not scale out to other
domains. Crowdwork design needs to raise these identity-based questions in their own domains and use them to help map these concerns to features.

3.2.1.4 Non-Functional to Functional Features: Some participants spoke explicitly about how non-functional features can become functional ones. For example, one participant in CROWD-USE suggested that crowdwork systems “have to monitor what’s going on, (like) some sort of version history so that they can revert back, making everything available to the public. Everyone can look at it. Everybody can see the progress of the goal.” This participant suggests integrating versioning mechanisms to share the historical progress of the work that has been done by the crowd. Another participant—a CROWD-BUILD developer of a citizen science reporting platform—echoes this use-case by speaking about data provenance, or the capabilities with which the system offers historical traces of the data reported by the crowd. The notion of progress, as articulated earlier by several participants in CROWD-BUILD, is an important indicator of how much work there is and has been completed. That it can be translated to requirements by participants who have not yet built crowdwork systems shows that such features are important and translatable.

CROWD-BUILD participants in the crisis domain echoed other non-functional to functional mappings, such as recommending techniques to create motivation for crowdwork. This is important given that motivation is a hard problem in the absence of online altruistic behaviors during crisis events. In particular, they recommended that gamification be used, where reward systems such as status symbols and
reputation-based rankings can be used to extrinsically motivate the crowd to pursue more work. Public scoreboards in such systems promote activity and can create playful competition among crowd members and increase overall quality of work, provided that the tasks are designed properly.

However, this participant also cautioned against gamification techniques, adding that gamification can prove to be more harm than good by alienating newer members against feeling included and valued. Instead, he suggested that normification—a strategy for equalizing reward, capability, and action among a community—can be used to increase overall effectiveness of a crowd (Preist, et al., 2014). Indeed, strategies for motivating the crowd to perform work for a crowdwork system contrast well with non-crowdwork systems that do not necessarily require motivational strategies to work on a problem. By translating feedback like this to software requirements, developers can ensure that they are paying attention to the needs of the crowd.

3.2.1.5 Other Responses: Participants also described other differences between crowdwork and non-crowdwork development. They described the pain in creating crowdwork systems because of no crowd to test the system. “Crowd apps are hard to test—programmers need crowd automation in testing,” says a CROWD-BUILD participant. Crowd-testing would involve complexities such as simulating collaborative work, automating tasking through various sequences, and mocking reward and reputation rankings, to name a few. However, what crowd-testing fails to cover are the attitudes that the crowd would have in using the system under different
circumstances and using varying skillsets. In contrast, most non-crowdwork systems have the luxury of not needing to worry about user-testing in this capacity. For these systems, there is a small set of particular personas or archetypes around which design and development work focuses. For crowdwork, it is much more difficult to pin down who will be using the system.

In addition, a participant in CROWD-BUILD is planning to use crowdwork for a research project involving integrating crowdwork within integrated development environments for inline feedback and general Q&A/support (Chen, et al., 2016). He notes limitations in recruiting the right crowd for it:

Crowdsourcing app development is still in the beginning stages to get work done. It’s probably successful to collect data to prove some concept, but collective intelligence is lacking—some complicated tasks aren’t able to be solved. Mechanisms to parallelize work into discrete chunks is missing. What if you have a serializable or hierarchical job? How do you serve (these configurations) to the crowd?

This participant articulates some of the biggest challenges with respect to crowdwork research. Collective intelligence, as he notes, has only been observed as a bottom-up phenomenon and hasn’t yet been wielded by crowdwork techniques for particular scenarios. This is a worthy pursuit, especially as development efforts seek to provide a public space for effective and collaborative crowdwork for important societal problems.

3.2.1.6 Summary: Crowdwork systems development involves a wide range of social and technical issues to consider. Based upon the questions asked during the interview, it is evident that there is a significant difference in non-functional requirements between development for crowdwork and non-crowdwork systems. In
addition to the technical complexities that system development faces, including technology and architecture choice, crowdwork systems take on a host of non-functional issues that are critical to the success of existing systems evaluated by the participants in the study. These issues are sometimes social in nature, which implies human-to-human interactions between crowd members within the system, but for the most part, they extend beyond the technical issues that developers frequently face with traditional systems. They include notions of trust between the crowd and the system, motivations to incentivize crowdwork, organizational identity and structure, and adoption. Generally speaking, participants in the CROWD-BUILD group articulated more of these issues than the CROWD-USE group, but the CROWD-USE group still contributed important feedback to the mappings from non-functional to functional requirements (e.g., with respect to progress display and social awareness).

As the crowdwork community learns more about design implications for crowdwork systems and efforts to design them, it is increasingly important to derive and document these functional and non-functional requirements to make visible the important issues in crowdwork system design and how to construct them effectively.

3.2.2 RQ2: How do developers design crowdwork systems? What challenges do they address? The next research question examines how developers approach a solution to a real crowdwork problem. While the previous section described the differences between crowdwork and non-crowdwork systems, as well as highlighted crowdwork challenges, this question is designed to observe how participants engaged with a problem building on their previous crowdwork
experiences. It is important to note that due to time constraints with some of the interviews, the brainstorming session could not be performed with all participants. However, the data collected does highlight how members in both CROWD-USE and CROWD-BUILD designed a crowdwork solution to a particular real-world crowdwork problem.

3.2.2.1 The Brainstorming Problem: In most of the interviews, I presented a real-world crowdwork problem to the participant. This problem was articulated as follows: “imagine yourself as the developer behind a new crowdsourcing website for ranking images of devastating areas hit by a recent hurricane. These images are captured by the Civil Air Patrol (CAP), are in the thousands, and are unfortunately not geo-located or categorized in any way. With access to these images, work out how you would design this new website so that the crowd can rank these images productively.” As the session went on and the participants provided various solutions to the problem, I would then start adding more complexity, including a use-case for providing geo-location on the images, as well as providing verification of the crowdwork itself. This was done to ask how the participants would react to and design for these new use-cases, as well as how to design for future unknown use-cases.
Figure 2: Example whiteboard sessions with physically-present participants (both CROWD-USE).

In general, participants from both groups provided lots of front-end and back-end implementation details to achieve two goals: solve the original problem, and provide usability from the crowd's perspective on performing the task. This was surprising, given that usability was not part of the objective. Participants envisioned various ways to render tasks to the user, illustrating what the crowd would see from the imaginary system and what processing elements would exist in the backend. I decompose what information I was able to receive from the brainstorming sessions into high-level concerns below.

3.2.2.2 Focusing on the Crowd: Participants from both groups spent time thinking about how tasking is perceived by the crowd, treating the crowd as a valuable resource earned, rather than given. A professional developer from the CROWD-USE group illustrated what would happen on the system: “from (the) logged-
in user’s perspective, call (upon the crowd) for helping to find the worst places (to image-rank), inspire (the crowd) to help and spend time.” Another professional developer from the same group discouraged user-registration, “Keep (it) as simple as possible, no user accounts.”, later corroborated by a crisis researcher in the same group, who stressed that user-registration would dissuade ad-hoc crowdwork.

Participants from CROWD-BUILD, on the other hand, spoke more about crowd attention spans, arguing for the case of designing tasks such that they are quicker to perform and more digestible: “Worker mindset – users cannot do more than 10 (tasks) at a time.” A crisis system developer argued for a more systematic approach by recommending to measure request-response cycles because workers “aren’t very patient,” which suggests that by virtue of the experience these participants have in their own system development, it is important to measure task performance so that they can iterate on task design to increase crowd performance and productivity.

But another important factor brought up constantly by both participant groups was motivation. From comments about picking new images to show to new users (CROWD-USE) to particular messages like, “help us determine the amount of destruction by giving your opinion on these live images captured by the government” (CROWD-USE), motivation seems to be an important design issue for participants in both groups. A CROWD-BUILD developer stresses this point:

Know users — how they are reacting. As end-user, you have to look at the images. What will motivate me to keep coming back? We have to work on these issues of keeping the task fresh and interesting. Mix the images properly. Add some message to encourage and motivate.
Most participants recommended putting up a motivational message to encourage workers to keep working, because they feel that in this domain of crisis response, workers are spending free cycles to perform tasks without compensation. That they recognized and dealt with this issue during the brainstorming sessions, even if some of them had no experience in developing crowdwork systems, indicates that motivation is an important property of crowdwork system design.

3.2.2.3 Progress: Surprisingly, only a few participants discussed issues of progress, and most of them came from the CROWD-USE group. One professional developer in this group invoked his own personal volunteering efforts to describe the importance of achieving a sense of progress in his work, “When you do volunteer work, you’re not expecting something back, but you still want to see the outcome. For example, when you work at the salvation army making food for people, you want to see people come by. Are they happier? Are they gaining weight? Have the number of people increased?” By analogy, this developer illustrates that crowd workers need to feel a sense of accomplishment in the bits of work that they performed. This is corroborated by other crowdwork systems that display progress to workers to share and spread social signals of accomplishment (Barrenechea, et al., 2015a; LaToza, et al., 2014).

Technical suggestions to display progress were made, including a progress bar at the top of the tasking screen to show that the particular user had performed ranking for X out of Y images. For the geolocation task, another participant from the CROWD-USE group recommended showing progress through density renderings on
a map of the affected area hit by the hurricane. The denser the area, the more work has been performed on it. These and other suggestions were made to allow the crowd to both view how much work remains and motivate them to keep working on the task.

3.2.2.4 Task Sequencing: Task sequencing refers to the programmatic arrangement or configuration of crowdwork tasks. There were not many instances of participants noting this feature, though sometimes the brainstorming problem provoked some task re-organization as different types of tasks were added and participants felt the urge to come up with ways to arrange them. Participants who did focus on sequencing did so by explicitly describing order: “provide crowdsourcing workflow flexibility—ask user to rank images first—and then give the option to geotag the particular image that was ranked.” (CROWD-BUILD) This participant provided an order for performing these tasks, which is a common issue discussed for workflow design (Kittur, et al., 2013). Another participant in CROWD-USE suggested that geolocation-based tasks should be performed only if crowd members ranked accurately (i.e. ranked in accordance with other results for that image), which implies some condition imposed on the server that guards the geolocation task. Faced with three tasks (ranking, geolocation, and image verification), another CROWD-BUILD developer provided the following order: verification, ranking, and then geolocation, his reasoning being that only verified images should be shown for ranking or geolocation, otherwise it is wasting workers’ time.

3.2.2.5 Gamification: Approaches to use gamification techniques were sparse among participants, though noteworthy. To retain crowd workers in performing more
continuous work, a professional developer in the CROWD-USE group recommended to treat task assignments between ranking, geo-locating, and verification like a trivia game: “Turn [tasks] into a quiz/trivia game: ‘do you want to test your geolocation or ranking skills? More exciting and challenging. Instead of ‘just come help’, make it a game.” His feedback turned into an abstraction applicable across domains that emerged from his brainstorming session, as he points out: “These tasks become options, now choose an order of them—call it a campaign—to be displayed. At the end, we might have a reward or something. A ‘campaign’ has many tasks (to complete). This can be applied to anything: a test, quiz, an ad, a form, whatever you want. It’s this bigger thing that can be applied to anything real-life.” What is being described here is turning task sequencing into a gamification technique and discovering ways to entice the crowd with ways to stay motivated through a sequence of crowd tasks. Gamification can indeed become a tool for developing crowdwork, increasing motivation and playfulness, as well as controlling the sequence of tasks to serve to the crowd.

3.2.2.6 Summary: Overall, participants performed the brainstorming session efficiently and with enthusiasm. Some participants engaged with more technical matters of creating a crowdwork system, which created parallels with traditional system development (i.e. creating database tables for users and tasks), but most participants—especially from the CROWD-BUILD group—wrestled with crowdwork issues such as crowd-centered design concerns, motivation, and task sequencing.
3.2.3 RQ3: What features are desirable for developing crowdwork systems? What kinds of abstractions/techniques or frameworks/tools are used by developers to create crowdwork systems? To answer the last research question, I asked participants about their opinions on crowdwork frameworks, specifically about the features that they look for, as well as whether they would use them in their own crowdwork projects (if they haven’t already). Not surprisingly, participants gave a wide variety of answers, but what was central to the discussion was the parallels that were drawn to previously identified challenges that they faced when using or building crowdwork systems. When asked if using a crowdwork framework would be considered when building their next crowdwork app, 9 said ‘Yes’ while 3 said ‘No’. I discuss the varying explanations given by participants in the following sections.

3.2.3.1 No Framework: When asked the question about whether or not crowdwork frameworks would be useful, some participants from both CROWD-USE and CROWD-BUILD said that either they “would not think to look for a crowdsourcing framework” (CROWD-USE) or that the “complexity of using a framework is not worth it.” (CROWD-BUILD) The reasons for these answers from both groups is worth exploring. As a professional developer, the CROWD-USE participant believes that placing more dependence upon third-party libraries for software projects causes potential problems in the future. This is a valid concern, given that most software projects must constantly keep pace with issues, updates, and deprecations with libraries that solve technical issues, such as object-relational
mapping, cryptographic storage (for passwords), pagination and tagging tables, to name a few.

The same participant’s second reason, given that his concern does not necessarily map directly to crowdwork concerns, is that crowdwork as a technique is “too hard to abstract that it would be applicable to most crowdsourcing. Same model that Github uses doesn’t apply to Teachermaps. It’s a losing battle of software engineering.” Here, this developer is describing the challenges that crowdwork faces with regards to becoming a repeatable pattern to cope with the complexities of crowdwork. With respect to solving the crowdwork problem during the brainstorming session, he identified workflow A/B testing, task states, and task sequencing as some of the issues that would be hard to abstract into a framework: “really [difficult] to evaluate efficacy of workflows and tasks, because there are so many variables. Depends on where in the world this event happens, how frequently, death toll, population count of affected area, etc.” This highlights the difficulty in methodically examining and testing out varying crowdwork workflows and tasks under different crowds and circumstances (e.g. different disaster events), exacerbating the challenges that crowdwork framework designers face to provide support in this area.

Other concerns about new crowdwork frameworks were given. A participant from CROWD-BUILD shared a neutral stance, offering that crowdwork frameworks would not be used in particular situations: “depends on the difficulty of task, type of request, budget, and personal experience with the crowd framework. Some people have had bad experience with using frameworks, so they go by themselves.” This
speaks to the fragmentation of use for varying development styles, and this researcher shares a common concern about using new frameworks that increase the learning curve and in development and deployment complexity. Another crowdwork system developer (CROWD-BUILD) likened crowdwork frameworks to content management systems, saying that he’s “hesitant to use them. They tie my hands as a developer.”

It is quite surprising to reveal that a fair number of developers interviewed would not choose to use these type of frameworks, given that they have identified crowdwork challenges in their own experiences. However, to properly contrast this feedback with that coming from participants who would be open to use a crowdwork framework, I discuss several properties that they have claimed are important to see in such frameworks.

**3.2.3.2 Task Sequencing:** During the brainstorming session, participants found the need to sequence tasks as more of them were being introduced within the problem. When asked about crowdwork framework features, two developers from CROWD-USE and two developers from CROWD-BUILD thought that task sequencing would be a desirable feature to have. They recommended having options to weigh tasks at times when, through some randomized shuffle, some tasks would be picked more often than others. Current frameworks like PyBossa offer this sequencing, and one CROWD-BUILD developer that uses PyBossa also mentioned its usefulness.

**3.2.3.3 Human-machine computation:** But there are other desirable factors that involve task sequencing at a higher level of abstraction. One professional
developer from CROWD-BUILD noted his stance of rejecting the use of a crowdwork framework was based on previous frustrations with integrating them into projects, but he also discussed features that were missing: “The framework needs to be able to take work units and redistribute them. There are things that are good to send to virtual assistants, MTurk, and individuals in the field. There are different categories of workers (to use), and there are good things to send out to machines as well. Intelligent routing to the system that is most appropriate is needed.” This suggests that crowdwork frameworks need mechanisms to route proper work to the crowd, as well as to other machines that can process that work, which reinforces the importance of having crowdwork systems integrate with computational elements more closely, thereby becoming human-machine computation systems. This theme is prevalent across crowdwork developers who recognize the limitations of using crowd and machine processing and would like to combine them for better quality and more accurate work.

3.2.3.4 API Control: One CROWD-BUILD participant revealed an unusual use case with respect to controlling crowd workflows and tasks—API control. This participant illustrates an example where having remote control over an application’s crowdwork configurations indicates programmatic control:

If I need to control the app, there are two ways: one is to go to the app to adjust some settings/reconfigure, which is doable. Another way is to control the app from the outside (from a different app). If I’m able to adjust the app through APIs and during some live deployment, I think that will be of great use, because I may change strategies based on the actual application, things like batch size and other parameters. If we have to pause the app to adjust these things and come back again in another app to resume the workflow, this is a hassle.
This developer wants to be able to reconfigure the crowdwork settings of a target application dynamically so that manual control isn’t always necessary. This yields future use cases where crowdwork configurations can be updated based upon remote procedure calls from other applications and clients, implying that a crowdwork framework would be able to support a remote API that exposes these administrative controls to authenticated clients. This could be helpful when, for example, the crowdwork results sent to another system cause the crowdwork system to change its task and workflow configuration.

3.2.3.5 Quality of Work: Another participant expressed concern over the extent to which the crowd can solve problems accurately. Faced with not having gold standards against which crowdwork results can be tested, this architect from CROWD-BUILD invoked his experiences dealing with crowdwork results under uncertainty: “feedback loops—bronze standard—still provide feedback to the crowd when you don’t know the answer. The platform generates artificial gold to the crowd.” Here, the architect is describing the need to prime the crowd in knowing how best to perform the work, even when there are no perfect results to compare against. Therefore, the “artificial gold” essentially shows how best to perform the work to maintain quality of work standards. For instance, example tasks can be served to each user before trying to perform real tasks so that a user knows how best to perform the work. Zooniverse, GalaxyZoo, MicroMappers, and other popular crowdwork platforms already use bronze standards. Several participants would like to see
crowdwork frameworks that accept bronze standards and feed them to the crowd for particular tasks.

3.2.3.6 Versatility: The most common topic discussed in answering this question was the extent to which a crowdwork framework could support a multitude of use cases. Since crowdwork development can be at times unpredictable with respect to how it will be used and how much work will actually get done, some participants said they would use such a framework only if it were able to adapt to system and crowd changes over time. This is what I define in this study as versatility. Versatility is the capacity with which a tool/framework can support a multitude of use-cases. For the purposes of this study, I use versatility and flexibility interchangeably, since participants used the word ‘flexibility’ to mean the same thing.

One crisis developer in CROWD-BUILD already uses a crowdwork framework called PyBossa for the MicroMappers system. She claims that “PyBossa is a one-size-fits-all solution for users and programmers. Get the code, change it based on your needs, update it as needed.” PyBossa is a mature heavyweight framework that enables both programmers and non-programmers to quickly bootstrap a crowdwork project without hassle. She claims that PyBossa can fully support most crowdwork-based use cases; if developers need to customize the look-and-feel or change tasks, they are able to do that. She likes the flexibility that is offered through PyBossa and suggests that other crowdwork frameworks must offer the same level of flexibility.

One crisis developer from CROWD-BUILD defines flexibility as desirable: “(I would) definitely pick a flexible crowdsourcing framework. Flexibility means that
(the) framework can be attached to other applications. Communication through it should be through APIs, otherwise...it limits its usability.” This references a previous use case that this developer referred to about remote API control, but at the same time, this suggests that the framework afford local API control as well. In other words, a crowdwork framework should have an API that allows application logic to control and direct workflow and task management, progress display, bronze/gold standard templates, task sequencing, etc. The same participant gives an example of what such a framework can provide:

Flexibility to define easy and complex tasks. Example: an easy task is our task (given a Twitter message, choose a category that best fits the message). A complex task could [be to] collaboratively design some new part of some document/article by controlling different levels of workers. Some workers are only allowed to write, and once they do, they cannot edit/delete. Some other workers can delete, some can edit, etc. The framework should be flexible enough to design as complex of tasks as one can think.

This exemplifies the capabilities of a desirable crowdwork framework, one in which complex crowdwork can be achieved through the careful design of crowd workflows and tasks. Permission controls and authentication mechanisms should be available, as well as defining an organizational structure that examines the crowdwork by decomposing it, not unlike the dynamic decompositions of tasks by workers within the CrowdForge framework (Kittur, et al., 2011). By virtue of providing these configurations, a framework increases its likelihood of being adopted within the development environments of varying crowdwork projects.

Versatility emerges as a key concern for many participants. A big data researcher from CROWD-USE described a favorable feature for crowdwork
frameworks as “Lego-piecing”, or “remove things that don’t need to be there for some cases, and add pieces for some other cases.” Another participant who is a professional developer from CROWD-USE stressed, “You don’t want to reinvent the wheel, but you don’t want something that’s massively complicated or hard to setup. You want something simple, usable, easy to plug-in. if it’s already there, well-documented, the average (common) developer would choose it.” These comments suggest that the piece-ability of frameworks is a generally important factor for tool adoption. Since there is a wide variety of use-cases to support crowdwork system development, frameworks that do not impose upon technology choice, require much configuration, or over-serve features that cannot be plugged-in, have a better chance of being used.

3.2.3.7 Summary: Based upon the plethora of concerns, comments, and suggestions from participants in both groups, crowdwork frameworks are difficult to adopt, yet there are still unaddressed use cases that can enable developers to more productively design crowdwork systems using them. These serve as implications for designing better, more adoptable crowdwork frameworks in the future.

Some of the most important features that developers are looking for in these frameworks include the versatility with which they can integrate into a variety of projects and use-cases. Task sequencing can aid development in declaring arrangements of tasks and workflows and serve them at runtime. API control can enable programmatic control over crowdwork from other clients and systems. Generating artificial gold or bronze standards to prepare the crowd for upcoming
work is also desirable. These are only some of the features desired by the participants for crowdwork frameworks.

The design space of crowdwork frameworks is not sparse; there are many different frameworks that developers can use. As participants noted, some frameworks lend a heavy hand to development efforts, but so much so that they require a lot of configuration or restrict the architectural and technological decisions that developers make in the development environment. Others focus on dynamic tasking, workflow design, and particular domains, yet they are more obscure and sometimes are not designed to be integrated into development projects. I speak at length about some of these crowdwork frameworks in Chapter 3.

3.3 Discussion

Through semi-structured inquiry and brainstorming, this study reveals insights about crowdwork development experience across a variety of application domains. It shares collective opinions about what differences there are between crowdwork and non-crowdwork-based software development, how developers approach a crowdwork problem, and how developers regard crowdwork frameworks and their features. This has implications on two fronts: one, to synthesize non-functional concerns raised about crowdwork development, and two, to use them to raise design implications for future crowdwork frameworks and their role in solving these concrete issues. In the context of this dissertation, the study functions not just
to reveal these concerns but also to raise one key concern among them—versatility—to inform the design of the Crowdrouter framework.

3.3.1 The Study Participants. One might argue that CROWD-USE participants were not needed in the study. However, by virtue of interviewing participants from both groups, I was then able to reveal some contrasts that sprung from differences in development experiences and differences in the types of crowdwork challenges each developer had encountered.

To start, the differences between the CROWD-USE and CROWD-BUILD participants were noticeable based upon their answers to crowdwork-related questions. In general, CROWD-USE developers thought that there were little to no differences between crowdwork and non-crowdwork systems, and they focused more on the technical challenges that crowdwork (and non-crowdwork) systems bring to bear on such problems like the one during the brainstorming session. They also recommended much more human-machine computation to solve the crowdwork problem, rather than just letting the crowd perform all of the work. In terms of whether or not these developers would use a crowdwork framework, they would be equally as likely to pick one up as developers from CROWD-BUILD.

On the other hand, CROWD-BUILD developers and researchers believe there were differences between both types of systems and spent more time discussing the higher-level non-functional concerns that made them different. They focused more on crowd concerns, including motivation, progress, and organizational structure and identity. During the brainstorming problem, they did focus on technical challenges of
the problem, but their design approaches were primarily focused on the crowd, not the application. Some developers in this group have built human-machine computation systems, therefore both groups spoke at length about the aggregation of crowdwork and machine processing. With respect to crowdwork frameworks, they would make use of them; indeed, some participants are using them in current projects.

Both groups shared many commonalities as well. Some developers from CROWD-USE are in-tune with crowdwork issues, and so their feedback during the interview indicated that crowdwork was indeed a host of functional and non-functional concerns. Conversely, there were developers in CROWD-BUILD that were not necessarily versed in known crowdwork issues, or had spent more time administrating and maintaining crowdwork systems as opposed to building them. This may imply that their perspectives on crowdwork may not be very different than those of non-crowdwork. Regardless, participants from both groups shared varying perspectives on the nature of crowdwork and framework design in support of future crowdwork systems.

3.3.2 Crowdwork Design Issues. My study attempts to glean crowdwork issues generated by the participants’ experience in use and development and articulate what they are. Note that there might be more features to outline here, but based upon the results of the study, these are the issues that surfaced as most salient.
1. **Progress and Work Visibility**: Members of the crowd want to know what work is available to perform and how it will make an impact/difference in the context of solving the problem at hand.

2. **Motivation**: Crowdwork system design needs to understand the resources and roles being allocated by the crowd to solve the problem.

3. **Human-Machine Feedback Loops**: Crowd and machine processing are powerful on their own, but they can also complement each other if integrated properly. Crowdwork systems need to explicitly define how crowd processing informs machine processing, and vice versa.

Progress and work visibility illustrates the extent to which crowdwork systems provide transparency for work to be completed by the crowd and the progress made by it. Most participants emphasized this importance in their own experiences, turning to methods for creating bronze and gold standards, creating chat channels among users, and providing awareness of others’ work and accountability towards crowdwork within the system.

The second factor relates to motivational and trust-related concerns for crowdwork. Crowdwork systems inherently ask a wide user base to perform work; without the crowd, the system does not achieve its objective. Therefore, a system blind to the needs of the crowd will not fare well in adoption and retention of work. Participants suggested techniques such as gamification and normification to address motivational challenges. Gamification can provide playful competition and
scoreboarding to tasks available within the system; normification instead focuses on the collective work of a crowd as a community and does not alienate newcomers by creating rankings or reputation-based systems.

Finally, human-machine computation became a central theme throughout the interviews and brainstorming sessions. As was revealed through the brainstorming session, participants from both CROWD-USE and CROWD-BUILD found that using the combined efforts of human tasking and machine computation can produce dramatic optimizations for large, wicked problems that either class cannot perform alone. Many participants also emphasized the similarity between using humans and machines to perform work. This is very much within the realm of future perspectives of crowdwork (Kittur, et al., 2013), but there are also alternative views for treating crowdwork as a community, rather than a resource (Starbird & Palen, 2011; Starbird, 2012a). Other participants stressed this distinction as well. Still, human-machine computation is an emerging topic within the crowdwork and artificial intelligence fields (Quinn & Bederson, 2009; Quinn & Bederson, 2011), and many approaches to human-machine computation remain untapped. Crowdwork systems should approach crowd and machine-based processing with reciprocity in mind; both the humans and machines should be accounted for, in terms of their needs and expectations.

I believe that there may be more crowdwork system design issues to explore. Synthesizing these issues and bringing them to the surface will hopefully provide
guidance to future crowdwork developers in thinking more critically about the role that humans play within computing.

3.3.3 Crowdwork Framework Design Implications. It is critical that crowdwork framework design learns from the insights gathered from developers who, based upon their experience using and building crowdwork systems, may consider such frameworks in the future. In addition, while there were participants who noted that they would not use these frameworks in the future, it is important to highlight the difficulties in adoption and to address them.

Features of crowdwork frameworks that were requested by developers in the study included the need to manage tasks and workflows efficiently. This refers back to brainstorming session, where developers were handling ways of sequencing and providing criteria upon crowdwork tasks. Developers from both groups expressed the need to weight, prioritize, and shuffle tasks as their crowdwork design solutions incorporated aspects of task manipulation based upon the accuracy with which the crowd would perform them. This also speaks to the level of workflow support that is needed in order to serve and process tasks according to specified criteria by the developer. Frameworks like PyBoss and Hive do offer some of these facilities, but in general, there is more work to be done to design frameworks such that they provide the task and workflow management features that developers need to construct crowdwork systems productively.

However, chief among the concerns surrounding the use of crowdwork frameworks is their versatility. Most developers from the study indicated that they
would not consider using such frameworks because of their obstinate and sluggish integration within development environments. By obstinate, I mean that existing frameworks forcefully require that crowdwork projects use pre-specified technologies to have the framework provide the use cases that users might find helpful. By sluggish, I mean that the complexities involved in configuring the framework to be used within the context of the project is often inflexible, time-consuming, or obscure. This speaks to the concern that frameworks are not versatile, highlighted by many participants who have used them in previous projects.

The goal for crowdwork framework design, then, is to approach the needs of crowdwork developers with an eye towards increasing versatility in providing various features that are desired. If frameworks are designed with this in mind, as well as offer desirable features like task and workflow management, API control, and creating bronze/gold standards, they increase the likelihood for developers to adopt them into their crowdwork projects.

3.4 Summary

This crowdwork study serves to capture insights about the sociotechnical challenges faced by developers who have used and built crowdwork systems. In one vein, developers articulated these challenges with respect to experiencing or designing crowdwork, and it is important to capture these insights from the outside (i.e. as a member of the crowd) and the inside (i.e. as the crowdwork developer). In another vein, however, and as a focal point for this dissertation, developers
articulated challenges in adopting frameworks for helping to construct crowdwork systems in the future. This set of data is critical to help create a new generation of frameworks that provide the desirable features that were articulated by these developers, as well as offer such features in a versatile manner.

In the context of this dissertation, then, my goal is to focus on versatility as the key requirement for the construction of a new crowdwork framework called the Crowdrouter. I design the Crowdrouter to address challenges of versatility echoed from developers expressing frustration in adopting crowdwork frameworks. This will increase the likelihood of developers adopting versatile frameworks like the Crowdrouter in constructing crowdwork systems in the future. Across the spectrum of existing crowdwork frameworks and tools, I situate the Crowdrouter as a lightweight framework, one that can be used in many different crowdwork contexts and avoids becoming an obstinate or sluggish tool, as is the case with other frameworks discussed. I now turn to speak about its design and features in the next chapter.
CHAPTER 4

CROWDROUTER

In this chapter, I introduce the Crowdrouter, a lightweight web development framework for architecting tasks and workflows for crowdwork systems. Unlike other crowdsourcing frameworks that provide boilerplate code to construct and manage workflows and tasks, the Crowdrouter aids the developer directly in designing them as first-class entities, enabling more control over how and when they are invoked within a crowdwork application. Developers can create workflows as pipelines through which tasks are served to the crowd in proper order and without interruption; they can swap out an old task with a new one without hassle; and they can gather crowd statistics over time, including GET and POST request distributions across workflows and tasks. Crowdrouter provides additional features as well to developers building crowdwork applications.

First, I illustrate the architectural design of the Crowdrouter, defining the important components that work together to provide the level of versatility being offered in the tool. Then, I describe its features and show how they can solve common use-cases when constructing crowdwork applications. Finally, I outline use-cases that the Crowdrouter cannot yet satisfy but also provide the necessary implementation steps to provide that functionality in the future.
4.1 Architecture

4.1.1 Core Entities. At its core, the Crowdrouter provides three key classes, each responsible for carrying out important duties to provide versatile and extensible crowdwork programming. I define them as follows:

**CrowdRouter**: A manager class which serves as the main point of invocation when using the framework. The chief method—`route()`—is used to begin routing requests through to a specified task. The CrowdRouter class allows the developer to declare which WorkFlows are capable of being performed through it. This creates a one-to-many relation between a CrowdRouter and a WorkFlow.

**WorkFlow**: A class that is responsible for managing tasks. Each WorkFlow class contains a declaration of which Task classes are executable upon calling its `run()` method, which exposes a template in which programmers can place logic to call specific Tasks under specified conditions.

**Task**: A class that represents a unit of work that the crowd must perform. Each Task class declares a `get()` and `post()` function, synonymous to two HTTP endpoints for retrieving and processing work for a task.

In order to prevent confusion, I will keep to the capitalized names of these components when I am describing their functionalities, while I reserve the names “workflow” and “task” to refer to the canonical crowdwork elements within an application. I will also make a distinction between “Crowdrouter”, which refers to the
developer framework in this dissertation, and “CrowdRouter”, the manager class defined above.

![CrowdRouter Entity-Relationship Diagram](image)

**Figure 3: Crowdrouter Entity-Relationship Diagram.**

Figure 3 shows the relationships behind these core classes. At a high level, the CrowdRouter class can have many WorkFlow classes, and each WorkFlow can have many Task classes. By default, the Crowdrouter tool comes with super types of these three classes—the AbstractCrowdRouter, the AbstractWorkFlow, and the AbstractTask. These three classes become inheritable, not only to provide developers with a wealth of functionality easily obtainable by creating concrete subclasses, but also to create a familiar yet powerful language for designing crowdwork in a flexible and usable fashion.
Figure 4: Example Crowdrouter entities.

Figure 4 shows an example implementation of these entities as concrete subclasses. The Crowdrouter class called MyCrowdRouter is a concrete subclass of the AbstractCrowdRouter class and declares two WorkFlow classes—ReportingWorkFlow and MatchingWorkFlow—that it can route requests to using the route() function. The ReportingWorkFlow then declares two Task classes—GeoTask and RankTask—as tasks that it can execute using the run() function. Each Task class declares two method implementations—get() and post()—for GET and
POST operations, respectively. During runtime, an instance of MyCrowdRouter is used to call its `route()` function, which will prepare workflow and task execution with packaged request and session data. If, for example, the ReportingWorkFlow and the GeoTask were invoked as a GET request, the ReportingWorkFlow will freely execute the `get()` function of GeoTask and return control flow back to the receiver. The receiver—in this case, a controller handling the GET request—then can use the path variable generated by the GeoTask to render a template back to the client.

```python
30 ... 31. def geo_task(request):
32    response = my_crowdrouter.route(ReportingWorkFlow, GeoTask, request, session)
33    return response["path"]
34 ...
```

**Figure 5: Example CrowdRouter route call.**

Figure 5 shows an example invocation of the MyCrowdRouter instance. It is simple enough to specify the WorkFlow and Task classes to which the request will be routed, and to receive the response after task execution has finished. The astute reader will question why the function signature of MyCrowdRouter’s `route(workflow, crowd_request)` definition is different from the function call at line 32 on Figure 5 `(workflow, task, request, and session)`. This is due to the decorators being used in these classes—`@crowdrouter`, `@workflow`, and `@task`. To properly provide templates on which custom programmer implementations can be placed while executing framework-specific code around it, these decorators must be declared atop the chief functions for each class. In this case, when the MyCrowdRouter `route()` function is called, the `@crowdrouter` decorator is first
called with the parameters passed in at the invocation site. This decorator has framework logic to properly wrap the request with metadata for proper routing and encapsulation (see Section 4.1.1 on CrowdRequests and CrowdResponses for more details) before calling the decorated function. For example, the run() method for ReportingWorkFlow will be invoked by MyCrowdRouter with a crowd_request variable (line 6 in Fig. 4) when it actually receives two variables—crowd_request and task—(line 15 in Fig. 4) before it executes during runtime due to the @workflow decorator.

Developers using the Crowdrouter tool can declare as many CrowdRouter, WorkFlow, and Task classes as needed to cover all of the work that needs to be done in their crowdwork application. If the Crowdrouter is being integrated into an existing crowdwork application, then it is best to refactor task code into Task classes, since the Task API behaves much like a RESTful API. This implies removing task code from HTTP controller endpoints, where client-to-server code is typically located. Then, the developer must ask, “how and when do the application’s tasks relate to each other?” because this directly influences how Task classes are declared within WorkFlow classes.

Declaring Task classes within a WorkFlow class does not weight, rank, or order them in any particular way (unless the pipelining feature is being used; see Section 4.2.2 for more details). By default, if an instance of a Workflow receives a request to execute a Task, it will do so, as long as that Task has been declared in the WorkFlow class (e.g., see line 9 in Fig. 4). Of course, if the developer so chooses, she can exercise
complete control over the order of Tasks by pipelining or by adding code that determines ordering inside of the Workflow subclass; both of these techniques are discussed below. Having identified tasks and assigned them to workflows, the last high-level task for a user of the CrowdRouter framework is to assign WorkFlow classes to a CrowdRouter class (e.g. see line 2 of Fig. 4). This CrowdRouter instance will then take over the routing of all HTTP requests that involve workflows and tasks within a crowdwork application. The crowdwork application would still need to route HTTP requests involving other aspects of its functionality separately. Note: a single instance of the CrowdRouter class should be sufficient for most applications—the primary units of concern for a crowdwork application are workflows and tasks—however advanced applications that might be hosting multiple crowdwork systems on a single site can take advantage of instantiating multiple instances of the CrowdRouter class and use them to keep the workflows and tasks of the multiple crowdwork sites properly separated.

4.1.2 CrowdRequests and CrowdResponses. In order to provide the versatility that developers need in a massive online landscape of heterogeneous and sometimes redundant toolsets, the Crowdrouter needs to be accessible through diverse web frameworks of choice. Unfortunately, not all frameworks are created equally; the HTTP request and response objects vary by type, location, and behavior, making it difficult to create a simple solution to route and transform an HTTP request to a response that must be returned to the client. For example, in the popular micro-web framework Flask, the session object comes from a HTTP library called werkzeug,
while in another popular, yet heavyweight, framework called Django, the session object is of a different type—a custom SessionStore class internally made for Django projects. Despite similar attributes in HTTP request and response objects from one framework to another—such as path, method, and form data—their behaviors make it tricky to parameterize so that the Crowdrouter knows how to access those attributes properly.

To solve this problem, the framework uses CrowdRequest and CrowdResponse objects to encapsulate differences in HTTP request and response libraries that appear in any Python web framework, respectively. For instance, to handle the variances in request objects, the framework uses an AbstractRequestStrategy class to provide a unified interface for retrieving request attributes against which the rest of the framework can query. It achieves this by invoking an AbstractRequestStrategy subclass to deal with its syntax and semantics, such as DjangoRequestStrategy or FlaskRequestStrategy. The CrowdRequest then wraps this strategy with framework-specific metadata, such as which WorkFlow and Task class is being executed, and then it gets threaded through the rest of the framework. The CrowdResponse acts similarly to handle differences in response structures. During runtime, these objects are passed through the initialization of a Task and WorkFlow class; without them, the Crowdrouter framework cannot function properly. Note, these new classes are shown in an updated UML diagram in Fig. 11 below.
4.2 Features

The Crowdrouter is a tool that provides an additional layer of programming abstraction to support design and management of workflows and tasks. Together, workflows and tasks build upon the structure of crowdwork, and much work in social computing has examined how best to design and serve these crowdwork elements (Kittur et al., 2013; Bernstein, et al., 2010). By promoting these elements as first-class entities, the Crowdrouter can expose a rich set of functionality to manipulate request-response flow, encapsulate task and workflow-based information, and hide their implementations from the rest of the system. This, in turn, allows developers to focus on the macro-picture of their crowdwork application. I outline some of the benefits that fall out from the design of the framework.

4.2.1 Effective Routing. In the traditional style of web development, applications serve RESTful endpoints that translate a URI into a controller action, whereupon some functionality—retrieving user profile information, processing form data, etc.—is performed on the server and the results are sent back to the client. In crowdwork applications (e.g. EmergencyPetMatcher (Barrenechea, et al., 2015a)), those endpoints represent the RESTful operations for accessing and performing crowdwork tasks. Unfortunately, there is not much support in reusing these endpoints or their actions. Controller actions may have duplicate code that performs logic to serve or process a task, or sometimes developers want to add new tasks or substitute existing tasks and must write new endpoints to handle them. Some of these tasks may share similar operations behind the scenes, thus dangerously
duplicating work and code across many sections of a web application. Such duplication cries out for refactoring and the CrowdRouter framework provides a natural target for this type of refactoring for crowdwork applications. By migrating task-specific code into separate Tasks, one isolates code for a single Task in one location and removes any duplication that came from “cut-and-paste” actions when the tasks were being created. Similarly, any code that is responsible for controlling how and when a Task is executed, now has a natural home in the Workflow class and no longer has to be hidden inside of the controllers of crowdwork applications. Again, the work of refactoring this code into individual WorkFlow classes has good potential for eliminating “cut-and-paste” duplication that can accumulate in controller-related code.

**Figure 6: Traditional Routing vs. Crowd Routing.**

Using the Crowdrouter, endpoints still represent tasking operations, but instead of placing their implementations directly onto controller actions, they are
stowed away in a Task class that represents a crowdsourcing task to serve. Figure 6 shows the contrast in endpoint-to-task design between traditional routing and routing using the Crowdrouter. By leveraging this level of indirection, the Crowdrouter enables routing for various use-cases, including one endpoint per task (as opposed to two endpoints, one for GET and another for POST), one endpoint per workflow (as in the case of pipelining, discussed below), or even one endpoint to handle all crowdsourcing tasks for the entire application. This last case could be made possible if (for example) all task and workflow classes were persisted in a database and mapped to an ID; a Crowdrouter route() call could then use a parameterized endpoint to fetch Task and Workflow instances through such IDs.

For example, consider an application using the crowd to rank damage severity for satellite/aerial imagery taken after a hurricane. There might be various HTML templates used to rank one image at a time, such as providing crowd-input tags on a text field below the image, or using radio buttons to provide a 1-5 rating scale. There may be a desire to test both Tasks and randomly serve them against the crowd, monitoring which template performs better than the other. Regardless, the task retrieval logic remains the same, while the templates for representing the task may differ. In a traditional application, endpoints would typically be mapped to each task that the crowd must perform, sometimes duplicating code for retrieving images for each task. However, with the routing capabilities offered by the Crowdrouter, a developer can map tasks to specific endpoints without duplication, since task retrieval code is encapsulated within a Task class that only varies in which template
to present to the client. In this manner, all crowdsourcing-based controller actions are mapped by the Crowdrouter to point to available Task classes for task-based implementations. This separation of concerns achieves higher flexibility in routing calls to tasks without having to generate additional endpoints on the server.

4.2.2 Pipelining. One of the core features offered by the Crowdrouter is the ability to string together task executions without interruption. This sequencing, called pipelining, is useful in cases where the application is bringing the crowd through a series of tasks that logically flow together and in a particular order, such as human-intelligence tasks (HITs) created in Mechanical Turk, image or geo-clicking tasks in Micromappers, or in non-crowd-based scenarios, online payment and billing workflows. These pipelines proceed uninterrupted because of the criticality of following ordered steps, but sometimes they branch to steps not normally taken due to errors or the need to gather additional information based on prior user input. Often, pipelining is replicated by hard-coding a redirection to a future task within a controller action after the current task has been served and processed, enabling the application to call another endpoint that serves that future task. Redirects can thus be used to bring clients through a sequence of actions that represent a workflow.

However, pipelining is difficult to scale. How can a task be made available through multiple workflows within the system? What components need to change when a task needs to be substituted, removed, or added? How does a workflow make use of a pipeline, yet have the flexibility to branch based on user input? To solve these challenges, the Crowdrouter makes use of the WorkFlow class, which not only
provides a template that the developer can use to create task execution strategies, but also offers pipelining capabilities via that template. For example, if a developer declared a WorkFlow class called MyWorkFlow and had three Task classes declared under it (ex. TaskA, TaskB, and TaskC), then it just takes one line of code within the MyWorkFlow class to properly sequence these Task classes together upon execution:

```python
@workflow
def run(self, task, crowd_request):
    return self.pipeline(crowd_request)
```

**Figure 7: An example implementation of the WorkFlow run method.**

The `pipeline()` function is an inherited class method for the WorkFlow class. It delegates the execution of the pipeline via a helper class called the CrowdPipeline, which uses session memory to provide state for client-to-task executions along the pipeline. It also makes two fundamental assumptions about pipelining: 1) the developer has listed the tasks in the desired sequential order when declaring the tasks array in a WorkFlow object (e.g. see line 10 in Fig. 4) and 2) a POST request on one Task produces output that is consumed directly by the GET request on the subsequent Task with no need to return a response to the client that initiated the pipeline (that is until the last Task has executed and generated the final response of the pipeline). If a developer can specify a self-contained pipeline that does not need any additional customization, then the single line of code above (line 3 of Fig. 7) is all that is needed to implement and execute this pipeline of tasks. If, however, some customization is required then additional work must be performed and additional API methods of the AbstractWorkflow class need to be implemented (as discussed
below). Pipelines are not appropriate if a developer needs to perform custom code in
between the POST request of one task and the GET request of the next. However, in
most cases, this is a reasonable assumption for many crowdwork-related workflows.

Pipelining in the Crowdrouter is achieved with the following algorithm:

1. Given: a current Task to be executed and a list of Task classes declared
   by a WorkFlow; iterate through the Task list along with their indexes
   and stop when the Task to-be-executed is found.

2. Retrieve a session attribute called the *session pipeline key*; this hash is
   generated by performing a hash digest using the WorkFlow name,
   current Task name, and the index by which the Task appears in the list.

3. If this key is found and matches the hash digest of the current iteration,
   then execute the underlying Task.

4. Otherwise, if no key was found in the session, use the generated hash
   digest and store it as the new pipeline key in the session.

5. Once the current task has finished execution, check if it was a POST
   request.

6. If not, then safely return the CrowdResponse back to the client.

7. If so, then the client is ready to move on to the next Task in the pipeline.
   Generate a GET request for the next Task using the CrowdResponse of
   the POST request without redirection.

8. Once the next Task's GET request is executed, save the new session
   pipeline key and return the results.
This process continues until there are no more Tasks in the list; after the last POST request by the last Task is executed, the pipeline erases session memory, returning state to normal. If the client tries to execute a Task in the middle of the pipeline without having executed previous Tasks, a pipeline error will be raised in the framework and the application will need to respond properly.

Figure 8 shows an example of this algorithm being executed for a WorkFlow consisting of three Task classes—GeoTask, RankTask, and VerifyTask. To begin, the client requests the first Task (GeoTask) (1), and the server responds by routing via the CrowdRouter `route()` call, which routes the request to the PipelinedWorkFlow class. This WorkFlow delegates pipelined executions to its CrowdPipeline class, which is responsible for executing the algorithm outlined above. The CrowdPipeline examines the request, finds it in the WorkFlow, and, upon noticing that the session pipeline key is empty, executes the Task’s GET action and then creates a new key in the session for later use. Once finished, a CrowdResponse object is created, and the CrowdPipeline sends it back to the client upon checking that it was not a POST request (2). The client receives this response and has a new template for the GeoTask to render to the end-user. When the user submits the work for the GeoTask, a new request is sent (3). This time, the CrowdPipeline finds an existing key in the session and uses it to determine a match with respect to the Task name, WorkFlow name, and index within the pipeline. Given the fact that the POST request matches these three criteria, the CrowdPipeline executes it, and, upon determining that it is a POST request, transforms the new CrowdResponse object into a new CrowdRequest
object that then gets executed as a GET request along with the next Task in the pipeline (4). The results from this execution then get sent back to the client (5).

Note that this process symmetrically repeats for every additional Task that is declared within the WorkFlow. Once the CrowdPipeline finds that the current POST request is targeted for the last Task in the pipeline, it executes it, wipes out the session pipeline key, and returns the results to the client, allowing the client to pursue other Tasks available in the system.

![Pipelined WorkFlow Diagram](image)

**Figure 8: Crowdrouter Pipelined Workflow Algorithm.**

If the application developer wants to insert custom code before, during, and after the pipeline, the framework enables them to do so via the WorkFlow API that offers the `pre_pipeline()`, `step_pipeline()`, and `post_pipeline()` functions.
For example, a developer might want to prepare some custom session data within the `pre_pipeline()` function such as the current time, then use `step_pipeline()` to check how much time it took a user to perform each Task along the pipeline. Developers might want to use `post_pipeline()` to clean up any session data or record session results in a database.

Pipelining is well-suited for crowdwork-based situations that involve sequencing of tasks without interruption, and it lays the foundation upon which more expressive routing operations can be performed, such as branching, task-shuffling, looping with a terminating condition, and so on.

4.2.3 Branching. To build off of the pipelining functionality within the Crowdrouter, developers may want to provide conditions that act as switches to transition clients to one task or another. These semantics may be expressed by way of extending the pipeline functionality to treat routing operations as tasks themselves. This duck-typing behavior can be achieved by way of exposing the interface to such operations as if they behaved just like Task instances, because if the CrowdPipeline class calls the Task `execute()` function and expects a CrowdResponse in return, then routing operations must do the same thing. The Crowdrouter framework uses an AbstractCrowdChoice class to make this integration possible by allowing developers to insert concrete subclass declarations of the AbstractCrowdChoice into the task list within the WorkFlow class. Figure 9 depicts an example of an AbstractCrowdChoice subclass where a GeoTask may be executed in place of a RankTask, provided that a “geo” attribute is evaluated as true.
class ChoiceGeoOrRank(AbstractCrowdChoice):
    t1 = GeoTask
    t2 = RankTask

    def choice(self, crowd_request):
        if crowd_request.get_data() ["geo"] == True:
            return t1
        else:
            return t2

Figure 9: A concrete class to express task branching.

The Crowdrouter framework allows developers to declare these classes into the task list within a WorkFlow definition. Once the `execute()` function has been called for this AbstractCrowdChoice implementation, the `choice()` function is invoked, which determines which Task is to be instantiated and executed. Since these AbstractCrowdChoice implementations simply delegate to actual Task implementations, the pipeline treats them as such, and clients have the capability to provide branching when it is needed.

Note that pipelining is not required if developers want to use branching on its own. Sometimes, tasks should transition to one task or another, but not necessarily be constricted to a fixed sequence. Simply calling `execute()` on an AbstractCrowdChoice object will delegate the request to an actual Task instance based upon the implementation of the `choice()` method.

4.2.4 A/B Testing. In an era of rapid development with quick turnarounds based on the use of a system or the churn in active users of a system, applications
need to adapt quickly to the demands of end-users. Because of the nature of crowdwork and the necessity for many users performing many tasks, this challenge is especially important. Based on these facts, what becomes an increasingly important issue is to compare task and workflow executions to determine which ones are most effective. However, this becomes a challenge when needing to refactor backend code to serve different configurations of tasks. Developers must either duplicate code at different endpoints to provide the tasking for various workflows, or they have to move the tasking code to active endpoints while deactivating others. As a result, comparing crowdsourcing tasks and workflows becomes difficult and unwieldy.

Because of its architecture, however, the CrowdRouter naturally supports these use-cases. If an application developer wants to switch out tasks, she can simply just replace that Task declaration with another Task in a WorkFlow. If she does not want to modify the crowdsourcing configuration at all, she can simply create a new CrowdRouter class and configure the right WorkFlows and Tasks to compare against an original CrowdRouter. Then it becomes easy to handle CrowdRouter instances within the controllers; just swap out one CrowdRouter for another. The ease with which A/B testing is achieved in the CrowdRouter is a promising direction for future use-cases for comparing and contrasting crowdsourcing configurations.

4.2.5 Authentication. The CrowdRouter exposes functions to guard control flow at the CrowdRouter, WorkFlow, and Task level. Each entity defines an is_authenticated() boolean function that can be implemented by the application
developer to provide validation using the CrowdRequest object. If the function evaluates to false, the framework will raise an authentication error that the application must handle; otherwise, the framework proceeds as normal. At the CrowdRouter level, the request will be evaluated before running any WorkFlow or executing any Task; at the WorkFlow level, the request will be evaluated before executing any task; and at the Task level, the request will be evaluated before its own execution. Developers may choose to implement any or all of these authentication measures across their declared entities. By default, they are turned off, however they are easy to activate. A developer simply needs to add a decorator on top of the entity’s class definition—@crowdrouter_auth_required, @workflow_auth_required, and @task_auth_required—if she wishes to use authentication.

4.2.6 Crowd Statistics. The Crowdrouter framework is also equipped to gather statistics based upon requests that are sent through it. The CrowdRouter class delegates crowd statistics gathering to a CrowdStats class, which is designed to use the single-JSON-file-based database called pickledb to persist the data. The data is stored as a nested JSON document with keys at the root being the statistic captured, while nested keys enumerate through the WorkFlow and Task instances, and at the lowest level, GET and POST keys.

The only statistic that the Crowdrouter supports at the time of writing is a “task count” statistic. Essentially, for every request routed via a Crowdrouter instance, it will count the number of times a GET and POST request were successful. Figure 10 shows an example crowdsourcing application built for a Crowdrouter proof-
of-concept called the SimpleCrowdRouter. In it, a web page shows a task count distribution among the available WorkFlows and Tasks.

![SimpleCrowdRouter Statistics](image)

**Figure 10: Task Count Distribution for SimpleCrowdRouter.**

Developers using the Crowdrouter framework need not configure any database in order to set up crowd statistics. In the initialization function for the CrowdRouter, all that is needed is a simple function invocation to initialize crowd statistics:

```python
self.enable_crowd_statistics(<path to database file>)
```

After this declaration, the framework will search for the input file and use it to store data from incoming requests, making sure not to truncate any new results when restarting the process under which the crowdsourcing program runs.

### 4.2.7 Other features.

Miscellaneous features within the Crowdrouter framework include the ability to randomly shuffle task executions via the `random()` function.
WorkFlow function. This essentially nullifies a specified Task from executing when invoking the CrowdRouter *route()* function and instead picks a Task from the WorkFlow task list at random. Another available WorkFlow function is the *repeat()* function, which creates a pipeline of a specified Task for a specified number of times. An example invocation of this function would be: `self.repeat(crowd_request, 3)`. This call would use the CrowdPipeline class to create a pipeline with 3 instances of the requested Task. This is a helpful feature so that developers do not need to declare a Task in the task list for as many times as needed.

**Figure 11: Crowdrouter Class Diagram**

These features are representative of an extensible framework comprised of many roles and responsibilities. Figure 11 illustrates the Crowdrouter framework via a UML class diagram. The CrowdRouter class behaves as the interface developers use to invoke functionality from the rest of the framework. In order to activate crowd
statistics, the CrowdRouter has an association with the CrowdStats class, whose responsibility it is to provide CRUD operations on the statistics gathered when invoked within the framework. The CrowdRequest and CrowdResponse objects, as explained earlier, are encapsulations of HTTP request and response objects wrapped with metadata for accessing important attributes. Gamma, et al.’s Strategy pattern (1994) is used here to link varying implementations of framework-specific request/response objects to the CrowdRequest object.

On the right side of the diagram, the WorkFlow class delegates responsibility of managing pipeline implementations to the CrowdPipeline class. With this relationship, it becomes easy to replace the CrowdPipeline with another pipelining implementation if desired; for example, a looping class could implement the same interface as the CrowdPipeline and be used to provide different semantics for looping through a WorkFlow. Finally, future development work will be examining the relationships among WorkFlow, Task, and AbstractCrowdChoice. AbstractCrowdChoice currently mimics the Task interface via the use of duck typing and, thus, can be added to a pipeline as if they were tasks themselves. A similar approach will be used in the future to extend CrowdRouter’s workflow support to include looping, nesting, and other semantics.

With respect to task and workflow configurations found in current and popular crowdwork systems, I will discuss the Crowdrouter’s ability to satisfy them in the Chapter 5. I now describe future directions for the Crowdrouter to enhance its feature-set to become a fully-functional tool supported by the open-source community.
4.3 Future Work

There are useful features within the Crowdrouter framework now that provide crowd-based abstractions for more productive crowdwork programming. However, many features on the horizon will help position Crowdrouter as an important contribution in this space.

4.3.1 WorkFlow Semantics. There is a need to express WorkFlows as a more powerful abstraction in routing the crowd. Branching only begins to touch upon the capabilities that the Crowdrouter affords. Other well-known semantics such as looping, shuffling, and nesting should aid in providing a familiar abstraction to the programmer while also enabling more expressive and extensible crowdsourcing configurations. I speak about some of these semantics and how they could be implemented in future work.

4.3.1.1 Looping: looping is an easy concept to grasp but frankly an unintuitive operation with respect to implementing it within client-to-server interaction cycles. At its core, the semantics should show that, for every iteration of the loop being a client-to-server interaction, there is a condition that must be evaluated, and if that condition is satisfied, another iteration commences, otherwise the looping is complete. In the context of routing the crowd through a WorkFlow within the framework, what is needed is a way of capturing state (such as we did for the pipelining functionality above) and invoking the loop condition to evaluate that state at the appropriate time to determine what happens next.
Future work on the Crowdrouter will examine the AbstractCrowdChoice class to serve as the basis for implementing the loop condition. Using a special variant on the AbstractCrowdChoice that only executes one Task when a condition is satisfied and returns nothing otherwise is one solution. However, what is missing is the GET-to-POST transition that the CrowdPipeline provides. Therefore, future work on this feature entails a redesign of or addition to the CrowdPipeline class to support two looping styles—while looping and for-looping. The CrowdPipeline will require some new semantics to not solely rely on a fixed task list to provide looping, but rather use a condition that specifies the continuation of a workflow implementation if satisfied.

4.3.1.2 Task shuffling: Task shuffling is available via the `random()` function as mentioned previously, but what if the developer wants to provide task shuffling across a pipeline? In order to solve this challenge, the framework can rely on crowd operations such as the AbstractCrowdChoice to provide the right semantics. More specifically, a variant of this class—e.g. AbstractCrowdShuffle—may contain semantics to essentially call upon a single Task from the WorkFlow task list at random and return it. Therefore, during the pipelining process, a call to `execute()` is mapped to a call on the AbstractCrowdShuffle’s shuffling strategy, which consequently delegates to the selected task. I imagine that this will be a useful feature for distributing tasks to a crowd without bias.

4.3.1.3 Nesting: Last, but certainly not least, is nesting. Nesting refers to the ability of executing a hierarchical set of tasks in which invoking a high-level tasks starts the execution of a set of sub-tasks that may themselves invoke additional sub-
tasks. An example of this would be having the ability to specify a branching operation within a branching operation, which hints at the need for a recursive algorithm inside of the existing framework classes to process these new Crowdrouter elements. Future work will investigate how to implement nesting so that developers can treat WorkFlow task specifications as an expression that must be computed in a fixed order. This also implies that pipelining will sequence these operations and will be able to execute a recursive strategy to process Task and Task operations efficiently and correctly.

4.3.2 Maintaining Progress. Crowdsourcing comes in many different flavors, but what seems to be an overarching goal is to show that the collective actions of a crowd can help solve a challenge that not one person (or sometimes algorithm) can solve alone. As is referred to by literature describing the importance of task visibility between requesters of work and the workers themselves (Martin, et al., 2014), the challenge of maintaining and displaying progress is a central one that the Crowdrouter also seeks to solve. However, because the nature of crowdsourcing is filled with variances in requester-worker relationships, reward systems, workflow design, and bottom-up vs. top-down tasking (to name a few), it is not exactly clear how best to translate a social issue into a technical one. Once translated, it becomes yet another challenge to serve the needs of a wide audience that may have different (sometimes contrasting) views of what crowdsourcing ‘progress’ looks like.

Future work along this line for Crowdrouter will start with the ability to discretize Tasks, WorkFlows, and CrowdRouters. Essentially this means that after a
certain amount of requests have been sent through these entities, they would be considered ‘complete’ and would then become unreachable. This would require the use of a persistence mechanism to keep track of state, as the Crowdrouter framework currently relies upon newly instantiated Task and WorkFlows objects every time a request gets routed. Once implemented, however, developers can make use of this information to display how many Tasks, WorkFlows, or even CrowdRouters are remaining for the crowd to complete and how much work has been completed so far.

Another notion of progress to explore is a local indicator of what work is remaining for every client when performing tasks through a pipelined workflow. In this case, progress is private to just a client and shows which tasks have been completed and which ones are remaining, similar to how survey tools like SurveyMonkey show which pages have been performed and which ones are left. To implement this is quite trivial; an extra variable within the session can capture the position within the task list, and developers can use this datum to properly display the position on which the client is performing work through the pipeline.

4.3.3 Service-oriented Capabilities. Crowdsourcing applications face issues of scale just as much as other types of applications. In fact, they are typically under much stress as a large number of users are performing the same tasks over and over again. Sometimes, as in human-machine computation settings, these tasks are computationally expensive to serve or process, making it difficult to integrate crowd-processing. To compensate for issues of scale, applications tend to scale vertically (i.e. add more computational resources such as memory/storage/CPU to the host machine)
or horizontally (i.e. use technologies that provide distributed support on top of additional machines using commodity hardware), and crowdsourcing applications should be able to do the same. Since crowd routing itself is not memory-intensive, storage-intensive, or CPU-intensive, scaling vertically does not cause any changes to the framework as more resources are available, though what may need more attention is the amount of requests that the web server can process at a time, and so therefore load balancing is needed to support varying levels of crowd-sized traffic coming in to the application.

However, to reduce cost and maintenance, what may be more preferable for designing scalable crowdsourcing applications is to scale horizontally. If the application developer wishes to scale horizontally, there are a few options: one, set up multiple independent application processes across multiple machines and provide load balancing on top to delegate requests to them in a round-robin fashion. This option is favorable because it increases the uptime of the application, but it also increases the amount of housekeeping to manage crowd workflows (i.e. making changes on one machine must be made on the others). Another option would be to remain at one machine and delegate the crowdwork to other middleware that can process the crowdwork and return results. However, that would increase the complexity of the application with the need to communicate with multiple machines synchronously. A third option is to use a distributed setup but also partition the crowd work effectively across the machines. Such an environment would have knowledge of
how crowd-based processing is distributed and encapsulate that knowledge within the framework.

This would necessitate a redesign of the Crowdrouter framework to additionally support crowd-workflows as services, rather than just components. It may be counter-intuitive to think of the Crowdrouter framework as a networked shared-data system, but if these crowd workflow and task entities are treated that way, then there are benefits to creating a computing environment where crowd traffic is effectively distributed among multiple machines. Each Crowdrouter ‘service’ would be synchronized with other services in the same network, partitioning out and replicating Crowdrouter entities across them to provide the availability and partitionability as described by Eric Brewer’s CAP theorem (2012). Consistency, on the other hand, could be managed manually by new changes in the Crowdrouter configuration. The application developer would make these changes, and the Crowdrouter services would again synchronize them across the network.

I see the features to create a distributed version of the Crowdrouter as a long-term goal that will require a re-architecting of how routing gets performed. There is an additional layer of complexity in satisfying a WorkFlow specification once the crowd entities do not live on the same machine. Care must be taken to reduce latency and maximize completeness in the face of failure.

4.3.4 Motivation. Participants highlighted the importance of needing mechanisms to address crowd motivation. Some examples were given during the interviews, including gamification and normification techniques to mobilize users in
performing tasks with reward or with meaning. This was a critical concern in the
development of crowdwork systems for participants because of the often-documented
*chicken-and-egg problem*: A crowdwork system cannot be useful without its users, but
users will not work for a system if there is not any user-activity or progress. Issues of
 adoption and trust come to the fore; crowdwork systems need to earn crowd trust,
which requires a long-standing commitment to the domain, maintenance, and
adaptability to crowd concerns (Barrenechea, et al., 2015a). These non-functional
concerns can address the motivational aspects of crowdwork.

The Crowdrouter does not currently offer features for implementing
motivational techniques like gamification or normification out-of-the-box. I argue
that although motivation, trust, and adoption are valid concerns that crowdwork
developers must pay attention to, they are application-level concerns—as opposed to
framework-level concerns—that must be addressed. Frameworks like the
Crowdrouter can potentially step in and support developers with Task or WorkFlow
strategies to provide skeletal implementations for crowd-based gamification or
normification, but they are largely left to the responsibilities of the developers who
must understand the needs and expectations of members of the crowd and design
their systems according to them. Fortunately, the Crowdrouter class-based
abstractions are bound to type hierarchies that can be inherited to provide deeper
levels of CrowdRouter, WorkFlow, and Task implementations. In this way, the
Crowdrouter framework can behave like a catalog of varying crowdwork
implementations, enabling developers to choose which implementation makes most sense to their use cases and crowd concerns.

4.3.5 End-Crowd Programming. Another avenue of future work to discuss is the capabilities for the crowd to define the work for themselves. It is not infeasible to imagine that the features afforded by the PyBossa framework for instantiating tasks or creating Task, WorkFlow, or CrowdRouter classes by the Crowdrouter framework can instead be contributed by the crowd. Akin to end-user programming, end-crowd programming can be an effective technique for bridging crowd-to-system interactions and solve socio-technical challenges with how to use, build, and maintain crowdwork systems in a feedback loop. This is a promising avenue for developers to reflect the structures of work that the crowd forms in digital volunteer settings—such as that found in crisis mapping environments (Palen, et al., 2015)—while designing a system to be more likely adopted.

With respect to this use-case, the Crowdrouter framework can be extended to expose a specification language for creating tasks and workflows within the crowdwork system. This language would be used to create structures for task design, including the problem to solve, the media (images, videos, music clips, etc.) to incorporate into the task, and the style of crowdwork with which to perform the task, including forms of input. As the developer participates in customizing the language for the domain in which the crowdwork system is deployed, the framework can then provide an endpoint that will serve and process the task for constructing other crowdwork tasks using the specification language. With respect to workflow design,
the crowd can also participate in re-arranging existing workflows (if the developer insists) or creating new ones via the tasks that are being served within the framework.

The challenge in providing this level of end-crowd programming is the risk of re-architecting the framework to offer boilerplate solutions to crowdwork systems development. Like the PyBossa framework, this can lead to architectural and technological decisions that the framework requires on the part of the developer, which decreases its ability to become a versatile and lightweight framework. Another challenge is in creating a specification language flexible enough to handle a variety of use-cases for end-crowd programming. Because CrowdRouter, WorkFlow, and Task classes are defined as object-oriented classes, there may need to be a shift in migrating to a persistent format which can be contributed by the developer or by the crowd, and the mechanisms by which new such crowdwork entities can be created by both stakeholders must also be created as well.

4.4 Prototype Implementation

The Crowdrouter is available as an open-source python package on GitHub (https://github.com/project-epic/crowdrouter) and currently indexed by the Python Package Index (PyPI). At the time of writing, it is at version 1.5.3 and supports all of the features documented in this dissertation, with an eye towards the future use-cases also documented here. Figure 12 shows the website to learn more about the Crowdrouter framework, found here: http://epic.cs.colorado.edu/crowdrouter/.
4.5 Framework vs. Abstraction

At the time of writing and as the Crowdrouter is under active development, there is great tension to properly position the framework between lightweight (crowd-entity management, workflow specification, crowd-programming libraries) and heavyweight (crowd-entity persistence and administration, framework configurations, boilerplate HTML crowd templates, administrative dashboards) feature-sets. Like more heavyweight crowdsourcing tools like PyBossa and Hive, it makes sense to provide the Crowdrouter with more features that allow developers to avoid challenges with respect to managing task order, encapsulating task and workflow definitions, building HTML templates from scratch, and the like. However,
it is also critical to maintain flexibility and extensibility when more features are added to the framework, to keep the learning curve low and configuration at a minimum, and to pay attention to the needs of application developers. On one hand, they might say that they do not want to configure extra database tables, access administrative dashboards, and upload media for automatic task-serving and processing; on the other hand, they might desire those features when they want to quickly scaffold a project but still have room for customization. The use-cases for crowdsourcing development become wide and divergent.

The overarching objective of the Crowdrouter framework is to become an abstraction, not just a framework, for designing and managing crowd tasks and workflows in crowdwork programs. I believe that the abstraction being discussed is the Crowdrouter architectural design, which illustrates the roles and responsibilities of the CrowdRouter, WorkFlow, and Task entities and how together they route and encapsulate crowdwork implementations. It is important to step away from implementation details and determine whether in its current form, the abstraction can satisfy these use-cases efficiently. If so, then future frameworks and tools can use the Crowdrouter design as a foundation upon which it can support both lightweight and heavyweight features for developers.
CHAPTER 5

EVALUATION

Having synthesized insights from my crowdwork interviews and presented the Crowdrouter framework, I now present the evaluation of the Crowdrouter framework with respect to versatility. This chapter argues that the Crowdrouter framework addresses many of the crowdwork-related challenges previously identified, laying a foundation upon which future crowdwork frameworks are built to address more advanced use-cases. The chief objective of this dissertation is to show that the Crowdrouter is versatile, i.e. meaning that it has the capacity to support a multitude of diverse use-cases for crowdwork system design.

To properly demonstrate versatility, this chapter shows ease of use in integrating the Crowdrouter into existing crowdwork systems, and it then shows how the Crowdrouter can support a variety of workflow and task configurations hosted by other crowdwork systems. To begin, I introduce two systems used to evaluate the ease of integration of the Crowdrouter. The first is the EmergencyPetMatcher (EPM) system, a crowdwork web application I created to help reunite lost and found pets as a collaborative effort during disaster events (Barrenechea, et al., 2015a). EPM facilitates three workflows for reporting, matching, and verifying pet reports and pet matches coming from the crowd. The second system is a smaller crowdwork system called the SimpleCrowdRouter that showcases the various features offered by the Crowdrouter. The goal is to analyze system property changes before and after the
Crowdrouter was introduced to these systems. This is to illustrate the impact that the Crowdrouter has on crowdwork systems like EPM and SimpleCrowdRouter.

The second evaluation effort shows how the Crowdrouter can support current workflow and task configurations that popular crowdwork systems use today. I describe three such crowdwork systems and their mechanisms to serve crowdwork workflows and tasks, and I compare their capabilities with those of the Crowdrouter. I then broaden the scope and tabulate ten crowdwork systems with their workflow and task configurations. I show how, given a variety of workflow and task configurations by crowdwork across diverse application domains, the Crowdrouter can fully support most if not all of them.

5.1 Versatility

Versatility—the extent to which a framework can support a varying set of systems and their use-cases—is not a clear-cut metric within software engineering research. Methods that attempt to do this can involve anecdotal evidence to support how well the framework integrated with the system, but such evidence is often scattered and unsystematic, making it difficult to assess. Another approach is to deploy the framework under evaluation across a variety of existing systems and determine the difficulty with which the tool integrated into the system. However, once a framework has been integrated within such systems, what sub-metrics should be used to determine the framework’s versatility?
In this dissertation, I measure versatility in two ways—impact and range. Impact can be measured by the amount of refactoring needed to successfully integrate the Crowdrouter framework into two systems—EPM and SimpleCrowdRouter. While the SimpleCrowdRouter application is simply a sandbox to illustrate features of the Crowdrouter and therefore can be re-architected, deconstructed, and refactored with relative ease, EPM is a much more mature system comprised of many interdependencies among internal components, classes, and models. I analyze properties of both systems that are important for crowdwork system design, as well as properties of their integration with the Crowdrouter, and I show that, given the contrasting complexities of both systems, the Crowdrouter still provided easy integration with both, reinforcing the versatility claim. This is what I call the impact measurement of the evaluation of the Crowdrouter framework.

The second approach—range—refers to the capacity with which the Crowdrouter supports varying crowdwork configurations. Based on this definition of versatility, it is important to assess the Crowdrouter’s ability to satisfy a variety of such configurations that appear in popular crowdwork systems used today. To do this, I present three crowdwork systems that are used extensively and—adopting a user’s perspective—describe how they arrange tasks and workflows in order to provide the crowd with work. I then broaden the scope to include other crowdwork systems and examine their crowdwork configurations. An analysis of these configurations is presented here and shows that the Crowdrouter can support most of them while leaving room for framework extensibility.
5.2 Impact

The first approach to measuring versatility is in measuring impact, or the changes needed to be made in order to integrate the framework into the crowdwork system. I first present two systems used to make this evaluation possible, and I then present the analysis performed to evaluate Crowdrouter impact within them.

5.2.1 EmergencyPetMatcher. EPM is an online, collaborative crowdwork system designed for pet advocates and enthusiasts at large to report, match, and verify lost and found pets during disaster events. It relies on the implementation of crowdwork patterns (Bernstein, et al., 2010, Kittur, et al., 2013) to allow the crowd to perform work in these three functional feature-sets: reporting pets, matching pets, and verifying pet matches. The user interface of EPM’s home page is shown in Fig. 13.
It is designed to get users involved quickly in the primary workflow of matching pet reports of missing animals with pet reports describing animals that have been “found” and are currently being housed in shelters or in custody by other pet owners. The user interface that EPM provides for this matching activity is shown in Fig. 14. Before this activity can occur, users must submit a pet report with a variety of attributes about the particular pet such as the pet type, its status, and the date it was lost or found, along with its name, breed, sex, etc. Once a pet report has been submitted and saved, it appears in the list of reports viewable on EPM's home page.

While matching pets, EPM provides a variety of features to make that process more efficient. For instance, if a user is examining a pet report for a lost Dalmatian puppy, then EPM will sort “found” pet reports of young Dalmatian dogs to the top of the matching list before showing other, less likely pet reports for consideration. Once a match has been made, EPM’s users are asked to vote on the match giving either an “up vote” or a “down vote” to indicate their confidence that the two pets in the proposed match are indeed the same animal. EPM calculates a threshold for the number of upvotes that are required for a “successful” match based on the number of active EPM users at the time.

Of course, a “successful” match in EPM is not actually a match but rather the time when EPM contacts the people associated with the two pet reports asking them to verify the match by meeting outside EPM to determine if a match has actually been made. If the owners of the two pet reports return to EPM to indicate that the
match was successful, then the match is moved to a “Reunited Pets” page to show the EPM community that their work is indeed having an impact.

Finally, EPM provides a variety of features to promote a sense of community among its users. Users can follow one another to receive notifications of what their friends and/or collaborators are doing, allowing people to work together on the same set of pets. EPM users can send messages to one another without having to reveal their personal e-mail addresses and they receive notifications if any of the pet reports or pet matches that they worked on in the past have changed in some way (for instance if a proposed pet match has been verified as a success). Additional details about EPM’s user interface, workflows, and functionality are documented in (Barrenechea, et al., 2015a).

Figure 14: Matching Pets in EmergencyPetMatcher.
5.2.1.1 Architecture: To achieve the scale at which EPM can be effective in online pet matching activities, it needs to rely on architectural principles that sustain dynamic variations in system use, maintain extensibility in demand for features, and cope with system complexity so that edits to a particular feature do not trickle out to other parts of the system. These expectations are indeed a gold standard for all software systems. However, choosing the wrong architecture can exacerbate the issues that are faced during maintenance, evolution, and deployment. Principles such as DRY (Don’t-Repeat-Yourself), separation of concerns, loose coupling, and high cohesion have been implemented in web application frameworks such as Django and Ruby on Rails, and so, at the outset of the project, the natural choice was to use one of these frameworks (Django) to bootstrap the functionality and design of the system. Figure 15 shows how EPM has been architected:
EPM is built using the popular Django web application framework. It uses a web server (Nginx) that receives HTTP requests coming from web clients and reverse-proxies the requests if it requires Django to respond to them. If Django is not needed to satisfy the request, such as when an image is being requested, the web server will fetch it from the file directory itself. The application server (Gunicorn) handles requests that require server-side logic, such as retrieving a PetReport object to display and render onto an HTML page. The app server will listen for particular URL routes that have been defined, and will invoke the Python interpreter when a URL has been matched with a view (i.e. controller). Database calls may be required to fetch domain models of interest (PetReports, PetMatches, etc.), in which case a backend
database server will retrieve those models so the overarching HTTP request to EPM can be satisfied. Once the request has been satisfied, execution will finish by creating an HTTP response that is sent back to the web server to then return back to the client. At the time of writing, EPM’s architecture follows a 3-layer architecture, with the web and application servers, Django project, and database system all housed within the same machine.

5.2.1.2 Crowdwork Architecture: EPM is a crowdwork application, which means that it relies upon some configuration for serving tasks and workflows to the crowd. In this fashion, the configuration is an architecture that describes tasks as the components and workflows the connectors of a system responsible for serving and processing crowdwork. Such an architecture is responsible for mobilizing the crowd to produce high quality work through usable, visible, and meaningful tasks and workflows. Through its arrangements, crowdwork architectural design can also introduce crowdwork patterns, such as allocating workers to verify other worker results to increase quality assurance (Bernstein, et al., 2010) or to include real-time tasking feedback for workers to produce better results (Dow, et al., 2012).

EPM’s crowdwork architecture tasks the crowd with well-defined workflows for creating pet reports, matching them together in pairwise units called pet matches, and then voting on their success/failure (see Figure 16). This architecture constitutes a pattern— I call it the create-connect-correct (CCC) crowdwork pattern—to enforce a feedback loop for crowdwork verification by workers in a bottom-up tasking environment. Workers create their own work by reporting on pet reports in the
system (creating), which propagate as future crowd tasks for linking pets together to form pet matching (connecting), and when there is a shift from production to verification, the pattern mobilizes the crowd to verify its own work by voting up/down on the proposed pet matches. The implementations of these crowdwork workflows and tasks are supported by user-centered design techniques that have been documented in previous work (Barrenechea, et al., 2012; Barrenechea, et al., 2015a). I believe the CCC pattern forms the basis of a design pattern for crowdwork systems that would be appropriate for other lost-and-found crowd tasking systems for use during and after disaster events.

![EmergencyPetMatcher Crowdwork Architecture](image)

**Figure 16: EmergencyPetMatcher Crowdwork Architecture.**

5.2.2 SimpleCrowdRouter. The SimpleCrowdRouter web application is a sandbox crowdwork system primarily built to demonstrate various features offered by the Crowdrouter framework. For that reason, its source code and static assets are packaged along with the Crowdrouter framework available as a Python package on PyPI. It is built using the Flask python web application framework and basic web-based assets to render three available crowd tasks—identifying tweet hashtags, ranking image damage severity (akin to the brainstorming problem presented within
the crowdwork study), and providing image captions. With the help of the Crowdrouter framework, the SimpleCrowdRouter application provides these tasks within a number of varying workflow configurations. Figure 17 illustrates the homepage of the application, including the Crowdrouter WorkFlow and Task instances that are made available to work on.

![Welcome to the SimpleCrowdRouter Example!](image)

**Figure 17: The SimpleCrowdRouter Example Web Application.**

The Crowdrouter’s features have been described in detail in Chapter 4, so I will not describe them again for the purposes of describing the SimpleCrowdRouter. However, I will mention that those features are being used to design workflows and tasks to present to the crowd in various ways. For example, the first WorkFlow class called BasicWorkFlow, which is called “Perform a Single Task” in Figure 17, contains
the three Task classes previously introduced called “Rank an Image”, “Answer a Question”, and “Pick Tweet Hashtags”. Members of the crowd can click on any of these Tasks to begin work on it, which are the semantics declared by the BasicWorkFlow to allow crowdwork executions on any of these tasks simultaneously and without order restrictions.

In contrast, the second and third WorkFlow classes shown in Figure 17 are pipelined workflows called RankingMultipleImagesWorkFlow (rendered as “Rank Multiple Images”) and AnswerMultipleQuestionsWorkFlow (rendered as “Answer Multiple Questions”). The RankingMultipleImagesWorkFlow class declares the semantics to repeat the “Rank an Image” task three times in a sequential order, while the AnswerMultipleQuestionsWorkFlow class inherits from RankingMultipleImagesWorkFlow to avoid re-declaring pipeline semantics and declares its repeatable task as the “Answer a Question” Task.

In addition, the SimpleCrowdRouter displays two alternative Crowdrouter features: one, the ability to view crowd statistics on requests sent to each Workflow and each of its Tasks classes; and second, the ability to authenticate as an administrator, which permits access to perform another WorkFlow called AuthWorkFlow. If an unauthenticated user tries to perform tasks under this WorkFlow, the Crowdrouter framework will disallow the request, raising an exception that the application must handle.

One final note to make about the SimpleCrowdRouter is that within the server application code, there are only two endpoints—perform_task and pipeline—
that are defined to handle more than 20 possible client-to-task interactions. This means that these endpoints are generic enough to receive requests coming from various URIs (such as /rank_image or /answer_questions) and have the flexibility to define a single route call to handle all of them using the Crowdrouter. While the perform_task endpoint uses the traditional CrowdRouter route() call, which accepts a specified WorkFlow and Task name, the pipeline endpoint uses its alternate function called pipeline(), which only requires a WorkFlow parameter. This allows the framework to use the session to remember which Tasks have been performed along a pipeline, making it useful for cases where a single endpoint can be routed to a single pipeline call, instead of having multiple endpoints for each Task along the pipeline.

5.2.3 Evaluating Impact. Both EPM and SimpleCrowdRouter system share similar crowdwork complexities (they both host three major crowd tasks), but there are grand differences in scale between them. Besides very basic login/logout authentication, as well as crowd-statistics reporting via the Crowdrouter, the SimpleCrowdRouter does not host any other application workflows (crowd or not). This is in contrast to the more mature architecture of the EPM system structured in part by the Django framework, which includes facilities for administrative workflows, social media-based authentication, email support, database migrations and modeling, and more. Given these differences, the analysis seeks to measure the sizes of both systems with respect to source lines of code (SLOC). This will provide a window
through which developers can view the impact of the Crowdrouter framework being integrated in the two disparately-sized systems.

For each system, I perform my analysis by taking four development “snapshots” as I was undertaking crowdwork-related development with and without using the Crowdrouter. These snapshots are different for each system because of their crowdwork configurations. For example, EPM has three separate workflows, each with one or more tasks, and it does not need authentication support because Django already provides it. In contrast, the SimpleCrowdRouter has three tasks and more than three workflows that attempt to demonstrate the routing of those tasks in varying ways; furthermore, the SimpleCrowdRouter does make use of the authentication features provided by the Crowdrouter framework.

In order to reason accurately about the differences in SLOC for each system with and without integrating the Crowdrouter, I performed the following checks and balances:

1) New workflows were added to both the EPM and SimpleCrowdRouter systems. This translated to defining new Crowdrouter WorkFlow classes in the versions of both systems that use the Crowdrouter, and creating more endpoints to handle identical workflow implementations in the versions without the Crowdrouter. This was done to show SLOC variance when using and not using the Crowdrouter framework.
2) I used the CLOC source-code measuring tool (http://cloc.sourceforge.net), which provides a breakdown of programming language-specific code found recursively within a target directory.

3) When it found blank or commented lines, CLOC reported them as separate metrics, therefore they are excluded from the analysis.

4) While there were some small variances in front-end (HTML, CSS, and Javascript) code across WorkFlow implementations, the analysis focuses solely on the differences in Python SLOC, since Python is the language that the Crowdrouter framework uses. However, the SLOC numbers displayed in the analysis refers to all sources of code for all languages found in each project.

<table>
<thead>
<tr>
<th></th>
<th>SimpleCrowdRouter</th>
<th></th>
<th>EmergencyPetMatcher</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Workflow Additions</strong></td>
<td></td>
<td></td>
<td><strong>Workflow Changes</strong></td>
<td></td>
</tr>
<tr>
<td>Additions</td>
<td></td>
<td></td>
<td>Changes</td>
<td></td>
</tr>
<tr>
<td>Basic Workflows</td>
<td>162</td>
<td>121</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>Add Pipelined Workflows</td>
<td>173</td>
<td>181</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>Add MixedWorkflow &amp; AuthWorkflow</td>
<td>185</td>
<td>307</td>
<td>1</td>
<td>13049</td>
</tr>
<tr>
<td>Add ChoiceWorkflow</td>
<td>210</td>
<td>390</td>
<td>1</td>
<td>13080</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Workflow Changes</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swapping First and Last Task in MixedWorkFlow</td>
<td>1</td>
<td>18</td>
<td>Basic Workflows</td>
<td>13023</td>
</tr>
<tr>
<td>Deleting Last Task in MixedWorkFlow</td>
<td>1</td>
<td>25</td>
<td>Add MixedWorkFlow (4 Tasks)</td>
<td>13049</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: SimpleCrowdRouter and EPM Workflow Additions and Changes.

Table 1 tabulates the results from this workflow change analysis. In both the CR (Crowdrouter) and non-CR case, the same use-cases were implemented in both systems to maintain consistency. However, in the CR cases, these systems made use
of the Crowdrouter to implement them, while in the non-CR case, they relied mostly upon controller endpoints and redirections of those endpoints. In general, this approach to implementation in the non-CR condition caused code duplication and other bad smells for refactoring, but it did achieve the effect of providing various workflow configurations. In the CR cases, controller logic was organized into Task classes, and WorkFlow classes were added to satisfy the requirements of the same crowdwork-related use-cases.

The criteria for determining impact under this evaluation framework is derived from two features of crowdwork workflow design—workflow operations (single, pipelined, branching) and workflow changes (swapping order of tasks, deleting tasks). They are used to generate the snapshots used for both the SimpleCrowdRouter and EmergencyPetMatcher systems, as I will outline in the next sections. The justification for using these criteria is recognizing that workflows are critical in serving and processing crowdwork in many crowdwork systems. These systems exhibit some arrangement of tasks to the crowd, which is made possible by workflow implementations. Establishing the criteria of workflow operations and changes against which to evaluate the impact of the Crowdrouter serves to 1) assess the way in which the framework will provide most impact to the crowdwork systems, and 2) provide a generalizable basis for evaluating crowdwork frameworks and their ability to integrate into other crowdwork systems. To reason about the level of impact that the Crowdrouter makes on the two systems, the analysis measures SLOC
differences when workflows are added to and changed within them\(^3\). I discuss these changes as I focus on each system.

5.2.3.1 SimpleCrowdRouter: The SCR system exemplifies small crowdwork system design where not too many features have been implemented, but it becomes a sandbox within which tools and frameworks such as the Crowdrouter can be tested and assessed for fit based upon the use-cases. The SCR was measured based on four snapshots:

1) A basic workflow was created, one that allows three tasks (ranking an image, answering a question, and picking Tweet hashtags) to be executed simultaneously, anytime.

2) In the second snapshot, two new pipelined workflows were added, one for repeating three instances of the image ranking task, and another to perform the same behavior for the answering-a-question task.

3) The third snapshot introduced a mixed, pipelined workflow of all three tasks in a specific order (picking tweet hashtags, ranking an image, answering a question, and repeat again), as well as an authentication-based workflow that required administrative privileges to access the three tasks.

4) Finally, to increase complexity, a decision (choicing)-based, pipelined workflow was added. The order for this workflow is: ranking an image, a

---

\(^3\) All analysis results are stored in the same code repository for each system. Therefore, they are made publically online – EPM source can be found here <https://github.com/Project-EPIC/emergencypetmatcher> and the SimpleCrowdRouter source can be found here <https://github.com/Project-EPIC/crowdrouter>.
choice between ranking another image or answering a question, and then answering a question.

In addition, the analysis captures two cases where workflow changes are made after the final snapshot. This is done to measure instances of workflow change and how much SLOC impact that causes in both the CR and non-CR versions of the SCR system. Figure 19 shows a relationship between SLOC and WorkFlows that were added as snapshots to SCR. There are few observations to make. First, within the first snapshot, it is evident that SCR must pay an upfront cost to set up basic WorkFlow and Task classes, yielding 162 SLOC in total and an extra file which defines these Crowdrouter-specific classes and allows the server application to refer to them. In the non-CR case, however, without this extra file, the server application simply defines three endpoints, one for each of the tasks and to make available to the crowd, which yields 121 SLOC. This is a cost that both EPM and SCR must pay in SLOC in order to use the Crowdrouter framework.
Figure 18: SimpleCrowdRouter with/without Crowdrouter.

In the second snapshot, however, the two cases converge. In the CR case, adding two new but fairly basic pipelined workflows have a smaller impact (+11 SLOC for new WorkFlow classes) than that of the non-CR case (+60 SLOC), which has to create new endpoints for implementing Task repetition for a specified number of times (using session memory). This cost for the non-CR case is in contrast to the Crowdrouter’s CrowdPipeline, which also uses session memory to hide these implementations and provide them through the WorkFlow API. In total, the SCR systems consists of 173 SLOC with the CR and 181 SLOC without the CR.

In the third snapshot—and with the introduction of the mixed workflow use-case—the non-CR system struggles with redirection code and additional endpoints to handle pipelining for non-uniform task sequences. This causes endpoint implementations to not only handle session memory to keep track of their location in the workflow, but also creates dependencies between themselves (e.g. endpoint
For the authentication-based workflows, this is less troublesome; SCR creates new endpoints to authenticate with session-based credentials and then redirects to the three original task endpoints. In total, these changes required 126 additional lines of source code.

In the CR-case however, not much changes. It declares a new WorkFlow class to handle the mixed workflow configuration, using the pipeline() functions available through the Crowdrouter framework. With respect to the authentication workflow, the CR-based SCR again creates a new WorkFlow class and uses the is_authenticated() function to authenticate clients when routing to it. This achieves the same effect as using authentication logic within a controller endpoint. These changes required an additional 12 SLOC for the CR-based version of SCR. In total at the end of the development work for this snapshot, the non-CR system consisted of 307 SLOC while the CR system contains just 185 SLOC.

In the final snapshot, a choosing-based pipelined workflow was introduced, which not only imposes an ordering restriction on sequenced tasks but also provides a branching opportunity with criteria to be assessed during runtime. Here, the CR-based system created an extra WorkFlow class and a new AbstractCrowdChoice class, which encapsulates the decision in its choice() function. Depending on the results of the condition, the AbstractCrowdChoice implementation will choose one Task (i.e. ranking an image) over another (i.e. answering a question). The condition was hard-coded to look for a key within the session, and if that key did not exist, it would choose the question task. Developers can implement whatever logic is desired to choose
between the two different Tasks. Based upon these new Crowdrouter classes, as well as using a new `pipeline()` function within a new endpoint specifically for this workflow, the CR system increases by 25 SLOC for 210 SLOC in total.

In the non-CR case, the SCR system had to create new endpoints and redirection code to handle the sequencing, and then, when it came time to implement choosing, it needed to wrap task implementations inside of the conditional that the CR-case used within the AbstractCrowdChoice class, implementing both cases to cover both possible executions. As a result, the non-CR implementations required an additional 83 SLOC bringing its total for all four snapshots together to 390 SLOC.

In addition, two workflow changes were made after the final snapshot—swapping the first and last task in the mixed workflow, and deleting the last task in the mixed workflow. While the non-CR-based SCR system required 18 SLOC changes to represent the first and last task swap within the mixed workflow, the CR-based system required just one; the MixedWorkflow specification just needed to swap the order of the first and last Tasks. The same effect is captured in the second workflow change. While the non-CR system needed to chop off the last task represented in an application endpoint, the CR-based system just deleted the last Task in the Workflow specification, and the job was done. This reflects the flexibility gained by encapsulating tasks and workflows using the Crowdrouter framework.

In summary, the Crowdrouter implementations improved upon task and workflow encapsulation, caused less endpoint definitions, made use-cases such as authentication, redirection, and choosing easy to cope with, which in turn made the
system easier to understand. The non-CR-based SCR system had to implement these features from scratch or use existing web framework abstractions to deal with them. However, dealing with session state while traversing through endpoints causes an additional layer of complexity for development, which can be avoided by using pipelining within the Crowdrouter.

One final note to make is that the non-CR case required ~300 lines of code to be generated to reach the final development snapshot, 75% more code than the CR-based system. Most of this code resides in the task duplications across all of the endpoints that mimic what is seen in typical web app development. It is easy to see that these duplications represent “bad smells” for refactoring (Fowler, 2002) and therefore can be encapsulated into functions or classes, thereby reducing the SLOC distance between the CR and non-CR cases. However, even with these refactorings, developers must still consider how to handle redirection-based challenges, session management, and higher-order operations such as choosing. After dealing with most of these problems manually, the question then shifts to whether the use cases for the developer’s project call for customized solutions to these problems, or whether framework-specific facilities such as those offered by the Crowdrouter can support and abstract them.

5.2.3.2 EmergencyPetMatcher: As smaller systems get larger, frameworks must be able to cope with more specialized use-cases. In the case of EPM, a large part of the system is not related to crowd work at all but rather to the features that provide social networking, user profiling, bookmarking, messaging, and database modeling,
to name a few. When a user completes a crowd-task, such as voting on a pet match, an instance of that work is stored in the database, an activity log is created, stored, and shared on the website, and a message is sent to the user thanking her for the work performed. Crowdwork frameworks that provide flexibility in integrating crowdwork into more mature systems, while also providing and supporting newer features, are very much a needed component in the development space of crowdwork applications.

For the EPM system, the following snapshots were taken:

1) For the first snapshot, three crowd tasks are measured. They are the MatchTask, ProposePetMatchTask, and the VoteTask. A Task involving the submission of pet reports was omitted from this analysis because it stands as a much more independent task, only performable if users had a pet to report on, and not one that was chainable within other more bite-sized tasks like voting, matching, and pet match proposing.

2) For the second snapshot, a mixed, pipeline workflow was introduced. This workflow sequenced the following Tasks in order: VoteTask, MatchTask, ProposeMatchTask, and VoteTask. In the context of EPM, this is used as a quick method for performing crowd tasks. In fact, the workflow begins with a click of a button at the home page that reads, “Help us out in one minute!”

3) For the third snapshot, a larger version of the workflow was introduced to illustrate how the system reacts to longer sequencing. The sequence is now:
VoteTask, MatchTask, ProposeMatchTask, VoteTask, VoteTask, MatchTask, and ProposeMatchTask.

4) For the fourth snapshot, a choosing workflow was introduced to provide comparison with that of the sequential workflow. It pipelines a pair of Tasks, the first being the VoteTask, and the second being the choice between a VoteTask or a MatchTask. Both workflows are displayed in the home page.

![EPM: WorkFlow implementations over SLOC](image)

**Figure 19: EmergencyPetMatcher with/without Crowdrouter.**

The EPM case analysis parallels much of what was observed in the SCR case. However, because EPM sports ~13KLOC instead of just hundreds, it is important to see what the impact of integrating the Crowdrouter becomes in a larger system. In the first snapshot, as observed in the SCR case, there is an up-front cost to using the Crowdrouter, and because the crowd-based controller actions are much bigger (mean size is 43 SLOC) than those of the SCR (mean size is 11 SLOC), the differences in SLOC will appear to be more drastic. In contrast to the SCR system, where there are
two controller endpoints through which all crowd work was routed, EPM still uses its original URIs to maintain RESTfulness within the various modules—as well as to deal with varying input parameters coming through those endpoints—that are mapped to the models it stores in the database. Within each endpoint, however, EPM uses a CrowdRouter object to route the incoming request to the specified WorkFlow and Task classes.

In the first snapshot, it is evident that the cost of keeping those endpoints intact while moving all of the crowd-based controller logic to new Crowdrouter classes creates a difference of nearly 150 SLOC. Here, the CR-based EPM system has 13023 SLOC while the non-CR based system has 12889 SLOC. In the second snapshot of the SCR analysis, just like in the SCR case, we also see a change in SLOC leadership; the CR-based case shows the system at 13049 while the non-CR case has 13083. This is due to the implementation of a new mixed, pipelined workflow that incorporates voting, matching, and propose-pet-matching tasks. The CR-based system implements a new WorkFlow class to handle the mixed workflow, as well as a new endpoint to handle the workflow, while the non-CR based system creates multiple new endpoints—one for each new task—and uses redirection logic as a means to tie them together, replicating the sequencing functionality as the crowd are taken to multiple tasks in a row.

However, it is not until the third snapshot, which includes a larger sequential workflow, that more differences in SLOC comes into view. Here, no new SLOC changes are introduced in the CR-based system (13049 SLOC); the existing mixed
WorkFlow is altered to reflect the larger WorkFlow configuration. At the same time, the non-CR system takes on new endpoints and strings them along at the end of the other endpoints to provide the same functionality (13218 SLOC).

Finally, when introducing the choosing workflow, the non-CR system struggles to keep pace with the demand for making decisions on top of maintaining sequence and redirection logic; it clocks in at 13417 SLOC with the addition of four new endpoints that provide the redirection and logic necessary to implement the sequencing and choosing. In contrast, the CR-based system takes on a new WorkFlow class for the choosing workflow, as well as a new AbstractCrowdChoice class to encapsulate the logic for the choosing criteria. Once added, the system adds a new endpoint to manage that workflow via its independent URI and is placed at the home page next to the mixed WorkFlow. This snapshot of the CR-based system weighs in at 13080 SLOC.

With respect to workflow changes, the CR-based EPM system only requires 1 SLOC change for satisfying the change to swap the first and last task in the mixed workflow, as well as deleting the last task in the same workflow. In contrast, the non-CR system requires 26 SLOC changes in the first workflow change and 94 in the second change. These changes include redirection changes to other endpoints so that the pipeline order is preserved, as well as fixing the URL configuration in the Django project to ensure that the first and last task URIs are switched. These changes also include task implementation changes, or deletions in the second workflow change. In summary, the Crowdrouter illustrates its impact by promising code reductions as
crowd workflows get more complex. In an analysis involving a small system (SCR) and a medium-size system (EPM), it was shown that SLOC differences between a CR and a non-CR-version of a system increased when the complexity of the workflows also increased. However, in both systems, increases in the CR-cases were fairly minimal and did not grow as fast as that of the non-CR cases. This is due to the explosion in endpoint creation, redirection logic, session-management, and higher-order workflow operations like sequencing and choosing that are needed to implement the same features that are abstracted away in the Crowdrouter framework. This exemplifies impact as a function of how many lines of code were introduced to the system as more workflows and operations involving those workflows were introduced.

5.2.3.3 Summary: I have shown how both the SimpleCrowdRouter and EmergencyPetMatcher systems react to workflow additions and changes both with and without using the Crowdrouter. For both systems, an up-front cost in SLOC was required in order to integrate the Crowdrouter, but as more workflow additions were made, they experienced a significant reduction in SLOC (in the SRC case, a 75% reduction). Furthermore, with respect to workflow changes, non-Crowdrouter versions of the systems deleted or added more SLOC to reflect those changes than in the case of the Crowdrouter-based systems, where only 1 SLOC was needed to make the same change.

These results suggest the ease with which developers can satisfy the kinds of requests to crowdwork system maintenance. With these criteria, the results indicate the extent to which the crowdwork framework—in this case, the Crowdrouter—can
offer adaptability to crowdwork systems development. For instance, one can imagine
the crowd demanding changes to the crowdwork system, from providing logical
sequences of available tasks, to swapping one task with another one. Faced with these
changes, the developer can use the Crowdrouter to make these changes without large
ripple effects that affect the system in general, and this is due to the encapsulations
of crowdwork implementations that the Crowdrouter offers to developers.

With respect to the criteria used to evaluate the Crowdrouter’s impact, one
may also imagine measuring the duration of time taken to make workflow additions
and changes. This metric—along with SLOC differences—can then be compared
against other crowdwork frameworks such as PyBossa or Hive to determine how long
it would take a developer to make workflow additions and changes to a given
crowdwork system. For the purposes of this evaluation, timing was not included in
the analysis because of the intimate knowledge I have with these systems. Future
evaluation approaches should include timing along with SLOC differences as they
compare crowdwork framework integrations with systems that are not intimately
familiar with the researcher.

5.3 Range

Range, as I define it, is the latitude with which the Crowdrouter can serve a
variety of use-cases, and it is used to help measure versatility. However, can it satisfy
workflow and task implementations that are being hosted on existing crowdwork
systems? I ask this question to help measure range for the Crowdrouter framework.
I explore three popular crowdwork systems out in the public, describe their workflows and tasks, and then compare their use cases against the features that the Crowdrouter offers developers in building crowdwork systems. I derive these assessments by viewing these systems from the crowd’s perspective rather than from the developer’s perspective, since having access to their source code is not possible. However, most of time, a crowd’s perspective can accurately illustrate how tasks and workflows are served by the system, which is enough information to determine how the Crowdrouter can satisfy their use-cases. I wrap up the range argument by assessing a larger set of crowdwork systems and show that the Crowdrouter does indeed support their use-cases based upon their higher-level crowdwork properties.

5.3.1 GalaxyZoo. Managed by the Zooniverse citizen science portal, GalaxyZoo is a citizen science project designed to use the crowd to help classify images of galaxies. As of 2008, there have been 40M individual classifications made by more than 100,000 members (Lintott, et al., 2008), and that number has since increased to be measured as 350 years’ worth of human effort (Palet, 2014). Registered or non-registered, users can quickly get to work in classifying millions of images of galaxies, describing shape, size, and color-based features, and then being invited to participate in discussions related to each image before moving on to the next image.

The primary workflow in GalaxyZoo is participating in a continuous stream of image classification of galaxies. For every image that appears, users are asked a series of questions about the features of the image, including, “what shape of galaxy is seen in the center of the image?” and, “is the center rounded, boxy, or disc-shaped?”
As users answer the questions, they may become more specific, asking users to clarify their assessments using multiple choice questions. Once the image has been classified by the user, he is invited to participate in a forum to discuss what other users said about the image, requiring an account to provide comments. The user then is taken to the next image, and the process is repeated.

Assessing the structure of this workflow, it is essentially a non-ending iteration of a single task that is served to the crowd. However, I make a distinction between looping and uniform single task iteration. Providing higher-order sequencing operations on top of crowd task and workflow arrangements suggests that looping has a semantics where a series of tasks are served to the client as long as a condition, assessed after each iteration, is satisfied. Simple iteration of a task, as observed by the workflow in the GalaxyZoo application, serves the task over and over again without conditional semantics to cover cases where the application may serve different, non-uniform tasks or provide reward after a non-uniform sequence of tasks (e.g. classifying an image, then comparing two different images, then clustering images, as separate tasks) has been completed. However, in the context of a uniform, one-task workflow such as the one served by GalaxyZoo, this iteration is essentially the same as a looping mechanism for one task.

Given these simple semantics, the Crowdrouter can provide uniform, one-task iteration just as it is shown in the image classification workflow. Simply declare a Task class for image classification with GET and POST actions, a WorkFlow class that contains this Task once in its specification, and then provide an endpoint in the
application that routes requests to the WorkFlow and Task configuration. If there is a reward to earn after a certain amount of task completions, the WorkFlow class can inspect the number of times a particular user (or in the case of anonymity, an IP address) has performed these tasks, and if that number has reached a specified threshold, reward the user after the Task’s POST action has been completed.

While the Crowdrouter has yet to implement looping semantics, in most cases, crowdwork applications like GalaxyZoo implement a much simpler procedure for serving tasks to the crowd, which is that they serve the same task over and over again without conditional semantics imposed to keep track of non-uniform task sequences. In this respect, the Crowdrouter fully supports the task and workflow provided by GalaxyZoo.

5.3.2 MTurk. Amazon’s Mechanical Turk is a crowdsourcing marketplace platform for both requesters (i.e. users who create human-intelligence tasks, or HITs) and workers (i.e. users who perform those HITs). The economic structure is as follows: requesters are able to create HITs on the platform and specify the number of times that the HIT will be available for a specified price per HIT. Typically, the price ranges around $4-6 an hour, which is lower than the federal minimum wage in the U.S., where most workers are found (Martin, et al., 2014). These rates can vary, and they typically depend upon the criteria that assesses workers on their history and certifications before they can start working on the HIT. Additionally, MTurk as a platform offers quality control mechanisms for requesters; it statistically monitors the activity and performance of workers and offers those in high-standing a master’s
certification, which grants access to masters-only HITs created by requesters so that a high quality of work is supported.

Workflows that help structure the HIT vary in style, media, information, work, and sequence. Based upon personal experience in performing HITs, I have found that they typically include an embedded instructional video on how to perform the HIT that must be watched prior to completing a task. In performing a task, I observed that some HITs are basic surveys or questionnaires to fill out with multiple pages, or that they are a more interactive flash or Javascript-based task to provide annotations on images, such as in the case of identifying cancerous cells in microscopic images (Chandler & Kapelner, 2013). Once finished with the task, workers are prompted that they have made the amount of money owed to them, and they are either invited to perform the HIT again, or in the case where no more HITs are available, they are returned to the main page to browse other HITs.

There are a variety of different HIT structures offered through the MTurk platform. A requester can choose to create HITs based upon predefined structures for categorization, data collection, moderation of an image, sentiment, survey, and image tagging, among others. As an observer, I can assume that these structures are predefined with the platform and can most reliably be mapped to Task implementations within the Crowdrouter. As for sequencing or providing higher-order workflow operations (i.e. branching, sequencing, and looping) on top of such tasks, however, to the best of my knowledge, MTurk does not offer such semantics to requesters. Instead, requesters can create multiple HITs, each relating to a Task, and
declare them as the same HIT type, which is defined by the work that is requested for workers to perform. Declaring HITs with the same HIT type will allow the platform to group them and make such HIT types searchable, increasing the likelihood that the crowd will perform them simultaneously, but not in a sequential fashion.

As for defining the properties of a HIT, the Crowdrouter still cannot encode many of the properties offered by MTurk to requesters. These include the amount of runs (i.e. executions) available for the HIT, the amount of pay assigned to each run, any certifications that are required to perform the HIT, and the amount of time allowed to perform it. Crowdrouter Tasks may encode these properties at the will of the developer, however, since Task classes are essentially subclasses of the AbstractTask and can define its own properties, but to date, there is no native support for enforcing such task-based constraints as offered through the MTurk platform.

In sum, MTurk offers a large marketplace for workers to perform HITs as independent tasks and make money doing so. The Crowdrouter can provide features for workflow operations that MTurk currently lacks, including the ability to repeat a specified amount of the same HIT, perform the same HIT ad nauseum, branch HITs based upon the results gathered from the previous HIT, and pipeline multiple different HITs together. These features are desirable from the perspective of the requester if she wishes to assign and arrange a diverse set of tasks that are contextually important, as recent research supports small delays between contextually-relevant HITs (Lasecki, et al., 2015). However, with respect to the
atomic unit of work (HITs) and the properties offered by MTurk to control its assignment and execution, the Crowdrouter falls short of providing these features. However, these features are easily encoded within the current framework and can even be defined within application logic as it routes such task-based requests from the crowd.

5.3.3 CrowdCrafting. Crowdcrafting (http://www.crowdcrafting.org) is an example crowdwork website built using the PyBossa framework for handling a variety of media-driven tasks. Through the structural configurations offered through PyBossa, Crowdcrafting is an out-of-the-box example of scaffolding a crowdwork system with a sample of crowdwork tasks available to perform by anyone, registered or unregistered. Based on the Crowdcrafting statistics page, it has accumulated a total of 42K users (~30K anonymous, 12K registered), 2044 projects (542 published, 1502 drafted), and more than 680,000 tasks with more than 2M task runs. Crowdcrafting, along with the Micromappers platform (described in the Background and Study chapters), are two notable examples of systems that have been built using the PyBossa framework and have received note-worthy use.

As a registered user on Crowdcrafting, I can perform work on other registered users’ projects that have been published, or I can create my own project. Once a project is created with its finer details, including name, description, purpose, and contact information, Crowdcrafting allows users to create tasks using a variety of offered templates for creating surveys, classifications, identifications, and countings. This mostly depends upon the media source a project-creator elects to use, as well as
where the media can be found (options for uploading media as links from the Amazon Simple Storage Service (S3), Google’s photo service Flickr, and the file synchronization service DropBox are available).

Once a task has been created, users are granted a number of various configurations for a task. The Task Presenter allows the project-creator to design the front-end templates according to the task specification described above; the Task Scheduler offers options for delivering tasks to the crowd; the Task Priority configuration allows users to weight the tasks, which affects the scheduling order; and the Task Redundancy option, which specifies how many times tasks are to be completed by the crowd before being finished. Most relevant to the features offered by the Crowdrouter are the Task Scheduler, Priority, and Redundancy options, so I will describe these in turn.

The Task Scheduler uses algorithms to create task assignments for users who are interested in performing work on a project. Options for scheduling tasks include depth-first (i.e. tasks with runs that have not been completed), breadth-first (i.e. tasks that have the least number of task runs), or random (i.e. random assignment irrespective of task run count). Choosing one algorithm over another affects the way in which task runs are assigned to the crowd. This is a feature set not currently available within the Crowdrouter because it does not yet support Task or WorkFlow runs that can be assigned to the crowd. However, the Crowdrouter does have facilities for keeping task and workflow visits in a separate database (I describe this in more detail in the Crowd Statistics section in Chapter 4), which can be used by the
developer to re-route requests to the Tasks that have lower visiting counts than other Tasks. Additionally, the Crowdrouter can in fact support random assignment by using the WorkFlow API to randomly assign a Task from a WorkFlow specification.

Task Priority options available on Crowdcrafting enable users to weight the tasks according to a number between 0 and 1. Tasks with higher priority influence the frequency at which the scheduler assigns those tasks. Currently, the Crowdrouter does not support this feature, but again, it has the flexibility to allow WorkFlow specifications to implement this feature easily. For example, if a WorkFlow contains three Tasks (RankTask, GeoTask, and VerifyTask), then a developer can implement a random roll to allow the RankTask to have a 66% chance of executing, while the GeoTask has a 20% chance, leaving a 14% chance that the VerifyTask will execute. These implementations are easily inheritable as well, giving developers a chance to share WorkFlow classes with others.

Finally, the Task Redundancy option allows project-creators to specify the number of task runs for each created task in the system. This is a feature that the Crowdrouter does not natively support, but with its crowd statistics feature, it allows developers to use task visits as a number to create execution-based thresholds for not just Task but WorkFlow and CrowdRouter objects as well. This allows more hierarchical management of crowdwork executions, as opposed to managing counts at a task (i.e. granular) level.

In summary, Crowdcrafting (and therefore, PyBossa) is an effective tool for hosting crowdwork projects quickly and without needing to code. If users are
development-savy, they have the ability to customize look-and-feel and task design for their own use-cases. However, with these benefits come limitations. PyBossa is a heavy-weight framework requiring much configuration for hosting projects, creating tasks, and deploying them from various defined media sources. The Crowdrouter, on the other hand, avoids these limitations by giving direct control to the developer to create and manage workflows and tasks with their own custom implementations, allowing easy integration into existing applications or into new ones. Although the Crowdrouter falls short on providing native support for task scheduling, weighting, and counting, it provides the appropriate abstractions for creating support for these features and also sharing those implementations as subclasses of the Crowdrouter entities.

5.3.4 Crowdwork System Comparisons. I now examine at a higher level a larger set of crowdwork systems, including the three systems that were discussed above. Here, I make the claim that the Crowdrouter tool can support the range of domains, tasks and workflows, and media types that these systems call for. Table 2 tabulates these crowdwork system properties.

As a point of clarification, a task is defined as a unit of work that combines elements of client and server to send and receive work from the crowd. There are many examples in crowdwork systems where this becomes ambiguous, especially in cases where the interface may change after completing some work, which might lead to more work, but this situation does not indicate a separate task, since it did not use asynchronous client-to-server communication or server-side logic to present it.
Instead, the update was only a part of the overall task itself. A workflow, on the other hand, is essentially a logical grouping of one or more tasks. I assume here that tasks are always grouped by at least one workflow, independent of whether the authors/creators/maintainers of the following crowdwork systems intentionally create such groupings. This makes the comparisons between these systems easier, especially when source code access is not possible. I make these pedantic clarifications for the purposes of reasoning about the features that the Crowdrouter provides for arranging tasks and workflows at a higher level of abstraction.

Here, it is surprising to see the low number of tasks and workflows that are custom designed for thousands to millions of members of the crowd to perform. In GalaxyZoo and its sibling Jungle Rhythms project (a project to classify scientific

<table>
<thead>
<tr>
<th>System</th>
<th>Domain</th>
<th>Crowd Size</th>
<th># Tasks</th>
<th># Workflows</th>
<th>Workflow Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>GalaxyZoo</td>
<td>Astronomy</td>
<td>Very Large</td>
<td>1</td>
<td>1</td>
<td>Single</td>
</tr>
<tr>
<td>Jungle Rhythms (Zooniverse)</td>
<td>Phenology</td>
<td>Very Large</td>
<td>1</td>
<td>1</td>
<td>Single</td>
</tr>
<tr>
<td>MTurk</td>
<td>Crowdwork</td>
<td>Very Large</td>
<td>Many</td>
<td>N/A</td>
<td>Single</td>
</tr>
<tr>
<td>Crowdcrafting</td>
<td>Crowdwork</td>
<td>Large</td>
<td>Many</td>
<td>N/A</td>
<td>Single</td>
</tr>
<tr>
<td>MapMill</td>
<td>Image Ranking</td>
<td>Medium</td>
<td>1</td>
<td>1</td>
<td>Single</td>
</tr>
<tr>
<td>OpenStreetMap</td>
<td>Mapping</td>
<td>Large</td>
<td>1</td>
<td>1</td>
<td>Single</td>
</tr>
<tr>
<td>EmergencyPetMatcher</td>
<td>Pet Reporting and Mapping</td>
<td>Small</td>
<td>5</td>
<td>4</td>
<td>Single, Sequential</td>
</tr>
<tr>
<td>Project BudBurst</td>
<td>Citizen Science</td>
<td>Medium</td>
<td>2-4</td>
<td>2</td>
<td>Sequential</td>
</tr>
<tr>
<td>Google Forms</td>
<td>Surveys/Questionnaires</td>
<td>Small-Large</td>
<td>Many</td>
<td>1</td>
<td>Single, Sequential, Branching</td>
</tr>
<tr>
<td>CrowdFlower</td>
<td>Crowdwork</td>
<td>Large</td>
<td>Many</td>
<td>Many</td>
<td>Single, Sequential, Branching</td>
</tr>
</tbody>
</table>

Table 2: Crowdwork Systems and their Crowdwork Properties
observations made for tropical tree phenology from the African rainforest), the task and its workflow provide a seamless, independent experience for the crowd to focus on. In these 1-1 workflow-to-task cases the Crowdrouter excels in providing support; simply declare the Task and WorkFlow class and serve them to the crowd. In cases where the tasks are repeating ad nauseum, such as in GalaxyZoo and Jungle Rhythms, this should not be confused with looping, which imposes sequential semantics with conditional semantics. To date, I have not yet observed looping semantics within crowdwork systems.

However, some crowdwork systems exhibit different tasking behaviors. Systems like EmergencyPetMatcher, Project BudBurst, Google Forms, and CrowdFlower all share sequential (i.e. 1-n) tasking behaviors in which a set of tasks must be completed within a workflow in order to consider the crowdwork complete for each user. For example, in EPM, there are two tasks that work together to successfully propose a pet match on the system: one, a user must choose a candidate pet report whose details are similar enough to be considered a match with the pet report currently being worked on, and two, he must confirm the linkage between these two reports to create a pet match. On Google Forms, a form-creator can create multiple page breaks in a new survey, which creates a sequential workflow with two separate tasks because of two instances of client-server interaction before completing the workflow. Project BudBurst allows citizen reporters to report on trees and shrubs in their local areas, but they must do so by first specifying the the group under which the observed tree falls, then its species, and then its location on a map.
The Crowdrouter works well with sequential cases because it was designed to handle them with its pipelining features. If developers need to add an extra task or remove an existing one within a sequential workflow, it is easy to include (or exclude) it into the WorkFlow specification. In cases where a workflow must be completely substituted with another workflow, again this is a simple use-case that the Crowdrouter satisfies; just replace the WorkFlow name within an existing controller action. If some logic is needed before, after, or in between the completion of tasks, the Crowdrouter supports an API for developers to implement continuations throughout a pipeline.

Finally, with respect to branching, there are a few crowdwork systems that support it. In the CrowdFlower case, the system runs a “Quiz Mode” that attempts to assess the level of accuracy that each worker puts into a task before starting it. Akin to MTurk’s method for evaluating workers based on certifications, this Quiz mode ensures a higher quality of work that the system then relies upon to maintain its customer base. If a worker fails the sample task under this mode, then the server essentially expels the worker from performing the task, which indicates a level of branching with the criteria to assess the results of a previous task (“Quiz mode” task) before continuing on to perform the real task at hand. Google Forms also supplies branching semantics as well; form-creators can provide logic in a specified question within the survey that determines the page to which users will navigate after completing the current page.
The Crowdrouter readily supports branching semantics within pipelines, as well as non-pipelined workflows. Developers can implement the conditions under which CrowdFlower disallows workers exhibiting poor accuracy within a WorkFlow specification that begins with a “Quiz Mode” Task. Within the specification, a developer can provide a subclass of the AbstractCrowdChoice, which provides branching opportunities for executing one Task T1 over another Task T2 (in this case, a redirection back to the home page, or the real Task to perform). For Google Forms, developers can create an AbstractCrowdChoice implementation if a question is targeted with a particular condition, which will send users to a new Task if that question is answered in a specific manner.

In summary, these crowdwork systems exhibit interesting crowdwork properties with respect to task and workflow arrangements. Some of them show single-based 1-1 workflow-task arrangements, others show sequential 1-n arrangements, and still others add branching on top of sequential arrangements. In the case of n-n, meaning many-workflow-to-many-tasks, the complexity does not necessarily increase; the Crowdrouter is flexible and extensible enough to handle cases where Tasks are reused by many varying WorkFlow classes to which the crowd is routed. This exemplifies a range of crowdwork properties that the Crowdrouter is equipped to solve.

However, the Crowdrouter framework has many more crowdwork use-cases to solve. In the case of examining crowdwork systems in more detail—such as MTurk and Crowdcrafting—it is evident that there are features that are missing within the
framework, including mechanisms to weight, prioritize, and count task and workflow executions. These are significant use cases that will be flagged for future development, but the Crowdrouter does offer some flexibility in having developers implement such use-cases manually and to be able to share that work with others, given that Crowdrouter entities are typed and can be used as a “marketplace” for various crowdwork solutions.

5.4 Crowdrouter Versatility

In summary, the Crowdrouter provides a certain degree of impact and range that together form the basis for versatility, a central concern articulated by participants in the crowdwork study. With respect to impact, the framework shows valuable reductions in SLOC as more workflows and tasks are introduced into a crowdwork system. This demonstrates the benefit Crowdrouter provides by treating tasks and workflows as first class concepts, increasing the level of abstraction to let developers treat crowdwork as a first-class concern, rather than as an invisible challenge to solve within an application. Having evaluated two systems—EPM and SimpleCrowdRouter—that used the Crowdrouter to encapsulate their workflows and tasks, integration of the Crowdrouter was simple and straightforward, reinforcing the claim that it is a versatile and lightweight framework.

With respect to range, the Crowdrouter shows flexibility and extensibility in handling various configurations of task and workflows from a survey of existing crowdwork systems. These systems show certain categories of crowdwork complexity,
including single, sequential, and branching operations atop tasks that are served to the crowd. After examining three of these crowdwork systems in more detail, I have shown that there are features—task weighting, prioritizing, and counting—that would be desirable to have within the Crowdrouter framework and to support a wide range of development for crowdwork as it relates to task design.

Together, impact and range highlight the versatility that the Crowdrouter provides, allowing it to easily integrate into existing or new systems, and to satisfy a wide range of crowdwork issues with respect to workflow and task arrangements. This suggests future work in identifying attempts to model other crowdwork challenges to design solutions within tools and frameworks. For example, if crowdwork research emphasizes the importance of supporting the ability for members of the crowd to recognize their impact in performing work, how can crowdwork tools support that ability? Crowdwork should be viewed as an engineering technique that developers can use to productively construct crowdwork systems. The Crowdrouter is a step in that direction.

Having presented an evaluation of the Crowdrouter framework with respect to versatility, I now turn to presenting my conclusions.
CHAPTER 6

CONCLUSIONS

In this chapter, I make final remarks on the work discussed in this dissertation and reflect upon the nature of crowdwork and future implications for tools and frameworks that support the design and development of crowdwork systems. After having tied together results from my crowdwork study and the features of the Crowdrouter into a framework for evaluating versatility, I now discuss future work that investigates how frameworks can be more productive for crowdwork developers.

This dissertation work has unpacked a variety of issues related to crowdwork that convey the importance of constructing crowdwork systems with the crowd in mind. To properly solve these issues, tools and frameworks should treat crowdwork as socio-technical systems (Kittur, et al., 2013) constituted primarily of issues surrounding the crowd (motivation, progress, adoption, trust, quality of work, etc.) and the system hosting the crowd (scalability, reliability, responsiveness, usability, etc.). Feedback loops between the role of the system and the crowd can be constructed to ensure that members of the crowd have an awareness of what the crowdwork will achieve and what the system must do to adapt to varying levels of crowd motivation, skillset, and accuracy.

As was revealed by the crowdwork study, some developers believe that crowdwork and machine computation must complement each other in order to solve the most challenging problems. Hence, formalizations for performing crowd
processing are needed so that tools and frameworks can process crowd and machine computation seamlessly, making more predictable the variations from the crowd and more optimized the work from the machine. Current crowd systems follow this line of thinking (Kittur, et al., 2011; LaToza, et al., 2014; Bernstein, et al., 2010) as they model crowd processing as a distributed computing paradigm much like MapReduce, and for their specific cases, dynamic or intelligent allocation of tasks is effective.

However, as revealed by multiple sources in pecuniary-based crowd markets such as MTurk (Martin, et al., 2014, Chandler & Kapelner, 2013), this has some limitations in sustaining motivation for the crowd. Creating motivation via monetary reward without contextualizing the scope of the problem leads to lower accuracy and quality of work. While it may be true that one approach to counteracting these negative effects is by introducing quality control to verify crowdworked tasks, this is only a partial solution to the inherent complexities that come with including humans in the loop. Treating human computation as a distributed computing model leads to Tayloristic perspectives that are tangential to the non-functional factors that produce higher-quality crowdwork systems.

Instead, crowd algorithms to enhance speed, accuracy, or quality of work should reframe its focus to take on a more macro view of the system under which they are implemented. They should address motivational factors, such as providing feedback loops to present and encourage work, provide collaborative opportunities among members of the crowd, display task and workflow progress, and make visible the impact that the crowd is making on system objectives. Reframed with the crowd
in mind, machine processing can increase throughput while increasing motivation for the crowd to perform work.

But this gets at a more pressing debate about the symbiosis of human and machine computation. While it is true that humans are innately proficient in particular tasks, such as comparison and identification, they will never be as proficient in other tasks as machines, such as estimation or aggregation. One study participant claimed that the gap between crowd and machine processing is shrinking, allowing machines to soon overshadow the capabilities with which humans have been gifted. Viewed in this perspective, crowdwork may just be a temporary solution for computational processes that are running to catch up and create more efficient, robust solutions. A future of crowdwork, tangential to Kittur and others’ visions (2013), that replaces humans with machines may indeed flow in that direction.

However, if we instead look to Licklider’s theories of man-computer symbiosis (1960), they reveal a cooperative linkage between our natural capabilities and those of the tools we build. They can complement each other by building upon strengths and weaknesses. Crowdwork builds upon the strengths of humans and leverages them to allow machines to solve important problems for society. It is crucial that, given the tools and frameworks to support this process, and given the fact that one day it might be true that machine processing overshadows human processing, that the tools and frameworks we build do not overlook the challenges that humans face presently in an online, exploitative environment.
This suggests a new generation of tools and frameworks to support crowdwork system development. New frameworks, such as the Crowdrouter, should not only pay attention to the concerns involving the crowd and the system, but also ones involving the *developer*, right in the middle, who needs to cope with crowdwork complexities and is looking for robust and reliable support. Versatility is a key challenge and objective; tools that can fit into various use-cases and application domains will prove reliable and useful, and will increase adoption for future crowdwork efforts. Future work needs to investigate how tools like the Crowdrouter behave in the wild, addressing concerns that the developer has with them, how they influence the design of the system, and how they address the concerns of the crowd. These, I submit, are the most important avenues for future research work in the intersection between crowdwork, human-computer interaction, and software engineering. They bring together the elements of form and context for the pursuit of better, more informed crowdwork.
BIBLIOGRAPHY


Gamma, E., Helm, R., Johnson, R., & Vlissides, R. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.


APPENDIX

SEMI-STRUCTURED INTERVIEW QUESTIONS

Demographics, General Programming Experience

• How many years of professional programming experience have you had?
  (professional meaning working at a company and job being software engineer)
• How many years of programming experience have you had in total (including non-
  professional - casual, amateur)
• If you could narrow down your specialty to one topic within software development, what would it be?
• What application domain of programming/computing would you declare yourself as in?

Crowdwork Experience

• What crowdsourcing programs (applications/websites/frameworks/stand-alone programs) have you seen, used, or heard about?
• Which ones have you actually used?
• Have you built any crowdsourcing programs?
• For the ones that you have used, tell me about the workflow (i.e. the task or tasks that you were instructed on performing).
• Did you perform multiple workflows?
• What did you notice about the workflows you performed?
  Were they straightforward or confusing?
  Did they include big-sized tasks, or much smaller bits of work?
  Did they have a reward, or was it free labor?
  Did they have creative work, or was it mindless work?
• Do you think that programming crowdsourcing apps is different from programming non-crowdsourcing apps?
  What problems do you think developers face when building crowdsourcing apps? Unique?
Brainstorming Session

• Tell participant about brainstorming a real-world crowdsourcing problem. Goal is to find software abstractions that might be useful for the problem. Remind them to use whiteboard/google doc to draw and write ideas.

• Imagine yourself as the developer behind a new crowdsourcing website for ranking images of devastating areas hit by a recent hurricane. Work out how you would design an example workflow on it (in coding if proficient, whiteboard/talking otherwise).

• How would you architect the system for crowdsourcing?

• What components would be important to have to make these workflows easy and/or versatile?

• How likely are you to pick up a crowdsourcing framework if and when you start working on your own crowdsourcing project? What features would you like to have in a crowdsourcing framework?