Empirical Software Engineering (2023) 28:10
https://doi.org/10.1007/s10664-022-10226-2

A controlled experiment on the impact of microtasking on programming

Emad Aghayi[1] · **Thomas D. LaToza**[1]

Accepted: 18 August 2022 / Published online: 25 November 2022

Abstract

In *microtask programming*, developers complete short self-contained microtasks through the use of a specialized programming environment. For example, given only a short description of the purpose of a function and a partially complete implementation, a developer might be asked to identify, test, and implement an additional behavior in the function. Adopting a microtask approach to programming tasks has been envisioned to offer a number of potential benefits, including reducing the onboarding time necessary for new developers to contribute to a project and achieving higher project velocity by enabling larger project teams and greater parallelism. To investigate the potential benefits and drawbacks of *microtask programming* we conducted a controlled experiment. We focused our investigation on the context in which microtasking is most widely used, implementing and debugging function bodies, and investigated the impact of microtasking with respect to onboarding, project velocity, code quality, and developer productivity. 28 developers worked to implement microservice endpoints, either in the form of traditional programming tasks described in an issue tracker or as programming microtasks. Our study did not examine the design effort necessary to prepare for microtask programming or consider how microtask programming might be applied to maintenance tasks. We found that, 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. Through qualitative analysis of how developers worked, we explore potential explanations of these differences. These findings offer evidence for the potential benefits that adopting microtask programming might bring, particularly in cases where increasing project velocity is paramount.

Keywords Crowdsourcing · Software development · *Microtask programming* · Evaluation

Communicated by: Igor Steinmacher

✉ Emad Aghayi
eaghayi@gmu.edu

Thomas D. LaToza
tlatoza@gmu.edu

[1] Department of Computer Science, George Mason University, Fairfax, VA 22030, USA

1 Introduction

Inspired by microtasking systems for other tasks (Retelny et al. 2017; Kittur et al. 2011; Bernstein et al. 2010; Hoseini et al. 2018; Kittur et al. 2013; Jiang and Matsubara 2014), *microtask programming* envisions a form of work in which new contributors join a software project and begin contributing within a short amount of time by completing programming microtasks (LaToza et al. 2018; LaToza et al. 2014). A microtask is a short, self-contained unit of work with a specific objective. Microtask programming divides individual tasks to implement a use case or fix a bug into numerous smaller microtasks.

Microtasks differ from traditional software issues, as might be found in an issue tracker, along three dimensions: the time they take to complete, how much context they require to do, and the specificity of the objective. Microtasks differ from traditional tasks in taking less time, requiring less context, and having more specific objectives. A traditional task in a project issue tracker (e.g., fixing a defect, implementing a new feature) might take hours to complete, whereas a microtask may be completed in under 15 minutes (LaToza et al. 2018; Aghayi et al. 2021). To fix a defect in a traditional project, a developer might reproduce the defect by manually running the program, inserting log or debug statements throughout the program, rerunning the program, examining the output, reading various files in code, proposing a fix, investigating the implications of this fix on various parts of the code, and then eventually implementing and testing the fix. In contrast, microtasks have much more specific objectives than this. For example, a microtask might involve identifying a behavior in a specific function, writing a few lines of code to implement it, and then testing it. Finally, in traditional projects, developers need to gain considerable prior knowledge before starting work on tasks. For example, developers need to understand architecture and design of the codebase, understanding how the defect that they are working to fix interact with these and, as a result, what functionality is likely to be relevant, what source files this functionality is implemented in, and the ways in which this is implemented (Wang and Sarma 2011; Steinmacher et al. 2015; Von Krogh et al. 2003; Jergensen et al. 2011; Fagerholm et al. 2014; Britto et al. 2020). In a microtask, developers need to know none of this context. Instead, developers may be given a single function. As part of the microtask itself, there is a description of the function, describing everything about the function a developer must know. In this way, the context of a microtask is greatly reduced from the context required in a traditional task.

Microtask programming adopts three core mechanisms for organizing work and motivating and supporting contributors. *Microtask programming* reduces the effort required to (1) onboard developers onto a project, including understanding the project structure and code and setting up a workspace (Steinmacher et al. 2015; Wang and Sarma 2011; Britto et al. 2020; Von Krogh et al. 2003; Fagerholm et al. 2013; Fagerholm et al. 2014). To reduce this traditionally lengthy process, developers work within a specialized preconfigured programming environment (Aghayi et al. 2021; Lasecki et al. 2015; LaToza et al. 2018; LaToza et al. 2015; Goldman et al. 2011a; Goldman 2012; Nebeling et al. 2012; Schiller and Ernst 2012; LaToza et al. 2014; LaToza et al. 2013; Williams et al. 2019). Preconfigured environments offer an environment in which code and dependencies are already set up and developers may immediately begin editing code.

Microtasking changes the nature of (2) coordination, transforming tasks done by a single developer into tasks done by multiple contributors. Whereas a single developer might work to prepare a pull request for a new feature, in *microtask programming* a number of crowd workers may contribute. Workers adopt partially completed work, continuing or revising it based on their own plan, and continuously offer feedback, through reviews, on each contribution made.

To incentivize the broader participation that microtasking enables, microtasking systems adopt (3) gamification mechanisms (Zanatta et al. 2018). Systems may measure fine-grained progress, recording each microtask completion. From this, they provide feedback at each step (e.g., review comments) as well as publicly visible signals of contributions (e.g., points on a leaderboard).

Microtasking programming mechanisms are interdependent, each changing the nature of programming work in ways that require new techniques to support. For example, by offering continuous feedback (3), the impact of the reduced context and knowledge developers have from a shortened onboarding experience may be reduced (1). And by changing the mechanisms through which coordination occurs (2), the impact of making tasks short and self-contained may be reduced.

Through these mechanisms, *microtask programming* has been envisioned to offer important potential benefits for software development (LaToza et al. 2018; LaToza and van der Hoek 2016). First, by reducing the context necessary to program from a whole codebase to an individual artifact and offering developers a preconfigured environment, *microtask programming* may reduce onboarding barriers. Second, by deconxtextualizing, decomposing, and reorganizing programming work, *microtask programming* is envisioned to enable programming tasks to be divided and shared among developers, enabling increased team size to increase project velocity.

At the same time, *microtask programming* may also bring new challenges managing contributor knowledge and awareness of specific code modules. Developers work without awareness of the complete program, potentially increasing the potential for work to go off track. Conflicts might occur, either from two overlapping changes to the same artifacts, or from conflicting changes in different artifacts. Lacking a global view of the codebase, developers may write lower quality code, as they are unaware of code with which to be consistent. By asking developers to rapidly switch between working on microtasks which focus on different artifacts within a project, developers may spend more time understanding new code and less time programming.

Existing studies of *microtask programming* have demonstrated its feasibility, showing that it is possible for it to be used to create small programs (Aghayi et al. 2021; LaToza et al. 2018; LaToza et al. 2013), user interfaces (Lasecki et al. 2015; Nebeling et al. 2012; Lee et al. 2018), tests,[1] and other artifacts (Goldman et al. 2012; Goldman et al. 2011b; Chen et al. 2016). Yet we are aware of no prior work that has offered a direct comparison of *microtask programming* to traditional programming.

We conducted a controlled experiment to answer four research questions:

– RQ1: How does *microtask programming* impact onboarding?
– RQ2: How does parallelism in *microtask programming* impact velocity?
– RQ3: How does *microtask programming* impact code quality?
– RQ4: How does *microtask programming* impact developer productivity?

Table 2 summarizes the definitions of the constructs and measures used to answer these questions. In this paper, we define onboarding time as the time it takes for developers to

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

join a project and complete their first contribution. We measured this by tracking the time developers required by developers to begin contributing and submit their first task (Fig. 2). We define project velocity as the amount of programming work that a development team completes per unit of time. We measured project velocity both as the average number of lines of code per and the number of correctly implemented behaviors completed by developers in a 4 hour programming session. 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. We measured code quality by asking a panel of experienced JavaScript developers to assess code for its clarity, simplicity, and consistency. We define developer productivity as the amount of programming work an individual developer completes in a unit of time. We measured developer productivity as the number of lines of code and correctly implemented behaviors completed per hour.

28 developers were randomly assigned to one of two conditions. All worked in 4 hours sessions to implement and debug function bodies of a small microservice in JavaScript for an online-shopping-application and then completed a post-task survey at the conclusion of the study. In the control condition, 14 developers worked individually in a traditional Integrated Development Environment (IDE) to complete issues described in an issue tracker. In the *microtask programming* condition, developers worked together as part of 7-person crowds to complete microtasks in a dedicated *microtask programming* environment. We recorded and analyzed developers' activity by collecting screencasts. We measured the time developers worked and assessed their output through a hidden unit test suite as well as through a panel of four anonymous reviewers.

We found that *microtask programming* significantly reduced onboarding time from 164 minutes to 44 minutes, a factor of 3.7. In increasing team size from an individual developer to a crowd of 7 developers, project velocity increased by a factor of 5.76. As rated by a panel of developers anonymous to condition, the quality of code written did not significantly differ. The productivity of individual developers in the *microtask programming* condition decreased by a factor of 1.3.

In the rest of this paper, we survey related work, introduce *microtask programming*, describe the study design, and report the results. Finally, we conclude with a discussion of limitations as well as opportunities and future directions.

2 Related Work

Our investigation of *microtask programming* builds on a number of prior studies of software development and crowdsourcing. In particular, a number of studies have investigated the challenges developers face in onboarding (RQ1), determinants of velocity for software projects (RQ2), the quality of code created in software projects (RQ3), and the productivity impacts of microtasking complex information work (RQ4).

Crowdsourced software engineering is the undertaking of any software engineering tasks by an undefined, potentially numerous, set of online workers recruited through an open call (Mao et al. 2017). TopCoder (Lakhani et al. 2010) and open-source software (OSS) development are two examples of models of crowdsourcing software development (LaToza and van der Hoek 2016). Crowdsourcing software engineering envisions the potential to reduce the time to market, generate alternative solutions, utilize specialists, learn via work, and democratize participation in software engineering (LaToza and van der Hoek 2016).

Software developers, including in OSS development, face barriers that require completing lengthy joining scripts to begin contributing (Von Krogh et al. 2003). This may dissuade the busy or casually committed from contributing and restricts the pool of millions of potential contributors to only the most committed. One solution to reducing these barriers comes in the form of microtask programming. Microtask programming decontextualizes the tasks done by workers, enabling a contribution to be made in isolation from other ongoing work and with no requirements for prior knowledge (LaToza et al. 2018).

Microtask programming is a specific form of crowdsourced software engineering, utilizing an open call to contribute through a specific form: the microtask. Microtasks are tasks which are short, self-contained, and with a specific objective. For example, in Apparition (Lee et al. 2018), developers are given a highly specific task (to craft visual behavior for a specific user interface element in a few lines of code) and may complete this task in a few minutes. In contrast, other forms of crowdsourcing work utilize tasks which are longer, not nearly as self-contained, and with broader objectives. For example, on TopCoder, developers may be asked to participate in a design competition. Developers are given a set of requirements and asked to create a series of UML diagrams. Developers have access to the whole codebase. And the competition may last for weeks. In this way, while microtask programming systems are examples of crowdsourcing, other forms of crowdsourcing are not microtasking.

When new developers join a software project, they face a number of onboarding barriers. Onboarding barriers include 1) identifying appropriate contacts and receiving timely feedback, 2) identifying proper tasks and corresponding artifacts, 3) understanding project structure and setting up a workspace, 4) outdated and unclear documentation 5) learning project practices and technical expertise (Wang and Sarma 2011; Steinmacher et al. 2015; Von Krogh et al. 2003; Jergensen et al. 2011; Fagerholm et al. 2014; Britto et al. 2020). Onboarding barriers impose a lengthy joining script that dissuades less motivated potential contributors from becoming a contributor (Steinmacher et al. 2016). These joining barriers are also an issue for traditional software development projects, requiring a variety of practices to onboard new software development. For instance, companies such as Google have extensive mentoring programs that new employees participate in when joining (Johnson and Senges 2010). However, even after four months, developers have only shallow knowledge of a software project (Sim and Holt 1998) and may require as much as three years to become fluent (Zhou and Mockus 2010). However, as the average turnover rate at companies such as Amazon and Google is around two years (Peterson 2017), many developers will never be fully onboarded.

To address these barriers, a number of programming environments have been designed to reduce one or more of these barriers, often as part of a *microtask programming* environment where quick onboarding onto a project is crucial. These environments often offer dedicated, preconfigured, environments, which require much less setup for developers to get started in a new project (last column of Table 1) (Warner and Guo 2017; Lasecki et al. 2015; LaToza et al. 2018; Goldman et al. 2011a; Goldman 2012; Nebeling et al. 2012; Schiller and Ernst 2012; Chen et al. 2017; Aghayi et al. 2021).

The velocity of a software project is the rate of progress of the project team. Velocity measures the amount of work that a project team completes (e.g., a number of requirements or user stories) in a unit of time (e.g., a sprint). From this definition, it might be reasonable to expect that, if instead of employing a single developer, a project were to instead employ several developers, velocity might increase (Perry et al. 2001). However, achieving this requires overcoming several challenges, including successful coordination, communication among team members, and orchestration of work (Espinosa et al. 2001). The team lead or software architect must split work into independent tasks so that developers do not interfere with each other's efforts. Managing and coordinating parallel development work

Table 1 Examples of *microtask programming* Systems

	Goal	Crowd activities	Client activities	Microtask context	Quality control mechanisms	Pre-configured IDE
Crowd Microservices	Build microservice	Develop, debug, and test	Define endpoints and ADTs	A description, function and unit tests	Unit tests, review by crowd	Yes
CrowdCode (LaToza et al. 2018)	Build library	Develop, debug, and test	Specify core functionsand ADTs	A function or unit test	Unit tests, reviews by crowd	Yes
Apparition (Lee et al. 2018)	Build UI prototype	Design UI	Designers sketch UI describe it interactively	Whole design	Reviews by designers	Yes
CrowdDesign (Nebeling et al. 2012)	Build web-app	Design UI elements	Designers assemble UI elements	A statement to a module	Reviews by manager	Yes
CodeOn (Chen et al. 2017)	Help in programming	Develop and explain code	Developers request help aloud	A statement to whole codebase	Reviews by requesters	Yes
Mercury (Williams et al. 2019)	Review code and sources on-the-go	Explore code, Review code	None	A description, function and Q/A resource	None	Yes

is hard (Espinosa et al. 2001). One study of 222 open source projects revealed that the interdependence and distribution of teams are key factors in increasing conflict levels (Filippova and Cho 2016). In many software projects, increasing the degree of parallelism is hard or impossible due to task inter-dependencies and the context required by tasks (Perry et al. 2001). Microtask approaches to programming have been explored to address some of these issues, enabling a higher degree of parallelism (Lasecki et al. 2015; Goldman et al. 2011b; Goldman et al. 2012; LaToza and Van Der Hoek 2015; Nebeling et al. 2012). Achieving this requires effective mechanisms for distributing work, managing context, generating microtasks, and minimizing conflicts.

A number of studies have examined the quality achieved by software projects, specifically the quality achieved by open source projects. There are several systematic approaches for evaluating the quality of software (Tian 2005; Ammann and Offutt 2016). Studies comparing the quality of code produced in open source projects have found that it is lower than that what is expected in industrial projects (Stamelos et al. 2002; Basili 2001; Aberdour 2007). There are several potential reasons, including the lack of a formal risk assessment process, defect discovery from black-box testing late in the process, unstructured and informal testing, and the quality assurance methodologies used (Aberdour 2007). Some open source projects do not use systematic quality assurance methods (Zhao and Elbaum 2000). One study of over 20,000 Open-source software (OSS) projects found that 38% lacked unit tests (Kochhar et al. 2013).

Measuring and even defining productivity in software engineering is challenging. The Oxford dictionary defines productivity as "the rate at which a worker, a company or a country produces goods, and the amount produced, compared with how much time, work and money is needed to produce them" (University, O.: productivity noun. shorturl.at/gqT69). Measuring the productivity of software developers is challenging as the output created by a developer in a development task can be hard to quantify (Meyer et al. 2014; Kersten and Murphy 2006). Software engineering researchers have not reached agreement on how to accurately measure productivity (Oliveira et al. 2017). Many studies have identified factors that influence developers' productivity, such as scope and schedule (Jensen 2014; Kamma and Jalote 2013; Maxwell 2003; Oliveira et al. 2017). Mercury (Williams et al. 2019) a mobile microtasking system that allows programmers to continue their work on-the-go is introduced and evaluated by a controlled experiment. Micro-producitivities in Mercury help programmers continue their work on-the-go and instill comfort in pausing work unexpectedly.

A number of studies have investigated the impact of microtasking on the productivity of information workers completing non-programming tasks (Iqbal et al. 2018). Microtasking approaches which offer concrete plans with actionable steps can enable workers to complete tasks with a higher level of productivity (Teevan et al. 2016; Iqbal et al. 2018). Microtasking may result in more frequent task switches, as workers constantly switch task when beginning each microtask (LaToza et al. 2018). However, workers in general often face interruptions and may multi-task and attempt to complete several tasks simultaneously (Czerwinski et al. 2004; Cutrell et al. 2000). Reaching full productivity after an interruption may take 25 minutes (Mark et al. 2005). While interruptions may decrease the productivity of workers on large tasks, they have less impact on a worker that completes the same task as a series of microtasks (Cheng et al. 2015). This is because interruptions which occur at task boundaries are less disruptive and microtasks introduce more task boundaries. In addition, more of the context needed to resume work is contained in the microtask itself (Trafton et al. 2003; Iqbal and Bailey 2008).

Other work has begun to investigate the impact of the use of crowdsourcing within software projects (Lakhani et al. 2010; Stol and Fitzgerald 2014). Several studies have examined the use of TopCoder, one of the most mature crowdsourcing platforms, to build software projects. One study compared TopCoder's software development process with conventional software development, finding that TopCoder had a lower defect rate (5 to 8 times lower) at lower cost and in less time (Lakhani et al. 2010). TopCoder claimed that their crowdsourced development approach reduced cost by 30%-80% compared with in-house software development or outsourcing (Mao et al. 2017). Crowdsourcing platforms such as TopCoder differ from microtasking in the granularity of tasks (LaToza and van der Hoek 2016), where tasks are far more larger and require days rather than minutes to complete (Saito et al. 2020; Aghayi 2020). Our work builds on these studies, offering the first study specifically comparing a microtask style of programming to traditional programming.

3 Microtask Programming

Microtask programming is a form of crowdsourcing programming where work is carried out in the form of *microtasks*. In a microtask, a transient crowd worker is given a short and self-contained task (e.g., label an image, write a one-sentence summary of a paragraph), and individual contributions are then aggregated together to create a more extensive product (e.g., a labeled dataset of images, a summary of a news article) (Doan et al. 2011; Weld et al. 2008; Von Ahn and Dabbish 2004; Ramakrishnan et al. 2004).

Several systems have applied microtasking to programming, devising mechanisms to decompose traditional programming tasks of implementing features or fixing defects into microtasks (Table 1). For example, Apparition breaks down the work of building a prototype user interface into microtasks (Lee et al. 2018). As a designer describes an interface in text, crowd workers implement the behavior of individual UI elements. In CodeOn, a developer working in a programming environment speaks a request for help, automatically generating a microtask for crowd workers to complete capturing relevant context (Chen et al. 2017). In CrowdDesign, crowd workers may build a web app (Nebeling et al. 2012). In Mercury (Williams et al. 2019), a programmer works individually to continue work on-the-go. All microtasks are completed in a mobile setting. Mercury helps programmers with individual productivity. It focuses on task resumption instead of the broader aspect of productivity. In CrowdCode, programming work is completed through a series of specialized microtasks (LaToza et al. 2014; LaToza et al. 2018). Workers write test cases, implement tests, write code, reuse existing functions, and debug. In CrowdMicroservices, crowd workers identify unimplemented behaviors from a description of a function which they then test, implement, and debug (Aghayi et al. 2021). While these systems vary in a number of important aspects, they share several important characteristics central to the experience of microtasking: a streamlined onboarding process, an increase in the degree of parallelism in work, mechanisms for ensuring quality, and a changed programming experience.

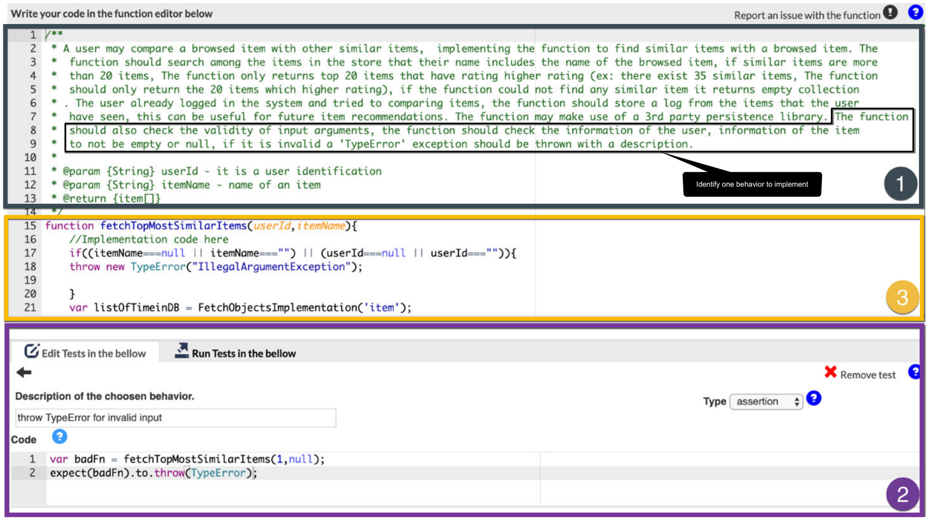
Core to the experience of microtasking systems is the idea of the crowd contributor, who can receive a microtask and complete it without needing to first complete a lengthy onboarding process (RQ1). Studies of onboarding find that this can be a substantial barrier (Steinmacher et al. 2015). *Microtask programming* environments envision reducing onboarding through several mechanisms, including offering a preconfigured environment and a self-contained programming task (Aghayi et al. 2021; LaToza et al. 2018). By offering a preconfigured environment, developers need not spend time installing and configuring necessary tools, downloading code from a server, identifying and downloading dependencies, and configuring the build environment. *Microtask programming* reduces the need to spend time understanding a codebase before contributing by decontextualizing programming tasks (Aghayi et al. 2021; LaToza et al. 2018).

Small microtasks bring the possibility of increasing parallelism in software development (RQ2). By parallelizing work across many developers, microtasks may enable work to be completed in less time. Key to achieving this is effective mechanisms for coordination and aggregation. This enables workers to obtain feedback on their contributions early, before wasting time on dead-end contributions, and to coordinate contributions to reduce conflicts. For example, Apparition incorporates a todo list and locks access to artifacts by other workers when a microtask is in progress to reduce conflicts (Lee et al. 2018). CodeOn facilities coordination by ensuring workers always work with the most recent code.

Microtask programming approaches use a variety of techniques to ensure the quality of the resulting software artifacts (RQ3). Low quality contributions may occur due to several reasons, including workers who do not have sufficient knowledge, workers who put in little effort, or workers who are malicious (Kim et al. 2017). Commercial crowdsourcing approaches address these quality concerns through mechanisms including rankings and ratings, reporting spam, pre-approving tasks, and skill filtering (Saengkhattiya et al. 2012). CrowdCode and CrowdMicroservices employ unit tests, code reviews, and gamification techniques to ensure quality (LaToza et al. 2018; Aghayi et al. 2021). In CrowdDesign, a manager reviews the contributions of crowd workers and accepts or rejects them (Nebeling et al. 2012).

In decontextualizing work and requiring developers to constantly switch their focus between different artifacts, *microtask programming* may potentially reduce developer productivity (RQ4). As workers rapidly switch between microtasks, developers may spend more time understanding specifications and code and less time programming. For example, in CrowdMicroservices, workers may read a new function specification, unit tests, and implementation at the beginning of every microtask.

In our study, we chose to examine the *CrowdMicroservices* system (Aghayi et al. 2021) as an example of a *microtask programming* approach (Fig. 1). In *CrowdMicroservices*, a


Write your code in the function editor below
Report an issue with the function
Identify one behavior to implement
1
3
Edit Tests in the bellow
Run Tests in the bellow
Remove test
2

```
/**
 * A user may compare a browsed item with other similar items,  implementing the function to find similar items with a browsed item. The
 *  function should search among the items in the store that their name includes the name of the browsed item, if similar items are more
 *  than 20 items, The function only returns top 20 items that have rating higher rating (ex: there exist 35 similar items, The function
 *  should only return the 20 items which higher rating), if the function could not find any similar item it returns empty collection
 * . The user already logged in the system and tried to comparing items, the function should store a log from the items that the user
 *  have seen, this can be useful for future item recommendations. The function may make use of a 3rd party persistence library. The function
 *  should also check the validity of input arguments, the function should check the information of the user, information of the item
 *  to not be empty or null, if it is invalid a 'TypeError' exception should be thrown with a description.
 *
 * @param {String} userId - it is a user identification
 * @param {String} itemName - name of an item
 * @return {item[]}
 */
function fetchTopMostSimilarItems(userId,itemName){
    //Implementation code here
    if((itemName===null || itemName==="") || (userId===null || userId==="")){
    throw new TypeError("IllegalArgumentException");

    }
    var listOfTimeinDB = FetchObjectsImplementation('item');
```

Description of the choosen behavior.

throw TypeError for invalid input

Type: assertion

Code

```
var badFn = fetchTopMostSimilarItems(1,null);
expect(badFn).to.throw(TypeError);
```

Fig. 1 In the CrowdMicroservices *microtask programming* system, developers complete microtasks in which they (1) identify a behavior from the description of a function, (2) write a unit test for the behavior, and (3) implement and debug code for the behavior

client, for example, a software design team, first defines a desired behavior of a microservice by authoring textual descriptions of endpoints. Contributors then log in to a web-based programming environment, view tutorial content explaining the environment, a project summary, and list of endpoints and their description. Developers fetch microtasks, which are assigned at random, and complete two types of microtasks. In *Implement Function Behavior*, developers follow a behavior-driven development methodology to (1) identify a single behavior from the description of a function that is not yet implemented, (2) write a unit test for the behavior, and (3) implement and debug the behavior (Fig. 1). To ensure that developers do not lock contributions to a function by taking too long to complete a microtask, developers have 15 minutes to submit the microtask; otherwise, the *microtask programming* environment skips the microtask. In the *Review* microtask, developers examine a microtask completed by another developer, review a diff of the code change, give feedback on the microtask, and assign it a rating to accept or reject it. To support gamification, contribution ratings are then used to generate a score for each developer. This score is publicly visible to the entire project crowd on a *leaderboard*. After all microtasks have been completed, a client may *publish* the microservice, automatically deploying the microservice to a hosting provider.

4 Method

The goal of our study was investigating the impact of microtasking on programming, focusing on the effects of working as part of a crowd on short, decontextualized, self-contained microtasks as compared to traditional individual programming work. Specifically, we investigated the effect of microtasking on onboarding (RQ1), project velocity (RQ2), code quality (RQ3), and developer productivity (RQ4). To investigate our research questions, we conducted a controlled experiment where participants implemented and debugged function bodies of a small JavaScript microservice. To compare *microtask programming*, as embodied in *microtask programming* environments, to traditional programming, we varied several aspects between each condition. As *microtask programming* decomposes a long task into a number of short tasks which are completed by a crowd, developers in the control condition worked individually, while developers in the *microtask programming* condition worked with other developers as part of a crowd. As *microtask programming* offers specialized programming environments to ease onboarding for developers, developers in the *microtask programming* condition were furnished with a specialized *microtask programming* environment, while developers in the control condition worked with their own preferred IDE and were responsible for onboarding activities onto the project.

We recruited 28 participants and randomly assigned them to a control or experimental condition. All worked in 4 hour sessions to implement a small microservice in JavaScript for an online-shopping application. In the experimental condition, participants worked using programming microtasks. Experimental participants were organized into two sessions. In each session, 7 participants worked together simultaneously as a crowd. In the control condition, 14 participants each worked individually, isolated from other participants without any coordination or exchange of contributions. The study consisted of three main parts: tutorials, a programming task, and a post-task survey. During the study sessions, we recorded and collected screencasts as well as collecting the code created in each session. We then evaluated the code produced through a hidden test suite as well as a panel of reviewers anonymous to conditions. From this data, we calculated several measures to examine the impact of *microtask programming* on onboarding, velocity, code quality and developer productivity.

4.1 Setting

Participants in both conditions worked within 4-hour study sessions. Control participants worked individually in 17 4-hour sessions. We excluded the data of 3 participants that dropped out from the study, leaving 14 participants. Microtask participants worked in a crowd, taking part in either of two independent 4-hour sessions. We invited 8 participants for each session, and in each session, one participant dropped from the study. All study sessions were conducted remotely, with all interactions between participants and experimenters occurring via Skype or email.

Participants in both conditions were free to use the Internet to find code or solutions to their challenges. We worked to create a setting close to each participants' everyday environment. In the registration form, we asked control participants to give us information about the tools they use in their daily development. We then installed these tools before the experiment. Developers in the control condition connected remotely to our laptop via TeamViewer or AnyConnect, and they were free to use or install any other tools with which they were familiar. As developers face lengthy joining scripts in their onboarding process (Von Krogh et al. 2003), we tried to simulate onboarding barriers. In the control condition, developers needed to set up the environment, which included installing npm dependencies and building the code. We chose to ask participants to connect to our laptop to control for potential differences between participants in the processing power of their computer. Participants in the experimental condition worked on their local machine, using the *microtask programming* environment through a web browser.

4.2 Participants

We recruited participants who met two inclusion criteria: (1) at least six months of experience in JavaScript and (2) proficiency in English. To simulate the geographically diverse nature of crowd work, we recruited participants broadly. We distributed an online flyer on social networks, including Facebook, Twitter, LinkedIn, and Slack groups. One hundred forty-four responded and completed a registration form, where we gathered demographic data. Sixty respondents met our inclusion criteria. We invited all 60 to participate, and 33 chose to participate. Two experimental participants and three control participants left the study due to issues including technical issues in remotely connecting, personal problems, or feeling overwhelmed. We excluded the data of these 5 in our analysis. We report results from the remaining 28 participants.

The participants were geographically diverse, residing in Brazil, Canada, India, the Netherlands, Spain, and the United States. Participants had a median of 4.0 (mean = 5.0) years experience programming, a median of 2.0 (mean = 2.6) years experience in JavaScript, and a median of 1.7 (mean = 2.6) years of industry experience. 35.1% reported being a student in computer science or a related field, and 64.9% reported working as a software engineer. Throughout this paper, we refer to control participants as C1 to C14 and experimental participants as M1 to M14. Participants were compensated with $20 in Amazon gift card credit per hour.

4.3 Task

Participants in both conditions worked to implement and debug function bodies of an online-shopping microservice. We chose this application as we expected its main use cases to be familiar to participants. A task description based on an online-shopping app's key use cases was created for a microservice and used in both conditions. In the control condition, this description was provided through nine GitHub issues. In the experimental condition, this description was provided through a Client-Request with nine endpoints. Conditions varied only in that the the experimental condition participants were given a signature of each function as part of the microtask. The rest of the tasks' decomposition and task descriptions were identical for both conditions. We piloted the task to ensure the task was of an appropriate size and difficulty. After two pilots, we found it to be too easy, so we added additional complexity to the service descriptions.

Participants in both conditions worked in a similar context, with several differences arising from the experimental manipulation. To simulate the conditions of a traditional software project, participants in the control condition began with a pre-existing microservice project consisting of 2035 lines of code, implemented using Express.js. It included an HTTP endpoint example, which participants were able to copy and edit to create function signatures. In the microtask condition, participants could only see the code for the function associated with each microtask. In *microtask programming*, the tutorials included an example unit test. Therefore, a unit test example was included in the codebase of control participants.*Microtask programming* makes use of a preconfigured IDE, in which all libraries and dependencies are installed. Because of that, experimenters for the control group created all configurations and installed all required dependencies of an Express.js[2] project before participants began work.

Participants in both conditions made use of the Firebase persistence API.[3] All participants had access to a wrapper of the API. In the microtask condition, participants could not see the implementation of the wrapper, as it was embedded in the preconfigured programming environment, while in the control condition the code of the wrapper was included in the codebase.

4.4 Procedure

In the control condition, each participant worked alone on his or her task in an individual study session. In the experimental condition, participants worked in shared sessions containing a crowd of 7 participants. All participants completed three steps within a 4 hours session: tutorials (20 minutes), a programming task (205 minutes), and a post-task survey (15 minutes).

Step 1: Tutorials: All participants first completed tutorial materials on the programming tools used in the study. Control participants completed a three part tutorial. It first included an IDE tutorial to ensure they understood the basic features of their chosen IDE. In the second part, as control participants needed to use GitHub to clone, commit, and push code to a repository, a quick review was given of these steps. Control participants were also given a quick tutorial of writing unit tests using Mocha. Participants could

[2]https://expressjs.com/

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

choose to skip tutorials they felt they did not need. Experimental participants completed a tutorial explaining the unfamiliar *microtask programming* environment they would be asked to use, including background information about the environment and concepts, through a 8 minute video and a series of written explanations. All participants had up to 20 minutes to complete this step.

Step 2: Programming Task: Control participants first opened their desired IDE, cloned the code from a GitHub repository, installed npm dependencies, and built the codebase. Control participants then went to the Issues page of the repository and selected issues to address. Control participants were free to work on issues in any order. Writing unit tests was not required, but participants were free to write unit tests as they saw fit. *Microtask programming* participants used the Crowd Microservices preconfigured web-based IDE and did not need to install any tools. All participants had 205 minutes to work on the programming task.

Step 3, Post-task Survey: At the conclusion of the task, all participants completed the post-task survey containing open-ended questions about their experiences in the study. Participants were asked to share challenges they experienced in onboarding, working with tutorials, readme pages, and instruction, setting up and using their programming environment, and understanding the codebase.

4.5 Data Collection and Analysis

We collected a broad range of quantitative and qualitative data from 6 different data sources (DS 1-6). Table 2 summarizes the data we collected.

DS1, Final Code: we collected the final code of control group sessions from the GitHub repository and from the *microtask programming* sessions.

DS2, IDE Log: we collected log data generated by the *microtask programming* environment, including participant actions fetching, skipping, and submitting microtasks, and asking or answering questions.

DS3, Screencasts: we collected screencasts of participants' work on their tasks, 112 hours in total. The experimenters watched the screencasts to identify onboarding activity as well as qualitatively describe how developers worked in the programming tasks.

Table 2 Summary of RQs, measures, and data sources

	Measures	Data source (DS)
RQ1: Onboarding time	Minutes to complete the first line of code	DS3: Screencasts DS2: IDE Log
	Minutes to complete the first microtask or issue	DS3: Screencasts DS2: IDE log
RQ1: Project Velocity	# correctly implemented behaviors per session	DS5: External Test Suite
	Lines of code written per session	DS1: Final Code
RQ3: Code quality	Mean scores of clarity, consistency, simplicity by panel	DS6: Quality Ratings
RQ4: Developer productivity	# correctly implemented behaviors per developer hour	DS5:External Test Suite
	Final lines of code written	
per developer hour	DS1: Final Code	
RQ 1-4	Qualitative report	DS4: Survey DS3: Screencasts

DS4, Post-task Survey: we collected data through a post-task survey which all participants completed. Each condition had a specialized survey containing open-ended questions addressing the specific nature of work in each condition. We used thematic analysis in the analysis of these surveys.

DS5, External Test Suite: to measure the output created by participants, we created an *External Test Suite*. Working from the nine issues or endpoints given to participants, we identified 39 distinct behaviors and wrote a unit test for each.

DS6, Quality Ratings: to assess the quality of the code written in each session, each of the session's final codebases was separately evaluated by four panelists. Panelists were anonymous to condition. Panelists evaluated 16 codebases (two created by *microtask programming* participants and 14 created by control participants) for their clarity, simplicity, and consistency. We then created an overall quality score by averaging scores across the 3 criteria.

To answer RQ1 (onboarding), we measured the time developers spent onboarding. In both conditions, we manually watched the screencasts (DS3: Watching Screencasts) and identified the points of time at which developers began activities strongly related to onboarding (Fig. 2). In addition, we used the *microtask programming* IDE's logs (DS2) to identify when developers completed the tutorials, fetched the first microtask, and finished the first microtask.

To answer RQ2 (project velocity) and RQ4 (developer productivity), we used the *External Test Suite* (DS5) and measured the lines of code developers wrote (DS1). We used the *External Test Suite* to measure the behaviors successfully implemented at the end of each session. In addition, we counted the lines of code written in each session.

The quantitative data collected to answer RQ1, RQ3, and RQ4 were normally distributed, based on a Kolmogorov-Smirnov test. To answer research questions RQ1, RQ3, and RQ4, we used a Welch test, as the two conditions were non-overlapping and had unequal variances.

5 Results

5.1 RQ1: How does *Microtask Programming* Impact Onboarding?

To investigate how a microtask style of work impacts the time necessary to onboard onto a new project, we measured the time developers spent onboarding. We define onboarding as

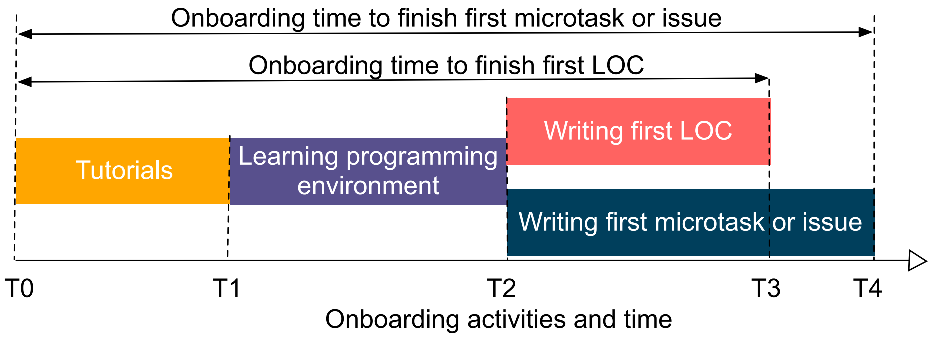

Onboarding time to finish first microtask or issue
Onboarding time to finish first LOC
Tutorials
Learning programming environment
Writing first LOC
Writing first microtask or issue
T0
T1
T2
T3
T4
Onboarding activities and time

Fig. 2 Activities related to onboarding. Developers read tutorials until cloned code or open *microtask programming* environment (T0 to T1), explored and learned programming environments (T1 to T2), began work on the first task until writing first line of code (T2 to T3), and began work on the first task until writing their first microtask or issue (T3 to T4)

the orientation time in which a new developer adjusts to and becomes productive within a project (Begel and Simon 2008). We conservatively identified the points of time at which developers began activities strongly related to onboarding (Fig. 2): time spent working in tutorials until participants first cloned code or opened the *microtask programming* environment (T0 to T1), time spent exploring and interacting with the programming environment, and other resources such as readme pages until participants began work on their first task (T1 to T2), time from beginning work on their first task until writing their first line of code (T2 to T3), and additional time completing their first implementation microtask or issue (T3 to T4). In characterizing onboarding bellow, we focus on reporting the time required to write the first line of code (T0 to T3) and the time needed to finish the first implementation microtask or issue (T0 to T4). In some cases, participants were toggling back and forth, such as rereading tutorials. In these corner cases, we considered the first checkpoint developers reached. For instance, we measured T4 as the time developers completed microtasks or issues regardless of developers' time spent rereading tutorials or searching on the Internet.

Overall, *microtask programming* significantly reduced onboarding time, measured both from the beginning of the session to the writing the first line of code (T0 to T3) and to completing the first task (T0 to T4) (Fig. 3). The time for *microtask programming* participants to finish the first line of code (T0 to T3) was significantly less (Welch's t(26) = 3.03, p = 0.002, data was normally distributed), reducing onboarding time by a factor of 1.3 to 39 minutes (SD = 13) compared to 52 minutes (SD = 8) for control participants. Microtask programming participants completed their first task (T0 to T4) significantly faster (Welch's t(26) = 15.2, p = 0.00001, data normally distributed) in 44 minutes (SD = 11) compared to 164 minutes (SD = 25) for control participants, reducing time by a factor of 3.7. Microtask participants spent 18 minutes interacting with the tutorials (T0 to T1), 13 minutes learning the programming environment (T1 to T2), 7 minutes writing their first line of code (T2 to T3), and 12 minutes completing their first task (T2 to T4). In contrast, control participants spent 8 minutes on the tutorials (T0 to T1), 0 minutes learning the programming environment (T1 to T2), 45 minutes writing their first line of code (T2 to T3), and 156 minutes to complete their first issue (T2 to T4). One cause of the shortened time to complete a microtask is the smaller granularity, as compared to a traditional issue. On average, the first submitted microtask had seven lines of code, including four in the function body and three in unit tests. Control participant's first issue included 51 lines of code in functions and 0 in unit tests.

To identify potential explanations of the differences in onboarding times, we analyzed the 112 hours of screencasts and data from the post-task interviews to identify challenges faced by participants in each group and how participants chose to address these challenges. Both control and *microtask programming* participants experienced challenges getting up to speed with their programming environment. Control participants faced several challenges and spent substantial time configuring their programming environment. While free to use any IDE or tool they desired, they still experienced challenges cloning code from GitHub, building code, and installing dependencies. In contrast, the preconfigured environment available to microtask participants enabled them to not spend time on these activities. However, microtask participants instead reported being overwhelmed by the many new concepts in the programming environment presented in the tutorials, including the behavior-driven development process.

> "There was a lot of feature involved and it was hard to understand it all and remember it without having ever used the service." - (M5)

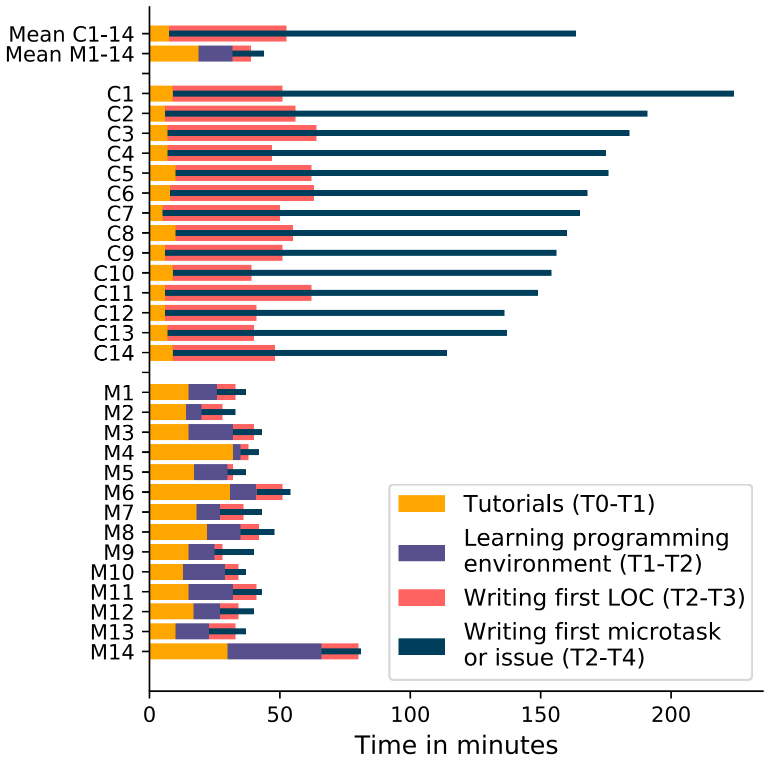


Fig. 3 Onboarding time for control (C1 to C14) and experimental (M1 to M14) participants, including mean onboarding time by condition

As a result, they initially reported being confused and unsure of how to contribute. Nevertheless, after *microtask programming* participants worked on their first tasks, they gradually learned how to use the novel workflow. In contrast, control participants did not have problems with how to start and spent 0 minutes initially learning the programming environment (T1 to T2). In resolving challenges, control participants generally used Internet searches while *microtask programming* participants instead used code documentation, the readme, and tutorials.

Despite the modest 2035 LOC size of the codebase, understanding the structure of the codebase took substantial time for control participants, replicating existing findings (Wang and Sarma 2011; Fagerholm et al. 2014). Control participants tried to understand the codebase by opening various files and reading code.

> "At first I was trying to determine how the project has been structured in terms of its architecture. The reason was it is going to help me to better locate the code and methods I would write..." - (C1)

Most reported that the codebase was standard and straightforward. We did not observe challenges by microtask participants understanding the structure of the codebase, as each microtask focused on only a single function.

Learning how to work with an API (the Firebase persistence API) was a challenge for control participants but not *microtask programming* participants. Participants in both conditions had access to a simple wrapper for the API, but the *microtask programming* environment hid its implementation and exposed only its signatures while control participants could see its implementation. Control participants also had access to more extensive documentation. While control participants could simply use the wrapper to complete their tasks, they instead spent time reading documentation on the Internet to understand the underlying implementation.

> "... not knowing the FireBase was and still is the biggest challenge. I had no experience..., so I was like, "How am I gonna do this¿' ... At these points, my productivity decreased very much..." - (C3)

The choice of the first task may also have impacted the necessary onboarding activities. *Microtask programming* participants were randomly assigned tasks, which they could skip if they desired. While familiarizing themselves with the environment, they preferred to skip hard microtasks until they found microtasks focusing on simple implementation logic like checking corner cases. After starting with simple microtasks, they were able to submit their first tasks quickly. In contrast, control participants choose issues by reading issue titles and did not exhibit any clear pattern in selecting issues. They sometimes worked on their first issue for hours.

Takeaways from RQ1: *microtask programming* reduced the onboarding time required by a developer to write their first line of code by a factor of 1.3 and the time to complete their first task by a factor of 3.7. The preconfigured but novel programming environment, reduced task size, and reduced need to understand the codebase structure and implementation may have contributed to these differences.

5.2 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 (LaToza and Van Der Hoek 2015). Measuring project velocity is a challenge, as it is difficult to recruit participants with professional experience who are willing to work for an unbounded amount of time. Therefore, rather than fix the amount of work to complete (i.e., the whole application) and measure time, we instead fixed time (i.e., 205 minutes) and measured the work completed. That is, we examined the impact of increasing the number of participants per session on the output generated by the session. We 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*. We assumed that if developers can correctly implement more software logic in a session, they would deliver a completed project in less time. Therefore, a larger number of lines of code or correctly implemented behaviors in a session indicates a higher project velocity.

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. As Fig. 4 indicates, 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

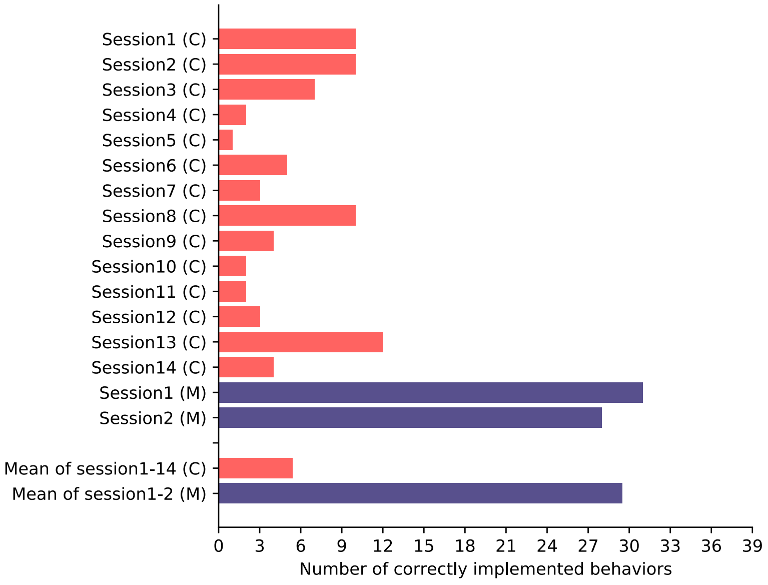


Fig. 4 The number of correctly implemented behaviors by session. The total number of behaviors was 39

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).

Takeaways from RQ2: Increasing the number of developers by a factor of 7 in the *microtask programming* condition increased the number of behaviors successfully implemented by a factor of 5.7 and the number of lines of code written by a factor of 9.1.

5.3 RQ3: How does *Microtask Programming* 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 (Martin and McClure 1983). To assess the quality of the implementation created in each session, a panel of four was assembled, with a median of 4 years of industry experience and a median of 2 years experience with JavaScript. Each of the 16 session's final implementation was evaluated by four panelists separately. Panelists evaluated 16 codebases (two created by *microtask programming* participants and 14 created by control participants) for their clarity, simplicity, and consistency. Panelists were anonymous to condition. For each metric, panelists gave a score from 1 to 5: 1: poor, 2: fair, 3: satisfactory, 4: very good, 5: excellent. We then calculated a mean score for each codebase by averaging across all 4 panelists. Finally, we created an overall quality score by averaging the 3 metrics.

Overall, we 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 (Table 3). However, this difference was not significant (Welch's t(14) = 1.35 and p

Table 3 Summary of findings comparing *microtask programming* to traditional programming

	Measures	Traditional Programming	Microtask Programming
RQ1: Onboarding time	Mins to complete the first line of code	52	39
	Mins to complete the first microtask or issue	44	164
RQ2: Project velocity	# correctly implemented behaviors per session	5.28 of 39	29.5 of 39
	Lines of code written per session	115	1050
RQ3: Code quality	Mean quality score by panel	3.3	3.7
RQ4: Developer productivity	# correctly implemented behaviors per developer hour	1.6	1.2
	Final lines of code written per developer hour	34	44

RQ1: Onboarding: the orientation time in which a new developer adjusts to and becomes productive. RQ2: Project velocity: the progress of the team towards project completion. RQ3: Code quality: the maintainability of code through its clarity, simplicity, and consistency. RQ4: Developer productivity: the amount of useful output created per unit of time

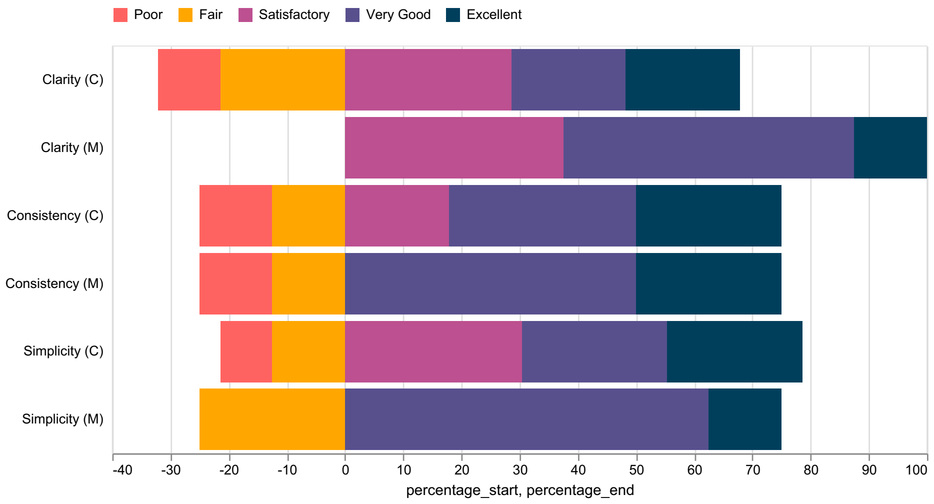


Fig. 5 The percentage of panel ratings of code quality for clarity, consistency, and simplicity for control and *microtask programming* codebases

= 0.09, data normally distributed). The effect size for Glass's delta is 2.14 and for Cohen's d = 0.99. This indicates that the mean quality score of the *microtask programming* codebases were at the 76_{th} percentile of the control group. 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. The diverging stacked bar chart in Fig. 5 shows the percentage frequency of panelists' scores, where each stack represents the frequency of each score. For instance, the "Consistency(M)" bar for *microtask programming* indicates scores of 12.5% poor, 12.5% fair, 0% satisfactory, 50% very good, and 24% excellent.

One difference between the *microtask programming* and control conditions which may have led to differences in quality was the presence of *Review* microtasks in the *microtask programming* condition. In *Review* Microtasks, participants gave feedback on each contribution made by others and accepted or rejected those contributions. We counted the number of submitted *Review* microtasks. Participants submitted 94 (48 accepted, 46 rejected) in the first session and 130 (87 accepted, 43 rejected) in the second. In the post-task survey, participants reported that the review microtasks were helpful.

Writing unit tests facilitate maintenance activities (Bavota et al. 2012). Participants in the *microtask programming* condition wrote substantially more unit tests than those in the control condition. *microtask programming* significantly increased the lines of test code written per developer (Welch's t(26) = 5.1, $p < 0.00001$) from 0.5 lines (SD = 1.5) to 97 lines (SD = 68). In the control condition, only one developer implemented one unit test.

Takeaways from RQ3: Code quality, as assessed by a panel anonymous to condition, did not significantly differ between traditional and *microtask programming*. *Microtask programming* participants made judgments about code quality through reviews and wrote significantly more unit tests.

5.4 RQ4: How does *Microtask Programming* 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 (Sadowski and Zimmermann 2019). Developers are more productive when they create more direct and indirect output in less time. We measured output through two complimentary measures: lines of code and the number of correctly implemented behaviors, as measured by executing the *External Test Suite*.

We measured direct productivity by considering developers' contributions to the final output. As all developers in each session contributed to the final output, we divided the final project velocity of each session by the number of developers in each session. The direct productivity of developers in the microtasking condition decreased by 25% in behaviors per hour and increased by 29% in lines of code per hour. 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.

One difference in the microtask condition was that developers wrote code in microtasks, which other developers may later have edited or deleted. To investigate the impact of this on productivity, we computed a measure of *incremental* direct productivity. For each microtask, we created a diff and measured the total number of lines of code updated, added or deleted. In addition, we ran the unit test suite to assess the delta in behaviors correctly implemented.

As measured through incremental contributions, *microtask programming* did not decrease productivity, measured either through behaviors per hour or lines of code. *Microtask programming* increased the number of behaviors successfully implemented per hour by a factor of 1.4, although this difference was not significant (Welch's t(26) = 1.54, p-value = 0.067, data normally distributed). Microtask participants completed 2.3 behaviors per hour (SD = 1.3) compared to 1.6 (SD = 1.0) for control participants (Fig. 6). *microtask programming* significantly increased the number of lines of code implemented per hour (Welch's t(26) = 2.5, $p < 0.010$, data normally distributed), increasing the lines of code written from 34 per hour (SD = 13) to 60 per hour (SD = 36). However, only 54% of the code developers initially submitted in a microtask still existed at the end of the session.

To understand how differences in what developers did might have impacted productivity, we analyzed the screencasts to identify the fraction of participants that engaged in specific activities. Microtask participants spent their time reading tutorials (92% of participants), coding (100%), debugging (14%), reviewing (100%), reading Newsfeed messages (100%), and reading and answering questions (100%). In contrast, control participants spent their time overcoming onboarding barriers (100%), understanding the codebase (100%), searching the Internet (42%), coding (100%), and debugging (7%). In addition, we used the log data in the *microtask programming* condition to assess how microtask participants spent their time. Microtask participants spent 60% (mean = 117 mins) of their time working on microtasks, 20% (mean = 34 mins) on *Review Microtasks*, and 40% (mean = 83 mins) on *Implement Function Behavior Microtasks*). They spent the remaining 40% of their time

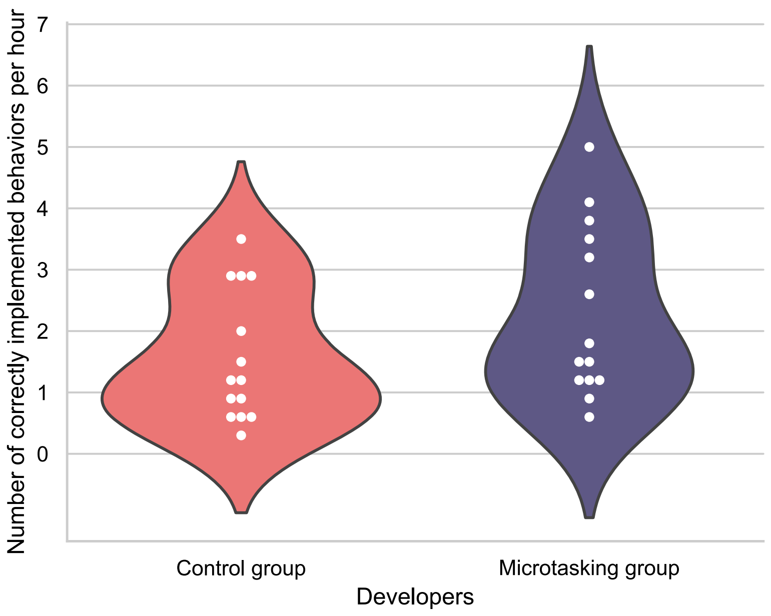


Fig. 6 The number of correctly implemented behaviors per hour by condition

on non-coding activities such as skipping tasks, reading and answering questions, reading Newsfeed messages, reading content on the dashboard, context switching, and taking breaks.

Task switching might potentially impact the productivity of developers. *Microtask programming* developers switched tasks 12.7 times more often than with traditional programming. *Microtask programming* developers worked on an average of 31.78 microtasks, while developers in the traditional condition addressed 2.5 different GitHub issues.

Some participants reported that the gamification elements in the *microtask programming* condition may have increased their motivation, which might have also impacted their productivity. This finding is consistent with the previous studies that found humans use different sets of nonverbal behaviors to express their prestige and dominance (Witkower et al. 2020). *Microtask programming* participants had differing opinions about the *Leaderboard*, viewing it as stressful, a source of motivation, or helpful gamification. Participants reported that before fetching a microtask, they looked at the *Leaderborad*, and that by watching the scores of others, they were motivated to increase their score and ranking. Participants could achieve this by submitting more microtasks with higher quality scores.

> "I like the competitiveness feel it brings with the points system, rating and all that. It definitely made me to contribute more." - (M13)

Takeaways from RQ4: The direct productivity of developers in the microtasking condition decreased by 25% in behaviors per hour and increased by 29% in lines of code per hour. Only 54% of the code initially written by microtask participants still existed at the end of the session.

6 Limitations and Threats to Validity

As with any empirical study, our study has several important limitations and potential threats to the internal and external validity of the results.

Our study had several potential threats to internal validity. The first potential threat to internal validity was that participants in the control condition remotely connected to our laptop to work. Although they were free to install any tool-set they desired, the remote connection might have reduced their productivity. Because of connectivity problems, two control participants in the middle of the study switched from TeamViewer to the AnyConnect tool. This switch distracted participants. In addition, several participants were unable to use their standard shortcuts. *Microtask programming* participants worked in the web-based IDE in the Chrome browser on their local devices.

A second potential threat to internal validity was the novelty of the *microtask programming* concepts and the environment for participants. *Microtask programming* participants had no prior experience with the programming environment. In contrast, control participants were working in programming environments with which they were already familiar. The unfamiliarity of the programming environment may have increased the onboarding barriers for *microtask programming* participants or made them less productive throughout the tasks. Our study results simulate a setting in which developers are working using *microtask programming* for the first time. because of that as Fig. 3 shows *microtask programming* developers spent 13 minutes on learning programming environment but control participants spent 0 minutes.

A third potential threat to internal validity is the study setting of observing participants as they worked. If experimenters did not observe the participants, they might be more productive. In the control condition, participants were observed synchronously while in the experimental condition participants were observed asynchronously. Being observed might change the work style of the participants. The experimenter tried to mitigate this risk by emphasizing that the goal of the study was was to evaluate the approach and by not interacting with participants during the study.

Our study also had potential threats to external validity. To imitate the open-call process of microtask systems, we recruited participants with a wide range of of backgrounds from our university and globally via social networks like Facebook, Twitter, LinkedIn, and Slack. The results might vary for developers who are exclusively more experienced or more novice.

The second threat to external validity is that the tasks given to participants might differ from those used in traditional development projects. To ensure that the task descriptions in both conditions were the same, the control condition task descriptions were more detailed than typical in traditional development. For example, they included descriptions of how to handle specific error messages. More detailed descriptions of the task may have made the task in the traditional programming condition easier, as developers had less design work to do. In this way, our results may underestimate the differences between microtasking and traditional development. Simultaneously, developers in both conditions benefited from the design work done to create the more detailed task descriptions. In addition, we did not measure the time required to do this work to create these task descriptions. In another study (Saito et al. 2020), we measured the effort needed to prepare function descriptions and method signatures in an industrial project making use of microtask programming. One designer and one dedicated software engineer worked for two weeks to analyze requirements and prepare the design materials, which were then used by a crowd of 6 workers to implement a web app over the course of 4 weeks. Of course, this encompassed both doing the design work itself, including writing the function signatures, and is not directly comparable to the effort required only to translate an existing design into function signatures. So it is not currently possible to accurately measure the time required to create function signatures from a completed function description. In principle, more of this design work might potentially be done by the crowd. Crowd design workflows have begun to be explored, which might potentially be adopted for this purpose (Weidema et al. 2016).

Another potential threat to external validity is the choice of the programming task. We chose a synthetic task designed to reflect typical microservice backends. For a more challenging programming task or with a larger codebase, our results might vary. In particular, onboarding barriers might be larger and the variance between participants might be higher for more challenging tasks. This might impact the effect of microtasking on project velocity and developer productivity either positively or negatively.

7 Discussion

Microtask programming envisions a software development process in which large crowds of transient developers build software through short and self-contained microtasks, reducing barriers to onboarding and increasing participation in software projects. To achieve this, *microtask programming* adopts four core mechanisms: a preconfigured IDE, decomposition of tasks into microtasks, new coordination mechanisms, and gamification and feedback. We conducted the first direct comparison between *microtask programming* and traditional programming in implementing and debugging function bodies of a microservice. We investigated the impact of *microtask programming* on onboarding, velocity, quality of code, and individual developer productivity. Table 3 summarizes the main findings.

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.

We found that, compared to traditional programming, *microtask programming* reduced onboarding time by a factor of 3.7. A key reason may be that microtasking required less of developers in terms of familiarizing themselves with the codebase. Developers could only see the code for the function for each microtask, and participants could not see the wrapper's implementation. However, in traditional development, developers could see, read and learn the whole codebase. The learning challenges traditional developers faced could not occur for microtask participants.

Another key reason for the reduced onboarding time for microtasking may be the availability of a preconfigured IDE. Without the preconfigured IDE, developers spent substantial time time and effort setting up the programming environment, interacting with Git to clone, commit, and push code, and learning how to correctly use a third-party API. These barriers mirror those reported as onboarding challenges in open source projects (Fagerholm et al. 2014; Fagerholm et al. 2013). The preconfigured environment enabled developers to skip these activities, removing these barriers. This offer important evidence about the value of a preconfigured environment for reducing onboarding effort. This suggests the potential for adopting a preconfigured environment more broadly, even beyond a microtasking context. Commercial tool vendors have begun to offer features towards this end, such as GitHub Codespaces (Microsoft, G.: Github codespaxces. https://github.com/features/codespaces), Gitpod (Gitpod: Gitpod. https://www.gitpod.io), AWS Cloud9 (Cloud9: Aws cloud9. https://aws.amazon.com/cloud9/). These web-based integrated development cloud platforms typically consist of a code editor, compiler, command line, debugger, an API, source version controller, or a graphical user interface (GUI) builder. However, none of these have yet supported automatically generating implementation or review tasks, TDD approach, offline Q/A, and gamification. These environments might be able to offer even more support for onboarding by adopting more of these features.

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. We found that *microtask programming* was surprisingly successful in minimizing these costs, where increasing the number of project contributors substantially increased project velocity.

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 (Cheng et al. 2015; Iqbal et al. 2018). Decomposing tasks into several microtasks was the key to achieving higher quality, as contributors could focus on smaller tasks without interruption. Our 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.

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.

In this study, we focused only on the impact of microtasking on green-field implementation and debugging of function bodies rather than on software maintenance or design. Many questions remain about how microtasking might be adopted in maintenance or design tasks. Our study largely mirrors the focus of microtasking and tool methodologies on supporting this aspect of work. Existing systems have not yet offered ways to microtask maintenance work. More work has been done in the area of design, such as envisioning ways in which the design tasks, which in our study were completed by the experimenters, might be down by the crowd (Weidema et al. 2016). However, these new microtask design techniques have not yet been connected to programming tasks, and it remains unclear how effective they might be for building the task specifications used in *microtask programming*. In the meantime, asking a client to construct these manually is clearly a substantial challenge. Much more work is needed to design new techniques and approaches for these problems and evaluate the impact of these approaches on microtasking.

8 Conclusions

This paper contributes findings from a controlled experiment comparing *microtask programming* to traditional programming, in the context of the task to implement and debug function bodies. Our findings show that, compared to traditional programming, *microtask programming* reduces onboarding time, increases project velocity, and decreases individual developer productivity. At the same time, the quality code created is not significantly reduced.

These findings begin to lay a foundation for adopting microtask programming in practice. In contexts where project velocity is important, there may be benefits to adopting a microtask programming approach to implement individual modules. To do so, developers might first describe desired functionality through an API, which a crowd might then implement and debug through microtask programming. More work remains to investigate how microtask programming might be adopted to a broader range of contexts, such as maintenance or design tasks.

Acknowledgements We thank the study participants for their participation.

Funding The research leading to these results received funding in part from the National Science Foundation under Grant Agreement No CCF-1414197 and Grant Agreement No CCF-1845508.

Data Availability The study replication package[4] data that support the findings of this study are available in Zenodo (Aghayi and LaToza). It includes the study materials, the test suite used to evaluate contributions, and the code written by the participants.

Declarations

Competing interests The authors declare they have no financial interests.

References

Aberdour M (2007) Achieving quality in open-source software. IEEE Softw, 58–64

Aghayi E (2020) Large-scale microtask programming. In: Symposium on visual languages and human-centric computing, pp 1–2

Aghayi E, LaToza TD Replication package of this study. https://doi.org/10.5281/zenodo.4922866

Aghayi E, LaToza TD, Surendra P, Abolghasemi S (2021) Crowdsourced behavior-driven development. J Syst Softw 171:110840

Ammann P, Offutt J (2016) Introduction to software testing. Cambridge University Press

Basili VR (2001) Open source and empirical software engineering. Empir Softw Eng, 193–194

Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D (2012) An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: International conference on software maintenance, pp 56–65

Begel A, Simon B (2008) Novice software developers, all over again. In: International workshop on computing education research, pp 3–14

Bernstein M, Little G, Miller RC, Hartmann B, Ackerman MS, Karger DR, Crowell D, Panovich K (2010) Soylent: a word processor with a crowd inside. In: Symposium on user interface software and technology, pp 313–322

Britto R, Smite D, Damm LO, Börstler J (2020) Evaluating and strategizing the onboarding of software developers in large-scale globally distributed projects. J Syst Softw, 169

[4]https://doi.org/10.5281/zenodo.4922866

Chen Y, Lee SW, Xie Y, Yang Y, Lasecki WS, Oney S (2017) Codeon: on-demand software development assistance. In: Conference on human factors in computing systems, pp 6220–6231

Chen Y, Oney S, Lasecki WS (2016) Towards providing on-demand expert support for software developers. In: Conference on human factors in computing systems, pp 3192–3203

Cheng J, Teevan J, Iqbal S, Bernstein M (2015) Break it down: a comparison of macro-and microtasks. In: Conference on human factors in computing systems, pp 4061–4064

Cutrell EB, Czerwinski M, Horvitz E (2000) Effects of instant messaging interruptions on computing tasks. In: Extended abstracts on human factors in computing systems, pp 99–100

Czerwinski M, Horvitz E, Wilhite S (2004) A diary study of task switching and interruptions. In: Conference on human factors in computing systems, pp 175–182

Doan A, Ramakrishnan R, Halevy AY (2011) Crowdsourcing systems on the world-wide web. Commun ACM 54:86–96

Espinosa A, Kraut R, Lerch J, Slaughter S, Herbsleb J, Mockus A (2001) Shared mental models and coordination in large-scale, distributed software development. International Conference on Information Systems, 64

Fagerholm F, Guinea AS, Borenstein J, Münch J (2014) Onboarding in open source projects. IEEE Softw, 54–61

Fagerholm F, Johnson P, Guinea AS, Borenstein J, Münch J (2013) Onboarding in open source software projects: a preliminary analysis. In: International conference on global software engineering workshops, pp 5–10. IEEE

Filippova A, Cho H (2016) The effects and antecedents of conflict in free and open source software development. In: Conference on computer-supported cooperative work and social computing, pp 705–716

Goldman M (2012) Software development with real-time collaborative editing. Ph.D. thesis Massachusetts Institute of Technology

Goldman M, Little G, Miller RC (2011) Real-time collaborative coding in a web ide. In: Symposium on user interface software and technology, pp 155–164

Goldman M, Little G, Miller RC (2011) Real-time collaborative coding in a web ide. In: Symposium on user interface software and technology, pp 155–164

Goldman M et al (2012) Software development with real-time collaborative editing. Ph.D. thesis Massachusetts Institute of Technology

Hoseini M, Saghafi F, Aghayi E (2018) A multidimensional model of knowledge sharing behavior in mobile social networks Kybernetes

Iqbal S, Bailey BP (2008) Effects of intelligent notification management on users and their tasks. In: Conference on human factors in computing systems, pp 93–102

Iqbal S, Teevan J, Liebling D, Thompson AL (2018) Multitasking with play write, a mobile microproductivity writing tool. In: Symposium on user interface software and technology, pp 411–422

Jensen RW (2014) Improving software development productivity: Effective leadership and quantitative methods in software management. Pearson Education

Jergensen C, Sarma A, Wagstrom P (2011) 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

Jiang H, Matsubara S (2014) Efficient task decomposition in crowdsourcing. In: International conference on principles and practice of multi-agent systems, pp 65–73

Johnson M, Senges M (2010) Learning to be a programmer in a complex organization: A case study on practice-based learning during the onboarding process at google. Journal of Workplace Learning

Kamma D, Jalote P (2013) Effect of task processes on programmer productivity in model-based testing. In: India software engineering conference, pp 23–28

Kersten M, Murphy GC (2006) Using task context to improve programmer productivity. In: International symposium on foundations of software engineering, pp 1–11

Kim J, Sterman S, Cohen AAB, Bernstein M (2017) Mechanical novel: crowdsourcing complex work through reflection and revision. In: Conference on computer supported cooperative work and social computing

Kittur A, Nickerson JV, Bernstein M, Gerber E, Shaw A, Zimmerman J, Lease M, Horton J (2013) The future of crowd work. In: Conference on computer supported cooperative work, pp 1301–1318

Kittur A, Smus B, Khamkar S, Kraut RE (2011) Crowdforge: crowdsourcing complex work. In: Symposium on user interface software and technology, pp 43–52

Kochhar PS, Bissyandé TF, Lo D, Jiang L (2013) An empirical study of adoption of software testing in open source projects. In: International conference on quality software, pp 103–112

Lakhani KR, Garvin DA, Lonstein E (2010) Topcoder (a): developing software through crowdsourcing Harvard Business School General Management Unit Case

Lasecki WS, Kim J, Rafter N, Sen O, Bigham JP, Bernstein M (2015) Apparition: crowdsourced user interfaces that come to life as you sketch them. In: Human factors in computing systems, pp 1925–1934
LaToza TD, Chiquillo E, Towne WB, Adriano C, van der Hoek A (2013) Crowdcode: a platform for crowd development. In: CrowdConf
LaToza TD, Di Lecce A, Ricci F, Towne B, Van der Hoek A (2018) Microtask programming. Transactions on Software Engineering, 1–20
LaToza TD, van der Hoek A (2016) Crowdsourcing in software engineering: models, motivations, and challenges. IEEE software, 74–80
LaToza TD, Lecce AD, Ricci F, Towne WB, van der Hoek A (2015) Ask the crowd: scaffolding coordination and knowledge sharing in microtask programming. In: Symposium on visual languages and human-centric computing, pp 23–27
LaToza TD, Towne WB, Adriano CM, Van Der Hoek A (2014) Microtask programming: building software with a crowd. In: Symposium on user interface software and technology, pp 43–54
LaToza TD, Van Der Hoek A (2015) Crowdsourcing in software engineering: models, motivations, and challenges. IEEE Softw, 74–80
Lee SW, Krosnick R, Park SY, Keelean B, Vaidya S, O'Keefe SD, Lasecki WS (2018) Exploring real-time collaboration in crowd-powered systems through a ui design tool. Computer-Supported Cooperative Work and Social Computing, 1–23
Mao K, Capra L, Harman M, Jia Y (2017) A survey of the use of crowdsourcing in software engineering. J Syst Softw, 57–84
Mark G, Gonzalez VM, Harris J (2005) No task left behind? Examining the nature of fragmented work. In: Conference on human factors in computing systems, pp 321–330
Martin J, McClure CL (1983) Software maintenance: the problems and its solutions prentice hall professional technical reference
Maxwell KD (2003) Software development productivity. Adv Comput 58:1–46
Meyer AN, Fritz T, Murphy GC, Zimmermann T (2014) Software developers' perceptions of productivity. In: International symposium on foundations of software engineering, pp 19–29
Nebeling M, Leone S, Norrie MC (2012) Crowdsourced web engineering and design. In: International conference on web engineering, pp 31–45
Oliveira E, Viana D, Cristo M, Conte T (2017) How have software engineering researchers been measuring software productivity?-a systematic mapping study. In: International conference on enterprise information systems, pp 76–87
Perry DE, Siy HP, Votta LG (2001) Parallel changes in large-scale software development: an observational case study. Transactions on Software Engineering and Methodology, 308–337
Peterson B (2017) Travis kalanick lasted in his role for 6.5 years—five times longer than the average uber employee. Business Insider
Ramakrishnan R, Baptist A, Ercegovac V, Hanselman M, Kabra N, Marathe A, Shaft U (2004) Mass collaboration: a case study. In: International database engineering and applications symposium, pp 133–146
Retelny D, Bernstein M, Valentine MA (2017) No workflow can ever be enough: how crowdsourcing workflows constrain complex work. In: Conference on Computer-Supported Cooperative Work and Social Computing, 1–23
Sadowski C, Zimmermann T (2019) Rethinking productivity in software engineering. Springer Nature
Saengkhattiya M, Sevandersson M, Vallejo U (2012) Quality in crowdsourcing-how software quality is ensured in software crowdsourcing. Master's thesis, Department of Informatics Lund University
Saito S, Iimura Y, Aghayi E, LaToza TD (2020) Can microtask programming work in industry? In: European software engineering conference and symposium on the foundations of software engineering, pp 1263–1273
Schiller TW, Ernst MD (2012) Reducing the barriers to writing verified specifications. Special Interest Group on Programming Languages Notices, 95–112
Sim SE, Holt RC (1998) The ramp-up problem in software projects: a case study of how software immigrants naturalize. In: International conference on software engineering, pp 361–370
Stamelos I, Angelis L, Oikonomou A, Bleris GL (2002) Code quality analysis in open source software development. Inf Syst J, 43–60
Steinmacher I, Conte TU, Treude C, Gerosa MA (2016) Overcoming open source project entry barriers with a portal for newcomers. In: International conference on software engineering, pp 273–284
Steinmacher I, Silva MAG, Gerosa MA, Redmiles DF (2015) A systematic literature review on the barriers faced by newcomers to open source software projects. Inf Softw Technol, 67–85
Stol KJ, Fitzgerald B (2014) Two's company, three's a crowd: a case study of crowdsourcing software development. In: International conference on software engineering, pp 187–198

Teevan J, Iqbal S, Cai CJ, Bigham JP, Bernstein M, Gerber EM (2016) Productivity decomposed: getting big things done with little microtasks. In: Conference extended abstracts on human factors in computing systems, pp 3500–3507

Tian J (2005) Software quality engineering: testing, quality assurance, and quantifiable improvement. Wiley

Trafton JG, Altmann EM, Brock DP, Mintz FE (2003) Preparing to resume an interrupted task: effects of prospective goal encoding and retrospective rehearsal. International Journal of Human-Computer Studies, 583–603

Von Ahn L, Dabbish L (2004) Labeling images with a computer game. In: Conference on human factors in computing systems, pp 319–326

Von Krogh G, Spaeth S, Lakhani KR (2003) Community, joining, and specialization in open source software innovation: a case study. Res Policy, 1217–1241

Wang J, Sarma A (2011) Which bug should i fix: helping new developers onboard a new project. In: International workshop on cooperative and human aspects of software engineering, pp 76–79

Warner J, Guo PJ (2017) Codepilot: scaffolding end-to-end collaborative software development for novice programmers. In: Conference on human factors in computing systems, pp 1136–1141

Weidema ER, Lopez C, Nayebaziz S, Spanghero F, van der Hoek A (2016) Toward microtask crowdsourcing software design work. In: International workshop on crowdsourcing in software engineering, pp 41–44

Weld DS, Wu F, Adar E, Amershi S, Fogarty J, Hoffmann R, Patel K, Skinner M (2008) Intelligence in wikipedia. In: AAAI, vol 8, pp 1609–1614

Williams A, Kaur H, Iqbal S, White RW, Teevan J, Fourney A (2019) Mercury: empowering programmers' mobile work practices with microproductivity. In: Symposium on user interface software and technology

Witkower Z, Tracy JL, Cheng JT, Henrich J (2020) Two signals of social rank: prestige and dominance are associated with distinct nonverbal displays. J Pers Soc Psychol 118:89

Zanatta AL, Machado L, Steinmacher I (2018) Competence, collaboration, and time management: barriers and recommendations for crowdworkers. In: Workshop on crowd sourcing in software engineering, pp 9–16

Zhao L, Elbaum S (2000) A survey on quality related activities in open source. SIGSOFT Software Engineering Notes, 54–57

Zhou M, Mockus A (2010) Developer fluency: achieving true mastery in software projects. In: International symposium on foundations of software engineering, pp 137–146