Studying Programmer Behaviour at Scale: A Case Study using Amazon Mechanical Turk

Jason T. Jacques
Department of Engineering
University of Cambridge
Cambridge, United Kingdom
jtj21@cam.ac.uk

Per Ola Kristensson
Department of Engineering
University of Cambridge
Cambridge, United Kingdom
pok21@cam.ac.uk

ABSTRACT

Developing and maintaining a correct and consistent model of how code will be executed is an ongoing challenge for software developers. However, validating the tools and techniques we develop to aid programmers can be a challenge plagued by small sample sizes, high costs, or poor generalisability. This paper serves as a case study using a web-based crowdsourcing approach to study programmer behaviour at scale. We demonstrate this method to create controlled coding experiments at modest cost, highlight the efficacy of this approach with objective validation, and comment on notable findings from our prototype experiment into one of the most ubiquitous, yet understudied, features of modern software development environments: syntax highlighting.

CCS CONCEPTS

• **Human-centered computing → Empirical studies in HCI**; • **Software and its engineering → Integrated and visual development environments**.

KEYWORDS

programming, behaviour, crowdsourcing

ACM Reference Format:
Jason T. Jacques and Per Ola Kristensson. 2021. Studying Programmer Behaviour at Scale: A Case Study using Amazon Mechanical Turk. In *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming (<Programming> '21 Companion), March 22–26, 2021, Virtual, UK*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3464432.3464436

1 INTRODUCTION

Writing code is hard. Reading code is harder. Be it code written by someone else, or your own code that you are coming back to years, weeks, hours, days, or even just minutes later, developing a consistent understanding of what will actually be executed can be immensely challenging. It is crucial that human authors are able to develop and maintain a consistent model of the computers execution to avoid software errors. Syntactically correct code that is logically erroneous can have significant and even fatal consequences (e.g. [20, 27]).

While the process of training and maintaining the correct interpretation by human reviewers of machine interpreters has many aspects, including education, language design, formal methods, and others, extracting generalisable models of how human readers understand and internalise the code in front of them has a number of challenges. Evaluation, both of existing practice and new tools, is crucial to this process. In the literature, studies may use either a captive audience of predominantly novice programmers, such as students [32], require large resources and suffer limited sample sizes [33], or otherwise struggle with generalisability. We explore some of these approaches further in *Alternative Approaches*.

In this paper, we propose a crowdsourced, web-based approach to evaluating human understanding, both of existing practice and of new and innovative tools. We consider this through the lens of a case study evaluation of the efficacy of syntax highlighting. Through this lens, we make the following contributions:

- We demonstrate a method to evaluate programmer behaviour in a semi-controlled manner, at scale, and with modest costs using web-based crowdsourcing.
- Highlight predictable correlations between self-reported experience levels and objective code quality measures including composition times and execution speed.
- Comment on interesting results from a case study, using these techniques, and suggest future applications of our approach.

2 ALTERNATIVE APPROACHES

Software development inherently produces copyrightable, potentially proprietary original works. The potential sensitivity of the material being generated, and the complexity of these task itself, can make evaluation of developer practices and evaluation of new tools challenging. To date, there has been three typical approaches to the study of developer behaviour: *in situ*, in the lab, and *post hoc* analysis.

In situ. This type of study is typically carried out in association with an organisation undertaking a software development project. This type of study may be limited by the confidentiality of the project and the ability to access and interact with the developers under study. These studies may take place over a long period of time and can potentially be disruptive to the project [33]. Additionally, the homogeneous nature of the developer environment limits the diversity of the data collected and commercial nature may preclude comparisons of multiple approaches to the same problem.

<Programming> '21 Companion, March 22–26, 2021, Virtual, UK

ACM ISBN 978-1-4503-8986-0/21/03…$15.00
https://doi.org/10.1145/3464432.3464436

In the lab. An alternative approach to understanding developer behaviour is to recruit developers directly to a lab-based study. This may be done either by recruiting students from related courses to participate in software development exercises, or alternatively hiring professional software developers. While students present a readily available candidate pool, their external validity is questionable due to their inexperience with the task under study and their limited exposure to alternative techniques [32]. More recently Massive Open Online Courses (MOOCs) offer another source of beginner programmer behaviour and code samples, however, data from these participants are likely to suffer the same homogeneity and unintentional but undue influence from the instructors as their in-person counterparts. Professional software developers potentially bring much broader experience to lab-based studies, however sample size is often limited by the expense of recruiting professional developers for the long periods of time required to develop substantial software projects [33].

Post hoc analysis. The modern software development process includes the use of version control systems and allows developer activity to be inferred and analysed after development is complete [16]. This approach can offer access to large code bases, which may be drawn from open source code repositories, typically includes metadata attributing code changes, and may be coupled with some form of issue tracking to allow the reasoning for a given change to be understood. However, the granularity of the available data may be limited by the *check in* frequency of the individual developers and fails to provide access to ethnographic data offering little scope to reconstruct contemporary activity that may have impacted coding decisions.

Each of these techniques offer differing perspectives on developer behaviour, however, each has challenges. By capitalising on the lower cost of crowdsourced work and the diverse nature of developers on these platforms a much more representative example of how code is developed can be collected [39]. Like lab-based studies, crowdsourced data collection can be carried out repeatedly with different participants to evaluate the differing approaches of individuals to the same problem. While crowdsourced data collection shares the retroactive nature of *post hoc* analysis, instrumenting the environment to record additional forms of metadata, including keystroke-by-keystroke interactions, can provide some of the richness typical of *in situ* or lab-based studies but missing from version control systems.

3 CROWDSOURCING CODER BEHAVIOUR

Crowdsourcing offers an approach to recruiting developers which avoids some of the challenges identified in *Alternative Approaches*.

Modern web-browsers offer an extremely flexible platform and have had the ability to support real-time collaborative programming environments for some time. For example, in 2011, Goldman et al. [10] demonstrated their tool *Collabode* and how this flexibility can be capitalised upon to support concurrent editing of source code and to facilitate true real-time collaboration of software development. Similarly, in their 2014 work, LaToza et al. [26] introduced *CrowdCode* to demonstrate how large software projects might be iteratively broken down and implemented by a distributed and transitory workforce of crowdworkers using a web-based environment. These nearly instantly deployable, low-overhead environments are particularly attractive for crowdsourcing. Instead of requiring workers to already use or download a particular development tool, by using the already familiar and ubiquitous web platform workers are able to participate in programming tasks from almost any device, anywhere, and at any time.

Crowdsourcing offers a large participant pool [12, 22], with diverse demographics and skillsets [44], at low cost and with rapid task turnaround [30, 38]. For commercial requesters, crowdsourcing offers an on-demand workforce [6] coupled with low overheads [23]. For academic requesters and researchers, the highly diverse workforce offers a much broader participant pool than is available when using more traditional methods such as walk-in lab sessions [4]. The additional financial incentive typically offered to crowdworkers can be used to influence which participants choose to undertake the task [18]. The comparatively low cost of crowdsourcing, can also facilitate much larger samples and more iterations that can be carried out [30, 38]. While the remote nature of crowdworkers may require a substantial investment in set-up and task creation, these one-off costs are greatly amortised by the large number of responses which can be gathered, and the low cost of changing and re-running an experiment [2, 9].

Of all the extant crowdsourcing platforms one of the first, and perhaps the most popular [17], is Amazon Mechanical Turk[1]. MTurk is a generic web-based platform which allows task requesters to specify tasks, commonly referred to as HITs (Human Intelligence Tasks), for completion by a large pool of on demand "workers". Mechanical Turk, broadly, offers the task requester two types of tasks: hosted and external. For more complex tasks, the external model allows much more control over task design. Tasks are hosted on a web server of the requester's choosing. This flexibility allows multi-page task designs and for data to be stored outside the MTurk system.

In many cases demonstrating that a human worker is capable of a given task is not the purpose of the research. The goal being to show that a solution or design offered is demonstrably better than previous work. Work by Heer and Bostock [15] on graphical perception has shown the suitability of the workforce in evaluating visualisation techniques. Their work contrasts lab-based evaluation showing the viability of more general artefact evaluation by crowdsourced participants, but specifically in the field of HCI. Offering specific advice for researchers who want to capitalise on this diverse, low-cost participant pool Heer and Bostock [15] highlight the importance of capturing indicative metadata from the workforce where possible, such as display configuration details using JavaScript, as this may influence user preferences and be indicative of user performance.

Prior work has considered the use of crowdsourcing in to engage with a programmers using platforms, including Amazon Mechanical Turk. In their related work, Tunnell Wilson et al. [41] survey developers using multiple-choice questions to determine developer preferences in the context of designing language features. Similarly, Rein et al. [35] used a survey-based approach to determine the comprehensibility of code to novice developers, specifically filtering for this demographic. In an interactive study, also targeting novice

[1] https://mturk.com

developers, Marwan et al. [29] utilise the block-based iSnap tool to investigate the impact of providing hints on developers' progress over time. In contrast, we combine a fully interactive text-based editor environment with extensive instrumentation to vastly increase the richness of the data available for analysis.

Web-based crowdsourcing environments typically provided limited support for this metadata collection. Amazon Mechanical Turk, for example, offers simple key-value storage for the data collected [1]. While requesters may use this to map any number of attributes[2], storage is typically mapped one-to-one with each field submitted by the user. Despite the limited typical use case, requesters are able to store a variety of metadata about the task at hand. However, for complex tasks, the values of interests, such as browser and operating system used, can be extracted using client side JavaScript code that then adds these values to the form at the time of submission[3]. This data is then stored either with the platform provider themselves, or using other off-platform data storage.

3.1 Programming Languages

Different programming languages can have highly distinctive interpretations of the same string of symbols. The choice of programming language that the participants will use is influenced not just by the aspects of the user behaviour or inferred understanding that we wish to explore, but also practical considerations such as those affecting the implementation and which languages we can expect our target demographics are likely to be familiar with.

According to the February 2021 TIOBE index [40]—one indicator of programming language popularity—C, Java, Python, C++ and C# represent the top five languages in use. With the notable exception of Python, these languages all feature very similar syntax and simple code samples in one of these languages are likely to be recognisable to those familiar with another. According to the latest *Redmonk* ranking—an alternative, competing ranking of programming language popularity— from July 2020, the top five languages as follows: JavaScript, Java, PHP, and Python, with C++ and C# in joint fifth place [31]. Again Python sticks out from the other five languages which otherwise share a number of syntactic similarities.

JavaScript, the top language identified by Redmonk and number seven in the TIOBE index, behind PHP, features syntax common with four of the top five TIOBE programming languages, such as the use of curly braces to delimit blocks. As part of its heritage as an interpreted scripting language, JavaScript is typically more forgiving of user error and, for example, allows programmers to assign values to undefined variables due to the nature of its dynamic typing. Despite the unusual prototype-based object model, simple JavaScript functions share many similarities to other C-style programming languages and by limiting exercises to short examples, user exposure to, and required knowledge of, these differences is minimised.

For our purposes, JavaScript also has the benefit of being natively executed by the web-browser. While it is possible to process

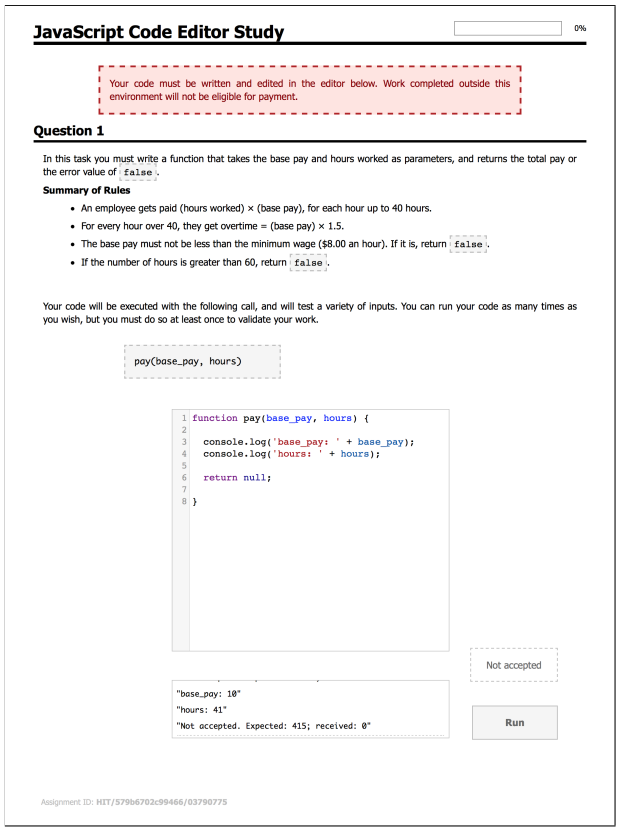

Figure 1: The *conditionals* question as seen by workers with syntax highlighting enabled. Note that the code has been executed, and "Not accepted" as indicated by unit test results in the virtual console area.

other languages using server-side processing, client-side translation techniques, or even more modern browser APIs such as Web Assembly, this may obfuscate issues with the code and add additional complexity to the implementation. Python, for example, may be run in the browser, on the client, using the latter technique [8]. Using the native JavaScript interpreter allows for code to be tested with minimal overheads, discussed below. This popularity, inherent familiarity, flexibility, and runtime support make JavaScript an opportune choice for crowdsourcing programming language tasks. For these reasons, JavaScript was chosen as the language used in our case study.

3.2 Web-Based Editor

Our environment was designed to offer only enough basic functionality to support the development of simple, one "file" programs. The environment provided a text editor, advanced syntax highlighting capabilities, real-time static analysis and error checking of the code, requester specified unit tests, and console redirection for rudimentary debugging support. The system was fully instrumented, capturing all available user interactions and can be seen in Figure 1.

3.2.1 Text Editor. The main component of the environment is the editor. Similar to the aforementioned *CrowdCode* [26], our editor

[2] While neither Amazon's specifications nor the W3C HTML specifications specify a maximum number of fields, technical limitations of the workers' client browser or the Mechanical Turk API may impact the number of values it is possible to submit and store for this particular platform.

[3] While spoofing these values is possible, suspicious values can often be detected. A 2020 study indicated that less than 3% of of users apply such countermeasures [34].

component is based on the open-source *CodeMirror*[4] text editor library. CodeMirror supports a wide variety of programming languages natively including support for indentation and basic syntax highlighting with themes. To facilitate static analysis and more advanced syntax highlighting the *Esprima*[5] parsing library was integrated to the editor.

While it is possible to implement more advanced editors in the browser, for example code-server [6] based on Microsoft's popular Visual Studio Code project [7], these editors offer a less controlled environment. Further, the complex and comprehensive environments may provide additional challenges for the level of instrumentation desired (see Instrumenting the Editor).

3.2.2 Console Output. The console is commonly used to inspect program state in JavaScript and can be used for `printf`-style debugging. To support the workers in this approach, the default `console.log()` functionality was overridden to display the console in the editing environment beneath the code itself. This output area was also used by the unit tests, indicating to the user the success or failure of each test and to provide hints as to why. By displaying this output as part of the interface, workers were not required to understand how their particular browser processes and outputs console logging nor were they unnecessarily exposed to the code which managed the editor itself.

3.2.3 Real-Time Error Checking. To support the developer in cases where they are unsure why the unit test cannot be executed, due to fundamental errors with the code, the *JSHint*[8] tool was also integrated. This allows the user to identify issues with the code that may stop it from executing at all. The output from the tool was shown next to the code with line numbers. Code Mirror natively supports red underlining of errors as supported by some other editors and IDEs. This was disabled for these tests. The decision not to use the potentially more familiar red underlining of the erroneous code with tool-tip style hints was made to allow a consistent style of error messages to be presented to the user whether or not syntax highlighting was enabled.

3.2.4 Instrumenting the Editor. To maximise the ability to recreate the exact user process that lead to the production of the submitted code, all interactions that could be captured unobtrusively were recorded. These interactions included user input device events (mouse movements, clicks, key presses) and system events (code execution, cursor movement, text selection, unit test results). These events were captured by adding `onevent` handlers and inspecting the resulting `event` object or other data structures as appropriate.

Due to the relatively large amount of real-time data captured, using a single upload of these events at the end of the task would result in a long wait period for the worker. As such, data was uploaded asynchronously as the worker carried out the task. This process was managed by a simple API provided by PHP and backed by a PostgreSQL database. Events were recorded both with a server-generated event ID and a client-side specified ID, in combination with the client side reported event time. This ensured that where events were recorded out of order, due to the event driven nature of the JavaScript code and the asynchronous nature of the upload, were able to be reconstructed in the order they occurred.

3.2.5 Replaying the Data. The system was developed to support video-style simulated playback of user interactions, allowing the development process to be reviewed afterwards in real or accelerated time. Even where such playback ability is not desired, this approach ensures that all pertinent data is captured and allows a later recreation of the users interactions. Further, video-style playback of captured data may support ethnographic approaches to the data analysis [13]. The playback component of the system offered real-time playback of the coding process and includes on-screen representations of the mouse movement, click events, and key presses while maintaining the representation of the editor environment in the state that the worker saw it at that point in time.

3.3 Validation

Software development, unlike many other types of tasks routinely crowdsourced, requires a certain skill level before the work can be included in a project or dataset. Unlike studies targeted at beginner or learner programmers, our case study attempts to understand the behaviour of developers who could already complete these basic tasks. To ensure that our participants met this level the task included both automated unit tests, and a survey of participant experience for cross-validation of self-reported skills. By testing workers' contributions against a given specification only workers who meet at least basic levels of programming ability were able to participate in our tasks.

3.3.1 Unit Testing. Testing the code is important not just for the requester, but also for the worker. Understanding why a complete code sample does not meet the required specifications can be difficult, especially where the task is posed only in prose without clear details of expected behaviour in corner-cases or with undefined inputs. Providing deterministic and repeatable tests offers an absolute benchmark for the worker to meet. For the requester, testing the code allows the skill level of the developer to be evaluated, potentially before the code has been submitted. When testing developer skill level, the ability or inability for the worker to pass a particular test can be used as a qualifying benchmark before allowing the worker to continue to more complex work or including their work.

The editing environment supports requester defined unit tests to interrogate the code. Tests were developed to provide a selection of valid, invalid, and corner-case inputs to thoroughly evaluate the developers' submitted work. Submitted code was required to pass the unit test before the worker was permitted to proceed. To make the testing process transparent to the user failed tests were displayed in the environment's console and included both the value provided and the expected output. The submitted code was available for review by the tests before the unit tests were executed. This approach allows code reflection to occur and supports checking for certain undesirable behaviour of the developers, such as coding to the test rather than the specification. For example, the use of a particular "magic number" can be searched for and specifically refuted. However, in our case study, this facility was not used in light of the relatively simple questions that were chosen.

[4] http://codemirror.net
[5] http://esprima.org
[6] https://github.com/cdr/code-server
[7] https://github.com/Microsoft/vscode
[8] http://jshint.com

3.3.2 Survey. To allow further introspective comparison and validation of our recruited participants, and to allow us to understand any unexpected trends or behaviours in our results, a brief survey was presented following completion of the two coding questions. This meta-data is used, where appropriate, to contextualise our results.

- Age: 18–24, 25–34, 35–44, 45–54, 55–64, 65+
- Gender: Male, Female, Other
- Normal coding computer: this computer only, this computer mainly, this computer and others, another computer
- Most used language: Java, Other JVM, C#, Other .NET, Other bytecode, Objective C, C/C++, Other compiled, JavaScript, PHP, Python, Shell (e.g. bash), Other scripting/web, Embedded (e.g. assembly), Others
- JavaScript experience: novice, intermediate, professional
- Years using JavaScript: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10+
- JavaScript editor: Eclipse, Netbeans, Other IDE, Notepad++, Textmate, Other GUI editor, vi/vim, emacs, Other terminal editor, None of these.
- Syntax highlighting (presented as four graphics): unhighlighted dark-mode, syntax-highlighted dark-mode, unhighlighted light-mode, syntax-highlighted light-mode.

4 CASE STUDY

Most modern IDEs, the primary interface with which individuals interact with text-based languages, follow a fairly uniform layout and provide somewhat consistent functionality. However, the process of arriving at this *de facto* standard environment was not through rigorous study, but rather collective reasoning, convention, and fiat.

To demonstrate our approach, we consider one of the most popular augmentations applied to text-based programming languages. This common tool provides a foundation for the exploration of the approach, highlight both the strengths and weakness of running remote, crowdsourced experiments of this type.

4.1 Syntax Highlighting

One technique frequently used to aid human readers to internalise the interpretation of code is syntax highlighting. Syntax highlighting encodes the meaning of the various lexical elements used by the language and systematically emphasises them with a variety of typographical conventions such as colour, typeface, and other micro-augmentations [19]. Syntax highlighting is an almost universal feature of text-based code editors and is even used in consumer-facing end-user products such as Microsoft Excel [43], which uses both coloured parenthesis matching and typographical convention, such as automatic uppercasing of function names, to infer meaning to the user. While syntax highlighting is almost universally applied by code editors, evidence in the literature is sparse for the efficacy of this practice.

One study that offers some evidence for the effectiveness of these techniques was reported by Sarkar [36]. In this read-only study from 2015, participants were asked to compute the results from Python functions, given a variety of arguments, for both highlighted and unhighlighted code. While the study showed some limited efficacy, the effectiveness of the treatment decreased with proficiency. Likewise, Dimitri [7] suggests that syntax highlighting has a positive effect on code completion time among users of the *Sonic Pi* environment[9]. However, follow-up work by Beelders and du Plessis [3] from 2016, who studied the comparative effect with C#, and Hannebauer et al. [14] from 2018, who considered novices using Eclipse, showed no statistical significance between code presented with and without syntax highlighting.

Curiously, the authors of these studies suggest both a positive correlation between syntax highlighting and user experience [3] and a diminishing one [7]. Further, even studies utilising larger samples highlight challenges with the possible small effect size when dealing with such treatments, further suggesting that proper evaluation may require larger sample sizes [14]. These mixed and constrained findings draw attention to the still inconclusive status of work in this area.

Some of the challenges associated with recruiting both sufficiently large, and sufficiently representative, samples from developers are discussed in Alternative Approaches. A web-based crowdsourced approach may offer a potential avenue to explore the complex interactions posited by these works. In fact, Beelders and du Plessis [3] capitalised on the flexibility offered by web-technologies to enhance their in-lab study.

4.1.1 Hypotheses. To further explore these confounds, and with particular regard to syntax highlighting, we posit the following hypotheses:

H1: Syntax highlighting improves coder efficiency through reduced code composition time.

H2: Syntax highlighting improves coder efficacy through reduced errors in resulting code.

4.2 Task Choice and Design

The difficulty of the tasks set may affect which conclusions can be drawn from the approaches taken by the developer. Trivial problems, such as correcting a "typo" will encourage non-programmers to attempt the task while limiting the variety of solutions that might be expected of experienced developers. Equally, where a wide-net is cast, such as our web-based approach, the tasks must not be so complex as to require extremely specialised knowledge that might be lacking in the general developer population.

One well-tested source of beginner level coding problems that both demonstrate a basic ability to program and provide scope for interesting solutions are MOOCs. While collecting data from actual MOOCs may be problematic as it self-selects for beginner programmers, as outlined in Alternative Approaches, these superficially simple questions offer broad scope for inventive or unexpected solutions.

The *MIT OpenCourseWare Introduction to Programming in Java* course [21] provides a selection of short, well-tested assignments. While the first assignment, a gravity calculator, depends on mathematical understanding of the problem, assignments two and three offer two typical data processing problems which can be adapted for the JavaScript programming environment: conditionals and iteration.

Conditionals: Pay Calculator. This problem requires the developer to calculate the correct wage for a given number of hours and pay

[9] https://sonic-pi.net

level following a series of rules. To make the problem easier to unit test a very basic function stub was provided and the worker asked to return a special value of false instead of printing errors as in the original question. The modified summary of rules is as follows:

- An employee gets paid (hours worked) × (base pay), for each hour up to 40 hours.
- For every hour over 40, they get overtime = (base pay) × 1.5.
- The base pay must not be less than the minimum wage ($8.00 an hour). If it is, return `false`.
- If the number of hours is greater than 60, return `false`.

The problem can be solved simply using a series of conditionals, catching the error cases and calculating the wage, including overtime compensation if applicable.

Iteration: Fastest Runner. In this question the developer is given two arrays, one of names and and one of times (run duration), and asked to identify the fastest runner. Similar to the first question, to ease unit testing the iteration problem was modified from requesting the developer to print the output to instead returning the name. While the original problem provides a program stub including the hard coded values and a skeleton `for` loop, the modified task used an implementation agnostic function stub with two parameters, an array of `names` and `times`. Conceptually not dissimilar to that of Sarkar [36], this approach allowed the code to be tested multiple times and ensured that the developer was unable to simply work out the correct answer and hard code a return value.

Typically this task is approached iteratively, looping through the array of names and times and noting the index of the minimum. The question as posed in the original assignment offered a basic `for` loop to encourage this approach. As the provided stub did not include a loop, more advanced software developers were not unduly dissuaded from providing novel alternative solutions to the problem not anticipated by the requester.

4.3 Procedure

Two-hundred participants were recruited using Amazon Mechanical Turk. The task was listed with a maximum completion time of two hours, in which to carry out both of the two questions posed, however, workers were not expected to take this amount of time provide working solutions. Workers were offered $1.00 USD for completion. Regardless of the the warning seen in Figure 1, all participants were paid.

Workers were introduced to the task and asked to agree to their participation in the experiment, as required by our ethics proceedure. Before beginning the task, workers were shown a modified program stub from a third *OpenCourseWare* question, as an example, and told they would need to add functionality to program stubs with similar complexity.

The task was configured with the two questions, one with syntax highlighting and one without. Questions order and which of the two questions to highlight were counterbalanced across the participants to account for any ordering effects of both the treatment and question.

Once each question had completed loading, all user interactions from the keyboard and mouse were recorded and sent back to the server. Workers were required to execute their code at least once, at which point it was unit tested. If the unit test failed, the worker was notified of these errors in the on-screen console. This provided them with both the expected value for each test and the data provided by the code. Workers were then required to revise the code until the test passed. On successful execution, activity recording was stopped and the user could optionally pause, taking a break before moving on to the next question.

Finally, the workers were asked to complete a short survey regarding their development environment and coding experience. The questions included, age, gender, which computers they use to develop software, their most used programming language, their self assessed level of JavaScript proficiency, their period of that experience, the editor they use most, and what type of syntax highlighting they typically use as outlined previously (see: Survey).

5 RESULTS

Our case study offers a lens with which to evaluate our remote, crowdsourced approach to studying programmer behaviour. However, the results of the syntax highlighting study itself provide a touch point with which to formulate an understanding of who these developers are and to ground our approach. The results of this case study are considered herein.

Of the 200 workers, 30 submitted the task without completing both questions, and the data for a further 9 indicated completion times in excess of the two hour maximum allotted for the task. While workers code was tested to ensure completion, the system recovered from termination of the task (such as a browser refresh or exiting the tasks) by advancing to the next question. This introduced scenarios where workers who had not entirely completed a question, or where the task was "resumed", to advance to the next question. This resulted in a post-filtering dataset of 161 workers who completed both of the set questions.

The majority of participants indicated being male, 87%, with 12% choosing female and 1% selecting "Other". The most frequently used languages by the participants was mixed, with 29% preferring Java, 15% C and C++, 13% Python, 12% C#, 10% JavaScript, 9% PHP, and 12% preferring other languages. 40% of participants described themselves as novices with JavaScript, 49% intermediate, and 11% professionals. The mean length of JavaScript experience, in years, for each category was 1.42, 3.44, and 6.28 respectively.

Regarding their development environments, 38% of participants indicated that the computer they used for the task was the only one on which they wrote code. 50% indicated using two or more computers, including the task computer (15% this mainly; 35% this and others) and 12% indicated that they used another computer to write software. 93% of participants indicated that they used some kind of syntax highlighting (59% light background; 34% dark background), compared to 7% using an editor that does not highlight their code (6% light background; 1% dark background).

5.1 Task Completion Time

Hypothesis 1 for the syntax highlighting cases study considered task completion time, anticipating shorter completion times for highlighted compared with unhighlighted code. The overall mean completion time per question was 558 seconds. Overall, the mean

Table 1: Summary of mean completion times in seconds. Standard deviation (SD) is shown in parentheses.

	Highlighted	Unhighlighted	Combined
Both Questions	534 (390)	583 (448)	558 (417)
conditionals	546 (399)	558 (417)	552 (407)
iteration	522 (382)	609 (471)	564 (428)

completion time for questions presented with syntax highlighting decreased: 534 seconds for highlighted questions and 583 seconds for unhighlighted questions. This also held true for each question presented. For the conditionals question mean completion times were 546 seconds highlighted and 558 seconds unhighlighted; for the iteration question 522 seconds highlighted and 609 seconds unhighlighted. This data is summarised in Table 1. However, despite this apparent positive effect, and in contrast to some prior work (i.e. [36]), there was high variability in completion times between users and a one-way ANOVA with repeated measures indicated no statistical significance between the log-transformed completion times of highlighted and unhighlighted solutions ($F_{1,160} = 1.303$, $\eta_p^2 = .008$, $p = .255$). Similarly, one-way ANOVAs for the log-transformed completion times for both the conditionals question ($F_{1,159} = 0.032$, $p = .859$) and the iteration question ($F_{1,159} = 1.951$, $p = .164$) indicated no statistical significance.

5.1.1 Possible learning effect. A deeper look at the completion time data indicates that the mean completion time for workers was lower for the second question, irrespective of the order in which questions were posed and whether or not this question was highlighted (first question, 605 seconds; second question, 511 seconds). This held for all possible question sequences, for example where first question was conditionals, highlighted (564 seconds) the worker then would complete iteration, unhighlighted for their second question (511 seconds). It might be conjectured that workers took some time to familiarise themselves with the environment, the expectations of the task, the precise semantics and syntax of JavaScript, or simply to get into "the zone". This anomaly points to the need, especially where a within subjects design is sought, to have an additional preparatory question or "training" period to familiarise the worker with the task, system, and expectations. The extent of this training is an important experimental design decision [24]

5.2 Programmer Interactions

The comprehensive instrumentation included in the task allows a more in-depth exploration of worker activity while completing the task. Ethnographic approaches, such as reviewing the simulated playback of the coding session can offer a personalised view of user activity and process. Equally, automated techniques can be applied to the individual inputs collected. Mouse trace visualisation offers some limited insight into user focus during the task. Keystroke-by-keystroke data offers potential insights in to the editing process and activity correlations which cannot be gathered from solutions nor easily understood from reviewing the simulations alone.

5.2.1 Simulated Playback. The instrumentation included in the task captured all significant events and was stored as to allow both real-time and accelerated playback of the coding process. In total over two days of user activity was recorded. A review of a small sample of the simulations found that much of the development time was spent reading the question specification, thinking on the problem, and attempting to understand code execution process. Each of these activities was interspersed with intense periods of actual development time, where the worker introduced code snippets either by typing them in or copy-and-pasting them from alternative sources and modifying them to the needs of the specific problem. The comprehensive instrumentation offered the ability to identify how code was introduced to the editor. In particular, this copy-and-paste behaviour was anticipated and was not prevented.

Users were implicitly permitted to source partial or representative solutions and modify them to their needs. Events were recorded both when users pasted in the code, but also when the editor was backgrounded. While it was not possible to capture from where the user retrieved these code samples, it might be assumed that the users copied the code from an electronic source and other websites are a likely candidate. The developers then would go on to modify the copied code to suit the specific problem. This type of copy-and-paste and modify is typical of modern developer behaviour [5]. The source, or an approximation of the source[10], can be inferred by carrying out a web-search for the code snippet as introduced. As the questions were modified from the original *OpenCourseWare* questions and the required implementation language changed, workers were unable to source code samples solving the presented problem exactly.

Despite the relatively small size of the problems presented to users, the large quantity of generated simulations and sporadic nature of the development process does not lend itself to consistent visual analysis. However, simulations offer unique insights into specific or unusual implementations generated by workers. Reviewing just the completed code demonstrated that some users employed interesting techniques to solve the questions. For the *iteration* question, eight workers demonstrated the use of `Math.min.apply()` and one worker solved the question with a single line of code. An understanding of how the developer arrived at their solution would be lost had only the final working code been captured or reviewed. However, with comprehensive instrumentation it may be possible to developer deeper insights into the developers' thought processes. This worker, for example, considered an iterative approach using a `for` loop, before pasting `Math.min.apply( Math, array );` from another source and adapting it to their needs (see Table 3). This play-by-play approach offers far improved level of granularity over some alternative techniques, such as post-hoc analysis of versioned source code repositories, such as git.

5.2.2 Mouse Usage. In addition to being available in the playback data, numerical data was also compiled concerning mouse movements and hover durations. This approach offers more condensed metrics which do not rely on reviewing the individual sessions. Overall, the mean distance travelled by the users was 30,364 pixels over a period of 188.9 seconds during which they clicked the mouse 43 times. Due to the resizable nature of the interface, and ability

[10] Many websites now duplicate the content of *StackOverflow* and other popular programming code exchanges making precise identification of a source problematic. However, the specific citation may be of less interest than the likely surrounding content which lead the worker to select a particular implementation on which to base their own.

Table 2: Mean location and click data for mouse interactions. Standard deviation (SD) is shown in parentheses.

Area	Distance (px)	Duration (s)	Clicks
Question	2,826 (3,761)	37.59 (93.07)	—
Editor	13,359 (15,223)	292.73 (234.48)	24.0 (31.0)
Console	2,243 (3,344)	35.76 (58.04)	8.6 (21.8)
Errors	254 (818)	1.95 (14.87)	—
Others	11,682 (12,260)	190.12 (207.43)	—
Total	30,364 (30,627)	558.16 (417.16)	43.0 (58.5)

to consult external resources, this metric is impacted by the users screen- and window-size and encodes many indivisible aspects of the user behaviour and environment.

To compile these figures the raw data from each mouse event was compared with pre-computed browser geometry and layout data for each question. In addition to knowing how long the user spent moving the mouse, how far, and to what, it is important to know how much dwell time was given to each area. While the markup of the pages which comprise the editor environment contain many distinct areas, only four are directly implicated in the development of the code: the question posed, the editor area, the console for debugging, and the real-time error display. This data is summarised in Table 2.

It is important to consider that of the four areas on interest the *error* area was dynamically displayed only when errors were presented to the user. To analyse whether the user was moving the mouse to inspect a given error, the display status of the error area was validated against the recorded code. Only when the error area would be displayed to the user were distance, time, or clicks attributed to this category.

5.2.3 Keystrokes and Editing. While visual analysis may offer additional insights for selected work sample (such as unusual solutions) or a given type of developer (for example, active mouse cursor users), systematic analysis of the code typed can be used to evaluate the impact of a given treatment of the editor space. Each keystroke, or editing action, has a measurable impact on the code. Individual keystrokes can be composited into a cohesive action. For example, a series of characters followed by a number of backspaces and an equal number of characters indicates a correction of a mistyped character. By considering the difference between the initial string and the corrected string, the intent of the developer can be identified. Similarly, the insertion of characters out of sequence, for example typing a logical expression followed by the insertion of one or more parentheses may reveal information about the developer's approach and process.

In analysing the keyboard data, the act of developing the code was considered through the lens of a *CRUD* system interaction. Each character on screen could be *created*, *read*, *updated*, or *deleted*. Keyboard events indicated three of these events, workers could type a character into the editor, replace a character (using either a select and type process, or an overwrite mode), or delete a character (using delete, backspace, or cut, potentially using select in combination with these actions). The developer also has two methods of inputting code: using the keyboard explicitly, or pasting one or more characters into the editor. This creates situations where each of these editing actions can occur in either single character or as multi-character events.

Table 3: Subset of editing data for *UserID* 489, question 2. Note the corrective edit made during events 469–475. The worker types, `I`, backspaces, and corrects their entry to `i`. Some time later, the worker replaces the text from position 83 (comprising the aborted `for` loop) with a pasted code snippet `Math.min.apply( Math, array );`.

Event	Time	Type	Position	Size	Content
…	…	…	…	…	…
440	70.608	type-insert	83	1	`f`
443	70.857	type-insert	84	1	`o`
446	70.967	type-insert	85	1	`r`
449	71.231	type-insert	86	1	␣
453	73.107	type-insert	87	1	`(`
456	74.161	type-insert	88	1	`v`
459	74.266	type-insert	89	1	`a`
462	74.399	type-insert	90	1	`r`
465	74.517	type-insert	91	1	␣
469	75.582	type-insert	92	1	`I`
471	76.138	type-delete	92	1	`I`
475	76.360	type-insert	92	1	`i`
478	76.889	type-insert	93	1	`;`
481	77.141	type-insert	94	1	␣
707	209.087	paste-over	83	30	`Math.min.apply( Math, array );`
…	…	…	…	…	…

User activity data was processed to tag each of the identified events. The script tagged each event as one of the six identified input methodologies: type-insert, type-over, type-delete, paste-insert, paste-over, select-delete. Table 3 provides an example of the tagged data. The tagged data was further processed to identify edits made by the user by noting the sequence and positions of the changes. These events can be summarised and noted as *in situ* edit events. This particular user carried out three of these corrective edit events, with a total of four characters affected, that is, the worker carried out two single character replacements (one of which seen in events 469–475) and a single two-character replacement where two characters were deleted before two characters were inserted at their respective positions. A little more than two minutes after this edit, the worker abandoned this iterative approach, pasting a 30 character code snippet over the content starting from position 83, the beginning of the incomplete `for` loop (event 707). Note that there were no other events which edited the code during this intervening time; discontinuous event IDs indicate other activity, including mouse movement, cursor positioning, and records of time spent away from the editor window.

5.3 Code Efficiency and Errors

Due to the inclusion of the unit test, all participants who completed the task did so to this objective measure of success. To further distinguish code and developer quality we considered two metrics: code performance and development-time execution errors. The former offers an objective measure in the form of the time efficiency of the

provided code, while the latter suggest insight into the developers' propensity to submit code in a known non-working state.

5.3.1 Code Performance. To evaluate the performance of the code produced by the workers, each code sample was executed by timing the code when executing the same unit tests the workers were required to pass. The tests were each conducted 100 times for each code sample, and an average taken to minimise the effects of external factors on the reported time. The mean execution time, for all 322 code samples, was 75.52 ms. For the individual questions the mean execution time was 68.83 ms for the conditionals question, and 76.22 ms for the iteration question with a range of 60–80 ms for both questions.

Execution speed is highly dependant on a number of variables, not only the hardware being used but also the software environment used to compile or interpret and execute the code. These speed tests were carried out on an Apple iMac with a 2.9 GHz quad-core Intel Core i5 processor and 8 GB of RAM. The code was executed with Node.js 0.12.0[11] which uses the v8 JavaScript engine, version 3.28.73. The v8 engine is the JavaScript processor included with Google Chrome, the browser used by 69.6% of our participants. This version of v8 is contemporary with the data collection and, specifically, the version included in the versions of Google Chrome used by 66.1% of our participating Chrome users.

5.3.2 Executions and Errors. In addition to the speed of the code, the number of executions and the errors produced by the workers might offer some insight into developer experience. JavaScript is a fairly forgiving language in that many syntactic features are optional, not least that offered by ASI (automatic semicolon insertion). This flexibility means that while not all errors are terminal, they might indicate interesting user behaviour.

On average, participants executed their code 4.47 times per question. The code was constantly evaluated using JSHint by the editor environment and, as previously described, users were shown these errors to the right of the editor area. To determine when errors were being overlooked, or intentionally ignored by users, a subsequent review was carried out using JSHint for each occasion the user chose to run their code. In total, the mean number of errors displayed per execution was 1.82 and 14.9% of users executed their code with one or more errors displayed. Users that did choose to execute their code with errors displayed did so 10 times on average, and the average number of errors displayed to these users on each occasion was 12.23.

5.4 Syntax Highlighting

The design of the study was intended to expose effects relating to syntax highlighting. While basic measures do not present statistically significant differences between our treatments with regard to hypothesis 1, there may be indicators to differences in worker behaviour worth further investigation. A summary of some metrics of interest can be seen in Table 4. Hypothesis 2 considered coder efficacy through reduced errors, again we see marginal reduction in the mean. However, a repeated measures ANOVA ($F_{1,160} = 0.167$, $\eta_p^2 = .001$, $p = .684$) does not indicate statistical significance.

[11] https://nodejs.org/en/download/releases/

Table 4: Summary of collected metrics, by syntax highlighting status. Mean values or percentage of participants.
Note that for the purposes of error calculations, *attempts* indicates the number runs which failed to pass the unit tests, plus one.

	Unhighlighted	Highlighted
Completion time (s)	582.77	533.55
Execution speed (ms)	72.20	72.84
Attempts	4.56	4.37
Editor exits	3.8	3.5
Exit duration (s)	70.81	71.34
Paste events	59.32%	56.21%
Key presses	489	447
Mouse clicks	44.6	41.5
Mouse moment (px)	31532	29196

5.5 Questions and Sequencing Effects

Participants completed the two coding questions in a random order and with random assignment of the syntax highlighting. The two questions were design to test distinct logical constructs, and as such are considered separately below. Further, as noted earlier, an apparent ordering effect is notable in the completion time is also explored.

5.5.1 Question Type. Each question tested a different approach to problem solving and programming ability. As such different behaviours, completion times, and code execution speeds can be expected. As each question was completed by all of the included 161 workers, many of the reported details (e.g. years of experience) are the same for all combinations. These are summarised in Table 5.

Participants produced working solutions to the conditionals question slightly faster than for the iteration question, and took fewer attempts. This may suggest the slightly higher complexity of the iteration question. Further impressing the complexity, workers spent far longer consulting external resources for the iteration question, exiting the editor environment twice as frequently.

However, workers expended far more mouse movements over the question area for the conditionals question. Recognising that users can be observed tracing the instructions, this can be expected due to the longer question text. This suggests that workers felt

Table 5: Summary of collected metrics, by question type. Mean values or percentage of participants.

	Conditionals	Iteration
Completion time (s)	552.24	564.08
Execution speed (ms)	68.83	76.22
Attempts	3.99	4.37
Editor exits	3.8	4.9
Exit duration (s)	42.06	100.09
Paste events	57.14%	58.39%
Key presses	516	420
Mouse clicks	43.0	43.1
Mouse moment (px)	30424	30304

Table 6: Summary of collected metrics, by coding "session". Mean values or percentage of participants.

	Session 1	Session 2
Completion time (s)	604.94	511.38
Execution speed (ms)	72.70	72.35
Attempts	4.73	4.20
Editor exits	3.8	3.6
Exit duration (s)	70.95	71.20
Paste events	61.49%	54.04%
Key presses	502	434
Mouse clicks	45.2	40.9
Mouse moment (px)	32599	28129
- *Question area*	3000	2652
- *Editor area*	14414	12304
- *Console area*	2467	2020

greater need to consult the rules that must be followed for the iteration question. This increased mouse movement in this area further lends credence to the use of the mouse pointer as a proxy for interest, even in non-interactive areas. The increased verbosity of the conditionals question is reflected in the higher keyboard usage in the conditionals questions. This may represent the more verbose solution, treating each rule independently, that was typically used.

Other worker behaviour was extremely similar between the two questions. Use of pasted code snippets, total mouse movement, and in particular number of mouse clicks are all very similar between the questions.

5.5.2 Question Order. The order of tasks presented can have predictable, and unpredictable, impact on user behaviour. The potential for learning effects and increased environmental familiarity should always be considered. While the order of the questions presented was randomised, the questions were presented sequentially immediately after one another in one of two "sessions", detailed in Table 6.

Workers produced working solutions to the second questions faster than for their first questions, and took fewer attempts. As suggested earlier, this may be indicative of a learning effect in play as workers gain familiarity with the editor environment. This is further suggested by the small, but measured gains in code execution speed, decreased typing, fewer consultations with external resources—though for fractionally longer—and a decreased use of pasted code. With a small expected effect size the variability introduced by the improved familiarity with the requirements of the task may have impacted detectability of changes to behaviour. In a future study, more extensive and interactive training may mitigate any potential learning effects.

5.6 Survey Data

Techniques which allow the large volume of raw data to be processed with a degree of automation can be used to create summary data about the workers as a whole. Some of these data points have been noted in the preceding text. By combining this with the responses to the survey a picture of developer behaviour can be stratified across a variety of parameters such as experience, gender, or country. Additionally, the approaches taken to the different questions can be considered: conditional and iteration; highlighted and unhighlighted; and as touched on earlier, question order.

Table 7: Summary of collected metrics, by self-reported experience level. Count, mean values or percentage.

	Novice	Intermediate	Professional
Number	64	79	18
Years experience	1.42	3.44	6.28
Completion time (s)	579.43	560.13	473.88
Execution speed (ms)	73.55	72.35	69.67
Attempts	4.64	4.45	3.92
Editor exits	4.0	3.5	3.4
Exit duration (s)	83.63	63.14	61.28
Paste events	54.69%	58.86%	63.89%
Key presses	534	436	377
Mouse clicks	37.5	46.5	47.7
Mouse moment (px)	29948	30495	31267
- *Question area*	2510	2828	3939
- *Editor area*	13296	13558	12709
- *Console area*	1888	2465	2534
Use dark theme	32.8%	34.2%	44.4%

5.6.1 Experience. Workers were asked to self-classify their JavaScript experience into one of three levels: novice, intermediate, or professional. Workers were additionally asked the number of years of experience they had programming with the language. Of the 161 included workers, the mean reported value was 2.96 years. When comparing the experience levels of the developers, there are distinct differences in their survey responses and behaviour. These are summarised in Table 7

For workers who report a higher level of experience, the average years of experience reported also increases. As might be expected, more senior developers took less time to complete their questions and demonstrated faster code taking fewer attempts to create a successful program. More senior developers also wrote their code using fewer keystrokes, however interestingly they also used the mouse more. This mouse usage was higher in the question area and console area commensurate with experience level. Finally, regarding the user environment, the fact that detected Chrome usage decreased and Firefox usage increased with self-reported developer seniority, might serve as an interesting anecdote.

5.6.2 Gender. While the impact of gender on developer behaviour has undergone limited study [28] and reporting can be somewhat controversial [37], asking workers their gender is a common feature of crowdsourced studies and as such is included here. While female participants were the smaller group, just 19 compared to 140 male, this may be a reflection of a population bias on MTurk rather than a measure of a specific bias among the developer population of the platform. Metrics are summariesd in Table 8. In addition, two participants chose "Other" as their preferred gender option. These participants are not included in the table and analysis in this subsection due to the very small sample size.

Table 8: Summary of collected metrics, by self-reported gender. Mean values or percentage of participants.

	Female	Male
Years experience	2.21	3.09
Completion time (s)	548.31	555.80
Execution speed (ms)	73.68	72.43
Attempts	4.00	4.51
Editor exits	5.0	3.4
Exit duration (s)	83.63	66.99
Paste events	46.05%	59.46%
Key presses	400	477
Mouse clicks	42.5	43.2
Mouse moment (px)	29719	30206
Use dark theme	15.8%	36.4%

While male workers managed to produce marginally faster code on average, this might be expected due to their higher reported years of experience. However, female developers completed their work faster, with fewer keystrokes, and in fewer attempts. Female developers pursued other resources more frequently, and for a longer duration but were less likely to introduce pasted code. This might indicate an increased adherence to the instructions to complete work in the editor, and is further supported by their reduced likelihood of running the code with an error message displayed. Just 10.5% of female workers did so compared to 15.4% of males.

6 DISCUSSION

Participants produced working solutions to the highlighted questions faster than for unhighlighted questions, and took fewer attempts. While these results show some promise, as detailed previously, these figures have high variability and cannot be statistically generalised to the population. Overall, highlighting appears to have low influence on the measured behaviours, with fairly low changes in keyboard and mouse use, commensurate with increased task duration. This may be indicative of a small effect size that would need increased sample size to properly detect. The unexpected, but extremely small, difference in code execution speed may warrant separate further investigation. This may offer avenues for future work, such as whether a lack of syntax highlighting might have reproducible bearing on the efficiency of generated code, and what mechanisms might be at play.

To ensure our participants provided a minimum standard of code, we utilised unit tests. This offered an objective filter for users to be included in our dataset. Where contributions must more complex and acceptable solutions cannot so easily predetermined, filtering could be carried out using an unrelated task with well defined unit tests to demonstrate ability. Alternatively, if appropriate, participants might be encouraged to follow a test-driven-development approach; thereby having participants produce both the tests and code that passes these tests, exemplifying the formal specification of the project.

While our environment is constructed from standard components, together it represents a bespoke environment. As noted in the section *Web-Based Editor*, we consciously chose to deviate from alternative implementations and layouts to support a more controlled and fully instrumented environment. As suggested in our results, the unfamiliar interface may have had a hand in the consistently higher completion times for the first of the two coding questions and the possible learning effect we report. However, an alternative hypothesis may be that developers take some period to get into "the zone." This confound, no matter its cause, bears further exploration.

Despite the constrained interface for case study experiments, one minor trade-off made to improve the user experience was the use of a resizable interface. While fixed dimensions and layouts are possible in web-based environments, they are not typically considered best practice [42], and further detract from more typical editor environments. However, allowing resizing did little to make our bespoke editor presentation more familiar to participants and required additional analysis techniques, including where workers re-sized the browser part way through the study.

As noted, the large amounts of data being captured, including user input device events (mouse movements, clicks, key presses) and system events (code execution, cursor movement, text selection, unit test results), resulted in not-insubstantial data payloads. Even limited concurrent testing revealed that real-time uploading of this data may negatively impact the performance of endpoints due to overheads inherent in traditional AJAX calls. While alternative approaches, such as web-sockets, are available we elected to use the more established approach. To abate the transfer issues identified, data was uploaded to the server asynchronously in periodic batches using the *jQuery*[12] library. This approach avoided hundreds of hits to our endpoint yet, by utilising upload capacity throughout the experiment, minimised the waiting time at the end of the coding session that might otherwise be experienced by users with slower internet connectivity.

A final technical consideration which affected the user experience was how tasks could be resumed. Crowdworkers may choose to operate on many tasks at once [25], and switch between these tasks [11]. To prevent participants loosing their work, but also from preparing answers in advance away from the environment, the implementation caused questions which were revisited (such as due to a browser refresh, or picking up the question in a latter HIT) to be skipped. This approach ensured that workers were paid, but introduced these spurious cases with only partial worker activity and incomplete solutions which needed removal from the dataset. Exactly how these challenges are handled must be carefully considered, and may, in fact, be platform dependant.

6.1 Threats to Validity

Our case study recruited 200 participants, of which 161 provided complete data, with a broad spectrum of self-reported skill level and whom successfully completed two software development tasks to an objective, predetermined standard. While our case study demonstrates that crowdsourcing can offer an interesting and efficacious source for studying developer behaviour at a, there are a number of notable limitations that are important to consider. Most notably, are crowdworkers "real" programmers?

As with in-lab, or in-class, studies, it is important to consider how representative the sample of developers is before generalising any results. Crowdsourcing platforms are no exception. While our

[12] https://jquery.com

developers demonstrated capability to solve these relatively simple challenges, and utilised a number of interesting techniques, they may not represent the type or skill-level of the developer of interest—or the population of developers as a whole. Whether or not these respondents are suitable for a given evaluation will depend on the technique, tool, or feature, under investigation. While the scale of crowdsourcing offers scope for pre- and post- filtering of responses to achieve the desired composition, it is especially important to validate any assertions made by anonymous, remote participants.

The remote nature of these experiments presents additional challenges for the experimenter compared to in-person studies. Techniques such as eye-tracking or speak-aloud approaches are much more challenging to implement remotely and may be afflicted by severe technical limitations. For example, as noted, the ability for participants to switch back and forth between tasks and lack of control over the environment external to the browser window–both virtual and physical–may result in confounds which may not be sufficiently ameliorated by the scale it is possible to achieve.

Where our approach offers tangible benefits, however, is in the richness of the data that is collected. Yet, it is important that studies of this type do not confuse random correlations in the data with rigorous hypothesis testing. The large amounts of data that are collected may facilitate numerous approaches to analysis that may indicate significant effects. While this comprehensive telemetry data may be used to identify areas of interests for future targeted experiments, it is important that normal standards of scientific rigour are applied to any conclusions drawn; most importantly including avoiding over testing of the data.

The size of the tasks undertaken were design to be relatively small. This achieved two goals: i) short experiment turn around, and ii) minimise sunk cost for unqualified speculative participants. As unqualified participants were able to begin the task, quickly allowing these individuals to assess their suitability was an important factor in our approach. The small code size of our questions is not representative of large, real-world projects. However, it is consistent with prior experiments in the literature (e.g. [36]). While this may be suitable to investigate fundamental concepts, in isolation, coding challenges of this size may not be suitable to consider more complex interactions; potentially including syntax highlighting investigated in our case study. For studies which require larger programming challenges, or ones which attempt to investigate collaborative environments (e.g. [10]) it may be necessary to prequalify programmer ability, potentially through a similar small qualification study and later invite these participants to complete these more complex experiments.

Whether or not our participants were objectively good, or representative of these self assessed groups remains an open question. Our approach, using unit tests, provided an objective benchmark for completion, one that is used in commercial environments. However, passing these tests does not indicate the quality of the code or skill of the developer. For example, as seen in the accompanying data, when looking to identify the smallest number in an array, User ID 442 chose to set the initial minimum to `999999999` (rather than, for example, a value from the array). While all of our test cases did in fact have a minimum below this value, which allowed their code to pass these tests, that is not true of all possible input arrays and as such this code could not be considered objectively good. Whether or not any given developer is "good", or even suitable for an individual experiment, remains an open question. However, this concern is not a preserve of crowdsourced studies.

It may be argued that the very nature of low-paid work may attract a lower-skilled workforce, unable to achieve improved pay elsewhere. To avoid this, our approach ensured that work met the aforementioned unit tests to ensure a minimal level of ability had been demonstrated. While the positive correlations shown between objective measures of performance and user provided survey responses suggests that self assessment was representative of objectively measured skills, our case study collected these responses after the coding exercise. This allowed respondents to gauge their ability against their recent work. As such, it is unknown whether self assessment would be suitable as a pre-task filter; in fact, any such approach may well encourage inflation to access to the task.

Finally, there may be concerns around the level of payment made to the respondents. Anecdotally, unprompted and out-of-band messaging from participants indicated that, at least some, were positively engaged, were open to further programming studies, and had substantive commercial programming experience at the payment level. However, while the commercial nature of crowdsourcing, and the open-market nature of these platforms allow workers to skip poorly paid tasks we would expect replications of this approach to ensure that payment levels meet required ethical standards and national minimum wages as appropriate. Further, while the on-demand commercial basis of the relationship between the worker and requester does not fully negate accusations of coercion, crowdsourcing does lack the ability to compel individual participation from those who have not already engaged in the study, something that might be of concern when carrying out similar experiments with students in educational settings.

7 CONCLUSIONS

In this paper we have presented our approach to studying programmer behaviour, at scale, using web-based crowdsourcing and specifically, Amazon Mechanical Turk. We have demonstrated that both large scale and detailed datasets can be gathered using highly instrumented developer environments, from across a diverse range of developers at various skill levels. We have highlighted a number of important design considerations, including technical limitations, user concerns, and key features of experiment design including the application of appropriate filters and controls. Our results validate our approach, demonstrating predictable findings and correlation between programmer experience and objective measures of code quality. Finally, we highlight a number of interesting avenues for future work as well as considerations and mitigation against some of the challenges limitations inherent to large scale crowdsourced studies of this type.

ACKNOWLEDGMENTS

This work was funded by an EPSRC studentship and EPSRC grant EP/R004471/1. Data available at https://doi.org/10.17863/CAM.66593.

REFERENCES

[1] Amazon. 2014. Amazon Mechanical Turk API Reference. http://awsdocs.s3.amazonaws.com/MechTurk/latest/amt-API.pdf

[2] Vamshi Ambati, Stephan Vogel, and Jaime G. Carbonell. 2010. Active Learning and Crowd-Sourcing for Machine Translation. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC 2010)*. ELRA, 2169–2174. https://works.bepress.com/jaime_carbonell/169/

[3] Tanya R. Beelders and Jean-Pierre L. du Plessis. 2016. Syntax Highlighting as an Influencing Factor When Reading and Comprehending Source Code. *Journal of Eye Movement Research* 9, 1 (2016). https://doi.org/10.16910/jemr.9.1.1

[4] Tara S. Behrend, David J. Sharek, Adam W. Meade, and Eric N. Wiebe. 2011. The Viability of Crowdsourcing for Survey Research. *Behavior Research Methods* 43, 3 (March 2011), 800. https://doi.org/10.3758/s13428-011-0081-0

[5] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 1589–1598. https://doi.org/10.1145/1518701.1518944

[6] Djellel Eddine Difallah, Michele Catasta, Gianluca Demartini, Panagiotis G. Ipeirotis, and Philippe Cudré-Mauroux. 2015. The Dynamics of Micro-Task Crowdsourcing: The Case of Amazon MTurk. In *Proceedings of the 24th International Conference on World Wide Web (WWW 2015)*. ACM, 238–247. https://doi.org/10.1145/2736277.2741685

[7] Giovanna Maria Dimitri. 2015. The Impact of Syntax Highlighting in Sonic Pi. In *PPIG 2015 - 26th Annual Workshop*. 12.

[8] Michael Droettboom. 2019. Pyodide: Bringing the Scientific Python Stack to the Browser - Mozilla Hacks - the Web Developer Blog. https://hacks.mozilla.org/2019/04/pyodide-bringing-the-scientific-python-stack-to-the-browser/

[9] Yotam Gingold, Ariel Shamir, and Daniel Cohen-Or. 2012. Micro Perceptual Human Computation for Visual Tasks. *ACM Transactions on Graphics* 31, 5 (Sept. 2012), 119:1–12. https://doi.org/10.1145/2231816.2231817

[10] Max Goldman, Greg Little, and Robert C. Miller. 2011. Real-Time Collaborative Coding in a Web IDE. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, 155–164. https://doi.org/10.1145/2047196.2047215

[11] Sandy J. J. Gould, Anna L. Cox, and Duncan P. Brumby. 2016. Diminished Control in Crowdsourcing: An Investigation of Crowdworker Multitasking Behavior. *ACM Transactions on Computer-Human Interaction* 23, 3 (June 2016), 19:1–29. https://doi.org/10.1145/2928269

[12] David Alan Grier. 2013. *Crowdsourcing For Dummies*. John Wiley & Sons.

[13] Shuchi Grover, Marie Bienkowski, Amir Tamrakar, Behjat Siddiquie, David Salter, and Ajay Divakaran. 2016. Multimodal Analytics to Study Collaborative Problem Solving in Pair Programming. In *Proceedings of the Sixth International Conference on Learning Analytics & Knowledge - LAK '16*. ACM Press, Edinburgh, United Kingdom, 516–517. https://doi.org/10.1145/2883851.2883877

[14] Christoph Hannebauer, Marc Hesenius, and Volker Gruhn. 2018. Does Syntax Highlighting Help Programming Novices? *Empirical Software Engineering* 23, 5 (Oct. 2018), 2795–2828. https://doi.org/10.1007/s10664-017-9579-0

[15] Jeffrey Heer and Michael Bostock. 2010. Crowdsourcing Graphical Perception: Using Mechanical Turk to Assess Visualization Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2010)*. ACM, 203–212. https://doi.org/10.1145/1753326.1753357

[16] James D. Herbsleb and Audris Mockus. 2003. Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '03)*. ACM, 138–137. https://doi.org/10.1145/940071.940091

[17] Paul Hitlin. 2016. Research in the Crowdsourcing Age, a Case Study. Pew Research Center. http://www.pewinternet.org/2016/07/11/research-in-the-crowdsourcing-age-a-case-study/

[18] Panagiotis G. Ipeirotis. 2010. Demographics of Mechanical Turk. (April 2010). http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1585030

[19] Jason T. Jacques and Per Ola Kristensson. 2015. Understanding the Effects of Code Presentation. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2015)*. ACM, 27–30. https://doi.org/10.1145/2846680.2846685

[20] Phillip Johnston and Rozi Harris. 2019. The Boeing 737 MAX Saga: Lessons for Software Organizations. *Software Quality Professional* 21, 3 (2019), 9.

[21] Evan Jones, Adam Marcus, and Eugene Wu. 2010. 6.092 Introduction to Programming in Java, January IAP 2010. MIT OpenCourseWare. http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/

[22] Aniket Kittur, Ed H. Chi, and Bongwon Suh. 2008. Crowdsourcing User Studies with Mechanical Turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2008)*. ACM, 453–456. https://doi.org/10.1145/1357054.1357127

[23] Frank Kleemann, G. Günter Voß, and Kerstin Rieder. 2008. Un(Der)Paid Innovators: The Commercial Utilization of Consumer Work Through Crowdsourcing. *Science, Technology & Innovation Studies* 4, 1 (2008), 5–26. http://www.sti-studies.de/ojs/index.php/sti/article/view/81

[24] A. J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants. *Empirical Software Engineering* 20, 1 (Feb. 2015), 110–141. https://doi.org/10.1007/s10664-013-9279-3

[25] Laura Lascau, Sandy J. J. Gould, Anna L. Cox, Elizaveta Karmannaya, and Duncan P. Brumby. 2019. Monotasking or Multitasking: Designing for Crowdworkers' Preferences. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, Glasgow, Scotland Uk, 1–14. https://doi.org/10.1145/3290605.3300649

[26] Thomas D. LaToza, W. Ben Towne, Christian M. Adriano, and André van der Hoek. 2014. Microtask Programming: Building Software with a Crowd. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, 43–54. https://doi.org/10.1145/2642918.2647349

[27] N.G. Leveson and C.S. Turner. 1993. An Investigation of the Therac-25 Accidents. *Computer* 26, 7 (July 1993), 18–41. https://doi.org/10.1109/MC.1993.274940

[28] Yuwei Lin. 2005. Gender Dimensions of FLOSS Development. *Mute Magazine* 2, 1 (2005), 38–42. https://www.metamute.org/editorial/articles/gender-dimensions-floss-development

[29] Samiha Marwan, Joseph Jay Williams, and Thomas Price. 2019. An Evaluation of the Impact of Automated Programming Hints on Performance and Learning. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. Association for Computing Machinery, Toronto ON, Canada, 61–70. https://doi.org/10.1145/3291279.3339420

[30] Scott Novotney and Chris Callison-Burch. 2010. Cheap, Fast and Good Enough: Automatic Speech Recognition with Non-Expert Transcription. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (HLT '10)*. Association for Computational Linguistics, 207–215. http://dl.acm.org/citation.cfm?id=1857999.1858023

[31] Stephen O'Grady. 2020. The RedMonk Programming Language Rankings: June 2020. https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/

[32] David B. Palumbo. 1990. Programming Language/Problem-Solving Research: A Review of Relevant Issues. *Review of Educational Research* 60, 1 (1990), 65–89. http://rer.sagepub.com/content/60/1/65.short

[33] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. 2000. Empirical Studies of Software Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*. ACM, 345–355. https://doi.org/10.1145/336512.336586

[34] Gaston Pugliese, Christian Riess, Freya Gassmann, and Zinaida Benenson. 2020. Long-Term Observation on Browser Fingerprinting: Users' Trackability and Perspective. *Proceedings on Privacy Enhancing Technologies* 2020, 2 (April 2020), 558–577. https://doi.org/10.2478/popets-2020-0041

[35] Patrick Rein, Marcel Taeumel, and Robert Hirschfeld. 2020. Towards Empirical Evidence on the Comprehensibility of Natural Language Versus Programming Language. In *Design Thinking Research : Investigating Design Team Performance*, Christoph Meinel and Larry Leifer (Eds.). Springer International Publishing, Cham, 111–131. https://doi.org/10.1007/978-3-030-28960-7_7

[36] Advait Sarkar. 2015. The Impact of Syntax Colouring on Program Comprehension. In *PPIG 2015 - 26th Annual Workshop*. 10. http://www.ppig.org/library/paper/impact-syntax-colouring-program-comprehension

[37] Zohreh Sharafi, Zéphyrin Soh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. Women and Men—Different but Equal: On the Impact of Identifier Style on Source Code Reading. In *20th International Conference on Program Comprehension (ICPC 2012)*. IEEE, 27–36. https://doi.org/10.1109/ICPC.2012.6240505

[38] Rion Snow, Brendan O'Connor, Daniel Jurafsky, and Andrew Y. Ng. 2008. Cheap and Fast—But Is It Good?: Evaluating Non-Expert Annotations for Natural Language Tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP '08)*. Association for Computational Linguistics, 254–263. http://dl.acm.org/citation.cfm?id=1613715.1613751

[39] Kathryn T. Stolee and Sebastian Elbaum. 2010. Exploring the Use of Crowdsourcing to Support Empirical Studies in Software Engineering. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10)*. ACM, 35:1–4. https://doi.org/10.1145/1852786.1852832

[40] TIOBE. 2021. February 2021 Index | TIOBE - The Software Quality Company. https://www.tiobe.com/tiobe-index/

[41] Preston Tunnell Wilson, Justin Pombrio, and Shriram Krishnamurthi. 2017. Can We Crowdsource Language Design?. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. Association for Computing Machinery, Vancouver, BC, Canada, 1–17. https://doi.org/10.1145/3133850.3133863

[42] W3C. 2018. Web Content Accessibility Guidelines (WCAG) 2.1. https://www.w3.org/TR/WCAG21/#reflow

[43] John Walkenbach. 2011. Basic Facts about Formulas. In *Excel® 2010 Formulas*. John Wiley & Sons, Ltd, 39–63. https://doi.org/10.1002/9781118257630.ch2

[44] Paul Whitla. 2009. Crowdsourcing and Its Application in Marketing Activities. *Contemporary Management Research* 5, 1 (Feb. 2009), 15–28. https://doi.org/10.7903/cmr.1145