Large-Scale Microtask Programming

Thesis proposal

Emad Aghayi
Department of Computer Science
George Mason University
Fairfax, VA 22030
eaghayi@gmu.edu

Committee
Thomas D. LaToza, George Mason University (Chair)
Paul Ammann, George Mason University
Kevin Moran, George Mason University
Vivian Motti, George Mason University
Christian Bird, Microsoft Research

Abstract

Microtask programming decontextualizes work into self-contained microtasks, reducing the context necessary to onboard onto a software project and the barriers necessary to contribute. At the same time, it may reduce the time to market for software by increasing parallelism. A number of prior systems have explored approaches for microtasking programming work, devising workflows through which large and complex programming tasks are decomposed into self-contained microtasks which may be completed in minutes.

However, existing approaches have important limitations, impeding their ability to scale to real-world programs and crowds: (1) Current approaches offer limited support for parallelism and conflict management. (2) Current approaches ensure quality by assigning feedback and management responsibility to the client or developer requesting the work. (3) Current approaches remain highly limited in the programming activities and software domains that they support.

To make microtask programming more scalable, I have designed and evaluated several new techniques. First, I designed and implemented techniques to address parallelism and conflict management limitations by re-envisioning the microtask scope and adapting behavior-driven development to offer contributors new types of feedback. I implemented my approach through creating a web-based programming environment, CrowdMicroservices. Second, in contrast to existing approaches which rely on feedback from either a client or manager which are only available after a microtask is completed, I developed new techniques that enable developers to receive initial feedback within the microtask itself. Third, initial evidences show that my microtask programming approach can support most of the development activities in backend and frontend.

To better explore microtask programming's scalability to large crowds of contributors, I will conduct the first microtask programming hackathon. And I will explore approaches to programming user interfaces by designing new workflows, including programming environments embodying these workflows. To evaluate my workflows and programming environments, I will conduct user studies, in which crowd workers will implement frontend applications.

Contents

1 Introduction **1**

2 Related work **3**
- 2.1 Decomposition, context, and scope of microtasks . . . 3
- 2.2 Parallelism and conflicts in microtask programming . . . 4
- 2.3 Achieving quality in microtask programming . . . 5
- 2.4 Fast onboarding of microtask developers . . . 6
- 2.5 Behavior Driven Development (BDD) . . . 6

3 Behavior-driven microtask programming **7**
- 3.1 Behavior-driven microtask workflow . . . 8
  - 3.1.1 Implement Function Behavior microtask . . . 8
  - 3.1.2 Review microtask . . . 11
  - 3.1.3 Interacting with microtasks . . . 12
  - 3.1.4 Assembling microservices . . . 12
- 3.2 CrowdMicroservices . . . 13
- 3.3 Studies of microtask programming . . . 14
  - 3.3.1 Study 1: Can microservices be implemented through a behavior-driven development microtask workflow? . . . 14
  - 3.3.2 Study 2: How Does Microtasking Impact Programming? . . . 15
  - 3.3.3 Study 3: Can Microtask Programming Work in Industry? . . . 18

4 Proposed work **19**
- 4.1 Remote microtask programming hackathon . . . 19
- 4.2 Co-pilot microfrontend programming . . . 22
- 4.3 Microfrontend programming . . . 22
  - 4.3.1 CrowdGUI workflow . . . 23

5 Research plan **26**

6 Acknowledgments **27**

1 Introduction

Microtask programming offers developers new ways to contribute to software projects, enabling workers outside a traditional software development team to take part in building software [34]. In microtask programming a workflow is used to decompose tasks into a sequence of microtasks. By reducing the necessary task context for newcomers to learn, microtask programming reduces the cost of onboarding and enables developers to contribute more quickly. Small contributions also open the possibility of increasing parallelism by reducing time to market. As many hands make light work, decomposing software development tasks into microtasks may enable work to be completed in less time by parallelizing work across many developers.

A variety of systems have explored the promise of applying microtask crowdsourcing to programming [8, 14, 27, 30, 31, 28, 32, 29, 41, 50, 38]. For example, in Apparition [27], a client developer narrates a description for a user interface in natural language, and crowd workers translate this description into user interface elements, visual styles, and behavior. In CodeOn [8], a client developer narrates a request for help from their IDE, and crowd workers use this request and relevant context to author answers and code. In CrowdCode[32, 31], programming work is completed through a series of specialized microtasks. Workers write test cases, implement tests, write code, reuse existing functions, and debug.

However, existing microtask programming approaches have limited scalability. First, current approaches offer limited support for parallelism and conflict management. Due to the increased parallelism assumed and greater potential for conflicts, microtasking approaches often apply a pessimistic locking discipline, where microtasks are scoped to an individual artifact and further work on these artifacts is locked while they are in progress. For example, in Apparition workers acquire write-locks to avoid conflicts. Similarly, in CrowdCode contributions are made on an individual function or test, which is locked while a microtask is in progress. However, conflicts may still occur when decisions made in separate microtasks must be coordinated and are not. For example, conflicts may occur when microtasks are separately required to translate function descriptions into an implementation and tests and differing interpretations of a function description are made. Second, current microtask programming approaches ensure quality by assigning the responsibility of feedback and management to the client or the developers requesting the work. Systems where the requestor is less directly involved in work and microtasks are automatically generated may have crowd members review and offer feedback after contributions are made. However, this approach is limited, as contributors who do not receive the traditional feedback offered in programming environments, such as syntax errors, missing references, and unit test failures, may submit work which contains these issues, which other contributors must then address later at higher cost. Third, existing approaches remain highly limited in the programming activities and software domains that they support. A long-term vision of microtask programming is to enable the software to be made entirely through microtasks [29]. However, current approaches only enable building small prototypes [35] or small artifacts [31]. In addition, the evidence available on how these approaches impact programming is limited, focusing only on their feasibility in completing programming tasks. No prior work has directly compared programming work done through a microtasking workflow to traditional programming.

To address these limitations, I propose to create new microtask programming approaches as well as conduct studies examining microtask programming. The central thesis of this approach is:

A microtask programming workflow enables a large crowd of developers to work efficiently in parallel to implement software.

I will address all three of the barriers to achieving greater scalability, improving support for parallelism and conflict management, improving quality assurance, and supporting a broader range of programming activities. First, I designed new techniques to address the parallelism and conflict management limitation by re-envisioning the scope of the microtask and adapting behavior driven development in microtask programming workflow. By decomposing large tasks into smaller tasks around behaviors, work can be assigned to several workers and completed more quickly in parallel. Due to the increased parallelism assumed and greater potential for conflicts, in contradiction to other microtasking approaches apply a pessimistic locking discipline to a big scope of programming work, my approach decrease the scope of microtasks and lock smaller part of programming work. My approach limits conflicts by decomposing work around Small behaviors, removing the potential for conflicts caused by tests and code written by different developers. The smaller scope of microtask leads to lower conflicts and higher degree of parallelism. In my approach each microtask encompasses the work to test and implement a behavior, a specific identifiable use case of a function. Developers work on a behavior end-to-end, identifying it from a high-level description of a function, writing a test to exercise it, implementing it in code, and debugging any issues that emerge. I found that increasing parallelism and resolve conflict it is feasible to envision the scope of microtasks to implement, test, or debug a small behaviors of a functions [2]. However, to date, I have only studied this with crowds of 9 developers. To examine microtask programming at scale, I will conduct a virtual hackathon with around 100 developers. Based on my findings, I will develop new techniques to scale microtask programming.

Second, I designed new techniques to better achieve quality in microtask programming. Existing approaches delay feedback beyond the point at which a developer completes a microtask, relying on a client, manager (e.g., [27, 14]) or later crowd contibutor (e.g., [32, 31]) to later offer feedback. Without this immediate feedback, developers write code which references fields and functions that do not exist, making poor implementation choices, or making implementation choices which conflict with other choices [31, 32]. As a result, time and effort are wasted, as further work is required to fix these issues, and the potential for undetected defects increases. I designed a new workflow which enables developers to receive feedback within the microtask itself, through syntax errors, running unit tests, and debugging code. I found that, with these new techniques, developers writing code with microtask programming could create code that was at a similar quality level as traditional programming.

Third, I designed new techniques to enable implementing microservices through microtask programming. A client, acting on behalf of a software team, describes a microservice as a set of endpoints with paths, requests, and responses. A crowd then implements the endpoints, identifying individual endpoint behaviors which they test, implement, and debug, creating new functions and interacting with persistence APIs as needed. In three separate studies, I found evidence that this approach can be used to build microservices, both by traditional crowd workers and when used by employees inside a software company. I will extend this work, investigating how to enable frontends to be created through microtasking without requiring a dedicated developer to oversee this work, as required in existing approaches.

To date, in my work, I have built a new approach to microtask programming, implemented in

the CrowdMicroservices [1] programming environment, and conducted three separate studies of microtask programming, evaluating the feasibility of my new techniques [2], comparing microtask programming to traditional programming, and evaluating the use of microtask programming in industry [49]. In my proposed work, I will conduct several additional studies as well as design a new approach for microtasking frontend GUI programming. This work has several contributions to crowdsourcing and software engineering:

1. A novel workflow for microtask programming, which applies behavior-driven development to offer immediate feedback on programming contributions within microtasks.
2. CrowdMicroservices, the first programming environment for implementing microservices through microtasks
3. Evidence that behavior-driven microtasks can be quickly completed by developers and used to successfully implement a microservice.
4. Evidence that microtask programming can be used to increase the fluidity of employee team assignments when applied within a company.
5. Evidence that, compared to traditional programming, applying microtasking can reduce onboarding time and increase project velocity while modestly decreasing individual developer productivity and having no impact on software quality.

2 Related work

My work builds on a broad body of work in crowdsourcing, particularly prior approaches to crowdsourcing software development. This work has investigated several key aspects of microtask workflow design, which I focus on below: decomposition and context, parallelism and conflicts, enabling fast onboarding, and achieving quality.

2.1 Decomposition, context, and scope of microtasks

A key challenge in microtasking in all domains is the manner in which work is decomposed into the tasks or microtasks which are completed by crowd workers. The choice of decomposition determines the workflow of the approach, encompassing the individual steps, the context and information offered in each step, and the types of contributions which can be made [46, 25, 5, 18, 24, 22].

Within approaches for crowdsourcing software engineering work, several points in the design space have been explored [38]. Depending on the choice of microtask boundaries, contributions may be easier or harder, may vary in quality, and may impose differing levels of overhead. Approaches to crowdsourcing programming explore novel workflows designed specifically for accomplishing specific software development tasks.

Techniques for decomposing programming work into fine-grained microtasks may be either manual or automatic. Manual approaches rely on a developer or client to author each microtask[49, 27, 8]. For example, in CodeOn [8], a developer working in a project requests small microtasks

[1] https://youtu.be/mIn2EOqsDYw

for others to complete. In Apparition [27], a similar workflow is used to build user interface prototypes. Requestors describe desired user interface elements and their user interaction through natural language todo items. Workers then view the output of the complete program, select a microtask from the todolist, and implement the requested functionality for a UI element. While working on the microtask, workers interact with an individual element, but otherwise have a global view of the entire codebase. In Crowd Design [41], work for building a web application is broken down by component. Crowd members work individually in an isolated environment on each component and complete design and testing tasks. Manual decomposition of work limits the scalability of a crowdsourcing system. As microtasks are generated manually by a single individual with a global view of the project, scalability is limited.

Other systems automatically generate microtasks from work finished previously by the crowd, reducing the work imposed on the client to create and manage microtasks. In CrowdCode, programming work is done through a series of specialized microtasks in which participants write test cases, implement tests, write code, look for existing functions to reuse, and debug [32, 31]. Other work has automatically generated microtasks through puzzle games, formulating complex tasks such as testing and verification as puzzles which can be completed with little or no programming knowledge [7, 50, 36, 6].

Task interdependence relates to the degree to which a task requires organizational units to affect the activities and work outcomes of other units [16, 3]. Increasing task interdependence forces team members to integrate their contributions with others. When task interdependency is low, the contributions of each unit or team member is additive. There exists a relation between the degree of task interdependency and the productivity of units or team members [54]. Higher degrees of task interdependencies require more information exchange, more task clarification in task assignment, creating more shared mental models, and more effort to integrate tasks [54]. Productivity decreases because of time spent establishing shared mental models and resolving conflicts among contributors. Studies have shown productivity dramatically declines when a team member needs the output of other units or team members as an input to their task [3]. In designing microtask workflows, it can thus be beneficial to decrease the degree of the interdependency of software development tasks by creating tasks with scopes that developers can work on with less interdependency. In my workflow, microtasks are largely self-contained and independent of the other microtasks, which provided a low level of dependency. Moreover, implementing tests and implementing behaviors are completed by the same crowd worker, which helps to reduce time spent resolving conflicts.

The context of a task in microtask programming systems may vary from several statements in a function to the whole codebase. For instance, in Apparition, each contributor can see the entire codebase to complete a task. In my approach, the context is a function's description, implementation, and unit tests and descriptions of external APIs and data types.

2.2 Parallelism and conflicts in microtask programming

By decomposing large tasks into smaller tasks, work can be assigned to several workers and completed more quickly in parallel. For easily parallelizable software engineering tasks, like writing a test, this paradigm has achieved widespread adoption in commercial platforms. Crowdsourced

testing platforms such as UserTesting [2], TryMyUI [3], and uTest[4] enable software projects to crowdsource functional and usability testing work by utilizing the crowd of tens or hundreds of thousands of contributors on these platforms.

Microtasking approaches for programming envision reducing the necessary time to complete programming tasks through parallelism. A key challenge occurs with conflicts, where two overlapping changes to an artifact are submitted at the same time. For example, in traditional software development, each contributor may edit the same artifacts at the same time, resulting in a merge conflict when conflicting changes are committed. This is an example of an optimistic locking discipline, where any artifact may be edited by anyone at any time. Due to the increased parallelism assumed and greater potential for conflicts, microtasking approaches often apply a pessimistic locking discipline, where microtasks are scoped to an individual artifact and further work on these artifacts is locked while they are in progress. For example, in Apparition [27] workers acquire write-locks to avoid conflicts. Similarly, in CrowdCode [31] contributions are made on an individual function or test, which is locked while a microtask is in progress. However, conflicts may still occur when decisions made in separate microtasks must be coordinated and are not [58, 28]. For example, conflicts may occur when microtasks are separately required to translate function descriptions into an implementation and tests and differing interpretations of a function description are made [31]. In my research, I explore an approach for limiting conflicts by decomposing work around behaviors, removing the potential for conflicts caused by tests and code written by different developers.

2.3 Achieving quality in microtask programming

Crowdsourcing approaches to programming have explored a variety of approaches for ensuring the quality of the resulting software artifacts. The quality is ultimately determined by the quality of individual contributions. There are many causes of low-quality contributions, including workers who do not have sufficient knowledge, who put forth little effort, or who are malicious [23]. A study of the TopCoder[5] crowdsourcing platform revealed six factors related to project quality, including the average quality score on the platform, the number of contemporary projects, the length of documents, the number of registered developers, the maximum rating of submitted developers, and the design score[37]. In TopCoder, senior contributors assist in managing the process of creating and administering each task and ensuring quality work is done [53]. Studies of software crowdsourcing companies identified 10 methods used for addressing quality, including ranking and ratings, reporting spam, reporting unfair treatment, task pre-approval, and skill filtering [48].

In microtask systems, crowd members are often assumed to be minimally invested in the platform or community. Crowd programming systems have addressed this problem by assigning the responsibility of feedback and management to the client or the developers requesting the work [14, 8, 27, 35]. Systems where the requestor is less directly involved in work and microtasks are automatically generated may have crowd members review and offer feedback after contributions are made [31]. However, this approach is limited, as contributors who do not receive the traditional feedback offered in programming environments, such as syntax errors, missing references,

[2]https://www.usertesting.com

[3]https://www.trymyui.com

[4]https://www.utest.com

[5]https://www.topcoder.com/

and unit test failures, may submit work which contains these issues, which other contributors must then address later at higher cost [32].

In my approach, I investigate approaches for offering immediate feedback within a microtask. Developers receive feedback by executing unit tests, by syntax errors, and by observing the execution of the program through debugging.

2.4 Fast onboarding of microtask developers

Another challenge in using crowd work within a software project is the process of onboarding. A number of studies have documented the joining scripts used and barriers that open source software developers face when onboarding onto a new project. These include installing necessary tools, downloading code from a server, identifying and downloading dependencies, and configuring the build environment [52, 56, 21, 10]. As a result, onboarding onto open source projects can require weeks of time, creating a substantial barrier dissuading casual contributors from joining.

Researchers have explored programming environments which aim to alleviate these barriers. Codepilot [57] reduces the complexity of programming environments for novice programmers by integrating a preconfigured environment for real-time collaborative programming, testing, bug reporting, and version control into a single, simplified system. In Collabode, multiple developers synchronously edit code at the same time, enabling new forms of collaborative programming [15, 14]. Apparition offers an online environment for building UI mockups, offering an integrated environment for authoring, viewing, and collaborating on the visual look and feel and behavior of UI elements [27]. CrowdCode offers an online preconfigured environment for implementing libraries, enabling developers to onboard quickly onto programming tasks [31].

In my approach, I build on these approaches, offering preconfigured environments and tools for fast onboarding specifically designed for implementing microservices.

2.5 Behavior Driven Development (BDD)

In this research, I apply behavior-driven development to microtask programming. BDD focuses on defining fine-grained specifications of a system's behavior in a way that they can be tested [51]. This enables writing executable specifications of a system [17]. An acceptance test in BDD is a specification of the system's behavior that verifies its behavior rather than its state. A survey of literature and current BDD toolkits identified several characteristics of BDD, including ubiquitous language, iterative decomposition, plain text descriptions of user stories and scenario templates, automated acceptance testing with mapping rules, and readable behavior-oriented specification code [17].

There are few studies investigating the impact of applying BDD. One reason may be that the original version of BDD is highly similar to test-driven development. Several proponents believe BDD helps teams to generate and deliver higher quality software quickly [51, 19, 42, 45, 51]. One study of 22 developers analyzed BDD's impact on the software life cycle [13]. The study found that BDD increased the quality of software products by 2% and 5% in relation to TDD and traditional iterative tests, respectively [17].

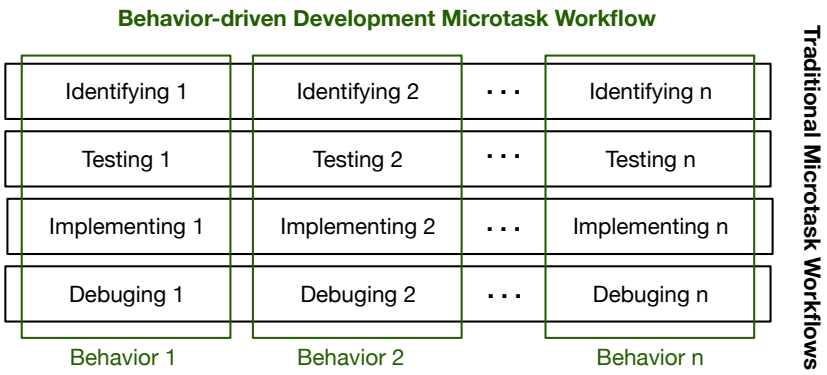

Figure 1: Traditional microtask programming decomposes large programming tasks into separate microtasks in which different types of contributions can be made, such as identifying all of the behaviors which should be tested in a function or writing a test. In behavior-driven microtask programming, work is instead decomposed by behavior, where an individual microtask incorporates work of several types for an individual behavior.

3 Behavior-driven microtask programming

In this section, I describe a behavior-driven approach to microtask programming. The main goal of the workflow is to reduce onboarding time and increase the potential for parallelism. In behavior-driven development, developers first write a unit test for each behavior they will implement, offering a way to verify that their implementation works as intended by running a test. As a workflow for microtasking, behavior-driven development offers a number of potential advantages. As developers work, they receive feedback before submitting, enabling the developer to revise their own work. Rather than requiring separate developers to test, implement, and debug a function through separate microtasks and coordinate this work to ensure consistency, a single contributor focuses on work related to an individual behavior within a function (Fig. 1). Fig. 2 surveys my approach.

In the following sections, I first describe my behavior-driven workflow and its application to implementing a microservice (Section 3.1). I then describe the implementation of my approach in the prototype CrowdMicroservices IDE in Section 3.2. Then I describe three separate studies I conducted. In my first study (published in JSS [2]), I investigated the feasibility of my new techniques. In my second study (in submission to TSE and described in Section 3.3.2), I directly compared microtask programming to traditional programming to investigate its benefits and drawbacks. In my third study, conducted together with collaborators at NTT[6] (published as an industry paper in ESE/FSE 2020 [49] and described in Section 3.3.3), I examined the use of microtask programming in an industrial context. This work won a Distinguished Research Award from the Information Processing Society of Japan. To get feedback and think more deeply about my research, I published and presented a summary of my approach in Graduate Consortium 2020 VL/HCC [1].

[6]https://www.ntt.co.jp/RD/e/index.html

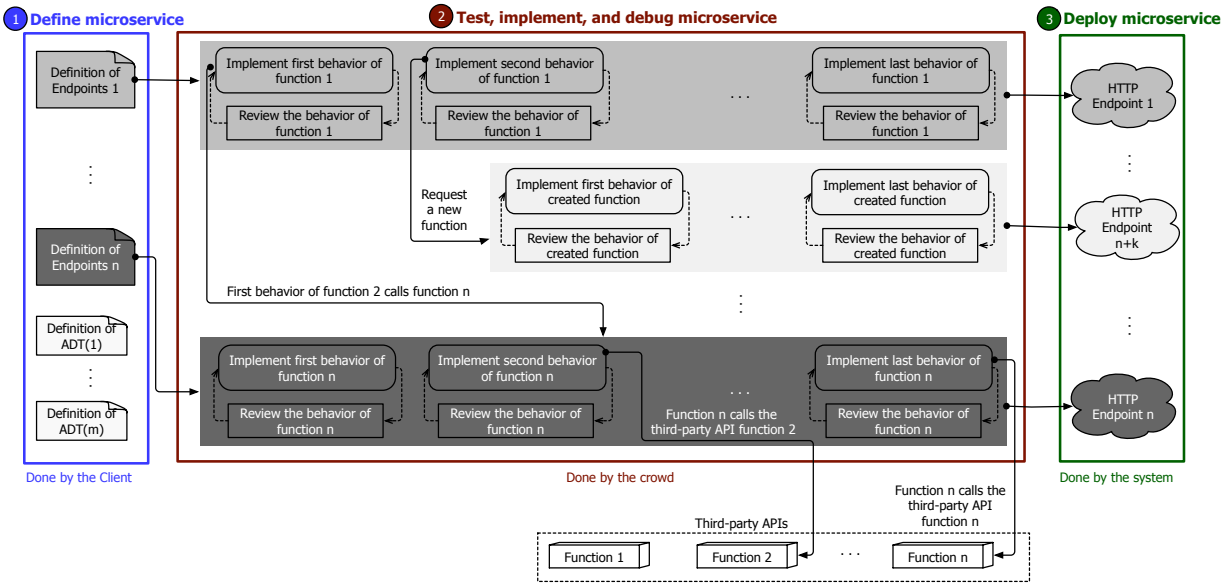

Figure 2: **(1) Define microservices**: The client first writes a `Client-Request` to define a microservice to implement.**(2) Test, implement, and debug microservice**: The system generates microtasks as necessary to implement each endpoint. **(3) Deploy microservice**: After implementation is complete, the client may deploy the microservice.

3.1 Behavior-driven microtask workflow

In my behavior-driven microtask workflow, contributions are made through two microtasks: `Implement Function Behavior` and `Review`.

3.1.1 Implement Function Behavior microtask

Workers perform each step in the `Implement Function Behavior` microtask through the *CrowdMicroservices* environment (Fig.4).

Step 1. Identify a Behavior. Workers first identify a single behavior that is not yet implemented. From the comments above each function body, workers identify behaviors (see (1) in Fig. 4). Workers may then read the function's unit tests and body to determine which of these are not yet implemented. When the function has been completely implemented, and no behaviors remain, the worker may indicate this through a checkbox.

Step 2. Test the Behavior. In the second step, workers author a test, either as a simple input/output pair, specifying inputs and an output for the function under test, or as an assertion-based test (see (2) in Fig. 4). The worker does this using the test editor. Workers may also invoke other functions, including those defined in third-party APIs. The worker may run the test to verify that it fails with the current implementation. If workers find that the test passes, indicating the behavior they identified in Step 1 is already implemented, they may return to Step 1 to identify a new behavior.

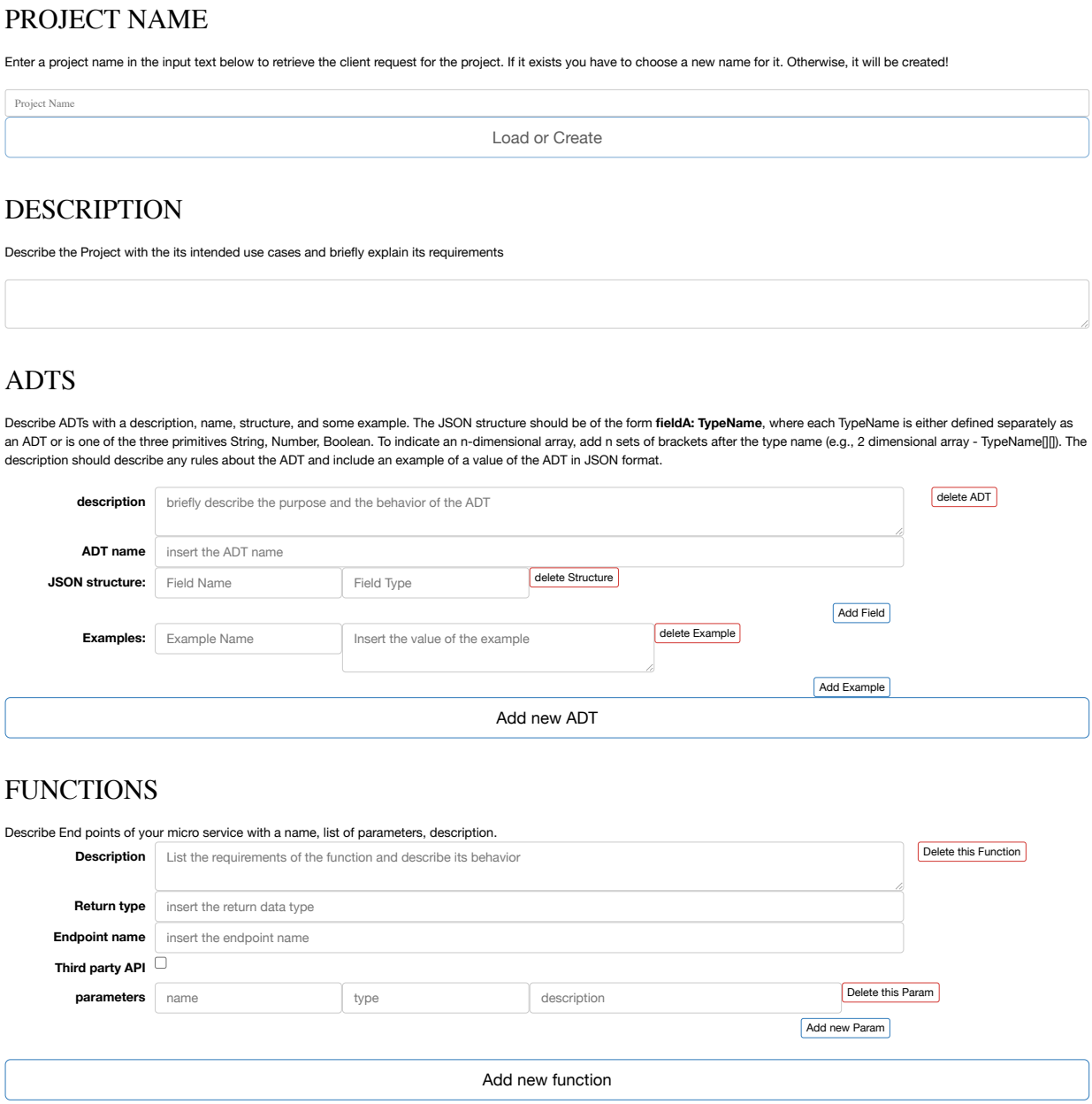

Figure 3: In the `Client-Request`, a client defines a microservice they would like created.

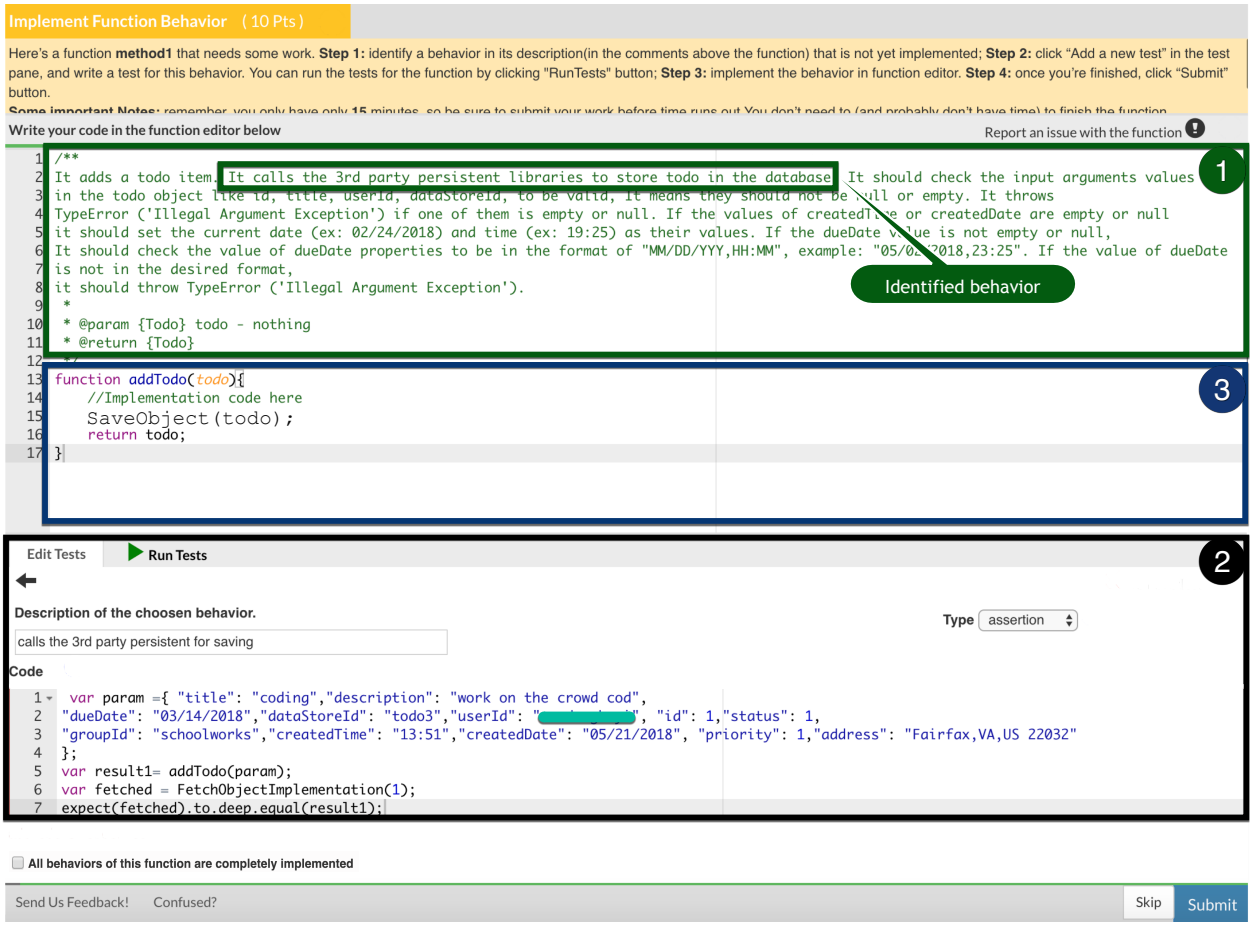

Figure 4: In the `Implement Function Behavior` microtask, workers first **(1) identify a behavior from the description of a function.** They then **(2) write a test in the test editor to verify the behavior** and **(3) edit the code for the function to implement it**. Finally, they (4) test it by running the tests, fixing issues they identify.

Step 3. Implement the Behavior. The worker next implements their behavior using the code editor (see (3) in Fig.4). When the behavior to be implemented is complex, the worker may choose to create a new function, specifying a description, name, parameters, and return type. They may then call the new function in the body of their main function. After the microtask is submitted, this new function will be created, and a microtask generated to begin work on the function. In some cases, the signature of the function that a worker is asked to implement may not match its intended purpose, such as missing a necessary parameter. In these cases, the worker cannot directly fix the issue, as they do not have access to the source code of each call site for the function. Instead, they may report an issue, halting further work on the function. The client can see that this issue has been created, resolve the problem, and begin work again.

Step 4. Debug the Behavior. A worker may test their implementation by running the function's unit tests. When a test fails, workers may debug by using the `Inspect code` feature to view the value of any expression in the currently selected test. Hovering over an expression in the code invokes a popup listing all values held by the expression during execution. In cases where a function that is called from the function under test has not yet been implemented, any tests exercising this functionality will fail. To enable the worker to still test their contribution, a worker may create a stub for any function call. Creating a stub replaces an actual output (e.g., an undefined return value generated by a function that does not yet exist) with an intended output. Using the `stub` editor, the worker can view the current output generated by a function call and edit this value to the intended output. This then automatically generates a stub. Whenever the tests are run, all stubs are applied, intercepting function calls and replacing them with stubbed values.

In pilot studies, I found that requiring workers to only submit microtasks that did not contain errors decreased the productivity of workers dramatically. In those cases, contributors spent the entire 15 minutes, the maximum, before being forced to skip the microtask, losing their work. I therefore enable workers to submit incomplete work and get feedback on incomplete implementations from reviewers.

Step 5. Submit the microtask. Once finished, the worker may submit their work. To ensure that workers do not lock access to an artifact for extended periods of time, each microtask has a maximum time limit of 15 minutes. I derived the 15 minutes from my previous research [31], where the average time for each microtask was less than 5 minutes. This constraint helps the crowd to focus on their microtask to complete or skip it in less than 15 minutes. Workers are informed by the system when time is close to expiring. When the time has expired, the system informs the worker and skips the microtask.

3.1.2 Review microtask

In the `Review` microtask, workers assess contributions submitted by other workers. Workers are given a diff comparing the code they submitted with the previous version as well as the tests of the function. Instead of being asked to make a binary choice to accept or reject the contribution, workers are asked to assign a rating of 1 to 5 stars. If the worker evaluates the work with 1 to 3 stars, the work is marked as needing revision. The worker then describes aspects of the contribution that

they feel need improvement, and a microtask is generated to do this work. If the worker evaluates the submitted contribution with 4 or 5 stars, the contribution is accepted as is. In this case, the assessment of the work is optional, which will be provided back to the crowd worker that made the contribution. When a contributor is notified that his or her contribution is accepted but it did not receive full stars, the workers receive a notification with feedback which they may use for increasing the quality of their future contributions.

3.1.3 Interacting with microtasks

After logging in, workers are first taken to a welcome page which includes a demo video and a tutorial describing basic concepts in the microtask programming environment. After completing the tutorial, workers are taken to a dashboard page, which includes the client's project description, a list of descriptions for each function, and the currently available microtasks. The system automatically assigns workers a random microtask, which the worker can complete and submit or skip. When workers begin a type of microtask which they have not previously worked on, workers are given an additional tutorial explaining the microtask. When participants work and become confused about a design decision to be made or about the environment itself, they may use the global `Question and Answer` feature to post a question, modeled on the question and answer feature in CrowdCode [28]. Posted questions are visible to all workers, who may post answers and view previous questions and answers. Each project defined in the `Client-Request` (Figure.3) has its own `Question and Answer`.

As workers complete microtasks, each contribution is given a rating through a `Review` microtask. Ratings are then summed to generate a score for each worker. This score is visible to the entire crowd that participated on the project on a global `leaderboard`, helping to motivate contributions and higher quality work. As workers probably watched the scores of others, they might be motivated to increase their score and increase their ranking above other workers. Crowd workers could achieve higher scores by submitting more either `Review` or `Implement Function Behaviors`, submitting each `Review` task worth five scores for the reviewer. Submitting an `Implement Function Behavior` gets a score based on the number of rating stars it received by a review. Each star is worth 2 points. Consequently, each `Implementation Function Behavior` based on its quality could collect score 2 to 10 scores.

3.1.4 Assembling microservices

My approach applies the behavior-driven development workflow to implementing microservices. Fig. 2 depicts the steps in my process. The microservice is first described by a client through the `Client-Request` page (Fig 3). Clients define a set of endpoints describing HTTP requests which will be handled by the microservice. Each endpoint is defined as a function, specifying an identifier, parameters, and a description of its behavior. As endpoints may accept complex JSON data structures as input and generate complex JSON data structures as output, clients may also describe a set of abstract data types (ADTs). Each ADT describes a set of fields for a JSON object, assigning each field a type which may be either a primitive or another ADT. In defining endpoints, clients may specify the expected data by giving each parameter and return value a type.

After a client has completed a `Client-Request`, they may then submit this `Client-Request` to generate a new *CrowdMicroservices* project. As shown in Step 2 of Fig. 2, submitting a

`Client-Request` generates an initial set of microtasks, generating an `Implement Function Behavior` microtask for each endpoint function. Workers may then log into the project to begin completing microtasks. As workers complete microtasks, additional microtasks are automatically generated to review contributions, continue work on each function, and implement any new functions requested by crowd workers.

Microservices often depend on external services exposed through third-party APIs. As identifying, downloading, and configuring these dependencies can serve as a barrier to contributing, *CrowdMicroservices* offers a pre-configured environment. As typical microservices often involve persisting data between requests, I chose to offer a simplified API for interacting with a persistence store. Through this API, workers can store, update, and delete JSON objects in a persistence store. Workers may use any of these API functions when working with functions and unit tests in the `Implement Function Behavior` microtask. Any schema-less persistence store may be used as an implementation for this API. In my prototype IDE, a development version used by workers simulates the behavior of a persistence store within the browser and clears the persistence store after every test execution. In the production version used after the microservice is deployed, the API is implemented through a Firebase store.

After the crowd finishes the implementation of a microservice, the client may choose to create and deploy the microservice to a hosting site (step 3 in Fig. 2). Invoking the `Publish` command first creates a new node.js GitHub project which includes each function implemented by the crowd. For endpoint functions, the environment automatically generates an HTTP request handler function. Each endpoint then contains the implementation of the function defined by the crowd. Next, this GitHub project is deployed to a hosting site (Heroku in my prototype implementation). A new project instance is created, and the project is deployed. After this process has completed, the client may begin using the completed microservice by making HTTP requests against the deployed, publicly available microservice. As some projects may require private rather than public deployment, the client may also provide information for a private repository to deploy to through the `Client-Request` (GitHub Info in Fig. 3).

3.2 CrowdMicroservices

I implemented my approach in the prototype *CrowdMicroservices* IDE [7]. *CrowdMicroservices* is a client-server application with three layers: 1) a web client, implemented in AngularJS, which runs in a worker's browser, 2) a back-end, implemented in Express.js, and 3) a persistence store, implemented using the Firebase Real-time Database [8]. *CrowdMicroservices* automatically generates microtasks based on the current state of submitted work. After a `Client-Request` defines endpoints, the system automatically generates a function and microtask to begin work on each. After an `Implement Function Behavior` microtask is submitted, the system automatically creates a `Review` microtask. After a `Review` microtask is submitted, an `Implement Function Behavior` is generated to continue work, if the contributor has not indicated that work is complete. If a review of an `Implement Function Behavior` contribution indicates issues that need to be fixed, a new `Implement Function Behavior` microtask is generated, which includes the issue and an instruction to fix it. After a microtask is generated, it is added to a queue.

[7] https://youtu.be/qQeYOsRaxHc

[8] https://firebase.google.com

When a worker fetches a microtask, the system automatically assigns the worker the next microtask and removes it from the queue.

I apply my approach to implementing microservices. Web application back-ends are often decomposed into numerous microservices, offering a natural boundary for crowdsourcing a module that is part of a larger system. In my approach, a client, for example a software development team, may choose to crowdsource the creation of an individual microservice. In situations where teams lack sufficient developer resources to complete work sufficiently quickly, a development team might choose to delegate this work to a crowd. A client, acting on behalf of the software development team, may define the desired behavior of the microservice by defining a set of endpoints.

3.3 Studies of microtask programming

To investigate the feasibility and impact of my new techniques for microtask programming, I conducted three studies.

3.3.1 Study 1: Can microservices be implemented through a behavior-driven development microtask workflow?

To investigate the feasibility of applying behavior-driven microtask programming to implementing microservices, I conducted a user study in which a crowd of workers built a small microservice. Since there are no prior systems for implementing microservices through microtasks, I focused on evaluating the feasibility of the technique rather than comparing it against alternative approaches. A key goal of microtask programming is to enable short contributions by transient contributors. Therefore, I evaluated how long it takes for a new crowd worker to onboard and make a contribution. Specifically, I investigated the 3 bellow questions.

I recruited 9 participants to build a small microservice for a *ToDo* application and then analyzed their environment interactions and the resulting code they created. At the beginning of the study, participants logged in to *CrowdMicroservices* and worked through tutorial content, watching a tutorial video, reading the welcome page, and reading a series of 6 tutorials on using the individual microtasks. Participants then began work by fetching a microtask.

I gathered data from several sources. Before beginning the study, participants completed a short demographics survey. During the study session, all participant interactions with *CrowdMicroservices* were logged with a timestamp and participant id. This included each microtask generated, submitted, and skipped as well as each change to a function or test. At the end of the first session, participants completed a survey on their experiences with *CrowdMicroservices*, focusing specifically on their experience with the behavior-driven development workflow. At the end of the study, five participants participated in a short 15 minute semi-structured interview. The open-ended questions focused on onboarding challenges, the granularity of microtasks, the ability to choose a task, motivation working using microtask programming, and interactions between crowd workers.

RQ1: Can crowd workers contribute through a behavior-driven microtask workflow?: To investigate the ability of participants to use the behavior-driven microtask workflow, I examined the log data to determine how many microtasks participants were able to successfully complete during the two sessions as well as the functions and tests they created. Overall, participants successfully submitted 350 microtasks and implemented 13 functions, one of which was defined by the crowd.

Participants created a test suite of 36 unit tests, writing an average of 3 unit tests per function. Participants wrote 216 lines of code, approximately 16 lines per function. Participants wrote 397 line of code in their test suite.

RQ2: How quickly can workers onboard? How long does it take to contribute? A wide range of volunteers, paid developers, or contract developers might participate in microtask programming projects. My approach could be used to support open source projects. Studies have identified several motives for developers to join open-source projects, including a desire to learn and develop new skills, share knowledge and skills, and participate in a new form of cooperation[13]. However, developers face barriers to joining these projects, including installing necessary tools, downloading code from a server, identifying and downloading dependencies, and configuring the build environment [52, 56, 21, 10]. Consequently, onboarding can require weeks, discouraging casual contributors from joining. *CrowdMicroservices* may reduce these barriers by providing a preconfigured environment that helps developers onboard quickly. In my study, I found that developers could submit their first microtask in less than 24 minutes. The median completion time for each microtask was approximately 4 minutes for `Implement Function Behavior` and 3 minutes for `Review` microtasks . These completion times are similar to the approximately 5 minute median completion times of prior microtasks with a more limited range of potential contributions [31], despite requiring developers to test, implement, and debug their behavior.

RQ3: Can a microservice be implemented through microtasks?: To assess the feasibility of using a behavior-driven microtask workflow to implement microservices, I investigated the success of the crowd in building an implementation consistent with the described behavior in the `Client-Request`. I first constructed a unit test suite, generating a set of 34 unit tests (written and visible only to the experimenters, not participants). Overall, unit tests for 79% (27) of the behaviors passed, and unit tests for 7 of the behaviors failed.

To further assess the implementation of the microservice built by the crowd, I used the final code written by participants to build a functioning *ToDo* application. I used the deploy feature in *CrowdMicroservices* to deploy and host the microservice. I then implemented a *ToDo* application front-end as a React application, using the deployed microservice as the back-end. I found that, apart from the defects I described above, the *ToDo* application worked correctly.

Takeaways from Study 1: In this study, I explored a novel workflow for organizing microtask work and offered the first approach capable of microtasking the creation of web microservices. I found that, using this approach, workers were able to successfully submit 350 microtasks and implement 13 functions, quickly onboard and submit their first microtask in less than 24 minutes, contribute new behaviors in less than 5 minutes time, and together implement a functioning microservice back-end containing only 4 defects. Participants were able to receive feedback on their contributions as they worked by running their code against their tests and debugging their implementation to address issues.

3.3.2 Study 2: How Does Microtasking Impact Programming?

To investigate the potential benefits and drawbacks of microtask programming, I conducted a direct head-to-head comparison between microtask programming and traditional software development, investigating the impact of microtask programming on onboarding, velocity, quality of code, and

individual developer productivity.

To investigate these factors, I conducted a controlled experiment. I recruited 28 participants and randomly assigned them to a control or experimental condition. All participants worked in a 4 hour session to implement a small microservice for an online shopping application in JavaScript. In the experimental condition, participants worked on programming microtasks. Experimental participants were organized into two sessions. In each session, 7 participants worked simultaneously as a crowd. In the control condition, 14 participants each worked individually. The study consisted of three parts: a training task, work on the coding task, and a post-task survey on their experiences. I recorded participants' work using screencasts to get a full picture of their activity.

RQ1: How does microtasking impact onboarding? To investigate how a microtask style of work impacts the time necessary to onboard onto a new project, I measured the time developers spent onboarding. I defined onboarding as the orientation time in which a new developer adjusts to and becomes productive within a project [4].

Overall, microtask programming significantly reduced onboarding time, measured both from the beginning of the session to the writing the first line of code and to completing the first task. The time for microtask programming participants to finish the first line of code was significantly less, reducing onboarding time by a factor of 1.3 to 39 minutes compared to 52 minutes for control participants ($p = 0.002$). Microtask programming participants completed their first task significantly faster in 44 minutes compared to 164 minutes for control participants, reducing time by a factor of 3.7 ($p = 0.00001$).

Open source projects have a number of substantial barriers that discourage developers from joining and incur high costs for those who participate. These high costs can prevent developers from ever joining a project. Even for a modest project of just over 2000 lines of code, developers spent 164 minutes before completing their first issue. Despite incurring new costs due to the unfamiliar environment and workflow, microtask programming substantially reduced these barriers, measured both in time to initially complete the first line of code and the first task. For larger projects with more to learn or for developers already familiar with microtask programming, the differences may be even larger. This suggests the potential of microtasking for expanding the pool of contributors available to open source projects.

RQ2: How does parallelism in microtask programming impact velocity? The project velocity is the rate of progress of a software development team. By decomposing larger programming tasks (e.g., implement a feature) into parallelizable microtasks, microtask programming is envisioned to increase the velocity of programming work within a software project [33].

I measured the work completed through two complimentary measures: lines of code and the number of correctly implemented behaviors, as measured by executing the *External Test Suite*.

Increasing the number of participants per session from 1 in the control condition to 7 in the microtask programming condition increased the amount of work completed per session, as measured both by the number of behaviors implemented and the lines of code written in each session. The number of behaviors successfully implemented increased by a factor of 5.7, from 13% (5.28 of 39) to 75% (29.5 of 39). The mean number of lines of code implemented increased by a factor of 9.1, from 115 lines of code (115 in functions and 0.4 in unit tests) to 1050 (275 in functions and 775 in unit tests).

By adding additional contributors to a project, the project velocity might be assumed to increase. However, achieving this in practice is challenging, as coordinating additional contributors may incur new costs. I found that microtask programming was surprisingly successful in minimiz-

ing these costs, where increasing the number of project contributors substantially increased project velocity.

RQ3: How does microtasking impact code quality? The code quality of an implementation encompasses how maintainable it is and the ease with which other developers may read or make changes to it. This includes following conventions such as meaningful variable names, appropriate code structures, and appropriate formatting [39].

To assess the quality of the implementation created in each session, a panel of four was assembled. Each of the 16 session's final implementation was evaluated by four panelists separately. Panelists evaluated 16 codebases for their clarity, simplicity, and consistency. Finally, I created an overall quality score by averaging the 3 metrics.

Overall, I found that panelists rated code created through microtask programming as higher in quality, with a quality score of 3.7 (SD = 0.12) vs. 3.3 (SD = 0.33) for control participants. However, this difference was not significant (p = 0.09). For each quality metric, microtask programming codebases were rated by panelists as being higher quality, with 3.8 vs. 3.2 for clarity, 3.6 vs 3.4 for simplicity, and 3.6 vs. 3.5 for consistency.

By reducing the context available to each developer, microtask programming might be expected to reduce quality. However, in domains outside programming, prior work has found that microtasking can, increase, rather than decrease, quality [9, 20]. Decomposing tasks into several microtasks was the key to achieving higher quality, as contributors could focus on smaller tasks without interruption. My findings reveal that microtasking did not reduce quality. Microtask Programming impacted the ways in which developers worked, requiring developers to write unit tests and offering a review of their work by others more frequently. In traditional software development, developers may only receive feedback after submitting all of their changes together through a pull request.

RQ4: How does microtasking impact developer productivity? Software productivity is the amount of useful output created per unit of time. Output may be directly created, such as by implementing software logic or writing unit tests, or indirectly created, such as by reviewing the contributions of other developers or answering questions on Q&A forums [47]. Developers are more productive when they create more direct and indirect output in less time.

I measured output through two complimentary measures: lines of code and the number of correctly implemented behaviors, as measured by executing the *External Test Suite*.

In the control condition, individual developer's productivity ranged from 0.3 to 3.5 behaviors per hour, with an average of 1.6 behaviors per hour. In the microtask programming condition, developers had productivity of 1.2 behavior per hour (29.5 behaviors in 205 minutes by seven developers). Developers in the control condition implemented on average between 5 and 58 lines of code per hour, with a mean of 34. 99.6% of the lines written were implementing functions and 0.4% in unit tests. Developers in the microtask programming condition implemented 44 lines of code per hour. 26% of the lines written were implementing functions and 74% in unit tests.

Microtasking reduced the productivity of individual developers, as measured by the behaviors correctly implemented per developer hour. However, it increased the final lines of code written per developer hour, largely by requiring developers to write more tests. At the same time, nearly half of the code written by microtask participants was discarded, as others edited or replaced it. The gamification elements may have helped motivate some participants while demotivating others. These results illustrate the complexity of productivity, where many factors may play an important role in shaping how much output developers create.

Takeaways from Study 2: Compared to traditional programming, microtask programming reduced onboarding time by a factor of 3.7, increased project velocity by a factor of 5.76 by increasing team size by a factor of 7, and decreased individual developer productivity by a factor of 1.3. The quality of code did not significantly differ. The preconfigured but novel programming environment, reduced task size, reduced need to understand the codebase structure and implementation, and preference for easier tasks may have contributed to these differences. Microtask programming participants made judgments about code quality through reviews and wrote significantly more unit tests.

3.3.3 Study 3: Can Microtask Programming Work in Industry?

My early studies were conducted entirely in artificial contexts, using artificial tasks and developers recruited specifically to work in the study. To examine the potential for using microtask programming in industry, I collaborated with a company. We investigated the potential of applying microtask programming in a company setting, reporting on a project undertaken at NTT[9] to build a web application through microtask programming. Japanese IT vendors such as NTT face a serious IT talent shortage. Microtask programming may help to address this by increasing the fluidity of developer assignments within large organizations.

In this study a web application was developed in four weeks. Two dedicated engineers were assigned to work full time on the project, a requirements engineer and a software engineer. In addition, 6 crowd workers, who were primarily assigned to other projects inside the company, were asked to make use of their available slack time to contribute to the project. We collected qualitative results on the outcome of the project, describing the activities that occurred and the results of the project. We then collected data on the productivity of crowd workers within the project. Finally, we examined the project participant's perceptions of the suitability and effectiveness of applying microtask programming in a company setting through collected interview and survey data.

Overall, the project achieved its objectives and a web application for managing finance closing processes was successfully created. Six crowd workers together implemented approximately 2800 LOC. All were able to initially onboard onto the project in less than 2 hours and successfully communicated via an issue tracking system (ITS). Crowd workers reported that they perceived onboarding costs to be reduced and did not experience issues with the reduced face-to-face communication, but did experience challenges with motivation.

In adopting microtask programming, there were four key ways in which development work differed from standard practice: (1) a finer granularity of work, (2) less face to face communication, (3) associating contributions with artifacts, and (4) enabling developers primarily assigned to a different project to contribute in their free time.

There were also differences in the nature of the review tasks. The focus of code reviews is traditionally on conventions, such as coding rules and format. In microtask programming, workers were also responsible to check whether the code satisfied its expected behavior. As a result, review tasks took longer than usual.

The results demonstrate the potential for microtask programming to increase the fluidity of project assignments within an organization. The project met its objectives and successfully com-

[9] https://www.ntt.co.jp/RD/e/index.html

pleted its system testing. Crowd workers assigned to other projects were able to contribute in their slack time and largely felt that the onboarding costs were reduced.

The project also revealed several challenges with applying microtask programming. The crowd workers reported that they found staying motivated to be harder. Crowd workers found the style of work used in microtasking to be unfamiliar and different. Motivation may also have been reduced because microtask programming was not typical practice and workers did not feel they needed to be as productive. Workers did not directly gain anything for their contributions, further reducing their motivation.

Takeaways from Study 3: Our results offer initial evidence for the potential value of microtask programming in increasing the fluidity of team assignments within a company. Crowd contributors to the project were able to onboard and contribute to a new project in less than 2 hours. After onboarding, the crowd workers were together able to successfully implement a small program which contained only a small number of defects.

4 Proposed work

My proposed work will focus on three directions. In the first direction, I will evaluate my large-scale microtask programming workflow and CrowdMicroservices system by running a hackathon. I will collect, analyze, and publish the results of this work. Next, continuing my collaboration with NTT from Study 3 (Section 3.3.3), I will examine a new approach used at NTT for scaling microtask programming to frontend development and analyze data conducted of a user study of this approach. Finally, building on the lessons learned from this system, I will design my own new approach for microtasking frontend web development scalably and efficiently. I expect each of these directions to result in further research papers.

4.1 Remote microtask programming hackathon

My studies have offered initial evidence that microtask programming can be effective in small crowds with a few developers. However, to date, the largest crowd has been only nine developers. Much of the promise of microtasking comes from large crowds, where parallelism might substantially reduce time to market. However, as the size of the crowds grow, the amount of coordination required and potential for conflicts (e.g., duplicate or conflicting work) increases. In my system, conflicts are reduced by having focused discussion organized around design decisions, in the Q&A system. However, it is not clear how effective this approach is in large crowds. And, as crowds grow larger, it is unclear what the impact of additional crowd workers may be on time to market.

Programming hackathon is a continuous event in which people in small teams produce a software prototype in a limited amount of time [44]. Hackathons vary wildly in their purpose and execution but generally have a typical structure and properties [26]. Hackathons often have 3 phases, pre-hackathon, hackathon, post-hackathon. In the pre-hackathons, the hackathon organizers generally specify the goal and scopes of the hackathon. They provide an abstract list of problems. A hackathon often begins with ideation and team building. After many coding hours and not much sleep in traditional programming hackathons, the teams briefly demo their prototype. Most demos

show only a small number of working features. At the end of the hackathon, in the post-hackathon phase, organizers decide to continue or abandon ideas demoed in the hackathon [26]. Compared to a traditional hackathon, the microtask programming hackathon will remove the ideation part of conventional hackathons. In contrast, in the microtask programming hackathon, there will be only one idea and one project. Instead of working on different projects, participants have to work on a project which the client defines. The microtask programming hackathon does not have team-building activities. All participants work together as individual contributors to build the software project.

To investigate these questions, I plan to conduct the first microtask programming hackathon. I expect to conduct a competition-based event in which a crowd of developers work intensively over 2 or 3 days to complete microtasks. In this study, I will investigate how our approach works with a large crowd of developers, resolves conflicts among developers, and impacts the time to market. This study will be conducted in collaboration with the Student-Run Computing and Technology (SRCT, pronounced "circuit")[10]. SRCT is a student organization at George Mason University which enhances student computing by developing and maintaining systems that provide specific services for Mason's community. They will act as the client and help develop the project's requirements. In discussing potential microservice projects with the leadership of SRCT, they suggested building a microservice for "Where", a map web app for the Mason campus that helps students find the way to their classes, study spots, and buses to catch. SRCT will assist in recruiting Mason students as crowd workers. The hackathon will be virtual and entirely remote, enabling a large group to work together without the need to be physically collocated. If successful, I hope to use the software students implement in the hackathon to launch a web app that can be used by students at Mason. To motivate participation, I will consider incentive mechanisms for awarding gift cards based on the level of participation of participants. I will distribute prizes of varying amounts to crowd workers. As workers complete microtasks, each contribution is given a rating. Ratings are then summed to generate a score for each worker. This score is visible to the entire crowd that participated in the project on a global leaderboard, motivating contributions and higher quality work. Hackathaons often used chat tools to help coordinate, such as Slack. Communications among crowd workers will happen only via Q&A. However, additional communication tools will assist in communication between the experimenter and crowd workers.

In this study, I will investigate three research questions:

RQ.1: What are the potential benefits and challenges of applying behavior-driven microtask programming to large-scale software development? To answer this research question, I will examine the feasibility of the approach, focusing on the benefits and challenges of facilitating coordination and communication among contributors. I will collect qualitative and quantitative data from several sources, including a survey, interviews, and programming environment logs. In the post-task survey, I will ask crowd workers and the client (SRCT members) open-ended questions about their experience. In semi-structured interviews with the client and several participants, I will ask open-ended questions about challenges and benefits of my approach. I also will collect quantitative data. I will measure the output of the team by counting the LOC written as well as creating a test suite to measure the functionality successfully completed.

RQ.2: What are the potential benefits and challenges for participants who contribute to a microtask programming hackathon? Several studies have focused on identifying the benefits

[10]https://srct.gmu.edu

and challenges of traditional hackathons for participants [43, 40, 26, 11, 12]. However, as no previous microtask programming hackathon has ever been conducted, it is unclear how these might translate into this new setting. To investigate this, I will analyze data from post-task surveys and interviews.

I hypothesize that participants will receive both social and technical benefits. I hypothesize participants may perceive several technical benefits from the new microtask programming form of a hackathon. I hypothesize the below benefits are unique and belong to the microtask programming hackathon type, and these benefits do not exist in traditional hackathons. The reasons for these differences in technical benefits might be the BDD workflow, the scope of self-contained microtask, and preconfigured IDE participants, the hackathon structure. Participants will be onboarded quickly, will not experience any barriers delays when beginning work, will have the opportunity to contribute to the final project. (1) Participants will perceive a benefit because the workflow in enabling them to be quickly on-boarded and begin programming work. (2) In hackathons, participants may be delayed when waiting for the output of other participants. For example, a participant implementing functionality might need to wait for another to finish designing a form. However, in the microtask programming hackathon, I expect participants will not experience any barriers or delays when beginning work. (3) In traditional hackathons, only a few projects "win" the competition, which may decrease the motivation of participants. In contrast, in the microtask programming hackathon, all participants will have the opportunity to contribute to the final project, even if the contribution is very small. (4) Participants may benefit from experiencing and learning about a new future software development technique, microtask programming.

Regarding the social benefits, I hypothesize that microtask programming will have the same benefits as traditional hackathons for participants. Likewise traditional hackathon[26, 55, 12], microtasking hackathon's participants will benefit from building connections, learning new things, enjoying. (1) Participants will be able to build new connections by meeting and collaborating with new people they otherwise would not. (2) The hackathon could offer new knowledge in meeting participants with different ideas of how to do programming work. (3) Participants may enjoy working with other participants they have never met before.

RQ.3 What are the potential benefits and challenges for clients of microtask programming hackathons? To answer this research question, I will interview the members of SRCT who will act as the project client. I will test three hypotheses. (1) I hypothesize that building software in 2 or 3 days attracts clients by decreasing the time-to-market. (2) Building software, not a demo or proof of concept, reduces the concerns of clients about participants leaving the project. The microtask programming hackathon's output is a software project. However, the output of traditional hackathons are prototypes or proof of concepts [26, 44]. After the microtask programming hackathon event, the client should not need to employ the winning teams to construct a new, real software application based on the program constructed during the hackathon. I expect them to instead be able to use the software built during the hackathon. (3) Given the previous benefits, I hypothesize that clients will be interested in conducting future microtask programming hackathons.

4.2 Co-pilot microfrontend programming

Study 3 (Section 3.3.3) revealed several limitations in our approach for microtasking frontends. Inspired by the role of the co-pilot in TopCoder [11] who is responsible for managing and overseeing development work, two dedicated developers were responsible for creating and decomposing all microtasks as well as themselves implementing the frontend microtasks. Microtasking was only adopted for parts of the project which required less project knowledge to complete or that were not complex. Tasks that required more knowledge still involved a dedicated software engineer, as they required a level of knowledge that crowd workers did not have. To better understanding challenges and benefits of apply muckraking to frontend we need more investigation. We will create a new workflow that addresses several of these limitations and which enables crowd workers to build a frontend through microtasking by using the Co-pilot developers.

To enable microtasking partially in frontend programming work, we will evaluate a new workflow to decompose frontend web development tasks into microtasks at the granularity of UI components. We will evaluate the workflow. We expect to better understand its benefits and limitations. We expect that it may enable crowd developers to implement more complex tasks. However, the workflow still requires dedicated developers, acting as a co-pilot, to be closely involved in implementing complex tasks.

4.3 Microfrontend programming

To better support applying microtask programming to complex programming tasks such as developing a frontend web app, I will create a novel microtask programming workflow, implement it in a system, and evaluate it by conducting a user study. To better understand how to employ microtasking in frontend web development, I have already completed one step (evaluating one workflow in an industrial context, Section 3.3.3) and will first complete a second step (evaluating a second workflow in an industrial context, Section 4.2). The proposed work will build on the lessons learned in these steps and aim to remove limitations we identify in these studies. The goal is to enable crowd developers to implement, debug and test a frontend entirely through microtasking, which was only partially realized in the prior workflows to date.

In the first study of frontend programming through microtasks, conducted with my collaborators at NTT (Section 3.3.3), crowd workers and two dedicated software engineers were able to implement a web-based software frontend and backend. However, dedicated software engineers implemented most of the frontend tasks. Microtasking was only adopted for parts of the project which required less project knowledge to complete or that were not complex. Tasks that required more knowledge still involved a dedicated software engineer, as they required a level of knowledge that crowd workers did not have. This included 1) designing the system architecture, 2) designing and configuring the database, 3) completing tasks that involved screen transitions, 4) completing tasks which required infrastructure knowledge. In the proposed study which included frontend web development work (Section 4.2), we will evaluate a new workflow to decompose frontend web development tasks into microtasks at the granularity of UI components. Through the evaluation of workflow in Section 4.2, I expect to better understand its benefits and limitations. I expect that it may enable crowd developers to implement more complex tasks. However, the workflow still re-

[11] https://www.topcoder.com

quires dedicated developers, acting as a co-pilot, to be closely involved in implementing complex tasks.

A general approach to deal with complexity is to divide and conquer: decompose complex work into smaller pieces and then compose contributions back together. However, a complex task may still be hard to understand even if it is split into smaller parts if there are many dependencies between parts. To make a task easier to understand, I need to minimize the coupling among them. The key challenge is creating a workflow for decomposing frontend tasks and finding the right scope for a microtask. Frontend tasks usually have many dependencies. For instance (see Figure 5), in the React frontend, some components are completely dependent on a parent component, which provide context with which they are initialized. Without the parent component, those components will have limited functionality. Whenever a developer want to implement one of these children components, the developer need to understand the large context of parent and siblings then implement all of them together. This large scope of a task is a barrier for microtasking. If it wants to be implemented in microtasking there should be decomposition techniques. In this decomposition resolving dependency of microtasks is also a serious challenge. Or, another example is in a frontend implementing if multiple actions may occur on a single HTML element, we can not break down those dependent actions in several components. The more potential ways in which it can change, the larger the component gets. And if a task has actions that affect multiple types of UI parts, the task will become huge. The choice of decomposition determines the workflow of the approach, encompassing the individual steps, the context and information offered in each step, and the types of contributions which can be made.

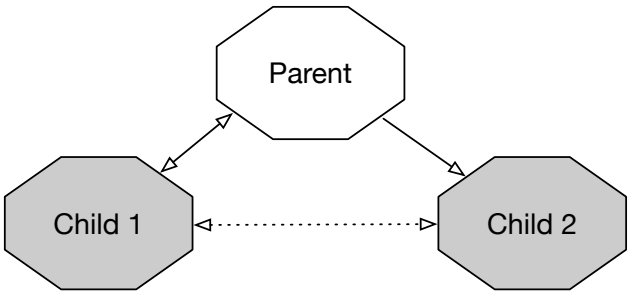

Figure 5: This figure depicts three parts of a frontend. One part is parent of two other. Child 1 has an input field, on typing that value must be sent to child 2 parts via Parent. A developer needs to be aware about a huge context of all 3 parts to implement one of them. These 3 parts need to be implemented together in one task because they are dependent. Finding a right scope to decompose this task and resolving the dependencies are main research question of this proposed work.

I will create a new workflow which enables crowd workers to build a frontend entirely through microtasking. I will implement my approach in a new prototype system, CrowdGUI. After implementing the workflow in a system, I will evaluate my approach by conducting a user study.

4.3.1 CrowdGUI workflow

My workflow will try to decompose frontend work into parts that can be implemented separately regardless of dependencies. It will build UIs in isolation to manage UI complexity. One obvious

Table 1: *CrowdGUI* enables workers to make contributions through four microtask types. Each offers editors for creating content, context views that make the task self-contained by offering necessary background information, and contributions the worker may make.

Microtask	Editor	Context view	Possible contributions
Write Component Description	Component description text editor	1) Description and signature of component 2) Implementation of the component	1) Component description 2) Report as not implementable
Edit a Component	Component code editor	1) Description and signature of component 2) Implementation of the component	1) Code and pseudocode 2) Report as not implementable
Request a New Component	Component description text editor	Description and signature of component	1) Component description 2) Component signature
Review	Review	Contribution and original task context	Review and Rating

way to decompose frontend work is through components. A component in my workflow is a well-defined and self-contained piece of frontend UI. The workflow decomposes the frontend to loosely coupled and highly cohesive components. The workflow will scope components in microtasks then these microtasks could be assign to crowd workers to be implemented. The microtask will follow the single responsibility principle; that is, a component should ideally only do one thing. The workflow will build interfaces from the bottom up. It builds small components then combines them into pages in a process called Component-Driven Development (CDD). Inspired by CDD, my workflow will decompose a frontend into React components. I choose React because it supports and focuses on cut-up the splitting interface into small repeatable patterns and developing patterns that create a page layout. In my workflow, crowd workers only focus on the small context of a single component. If the component has a dependency, it is explicitly specified in the component's description. The crowd worker assumes the dependent component is implemented and worked correctly by another crowd worker and only needs to call a callback function, get or pass the data to a state.

All work begins with a client request. The client logins into the CrowdGUI, then the client-request page describes a frontend app into several components. The client decomposes a frontend into a list of components. For each, the client creates a brief description for each component. The client request will be similar to Figure 3, but instead of defining functions, the client describes components. Next, crowd workers login into CrowdGUI. They are shown a welcome screen that contains some basic information about CrowdGUI and tutorials explaining the CrowdGUI. The crowd worker then provided a random microtask automatically generated by the system. The crowd worker may choose to either complete or skip this microtask.

Table. 1 summarizes the context and possible contributions of each microtask in CrowdGUI environment. In the CrowdGUI, contributions are made through 4 different types of microtasks: *Write Component Description*, *Implement a Component*, *Request a New Component*, and *Rview*. After preparing the client request, CrowdGUI generates *Write Component Description* microtasks for each component and assign it to workers. It is possible workers want to change description of a component any time, CrowdGUI gives them the ability to reissue a *Write Component Description* and edit description of component. *Edit a Component* is generated by the CrowdGUI until all components would be completely implemented. Sometimes workers want to decompose a component and add another component to the UI. Worker creates a *Request a New Component* then another worker will implement it. For either *Write Component Description* or *Edit a Component* the CrowdGUI generates Review microtasks.

Figure 6 depicts 4 simple examples of microtasks generated to implement three components.

Figure 6: An example of microtasks generated to implement 3 dependent components. From an initial client request describing this function, for each component a *Write Component Description* microtask is first generated. I only depicted one of them, 1, in this microtask worker specify dependencies and submit the task. After submiting the 3 *Write Component Description* microtasks, the CrowdGUI generates 2, 3, and 4. Next, workers implement these generated *Edit Component* microtasks.

Following the creation of a new frontend from the client request, *Write Component Description* (1) microtask is generated for each of 3 components. In Figure 6 1, a worker firstly reads the line 1 which is written by the client. The worker could also see a list of all components and their descriptions in the CrowdGUI environment. The worker decides that *TodoContainer* is a parent component that needs to get data from one child by a callback and then pass it to another child to present the data. Then worker writes texts in lines 2 and 3 to include two child components. The worker adds a text to specify a callback method that might get data from an input text and send it to another component (A). Finally, the worker submits the microtask. Whenever *Write Component Description* is submitted a *Review* microtask is generated to accept or reject the contribution.

As Figure. 6 2 shows, an *Edit a Component* is generated whenever a *Review* accept the *Write Component Description* for that component. In the *Edit a Component* microtask worker only write codes or pseudocode (B). Worker read the function description then without knowing about the implementation of two children component added lines 7, 8,9, and 10. In line 7 and 9 it gets data from the children component via a callback function and in line 10 pass it as a state.

Similar to the *TodoContainer* components, the CrowdGUI generated *Write Component Description* microtasks for *TodoItem* and *TodoInput* components and crowd added descriptions to address the dependencies. I did not sketch the *Write Component Description* microtasks. These *Write Component Description* microtasks are accepted after their corresponding *Review* microtasks. Next, another worker is assigned to work on *Edit a Component* (3). The worker reads the description and figure out this component needs to get data from another component (C), therefore, worker implements line 4 (D). In a same scenario, a worker by reading E in the task description of 4, implement code in line 7 (F).

In my workflow, dependencies are identified and described in *Write Component Description* microtask. Next, in *Edit Component* the microtasks component with its dependencies is implemented. As Figure 6 shows, *TodoItem* component (3) needs to update its state (line 4) by the adding event in its sibling component, *InputTodo* component (4). *InputTodo* component has an input field on typing; that value must be sent to the other child of the component via parent. One solution is the parent component with a callback method (4, line 7 and 9) to get the value from one child (*InputTodo*) and give it to another child,*TodoItem* (line 6 and 10). The *InputTodo* prepare data and pass it to the parent component (*TodoContainer*) in a callback function and the other child present it. In this workflow, workers without knowing about the context of other component implement codes. Whenever they face with a wrong specification in component description or in the implementation workers can re-issue a *Write Component Description* to address that issue. The CrowdGUI is responsible for generating and keeping the last version of the component.

5 Research plan

In Spring 2021, I plan to complete and publish the proposed work described in section 4.2. At the beginning of the fall of 2021, I will begin designing and performing the formal evaluation of large-scale microtask programming and publishing my results (section 4.1). At the beginning of spring of 2022, I will complete the last proposed work (section 4.3). For the remainder of the 2022 year, I will finish writing my dissertation and publish the user studies results. I plan to graduate at the end of 2022.

6 Acknowledgments

I want to thank Thomas LaToza for advising me on this work. This work has significantly benefited from discussions with other faculty, students, and professors, including Maryam Arab, Abdulaziz Alaboudi, Austin Henley, Michael Lee, Sahar Mehrpour, Steve Oney, Alex Polozov, David Samudio. I would also like to thank my Master's advisors Hamidreza Memarzadeh-Tehran and Mohammad Khansari, for encouraging me to pursue a Ph.D. I would also like to thank my family for their continuing support of my career, especially my wife, Fatemeh, and my mother, Shamsi. This research was supported in part by the National Science Foundation under grants CCF-1414197 and CCF-1845508.

References

[1] Emad Aghayi. Large-scale microtask programming. In *Symposium on Visual Languages and Human-Centric Computing*, pages 1–2, 2020.

[2] Emad Aghayi, Thomas D. LaToza, Paurav Surendra, and Seyedmeysam Abolghasemi. Crowdsourced behavior-driven development. *Journal of Systems and Software*, 171:110840, 2021.

[3] Hayward P Andres and Robert W Zmud. A contingency approach to software project coordination. *Journal of Management Information Systems*, pages 41–70, 2002.

[4] Andrew Begel and Beth Simon. Novice software developers, all over again. In *International Workshop on Computing Education Research*, pages 3–14, 2008.

[5] Michael S Bernstein, Greg Little, Robert C Miller, Björn Hartmann, Mark S Ackerman, David R Karger, David Crowell, and Katrina Panovich. Soylent: a word processor with a crowd inside. In *Symposium on User Interface Software and Technology*, pages 313–322, 2010.

[6] Dimitar Bounov, Anthony DeRossi, Massimiliano Menarini, William G. Griswold, and Sorin Lerner. Inferring loop invariants through gamification. In *Conference on Human Factors in Computing Systems*, pages 1–13, 2018.

[7] Ning Chen and Sunghun Kim. Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles. In *Automated Software Engineering*, pages 140–149, 2012.

[8] Yan Chen, Sang Won Lee, Yin Xie, YiWei Yang, Walter S. Lasecki, and Steve Oney. Codeon: On-demand software development assistance. In *Conference on Human Factors in Computing Systems*, pages 6220–6231, 2017.

[9] Justin Cheng, Jaime Teevan, Shamsi T Iqbal, and Michael S Bernstein. Break it down: A comparison of macro- and microtasks. In *Conference on Human Factors in Computing Systems*, pages 4061–4064, 2015.

[10] Fabian Fagerholm, Alejandro Sanchez Guinea, Jay Borenstein, and Jürgen Münch. Onboarding in open source projects. *IEEE Software*, pages 54–61, 2014.

[11] Myrna Flores, Matic Golob, Doroteja Maklin, Martin Herrera, Christopher Tucci, Ahmed Al-Ashaab, Leon Williams, Adriana Encinas, Veronica Martinez, Mohamed Zaki, et al. How can hackathons accelerate corporate innovation? In *International Conference on Advances in Production Management Systems*, pages 167–175, 2018.

[12] Allan Fowler, Foaad Khosmood, Ali Arya, and Gorm Lai. The global game jam for teaching and learning. In *Conference on Computing and Information Technology Research and Education New Zealand*, pages 28–34, 2013.

[13] Rishab Aiyer Ghosh. Understanding free software developers: Findings from the floss study. *Perspectives on Free and Open Source Software*, pages 23–47, 2005.

[14] Max Goldman. *Software Development with Real-time Collaborative Editing*. PhD thesis, Massachusetts Institute of Technology, 2012.

[15] Max Goldman, Greg Little, and Robert C. Miller. Real-time collaborative coding in a web ide. In *Symposium on User Interface Software and Technology*, pages 155–164, 2011.

[16] Morton Goldman, Joseph W Stockbauer, and Timothy G McAuliffe. Intergroup and intragroup competition and cooperation. *Journal of Experimental Social Psychology*, pages 81–88, 1977.

[17] Luis Alberto Cisneros Gómez et al. *Analysis of the impact of test based development techniques (TDD, BDD, AND ATDD) to the software life cycle*. PhD thesis, Polytechnic of Leiria, 2018.

[18] Mohamad Hoseini, Fatemeh Saghafi, and Emad Aghayi. A multidimensional model of knowledge sharing behavior in mobile social networks. *Kybernetes*, 2018.

[19] InnerSourceCommons. Behavior driven development, Jul 2020.

[20] Shamsi T Iqbal, Jaime Teevan, Dan Liebling, and Anne Loomis Thompson. Multitasking with play write, a mobile microproductivity writing tool. In *Symposium on User Interface Software and Technology*, pages 411–422, 2018.

[21] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. The onion patch: migration in open source ecosystems. In *Special Interest Group on Software Engineering Symposium and the European Conference on Foundations of Software Engineering*, pages 70–80, 2011.

[22] Huan Jiang and Shigeo Matsubara. Efficient task decomposition in crowdsourcing. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 65–73, 2014.

[23] Joy Kim, Sarah Sterman, Allegra Argent Beal Cohen, and Michael S. Bernstein. Mechanical novel: Crowdsourcing complex work through reflection and revision. In *Conference on Computer Supported Cooperative Work and Social Computing*, 2017.

[24] Aniket Kittur, Jeffrey V Nickerson, Michael Bernstein, Elizabeth Gerber, Aaron Shaw, John Zimmerman, Matt Lease, and John Horton. the future of crowd work. In *Conference on Computer Supported Cooperative Work*, pages 1301–1318, 2013.

[25] Aniket Kittur, Boris Smus, Susheel Khamkar, and Robert E Kraut. Crowdforge: Crowdsourcing complex work. In *Symposium on User Interface Software and Technology*, pages 43–52, 2011.

[26] M. Komssi, D. Pichlis, M. Raatikainen, K. Kindström, and J. Järvinen. What are hackathons for? *IEEE Software*, 32:60–67, 2015.

[27] Walter S Lasecki, Juho Kim, Nick Rafter, Onkur Sen, Jeffrey P Bigham, and Michael S Bernstein. Apparition: Crowdsourced user interfaces that come to life as you sketch them. In *Human Factors in Computing Systems*, pages 1925–1934, 2015.

[28] 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*, pages 23–27, 2015.

[29] T. D. LaToza, W. Ben Towne, A. van der Hoek, and J. D. Herbsleb. Crowd development. In *Workshop on Cooperative and Human Aspects of Software Engineering*, pages 85–88, 2013.

[30] Thomas D LaToza, Micky Chen, Luxi Jiang, Mengyao Zhao, and André Van Der Hoek. Borrowing from the crowd: A study of recombination in software design competitions. In *International Conference on Software Engineering*, pages 551–562, 2015.

[31] Thomas D LaToza, Arturo Di Lecce, Fabio Ricci, Ben Towne, and Andre Van der Hoek. Microtask programming. *Transactions on Software Engineering*, pages 1–20, 2018.

[32] Thomas D LaToza, W Ben Towne, Christian M Adriano, and André Van Der Hoek. Microtask programming: Building software with a crowd. In *Symposium on User Interface Software and Technology*, pages 43–54, 2014.

[33] Thomas D LaToza and Andre Van Der Hoek. Crowdsourcing in software engineering: Models, motivations, and challenges. *IEEE Software*, pages 74–80, 2015.

[34] Thomas D LaToza and Andre van der Hoek. Crowdsourcing in software engineering: Models, motivations, and challenges. *IEEE software*, pages 74–80, 2016.

[35] Sang Won Lee, Rebecca Krosnick, Sun Young Park, Brandon Keelean, Sach Vaidya, Stephanie D. O'Keefe, and Walter S. Lasecki. Exploring real-time collaboration in crowd-powered systems through a ui design tool. *Computer-Supported Cooperative Work and Social Computing*, pages 1–23, 2018.

[36] Sorin Lerner, Stephen R Foster, and William G Griswold. Polymorphic blocks: Formalism-inspired ui for structured connectors. In *Conference on Human Factors in Computing Systems*, pages 3063–3072, 2015.

[37] Ke Li, Junchao Xiao, Yongji Wang, and Qing Wang. Analysis of the key factors for software quality in crowdsourcing development: An empirical study on topcoder. com. In *Computer Software and Applications Conference*, pages 812–817, 2013.

[38] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, pages 57–84, 2017.

[39] James Martin and Carma L McClure. *Software Maintenance: The Problems and Its Solutions*. Prentice Hall Professional Technical Reference, 1983.

[40] Lukas McIntosh and Caroline D. Hardin. Do hackathon projects change the world? an empirical analysis of github repositories. In *Technical Symposium on Computer Science Education*, page 879–885. Association for Computing Machinery, 2021.

[41] Michael Nebeling, Stefania Leone, and Moira C. Norrie. Crowdsourced web engineering and design. In *International Conference on Web Engineering*, pages 31–45, 2012.

[42] Dan North. Introducing behaviour driven development. *Better Software Magazine*, 2006.

[43] Jari Porras, Jayden Khakurel, Jouni Ikonen, Ari Happonen, Antti Knutas, Antti Herala, and Olaf Drögehorn. Hackathons in software engineering education: lessons learned from a decade of events. In *International Workshop on Software Engineering Education for Millennials*, pages 40–47, 2018.

[44] M. Raatikainen, M. Komssi, V. d. Bianco, K. Kindstöm, and J. Järvinen. Industrial experiences of organizing a hackathon to assess a device-centric cloud ecosystem. In *Annual Computer Software and Applications Conference*, pages 790–799, 2013.

[45] Mazedur Rahman and Jerry Gao. A reusable automated acceptance testing architecture for microservices in behavior-driven development. In *Symposium on Service-Oriented System Engineering*, pages 321–325, 2015.

[46] Daniela Retelny, Michael S. Bernstein, and Melissa A. Valentine. No workflow can ever be enough: How crowdsourcing workflows constrain complex work. *Conference on Computer-Supported Cooperative Work and Social Computing*, pages 1–23, 2017.

[47] Caitlin Sadowski and Thomas Zimmermann. *Rethinking productivity in software engineering*. Springer Nature, 2019.

[48] Makon Saengkhattiya, Mikael Sevandersson, and Unai Vallejo. Quality in crowdsourcing-how software quality is ensured in software crowdsourcing. Master's thesis, Department of Informatics, Lund University, 2012.

[49] Shinobu Saito, Yukako Iimura, Emad Aghayi, and Thomas D. LaToza. Can microtask programming work in industry? In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 1263–1273, 2020.

[50] Todd W Schiller and Michael D Ernst. Reducing the barriers to writing verified specifications. *Special Interest Group on Programming Languages Notices*, pages 95–112, 2012.

[51] John Ferguson Smart. *BDD in Action*, volume 12. Manning Publications New York, 2014.

[52] Igor Steinmacher, Marco Aurelio Graciotto Silva, Marco Aurelio Gerosa, and David F Redmiles. A systematic literature review on the barriers faced by newcomers to open source software projects. *Information and Software Technology*, pages 67–85, 2015.

[53] Klaas-Jan Stol and Brian Fitzgerald. Two's company, three's a crowd: A case study of crowdsourcing software development. In *International Conference on Software Engineering*, pages 187–198, 2014.

[54] Susan G Straus and Joseph E McGrath. Does the medium matter? the interaction of task type and technology on group performance and member reactions. *Journal of applied psychology*, 1994.

[55] Erik H Trainer and James D Herbsleb. Beyond code: prioritizing issues, sharing knowledge, and establishing identity at hackathons for science. In *CSCW Workshop on Sharing, Re-use, and Circulation of Resources in Scientific Cooperative Work*, 2014.

[56] Georg Von Krogh, Sebastian Spaeth, and Karim R Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, pages 1217–1241, 2003.

[57] Jeremy Warner and Philip J. Guo. Codepilot: Scaffolding end-to-end collaborative software development for novice programmers. In *Conference on Human Factors in Computing Systems*, pages 1136–1141, 2017.

[58] Alexandre Lazaretti Zanatta, Leticia Machado, and Igor Steinmacher. Competence, collaboration, and time management: Barriers and recommendations for crowdworkers. In *Workshop on Crowd Sourcing in Software Engineering*, pages 9–16, 2018.