Noname manuscript No.
(will be inserted by the editor)

Crowdsourced Microservices

Behavior-Driven development Applied to Microtask Programming

Emad Aghayi · Thomas D. LaToza · Paurav Surendra · Seyedmeysam Abolghasemi

Received: date / Accepted: date

Abstract Behavior-Driven Development (BDD) as a popular agile software development processes commonly used in microservices development. Microservices decompose large and complex web back-ends into single-purpose services, enabling complex back-ends to be built, reused and deployed quickly. Besides, workflows in microtasks crowdsourcing organizes complex work, decomposing large tasks into small, relatively independent microtasks. In this paper, we introduce a BDD-based workflow for implementing microservices through microtasks crowdsourcing. In our approach, 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 follows the BDD-based workflow and implements the endpoints, identifying individual endpoint behaviors which they test, write, and debug, creating new functions and interacting with persistence APIs as needed. To evaluate our approach, we conducted a feasibility study in which a small crowd worked to implement a small todo microservice, then we built a test suite and evaluated the final output of the crowd. The crowd created an implementation with only four defects, completing 350 microtasks and implementing 13 functions. We discuss the implications of these findings for incorporating crowdsourcing into software projects.

Keywords Programming Environments · Behavior-Driven Development · Crowdsourcing · Microservices

1 Introduction

The back-end of a web application needs to be very quickly implemented to address the change requests of clients. This requirement forced styles of software

E. Aghayi, Thomas D. LaToza, P. Surendra, and S. Abolghasemi
Department of Computer Science, George Mason University, 4400 University Drive, Fairfax, VA 22030
E-mail: {eaghayi, tlatoza, psurendr}@gmu.edu , sabolgha@cs.odu.edu

to shift from monolithic to microservices style. As a result, modern web applications often consist of a front-end, executing in the browser, and a back-end, organized as a set of interconnected microservices. In contrast to monolithic back-ends in which all functionality is hosted by a single web-server, microservices decompose back-ends into many interconnected services. Decomposing back-ends into microservices brings many benefits in development, deployment, and reuse. Microservices are often built, deployed, and maintained by small dedicated software development teams, enabling an individual team to own functionality end-to-end with minimal dependencies on other teams [29]. While HTTP requests are stateless, microservices often involve persistence, requiring interactions external services which can persist state between requests. Nowadays Behavior-Driven Development (BDD) is frequently used in the microservices development style. BDD[30, 1] help software to be fast developed, efficiently maintained, and instantly deployed.

In situations where business goals change, new competitors emerge, or unforeseen issues arise, it is valuable to implement new microservices quickly. Through crowdsourcing, many developers, either within or outside an organization, might contribute. By decomposing long tasks into short *microtask* which can be completed in parallel, tasks which otherwise might take days might be completed far more quickly. A variety of systems have demonstrated the promise of the application of crowdsourcing to programming [5, 10, 9, 18–23, 28, 33]. For example, in Apparition [18], a client developer narrates a description for a user interface in natural language, and crowd workers translates this descriptions into user interface elements, visual styles, and behavior. In CodeOn [5], 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. Not only no prior system has offered an approach for crowdsourcing the implementation of microservices, but also no prior crowdsourcing programming benefited from agile methodologies such as BDD.

In our previous research [20], we proposed a set of design principles for microtask programming reflecting the lessons we learned from their previous researches[23, 22] and applied them to CrowdCode. Specifically, CrowdCode provided techniques for ensuring quality, providing feedback to contributors, debugging modularly, enabling contributors[20]. CrowdCode implemented three design principles 1) Decontextualize Contributions. 2) Automatically Generate Microtasks. 3) Achieving Quality through Iteration. Our experiments showed the techniques that we used in the first and last principals have serious problems. For instance, in the CorwdCode's workflow contributors on microtasks separately interpreted the description of the function and contribute, different interpretation was a source of conflict among crowd. Thus we had to find solutions for problem existed in the CrowdCode. Clearly, our workflow was not efficient, providing an effecient workflow might alleviate drawbacks in the both principals . Microtasks crowdsourcing systems rely on the concept of a workflow which decomposes a larger task into a sequence of microtasks. Besides, a fundamental challenge in crowdsourcing work is the design of a workflow. Based on the lessons we learned from CrowdCode and strengths of the BDD

approach, we propose a new workflow equipped by BDD to fix the problems existed in the first and third design principals. Thus, we designed a workflow based on BBD, then implemented the workflow by building a tool.

In this paper, we introduce BDD-base workflow and tools for crowdsourcing the implementation of a microservice through short, self-contained contributions. Inspired by techniques for implementing libraries through *microtasks* [22, 20], we offer the first technique which enables the implementation of a microservice through microtasks. Microservices offer a natural decomposition boundary in large web applications with a well-defined interface, offering an ideal way for a traditional software project to quickly crowdsource application functionality. Rather than engage in an extended onboarding process for new developers to become familiar with a large software project or require existing developers to manually create and manage each individual code contributions, microservices offer an alternative model for crowdsourcing in which a client team describes desired functionality which is implemented entirely by the crowd.

As in all microtask crowdsourcing systems, key to the design is the choice of workflow. In designing our workflow, our key goals were to reduce the need for ongoing client involvement while still enabling feedback within and after each microtask. In our approach, a client first defines the intended behavior of the microservice as a set of endpoints. Each endpoint is then iteratively constructed by the crowd. Inspired by BBD, each microtask asks a worker to identify a behavior and write a test first, before then implementing and debugging the behavior. Each microtask's goal is to make a new test pass, offering immediate feedback within the microtask itself. A secondary review microtask then offers additional feedback on each contribution, which triggers additional work if required.

We implemented our approach in a cloud IDE, *Crowd Microservices*. *Crowd Microservices* includes an editor for clients to describe requested endpoints, an environment in which crowd workers can identify, test, implement, and debug behaviors, and an infrastructure for automatically generating and managing microtasks. Persistence is exposed through an external API, which supports a sandboxed environment for development. After completion, the microservice may be automatically deployed to a hosting site. To evaluate our approach, we conducted a user study with 9 crowd workers, who implemented a simple *Todo* microservice, then we built a test suite, evaluated the final output of the crowd. Participants submitted their first microtask 24 minutes after beginning, successfully submitted 350 microtasks, implemented 13 functions and 36 tests, completed microtasks in a median time under 5 minutes, and correctly implemented 27 of 34 behaviors.

In this paper, we contribute

1. the first microtask crowdsourcing system equipped with BDD.
2. a novel BDD workflow for crowdsourcing programming which offers workers immediate feedback on their contributions.
3. *Crowd Microservices*, a cloud-based IDE for building microservices.

4. evidence that behavior-driven microtasks can be completed quickly by crowd workers and used to implement a microservice.

In the rest of the paper, we first illustrate *Crowd Microservices* through an example. We then present the design and evaluation of our system. Finally, we conclude with a discussion of limitations as well as opportunities and future directions.

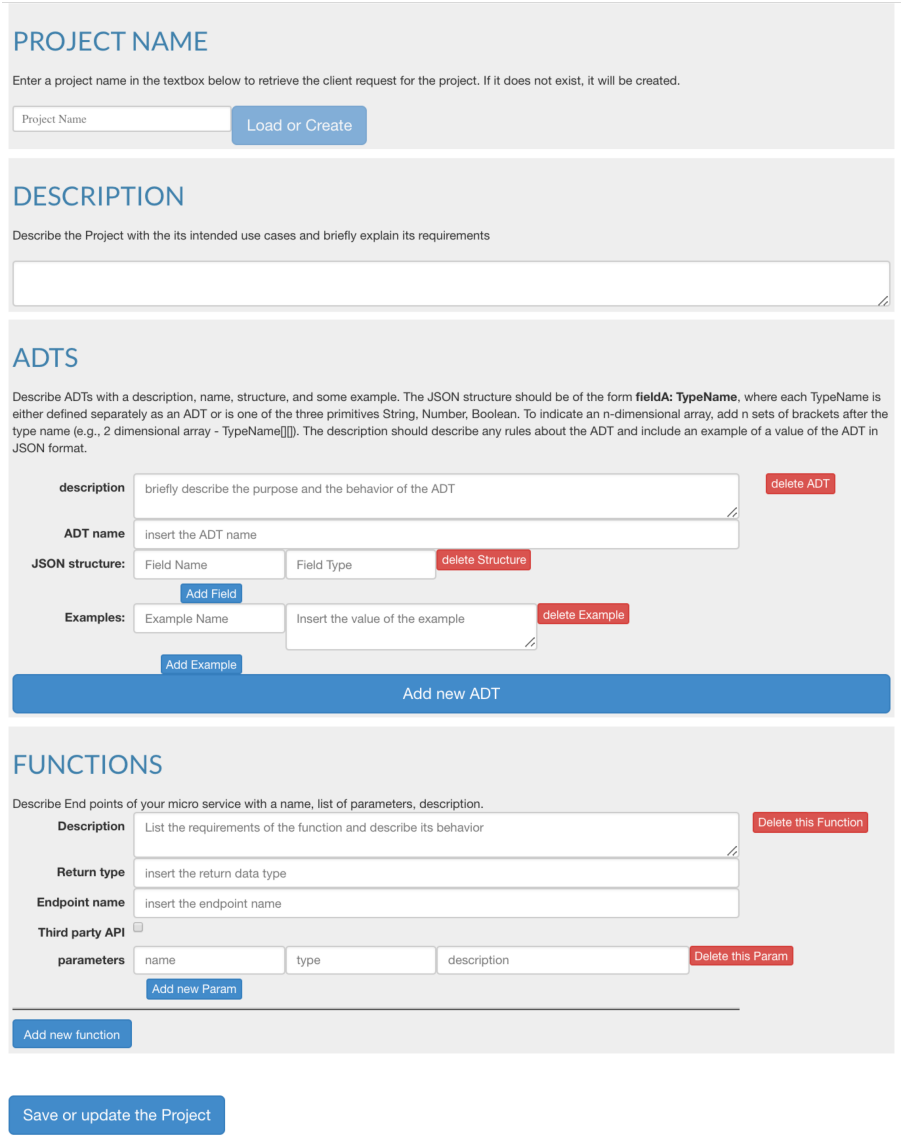

Fig. 1 In the *Client Request*, a client defines a microservice they would like created by describing a set of endpoints and corresponding data structures.

2 MOTIVATING EXAMPLE

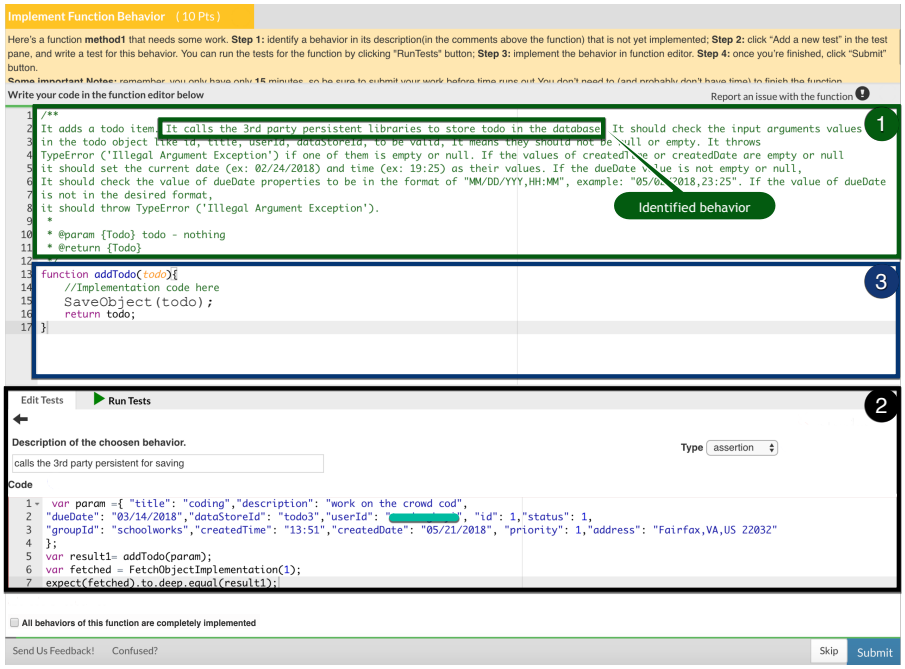

Fig. 2 In the *Implement Function Behavior* microtask, workers first (1) identify a behavior from the description of a function. They then (2) write a test in the test editor to verify the behavior and (3) edit the code for the function to implement it. Finally, they (4) test it by running the tests,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

Bob decides that he would like to add a Todo widget to his app. Hoping to add it as soon as possible, he decides to create the UI himself and use *Crowd Microservices* to implement the functionality to make it functional. Using the *Client-Request* page(Fig. 1), he creates a new Todo microservice project, providing a brief description, defining a set of endpoints, and describing the format of the JSON data used in the request and responses through the data structures editor.

Alice logs in to *Crowd Microservices*, notices the Todo microservice project has a number of open microtasks, and decides to contribute. She watches a short video[1] and reads a short tutorial to familiarize herself with the environment. Viewing the dashboard for the project, she quickly reads the brief description from the client and a list of descriptions of the functions created by the crowd so far. She clicks *Fetch a Microtask*, and *Crowd Microservices* assigns her an *Implement Function Behavior* microtask (Figure 2). Following the behavior-driven workflow, she first reads the description of the function in

[1] https://www.youtube.com/watch?v=qQeYOsRaxHc

the comments above the body, identifying a behavior that does not yet seem to be implemented. Next, she writes a traditional unit test, writing test code with assertions and running the test to verify that it fails. Next, Alice implements the behavior using the function editor. Alice tests her implementation, running the function's tests, but one fails. To understand why, Alice using the debugger to inspect values in the function's execution, hovering over several variables to see their values. Identifying the issue, Alice fixes the problem, finds that all of the tests now pass, and submits.

As Alice works, the crowd simultaneously completes other microtasks. After logging in, Dave is assigned an *Implement Function Behavior* microtask. Unsure how to implement any of the remaining behaviors, he clicks *Skip* and fetches another microtask. This time, he thinks that he can complete this microtask and writes a test and implementation. After a test fails, he realizes he does not know how to correctly call a third-party API function. Using the *Question and Answer* feature, he asks, *How can I store a todo object in the database?* A worker responds, and he figures out how to fix the problem. However, he then sees an alert: *You have spent more than 14 minutes on the current microtask, so try to submit your task in one minute before the system automatically skips it.* Wanting to submit his partially completed work, Dave submits with failed tests. Fetching another microtask and inspecting the implementation, he see that that all behaviors have already been implemented. So he clicks the corresponding checkbox and submits.

After logging in, Oliver is assigned a *Review* microtask which asks him to assess the behavior implemented by Alice. Oliver reads the description of the function, a diff of the code written by Alice, and the tests. Looking at the implementation, he finds it seems incomplete, so he rates her contribution a 2 on a 5 point scale and gives her feedback, *"The behavior asked you to evaluate all input arguments of the function, but you just checked the validity of the date."* Oliver submits, and Alice receives a notification that her work was reviewed and received 2 stars.

After being assigned an *Implement Function Behavior* microtask, Jon decides to implement a format check for the *todoDate* parameter. Believing this to be fairly complex, he decides it would be best implemented in a separate function. He invokes the *create a new function* feature, creating the function checkTodoDataFormat for others to implement. Specifying its behavior, he writes a description and signature. He then calls this new, currently empty, function from the body of the function he is working on. To verify his work, he runs the tests. But as checkTodoDateFormat is not yet implemented, his tests fail (Fig. 5). Jon uses the *Stub* editor (Fig. 6) to replace the actual output with a stub value representing the desired output. This automatically replaces all calls to this function with the inputs and output Jon specifies. Jon runs the tests again, they pass, and he submits.

As the crowd works, each is assigned a score based on the ratings of their contributions. These scores are visible on a global *leaderboard* visible to the entire crowd, encouraging everyone to work hard to place higher.

While the crowd was working, Bob implemented the front-end, inserting requests based on the behavior of the endpoints he specified. After all the microtasks have been completed and the implementation finished, he clicks a button to deploy the microservice. He loads his web app in his browser, seeing the Todo interactions handled by the microservice.

3 RELATED WORK

Our work builds on a number of ideas and techniques pioneered in prior approaches to crowdsourcing and crowdsourcing software work in particular. Behind this work lies several key problems in microtask workflow design, which we focus on below: decomposition and context, parallelism and conflicts, enabling fast onboarding, and achieving quality.

3.1 Decomposition, Context, and Scope

A key challenge in microtasking in all domains is the manner in which work is decomposed into tasks or microtasks which are carried out by crowd workers, as the choice of decomposition creates a workflow and the associated individual steps, the context and information required, and the types of contributions which can be made [31,17,2,12,16,14]. Within crowdsourcing approaches for software engineering work, several points in the design space have been explored [27]. Depending on the choice of microtask boundaries, contributions may be easier or harder, may vary in quality, and may impose differing levels of overhead.

Several techniques decompose programming work into fine-grained microtasks. To generate the microtasks for the crowd, many systems rely on a developer or client to manually author each microtask. One approach, such as in CodeOn[5], is for a developer working in a project to request small microtasks for others to complete. Apparition [18] offers a similar workflow for building user interface prototypes in which requestors describe behavior in natural language todo items and workers view an implementation of the entire UI, select a microtask from a todolist, and implement the requested behavior 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 [28], work for building web apps is broken down by component and crowd members work individually in an isolated environment on each component, their workflow contains only designing and testing tasks; it doe not have coding which is one of the challenging processes in software engineering. Other techniques 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 [22,20]. Another

strand of work has explored creating puzzle games, which formulate complex tasks such as testing and verification as puzzles which can be completed with little or no programming knowledge [4,33,25,3].

3.2 Parallelism and conflicts

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 changes made at the same time conflict. 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 [18] workers acquire write-locks mechanism to avoid conflicts. Similarly, in CrowdCode [20] each contribution occurs 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[39,21]. In CrowdCode, conflicts occurred when workers completed separate microtasks to translate function descriptions into an implementation or tests and each made differing interpretations of a function description [20].

3.3 Fast Onboarding

Another challenge with using crowdsourced contributions to a software project is the process of *onboarding*. Several studies have examined the *joining script* and barriers that open source software developers face onboarding onto a new software project, including installing necessary tools, downloading code from a server, identifying and downloading dependencies, and configuring their build environment [34,36,13,6]. As a result, onboarding onto open source projects can require weeks of time, creating a substantial barrier to casual contributors.

[2] https://www.usertesting.com
[3] https://www.trymyui.com
[4] https://www.utest.com

Microtasks	Editor	Context view	Possible Contributions
Implement Function Behavior	Function and test editor, test runner, stub editor	Description and signature of function, description of requesting function, ADTs	(1) Implement behavior (2) Report an issue in function (3) Mark function as completed
Review	Rating and review text	Description, signature, and implementation of function; function unit tests; ADTs	(1) Rating and review

Table 1 *Crowd Microservices* enables workers to make contributions through two types of microtask. 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.

Researchers have explored designing environments to alleviate these barriers. Several systems offer a preconfigured, online environments in which developers can make microtask contributions for specific types of programming work. In Collabode, multiple developers synchronously edit code at the same time, enabling new forms of collaborative programming [10,9]. 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 [18]. CrowdCode offers an online preconfigured environment for implementing libraries, enabling developers to onboard quickly onto programming tasks [20].

Simplified, preconfigured online environemnts for programming have also been designed to enable teachers to support student assignments. Codeopticon [11] enables instructors to continuously monitor multiple students and help them code. OverCode [8] and Foobaz [7] help mentors to cluster student code submissions, enabling teachers to give feedback on clusters of submissions rather than individual submissions. Codepilot [37] 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.

3.4 Achieving Quality

Crowdsourcing for programming have explored a variety of approaches for ensuring the quality of the result software artifacts creates. There are many potential causes of low-quality contributions, including workers who do not have sufficient knowledge, who put forth little effort, or who are malicious [15]. 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

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

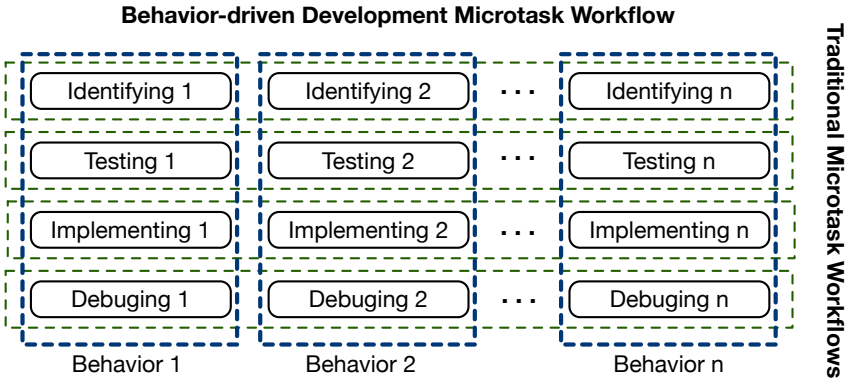

Fig. 3 Traditional microtask programming decompose large programming tasks into separate microtasks in which different types of contributions can be made, such as offering a separate microtask in which a worker might identify all of the behaviors which should be tested in a function or . In the behavior-driven microtask workflow, work is instead decomposed by behavior, where an individual microtask incorporates work of several types for an individual behavior.

the design score [26]. In TopCoder, senior contributors assist in managing the process of creating and administering each task and ensuring quality work is done [35]. Interviews with several software crowdsourcing companies identified 10 methods used for addressing quality, including ranking/rating, reporting spam, reporting unfair treatment, task pre-approval, and skill filtering [32].

In microtask systems, crowd members are often assumed to be little invested in the platform or community. Many crowd programming systems have addresses this problem by assigning the responsibility of feedback and management to the client or developers responsible for requesting the work [9, 5, 18, 24]. 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[20]. 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[22].

4 CROWD MICROSERVICES SYSTEM

In this paper, we describe a behavior-driven development workflow for microtasking programming. 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 the 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. 3).

We apply our approach to the problem of 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 our 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.

In the following sections, we first describe our behavior-driven workflow and then describe how it is applied to the task of implementing a microservice. Fig. 4 surveys our approach.

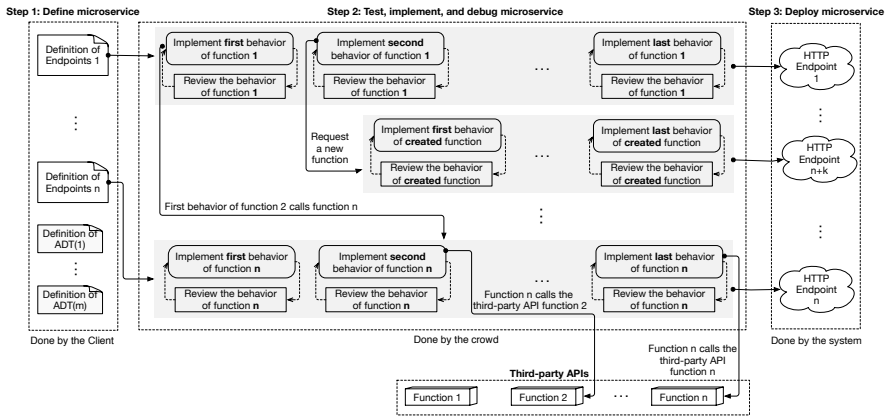

Fig. 4 Work begins with a client request which defines a microservice to implement through a list of endpoints. From this, the system generates initial microtasks to begin implementing each endpoint. In each microtask, an individual behavior is identified, tested, implemented, and debugged, including calling third-party API or crowd-generated functions as needed and requesting additional functions to be written. Each contribution generates a corresponding review microtask, where the contribution is either accepted or issues raised, generating a microtask to resolve any issues. After the crowd has completed implementation of the endpoints, the client may choose to deploy the microservice.

4.1 Workflow

In our behavior-driven microtask workflow, contributions are made through two microtasks: *Implement Function Behavior* and *Review*. Table 1 summarizes the context and possible contributions of each.

4.1.1 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 *Crowd Microservices* 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. As participants work and are 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 [21]. Posted questions are visible to all workers, who may post answers and view previous questions and answers.

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 on a global *leaderboard*, motivating contributions and high quality work to place higher.

4.1.2 Implement Function Behavior Microtask

Workers perform each step in the *Implement Function Behavior Microtask* through the *Crowd Microservices* environment (Fig.2).

1. Step 1: Identify a Behavior. Workers first work to identify single behavior that is not yet implemented from the comments describing the function located above its body (see (1) in Fig. 2). When the function has been completely implemented and no behaviors remain, the worker may indicate this through a checkbox.
2. Step 2: Test the Behavior. In the second step, workers author a test 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. 2). The worker may run the test to verify that it fails with the current implementation. In the test editor worker can write assertion unit tests to evaluate the behavior. Also, in the test editor worker can invoke third-party APIs or the other functions.
3. Step 3: Implement the Behavior. The worker next implements their behavior using a code editor (see (3) in Fig.2). 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 is able to see that this issue has been created, resolve the problem, and begin work again.

4. Step 4: Debug the Behavior. A worker may test their implementation by running the function's unit tests. As it is shown in Fig. 5 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 (Fig. 6), 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 the stubbed values.
5. 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. We derived the 15 minutes from our previous research [20] 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.

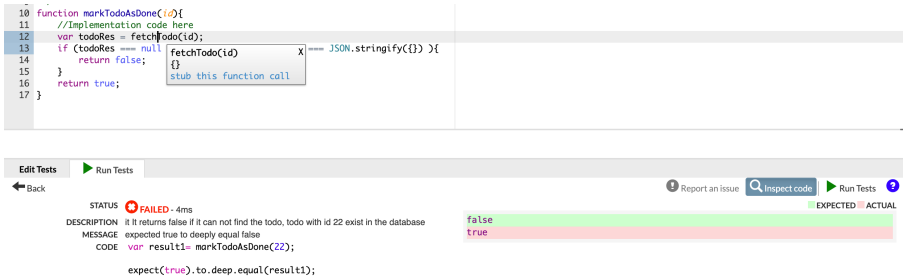

Fig. 5 Workers can debug failed tests by by using the *Inspect code* feature to view the value of any expression. In cases where a function that is called from the function has not yet been implemented, any tests exercising this functionality will fail. To enable the worker to still test their contribution, a worker may create a stub for any function call.

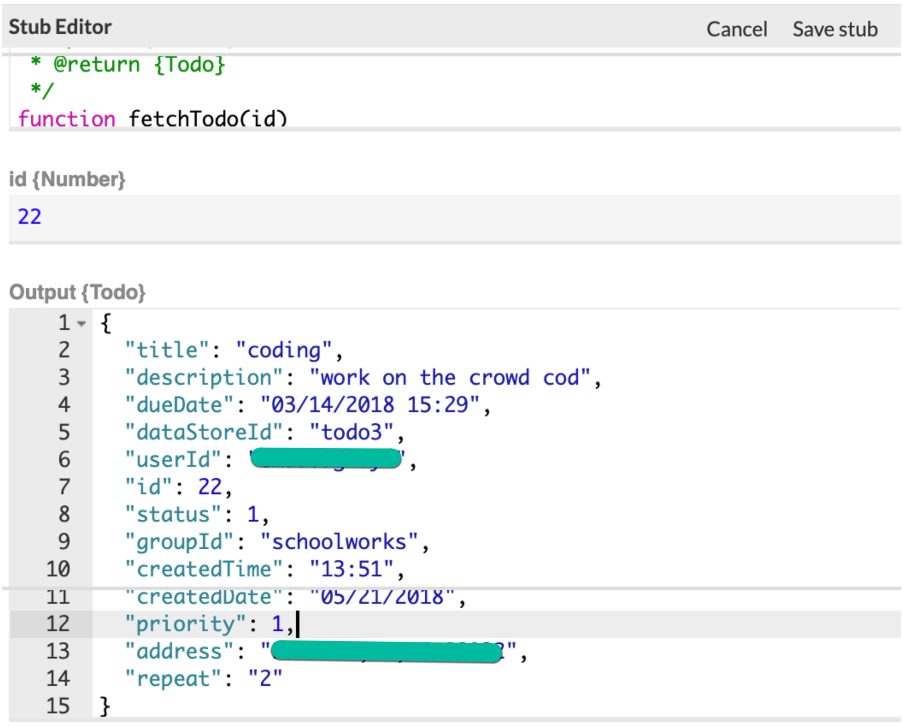

Fig. 6 Workers can view the current output generated by a function call and edit this value to the intended output. This then automatically generates a stub. Whenever the tests are run, all stubs are applied, intercepting function calls and replacing them with the stubbed values.

4.1.3 Review Microtask

In the *Review* microtask, workers assess contributions submitted by other workers. Workers are given a diff of the code submitted with the previous version as well as the tests of the function. Instead of binary classification of rejecting or accepting microtasks, 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.

4.2 Working with Microservices

Our approach applies the behavior-driven development workflow to implementing microservices. Fig. 4 depicts the steps in our process. The microservice is first described by a client through the *Client-Request* page (Fig 1). 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 *Crowd Microservices* project. As shown in step 2 of Fig. 4, 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 by the system 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, *Crowd Microservices* offers a preconfigured environment. As typical microservices often involve persisting data between requests, we 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 our 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 persistence 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. 4). 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 for the endpoint. An example is shown in Fig.7. In this example, a signature *function fetchTodosBasedOnStatus(userId, status)* would generate a GET */fetchTodosBasedOnStatus* endpoint with the parameters as fields in the body of the request. Each endpoint then contains the implementation of the function defined by the crowd. Next, this GitHub project is deployed to

a hosting site. In our prototype, projects are deployed to the Heroku hosting site. A new project instance is created, and the project deployed. After this process has been completed, the client may then begin using the completed microservice by making HTTP requests against the deployed, publicly available microservice. Since clients may have privacy concerns and do not want to deploy the microservice publicly on the Heroku, the client can provide information about its private repository in *Client-Request* page, then *Crowd Microservice* will push the output of the crowd on the private repository.

4.3 Implementation

We implemented our approach as a prototype *Crowd Microservices* IDE. *Crowd Microservices* is a client-server application with three layers: 1) a web client, implemented in AngularJS, which runs on a worker's browser, 2) a back-end, implemented in Node.js, and 3) a persistence store, implemented using Firebase Real-time Database [6]. *Crowd Microservices* 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. When a worker fetches a microtask, the system automatically assigns the worker the next microtask and removes it from the queue.

5 EVALUATION

To investigate the feasibility of applying behavior-driven microtask programming to implementing microservices, we conducted a user study in which a crowd of workers built a small microservice. Specifically, we investigated (1) the ability of crowd workers to make contributions through a behavior-driven microtask workflow, (2) the time necessary to contribute a test and implementation of a behavior, and (3) the feasibility of implementing and testing a microservice entirely through microtasks. We recruited 9 participants to build a small microservice for a Todo application and then analyzed their environment interactions and the resulting code they created.

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

5.1 Method

We recruited nine participants by advertising on Facebook, LinkedIn, and Twitter and through flyers (referred to as P1-P9). Participants connected to our system from the US, Spain, England, and India. Each had prior experience in JavaScript. Participants included one undergraduate student in computer science or a related field (P5), one instructor (P9), and seven graduate students in computer science or related fields. As typical in open contribution platforms, participants exhibited a diverse range of experience, with prior experience in JavaScript including less than 6 months (P1 and P2), 7-12 months (P3, P4, and P5), and more than 4 years (P6, P7,P8, and P9).

We split our study into two sessions to reduce participant fatigue as well as to simulate participants returning to work after a delay, as is common in microtasking. All nine participants participated in the first session and five (P1, P5, P6, P8, and P9) participated in the second session. The first session was 150 minutes, and the second 120 minutes. One participant (P8) left the second session early after approximately one hour. All participants worked entirely online at their own computers, and their interactions with other participants were only via the *Question and Answer* feature. Participants were paid 20 dollars per hour for their time through gift cards.

The crowd worked to build a microservice for the back-end functionality of a Todo app. The microservice included functionality for creating, deleting, updating, fetching, reminding, and archiving of todo items. This functionality was described through 12 endpoints.

We gathered data from several sources. Before beginning the study, participants completed a short demographics survey. As participants worked, *Crowd Microservices* logged each microtask generated, submitted, and skipped as well as each change to a function or test, annotating each with a timestamp and participant id. At the end of the first session, participants completed a survey on their experiences with *Crowd Microservices*, and five of the participants participated in a short interview with the experimenter.

Microtasks types	Completed		Skipped		Median time (mm:ss)	
	Session 1	Session 2	Session 1	Session 2	Session 1	Session 2
Implement Function	112	63	39	22	4:12	3:27
Behavior Review	112	63	5	9	2:41	2:25
Total	224	126	44	31		
Overall Total	350		75			

Table 2 Microtask completions, skips, and completion times

At the beginning of the study, participants logged in to *Crowd Microservices* and worked through tutorial content by watching an initial tutorial video, reading the welcome page, and then reading a second series of 6 tutorials on using the individual microtasks. Participants then began work by fetching a

microtask. As in typical work contexts, participants were allowed to use Internet searches as they saw fit.

5.2 Results

5.2.1 Feasibility of behavior-driven microtasks

To investigate the ability of participants to use the behavior-driven microtask workflow, we 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 (Table 2). Participants created a test suite of 36 unit tests, writing an average of 3 unit tests per function. We analyzed the number of lines of code in each function and test, counting the final numbers of lines in each at the end of the study. Participants wrote 216 lines of code, approximately 16 lines per function. Participants wrote 397 line of code in their test suite.

Several participants reported that identifying a behavior was not difficult. Three participants reported that they preferred to focus on easy behaviors first:

> "I chose the easiest behavior to implement first. This was usually to check if the input was null or empty. If that was already implemented, I just went in order." - (P5)

Others reported that some behaviors were not clear, leading them to focus first on those which were:

> "I chose based on being more clear and simple to me. Sometimes it wasn't clear what exactly that behavior means." - (P8)

Interview at the end of study with 5 participants reveals the *leader-board* plays an important role in the motivation of the participants since they were watching the user-name of the other and their scores, they were trying to get more score to place in higher ranking than the other participants. The thing that helps participants to place in higher ranking is submitting more Review microtask and submitting *Implement Function behaviors* microtasks with higher quality which be accepted in *Review* microtask. The minimum points for the first session were 54 points and for the second session was 13 points, the maximum amounts for first and second session were 242 and 151 points. For the first session average of the score were 129.5 pints, for the second was 103.2 points.

Throughout both sessions, workers iteratively implemented and revised function implementations, reflecting contributions from several participants. Participants submitted 175 *Review* microtasks. In 82 of these, they accepted the contribution by giving it a rating of 3 or more stars. One participant reported that

> "The main challenges were there that some developers have left their changes only with some meaningless line of codes. A bad implementation is better than an incomplete implementation. " - (P7)

5.2.2 Speed of onboarding and contributing

During the two sessions, participants worked for a total of 31.5 hours. Participants spent 21 hours on microtasks that were submitted, including 39% of their time on *Implement Function behaviors* microtasks and 27% on *Review* microtasks. The remaining time was spent familiarizing themselves with the *Crowd Microservices* environment, completing the post-task survey, and working on microtasks that were skipped rather than submitted.

After participants completed the tutorials and began work, participants spent additional time familiarizing themselves with the environment. On average, participants submitted their first microtask after 24 minutes. In some cases, participants skipped multiple microtasks to find an easy one with which to begin. Participants skipped 17% of all microtasks. Participants skipped microtasks when they were not able to complete a microtask or when they ran out of time.

Participants were able to complete both types of microtasks in a short period of time (Table 2). The median completion time was approximately 4 minutes for *Implement Function Behavior* and 3 minutes for *Review* microtasks. These completion times are similar to the under 5 minute median completion times reported for a microtask workflow which organizes implementation work into different and separate editing, debugging, and testing microtasks[20]. Each *Implement Function Behavior* gathered all 4 separate microtasks in Microtask Programming [20] into one microtask, so we expect the median time be more, but it is not. In fact As shown in Fig. 3, in our workflow the implementation microtask (*Implement Function behaviors* microtask) works on behavior instead of works on all behaviors, it has smaller scope and context in comparison with the Microtask Programming [20].

5.2.3 Feasibility of implementing microservices

As it is shown in Step 3, Fig. 4 after the participants finished the implement of the microservice, it is deployed on a host by a command in the *Crowd Microservices*. To assess the feasibility of using a behavior-driven microtask workflow to implement microservices, we investigated the success of the crowd in building an implementation consistent with the described behavior in the client request. Although it is not required to run any more test by the client to evaluate the output of the crowd, we first constructed a unit test suite, generating a set of 34 unit tests, which is publicly available as part of our replication package[7]. We found that the unit tests for 79% (27) behaviors passed, and the unit tests for 7 of the behaviors failed. To investigate the

[7] https://github.com/devuxd/crowd-todo-microservices

reasons for these failures, we examined the implementation created by the participants. We found that four of the failures were due to a single defect in one function involving a missing conditional, and the three remaining failures were either due to defects with behaviors not implemented correctly or not implemented. After addressing these issues, we found that all of the unit tests passed.

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

Crowd Microservices offers an API for interacting with a persistence store, which participants made use of in their implementation. For example, in Fig.7, *fetchAllTodos* is an example of an invocation of the persistence API. Participants made a total of 15 calls to the persistence API, or 1.25 per function. In some cases, participants interacted with the persistence store indirectly, by calling other functions implemented by the crowd which made use of the persistence store. When asked in the post-task survey, most participants reported that they used the persistence API without any problems. Some participants reported that additional documentation would be beneficial:

> "I used the API a little bit, and I felt like the documentation could be better with more examples." - (P5)

6 LIMITATIONS AND THREATS TO VALIDITY

Our study had several limitations and potential threats to the validity of the results.

In our study, we chose to recruit a wide range of participants, recruiting participants locally from our university as well as globally through social networking sites. This yielded participants with a wide range of backgrounds, with their experience in JavaScript ranging from 2 months to 6 years. We chose this process as it mirrors the process of an open call, where contributors with a wide range of backgrounds may contribute. However, in practice crowdsourcing communities may exist in many forms, attracting many novice contributors looking for an entry point into more challenging work or attracting experienced workers who attract other experienced workers. Our results might differ if workers were exclusively more or less experienced.

Another potential threat to validity is the choice of task. In selecting a task, we sought to identify a task that is representative of typical microservices that workers create. We chose the *Todo* application as a canonical example of a web application, often used to compare different approaches to building web

applications[8]. Larger microservices may involve more complex endpoints where individual behaviors are more challenging to identify.

Our results might also vary with different contexts in which work took places. To simulate the constant process of hand-offs that occur in microtask work, where workers complete tasks that others began, we chose to have participants work synchronously, maximizing the number of hand-offs that occur. To simulate participants coming back to work that they had begun earlier, we divided our study into two sessions. Of course, in practice, microtask work involves less predictable schedules, where contributors may come and go at arbitrary times. This may introduce additional challenges, where new participants that are unfamiliar with either the environment or anything about the project are constantly introduced. On the one hand, this might reduce performance, as such participants are less experienced. On the other, compared to participants in our study who had access to no workers who were already familiar with the environment and project, such participants might have an easier time onboarding, as more experienced workers would be available to answer their questions. Better understanding the impact of transient behavior on microtasking in programming is an important focus for future work.

In microtask work, workers are often assumed to not have any prior information about the environment, the project, or other workers. However, in practice, over time participants do gain experience with the project and especially the environment. We would expect that over time a contributor to our environment would experience fewer challenges using the environment and become more productive, reducing the average time for completing microtasks or increasing the amount of work contributors complete in each microtask.

7 DISCUSSION

Microtask programming envisions a software development process in which large crowds of transient workers build software through short and self-contained microtasks, reducing barriers to onboarding and increasing participation in open source projects. In this paper, we explored a novel workflow for organizing microtask work and offered the first approach capable of microtasking the creation of web microservices. In our behavior-driven microtask workflow, each microtask involves a worker identifying, testing, implementing, and debugging an individual behavior within a single function. We 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.

[8] http://todomvc.com

```
router.get('/fetchTodosBasedOnStatus', async (req, res, next) => {

    var userId = req.query.userId;
    var status = req.query.status;
    if (userId == null || userId === "") {
        throw new TypeError('Illegal Argument Exception');
    }
    var result = [];
    var allTodos = fetchAllTodos(userId);
    if (allTodos.length > 0) {
        for (var i = 0; i < allTodos.length; i++) {
            if (!(allTodos[i].status !== 1 ||
                    allTodos[i].status !== 2 ||
                    allTodos[i].status !== 3)) {
                throw new TypeError("Illegal Argument Exception");
            }
            else {
                if (allTodos[i].status === status) {
                    result.push(allTodos[i]);
                }
            }
        }
    }
    res.json(result);
});
```

Fig. 7 An example of a microservice endpoint implemented by the crowd.

While our method has demonstrated success in implementing an individual microservice, there remain a number of additional challenges to address before a large software project could be built entirely through microtasks. In our method today, the client performs the high-level design tasks of determining the endpoints, data structures, and other design work. Moreover, while microtasking reduces the context a worker must learn to successfully contribute to a project, this context is not zero. Workers must still learn about the function they are working on and the current state of its implementation. This overhead is visible in the productivity of work. In 21 hours working on submitted microtasks, 9 participants wrote only 216 lines of code and 397 lines of test code.

Other work has explored techniques for decomposing design work into microtasks, such as through structuring work around tables of design dimensions and design alternatives[38]. Such techniques might be adapted to a microtask programming workflow to enable the crowd to design the initial endpoints and data structures, as well as other high-level decisions, which might then be handed off to others who then implement this design. Similar to workflows such as TopCoder's, a senior crowd worker might also help ensure consistency across the design. Beyond upfront design, support is also necessary for maintenance situations where requirements change. This might result in changes in endpoints, data structures, and design decisions, requiring further downstream changes in the implementation. Such challenges might be addressed

through new types of microtasks which identify changes, map these changes to specific impacted artifacts, and ask workers to update the corresponding implementation.

A key advantage of microtask programming approaches which incorporate automatic microtask generation is in enabling the potential for scale. Rather than requiring a single worker acting as a client to manually generate each microtask, microtasks are generated automatically as the crowd works. Rather than potentially exposing contributors to the entire codebase and all of its ongoing changes, contributors must only understand an individual function. In this way, in principle, large crowds might be able to work together to build large applications quickly. For example, if a microservice ultimately resulted in 1,000 behaviors being identified, each behavior could then be worked on by a separate worker in a separate microtask. To the extent that these 1,000 microtasks can be done in parallel, this would then enable software development to occur with 1,000 concurrent microtasks, dramatically decreasing the time to complete work. Of course, many sequential dependencies might still exist, where, for example, the necessary existence of a function is not revealed until a previous function has already been implemented. Understanding just how many sequential dependencies exist in software development work and how much parallelism is truly possible is thus an important focus for future work.

8 CONCLUSION

In this paper, we described an approach for implementing microservices through microtasks. Our results demonstrate initial evidence for feasibility, demonstrating that workers could successfully complete a microtask which asked them to identify, test, implement, and debug a behavior in less than 5 minutes as well as onboard onto the project in less than 30 minutes. Our results also offer the first example of a microservice implemented entirely through microtask contributions. Important future work remains to investigate how this approach might be incorporated into a larger software project as well as exploring how a higher degree of parallelism might reduce the time to market in building software.

Acknowledgements We thank the participants in the study for their participation. This work was supported in part by the National Science Foundation under grant CCF-1414197.

References

1. Beck, K.: Test-driven development: by example. Addison-Wesley Professional (2003)
2. Bernstein, M.S., Little, G., Miller, R.C., Hartmann, B., Ackerman, M.S., Karger, D.R., Crowell, D., Panovich, K.: Soylent: a word processor with a crowd inside. In: Symposium on User Interface Software and Technology, pp. 313–322 (2010)
3. Bounov, D., DeRossi, A., Menarini, M., Griswold, W.G., Lerner, S.: Inferring loop invariants through gamification. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, pp. 231:1–231:13 (2018)

4. Chen, N., Kim, S.: Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles. In: Automated Software Engineering, pp. 140–149 (2012)
5. Chen, Y., Lee, S.W., Xie, Y., Yang, Y., Lasecki, W.S., Oney, S.: Codeon: On-demand software development assistance. In: Conference on Human Factors in Computing Systems, pp. 6220–6231 (2017)
6. Fagerholm, F., Guinea, A.S., Borenstein, J., Münch, J.: Onboarding in open source projects. IEEE Software pp. 54–61 (2014)
7. Glassman, E.L., Fischer, L., Scott, J., Miller, R.C.: Foobaz: Variable name feedback for student code at scale. In: Symposium on User Interface Software and Technology, pp. 609–617 (2015)
8. Glassman, E.L., Scott, J., Singh, R., Guo, P.J., Miller, R.C.: Overcode: Visualizing variation in student solutions to programming problems at scale. Transactions on Computer-Human Interaction pp. 1–7 (2015)
9. Goldman, M.: Software development with real-time collaborative editing. Ph.D. thesis, Massachusetts Institute of Technology (2012)
10. Goldman, M., Little, G., Miller, R.C.: Real-time collaborative coding in a web ide. In: Symposium on User Interface Software and Technology, pp. 155–164 (2011)
11. Guo, P.J.: Codeopticon: Real-time, one-to-many human tutoring for computer programming. In: Symposium on User Interface Software and Technology, pp. 599–608 (2015)
12. Hoseini, M., Saghafi, F., Aghayi, E.: A multidimensional model of knowledge sharing behavior in mobile social networks. Kybernetes (2018)
13. Jergensen, C., Sarma, A., Wagstrom, P.: 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, pp. 70–80 (2011)
14. Jiang, H., Matsubara, S.: Efficient task decomposition in crowdsourcing. In: International Conference on Principles and Practice of Multi-Agent Systems, pp. 65–73 (2014)
15. Kim, J., Sterman, S., Cohen, A.A.B., Bernstein, M.S.: Mechanical novel: Crowdsourcing complex work through reflection and revision. In: Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (2017)
16. Kittur, A., Nickerson, J.V., Bernstein, M., Gerber, E., Shaw, A., Zimmerman, J., Lease, M., Horton, J.: the future of crowd work. In: Conference on Computer Supported Cooperative Work, pp. 1301–1318 (2013)
17. Kittur, A., Smus, B., Khamkar, S., Kraut, R.E.: Crowdforge: Crowdsourcing complex work. In: Symposium on User Interface Software and Technology, pp. 43–52 (2011)
18. Lasecki, W.S., Kim, J., Rafter, N., Sen, O., Bigham, J.P., Bernstein, M.S.: Apparition: Crowdsourced user interfaces that come to life as you sketch them. In: Human Factors in Computing Systems, pp. 1925–1934 (2015)
19. LaToza, T.D., Chen, M., Jiang, L., Zhao, M., Van Der Hoek, A.: Borrowing from the crowd: A study of recombination in software design competitions. In: International Conference on Software Engineering, pp. 551–562 (2015)
20. LaToza, T.D., Di Lecce, A., Ricci, F., Towne, B., Van der Hoek, A.: Microtask programming. Transactions on Software Engineering pp. 1–20 (2018)
21. LaToza, T.D., Lecce, A.D., Ricci, F., Towne, W.B., van der Hoek, A.: Ask the crowd: Scaffolding coordination and knowledge sharing in microtask programming. In: Symposium on Visual Languages and Human-Centric Computing, pp. 23–27 (2015)
22. LaToza, T.D., Towne, W.B., Adriano, C.M., Van Der Hoek, A.: Microtask programming: Building software with a crowd. In: Symposium on User Interface Software and Technology, pp. 43–54 (2014)
23. LaToza, T.D., Towne, W.B., van der Hoek, A., Herbsleb, J.D.: Crowd development. In: Workshop on Cooperative and Human Aspects of Software Engineering, pp. 85–88 (2013)
24. Lee, S.W., Krosnick, R., Park, S.Y., Keelean, B., Vaidya, S., O'Keefe, S.D., Lasecki, W.S.: Exploring real-time collaboration in crowd-powered systems through a ui design tool. Computer-Supported Cooperative Work and Social Computing pp. 104:1–104:23 (2018)
25. Lerner, S., Foster, S.R., Griswold, W.G.: Polymorphic blocks: Formalism-inspired ui for structured connectors. In: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, pp. 3063–3072 (2015)

26. Li, K., Xiao, J., Wang, Y., Wang, Q.: Analysis of the key factors for software quality in crowdsourcing development: An empirical study on topcoder. com. In: Computer Software and Applications Conference, pp. 812–817 (2013)
27. Mao, K., Capra, L., Harman, M., Jia, Y.: A survey of the use of crowdsourcing in software engineering. Journal of Systems and Software **126**, 57 – 84 (2017)
28. Nebeling, M., Leone, S., Norrie, M.C.: Crowdsourced web engineering and design. In: Web Engineering, pp. 31–45 (2012)
29. Newman, S.: Building microservices: designing fine-grained systems. " O'Reilly Media, Inc." (2015)
30. North, D.: Introducing behaviour driven development. Better Software Magazine (2006)
31. Retelny, D., Bernstein, M.S., Valentine, M.A.: No workflow can ever be enough: How crowdsourcing workflows constrain complex work. Conference on Computer-Supported Cooperative Work and Social Computing pp. 1–23 (2017)
32. Saengkhattiya, M., Sevandersson, M., Vallejo, U.: Quality in crowdsourcing-how software quality is ensured in software crowdsourcing. Master's thesis, Department of Informatics, Lund University (2012)
33. Schiller, T.W., Ernst, M.D.: Reducing the barriers to writing verified specifications. Special Interest Group on Programming Languages Notices pp. 95–112 (2012)
34. Steinmacher, I., Silva, M.A.G., Gerosa, M.A., Redmiles, D.F.: A systematic literature review on the barriers faced by newcomers to open source software projects. Information and Software Technology pp. 67 – 85 (2015)
35. Stol, K.J., Fitzgerald, B.: Two's company, three's a crowd: A case study of crowdsourcing software development. In: Conference on Software Engineering, pp. 187–198 (2014)
36. Von Krogh, G., Spaeth, S., Lakhani, K.R.: Community, joining, and specialization in open source software innovation: a case study. Research Policy pp. 1217–1241 (2003)
37. Warner, J., Guo, P.J.: Codepilot: Scaffolding end-to-end collaborative software development for novice programmers. In: Conference on Human Factors in Computing Systems, pp. 1136–1141 (2017)
38. Weidema, E.R.Q., López, C., Nayebaziz, S., Spanghero, F., van der Hoek, A.: Toward microtask crowdsourcing software design work. In: Workshop on CrowdSourcing in Software Engineering, pp. 41–44 (2016)
39. Zanatta, A.L., Machado, L., Steinmacher, I.: Competence, collaboration, and time management: Barriers and recommendations for crowdworkers. In: Workshop on Crowd Sourcing in Software Engineering, pp. 9–16 (2018)