Microtask Programming

Thomas D. LaToza, Arturo Di Lecce, Fabio Ricci, W. Ben Towne, *Member, IEEE*, and André van der Hoek, *Member, IEEE*

Abstract—Traditional forms of crowdsourcing such as open source software development harness crowd contributions to democratize the creation of software. However, potential contributors must first overcome joining barriers forcing casually committed contributors to spend days or weeks onboarding and thereby reducing participation. To more effectively harness potential contributions from the crowd, we propose a method for programming in which work occurs entirely through microtasks, offering contributors short, self-contained tasks such as implementing part of a function or updating a call site invoking a function to match a change made to the function. In microtask programming, microtasks involve changes to a single artifact, are automatically generated as necessary by the system, and nurture quality through iteration. A study examining the feasibility of microtask programming to create small programs found that developers were able to complete 1008 microtasks, onboard and submit their first microtask in less than 15 minutes, complete all types of microtasks in less than 5 minutes on average, and create 490 lines of code and 149 unit tests. The results demonstrate the potential feasibility as well as revealing a number of important challenges to address to successfully scale microtask programming to larger and more complex programs.

Index Terms—D.2.6: Programming Environments; D.2.9: Management

1 Introduction

Crowdsourcing software engineering offers a number of opportunities for reducing time to market, generating alternative solutions, employing specialists, learning through work, and democratizing participation in software engineering [1][2][3]. One of the oldest and most prevalent forms of software crowdsourcing today is open source software development (OSS). In OSS, contributions are solicited from the crowd, opening the contribution of features and bug fixes to hobbyists, professionals, and even companies [4]. OSS is a form of commons-based peer production envisioned to achieve three key structural attributes: (1) it is possible to decompose output into separate contribution units (e.g., addressing issues in an issue tracker), (2) contributions are sufficiently fine-grained to capture contributions from those whose motivation will only sustain small efforts, and (3) a low-cost mechanism defends against incompetent and malicious contributions and integrates them into a whole (e.g., pull requests) [5]. However, open source projects today do not fully realize the potential of this model, as they impose significant *joining barriers* on potential contributors. These include: (1) identifying appropriate contacts and receiving timely feedback, (2) identifying appropriate tasks and corresponding artifacts, (3) understanding project structure, complex code, and setting up a workspace, (4) outdated, unclear documentation and information overload, and (5) learning project practices, domain knowledge, and technical expertise [6]. Taken together, these barriers impose a lengthy *joining script* [7], dissuading the busy or casually committed from contributing and limiting the pool of millions of potential contributors to only the most committed.

One solution to this problem commonly used in other domains is the *microtask*. A microtask is a short, self-contained unit of work with a clear objective. Organizing work in microtasks decontextualizes the tasks done by workers, enabling a contribution to be made in isolation of other ongoing work and with no requirements for prior knowledge. Decontextualizing work into microtasks has been applied to problems in a number of domains. For example, in Soylent work copy editing a document is decontextualized and decomposed into "find", "fix", and "verify" microtasks in which separate workers to identify opportunities for revisions, generate potential revisions, and vote on revisions [8]. Players of the game FoldIt solve puzzle tasks, the results of which are then used to compute solutions to challenging protein folding tasks [9].

Microtask crowdsourcing has found application to software development for tasks where decontextualization is possible. Developers today routinely complete microtasks on StackOverflow, where the asker of the question is responsible for decontextualization by identifying a short, clear objective in the form of a question, often structured as a short snippet of code to be generated or revised [10]. Sites such as uTest[1] match freelance testers to projects, helping projects rapidly recruit from the over 300,000 registered testers.

But can programming be microtasked? Is it possible for a developer that today might make a ten-minute contribution answering a question on StackOverflow to instead spend the same ten minutes writing code for an open source project? Achieving this goal requires new methods

- *Thomas D. LaToza is with the Department of Computer Science, George Mason University, 4400 University Drive, MS 4A5, Fairfax, VA 22030. E-mail: tlatoza@gmu.edu.*
- *Arturo di Lecce is with Cuebiq Srl, Italy. E-mail: arturodilecce@gmail.com.*
- *Fabio Ricci is with Bosch Rexroth. E-mail: f.ricci89@gmail.com.*
- *W. Ben Towne is with the Institute for Software Research, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213. Email: wbt@cs.cmu.edu.*
- *André van der Hoek is with the Department of Informatics, Donald Bren School of Information and Computer Science, University of California, Irvine, 5029 Donald Bren Hall, Irvine, CA 92697-3440. Email: andre@ics.uci.edu.*

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

to reduce the context demands and joining barriers traditionally imposed by open source projects to broaden participation [2]. However, broadening participation is a distinct goal from, and may even be in opposition to, increasing team productivity. Compared to a single developer, a team of five incurs coordination overhead unnecessary for a single developer. Similarly, increasing the number of contributors to an open source project from 20 to 2000 may well mean that the number of person hours per contribution increases, as the contributors have spent less time on the project, have less expertise, and work more slowly. Rather than increase team productivity, the goal is to harness contributions from the 1980 transient contributors whose potential to contribute would otherwise go unused, as Shirky argues [11].

To address the challenge of microtasking programming, we propose a set of design principles for new programming environments and describe the design of a system following these principles. To enable fine-grained contributions, work is organized into local changes to a single artifact such as a function or test. The system itself is then responsible for creating, managing, and assigning microtasks, tracking the state of artifacts to determine what needs to be done and propagating interface changes between artifacts when necessary. To ensure the quality of the final program created, artifacts are iteratively assessed and revised through sketching, repeated edits, reporting issues, a review process, and testing. To contribute, workers simply login, complete a short tutorial, and begin work on an assigned microtask.

Our work is part of a longer-term exploration of the use of microtasks in software engineering. In this paper, we consider programming in the small, where a well-defined request for a component posed by a client is implemented through coding, testing, and debugging. Our approach builds on early explorations and prototypes examining mechanisms for deriving microtasks for programming and small pilot experiments [12][13]. Other work has examined the relationship of microtask crowdsourcing to other crowdsourcing models [1] and considered mechanisms for coordinating developers working on programming microtasks [14]. In this work, we do not consider microtasking the creation of software requirements, architecture, design, or user interfaces. In other work, we have begun to explore how software design tasks might be crowdsourced [15][16].

In this paper, we contribute a new set of design principles for microtask programming reflecting the lessons learned from our earlier approaches [12][13]. In particular, we contribute new techniques for ensuring quality, providing feedback to contributors, debugging modularly, enabling contributors to more easily reuse code, and on-boarding new contributors more quickly. We also contribute a new empirical study exploring the potential and challenges of microtask programming, examining specifically the feasibility, speed of onboarding, speed of contributions, and effects of quality control mechanisms.

We first illustrate microtask programming through an example. We then present our approach and describe the design of a new microtask programming environment. To evaluate the possible feasibility of microtask programming, we describe results from a user study where two small crowds used microtask programming to build small programs. We conclude by discussing the prospects and challenges for the application of microtasking to programming.

2 Motivating Example

Doug just started an open source project to build a better drawing application. After putting together an initial vision of the features it should have, he decides the next step is to start building a prototype implementation. He decides to microtask the creation of this prototype using the CrowdCode microtask programming environment. He first specifies the core logic of the drawing program as a component, consisting of four functions and the abstract data types (ADTs) describing the data that these functions will consume and produce. Doug enters this information in the CrowdCode admin interface, gives the project a name, posts a link to the project on his website, and announces his new drawing application on an online programming forum.

Alice finds the project, clicks through, and logs into CrowdCode. She first sees a short tutorial, describing the major elements of the CrowdCode interface including the task description, microtask pane, chat, activity feed, and leaderboard. She is then assigned a microtask to *Edit a Function*. She reads a second tutorial describing the microtask, identifying its goal, the possible actions she may take, emphasizing that she should only write a few lines of code or pseudocode, and that she need not complete the function's implementation. She then begins work on the *Edit a Function* microtask, reading the function `moveElement` contained within. Discovering it already contains pseudocode, she replaces some of it with a partial implementation. She decides that implementing functionality to checking preconditions is behavior best implemented elsewhere, so she writes a function declaration stub for a new function `validElementType` at the bottom that checks preconditions and adds a call in `moveElement` (Fig. 1, bottom). Working to finish up, she notices that the ten-minute count down timer has nearly expired. Writing down one final thought, she adds her own pseudocode describing a part of the function that still needs to be completed and clicks submit.

As Alice works, others simultaneously work on seven separate microtasks, building code and tests for the remainder of the drawing program's API. Bob is assigned a microtask to test `moveElement`. The microtask provides the description and signature of the function, text describing a test case to move a rectangle, and a request to implement the test case as a unit test. Using the test builder interface, Bob first browses a panel listing the available data types to understand the fields of the Element data type. He then realizes he does not know what coordinate system is being used. Not immediately seeing an answer, Bob asks a question in the chat, to which Alice replies. Bob finishes the test and clicks submit.

Charles is riding the bus home and really wants to just

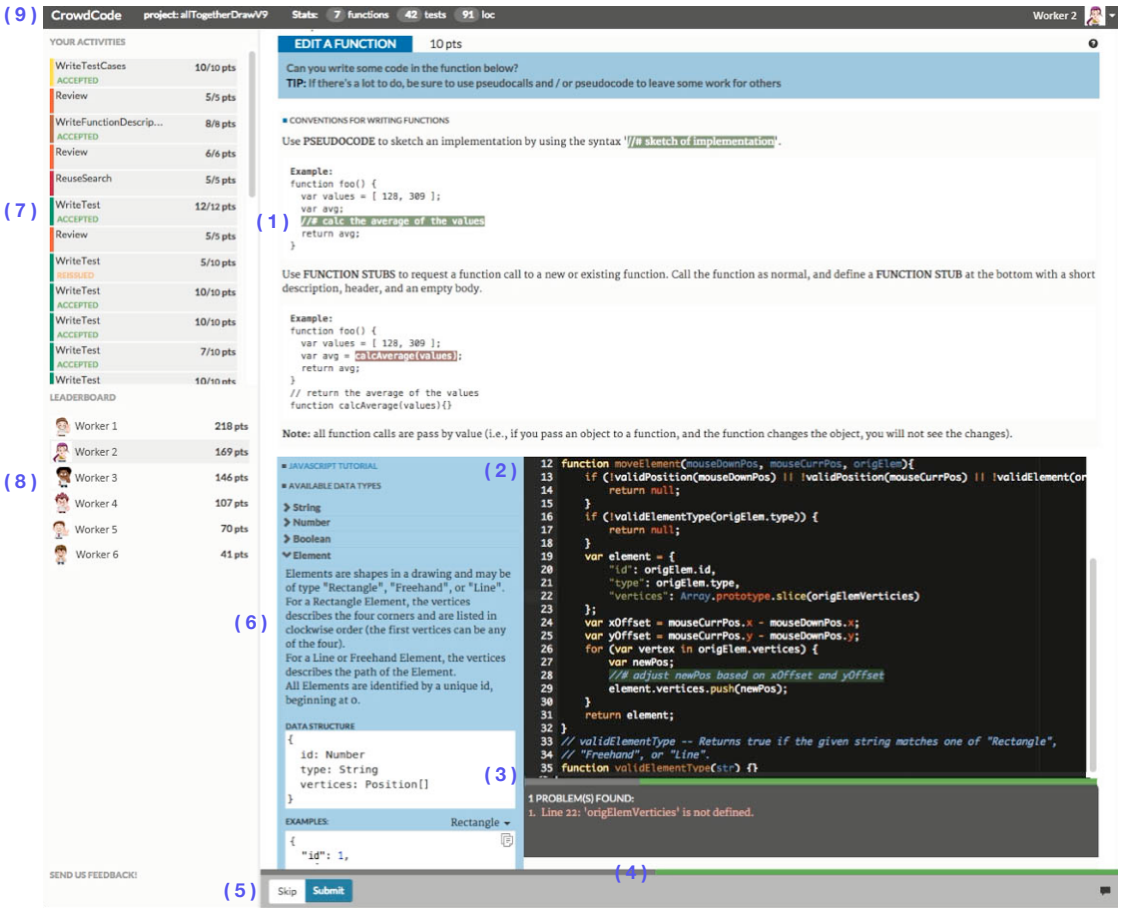


Fig. 1. The CrowdCode microtask programming environment includes: (1) the current microtask, (2) a code editor, (3) the number of remaining statements that can be added, (4) the time remaining, (5) controls to submit or skip the microtask, (6) the ADT browser, (7) the activity feed listing the worker's submitted microtasks, (8) a leaderboard, and (9) current project-wide statistics.

program. He logs in to CrowdCode and is assigned a *Write Test* microtask. Unexcited, he clicks skip and is assigned a new *Edit a Function* microtask, continuing where Alice left off on `moveElement`. More enthused, he quickly implements the algorithm Alice sketched and edits a few other lines to make the code a little cleaner. He then clicks submit.

Dave reviews Charles' work. He notices a subtle issue in the algorithm. In his review, he describes the issue and rates the work as 3 stars. Ellen is assigned an *Edit a Function* microtask to address the issue, which she fixes. Assigned to review Ellen's contribution, Alice is pleasantly surprised to see how work on `moveElement` has progressed since her earlier contribution and gives Ellen's work 5 stars.

As CrowdCode tracks the state of `moveElement`, it determines that it has no remaining pseudocode to be completed and several completed tests. After executing each test using the distributed test runner, it receives a report of a test failure and creates a *Debug Task Failure* microtask. Charles is assigned the microtask to debug `moveElement`, selects the failing test, and hovers over expressions to see their value in the execution. Hovering his mouse over a call from `moveElement` to `validElementType`, he finds that `validElementType` returns false, contradicting the behavior it promises in its description. Using the stub editor, he edits the return value of `validElementType`, replacing it with true. Rerunning the tests, he sees that they now pass and clicks submit. This automatically generates a new test for `validElementType` which is automatically executed and fails.

Frank logs in to CrowdCode and is assigned a *Debug Test Failure* for `validElementType`. Looking at the failing test, he sees that it is returning false for a shape that should be valid. Editing the code, he sees that the function is missing a parameter required to check the validity of the shape. After editing the signature to add the parameter, he adds some functionality, reruns the test, sees that it passes, and submits. Microtasks are then automatically generated to update each call site and test of `validElementType`.

3 Related Work

Microtask programming draws on ideas from several areas: open source software development, applying microtask crowdsourcing to complex work, and crowdsourcing software development tasks.

One form of crowdsourcing is open source software development, where source code is freely shared and anyone may contribute [4]. Traditional methods for contributing to open source projects differ from microtask crowdsourcing in several important respects, particularly in that tasks are at the granularity of implementing a feature or fixing a defect with a duration of hours or days rather than minutes. Workers must also complete a lengthy joining script and

may even be subject to harsh feedback from senior members when they experience challenges in completing this script [17][7][6]. Software crowdsourcing models such as competitions are designed to identify high quality solutions by making it easy for many to contribute. But platforms for competitions still impose significant barriers to contribution. For example, one study of the developer competition platform TopCoder found that potential contributors face barriers from a lack of documentation, predicting the time commitment they were being asked to invest, understanding code structure and architecture, information overload, and from poor platform usability [18].

A number of crowdsourcing systems have explored approaches for applying microtask crowdsourcing to complex tasks. These systems rely on the concept of a workflow which decomposes a larger task into a sequence of microtasks. For example, Soylent [8] partitions proof reading and editing tasks into individual steps to identify a problem, make a fix, and verify a fix, each of which takes the form of a microtask which can be separately performed by a member of the crowd. TurKit [19] provides a framework for authoring scripts to create and run tasks in Mechanical Turk, enabling more flexible workflows. CrowdForge [20] expands those solutions by enabling the crowd to partition work. However, these systems are limited in that they assume that all of the microtasks can be enumerated and generated at the very beginning through a workflow where each input data element is transformed through a fixed sequence of steps. This approach cannot be used in microtasking programming, as programming tasks cannot be decomposed in the same way and fully enumerated up front. For example, it is impossible to predict, from a specification of requirements alone, that it will be necessary to generate a microtask to fix a function to pass a test when even the existence of this function and test depends on the completion of prior microtasks. In this way, applying a microtask approach to programming work requires new techniques to dynamically generate microtasks in response to the current state of the work product.

Much work has explored approaches to crowdsourcing software development tasks [21]. Most of these have focused on software development tasks other than programming such as question answering, verification, testing, or UI design. Companies increasing employ crowd workers for testing [22]. Developers today often rely on the vast wisdom of the crowd offered on Stack Overflow by the many contributors that curate a knowledge repository of answers to programming problems [10]. Several systems have explored the use of crowdsourcing for recommending fixes to bugs [23][24] and compilation errors [25] and to checking and fixing unit test assertions [26]. To leverage larger pools of workers, several systems have adopted a gamification paradigm enabling non-specialists to contribute to verifying software models for correctness [27][28] or verifying the absence of security vulnerabilities [29]. Approaches for crowd-based requirements engineering aim to increase the involvement of end users in shaping the software they use [30][31][32]. Other work has explored using Mechanical Turk workers to test GUI functionality and usability [33].

Within approaches for crowdsourcing software development, a few systems have explored the application of microtask crowdsourcing to programming. Work has begun to explore an extension to the StackOverflow question and answer model in which, instead of employing a crowd to answer questions, the crowd itself directly edits the code in the codebase in response to questions [34]. In Apparition, crowd workers take small UI requests made by a client and implement them as a mockup [35]. In CrowdDesign, workers work to build small code snippets that generate visual output [36]. These systems demonstrate the potential for applying microtasking approaches to programming. But all are limited in scope, designed to respond to enable developers to manually contract out small tasks to others rather than for groups of developers to come together to produce programs, as occurs today in open source software development.

Collabode enables an "original programmer" to describe custom microtasks in prose, which are then completed by workers [37][38]. An evaluation of the system found that, while it was possible to use microtasks for programming, there were several significant issues with the workflow used. As workers relied on a global view of the entire codebase, it was sometimes distracting to see changes being made elsewhere. Managing the crowd imposed a large overhead for the requester, as they needed to answer questions about the request and evaluate each contribution in detail. Moreover, code often had subtle bugs, which was difficult for the requestor to find through code inspection. As workers were anonymous, they sometimes did not take responsibility for their work. These considerations directly motivated our design principles for microtask programming, as we describe in the following section.

Overall, the many existing approaches to crowdsourcing offer a number of important building blocks. Our work specifically targets the problem of reducing contribution barriers in open source software development, where groups of developers together work to build programs. Offering developers the ability to make small programming contributions in this setting poses several challenges which existing work has not yet tackled. How can developers work on automatically generated programming microtasks without needing the context of the entire system? How can microtasks be generated when it is impossible to generate all of the microtasks upfront? How can the quality of work be ensured, without having a client made manually responsible for inspecting each of the contributions made by the crowd? We next present a set of design principles for addressing these challenges.

4 Design Principles

In this section, we offer a set of design principles characterizing the microtask programming methodology. We offer motivation for our principles through examining existing systems as well as from our own experience designing and evaluating a series of early prototypes of our system.

4.1 Decontextualize Contributions

Traditional software development tasks require developers to first learn *context* such as the location of features in code, steps to build and run code, and who to ask for information [39][40][6][41]. Learning context creates a lengthy onboarding script, delaying the time at which developers can first make a valuable code contribution [7]. While it may not be possible to entirely eliminate the need for context, reducing context reduces the cost of onboarding and enables developers to contribute more quickly.

4.1.1 Local edits to a single artifact

Offering developers a single task to perform on an individual small artifact (e.g., a function or test) has several advantages. Rather than forage through a codebase for the appropriate artifact in which to make a change [42], developers are already offered the artifact. Reducing this context may substantially reduce the knowledge required to work productively in a codebase.

Allowing developers the ability to interact with only a single artifact at a time creates several challenges. Rather than rely on developers to visit call sites or function definitions to understand what a related function does, such information should instead be embedded in the interfaces of other functions. Rather than rely on developers to infer the runtime type of parameters by reading related code, parameters should be explicitly given static types. Careful consideration is required of the additional context developers may require to make changes.

4.1.2 Provide a preconfigured environment

Installing appropriate tools, downloading code from a server, identifying and downloading appropriate dependencies, and configuring a workspace to build a project all constitute significant barriers to contribution [6]. Offering a preconfigured environment may substantially reduce these barriers, thereby reducing onboarding time. Preconfigured environments might be offered through a preset virtual machine containing a configured development environment, installed libraries, and scripts or through a web application.

4.2 Automatically Generate Microtasks

Microtasks describe the next steps for progress to be made. Accurately and completely capturing all steps that are necessary is important so that work is not lost and progress is not stalled. At any given point in time, there may be many concurrent microtasks in progress. One approach would be to have a manager explicitly create each microtask and evaluate each resulting contribution. However, for short programming microtasks by transient contributors, this approach takes more time for the manager of the microtask than if they had simply done the work themselves, negating the benefits of microtasking [37][38]. Moreover, it also assumes a manager or managers that are always available whenever work is ongoing at any time. Traditional microtask crowdsourcing systems in domains outside programming have addressed the issue by instead using completely automated microtask generation, where a client describes an initial request, the system is responsible for automatically and immediately generating the full set of microtasks to complete, and mechanisms such as redundancy and voting ensure quality [43]. In this way, there is no longer a cost for each microtask to be manually authored, managed, and evaluated by a requestor, and the work is instead organized by the system with contributions from the crowd.

However, automatic microtask generation brings new challenges. In traditional crowdsourcing approaches, microtasks are generated through a fixed sequence of steps, such as a MapReduce [44] workflow describing the steps where each input is transformed into an output [8][20]. In contrast, software tasks are dynamic and the functions and tests that might be needed, as well as the issues and bugs that might emerge, cannot be enumerated upfront. A different approach is required that generates microtasks dynamically as the program emerges.

4.2.1 Track artifact state

To track the current state of a program and its progress towards satisfying its requirements, the state of each artifact within the program can be separately tracked. Consider the state of a function as it is being created. At any point in time, a function might need a signature to be written, an implementation to be completed, or its code to be debugged. By tracking attributes describing what is or is not complete, a state machine can be used to describe the states through which the artifact may transition and order the work to be done (e.g., imposing a constraint that a function must have a signature before beginning its implementation). Whenever a microtask is submitted, the artifact may change its attributes (e.g., record that a signature has been added) and transition to a new state reflecting these attributes changes. Each transition may then generate an appropriate microtask. The evolution of an artifact need not even be monotonic. An artifact might transition back to a previous state after, for example, a developer adds pseudocode while fixing a defect.

4.2.2 Signal interface changes across dependencies

As workers edit artifacts, edits to an interface of an artifact may necessitate changes to artifacts elsewhere in the system. For example, adding functionality may require additional information to be passed in through a new parameter, which then requires all function call sites and tests invoking the function to be updated to provide this information. As a microtask spans only a single artifact, this requires a mechanism for signaling to other artifacts that an interface change has occurred.

In responding to interface change notifications, artifacts must generate appropriate microtasks. A key decision is to which interface changes dependent artifacts should respond. For example, a function might have a behavior change in its implementation, a change to the description of its behavior, or a change to its signature. However, it is difficult to determine in general if an implementation change or change to a function's description does or does not signal a change to the function's interface that might impact callers of the function. In our experience with early

prototypes, we found that workers often change descriptions simply to clarify the text and fix formatting and typographic errors. Signaling changes to callers of these function in such cases created many unnecessary microtasks, which substantially decreased productivity. A better alternative is to signal changes only for changes that clearly will require a change such as signature changes and rely on tests to catch other behavioral interface changes in the less frequent cases when they occur.

As developers work, they make use of task context describing the interfaces of other related artifacts. Developers may find that this task context is inconsistent with their artifact. For example, we found that workers writing tests sometimes realized that a behavior of a function could not be tested as described in the test case description [13]. As a result, it was impossible to complete the microtask they were asked to do. It is thus important to allow workers to report issues with task context, halting work on the microtask and creating a microtask for the other artifact to resolve the issue.

4.3 Achieving Quality through Iteration

A key consideration in all crowdsourcing approaches is designing mechanisms to protect against poor contributions and ensure high quality work [45]. Yet using crowdsourcing to solicit contributions from many contributors can also increase software quality, as the diverse ideas contributed by the crowd offer the building blocks to reach higher quality designs [15]. Rather than using a single manager or architect to oversee the project, the crowd itself is responsible for setting the direction of the project as work progresses through each of the individual contributions made. It is thus important to consider in detail the effects of decomposition, workflow organization, and coordination mechanisms on the quality of the output produced [46].

4.3.1 Encourage revision through sketching

Long contributions prevent the crowd from offering feedback on work as it is being done, enabling contributions to go far off track without input from the crowd. Low quality contributions may arise for many reasons. Workers may go off track because they are confused, because they are not knowledgeable enough to contribute a high-quality solution, because they put forth little effort, or because they wish to be actively malicious. Whatever the reason, even an individual low-quality contribution can significantly reduce the overall quality by introducing decisions that make subsequent implementation work more challenging or through introducing defects.

One mechanism to guard against low quality contributions is to reduce the amount of damage possible by any individual contribution. By making contributions smaller, many workers have the opportunity to contribute to the same artifact, and any problematic contribution can be revised. In this way, a worker who makes a mistake may have it quickly revised. However, we found in our preliminary studies that, despite explicit instructions otherwise, developers sometimes expected to continue work on a single artifact until its completion, reflecting the expectations developers bring to programming tasks. For example, in one informal study with an early prototype, a worker worked for 66 minutes within a single *Edit a Function* microtask. From this, we learned that is important to create mechanisms that limit contribution size such as bounding the maximum time or size of a contribution.

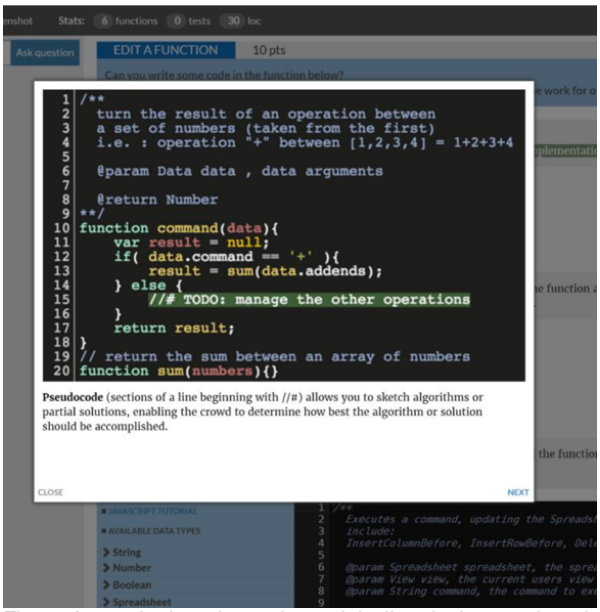


Fig. 2. A step in the microtask tutorial offered when workers begin their first *Edit a Function* microtask.

Programming tasks often require more than a few lines of code. In these cases, a developer may first begin working on a function and handoff work to others to continue. In such cases, it is important for the developer to have ways to communicate the intent of their approach through mechanisms for *sketching* an implementation design. Sketches may take the form of pseudocode, where a developer writes a high-level plan for an algorithm without worrying about thinking through all the details. In cases where steps in an algorithm may contain whole units of functionality, a developer may write function *stubs*, describing functionality that the function is expected to offer. In this way, developers are able to make contributions at a higher level of abstraction which subsequent developers may then fill in.

4.3.2 Support reviews and tests

A second mechanism for guarding against low-quality contributions is to explicitly check the quality of a contribution. Creating an opportunity to review contributions provides a quality gate through which contributions must pass and offers possibly valuable feedback to the contributor. Reviews can quickly identify work from new contributors that is headed in an unproductive direction and educate confused workers about the correct way to work.

Reviews offer a form of redundancy, where two contributors together must sign off on work for it go forward. Of course, as with any redundancy, the reviews may themselves be mistaken. If reviewers cause contributions to be discarded, this can be particularly problematic in causing valuable contributions to be discarded. One solution is to only allow reviews to be used to revise work. In this way,

further redundancy is introduced, as subsequent contributors must again react to the previous contributions. For example, a contributor revising a contribution in response to the review may decide that the requested revision is itself mistaken and ignore it. In this way, subsequent workers can then see the previous contributions and take whatever actions they see fit.

As in traditional software development, tests offer an important quality gate. Moreover, by dividing writing tests and implementing features into separate microtasks, tests provide an opportunity for another developer to provide a new perspective on the function, embodied in their test, which may reveal issues the implementer might not have considered. However, it may be in some cases that the test itself is incorrect rather than the function. It is thus important to offer facilities for revising both functions and tests as necessary.

5 System

Based on our design principles, we have developed the CrowdCode environment for microtask programming. In the following sections, we describe the user workflow, system architecture, and its implementation.

5.1 Workflow

All work begins with a client request which describes an API to be implemented through a set of function descriptions and signatures, a set of ADT descriptions, and a set of acceptance tests. All worker contributions are made through microtasks. As workers first visit CrowdCode, they are shown a Welcome screen explaining the basic ideas of CrowdCode and offering a short interactive tutorial explaining the primary interface elements. When a worker first begins a microtask they have not previously done, they are provided an additional tutorial, explaining in detail how to perform the microtask using a series of examples (e.g., Fig. 2). The worker is then provided a microtask automatically assigned by the system, which they may choose to either complete and submit or skip.

Table 1 lists the microtasks in the CrowdCode environment. Fig. 3 depicts a simple example of the microtasks generated to implement a function. Following the creation of a new project from a client request, *Edit a Function* (Fig. 3.1) and *Write Test Cases* (Fig 3.2) microtasks are generated for each API function. Whenever functions are submitted which contain pseudocode, a new *Edit a Function* microtask is generated to continue work on the function.

Writing tests is decomposed into two steps. In the *Write Test Cases* microtask, a worker first specifies a test plan for the function by enumerating a list of test case descriptions identifying behaviors that should be tested. In the second step, each test case generates a separate *Write Test* microtask for the test case to be implemented as a unit test, enabling workers to separately complete each test in parallel (Fig 3.4).

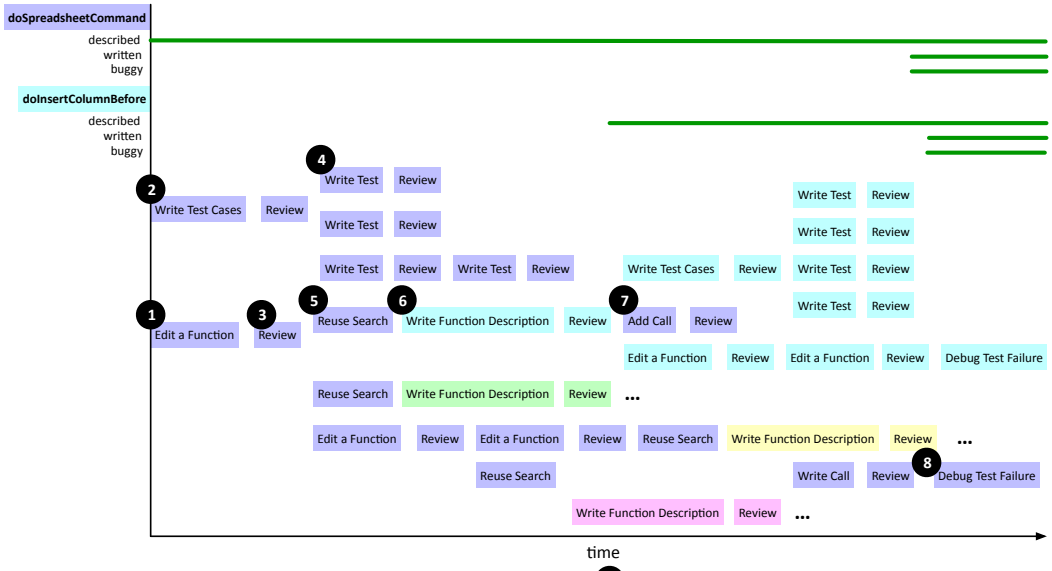
When functions are submitted which contain one or more new function stubs, *Reuse Search* microtasks are generated in which workers have the opportunity to find existing functions offering similar functionality to the requested stub or indicate that no such existing functions exist (Fig. 3.5). In this case, a new function is created and a *Write Function Description* microtask is generated to author a signature and textual description of the new function from the provided function request and implementation of the requesting function (Fig. 3.6). Work then recursively continues and microtasks are generated to *Edit a Function* and *Write Test Cases*. In parallel, an event is sent to the calling function and a *Write Call* microtask is generated and queued (Fig. 3.7).

When a function contains no remaining pseudocode, it is *written* and ready to be tested. As each test becomes implemented, it is executed against the function. If a function fails one or more tests, a unique *Debug Test Failure* microtask is generated for each failing test, providing all passing tests and exactly one failing test (Fig 3.8). Workers can then edit the function so that it passes the previously passing tests and the additional failing test, insert new pseudocode and stubs, or report an issue with one or more of the function's tests.

Of course, the defect may not be in the function with the failing test. This poses a challenge: how can fault localization be decontextualized to enable developers to work modularly with a single function, when it seemingly first requires knowing the function containing the defect? To

TABLE 1
CrowdCode Microtasks

Microtask	Editor	Context views	Possible contributions
Write Function Description	Function description editor	Stub Viewer with requested function, implementation of requesting function, ADTs	(1) Function description, (2) Report function as not implementable
Edit a Function	Code editor	ADTs, diff of change to function signature (if any)	(1) Code, pseudocode, and stubs, (2) Report function as not implementable
Debug Test Failure	Code editor	ADTs, Test Runner	(1) Code, pseudocode, and stubs, (2) Report issue in test
Reuse Search	Function search	Stub View of requested function, implementation of requesting function	(1) Identify existing function providing requested behavior, (2) No function found
Write Call	Code editor	Description and signature of identified function, ADTs	(1) Code, pseudocode, and stubs
Write Test Cases	Test case editor	Description and signature of function, ADTs	(1) List of test cases, (2) Report issue with function
Write Test	Test editor	Test case, description and signature of function (and diff, if any), ADTs	(1) Test, (2) Report issue with function, (3) Report issue with test case
Review	Review	Contribution and original task context	(1) Review and rating


doSpreadsheetCommand
described
written
buggy
doInsertColumnBefore
described
written
buggy
Write Test
Review
Write Test Cases
Edit a Function
Reuse Search
Write Function Description
Add Call
Write Call
Debug Test Failure
...
time


1
EDIT A FUNCTION
10 pts
Can you write some code in the function below?
TIP: If there's a lot to do, be sure to use function stubs and/or pseudocode to leave some work for others
CONVENTIONS FOR WRITING FUNCTIONS
Use PSEUDOCODE to sketch an implementation by using the syntax '//# sketch of implementation'.
Use FUNCTION STUBS to request a function call to a new or existing function. Call the function as normal, and define a FUNCTION STUB at the bottom with a short description, header, and an empty body.
Note: all function calls are pass by value (i.e., if you pass an object to a function, and the function changes the object, you will not see the changes).
JAVASCRIPT TUTORIAL
AVAILABLE DATA TYPES
String
Number
Boolean
Spreadsheet
Row
Cell
Style
Position
Range
View
Report function as impossible (or inadvisable) to implement
2
WRITE TEST CASES
10 pts
Can you describe some test cases in which this function might be used?
Are there any unexpected corner cases that might not work?
TIP: You don't need to specify concrete, executable tests, only high-level descriptions of scenarios to be tested.
FUNCTION DESCRIPTION
Report an issue with the function
Describe a test case
parameters are invalid
command is InsertRowBefore
command is DeleteColumn
3
REVIEW WORK
5 pts
A worker was asked to edit the code of the function doSpreadsheetCommand.
Can you review this work?
DIFF OF EDITS TO FUNCTION
RATING SYSTEM
1 Star: Incoherent or unfocussed
2 Stars: Unconvincing or weak
3 Stars: There are some weakness
4 Stars: Good quality, without weakness
5 Stars: Excellent without weakness
RATING
WORK ACCEPTED
REVIEW
nice decomposition!

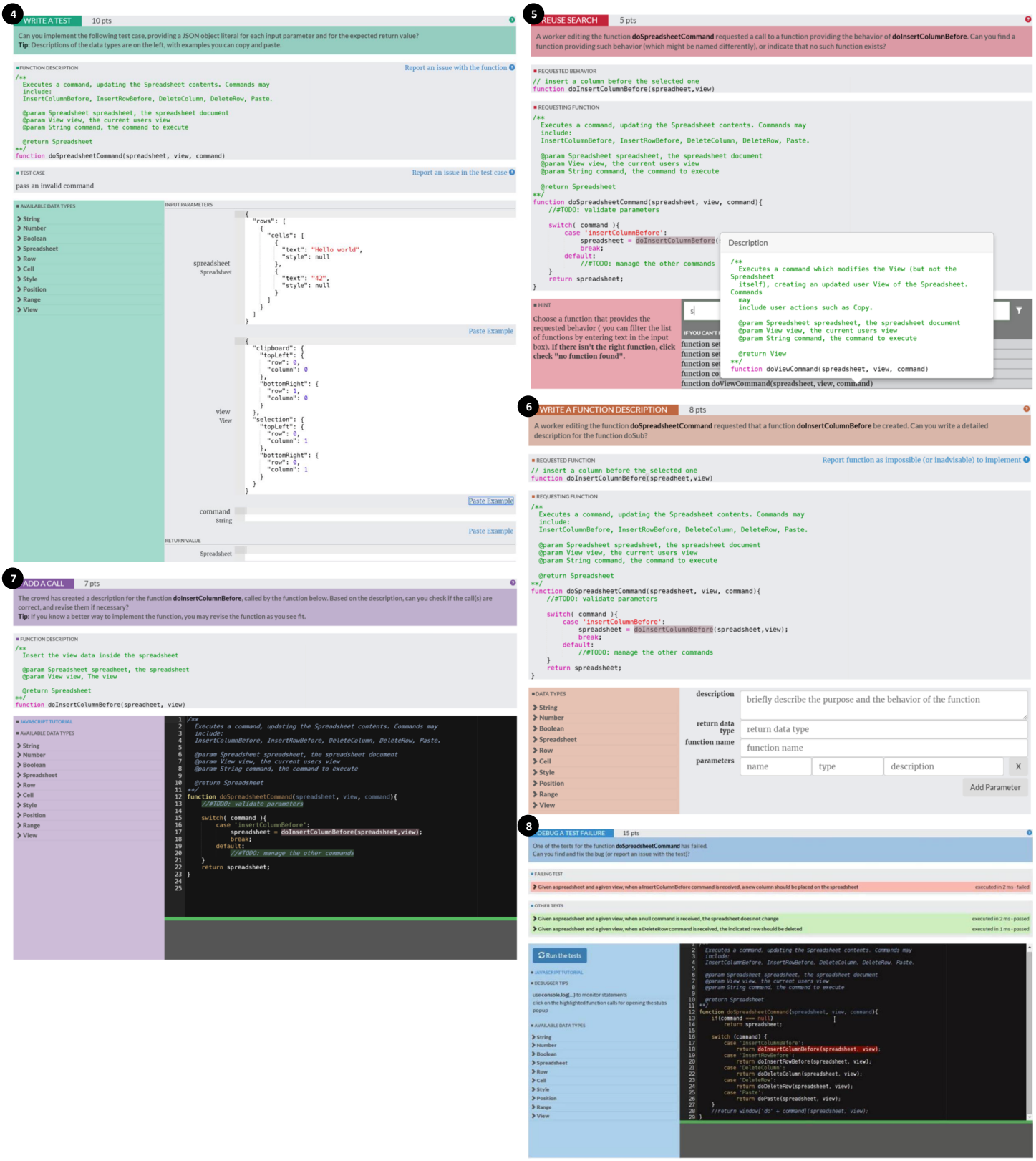


Fig. 3. An example of microtasks generated to implement a function doSpreadsheetCommand. The top diagram depicts the microtasks generated (boxes) over time (x axis) and the current state of each artifact (green lines). From an initial client request describing this function, two microtasks are first generated to (1) *Edit a Function* and to (2) *Write Test Cases.* In microtask (1), a request for a function doInsertColumnBefore is made. After submission, a corresponding *Review* microtask is generated and which accepts the submission. The request for doInsertColumnBefore then generates a microtask to conduct a (5) *Reuse Search* to determine if a new function should be created. No existing function is found. As Reuse Search microtasks are not reviewed, a (6) *Write Function Description* microtask is generated to describe the requested function. This creates a new doInsertColumnBefore function, initially in the described, !written, and !buggy state, generating microtasks to *Edit a Function* and *Write Test Cases* to begin its implementation. After it is described (and while it is being implemented and tested), an *(7) Add Call* microtask is generated to update the call site from the original request to match the description written in (6). In parallel to this work, each of the test cases specified in (2) generate corresponding *Write Test* microtasks, such as (4). After doSpreadsheetCommand transitions into the described and written state and its tests are implemented, the tests are executed, resulting in a test failure, which then generates a (8) *Debug Test Failure*

address this challenge, CrowdCode uses a modular debugging process using stubs. At any point when debugging a function, workers may hover over a function call to view the actual parameters and return value. If a worker sees a return value for a function that does not appear to match the description of a function, the worker may then edit the output of the function, automatically creating a *stub*. Rerunning the tests then enables the worker to check if the proposed change to the function behavior would fix the defect and cause the test to pass. If the microtask is submitted with a stub, the requested change in behavior is then propagated to the invoked function, creating a corresponding test which is then run and will fail (unless the function has concurrently changed). In this way, the fault localization process recursively continues across function invocations, creating a new *Debug Test Failure* microtask for each relevant function.

After each microtask is submitted, a corresponding *Review* microtask is first generated before the microtask is marked as completed (except for the *Debug Test Failure* and *Reuse Search* microtasks which are not reviewed). *Review* microtasks show workers the contribution and the original task context and ask workers to provide a quality rating on a five-point scale. Microtasks that receive a score of 1-3 are marked as *Reissue* and must include a review explaining the reason; otherwise, the microtask is marked as *Accepted* and the review text is optional. Workers are then informed of the review with a notification in their activity feed. Once accepted, the microtask is completed and the microtask content is used to update the corresponding artifact. If marked as a *Reissue*, a new microtask is generated which copies the original microtask and includes both the original contribution and review. This microtask is then assigned to a new worker to address the reported issue.

In viewing a microtask's task context, workers may find an issue with the task context which prevents them from completing the microtask (e.g., a test case that it is impossible to test). In such cases, workers may report an issue with a specific artifact referenced in the task context by describing the issue and submitting it instead of the microtask. A new microtask for the artifact is then generated to address the issue.

To ensure that contributions do not go off track and receive frequent feedback, microtasks are limited in time and contribution size. Microtasks are limited in duration to 10 minutes, indicated to workers through a bar showing the remaining time (Fig. 1-4). After 6 minutes, a warning message is displayed. If the microtask is not submitted after 10 minutes, it is automatically skipped and the work discarded. Additionally, contributions in the *Edit a Function* and *Debug Test Failure* microtask are limited to a net increase of ten statements, in addition to any pseudocode (Fig. 1-3).

CrowdCode provides workers with an overall sense of the progress of the project, listing the total lines of code across all functions and the number of functions and tests (Fig. 1-9). Workers can interact with other currently logged in workers through a global chat. Finally, CrowdCode provides a basic gamification system encouraging contributions. Workers submitting microtasks are awarded points proportionally to the review score. To encourage workers to take up microtasks that others do not wish to do, skipping a microtask increases its value by 20%. Scores for each worker are indicated in a leaderboard (Fig. 1-8).

A project is complete when there are no remaining uncompleted microtasks. To use the code implemented by the crowd, the client may visit the administrative page for the project at any point and download a current or past version of the code created.

5.2 Services

Microtask programming is fundamentally distributed, requiring coordination and synchronization between contributions being made separately by each contributor. CrowdCode coordinates work through a central service that is responsible for processing new contributions, updating the state of artifacts, and generating new microtasks.

Microtask generator. After contributions have been submitted by clients and approved by a reviewer, contributions are used to update the current version of the artifact. Based on these updates, the attributes of the artifact may change. Three attributes determine the overall state of each function: whether it has a function description (described), whether it has a complete implementation that contains no pseudocode (written), and whether it currently has any failing tests (buggy). After a function has been updated, these attributes are examined to determine which microtask should be generated next. If a function is not described, a *Write Function Description* microtask is generated. If a function is not written, an *Edit a Function* microtask is generated. If a function is buggy, a *Debug Test Failure* microtask is generated. Fig 3. depicts an example of artifact transitions and the resulting microtasks generated.

Work on a function is complete and no microtasks are generated when it enters the described, written, and not buggy state. However, the function may again enter the buggy state when a function which calls it adds a new stub and propagates a test to the function. In such a state, workers may freely edit the function and add pseudocode and it might again transition to a not written state.

Each function has a set of dependent functions which contain a call site to the function. A message is sent to each of a function's dependencies whenever it changes its interface by changing its name or parameters. This message then generates an *Edit a Function* microtask for each caller. Certain microtasks also enable reporting an issue with a function (Table 1), which generates an *Edit a Function* microtask for the function.

Distributed test executor. To reduce server resource usage and facilitate scalability to large crowds, all tests are executed on clients rather than the server. Each client maintains a test execution service which executes a test and returns a test result. After every edit to a function, a run of all tests is scheduled for execution by the server. This work is distributed to clients, and the clients report back with the outcome of each test execution. Test failures generate *Debug Test Failure* microtasks. Additionally, within the *Debug Test Failure*, workers can directly run all of the tests for a function.

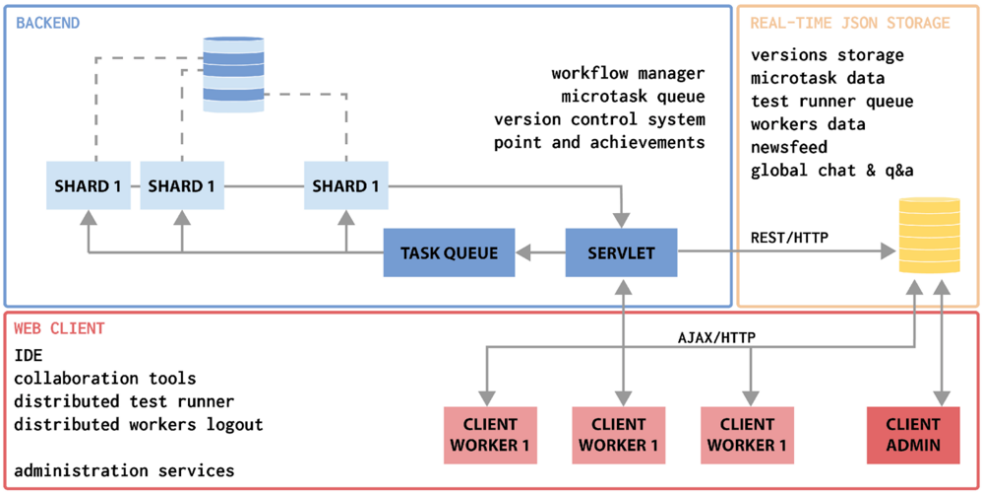


Fig. 4. CrowdCode is architected as a client-server application with a trusted backend, untrusted web clients, and a real-time JSON datastore.

Version control system. To enable tests to be executed on clients, clients must have the corresponding implementation of functions under test. As functions may arbitrarily invoke other functions, clients must have code for all functions in the system. CrowdCode thus maintains a simple version control system, synchronizing current code and tests from the server to all clients. The server ensures that there is never more than one active microtask for each artifact, so merge conflicts do not occur. Whenever a contribution is accepted and an artifact is updated, a new version of the artifact is created. This new version is then synchronized with all currently active clients.

Activity. Various backend operations keep track of states used to populate client views. This includes the activity feed for each worker, leaderboard, and project statistics. The Activity service manages this state and broadcasts updates to clients.

5.3 Architecture

CrowdCode is architected as a client-server system consisting of web clients, a backend, and a real-time NoSQL data store (Fig. 4). The backend and NoSQL datastore both expose a RESTful interface. Clients retrieve and submit microtasks from the backend, which processes microtasks and updates data in the real-time NoSQL datastore. The backend and datastore are sharded for scalability, where the backend is organized into separate components consisting of an individual entity including artifacts, workers, and the project and is backed by a separate region of the datastore. When the server receives multiple concurrent requests, each component is able to handle requests to its own state and update its state in parallel with other components without creating race conditions or resource contention.

In web application architecture, clients are traditionally considered untrusted, as any user of the web application may edit the code that runs on the client. In contrast, servers are considered trusted, as only the application developers who own the server can change the code that runs there. For example, a malicious crowd worker might edit the client implementation to delete all of the project's code or give themselves a more favorable activity history. To prevent such attacks, all clients must submit all changes to application state through the server, which verifies the requested change before updating application state and posting updates to the NoSQL datastore. After updates have been posted to the datastore, the datastore directly broadcasts updates application state to the clients, including updates to the Version Control System and Activity Services.

6 Evaluation

As microtask programming represents a significant departure from a traditional development approach, it brings many basic questions about its feasibility. In contrast to traditional software development approaches, microtask programming is intended to enable developers to contribute quickly, both in onboarding to a new project and making short contributions. The novelty of the methodology raises many questions about the quality of the resulting programs produced. For these reasons, we sought to evaluate (1) the feasibility of the approach, (2) the speed of onboarding, (3) the speed of making contributions, and (4) the effects of the quality control mechanisms. To investigate these questions, we conducted a user study. Participants worked entirely online, interacting only through the platform. 14 developers were divided into two crowds which each separately used CrowdCode to collectively work on a programming task.

6.1 Method

We recruited fourteen participants to work remotely from the US, Argentina, Brazil and Portugal. Participants were recruited through personal contacts. All participants had a computer science related degree and between 2 months and 12 years of industrial experience, with an average of 4 years. All participants had previous experience programming in JavaScript. Participants were compensated $100 for their time.

All interactions with the experimenters occurred via

email and IM, and all interactions between participants occurred through CrowdCode. We organized participants into two groups (6 in Session A and 8 in Session B) to offer two opportunities to observe crowd work and reduce the impact of variation between groups on results. Both groups were recruited from the same participant population and included participants across a range of expertise levels.

Several forms of data were collected during the study sessions. The server was instrumented to log all changes to artifacts, microtasks generated, submitted, skipped, and reissued, and use of the chat feature. To log more fine-grained data of participants' interactions, screen recordings were captured from all participants. Additionally, participants completed two surveys on their experiences. Midway through the sessions, participants completed a survey on their experiences and challenges with microtask programming. At the end of the session, participants were asked to complete a second survey on their experience. In total, each session lasted 5 hours.

At the beginning of each session, participants were given a brief written introduction to the purpose of the study through an email, asked to install a screen recorder, were provided instructions to log in to CrowdCode, and asked to begin by logging in. After logging in, participants were then given a series of tutorials within the CrowdCode environment explaining the overall environment as well as introducing each microtask type whenever they worked on a microtask of a new type. Each participant worked independently at their own computer.

Participants in each session worked together to implement a component for the core behavior of a simple interactive drawing application. Functionality focused on creating, manipulating, and rendering drawing elements. Specifically, functionality focused on translating user inputs specified as mouse actions into updates to an underlying model of the drawing and generating rendering primitives from a model of the drawing. The task was specified through a client request specifying a signature and a short description in the form of comments for each of four functions (`createElement`, `createAction`, `renderDrawing` and `moveElement`). These were supplemented with ADTs describing the parameters to these functions (`Element`, `Position`, `Segment` and `Action`) and several representative examples of each of these ADTs. Other than the function signatures, no code was provided in the client request. From this client request, CrowdCode then automatically generated two microtasks for each of the four functions: 4 *Edit a Function* microtasks and 4 *Write Test Cases* microtasks. As participants began work, they were automatically assigned one of these microtasks by the environment.

6.2 Results

6.2.1 Feasibility

During the two sessions, participants submitted 1,008 microtasks and implemented a total of 22 functions, including 8 API functions (4 functions in 2 sessions) and 14 functions participants created from scratch. Participants' final output encompassed 490 lines of code and 149 unit tests with 2920 lines of code. On average, participants created 7.8 tests per function. Nearly half (47%) of the tests created were for the 36% of functions which were API functions, perhaps due to their greater complexity or because they were first. Across the total of 70 hours of participant time in the two sessions, 44 hours were spent on microtasks that were submitted (referred to as "contribution time"). Participants spent the most time on *Review* microtasks (37%), *Write a Test* microtasks (22%), and *Edit a Function* microtasks (17%). Table 2 lists the time spent per microtask type. The remaining non-contribution time was spent on reading study materials, completing the two surveys, working on microtasks that were skipped rather than submitted, and waiting for microtasks to be assigned.

All 1,008 microtasks workers submitted were automatically generated by the system. For example, Fig. 5 depicts the microtasks generated for the function `createAction` in Session A. *Edit a Function* and *Write Test Cases* microtasks were iteratively generated, completed, and reviewed until

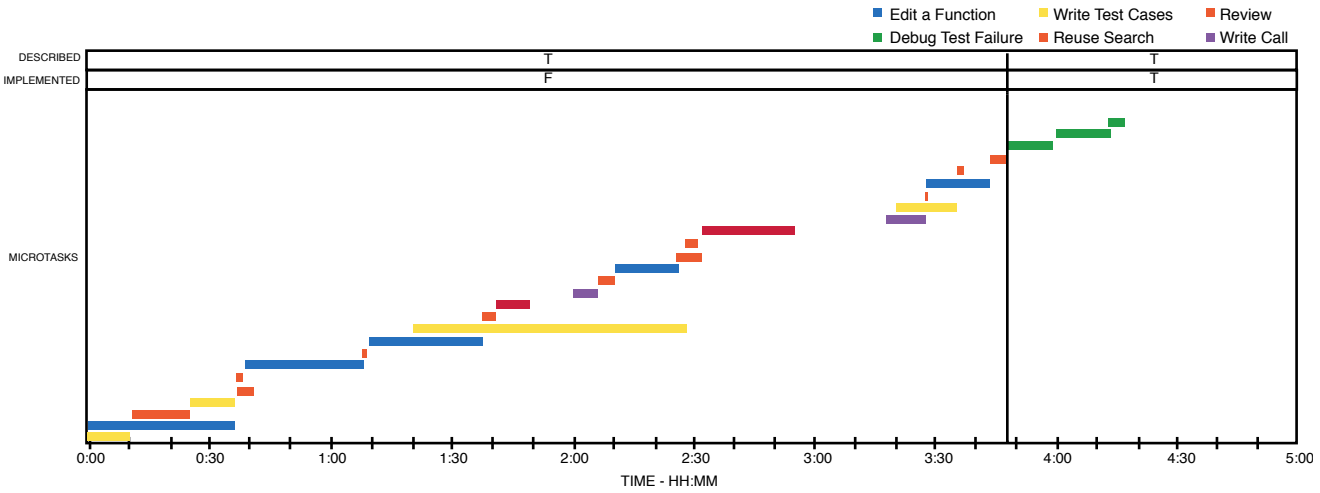


Fig. 5. Microtasks generated for the function createAction during Session A. Each bar corresponds to the time at which an individual microtask was first generated and the time at which it was submitted (microtasks were assigned to workers at a time after being generated). The top two rows depict the times at which the function changed state.

the work was complete, generating additional microtasks to respond to requests for additional functions. The submission of the last microtask (a *Review* microtask) before the dashed line resulted in the function transitioning into the implemented state, triggering the execution of tests and generating separate *Debug Test Failure* microtasks for each failing test. Other functions behaved similarly. Functions sometimes transitioned multiple times between the implemented and not implemented state. Participants rarely changed function interfaces once defined, with only two changes in Session A and zero in Session B.

Participants implemented 22 functions and 149 unit tests through the submission of 1,008 microtasks, demonstrating the feasibility of programming through microtasks.

6.2.2 Speed of onboarding

After logging in to CrowdCode, participants first read a brief tutorial introducing microtask programming and the basic user interface elements of the platform and were then assigned their first microtask. At this point, participants often spent time familiarizing themselves with the environment. In some cases, participants began working with their first assigned microtask. In other cases, participants skipped several microtasks because they wanted to look at several types of microtasks before starting to work. Overall, participants on average submitted their first completed microtask after 14 minutes and 32 seconds.

The first microtasks participants completed were often the most challenging, as they learned aspects of microtask programming such as pseudocode, requesting functions, and reviews. One reported that "*It was pretty easy once you already started completing a few microtasks. At first it may seem a bit strange because you have little time, but you get used to manage it.*" *Review* microtasks helped identify issues with early microtask submissions. On average, the review score for the first submitted microtask was 2.8 out of 5. 78% received a score of 3 or less, leading them to be reissued to a second participant.

Rather than incur learning costs all at once upfront, learning costs for specific microtasks were deferred until participants first encountered that microtask. At this point, participants again needed to learn a new microtask, incurring time and reducing quality. Participants first read a second microtask-specific tutorial before beginning work on the microtask. Completing a microtask of a new type raised completion times from a median of 1 minute and 31 seconds to 5 minutes and 29 seconds as well as decreasing median quality scores from 5 to 3. One participant reported that "*I could not understand each task until it went around a couple of times*" and "*At the beginning, everything was very confused. Gradually I began to understand*". Another suggested that an interactive demo microtask enabling them to complete a sample microtask might help speed learning.

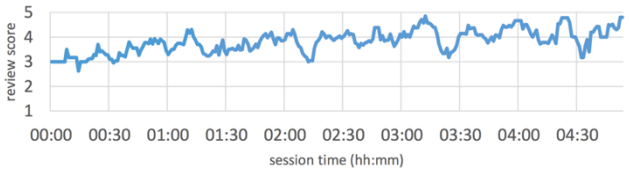


Fig. 6. Average review score across the two sessions, averaged using a ten-minute sliding window.

As participants through the sessions gained experience with CrowdCode, average review scores tended to increase before reaching a plateau. Fig. 6 depicts the moving average review score with a ten-minute time window.

On average, participants were able to complete their first microtask less than 15 minutes after they began onboarding.

6.2.3 Speed of contributions

Overall, workers were able to complete all microtask types in both sessions in a median time under 5 minutes (Table 2). Completion times were greater for larger microtasks such as *Edit a Function*, *Write Test Cases*, and *Debug Test Failure*, with median completion times reaching as high as 4:57 or 4:00 for *Write Function* and *Debug Test Failure*. Shorter microtasks had median completion times below 2 minutes, including *Reuse Search, Write Test, and Review*. In all cases, median completion times were less than half the ten-minute cutoff.

Participants skipped 13% of all the microtasks they began. Skips occurred for a variety of reasons. In some cases,

TABLE 2
MICROTASKS COMPLETED, SKIPPED, AND REISSUED

Microtask type	Completed		Skipped		Reissued		Median time (mm:ss)		Total time (hh:mm:ss)	
Session	A	B	A	B	A	B	A	B	A	B
Review	260	227	22	22	-	-	1:27	1:14	9:29:32	6:43:43
Write Test	158	102	22	7	40	41	1:29	1:21	6:35:41	3:15:12
Edit a Function	44	56	25	21	16	22	4:57	2:31	3:59:28	3:40:03
Write Test Cases	40	30	9	4	11	13	3:50	2:28	2:57:02	2:05:53
Debug Test Failure	14	18	5	6	1	4	2:32	4:00	0:57:22	1:21:21
Write Function Description	8	16	3	0	0	10	3:12	2:44	0:30:21	1:03:58
Write Call	7	9	2	2	0	3	1:37	2:28	0:15:02	0:36:49
Reuse Search	9	10	3	0	0	3	0:42	1:35	0:06:37	0:22:33
Total	540	468	91	62	68	96			24:51:05	19:09:32
Overall total	1008		153		164				44:00:37	

participants skipped microtasks because they seemed unable to complete them. In other cases, participants skipped several microtasks in succession simply to explore the available microtask types before choosing which to begin with. 17% of the total microtask skips occurred as participants ran out of time on a microtask and the system automatically skipped the microtask. Some participants felt rushed at points, particularly participants that erroneously viewed the microtask to ask for more than it did. "*The ten minutes for tasks sometime made me feel in a hurry, especially when I had to start to write a new function from scratch and I had to read and understand the specifications.*"

To measure contributions on microtasks where participants contributed code, we counted the number of lines touched, including lines added, edited, and removed. Participants added an average of approximately 3 lines of code and removed and edited approximately 1 line of code each, touching an average of 5 lines.

Participants were able to complete microtasks of all types in a median time under 5 minutes.

6.2.4 Effects of quality control mechanisms

In both Session A and Session B, participants did not have time to completely implement the components within the 5-hour sessions. At the end of each session, there was still additional work to be done to fully implement the functionality specified in the client request. To assess the overall progress and quality of the contributions made by participants, we constructed a test suite (not provided to participants) and then edited the final code in each session until all tests in the test suite passed. In Session A, 3% of the contributed code (7 lines) had to be edited and an additional 3% (6 lines) of code had to be added. This included fixing small typographical errors (e.g., adding an array index, fixing misspelled variable names) and implementing several lines participants had sketched with pseudocode. In Session B, 12% (33 lines) of the code had to be edited and an additional 25% (69 lines) added. Additions included implementing two functions that participants created but had not yet implemented.

Throughout the sessions, participants iteratively implemented and revised function implementations. Individual functions often reflected contributions from several participants. Fig. 7 depicts the implementation contributions made in Session A. On average, approximately 3 workers contributed to the code of each function, increasing workers' perception of the quality of the resulting code. One participant reported "*Every bit of code was checked by at least three people, usually more. I felt that it came out pretty solid.*" Implementation contributions often reflect a single step which later contributions then built on. Fig. 8 depicts three contributions made to the function `createAction` during Session A. One worker first implemented one case of the function, leaving pseudocode for additional cases. In subsequent iterations, one worker reworked the case logic and another worker implemented logic for additional cases.

To identify and correct defects, participants used the test and review systems. Workers implemented an average of 6.8 tests per function. As tests failed, *Debug Test Failure* microtasks were generated. Failing tests were often caused by typographic errors, incorrect use of JavaScript syntax, incorrect function invocations, and erroneously implemented functionality. 37% of participants' contribution time was spent on *Review* microtasks. The review system helped to identify defects such as typographic errors in the code, missing test cases, erroneous tests, and incorrect use of data types or functions. One participant observed that

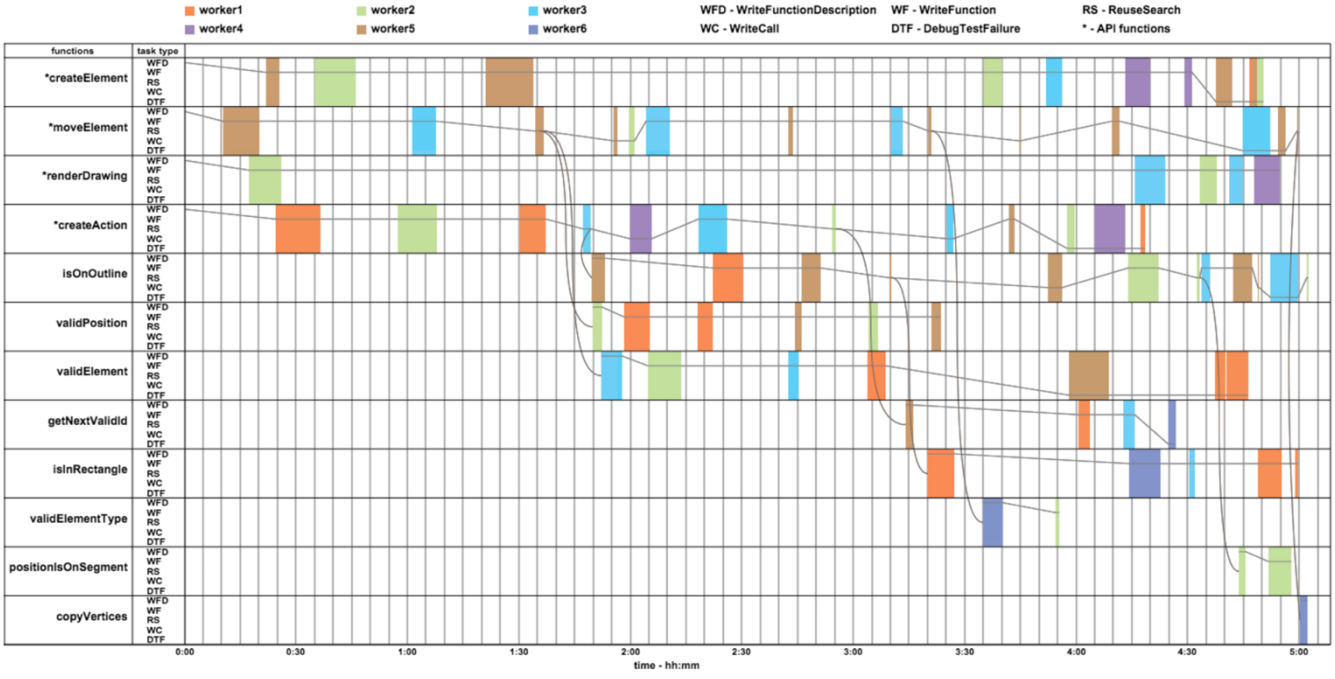


Fig. 7. A timeline of microtasks contributing descriptions or code to functions in Session A (test contributions and reviews are not shown). Functions were iteratively implemented by successive micotasks completed by different participants. As work progressed, participants requested the creation of new functions.

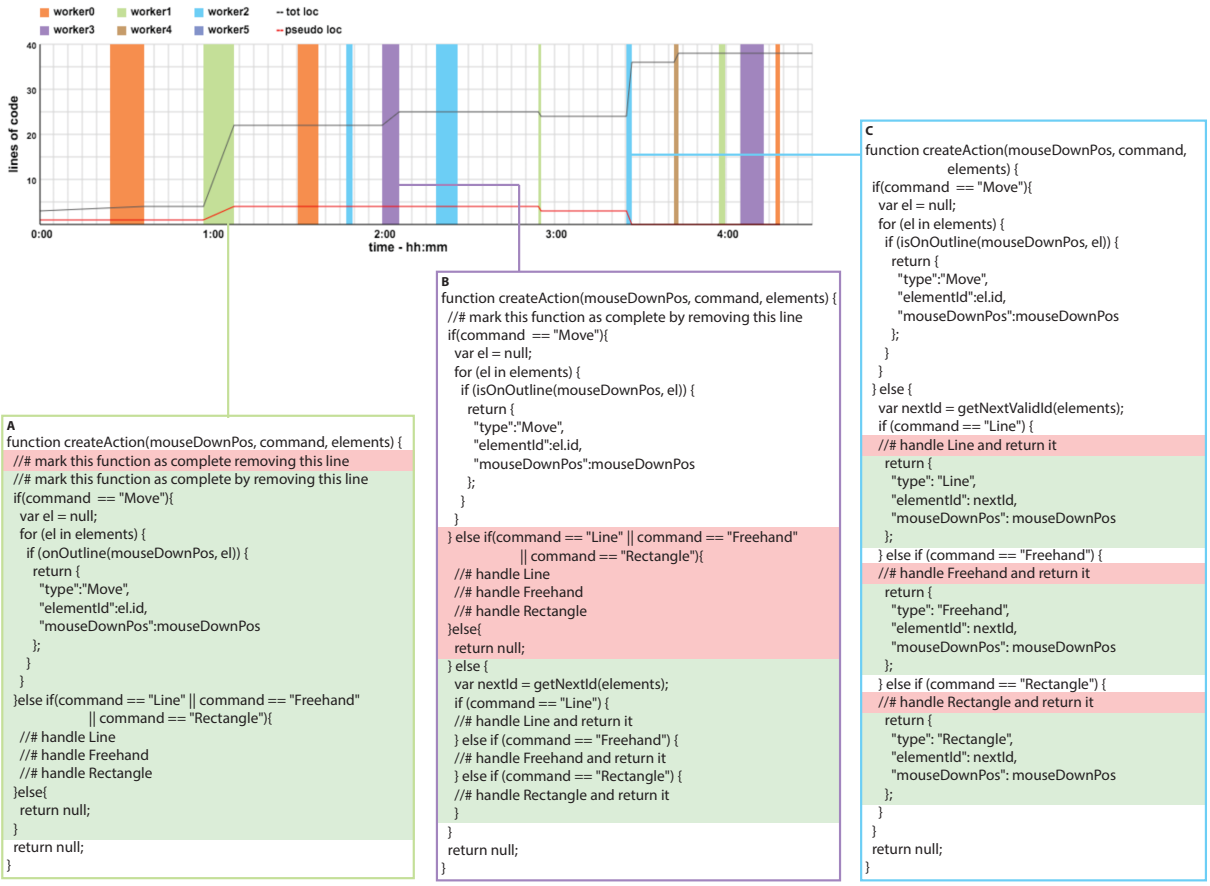


Fig. 8. Examples of contributions made by participants to the function `createAction` in Session A.

"You can learn from your mistakes that others point out or better ways to accomplish the same goal of a certain function".

Participants used the issue reporting system to report issues they identified with the current state of artifacts, identifying 63 issues. Many were related to text that was excessively vague or contradictory. The most common source of issues was in the *Write Test* microtask, where participants reported 43 issues. Most dealt with the test case being poorly described or difficult to understand. For example, one participant in Session B reported that a test case to *"Test if the function returns true when the element is correctly well-formed"* was not clear in defining well-formed. Participants also reported 13 issues from the *Debug Test Failure* microtask. After discovering that the code was correct but the test was wrong, participants used the issue reporting system to report an issue in the test. 11 times in Session A and 1 in Session B workers incorrectly interpreted the function specifications. For example, in Session A the test case *"creates a rectangle element with the previous element being a rectangle..."* was reported because *"the previous element is only used for Freehand, and is not defined in the other cases"*. Not all incorrect or divergent interpretations of function specifications were caught and flagged through the issue reporting system. Others were caught only when the tests were executed, creating a *Debug Test Failure* microtask to resolve the divergence.

Participants extended prior contributions, identified and reported issues with related artifacts, and fixed defects caught by tests.

7. Limitations and Threats to Validity

There are three major threats to the validity of our results: the selection of participants, the choice of the programming task, and the context in which participants worked.

First, our selection of participants may threaten the generalizability of the results if our participants are not representative of the envisioned users of microtask programming. As crowdsourcing, microtask programming draws strength from the diversity of its contributors. Thus, we recruited broadly, drawing participants from three continents and anywhere from 2 months to 12 years of industrial experience. Our results might differ for developers that are exclusively junior or exclusively senior or for participants with little development experience.

The second threat to validity is the choice of programming task. Microtask programming is intended to enable crowdsourcing the implementation of component logic. Thus, we chose a task designed to be reflective of typical application logic, and our results evaluated the feasibility for this task. Of course, building complete applications requires myriad additional software development tasks, including software design and architecture, GUI design and

implementation, and many others. Extending microtask programming to support such activities is an important focus of future work.

The third threat to validity is the context in which the task was performed. Microtask programming is intended for crowds of transient developers, who have no prior common affiliation or team membership and who coordinate and exchange information exclusively through the environment. We thus sought to recreate this context, ensuring that participants were not collocated and worked only through the environment. Another critical aspect of the context was to ensure that multiple workers were working concurrently within the same project, as in the intended context. To achieve this, we structured the study so that all workers began at the same time and worked in a contiguous block of time. This represents a simplified context, which might occur for crowds where workers have time to spend a few hours contributing. More work is needed to understand how workers behave when workers are constantly joining and leaving, particularly regarding the learning effects that workers experience over time through engagement with a microtask programming platform or with a specific project.

8. Discussion

Microtask programming envisions a software development process in which transient developers make short, self-contained contributions. This model might open contributions to open source projects to a long-tail of contributors who do not have the time for traditional open source onboarding. Enabling this model requires new mechanisms for decontextualizing large tasks into microtasks, generating microtasks, and ensuring quality.

In this paper, we proposed a microtask programming approach in which contributions are decontextualized based on function boundaries, microtasks are automatically generated by the system, and quality is ensured through continual iteration from many contributors. Our results offer evidence for the basic feasibility of such an approach. Developers were able to make over 1000 contributions to a range of tiny programming tasks, including editing code, describing functions, writing test cases, implementing unit tests, and debugging. Developers were able to make these contributions quickly, making code contributions to functions in under five minutes and creating tests in less than 90 seconds. Building on the contributions of others, developers were able to complete ideas sketched in pseudocode and fix issues introduced by others. Developers nearly finished implementing the components within the limited task times. Together, these results demonstrate that it is feasible to use short, decontextualized contributions made by the crowd to implement small components.

Yet the preliminary evidence of feasibility raises more complex questions about the value and appropriate use of microtask programming. Enabling small contributions comes at the cost of increased overhead, as each new contributor must first learn how to work in microtasks before starting and then make sense of the context and artifact before beginning each microtask. After a contribution is made, a handoff occurs and the next contributor must then get up to speed on the current state of the artifact. This overhead is clearly visible in the overall productivity numbers of the participants in our study. Across 44 hours of work time, developers wrote only 490 lines of code and 2920 lines of test code.

By forgoing the benefits and drawbacks of context for a short onboarding experience, microtask programming brings many potential new opportunities. Shirky argues that lowering the barriers to joining and contributing enables crowdsourcing to tap the cognitive surplus of otherwise wasted resources, utilizing contributions from those who would otherwise be uninterested or unable to contribute [11]. Given the large size of the overall developer population, even small increases in contributions to open source projects could have a dramatic effect.

Small contributions also open the possibility to increasing parallelism in software development. As many hands make light work, decomposing traditional software development tasks of implementing a feature or fixing a bug into tens or hundreds of microtasks might enable some of this work to be completed in less clock time than traditional approaches by parallelizing work across many developers.

Realizing the potential of microtask programming suggests a wide range of research questions. Our initial investigation explored decontextualizing programming at the function level, a design that greatly increases the potential for parallelism but at the cost of significant overhead. One might then consider a spectrum of task sizes from implementing a feature of fixing a defect down to contributions at the level of small edits to individual functions. Indeed, there is likely a large variation in contribution size even in traditional development. What is the tradeoff between parallelizing work and single ownership? A host of other design decisions might also vary, in how contributors coordinate, in contributors' awareness and knowledge of the system? Moreover, contributions might not be all homogenous in size and context, but might vary with differing roles and responsibilities. This raises many fundamental questions for workflow design and coordination, both for software development in a crowdsourcing context and more generally for team software development.

8.1 Workflow design

A fundamental challenge in crowdsourcing work is the design of a workflow, describing the set of tasks that exist and the dependencies between these tasks. Designing effective workflows is an important challenge in crowdsourcing. This requires careful consideration of the context offered each worker and the difficulty of completing microtasks given this context, the dependencies between tasks and how information flows through these dependencies, and the size of the contributions workers make.

Developers in our study were largely successful working on programming tasks with only a single artifact as their context rather than the whole codebase. Less context

reduces the amount of information developers must consume before beginning the task and reduces the amount of information that must be shared about in progress changes. Yet less context also increases the probability of conflict. For example, in our workflow developers on the *Write Test Cases* and *Edit a Function* microtasks both separately interpreted the description of the function. When these interpretations diverged, conflicts resulted. These divergent interpretations could be ultimately caught through the issue reporting system or by executing the tests against the functions. But a workflow that reduces the frequency with which divergent interpretations occur may be more efficient.

Another challenge occurs in understanding the consequences of decisions made on future tasks. For example, developers working on the *Write Test Cases* microtask were often very thorough, enumerating a long list of test cases for different scenarios. Developers then separately translated each test case into a unit test. This resulted in developers creating over five times as much test code as functional code and as well as creating tests which were redundant. While the developers enumerating test cases had the context to see the other test cases, they did not have access to the function or test implementations (which were not yet produced) to understand the redundancy in the tests being created. Moreover, developers who created the list of test cases were evaluated based on the thoroughness of their test plan rather than bearing the cost of implementing the tests.

Another key dimension of workflow design is thus not only the size and context of tasks but also the choice of decomposition, determining which subtasks are done together and which are done apart. A wide range of decompositions might be envisioned. For example, rather than having a workflow in which microtasks involve writing all test cases for a function and editing function code to implement multiple test cases, workers might instead write a single new test and implement the behavior only for this test. Investigating the space of possible workflows designs for microtask programming is an important area for future work.

8.2 Coordination

In traditional microtask crowdsourcing workflows, workers are assumed to be transient and free to come and go without commitment to future contributions. To enable this, each microtask exists as a unit of work assignable to any contributor, and each worker's ownership and involvement in the contribution could end after submitting the microtask. In contrast, in traditional development and open source development contributors may claim ownership over a task, discuss feedback they receive, update their contribution, and build up ownership to make similar contributions over time. Developers may then assume to have knowledge and awareness of a set of the system that they own and coordinate with others to maintain that awareness.

Developers in our study demonstrated that contributions can be made without ownership and with minimal awareness of the rest of the system. But transient contributions impose a cost, where contributors are less able to learn and specialize over time, where contributors are less able to share their knowledge when helpful, and there is a greater possibility of workers making diverging interpretations of design decisions.

One approach to this challenge is to remove the assumption that contributors may leave at any time and enable contributors to commit to involvement over a fixed period. For example, in Flash Teams, free agents join a project and then commit to involvement for a fixed period ranging from a specific phase to the entire life of the project [47]. This then enables the use of more traditional team and management structures. But committers are not able to remain truly transient and noncommittal in their involvement, limiting the potential for harnessing Shirky's long-tail of casual contributors [11].

Another approach is to create new coordination mechanisms for transient contributions. We have explored a variant of microtask programming in which contributors are able to explicitly coordinate about design decisions by creating, addressing, and discussing questions about design decisions [14]. Contributors still exert no ownership over artifacts and are still assumed to be transient without future commitments, even to answer questions that others have raised. But if contributors do choose to continue contributing, they are able to share their knowledge through the discussion system. In this way, contributors are not required to commit a priori to future contributions. But if they do stay involved, the system is able to leverage their expertise.

Many other coordination approaches might be possible. Rather than assuming all contributors are transient or nontransient, there might be hybrid crowds in which some contributors are transient and others act as team members. The programming competition platform TopCoder uses such a model, where senior contributors may choose to take a role as a co-pilot and manage the process of creating and administering each task [48]. Platforms such as MobileWorks have shown success using diversified roles in a microtask setting [49]. Applied this model to programming, contributors might be promoted or elected to leadership roles and given responsibility for overseeing coordination amongst team members by giving feedback on contributions, detecting and managing conflicting design decisions, and facilitating knowledge sharing. Investigating and exploring such new coordination approaches is an important topic for future research.

8.3 Motivation

Motivating and incenting workers to join and contribute is an additional key challenge. In order for the expected benefits of broader involvement in open source projects to materialize, developers that choose not to contribute today must be motivated to join and contribute. It is thus crucial to consider the potential factors that might influence joining and contributing to microtask programming projects.

A key source of insight comes from studies of open source software development communities as they exist today. Across several studies, a consistent finding has been

that it is a long and time-consuming process to join an open source project [17][7][6]. Our results offer evidence that microtasks can reduce these joining costs, as our participants were able to make their first contribution on average in only 15 minutes. By reducing joining costs, potential contributors to current open source projects without the motivation or time to endure a lengthy joining script might choose to contribute to a project employing microtasking.

But would the same motives that bring a potential contributor into an open source project still be present in a microtasked project? Reputation and career development through skill development are important motives in open source projects [4]. One might imagine that reducing context might enable developers to contribute more quickly but less meaningfully, as all contributors are reduced to completing tasks that are easy in requiring little context but offer less reputation and less skill development.

However, decontextualization does not mean all tasks will require little skill or offer little reputation. Some tasks will have an outsize influence, impacting the direction and shape of future work. For example, in the current workflow, the first task on a new function offers the worker the opportunity to sketch the implementation of a function, helping influence the later work to be done to translate this sketch into a full implementation. In cases where the function is particularly large, additional functions might be created to implement aspects of this approach. In this way, a single microtask might influence tens or hundreds of subsequent implementation, testing, and debugging tasks. Similarly, review tasks offer feedback on contributions and play an important role in directing and gating low quality contributions, potentially preventing poor contributions from snowballing into large amounts of wasted effort.

In decomposing large tasks into small tasks, microtasking makes possible greater specialization, as an individual might choose to focus on a specific type of task. In this way, microtasking offers new opportunities for skill development, as developers might choose to complete microtasks requiring specific expertise. For more experienced developers, this might mean focusing on tasks that set direction or help to mentor more junior developers. In giving more visibility into smaller contributions and reducing the barriers keeping less experienced developers out, microtasking offers new opportunities for mentoring.

Decomposing tasks also offers greater transparency into work. Rather than recording work done by contributors at the granularity of a commit, microtask development enables capturing more fine-grained information at the level of individual tests and short snippets of code. Each contribution is reviewed, generating a review score assessing quality; system events such as test failures are logged by the system. All of this offers new opportunities for surfacing additional reputation information about contributors. This information might be used in a host of ways, such as dashboard displays which publicize reputation information on the quantity and quality of contributions.

9. Conclusion

This paper has explored the possibility of programming through microtasks. Our results offer evidence for the basic feasibility of completing small programming tasks and combining these contributions to build small programs. We found that microtasking enables developers to onboard onto a project quickly. On average, workers were able to onboard and submit their microtask in less than 15 minutes. Rather than spend hours or days addressing a single issue, microtasks enable developers to contribute quickly, as we found that developers were able to complete microtasks in a median time under five minutes. These findings begin to lay a foundation for the use of microtasks in programming and opens the door to realizing the potential benefits of increased participation in open source projects through diminished onboarding costs as well as the possibility of reduced time to market through increased parallelism.

At the same time, many further challenges remain in using microtask programming to create larger and more complex programs, in better understanding the effects and implications of decomposition on the quality and speed of programming work, and in designing new and more effective workflows that best achieve short onboarding times as well as high parallelism. Even if some software development tasks remain so inherently complex that they cannot be microtasked, even partial microtasking may increase participation in open source communities by offering contributors a new and faster way to make begin to make important, if limited, contributions more quickly and easily and a gentler onboarding pathway into taking on larger and more complex tasks within projects as contributors gain contextual knowledge about a project.

Acknowledgment

We thank Steven Morad, Patrick Nguyen, and Eric Chiquillo for their contributions to early versions of CrowdCode, we thank the participants in the study for their participation, and we thank Jonathan Bell for editorial assistance. This work was supported in part by the National Science Foundation under grants NSF IIS-1111446, IIS-1302522, and CCF-1414197.

References

[1] T. D. LaToza and A. van der Hoek, "Crowdsourcing in Software Engineering: Models, Motivations, and Challenges," *IEEE Softw.*, vol. 33, no. 1, pp. 74–80, Jan. 2016.

[2] T. D. LaToza and A. van der Hoek, "A Vision of Crowd Development," in *International Conference on Software Engineering (ICSE), NIER Track*, 2015, pp. 563–566.

[3] K.-J. Stol, T. D. LaToza, and C. Bird, "Crowdsourcing for Software Engineering," *IEEE Softw.*, vol. 34, no. 2, pp. 30–36, 2017.

[4] K. Crowston, K. Wei, J. Howison, and A. Wiggins, "Free/Libre Open-source Software Development: What We Know and What We Do Not Know," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 1–35, Mar. 2008.

[5] Y. Benkler and H. Nissenbaum, "Commons-based Peer Production and Virtue*," *J. Polit. Philos.*, vol. 14, no. 4, pp. 394–419, 2006.

[6] I. Steinmacher, M. A. G. Silva, M. A. Gerosa, and D. F. Redmiles, "A systematic literature review on the barriers faced by newcomers to open source software projects," *Inf. Softw. Technol.*, vol. 59, pp. 67–85, 2015.

[7] G. von Krogh, S. Spaeth, and K. R. Lakhani, "Community,

joining, and specialization in open source software innovation: a case study," *Res. Policy*, vol. 32, no. 7, pp. 1217–1241, 2003.

[8] M. S. Bernstein *et al.*, "Soylent: A Word Processor with a Crowd Inside," in *Symposium on User Interface Software and Technology (UIST)*, 2010, pp. 313–322.

[9] F. Khatib *et al.*, "Algorithm discovery by protein folding game players," *Proc. Natl. Acad. Sci.*, vol. 108, no. 47, pp. 18949–18953, 2011.

[10] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design Lessons from the Fastest Q&A Site in the West," in *Conference on Human Factors in Computing Systems (CHI)*, 2011, pp. 2857–2866.

[11] C. Shirky, *Here Comes Everybody: The Power of Organizing without Organizations*. Penguin Books, 2009.

[12] T. D. LaToza, W. Ben Towne, A. van der Hoek, and J. D. Herbsleb, "Crowd development," in *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2013, pp. 85–88.

[13] T. D. LaToza, W. Ben Towne, C. M. Adriano, and A. van der Hoek, "Microtask Programming: Building Software with a Crowd," in *Symposium on User Interface Software and Technology (UIST)*, 2014, pp. 43–54.

[14] T. D. LaToza, A. Di Lecce, F. Ricci, W. B. Towne, and A. van der Hoek, "Ask the crowd: Scaffolding coordination and knowledge sharing in microtask programming," in *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2015, pp. 23–27.

[15] T. D. LaToza, M. Chen, L. Jiang, M. Zhao, and A. van der Hoek, "Borrowing from the Crowd: A Study of Recombination in Software Design Competitions," in *International Conference on Software Engineering (ICSE)*, 2015, pp. 551–562.

[16] E. R. Q. Weidema, C. López, S. Nayebaziz, F. Spanghero, and A. van der Hoek, "Toward Microtask Crowdsourcing Software Design Work," in *3rd International Workshop on CrowdSourcing in Software Engineering (CSI-SE)*, 2016, pp. 41–44.

[17] C. Jergensen, A. Sarma, and P. Wagstrom, "The Onion Patch: Migration in Open Source Ecosystems," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 70–80.

[18] A. L. Zanatta, I. Steinmacher, L. S. Machado, C. R. B. de Souza, and R. Prikladnicki, "Barriers Faced by Newcomers to Software-Crowdsourcing Projects," *IEEE Softw.*, vol. 34, no. 2, pp. 37–43, Mar. 2017.

[19] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller, "TurKit: Human Computation Algorithms on Mechanical Turk," in *Symposium on User Interface Software and Technology (UIST)*, 2010, pp. 57–66.

[20] A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut, "CrowdForge: Crowdsourcing Complex Work," in *Symposium on User Interface Software and Technology (UIST)*, 2011, pp. 43–52.

[21] K. Mao, L. Capra, M. Harman, and Y. Jia, "A survey of the use of crowdsourcing in software engineering," *J. Syst. Softw.*, vol. 126, pp. 57–84, Apr. 2017.

[22] N. Leicht, I. Blohm, and J. M. Leimeister, "Leveraging the Power of the Crowd for Software Testing," *IEEE Softw.*, vol. 34, no. 2, pp. 62–69, Mar. 2017.

[23] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What Would Other Programmers Do: Suggesting Solutions to Error Messages," in *Conference on Human Factors in Computing Systems (CHI)*, 2010, pp. 1019–1028.

[24] D. Mujumdar, M. Kallenbach, B. Liu, and B. Hartmann, "Crowdsourcing Suggestions to Programming Problems for Dynamic Web Development Languages," in *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, 2011, pp. 1525–1530.

[25] C. Watson, F. W. B. Li, and J. L. Godwin, "BlueFix: Using Crowd-sourced Feedback to Support Programming Students in Error Diagnosis and Repair," in *International Conference on Software Engineering (ICSE)*, 2012, pp. 228–239.

[26] F. Pastore, L. Mariani, and G. Fraser, "CrowdOracles: Can the Crowd Solve the Oracle Problem?," in *International Conference on Software Engineering (ICSE)*, 2013, pp. 342–351.

[27] W. Li, S. A. Seshia, and S. Jha, "CrowdMine: Towards Crowdsourced Human-assisted Verification," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1254–1255.

[28] K. Moffitt, J. Ostwald, R. Watro, and E. Church, "Making Hard Fun in Crowdsourced Model Checking: Balancing Crowd Engagement and Efficiency to Maximize Output in Proof by Games," in *Proceedings of the Second International Workshop on CrowdSourcing in Software Engineering*, 2015, pp. 30–31.

[29] W. Dietl *et al.*, "Verification Games: Making Verification Fun," in *Workshop on Formal Techniques for Java-like Programs*, 2012, pp. 42–49.

[30] E. C. Groen *et al.*, "The Crowd in Requirements Engineering: The Landscape and Challenges," *IEEE Softw.*, vol. 34, no. 2, pp. 44–52, Mar. 2017.

[31] P. K. Chilana, A. J. Ko, J. O. Wobbrock, and T. Grossman, "A Multi-site Field Study of Crowdsourced Contextual Help: Usage and Perspectives of End Users and Software Teams," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2013, pp. 217–226.

[32] W. Maalej, H.-J. Happel, and A. Rashid, "When Users Become Collaborators: Towards Continuous and Context-aware User Input," in *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2009, pp. 981–990.

[33] E. Dolstra, R. Vliegendhart, and J. Pouwelse, "Crowdsourcing GUI Tests," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 332–341.

[34] Y. Chen, S. Oney, and W. S. Lasecki, "Towards Providing On-Demand Expert Support for Software Developers," in *Conference on Human Factors in Computing Systems (CHI)*, 2016, pp. 3192–3203.

[35] W. S. Lasecki, J. Kim, N. Rafter, O. Sen, J. P. Bigham, and M. S. Bernstein, "Apparition: Crowdsourced User Interfaces That Come to Life As You Sketch Them," in *Conference on Human Factors in Computing Systems (CHI)*, 2015, pp. 1925–1934.

[36] M. Nebeling, S. Leone, and M. C. Norrie, "Crowdsourced Web Engineering and Design," in *Web Engineering: 12th International Conference, ICWE 2012, Berlin, Germany, July 23-27, 2012. Proceedings*, M. Brambilla, T. Tokuda, and R. Tolksdorf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 31–45.

[37] M. Goldman, G. Little, and R. C. Miller, "Real-time Collaborative Coding in a Web IDE," in *Symposium on User Interface Software and Technology (UIST)*, 2011, pp. 155–164.

[38] M. Goldman, "Software Development with Real-time Collaborative Editing," Massachusetts Institute of Technology, Cambridge, MA, USA, 2012.

[39] M. Kersten and G. C. Murphy, "Using Task Context to Improve Programmer Productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2006, pp. 1–11.

[40] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić, "The Emergent Structure of Development Tasks," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005, pp. 33–48.

[41] C. Parnin and C. Gorg, "Building Usage Contexts During Program Comprehension," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, 2006, pp. 13–22.

[42] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2008, pp. 1323–1332.

[43] A. Doan, R. Ramakrishnan, and A. Y. Halevy, "Crowdsourcing Systems on the World-Wide Web," *Commun. ACM*, vol. 54, no. 4, pp. 86–96, Apr. 2011.

[44] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, 2004, p. 10.

[45] R. E. Kraut *et al.*, *Building Successful Online Communities: Evidence-Based Social Design*. The MIT Press, 2012.

[46] A. Kittur *et al.*, "The Future of Crowd Work," in *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, 2013, pp. 1301–1318.

[47] D. Retelny *et al.*, "Expert Crowdsourcing with Flash Teams," in *Symposium on User Interface Software and Technology (UIST)*,

2014, pp. 75–85.
[48] K.-J. Stol and B. Fitzgerald, "Two's Company, Three's a Crowd: A Case Study of Crowdsourcing Software Development," in *International Conference on Software Engineering (ICSE)*, 2014, pp. 187–198.
[49] A. Kulkarni, P. Gutheim, P. Narula, D. Rolnitzky, T. S. Parikh, and B. Hartmann, "MobileWorks: Designing for Quality in a Managed Crowdsourcing Architecture," *Internet Comput.*, vol. 16, no. 5, pp. 28–35, 2012.

Thomas D. LaToza received degrees in psychology and computer science at the University of Illinois, Urbana-Champaign in 2004 and a Ph.D. in software engineering at Carnegie Mellon University in 2012. He is an assistant professor in the Department of Computer Science at George Mason University. He works at the intersection of software engineering and human-computer interaction, investigating how humans interact with code and designing new ways to build software. He currently serves as guest editor of the IEEE Software Theme Issue on Crowdsourcing for Software Engineering, serves as co-chair of the Fourth International Workshop on Crowdsourcing in Software Engineering, and served as co-chair of the Seventh Workshop on the Evaluation and Usability of Programming Languages and Tools. His work is funded in part through a $1.4M grant from the National Science Foundation on Crowd Programming.

Arturo Di Lecce received a B.S. degree in automation engineering and a M.S. degree in computer engineering from Politecnico di Milano, Italy. After several years as a freelance web developer and one year as visiting research scholar at University of California Irvine, he currently works as full-stack software engineer at Cuebiq Srl, Italy.

Fabio Ricci received B.S and M.S degree in computer engineering, in 2011 and 2016, from Politecnico di Milano, Italy. He worked for several months at Global Energy Network Institute in Sand Diego and one year as a visiting research scholar at the University of California Irvine. He currently works at Bosch Rexroth as a full stack developer for web HMI.

W. Ben Towne earned a B.S. in Electrical and Computer Engineering and a B.A. in Community Development from Lafayette College in 2009 and an MS in Computation, Organizations, & Society from Carnegie Mellon University's School of Computer Science in 2012, finishing a PhD in Societal Computing from Carnegie Mellon as this article undergoes review. He has worked for MIT Lincoln Labs, IBM Watson Research Center, and Ashoka. He has published several peer-reviewed papers in various journals and conferences. He is especially interested in platforms supporting large-scale collaboration, especially for complex problems. He is a member of Sigma Xi, Tau Beta Pi, the Engineers' Society of Western PA, and the American Society for Quality. He is a member of IEEE, IEEE Eta Kappa Nu, and the IEEE Society on Social Implications of Technology (SSIT).

André van der Hoek received joint B.S. and M.S. degrees in business-oriented computer science from the Erasmus University Rotterdam, The Netherlands, and a Ph.D. degree in computer science from the University of Colorado at Boulder. He serves as chair of the Department of Informatics at the University of California, Irvine. He heads the Software Design and Collaboration Laboratory, which focuses on understanding and advancing the roles of design, collaboration, and education in software development. He has served on numerous international program committees, is a member of the editorial board of ACM Transactions on Software Engineering and Methodology, and was program chair of the 2010 ACM SIGSOFT International Symposium on the Foundations of Software Engineering and program co-chair of the 2014 International Conference on Software Engineering. He was recognized in 2013 as an ACM Distinguished Scientist, and in 2009 he was a recipient of the Premier Award for Excellence in Engineering Education Courseware. He is a member of the IEEE.