Towards Providing On-Demand Expert Support for Software Developers

Yan Chen[1], **Steve Oney**[1], **Walter S. Lasecki**[2,1]
School of Information[1], Computer Science & Engineering[2]
University of Michigan – Ann Arbor
{yanchenm,soney,wlasecki}@umich.edu

ABSTRACT
Software development is an expert task that requires complex reasoning and the ability to recall language or API-specific details. In practice, developers often seek support from IDE tools, Web resources, or other developers to help fill in gaps in their knowledge on-demand. In this paper, we present two studies that seek to inform the design of future systems that use remote experts to support developers on demand. The first explores what types of questions developers would ask a hypothetical assistant capable of answering any question they pose. The second study explores the interactions between developers and remote "experts" in supporting roles. Our results suggest eight key system features needed for on-demand remote developer assistants to be effective, which has implications for future human-powered development tools.

Author Keywords
Pair programming; intelligent assistants; crowdsourcing

ACM Classification Keywords
H.5.m Information Interfaces and Presentation (e.g. HCI): Miscellaneous; K.6.1 Management of Computing and Information Systems: Software Development

INTRODUCTION
Software development is an expert task that requires complex reasoning skills within the bounds of formal language syntax and Application Programming Interface (API)-specific functionality. Support for recalling keywords, function names, argument types and methodologies, and other details can come from a variety of sources. Current support usually comes either from tools built directly into Integrated Development Environments (IDEs) or from other developers.

IDE tools provide contextualized, easily-accessible, and on-demand support for developers, but are generally limited in the types of feedback they can provide (e.g., syntax error highlighting and function auto-complete) because the system cannot truly understand user queries or the context of the problem. We find that these limitations preclude many questions that developers *would* ask while programming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CHI'16, May 07-12, 2016, San Jose, CA, USA
Copyright © 978-1-4503-3362-7/16/05 $15.00.
http://dx.doi.org/10.1145/2858036.2858512

To overcome the limitations of automatic approaches, support from other human developers is often enlisted. This can take several forms, each with their own trade-offs. In a work environment, an expert colleague can provide useful support in the specific languages and frameworks in use, but is unlikely to be available on demand or able to support frequent questions. Web forums can provide a wealth of information about general questions, but typically do not provide highly personalized support or quick responses to developer queries. Additionally, many of the most successful developer forums restrict the types of questions that can be asked and when they can be asked. For example, Stack Overflow [41] explicitly discourages the types of project-specific code requests that we observed in our motivating studies. It also requires some level of reciprocity from users in the form of useful answers to other developers' questions. Providing sufficient context for others to help solve a problem can also be a time-consuming task that often takes multiple turns of interaction to complete.

Hired freelance developers can provide tailored support for specific questions, immediate answers, and even the ability to hand off sub-tasks for independent completion. However, in addition to the monetary price of hiring a freelancer, the traditional hiring process adds significant time and preparation effort costs (interviews, initiation, etc.). Lastly, none of these expert solutions provide the in-context support that IDE tools do, adding the need for an additional context switch to and from a developer's workflow.

The rise of online expert crowd platforms, such as Upwork [14], allows on-demand, programmatic hiring of developers for pay. However, the complexity of the hiring and onboarding process needed to find reliable workers who have sufficient knowledge of the project remains unchanged. Reducing the hiring overhead is critical to making expert support in development processes feasible. Ideally, asking for help would be as simple as informally saying the question at hand, and providing in-context support within an IDE.

In this paper, we introduce a new space of support systems and present two studies that inform the design of such systems in the future. In the first, we explore the types of questions developers might want to ask an on-demand support agent. In the second, we examine the logistical challenges of providing on-demand remote developer support.

Contributions
Our studies demonstrate the need for systems that can answer developer queries during a live coding session, and explore the challenges that arise in this domain. Specifically, we make the following contributions in this paper:

- A "hypothetical assistant" study of what questions developers would ask when they are programming if complete support was provided
- The first study of how people ask for help when on-demand, reactive expert help is available during a Web development task
- Design takeaways and recommended features for systems that aim to recruit on-demand support for developers

With this work, we present the technical challenges and design opportunities of instant expert support. We also hope to catalyze further work in the community on this topic.

MOTIVATING SCENARIO
To illustrate the potential advantages of the features we discuss, consider the case of Anita, a software developer working on a project that she would like to release as open-source.

Scenario
Anita begins development using an IDE with on-demand support enabled. As she codes, the IDE reminds her of code blocks that need comments to be understandable to others.

After implementing the basic structure of the client for the tool she is building, she decides to integrate an API call to a service that she has never used before. Instead of breaking from her coding task to start looking into online documentation and forum examples, she requests (from her IDE) expert help to write the function that contains this API call, and asks that the response explain the implementation decisions and functionality so that she can better learn the API for the next time she needs to use it. While the operation completes based on a simple spoken natural language request, she continues to develop her tool as she otherwise would have.

When feedback arrives that her function is written, she checks it and sees that the support developer has left extra details about the implementation because it used a software design pattern that she had not used before very frequently. After understanding the code, she approves the automatic integration into her current code, and then resumes building her tool.

Later, when Anita is unsure of why a returned value is not being parsed as expected, she makes another request to an expert to see if they know anything more about this particular library function. Within seconds, a support developer who is experienced with the frameworks and libraries being used in this part of the tool is looking at Anita's code and explains via chat that a segment of code that they have highlighted would need to pass an additional parameter to the function in question in order to get the right return type. Anita adds the fix and thanks the developer, all without every leaving her IDE window. With her tool completed faster and with fewer distractions than she expected, Anita posts it online.

Comments and Discussion
Since Anita does not have to interrupt her workflow to find information and delegate tasks, she is less distracted and more productive. She also does not have to lose time specifying tasks in complete detail. Instead, the system automatically extracts additional information based on code structure, previous documentation, and her interactions with the code. The ability to specify when the remote developer should provide additional details of a solution helps provide learning benefits when desired, while not wasting time or helper effort when the explanation is not needed. Additionally, because the system recruits experts on demand, Anita is only billed for the time and support that was actually needed. This is in contrast to the naive solution to providing on-demand support by keeping a single developer on retainer for the entire session.

Challenges
Computers alone will not be able to provide the functionality described above any time in the near or medium-term future. The natural language understanding, program synthesis, and problem solving components of this system are each well beyond what is currently possible with AI alone, and the combined challenge is disproportionately harder to solve.

In order to overcome this, we need to leverage human intelligence and expertise. However, little is known about what people would actually choose to ask if the scope of support tools was not limited a priori by the system, or what challenges human experts face when asked to provide on-demand support. Increasing understanding of these two aspects of the design space will allow system builders to better reason about the challenges that need to be overcome to create fast, usable, and efficient assistance tools for software developers.

BACKGROUND & RELATED WORK
This work builds on previous research on collaboration and expertise-finding in the context of software development.

Finding Expertise in Software Development
The ability to identify and communicate with domain experts is an important determinant of software developers' effectiveness. One study found that effective engineers tend to communicate more frequently with experts who are outside of the engineer's domain of expertise [3]. Finding helpers is a mutual process which not only requires helpers to have the relevant expertise, but also requires help-seekers to provide the right contextual information.

Community Question Answering
A number of Community Question-Answering (CQA) websites allow software developers to post questions to a large community. The most widely used CQA website for developers is Stack Overflow [41]. In addition to providing a forum where developers can ask questions, CQA sites aid developers by providing a repository to answers of previous questions. Answer Garden [1, 2] pioneered work in this area by building an "organizational memory" through a growing database of questions and answers. However, we find that many of the types of questions that developers would prefer to be able to ask on-demand experts are not appropriate for CQA sites for

a number of reasons. First, many of the questions that developers asked during our pair programming study required that answers be immediate in order to be useful. On Stack Overflow, it takes a median of six hours for users to receive a response to their question that they will accept [36].

Further, it can take significant time and effort to compose a question that is appropriate for a CQA site. To formulate an effective question (one that will eventually be answered) developers must start by capturing all of the potentially relevant aspects of their project and system setup. This step can preclude asking questions where the relevant context is difficult to capture. In fact, one of the main reasons for questions to remain unanswered is that they are too short or do not provide enough context [4].

By contrast, we show that an on-demand expert support system for developers should be able to automatically capture developers' work context to handle otherwise ambiguous questions. Developers should be able to point to a snippet of code, ask "What does this parameter do?", hand off execution of planned coordination to the system, and receive a meaningful response within minutes.

Real-Time Crowdsourcing

The model and features we propose in this paper rely on the ability to quickly recruit experts in order to improve question response time. Pera and Ng have shown that CQA sites can improve their algorithms for question-matching, which would help developers find relevant archived answers [42]. Another way to improve the response time of CQA sites is to route questions to appropriate experts, as Riahi et al. propose [44]. However, both of these techniques still require that developers spend time to carefully formulate their questions while including the relevant contextual details.

Instead, we propose building on previous techniques for real-time crowd recruitment. VizWiz [7] introduced model for eliciting real-time responses from the crowd by keeping workers engaged in example tasks until their assistance is needed in real time. This model can elicit responses from the crowd in a matter of seconds. Adrenaline [6] takes a similar approach by queuing idle workers so they may complete other tasks while they wait. LegionTools [28] is the first task-independent open-source tool for recruiting and managing real-time crowds. Legion extends the real-time crowdsourcing model to include continuous tasks — tasks that span as long as the worker chooses to stay engaged — over small, disjoint microtasks. This helps retain context and greatly improves answer latency.

Aardvark improves its response time by routing questions to experts who are currently online [21]. Chorus [31] enables on-demand conversational interaction by recruiting multiple workers for conversational interactions with users. Apparition [29] enables prototyping interactive systems in real-time by introducing self-coordination mechanism to reduce task conflict among workers.

By combining these techniques with an IDE-integrated communication mechanism, future tools can enable quick and meaningful responses to software development questions.

Expert Guidance

Beyond CQA websites, several commercial systems enable more personalized mentorship for software developers. Code Mentor [23] and hack.hands() [25] allow software developers to create requests that connect them with experts (as judged by self-report and community reviews) quickly. Help requesters and experts can communicate through a shared code editor, chat window, and video chat. Based on the findings of our studies, we propose a "per-question" assistance model rather than the mentorship model introduced by these systems. In a per-question model, requests for help can be asked and answered asynchronously rather than requiring the developer to start a new help session for every question. Further, this model would allow the question to be routed to multiple potential experts; if one does not know the correct answer, another can respond. This is unlike the mentorship model, where if a code mentor does not know the correct answer to a question, the developer must initiate a new session. Finally, the "per-question" model would enable better expert selection because unlike the mentor model, the question itself would be known before connecting the help requester and expert.

Beyond Answer Garden (discussed above), several systems have proposed systematically routing help requests to an appropriate expert. Expertise Recommender [37] and SHOCK [35] both include mechanisms to route questions to users who are capable of answering them. Expertise Recommender relies on collaborative filtering to build expert profiles, whereas SHOCK automatically builds expertise profiles based on tasks the user has performed on the Web. The designs of these systems may help guide the question-routing architecture for our proposed question-routing system.

In the domain of software development, Mockus and Herbsleb proposed Expertise Browser [39] as an approach to find experts in the context of a shared project. It relies on repository commit data from prior projects to quantify expertise in the context of geographically distributed development teams. However, we propose creating systems where software developers can seek guidance from outside of a project team, which requires a different method of determining expertise.

Programming Team Communication

Despite their reputation for preferring solitude, software developers who work in teams can spend up to half of their time communicating [20]. Team organization and communication mediums play an important role in the effectiveness of communication between developers. The model of task delegation system we propose in this paper is related to Harlon Mills' idea of a "surgical team" development model where the developer with the most experience with a particular task delegates more mundane tasks to other developers [38, 11]. On-demand code assistants would enable developers to dynamically create such supporting roles without the need to be part of a formal organization.

Pair Programming

Pair programming is a method in which two people work together side-by-side at one computer [12]. Pair programming has been shown to yield better design, more compact code, and fewer defects for roughly equivalent person-hours [51], but it requires the collaborators to be available for long sessions, even when minimally needed, which is inefficient.

Distributed pair programming is a derived version of pair programming that uses a dedicated IDE to allow every participant to edit the same code locally [5, 46]. Although the distributed pair programming approach removes the issues of distance work and dispersed team [40], how to coordinate the work and maintain the context for both old and new participants still remain largely unexplored. Our studies instead aim to automate coordination by "dropping" developers into a task long enough for them to solve the problem and move on.

Team Information Needs
A number of researchers have also categorized the types of questions programmers asked in different contexts. Sillito et al. described 44 types of questions programmers ask when evolving a large code base [48]. Ko et al. categorized six types of learning barriers in programming systems for beginners and proposed possible solutions from programming system sides [27], and also documented communication amongst co-located development teams [26]. Guzzi et al. analyzed IDE support for collaboration and evaluated an IDE extension to improve team communication [18].

Whereas these studies of information needs focused on existing team structures, this paper introduces a new path for information seeking via on-demand expert support, and the studies presents qualitatively different data and implications. Unlike existing team structures, this paper proposes a team structure where a project stakeholder requests remote help from experts who are not stakeholders. This difference has significant implications for team trust, communication preferences, and context sharing.

Collaborative Development
Systems like Codeopticon [16] and Codechella [17] provide ways that helpers (i.e., tutors/peers) can efficiently monitor multiple learners behavior and provide proactive on-demand support. Commercial IDE tools such as Koding [49], Codenvy [22], and Cloud9 [24] enable users to code collaboratively online in real-time. Although these systems reduce many of the barriers developers face when working at a distance [40] and time spent on environment configuration, they do not support the case when developers are actively seeking help [50].

IDE-Integrated Help Finding Tools
The on-demand support systems we propose based on the studies presented in this paper are forms of Recommendation Systems in Software Engineering (RSSEs) [45]. Unlike most CQA websites, they would accept questions and provide answers within the context of a developer's IDE. This would enable a question routing and capturing mechanism to leverage previous code context to improve the specificity of the question and the ease of integrating answers back into developers' code.

Previous RSSEs have helped developers capture the answers to different kinds of questions. Through laboratory studies, Brandt et al. found that programmers rely heavily on example code from the Web [10, 9]. Brandt used these studies to guide the design of Blueprint [8], a system that allows programmers to rapid search for a query in an embedded search engine in their local IDE. Hartmann et al. also explored ways to aid developers in recovering from errors by collecting and mining examples of code changes that fix errors [19]. However, the results of the studies presented in this paper show that developers have broader information-finding needs that can be difficult to respond to through solely automated means.

Our studies suggest that multimodal interaction, such as blending code highlighting with natural language audio descriptions, may help enrich the context of developer requests. This idea has been explored in different contexts [13], but not yet in the context of creating virtual programming pairs.

Crowdsourcing
Most work has investigated how to improve crowd workflow by converting parallel tasks to series tasks [46], and letting the crowd workers guide the workflow [34]. However, little work has focused on exploiting expert crowds to help with complex tasks such as programming in real time. The design findings presented in this paper will help guide systems that can coordinate crowd experts in real time.

Crowd-Enabled IDE Tools
A few tools have enabled crowds to aid in development tasks. Latoza et al. [32] developed CrowdCode, which decomposes programming into self-contained function-based microtasks. In CrowdCode, clients make requests to the crowd with self-written specifications of the desired function's purpose and signature. Crowd workers are then automatically assigned to these tasks. However, such a system is limited in how much it can reduce the end user's time expenditure, since the request authoring process requires a detailed problem specification.

Having the crowd to efficiently assist programming has been challenging in terms of qualified worker recruitment and efficient work evaluation. Collabode [15] allows users to define function-based microtasks, including testing and debugging a function, which allows workers to evaluate the previous work. It also provides workers the choice of skipping the microtasks which do not match with workers' skill sets. We propose recruiting helpers and allowing them to choose the tasks, allowing users to decide whether the answer is valuable.

Expert crowds
Flash teams [43] introduced a framework that allows users to authorize modular tasks to link expert crowd workers and assemble small teams for complex tasks. However, the application requires high user input effort for workflow management.

INTERACTING WITH A HYPOTHETICAL ASSISTANT
To understand how developers would use on-demand expert support, we observed how developers interacted with a "hypothetical assistant". Our use of a hypothetical assistant is similar to Shewbridge et al.'s use of a "faux 3D printer" to explore future uses of 3D printing at home [47]. It allows us to better understand the types of questions that software developers would want to ask an intelligent assistant in the "best case" scenario: if there are no limitations on its time, compensation, or capability.

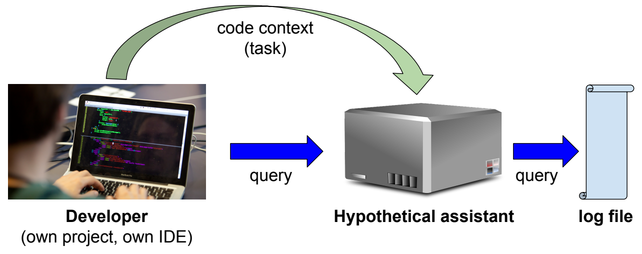

Figure 1. Setup for our hypothetical assistant study. Developer participants were asked to bring their own task to complete, and ask questions from our hypothetical assistant as if it could answer any support question needed. Participants were still allowed to use traditional online resources and augment them with the assistant as they saw fit. Context related to their programming task was collected at the beginning of each study, and audio of their questions was recorded during the session.

Study Design

We recruited five participants from the authors' university, each with different levels of programming experience. We asked participants to work on their own programming projects on their computer and IDE in a laboratory for 45 minutes. We instructed participants to imagine that they had an intelligent human assistant nearby who is capable of answering any questions that they express verbally. Participants could use their hypothetical assistant to answer questions or perform a variety of types of work. The hypothetical assistant served as a conceptual prop that participants could make requests to, while we recorded audio, as Figure 1 illustrates. After the study, we conducted an interview regarding the hypothetical intelligent assistant and looked for commonalities.

Participants' Request Categories

We filtered out non-programming questions and found that participants' requests fell into seven categories. Table 1 shows the seven categories, the number of sessions in which each question type appeared, and their overall frequency. We explain each category in more detail in this section. We will reference these categories in the design section.

Memory Aids

In *memory aid* requests, participants knew semantically what they were looking for, but sought information regarding the exact syntax required. For example,

P1: *"…I forgot how to use the syntax for glmer() function in R, I forgot like specifically how to write the random variable like to feed to the function, so can you find that out for me?"*

P5: *"What's the command to put in terminal in the command line when trying to make a file and executable?"*

Explanatory Requests

In contrast to memory aids, *explanatory requests* seek explanations in addition to specific information.

P1: *"I'm curious about what the [a] argument actual means or like does in that functions can you find that as well."*

High-Level Strategic Guidance

Unlike the previous two request categories, *high-level strategic guidance* questions ask for the best way to approach a problem. Here, participants have a high-level idea of the task they wanted to perform, but were not sure how to translate their idea into code, or if it is possible.

P1: *"I'm trying to like make a Bayesian network model based on the current existing data, and I just wonder like is there a good library or package that I can use from R…"*

P3: *"Wondering if there is a way to ensure to understand if the radio button in HTML can be color coded."*

Code Requests

Participants also asked for specific blocks or portions of code from the hypothetical assistant.

P3: *"How to make an AJAX get call from JavaScript"*

Debugging Requests

Two participants also requested help debugging their code. Although this type of request was relatively uncommon, we believe this might be due to our study's short duration. Debugging examples are:

P1: *"I'm trying to make a new variable from R by selecting like some parts of the data columns, but I kept getting an error… is there a way to like which column name has been mistype, or doesn't exist in the existing data frame?"*

P1: *"I'm trying to integrate MySQL database to R using R MySQL library I'm using the db connect function, but I keep getting an error if I use something like remote hostname other than localhost I'm not sure what's the problem. [sic]"*

P1: *"I'm using [a] function from [a] package from R, I do … I'm getting unconstrained network result from, I want to constraint the number of network by 2, but sometime it gets too much time to compute then getting out unconstrained network which is very strange for cause I thought it would cost less time than the constrained one. Can you find out why? And suggested how should I fix my code?"*

Code Refactoring

Two participants also requested help *refactoring* their code to improve its structure. Many of these code refactoring requests also had an educational component; participants wanted to refactor their code to improve their coding style.

P1: *"I'm using ROC curve to evaluate logistic regression, not sure whether that's a good idea or not, can you find other kinds of function that I can use to learn…?"*

P4: *"I'm setting up a variable method to manipulate the class I'm making in the init() function which I'm supposed to do, I'm also curious as to whether I have to make persistent variables elsewhere outside of init() method, whether I can make a new variable on self anywhere."*

P4: *"What are the best like refactoring patterns going forward that could make a non object-oriented script."*

Effort-Saving Requests

Four participants also made *effort-saving requests* to automate parts of writing and debugging code that they appeared to find tedious. Interestingly, a large number of effort-saving requests involved writing tests for code.

Description	# Sessions (out of 5)	# / Session
Memory Aids: Participants sought a specific function name	2	0.6
Explanatory Requests: Participants sought examples or explanations of their code	4	1.8
High-Level Strategic Guidance: Participants sought best ways to approach problems	5	4.6
Code Requests: Participants sought specific pieces of code	2	0.6
Bug Fixing: Participants sought specific solutions to program errors	2	1.0
Code Refactoring: Participants asked for code improvements	2	0.8
Effort-Saving Requests: Participants handed off tasks to save time and effort	4	4.0

Table 1. Common query types observed during our hypothetical assistant study, with corresponding frequencies. Each of these query types suggests a support role that remote software development assistants can play in future systems. "Number of Sessions" indicates the number of different sessions that each type of question occurred in, while "Number of Queries per Session" indicates the number of queries that referred to solving one of these query types, on average.

P5: *"Do some unit test for me to set key and get key."*

P3: *"I can test it on IE, can you please test this on Safari? Because I don't have a Mac machine"*

P4: *"Make some of the document that I'm testing, like I'm making a [a] document that's kind of a pain to make, it will be great if someone can do that."*

Findings and Interviews

Overall, each participant asked an average of 15 questions. Participants often phrased questions in a way that required knowledge of their code base. In other words, only 18% of participants' questions were "self-contained".

To better understand participants' concerns and suggestions, we conducted a follow-up interview. In this interview, we gathered their opinions on the advantages, disadvantages, and desired features of a hypothetical code assistant. Every participant indicated that timely responses would be crucial to their adoption of such an assistant. Four participants also cited personalized answers and free form questions as another potential advantage of such an assistant over current systems.

P1: *"A lot of time I like ask questions person to person. It's much easier to articulate my questions... "*

The most cited concern amongst participants was data privacy; two participants would prefer to have fine-grained control over what parts of their code they transmit when they ask a question. Three participants were also concerned about a potential lack of learning benefits compared to traditional sources of information.

P5: *"One bad thing that I think the system... goes against the very nature of computer scientists, because computer scientists need to solve those problems on their own"*

One participant, who is a non-native English speaker with a heavy accent, saw relying on voice input as a potential disadvantage. This participant found that they would prefer to type questions rather than speaking them.

Finally, we asked participants the media with which they would prefer an intelligent assistant responded. Although participants had a strict preference between text (3/5) and voice (2/5), most particpiants' chosen medium depended on the type of response. One participant preferred a combination of the two: voice feedback with bullet point reminders. Participants also differed in their preferred style of response. Two participants preferred a "teach me" style of feedback, which they would translate into steps. However, two participant preferred a "show me" style of feedback when the question is short, with directed instructions.

REMOTE PAIR PROGRAMMING SUPPORT

We ran a second study to explore how developers try to onboard remote experts on demand. Unlike the hypothetical assistant study, our study focused on communication challenges, such as on-boarding challenges, contextual needs, and interaction difficulties.

Study Design

Whereas the first study referenced a hypothetical intelligent assistant (which we think of as a "best-case" assistant), our study used human assistants. We paired participants so that every session had one "requester" and one "helper". We recruited 24 participants for 12 sessions, and gave requesters Web development tasks that involved displaying JSON data from a URL. Helpers were asked to respond to any queries from requesters (through voice and text) during the study. Figure 2 illustrates our study setup.

We determined whether each participant would be a helper or requester based on their Web programming experience. To measure this, we gave participants a pre-task survey of their JavaScript and HTML skills. We assigned participants who scored six or above (out of nine points) to helper roles and those who scored five or below to requester roles. This simulates the case where a future system is able to recruit the "right" helper for a job. Our experiments are intended to explore the interaction between helpers and requesters after the recruiting has been completed.

All 12 requester participants were students at the authors' university, as were half (six) of the helpers. We recruited the other six helpers from Upwork [14], a popular freelancing platform. We specifically recruited Upwork participants whose profiles indicated that they were professional programmers. We used the same survey to ensure they were equally qualified to be helpers as our local participants.

During each session, requesters and helpers were physically separated, and requesters can ask for support from helpers via Skype, which connected them through the session. We asked requesters to share their screen in order to provide enough context to the helpers. We collected audio recordings during the study to capture the conversations that the pairs had, and the responses to the follow-up questions we asked them.

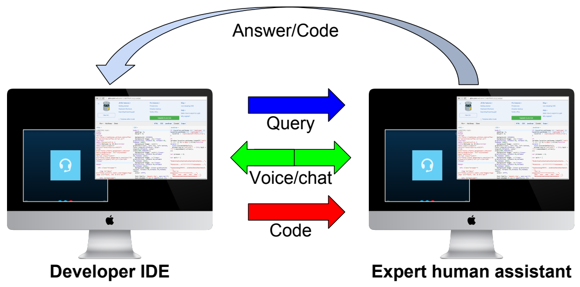

Figure 2. Setup for our human expert assistant study. In each trial, one "requester" participant (developer) was paired with one "helper" (expert, based on a pre-study skill assessment). Participants could chat via either text or voice, and requesters were able to share their screen with helpers as needed using Skype. The helper was tasked with assisting the requester *reactively*, meaning that they only responded to queries, and did not proactively propose solutions or approaches. This simulates a "best case" (repeated, non-multiplexed helper) on-demand model where human experts are not expected to be continuously available between end-user queries.

Unlike most pair programming paradigms, we specifically asked (and reminded them as needed) the helpers to be strictly *reactive*. This is in contrast to many pair programming paradigms where experts proactively monitor participants' code. Instead, helpers would only look at requesters' code when the requesters explicitly asked a question. This allowed us to better simulate the environment that intelligent code assistants would operate in: responding only to explicit requests for help and answering questions without having the complete context in which they were asked. This also allows for increased flexibility in the types of experts or crowds that can support such a system in the future.

We told requesters that they could also search for help using any standard Web resources (e.g., Google, Stack Overflow, or library documentation sites). By letting requesters use third-party resources, we gain a better understanding of the types of questions they prefer to answer through web resources versus where consulting a helper was preferred. After the study, we asked participants a few questions to get feedback on their experience with respect to the setting and helper, with specific examples of issues that arose during the study.

After analyzing the data, we generated a list of features related to participants' concerns during the study, and then recontacted all of the 24 previous participants. We received 11 replies (5 helpers, 6 requesters). We asked for their feedback on how effective they think the suggestions would be, given their experience during the study.

Findings

To understand the experience that participants had during the study, we transcribed and analyzed all of the conversations and interviews from our audio recordings. Following the thematic analysis method, we first read through each turn of the requester-helper conversations, and the interview responses. This resulted in the identification of 6 distinct user information needs (see Table 2). We also counted the number of sessions containing at least one occurrence of each information need, and the number of conversations that each need occurred in among the 12 pairs.

We defined a conversation to be a single, complete request-response interaction between requesters and helpers. If turns of a conversation were about the same initial requester query, we counted them as a single conversation. If a requester asks a new question, it begins a new conversation if and only if there was new information elicited by the query itself. For example, if a requester asks a question, the helper replies with "Say that again?", and the requester repeats their question, then this second query does not elicit new information, and thus would still be part of the same conversation. The same is true if the situation is reversed and the requester says "What?" to the helper's response. However, if a helper first provides a response, and the requester replies by clarifying their query further, then we count the two questions as belonging to separate conversations. This rule separates interactions into focused pieces, and avoids imbalance due to participants' varying styles of interaction.

Below, we provide evidence to support each of the claims in this section. In addition, we derived system feature implications that address the participant needs that we observed.

Experience

Seven participants, including both requesters and helpers, explicitly mentioned they would like to provide or have some kind of assessment of the requesters' coding background in order to receive or provide help more efficiently. For example, if helpers knew that the requester is a jQuery beginner, then the helper could avoid simply telling them to write an AJAX call to make a request.

H1: *"If I knew where she was coming from, it made it more efficient, because I won't have to ask do you know what this is, how to make this thing, or I could just say..."*

H1: *"...do you know console.log?"* R1: *"Yes"*

Context

Even though we designed the study such that the helpers do not know what the requester's task is beforehand, requesters often rephrased part of the task to the helpers to provide context. We observed that helpers often helped refine the requester's original questions and tried to get more context about the questions. Also, since we asked helpers not to be proactive and only use screen sharing when they need more information, they did not observe all the changes that the requesters made since the last time they saw the requesters' shared screen. For example,

H1: *"She had it working correctly in terms of event title coming out on the web in the output, then later on when she was coding, and she was doing some quick copy and paste and stuff like that and some quick kinda changes and tiny bit of JavaScript, and I think she resize the, like different sizes in JSBin, or whatever it's called, so I couldn't see all JavaScript anymore, she'd changed something ..., I couldn't see what she did...I couldn't see what she did later, it's literally not like visible to my screen"*

R11: *"How can I make to fetch information line?"* H11: *"so what exactly are you requesting...what kind of information are you fetching?"* R11: *"it's kind of like JSON file, I need to*

Patterns	Description	# Sessions (out of 12)	# / Session	# Interviews (out of 12)	# / Interview
Background	Helpers wanted to know the requester's experience level and background	11	1.5	7	0.7
Context	Helpers wanted to know what the high-level goals and context were	11	1.5	9	0.8
Sharing	Participants wanted a shared editor that lets helpers type code directly	11	2.1	11	1.1
Real-Time Response	Participants needed immediate responses (some preferred voice, others text)	12	N/A	10	0.9
Integrated System	Participants did not like switching windows, and would prefer a single system	9	N/A	7	0.8
Personalized Help	Requesters wanted help suited to their intent, e.g., specifying "teach me" when more explanation was desired	9	1.0	10	1.2

Table 2. Common participant information needs that we observed during our human expert assistant study, with corresponding frequencies. Each of these needs would limit the success of a naive approach to providing remote assistance for software developers. "Number of Sessions" indicates the number of different trial sessions that each information need occurred in, while "Number of Conversations per Session" indicates the average number of times each need arose in a conversation (stemming from requester queries). "Number of Interviews" indicates the number of unique interviews in which the need was mentioned at least once. "Number of Mentions per Interview" is the average number of times that a participant mentioned the need in the post-trial interviews.

extract several information from the JSON file." H11: *"Right, do you have an URL or something?"* R11: *"Yeah"*

H5: *"If I could know what the problem is, the problem statement that he was solving for, so I probably would be able to help better."*

Sharing

During the study, we asked requesters to share their screen. Both requesters and helpers mentioned that it would save time if helpers could type in the requesters' editor or point to which line of code they were referring to. We observed that helpers often used the shared screen to tell requesters where to look and what to type. However, requesters were sometimes unable to follow helpers' suggestions. This process took longer than expected, potentially because of the ambiguity of the helpers' instructions (e.g., when multiple lines use the same terms mentioned), or the requesters were not familiar with the programming languages (e.g., what/where a callback function is), or communication issues (e.g., English proficiency, or Skype signal quality). For example,

H1: *"…just bring the cursor to the left into the after the curly brace…"* R1: *"after the curly brace, here?"* H1: *"yeah, and then put a comma, and put a string call JSON, and quotes"* R1: *"(typing)"* H1: *"no no the quotes …"*

R3: *"The code she sent can run on her side, but I'm not able to run it on my side. So if she could share it in some way, like do a comparison of the output, that will be great."*

Real-time Response

We provided voice, text, and sharing of the requester's screen as communication channels during the study. Each of these channels delivers different information. Pairs mainly spoke to each other, even when helpers were instructing requesters what to type. In the post-task interview, two participants mentioned the real-time response is good for technical support based on their prior experience (H1, R3). The participants' screen recordings also showed that when helpers were waiting for a request, they complete non-support tasks, such as coding their own project, reading news, etc.

However, more than half of participants were non-native English speakers, which led to voice communication issues. Thus, many requesters asked helpers to communicate through text. Furthermore, some requesters felt unconformable with voice and hesitated to ask questions as a result.

H6: *"In general, I won't write for the requester, but more like tell them [sic]…"*

Integrated System

Participants used multiple systems during the study: Skype, code editor, browser, etc. We observed that participants were often annoyed when switching their focus back and forth between windows. For example, if a helper sent a code snippet to a requester via Skype, the requester would have to go to Skype, and copy/paste the code to their editor. On the other hand, when the helpers wanted to run the requesters' code on their own machine, they had to copy the code from Skype and paste back into their editor. Beyond these issues, participants also had problems with connections and platform switching.

R8: *"The hangout got disconnected. I was talking and developing, and no clue when it was not working. Also, I have to go back the window and check the text helper sent."*

H3: *"Like every time I have to come back and check [Skype], it's very tiny, maybe it's because I shrink it, and then I have to open it up… if we could use JSBin, and then there is a chat or voice on the side bar, that will be faster, so it's like I type and he can see and can talk…"*

Personalized Help

Often, we observed that requesters had to iterate on their original questions, even if the helper provided a related answer. Both requesters and helpers mentioned that they would like to provide or know the level of responses they want to receive from the helper or they should provide to requesters. This could be a high level answer, such as goal, or a detailed answer, like a function name. For example,

H1: *"Do you want to try to type something or you want to me to explain what to do?"* R1: *"Please explain what to do."*

R6: *"It would've been helpful if he could intervene to see if I'm doing something wrong ... that I couldn't figure out."*

SYSTEM DESIGN SUGGESTIONS

After analyzing the data from the hypothetical assistant study and the remote pair programming study, we found that both requesters and helpers expressed their concerns regarding our research question of how to provide or receive help more efficiently. We drew system design implications from these concerns and suggested six features to both helpers and requesters. We contacted our prior participants (11 of 24 replied) and interviewed them about the six feature suggestions we developed based on their experiences and concerns. We used prototypes to explain three of these features. We discuss the interviews and their design implications below.

Helper Page

To make helpers efficiently find the questions they are capable of answering, we suggested creating a webpage that lists requesters' unanswered questions. Here, we discuss two features related to this page.

Question List

We prototyped this question list on a webpage that contains all the unanswered requests that the requesters made, code in the working file, requesters' highlighted code, a Cloud9 [24] link containing all the files in the repository that the requesters work on, and the requesters' Skype usernames. All of prior helpers claimed that this information is enough for them to determine if they are capable of answering a question. For some helpers, a well-written question in natural language was sufficient (H6, H10).

H4: *"If it's just whether you are capable of answering the question, I mean I would say like just have the question as specific as possible, with natural language."*

Background Assessment

With the concerns about the requesters' programming experience, we suggest adding an assessment of requesters' programming abilities. We came up with four types of assessments and let the helpers choose their favorite: 1) the number of lines of code they have written in the past, 2) the frequency of their coding experiences, 3) the number of years they have been programming, and 4) a self-rating of Beginner/Intermediate/Expert.

Three out of five helpers chose assessment 2 (coding frequency) because they thought the other options could not be truly associated with the requesters' actual skill levels. In addition, they mentioned they would probably need more than the frequency: it would be ideal if they could also have the summary of code they have previously written. The other two helpers chose assessment 4 because they thought the other options could be biased. We also asked them whether they want the assessment result to be given in comparison to their own skill level or as an absolute measure. All preferred an absolute measure.

H6: *"I choose [assessment] 2. Because the total amount of code he wrote is a vague concept compared to how much his wrote in the most recent days and weeks."*

H10: *"[I choose assessment] 4. 1 is hard to tell. You can code a lot and don't understand anything. 2 is not a thing you can measure, someone's skill. 3 is not a measure."*

Suggested Support Features

On the requester side, we found that six participants directly mentioned the unfamiliarity they had with the JSBin editor. They would prefer to code in their own editor. Therefore, we suggested to develop a package on a widely used editor. Below, we discuss four suggested features that are needed.

Code-Commenting Reminder

We found that helpers often want to know the broader context of requesters' questions (their project and task goals), even when it was irrelevant to the request. Conversely, requesters also wanted helpers to be aware of their high-level task and plan. To address this challenge, we prototyped a feature in an editor that would automatically generate a descriptive comment reminder (e.g. *Please describe what this method does*) above important methods, such as API calls, which could make the code more contextual and readable. All six interviewed requesters found this to be a useful feature that reminds them to comment their code and aids helpers' understanding of their code.

R7: *"It would be a total nagging thing, so like you are risking annoying people, but that's exactly the thing that you need to nag, that you need to keep up with the commenting."*

Two participants suggested that this feature would be more useful if they could better understand how these methods were being used in the code and where they were called. This would allow helpers to better understand requesters' intentions. We also suggest making this feature easy to control; three participants mentioned that they would prefer to be able to quickly enable and disable this feature.

R12: *"[Often,] I forget to write my comments, it would be great to be reminded, it would save me a lot of time later. There is a small drawback when I don't have the time to do it, I don't want something to keep bugging me and telling me,...But if you can dismiss it, or if you can turn it off, it won't be annoying."*

Contextualized Explanations

One setting in which requesters *do* prefer text responses is formatted code snippets. Helpers often want to include explanations with proposed changes to requesters' projects (either code or comments). Also, requesters may forget where and what they previously asked if the response time is long, thus, ideal responses would not only provide them the answers, but also where and what they asked.

H6: *"If I can directly see what he is typing, not just his screen. The screen resolution is not high, and I wish we could have something like Google Docs."*

To overcome this, we propose adding a contextualized explanation feature that checks all changes to the project code made by a helper, and displays the proposed edit along with an explanation generated by helpers. This may be similar in style to a modern work processor comment (e.g., in Google Docs, or Microsoft Word), which appears in a bubble to the side of the main text, with a visible anchor linking the explanation and the code. This not only helps requesters remain knowledgeable about the specifics of their code base, even when being supported by remote experts, but it also provides a chance to learn from the edits made, if desired. Three participants directly mentioned they would prefer to have such feature versus having the helpers type directly for them.

Text and Voice

Participants could both type and speak to each other during the study, and they had different preferences in terms of asking or answering questions. Four out of six requesters preferred using text to communicate to make exchanging sample code easier. From the hypothetical assistant study, three out of five participants preferred text as well, and the rest mentioned it depends on the questions. We prototyped the feature that the requesters could ask a question in voice in the editor, and we suggest that the system should also allow participants to ask questions in text as well.

R11: *"I like a mix of both, but if only one, I prefer text because [a helper] can send the example… text will be very slow if I want to explain what I'm doing, voice can do this interaction very quick."*

R7: *"[My preference is] probably voice, I found in general that whenever I ran into a problem, as soon as I can articulate the problem fully, I can find the answer to the problem"*

Teach/Show Me Button

Ten participants expressed their wish that the helper could provide more personal help. They may want helpers to educate them such that they can fill in their knowledge gaps. Or they only want to speed their productivity up by simply completing the tasks. To be effective, this feature should be combined with a background assessment tool that allows helpers to better gauge requesters' backgrounds. We suggest adding an option for requesters to indicate their expected response: if they want the helper to teach them about the question, or if they just want the helpers to show them how to do a given task. One requester mentioned that she usually wants to integrate the example into her code so she prefers to understand the code first (R8).

R2: *"Also, if he can do this thing together, instead of he answers my general question, I prefer more detail question, I prefer he asked me something."*

FUTURE WORK

In future work, we aim to explore each of these features by creating a functional system and deploying it with real users. Further work is needed to iterate on the design of our approaches to addressing the user needs we found. This work also informs the design of collaborative tools intended to help teams of developers work more effectively.

Supporting Repeated Interactions

Over time, helpers and experts might build up trust relationships. Developers might prefer to route their questions to experts with whom they have a prior relationship when possible. They may also want to give these experts additional permissions to edit or view code. We are exploring ways to support such ongoing relationships, even given dynamic workforces.

Note that we explicitly limited the type of support to *reactive* support — a developer must ask a question before receiving support. We are also considering ways to make help efficiently *proactive* — allowing experts to give users suggestions even before they explicitly ask a question. This way, if an expert sees a potential issue a developer will face, they can proactively suggest how they can best accomplish their task.

Privacy Considerations and Crowds

A key remaining challenge will be to preserve the privacy of both the end users and their projects. Prior work has explored potential ways to preserve privacy in images and other media [33, 30], but not with complex, context-dependent tasks like programming. An interesting open question is how expert programming sub-tasks can be coordinated between multiple helpers to preserve private information about the project and approaches used, in a manner that still allows the task to be completed successfully. The same approaches would allow multiple developers to support a single end-user developer quicker and more accurately than any single helper could.

Motivation and Compensation

Another interesting open question is how to best motivate and compensate experts for their help. The design space for on-demand support is larger than any particular system, and no one mechanism for compensation is inherent to the idea of on-demand support. On the other hand, many CQA websites rely on a reputation-based system to motivate experts to answer questions [41]. Future work will study different cost structures for on-demand support through real deployments instead of in-lab studies, which may change the way that participants view costs.

CONCLUSION

In this paper, we have informed a broad space of support systems and presented a first glimpse into the challenges and opportunities of on-demand expert support for software development tasks. Our studies provide insight into how and when developers choose to reach out for assistance, and the challenges of supporting developer needs via a remote expert. We have presented a set of features to address the primary needs that appeared in our study. We then validated the proposed utility of these features in a follow-up design iteration with participants from our trials. In summary, this work informs, and brings into focus, a previously under-studied paradigm of expert developer support.

ACKNOWLEDGEMENTS

We would like to thank all of our study participants and our reviewers for their assistance.

REFERENCES

1. Mark S Ackerman. 1998. Augmenting organizational memory: a field study of answer garden. *ACM Transactions on Information Systems (TOIS)* 16, 3 (1998), 203–224.
2. Mark S Ackerman and David W McDonald. 1996. Answer Garden 2: merging organizational memory with collaborative help. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*. ACM, 97–105.
3. Thomas J Allen. 1984. Managing the flow of technology: Technology transfer and the dissemination of technological information within the R&D organization. *MIT Press Books* 1 (1984).
4. Muhammad Asaduzzaman, Ahmed Shah Mashiyat, Chanchal K Roy, and Kevin A Schneider. 2013. Answering questions about unanswered questions of stack overflow. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 97–100.
5. Prashant Baheti, Edward Gehringer, and David Stotts. 2002. Exploring the efficacy of distributed pair programming. In *Extreme Programming and Agile MethodsXP/Agile Universe 2002*. Springer, 208–220.
6. Michael S Bernstein, Joel Brandt, Robert C Miller, and David R Karger. 2011. Crowds in two seconds: Enabling realtime crowd-powered interfaces. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 33–42.
7. Jeffrey P Bigham, Chandrika Jayant, Hanjie Ji, Greg Little, Andrew Miller, Robert C Miller, Robin Miller, Aubrey Tatarowicz, Brandyn White, Samual White, and others. 2010. VizWiz: nearly real-time answers to visual questions. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*. ACM, 333–342.
8. Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 513–522.
9. Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009a. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1589–1598.
10. Joel Brandt, Philip J Guo, Joel Lewenstein, Scott R Klemmer, and Mira Dontcheva. 2009b. Writing Code to Prototype, Ideate, and Discover. *Software, IEEE* 26, 5 (2009), 18–24.
11. Frederick P Brooks. 1975. *The mythical man-month*. Vol. 1995. Addison-Wesley Reading, MA.
12. Alistair Cockburn and Laurie Williams. 2000. The costs and benefits of pair programming. *Extreme programming examined* (2000), 223–247.
13. Philip R Cohen. 1992. The role of natural language in a multimodal interface. In *Proceedings of the 5th annual ACM symposium on User interface software and technology*. ACM, 143–149.
14. Upwork Inc. (formerly oDesk). 2015. Upwork. (2015). `https://www.upwork.com` Accessed: September, 2015.
15. Max Goldman, Greg Little, and Robert C Miller. 2011. Real-time collaborative coding in a web IDE. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 155–164.
16. Philip J Guo. 2015. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. In *Proceedings of the 28th annual ACM symposium on User interface software and technology*. ACM.
17. Philip J Guo, Jeffery White, and Renan Zanelatto. 2015. Codechella: Multi-User Program Visualizations for Real-Time Tutoring and Collaborative Learning. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE.
18. Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. 2015. Supporting Developers' Coordination in the IDE. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. ACM, 518–532.
19. Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1019–1028.
20. James D Herbsleb, Helen Klein, Gary M Olson, Hans Brunner, Judith S Olson, and Joe Harding. 1995. Object-oriented analysis and design in software project teams. *Human–Computer Interaction* 10, 2-3 (1995), 249–292.
21. Damon Horowitz and Sepandar D Kamvar. 2010. The anatomy of a large-scale social search engine. In *Proceedings of the 19th international conference on World wide web*. ACM, 431–440.
22. Codenvy Inc. 2012. Codenvy. (2012). `https://codenvy.com` Accessed: September, 2015.
23. Codementor Inc. 2014. Code Mentor. (2014). `https://codementor.io/` Accessed: September, 2015.
24. Cloud9 IDE Inc. 2010. Cloud9 IDE. (2010). `https://c9.io` Accessed: September, 2015.
25. Pluralsight Inc. 2013. hack.hands(). (2013). `https://hackhands.com/` Accessed: September, 2015.
26. Andrew J Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development

teams. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 344–353.

27. Andrew J Ko, Brad Myers, Htet Htet Aung, and others. 2004. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 199–206.

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

29. Walter S Lasecki, Juho Kim, Nick Rafter, Onkur Sen, Jeffrey P Bigham, and Michael S Bernstein. 2015. Apparition: Crowdsourced User Interfaces that Come to Life as You Sketch Them. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 1925–1934.

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

31. Walter S Lasecki, Rachel Wesley, Jeffrey Nichols, Anand Kulkarni, James F Allen, and Jeffrey P Bigham. 2013. Chorus: a crowd-powered conversational assistant. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 151–162.

32. Thomas D LaToza, W Ben Towne, Christian M Adriano, and André van der Hoek. 2014. Microtask programming: Building software with a crowd. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, 43–54.

33. Greg Little and Yu an Sun. 2011. Human OCR: Insights from a complex human computation process. In *Workshop on Crowdsourcing and Human Computation, Services, Studies and Platforms, ACM CHI*.

34. Greg Little, Lydia B Chilton, Max Goldman, and Robert C Miller. 2010. Turkit: human computation algorithms on mechanical turk. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*. ACM, 57–66.

35. Rajan M Lukose, Eytan Adar, Joshua R Tyler, and Caesar Sengupta. 2003. Shock: communicating with computational messages and automatic private profiles. In *Proceedings of the 12th international conference on World Wide Web*. ACM, 291–300.

36. Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. 2011. Design lessons from the fastest q&a site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2857–2866.

37. David W McDonald and Mark S Ackerman. 2000. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. ACM, 231–240.

38. Harlan D Mills. 1980. Software engineering education. *Proc. IEEE* 68, 9 (1980), 1158–1162.

39. Audris Mockus and James D Herbsleb. 2002. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th international conference on software engineering*. ACM, 503–512.

40. Gary M. Olson and Judith S. Olson. 2000. Distance Matters. *Human-computer interaction* 15, 2 (2000), 139–178.

41. Stack Overflow. 2015. Stack Overflow. (2015). `https://stackoverflow.com/` Accessed: September, 2015.

42. Maria Soledad Pera and Yiu-Kai Ng. 2011. A community question-answering refinement system. In *Proceedings of the 22nd ACM conference on Hypertext and hypermedia*. ACM, 251–260.

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

44. Fatemeh Riahi, Zainab Zolaktaf, Mahdi Shafiei, and Evangelos Milios. 2012. Finding expert users in community question answering. In *Proceedings of the 21st international conference companion on World Wide Web*. ACM, 791–798.

45. Martin P Robillard, Robert J Walker, and Thomas Zimmermann. 2010. Recommendation systems for software engineering. *Software, IEEE* 27, 4 (2010), 80–86.

46. Julia Schenk, Lutz Prechelt, and Stephan Salinger. 2014. Distributed-Pair Programming can work well and is not just Distributed Pair-Programming. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 74–83.

47. Rita Shewbridge, Amy Hurst, and Shaun K Kane. 2014. Everyday making: identifying future uses for 3D printing in the home. In *Proceedings of the 2014 conference on Designing interactive systems*. ACM, 815–824.

48. Jonathan Sillito, Gail C Murphy, and Kris De Volder. 2008. Asking and answering questions during a programming change task. *Software Engineering, IEEE Transactions on* 34, 4 (2008), 434–451.

49. Devrim Yasar Sinan Yasar. 2012. Koding. (2012). `https://koding.com` Accessed: September, 2015.

50. Igor Steinmacher, Marco Aurélio Graciotto Silva, and Marco Aurélio Gerosa. 2014. Barriers faced by newcomers to open source projects: a systematic review. In *Open Source Software: Mobile Open Source Technologies*. Springer, 153–163.

51. Laurie Williams and Robert Kessler. 2002. *Pair programming illuminated*. Addison-Wesley Longman Publishing Co., Inc.