Automatically Testing a Conversational Crowd Computing Platform

Orestis Kanaris[1] , **Sihang Qiu**[1] , **Ujwal Gadiraju**[1] , **Jie Yang**[1]
[1]TU Delft
{o.kanaris}@student.tudelft.nl, {s.qiu-1, u.k.gadiraju, j.yang-3}@tudelft.nl

Abstract

The rise in the use of crowd computing platforms led to the birth of Dandelion, a conversational crowd computing platform developed at TU Delft with the main goals being to connect students with researchers and to allow students to report on their well-being by using a friendly interface. Dandelion was tested manually up to the time of drafting this paper; thus, the primary motivation behind this paper is to ensure the robustness and measure the responsiveness of Dandelion.

Robustness was exercised by utilizing a simulated user behaving unexpectedly. The testing framework then classifies the behaviour of Dandelion according to the C.R.A.S.H. scale. The testing framework is validated by altering Dandelion's behaviour and ensuring that the test results reflect the change. Furthermore, a lower bound to the run time of a task will be estimated using a Multi-Agent System (M.A.S.) simulation on Dandelion.

Upon verifying the correctness of the robustness test, a faulty assumption was uncovered on which the user's input validation was based. Furthermore, the M.A.S. simulation run estimated a lower bound of ≈ 5.788 seconds, while revealing a lack of user's input validation before posting them in the database.

Keywords— crowd computing, testing, testing conversational agent, conversational crowd computing agent

1 Introduction

Crowd computing, according to Murray et al. [17], is defined as the "opportunistic networks that can be used to spread computation and collect results [...] which combine mobile devices and social interactions to achieve large-scale distributed computation".

The growing popularity of crowd computing platforms [10], led to the birth —initially for research purposes— of the Dandelion[1] platform, a *conversational* crowd computing platform developed by a sub-team of TU Delft's WIS team and two BSc students. Dandelion is used for performing questionnaires on students' well-being, timetable queries for TU Delft students, diary logging and more (see TU Delft's Wellbeing page[2] for more information). By the time this report was drafted, Dandelion was tested chiefly by manual testing.

Manual testing, although overall effective, usually fails to identify some types of errors, since it is not feasible to exercise all of the possible interactions; hence the developers test generally the components that they expect to be the most error prone. While developing the Dandelion platform we were faced with the need to have an automated test tool.

Testing is one of the most critical aspects of the software development cycle [22; 1]. For each testing phase, there exists at least one framework specifically developed for it. Many of these frameworks are either open-source or available for free.

There is an apparent gap in the market for frameworks that automatically test conversational crowd computing platforms —which was experienced first hand— but it can be also observed through the lack of available software and research on this subject. This led the Dandelion team to research and develop such a framework in order to ensure the platform's reliability.

While software testing covers many different aspects of the software, this paper covers testing Dandelion's *Robustness* and *Responsiveness*. Robustness, as defined in IEEE standard $24765 : 2010$, is the degree to which a system operates correctly under exceptional inputs or stressful environmental conditions [16; 1]. This paper will chiefly focus on the exceptional inputs rather than the stress test aspect of robustness.

The reasoning for selecting the above two aspects was the following: Robustness was chosen because of the necessity for such a platform to be able to handle errors and exceptional inputs[16]. Such a scenario (users inputting wrong data) is expected when dealing with a conversational agent due to the freedom a user has. The problem is that a program that lacks robustness might very quickly lose the trust of its users [23].

A responsive website responds to user's behaviour and environment based on screen size, platform and orientation. It is crucial for web pages because it ensures an optimal experience for all users [20]. Since Dandelion is running on Telegram[3], all aspects concerning the interface design and the responsiveness on the user's side are already handled by Telegram. Therefore, to ensure optimal user experience, Dandelion needs testing regarding its response latency.

Concluding, in the case of conversational crowd computing platforms, a robust platform is resilient to any input given by the user and a responsive platform does not exceed the accepted reply limits of up to 30 seconds [7].

The aim of this paper is to answer the following questions:

[1]https://delftdandelion.com/

[2]https://www.tudelft.nl/en/covid/wellbeing#c65880

[3]https://telegram.org/

- **RQ1:** How to automatically test the robustness of a conversational crowd computing platform.
- **RQ2:** How to reliably measure the responsiveness of a chatbot for crowd computing.

This novel testing framework is a Multi-Agent System (M.A.S.) which creates virtual worker(s) whose aim is to exercise the system as a whole [2; 6] while giving helpful insight on the system's bottlenecks. These virtual workers communicate directly with the system's backend; thus being able to also stress test the API.

The framework aims to test the system's robustness on correct and malicious input [16]. It is of paramount importance for a system to be robust to ensure its usability and that there will be no system failures due to malicious or wrong inputs.

The simulation part of the research loosely builds around Sihang Qiu et al. VirtualCrowd work [21], with main dissimilarities being that VirtualCrowd considers the qualification of the workers before assigning a task to them while also allowing for Python plugins to alter the simulation's behaviour.

Although this was the initial approach, early into the development phase, it was discovered that such an approach isn't feasible due to how tightly coupled the implementation of Dandelion is to Telegram. Simulating Dandelion's task assignment is not feasible due to the limited time frame that this project has. It was therefore decided to assign the task to all simulated workers.

Paper Structure

This paper is structured as follows: Section 2 describes the approach taken for tackling the testing challenge mentioned above and the evaluation process of the framework. Section 3 describes the experimental setup and the results gathered from the experiments. Section 4 describes the ethical implications of this project and what steps were taken to ensure that the research is reproducible. Section 5 explains the significance of the results (presented in Section 3) in the context of Dandelion. Finally, Sections 6 and 7 present the conclusion and future improvements that may be implemented.

2 Methodology

The aim of this section is to provide the reader with an explanation of the implementation of the software components (i.e., robustness tester and M.A.S. simulation) and also to explain how the testing framework will be evaluated to ensure its correctness.

2.1 Robustness

The framework will test Dandelion using a technique similar to fuzz testing. For starters, the test will be performed using a simulated telegram user to test the bot's conversational aspect. The simulated user will be governed using the python-telethon[4] library.

The simulated user will initialize the conversation by typing `/start`. As soon as the conversation is started, a `POST` request will be sent to Dandelion, containing a predefined task of 7 questions covering all of the answer types Dandelion allows.

By default, each question in a Dandelion task ends with a text explaining to the user what is the expected answer type (i.e., text, image, audio etc.). This ending text will be extracted, analyzed and mapped to its respective class, and then the fuzzy test will generate a random answer of an arbitrary data type and wait for the bot's response.

The robustness test UML can be seen in Figure 1.

The result of each test case will be categorized according to the first letter of the `C.R.A.S.H.` severity scale [14; 13]. The acronym C.R.A.S.H. stands for **C**atastrophic **R**estart **A**bort **S**ilent **H**indering [12]. Note that returning the correct error message (i.e: "Please answer with text only" or "An image is not a valid answer") is not considered a failure [16].

The pipeline behind the categorization is as follows: Dandelion entering a broken loop or raising an exception will be mapped to a Restart. An Abort error is when the bot stops replying. A Silent error is when a wrong input type is entered, but the bot continues as if nothing happened and a Hindering error is when an incorrect error code is returned. For example, a text-only answer leading to a "Please enter an image" prompt. Everything else is mapped to a Correct answer. As one can notice, there is no mapping for a Catastrophic error; this is because the nature of Dandelion makes it very unlikely for such an error to occur due to the libraries used for its development. Hence it is deliberately left out.

The results will then be presented in Section 3 and explained in Section 5.

2.2 Responsiveness

The responsiveness test aims at answering RQ2, which will provide helpful insight into Dandelion's lower bound duration required for running a full task. Such a metric allows the developers to deploy response delays with greater accuracy. Such a feature is instrumental; according to Karma Choedak, "chatbot users were more satisfied with their interactions when chatbots deploy response delays as opposed to near-instant responses", but "longer response latency in a spontaneous online communication is considered a strong sign of deception, and truth-tellers tend to have shorter response time lags" [3]. To ensure such a controlled delay timeframe, there needs to be data on the backend's latency to adjust the reply delays [4], while also having a good understanding of the system's pipeline. This is where a M.A.S. simulation is needed since it is not feasible to request thousand of users to use it.

Multi-Agent System Simulation

A Multi-Agent System (M.A.S.) simulation will be used to simulate the conversational flow of Dandelion while abstracting the frontend (i.e., being decoupled from Telegram's API). Such a simulation is beneficial since it will provide helpful insight into the communication between simulated users and the conversational manager on a massive scale; something that is not possible to do with Dandelion's actual conversational manager since the latter is coupled to the Telegram API, which does not allow for simulated / test users due to the high cost incurred. To create a user, a unique telephone number is required, which approximately costs €10 per SIM, leading to a high cost when the number of users is in the range of thousands.

The M.A.S. simulation UML is depicted in figure 2. The UML shows that workers are generated according to a Poisson distribution queue [11; 21]. This implementation ensures that workers enter the queue (and start the task) in random intervals, thus modelling a "realistic" scenario. The total number of workers generated is according to the total run time and the Poisson parameter that the testers pre-set. Now assuming N simulated workers (N is in the range of thousands) were generated, all N of them only communicate with the *Simulated Conversation Manager*, which takes care of serving each simulated user, their next question.The simulated conversation manager communicates with the actual Dandelion backend in order to fetch questions and save answers.

The Simulated Conversation Manager also sends the errors and a log file of the conversations to the Test Manager, which turns them over to the Tester to analyze them. This is for now done manually.

A feature of Dandelion that it is not supported in the simulation is the *Watchdog*. In Dandelion, when a task is issued, it has a set of rules which will then be used to match users that are "fit" to get that task (e.g.: study = "CSE"). Dandelion also allows for a task to be

[4] https://docs.telethon.dev/en/latest/

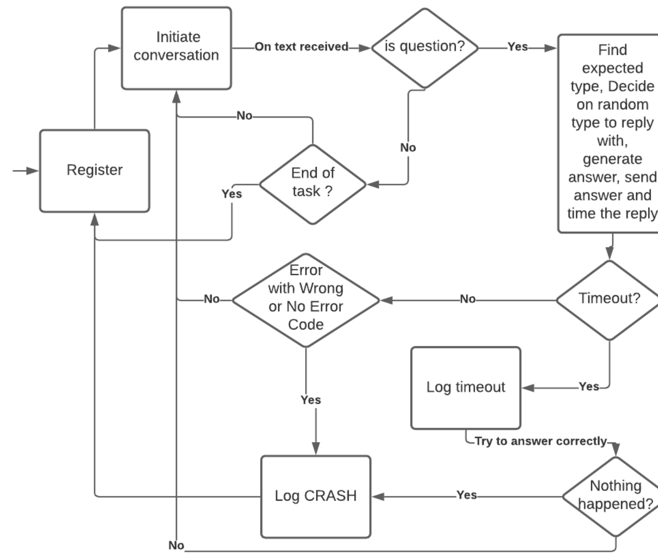


Figure 1: UML of the robustness pipeline

scheduled for a future date a time. Both of the features mentioned are not supported in the simulation, since it would require some sort of re-implementation of Telegram, while the insight gained would not be worth the trouble.

The M.A.S. simulation also serves as an infrastructure to be used in order to estimate the increase of the duration of each task as the number of users served increases. This result, of course, will be a very rough estimate since the model does not consider multiple factors, i.e., human delays, Telegram API delays, different internet latency per user, etc. Nevertheless, it is still a valuable estimation of the maximum number of users that Dandelion can serve simultaneously.

The statistics generated by the M.A.S. simulation are shown in Table 2.

2.3 Framework Evaluation

A testing framework which is not tested is likely to produce several False Positives / Negatives in its test results, which need to be manually identified, hence defeating the framework's purpose in the first place. Therefore, it is paramount to ensure that the testing framework produces as few false flags as possible.

The philosophy behind the reliability validation pipeline is as follows: Introduce bugs in known and tested parts of the bot and ensure that the test results reflect this change, if compared with previous results. For example, remove the `image handler` from the bot when the expected answer is an image, and ensure that the bot will log it as belonging to C.R.A.S.H. While this approach may be naive, it will ensure that most of the framework's implementation is correct.

3 Experimental Setup and Results

It is paramount for an independent researcher to be able to reproduce the experiment that led to one's research conclusions as explained in Section 4. The tests were performed on an HP Workstation studio G5 with 16GB RAM and an Intel Core i7-8750H CPU @ 2.20GHz (6 cores, 12 threads) running Ubuntu 20.04.2. The specific versions of all libraries used can be found in the requirements.txt. The specs are essential for replicating the M.A.S. simulation results since the response time that the tests will indicate may vary depending on the power of the hardware the test is run. However, knowledge of the hardware specs is not essential for the robustness test, since the framework already took into account potential pitfalls (i.e., connection delays, a slow server, etc.) by adding calls to the `sleep` function to give enough time to the server to respond.

3.1 Evaluation of the Robustness Testing Framework

The idea behind the evaluation of the testing framework is, as mentioned in Section 2.3, to introduce various errors to Dandelion and verify that the testing framework will "notice" them.

The results from the evaluation runs are shown in Figure 3. Each graph depicts a different type of test, as follows: On the top left corner (1), the framework ran without fuzzing, basically identifying what datatype each question expected as an answer, thus giving it. This test proves that the framework can imitate a human; hence there is no inherent disability. On the top right corner (2), stands the result of running the testing framework on the Dandelion platform as is. The framework answered 67 questions in total. The results allow the reader to compare them with the results from the tests on a broken Dandelion (which will be discussed in the next sentence) and realize the capabilities of the framework in identifying errors in Dandelion.

The bottom left corner (3) depicts the results of running the fuzzy test on a version of Dandelion which always asks for the same data type regardless of what the question expects. This test was run to validate the robustness of Dandelion on all data types. One can clearly see the rise of Restart errors compared to the regular run. Finally, in the bottom right corner (4), the testing framework tests a version of Dandelion that simulates a broken "fuzzing" component (i.e., after a random amount of time, Dandelion stops working). The spike in the Restart errors is expected in this scenario since every run eventually will come to a halt.

It is evident from the differences between the Figures 3.3 and 3.4 —the increase in the Restart errors and the decrease in the Correct outputs, as the Dandelion platform "worsens"— that the testing framework is sensitive to bugs on the Dandelion platform. The test results currently do not provide a full-on error coverage and analy-

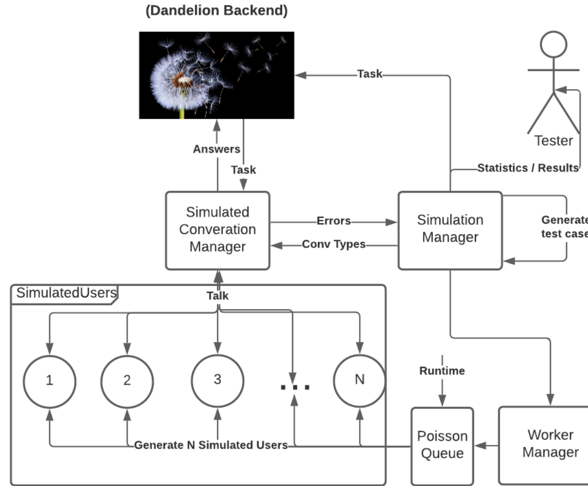


Figure 2: UML of the MAS simulation

Result Type	Frequency
Correct	17
Restart	8
Silent	42
Hindering	0
Total	67

Table 1: Results of running the testing framework on Dandelion.

sis, but they provide the tester with the expected and actual input and the error type. The tester then needs to investigate further the failure and interpret the data.

3.2 Robustness Test

The robustness test, as explained in Section 2.1, is meant to exercise Dandelion's response on inputs that may not match the expected type. The results will then be classified in an adapted C.R.A.S.H. scale [16]. Due to the nature of Dandelion, there is no Catastrophic nor Abort error; hence the errors can only be of type Restart, Silent or Hindering. Correct behaviour will be classified as Correct. Note that a Restart error will also cause the test to terminate prematurely and restart.

The test task was run nine times, and each task contained seven different questions, all with a different expected type. The expected types were: MCQ, Text, Image, Open, Audio, Video, Number (Location was not exercised due to limitations of telethon). If no restart errors were to occur, the framework would answer $9 \cdot 7 = 63$ questions (excluding control flow operations).

Since the framework aimed at trying all types of inputs, there was a 60% chance to input a correct type; otherwise, a random type answer would be sent to the bot. This randomness led to few restarts; hence the total number of questions the framework answered is 67, with 50 out of 67 answers leading to a C.R.A.S.H. The results can be seen in Table 1.

3.3 MAS Simulation

The MAS Simulation was created and run, as explained in Section 2.2. The current simulation implementation has no delays in answering the questions, and the only interactions with the Dandelion API are: user creation, fetching questions (which is done only once) and posting answers. This implementation provides a lower bound of the mean and standard deviation of the time needed for a whole task execution (for a simulated user), thus giving a lower bound estimate of the API delays. The results are shown in Table 2. The experiment's parameters were: maximum run time of 40000 “seconds“ and the Poisson worker generation parameter is $\lambda = 80$.

Mean (μ)	STD (σ)	#Users
5.788294135453459	0.9262387774394841	501

Table 2: Mean and standard deviation of a MAS Simulation run

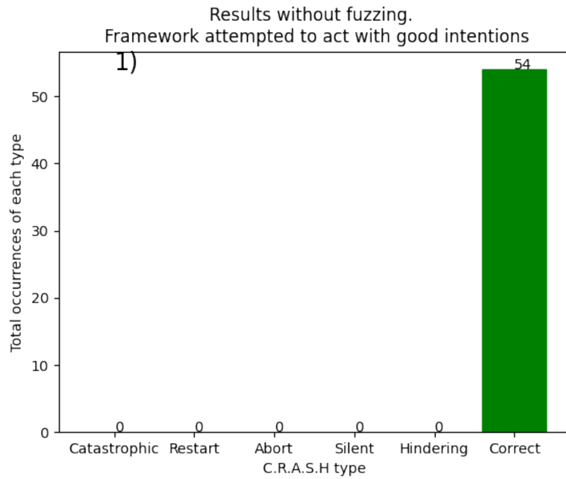
The reason behind not running it for thousands of users, as mentioned in the use case, is the time required. Since no significant additional insight would be gained, it was decided to terminate the simulation prematurely. If one needs to estimate the run time for thousands of users, the distribution in the API delays can be measured and added as a constant to the execution time after each epoch and then run it for thousands of users (without storing the answers or creating users). The result would be a rough approximation of the actual number.

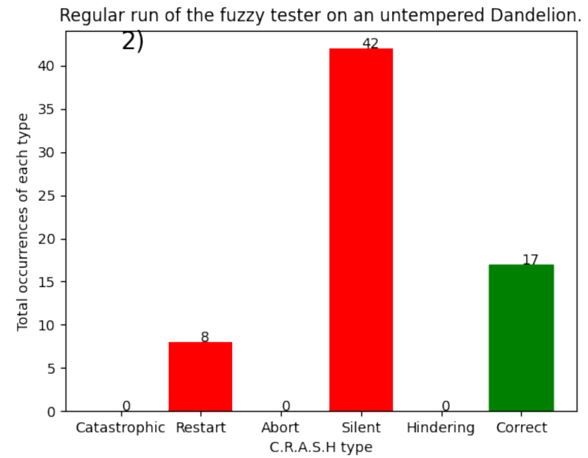
4 Responsible Research

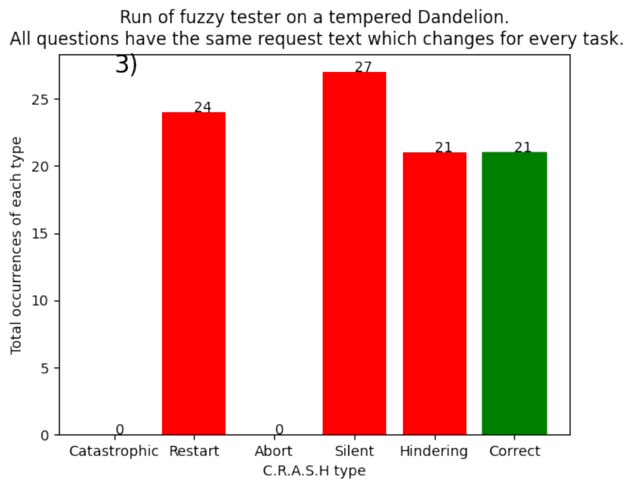
Conducting a responsible research is of uttermost importance to science since it promotes scientific inquiry, encourages public faith in scientific knowledge and development for the public good, and supports a research environment that allows scientists to collaborate toward common aims.

4.1 User Privacy

The Dandelion platform handles personal user data, which primarily make a user identifiable through Telegram. Dandelion submitted a data management plan to the TU Delft ICT, which they accepted. Even though the framework that this paper is proposing will interact with Dandelion, it will not have access to the data that Dandelion has stored. The database load and performance test will be performed







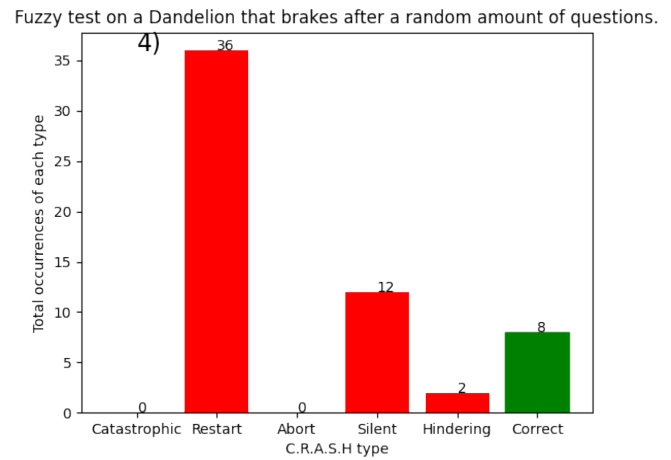


Figure 3: The results from the four experiments that were performed to evaluate the effectiveness of the robustness testing framework. Each graph depicts the results of a different experiment.

on a duplicate (fake) instance of Dandelion's database to not affect the existing data.

The framework will have no interaction with actual humans since the users for the MAS are simulated outside of Telegram.

Moving on, one might wonder, what about the robustness test discussed in Section 2.1 which will be performed by taking control of a Telegram account? The Telegram account will be "brand new" meaning that it will be created with the sole purpose of being used for the test and deactivated after. There is a SIM currently available just for this purpose. Once again, no users are involved in this process, so to the best of my knowledge, the research yields no user privacy concerns.

4.2 Research Reproducibility

According to Goodman et al., a U.S National Science Foundation (NSF) subcommittee on replicability in science said: "reproducibility refers to the ability of a researcher to duplicate the results of a prior study using the same materials as were used by the original investigator [...] reproducibility is a minimum necessary condition for a finding to be believable and informative" [9].

Evaluating Software Testing Techniques (STT) rely on many different concepts, sometimes even outside of the scope of computer science. According to Francisco G. De Oliveira, "to achieve reproducibility in software testing, there should be a compendium (i.e. a package) named Reproducible Software Testing Research Compendium (RSTRC), containing all of the elements described and needed to re-execute the experiment" [19]. Each study parameter belongs to one of the following four categories: operationalization, protocol, population and experimenter. Together they cover the general configuration of an experiment, as discussed by Gómez et al. [18].

- `Operationalization`: Aspects that describe the process of transforming a concept into its materialization. In this case: The contents of Section 2 and the contents of the README.md[5] explain the testing conditions when writing this paper and also how to run the tests (i.e., file structure, runner file etc.) to evaluate Dandelion.
- `Protocol`: Set of materials, equipment, forms and procedures used. In this case: The Python scripts for the tests which can

[5] https://gitlab.ewi.tudelft.nl/cse3000/2020-2021/rp-group-45/rp-group-45-okanaris/-/blob/master/README.md

be found in this repository[6].

- `Population`: The objects and subjects used. In this case: The Dandelion[7] conversational crowd computing agent.
- `Experimenter`: The people involved in the experiment. In this case: The writer of this paper.

Based on the upon items, I am confident that the necessary tools are provided for anyone to replicate the testing experiment and produce very similar results.

Running the test and generating exactly the same results —even though it is possible — would take some time due to the randomness of the framework.

5 Discussion

This section is about the interpretations of the results shown in Section 3.

Improved Testing Coverage of Dandelion

As it was already mentioned in Section 1, the development team mostly tested Dandelion's user interface manually. Early in the development cycle came the need to validate the data type of a user's input — the telegram-bot-library provides a handler for each data type accepted in the Telegram app. The validation of a user's input was based on the bold assumption that by only adding the respective data handler of the data type that the user is expected to input, every reply of a different data type will be rejected by Telegram itself. Dandelion has a big "switch statement" that checks what data type the user is expected to input for $question_i$ and adds the respective handler to the question. Hence for example, if the question expected an *image* as a reply, but the user inputs a video, the conversation manager would accept it and eventually commit it in the MongoDB database labelling it with the expected data type which is different from the actual one.

Note that Dandelion uses mongoengine [8], a python Object Document Mapping (ODM) that, among others, attempts to "enforce a schema" to a MongoDB database. The current implementation matched the datatype that the object was labelled with, with the respective mongoengine type, upon retrieval from the database. But since they were mislabeled, mongoengine will throw —an unhandled— `TypeError`, which lead to a Restart in the C.R.A.S.H. scale.

This discovery is very important since a single answer of a wrong type would deem the answers document of that particular task (all the answers from all users who participated in that task) impossible to automatically retrieve, with the method currently in place. To retrieve them, one would have to manually go through the whole answers document, spot the problematic entry and delete it. It might not sound very hard if the mistake is obvious (i.e., a text file is labelled as a video). Still, it is time-consuming to identify the file type, especially if they are stored in the Telegram server (one needs to download and go through all of the files).

Another discovery was that Dandelion sent some data types (i.e., Location and Number) to the user with the wrong prompt; specifically, it asked the user to input text rather than a Number or a Location. Such an error is not as severe as the one mentioned above since the user can identify the expected data type by themselves. But it may cause errors in the case of: "Please input your age. This question must be answered with Text only". In this example, the user can understand that the question expects them to input a number (since age is a number), but they may input it in a textual form (e.g.: "seventeen") rather than a numerical form (e.g., 17) which is the one that the question expects to process, eventually leading to the same `TypeError` as the one mentioned above.

After these discoveries, the development team will have to go back to the drawing board and decide on how to correctly ensure that the user inputs the correct data type and, most importantly, prompt the user to do so once they inputted an answer of the wrong type — the lack of correct prompts is evident from a large number of Silent errors.

Findings from Running the Simulation

The experiment run of the M.A.S. simulation gave a lower bound of ≈ 5.788 seconds for running the simulation. Having this number is useful since, having this in mind, delays can be added to the answer sending mechanism of Dandelion in such a way as to simulate the time it would take humans to respond to a message [8] while also not exceeding the reply threshold of 30 seconds [7].

Furthermore, during the simulation implementation, it was discovered that the answer saving API is not robust against wrong input. The reason for this is that during the development of Dandelion, it was thought that the only calls to the API would come from the Dandelion frontend, where the format was hard-coded. This implementation makes the API vulnerable to attacks since a wrong format causes the server to shut down due to a JSON Exception, which the developer team needs to review.

6 Conclusions

This paper proposes a method of testing conversational crowd computing platforms in terms of the platform's robustness by executing a fuzzy test and classifying the results on the C.R.A.S.H. scale while also gathering an estimate of the platform API's responsiveness (data that are useful in order to accurately simulate the time that a human takes to reply) using a Multi-Agent-System (M.A.S.) simulation.

The testing framework was used to test TU Delft's Dandelion platform, a conversational crowd computing platform used to connect students with researchers while also allowing students to report their well-being status; Dandelion was manually tested up to now.

The robustness test reported vital vulnerabilities, most notably, a lack of validation of the input type. The M.A.S. simulation, apart from estimating a lower bound of ≈ 5.788 seconds for a simulated task of 4 questions, also identified a significant lack of input sanitizing in the API used for answer saving. The framework was verified by introducing errors in Dandelion and verifying that the results of the framework were significantly different than the ones from when Dandelion was unaltered.

7 Future Work and Recommendations

The framework described in the previous sections is a simulation-based MAS testing tool that aims to the backend of Dandelion while also testing its conversation manager for any C.R.A.S.H. While it is very helpful to know what inputs causes what, there is no particular knowledge on what happens with the rest of the users as soon as a C.R.A.S.H. happens. Hence in a future iteration of this framework —assuming that Telegram finally allows for test users, or bots testing bots — there should be a simulation to ensure that there are no cascading failures, if for some reason dandelion crashes for one of the users. This can also be done by leveraging some workers from an existing crowd computing platform, for example, Amazon Mechanical Turk[9]. The workers would be asked to "break" the platform and

[6]https://gitlab.ewi.tudelft.nl/cse3000/2020-2021/rp-group-45/rp-group-45-okanaris

[7]https://delftdandelion.com/

[8]http://mongoengine.org/

[9]https://www.mturk.com/

report their findings [15]. The results and improvements that will come from these tests will improve the usability and the user experience a lot, since it is very frustrating as a user for a program to just suddenly break down without knowing why.

Furthermore, as mentioned in RQ2, this simulation framework only aims to generate answers that are just good enough for the simulation purposes. The Dandelion platform, only validates the data type of an answer, not the contents, a random input of a correct data type is currently a valid answer. This means approach will not be usable by intelligent platforms that expect that the input adheres to some pre-set rules. This design choice was decided due to the strict timeframe of the research project and can be addressed by implementing an intelligent input generation mechanism. A proposed technique is to use an auto-regressive language model [5] for example, one of OpenAI's gpt algorithm.

Another improvement is to add some missing functionality to the Telethon library. For example, it is currently not possible to send location or polls, something that limits the testing capabilities since it cannot be tested how the bot handles these particular inputs.

Finally, the bugs discovered need to be fixed and then retest Dandelion to ensure its robustness.

References

[1] Iso/iec/ieee international standard - systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, 2010.

[2] G Caire, M Cossentino, A Negri, A Poggi, and P Turci. *Multi-Agent Systems Implementation and Testing*.

[3] Karma Choedak. *The effect of chatbots response latency on users' trust*, May 2020.

[4] ImTheDeveloper DEV Community. Optimising your telegram bot response times, Oct 2019.

[5] Robert Dale. Gpt-3: What's it good for? *Natural Language Engineering*, 27(1):113–118, 2021.

[6] Scott A. Deloach, Mark F. Wood, and Clint H. Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(03):231–258, 2001.

[7] Facebook. Responsiveness requirements - messenger platform.

[8] Ulrich Gnewuch, Stefan Morana, Marc Adams, and Alexander Maedche. Faster is not always better: Understanding the effect of dynamic response delays in human-chatbot interaction. *Research Papers*, 113, 2018.

[9] Steven N. Goodman, Daniele Fanelli, and John P. A. Ioannidis. What does research reproducibility mean? *Science Translational Medicine*, 8(341):341ps12–341ps12, 2016.

[10] Vimi Grewal-Carr, Greg Howard, Carl Bates, and Harvey Lewis. The three bilion enterprise crowdsourcing and the growing fragmentation of work, 2016.

[11] Leif Gustafsson. Poisson simulation as an extension of continuous system simulation for the modeling of queuing systems. *SIMULATION*, 79(9):528–541, 2003.

[12] P. Koopman and J. Devale. The exception handling effectiveness of posix operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, 2000.

[13] Philip Koopman, Kobey Devale, and John Devale. Interface robustness testing: Experience and lessons learned from the ballista project. *Dependability Benchmarking for Computer Systems*, page 201–226.

[14] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek. *Automated robustness testing of off-the-shelf software components*. 1998.

[15] Di Liu, Randolph G. Bias, Matthew Lease, and Rebecca Kuipers. Crowdsourcing for usability testing. *Proceedings of the American Society for Information Science and Technology*, 49(1):1–10, 2012.

[16] Zoltán Micskei, Henrique Madeira, Alberto Avritzer, István Majzik, Marco Vieira, and Nuno Antunes. *Robustness Testing Techniques and Tools*, page 323–339. 2012.

[17] Derek G. Murray, Eiko Yoneki, Jon Crowcroft, and Steven Hand. *The case for crowd computing*. 2010.

[18] S. Vegas O. S. Gómez, N. Juristo. Understanding replication of experiments in software engineering: A classification. *Information and Software Technology*, 56(8):1033, 2014.

[19] Francisco G. De Oliveira Neto, Richard Torkar, and Patricia D. L. Machado. *An Initiative to Improve Reproducibility and Empirical Evaluation of Software Testing Techniques*. 2015.

[20] Rachel Andrew Peter LePage. Responsive web design basics.

[21] Sihang Qiu, Alessandro Bozzon, and Geert-Jan Houben. Virtualcrowd: A simulation platform for microtask crowdsourcing campaigns.

[22] Azeem Uddin and Abhineet Anand. Importance of software testing in the process of software development. pages 2321–0613, 01 2019.

[23] J. Voas. Untested software threatens infrastructures. *IEEE Software*, 16(2):89–90, 1999.