A user study of command line synthesis from natural language

***Abstract*—Developers may have difficulty implementing desired program behaviors, even when they can describe their goal in English. Existing resources, such as question-and-answer websites, catalog specific tasks that someone wanted to perform in the past. However, these resources have limited value for generalizing to new tasks or to compound tasks that require combining multiple operations.**

NLP-inspired tools translating English sentences to equivalent source code are a promising approach. The accuracy of such tools is modest, and in particular is much lower than for natural-language-to-natural-language tools.

Recent work aims to improve the performance of such tools against reference benchmarks. However, our aim is different. Our objective is to determine whether such improvements are essential.

We investigated whether current tools improve developers' productivity and how developers perceive these tools. We performed a controlled study in which developers complete Bash scripting tasks assisted by an automatic English to Bash translation tool.

***Index Terms*—machine translation, bash, natural language, neural networks**

We submit the methodology as-is as pre-registration for our study.

I. Methodology

We wish to know whether an imperfect AI model can help developers. To answer this question, we performed a controlled user study to test whether developers benefited from using an AI assistant that suggests Bash one-liners from english descriptions to complete file system tasks.

We follow the format suggested by [1] for reporting experiments in software engineering. This study is pre-registered [2] at https://doi.org/10.5281/zenodo.6600939.

A. Goal

The goal of this study is to analyze whether developers benefit from using an AI assistant that translate english descriptions into Bash one-liners to complete Linux file system tasks. The AI assistant is imperfect, that is, it can give incorrect suggestions to the developer, which is true for any AI model. To assess the potential benefits, we measured the time it take for developers to complete Linux file system tasks using Bash one-liners and wether they could complete the task within a time limit.

To this end, we formulated the following research questions:

RQ1 Can imperfect AI models help developers?

RQ2 How do developers perceive imperfect tool suggestions?

We refined RQ1 in two research questions:

RQ1.1 Can imperfect AI models help developers complete tasks faster?

RQ1.2 Can imperfect AI models help developers complete tasks more correctly?

We answered the first research question with a controlled experiment and the second with an exit survey.

If imperfect predictions significantly reduce developers' effort, then a focus on model's performance alone may lead researchers to dismiss useful approaches.

Although our experiment considers one specific tool, the implications would apply much more broadly, even beyond software engineering tools and tasks. This is a valid generalizability claim to make because the existential argument for this tool shows that it is worth investigating for any other tools using ML models. Until you try it out on your end users, there is currently no way to know how helpful your model's suggestions are in practice and what is the model's performance threshold, if any, for the tool to be practically helpful to developers.

B. Participants

We recruited undergraduate and graduate students who are familiar with the Unix command line and the Bash command language, either from taking a class or from programming experience. Participants received a $20 gift-card compensation for their participation (approximately one hour).

C. Experimental Materials

All the experimental materials are publicly available at https://doi.org/10.5281/zenodo.6600939.

1) ***Website****:* The experiment website contained an overview of the experiment, the consent form, and instructions for participating in the experiment.

2) ***Consent****:* Participant consent was obtained electronically at the start of the experiment. Participants could withdraw from the study at any time without disclosing a reason.

3) ***Demographic Survey****:* Before starting the experiment, participants filled out an anonymous survey that contained the following questions:

1) What year did you enter college?
2) How many courses with programming exercises or projects have you taken in your life (including any current enrollments)?
3) How many years of programming experience, in any language, do you have professionally (i.e., for which you got paid, including internships)?
4) How familiar are you with Shell scripting (including Bash scripting)?

Beginner (Using Shell scripting or commands infrequently or having some class experience)
Experienced (Using Shell scripting or commands somewhat regularly or writing a few scripts)
Expert (Using Shell scripting or commands extensively or developing production code)

5) Have you used the Tellina tool before?

*4) **Environment**:* Each participant ran the experiment on a computer or virtual machine running Rocky Linux 8 [3]. Participants ran commands in Rocky Linux's default terminal.

Whenever a user ran a command that does not solve the given task, the Meld [4] visual comparison tool quickly and concisely [5] showed how the files and directories (or, the command's output) differed from the expected solution.

*5) **File system**:* Users ran commands from a directory containing the files and folders from a GitHub repository[1]. The directory structure consisted of 29 folders and 62 files and is 4 levels deep. We changed all constant values in the task descriptions to match the contents of this file system.

*6) **Natural Language to Code Interface**:* We chose the task of converting natural language to bash commands because there was a recent competition [6] in which 7 tools [7], [8], [9], [10], [11], [12], [13] competed. The competition focused on comparing approaches through metrics including accuracy, energy consumption, and latency.

Compared to other code completion tasks, file-system completion task in Bash have multiple advantages: 1) All developers are familiar with the business domain since everyone know what a file system is; 2) Bash one-liners are fast to write and execute, and do not require to install a compiler or interpreter. 3) Participants can focus on writing Bash one-liners since they do not have to worry about language syntax, or API documentation to complete the tasks.

To choose between the 7 tools that competed, we considered usability as it is the most relevant for developers adoption and usage. That is, how easy is it for a developer to submit a query to the tool and receive the results. We preferred a web-based tool, rather than a command-line tool. A web tool that works in a similar way to existing tools developers use (e.g., Google, Stack Overflow), does not clutter or interrupt the shell session, and does not require learning new special-purpose shell commands. We chose Tellina [13]. Section **??** compares Tellina to the other tools. Our experiment is not an evaluation of this tool per se, and could be run with any reasonable assistant that suggests Bash one-liners.

***Tellina**:* Tellina is an open source tool supporting 102 unique utilities, 206 unique flags and 15 reserved tokens. Tellina's model is trained on 10,000 pairs of English descriptions and Bash one-liners. The model has a BLEU [14] score of 50.9/100 and a top-3 full command accuracy of 45%. The full command accuracy expects an exact match between the predicted solution and the expected solution: it does not account for partial solutions. When accounting for partial solutions where incorrect command arguments do not count as errors (i.e., the name of the input file is incorrect), the top-3 partial accuracy performance is 61%, In other words, Tellina is wrong about half the time.

Tellina's web application [15] has an interface similar to the Google search engine: the user types a natural language sentence describing a task, then the website displays the model's top 20 bash command translations of the sentence.

The Tellina web application shows recent queries from all users. The user study used a modified version that lacks this feature to avoid influencing participants with queries of other participants. The web application also shows example queries; we ensured that they differ from all the tasks in the experiments.

*7) **Exit survey**:* After the experiment, participants filled out a survey, which qualitatively measured the perceived usefulness of Tellina. Questions marked with **[likert]** are coded on a 1 to 7 scale [16].

1) Did you try Tellina at least once to help you solve a task? **[binary]**
2) I want to use Tellina in the future. **[likert]**
3) Partially correct suggestions made by Tellina helped me find a solution. **[likert]**
4) Suggestions made by Tellina slowed me down. **[likert]**
5) Incorrect suggestions made by Tellina were easy to correct. **[likert]**
6) What features of Tellina were the most helpful to you? **[open]**
7) What mistakes made by Tellina affected you most? **[open]**
8) Please list the features that you think should be added to Tellina. **[open]**
9) Please give us any additional comments you have about Tellina. **[open]**

D. Tasks

We sourced the tasks from real posts from five websites (table I) that offer programming help.

TABLE I: Popular websites for help on Bash problems.

Stack Overflow	https://stackoverflow.com/
Super User	https://superuser.com/
Unix & Linux	https://unix.stackexchange.com/
Command Line Fu	https://www.commandlinefu.com/
Bash One-Liners	https://www.bashoneliners.com/

1) Collection: We collected a total of 125 commands. From each of the five websites, we collected the 25 most popular posts that satisfy the following criteria:

- the solution is a Bash one-liner.
- the solution uses default commands included in Linux distributions.
- the solution uses commands supported by Tellina [13].
- the solution is deterministic and can be reproduced as a file system task.

For Stack Exchange websites (i.e., Stack Overflow, Super User, Unix & Linux), we only considered posts tagged as *bash* questions that had an accepted solution.

[1]https://github.com/icecreammatt/class-website-template

2) Sampling: We randomly sampled 17 posts out of the 125 to be used as tasks for the participants. 12 posts were for the experiment and the remaining 5 posts were for the experiment's training. Ten of the 17 solutions consisted of one Bash command, and the remaining 7 had two Bash commands (e.g., connected by a pipe). The solutions of the 17 posts used 13 distinct commands (frequency indicated in parentheses): `find` (8), `wc` (2), `grep` (1), `sort` (1), `cat` (1), `mv` (1), `cp` (1), `mkdir` (1), `comm` (1), `diff` (1), `du` (1), `rmdir` (1), `basename` (1). We observed the same long-tailed distribution of commands as [13] with the command `find` being most common and followed by a long tail. In total, the solutions used 39 flags and 44 arguments, averaging 2.3 flags and 2.6 arguments per Bash one-liner.

3) Rewording: When a post referred to a file or directory name or extension, we adapted the question and solution to the local files and folders in the experiment file system (section I-C5).

We paraphrased the questions to prevent posts from being trivially found on the web by the user copy-and-pasting the task prompt into a search engine. After our paraphrasing, the webpage containing the original post did not appear in the top 10 Google search results (but other webpages that solved the problem may be in the top 10 Google search results).

We controlled for the clarity of the reworded questions by running the questions in the pilots described in section I-I and reworded ambiguous or unclear questions. Rewording questions introduced implicit bias, which is also present in the original questions. We took care to be complete, precise, consistent and to not assume previous knowledge on the Linux filesystem or directory recursivity expectations. We addressed bias issues reported during the pilots.

4) Operationalization: A task consisted of a prompt and an expected outcome. The prompt is an English description of a file system operation (e.g., "Find all files whose name starts with foo"). The expected outcome is either a change in the file system, such as deleted/added/modified files, or the command output displayed on the terminal (e.g., `cat foo.txt`).

E. Procedure

Each participant completed the study in a location of their choosing, unobserved to mitigate the Hawthorne effect [17], [18].

Each participant first completed a general training, solving 3 blank tasks (i.e., tasks that do not count towards the experiment). The general training familiarized the participant with the environment, how tasks are presented, the use of the Meld diff tool, the help commands available in the shell, and how to give up on a task. Before the participant used Tellina, the participant went through Tellina training. This was like general training, with 2 blank tasks to solve with the use of Tellina. This familiarized users with the Tellina web interface. For some users (those who were allowed to use Tellina on their first set of tasks), the two trainings occurred back-to-back at the beginning of the experiment. For other users (those who were allowed to use Tellina on their second set of tasks), the general training occurred at the beginning of the experiment, and the Tellina training occurred midway through the experiment.

TABLE II: Factorial counterbalanced experiment design. For example, participants in group A started with tasks A to F under the control treatment. They finished with tasks G to L under the Tellina treatment.

		Treatment Order	
		Control, Tellina	Tellina, Control
Taskset Order	$T_{\text{A..F}}$, $T_{\text{G..L}}$	Group A	Group B
	$T_{\text{G..L}}$,$T_{\text{A..F}}$	Group C	Group D

A participant could attempt a task multiple times. Every attempt resets the file system to its original state in order to start over from a fresh slate and enforces the use of a Bash one-liner to solve the task. Each task had a 6-minute timeout after which the experiment automatically skipped to the next task. In addition, each set of task had a time limit of 25 minutes.

F. Hypotheses and Variables

The independent variable *Tellina Access* (*TA*) is whether participants have access to Tellina or not for a task. Its values (the experimental treatments) are:

- *No (Control):* The participant may use any local resource (such as man pages and experimentation on the command line) and any Internet resource (such as tutorials, question-and-answer websites, and web search). This emulates how a programmer would normally solve a file system task.
- *Yes (Tellina):* The participant may use Tellina and any of the resources available in the control treatment.

We answer RQ1 using the dependent variables *Success Rate (SR)* and *Time Spent on Tasks (TST)*. *SR* is the fraction of tasks solved by the participant for each condition; it is between 0 and 1, inclusive. *TST* is the total time in seconds that the participant spent on the tasks. We measure the time spent from the moment the task's prompt appears on the participant's terminal until they submit the right answer, run out of time, or abandon the task (which counts as the maximum time).

The null hypotheses are:

$H1_0$: Access to Tellina has no significant effect on *Success Rate* (SR).

$H2_0$: Access to Tellina has no significant effect in *Time Spent on Task* (TST).

For RQ2, we considered the responses from the survey. We used the four Likert-scale questions to quantify Tellina's perceived usefulness and the four open-ended questions for qualitative comments and feedback.

Additionally, we collected the dependent variables *Number of Attempts* (NA) and *Recovery Event* (RE) for exploratory purposes. *NA* counts the number of attempts to complete a task. *RE* keeps track if the participant encountered a situation where they left the experiment and resumed it, such as a personal emergency or the experimental infrastructure crashing.

G. Experiment Design

The experiment used a within-subjects design, where each participant experienced every treatment on the independent variable: each participant does some tasks in one condition and other tasks in the other condition. We adopted a factorial counterbalanced design (table II) to avoid learning effects or fatigue bias due to the order of tasksets or treatments. The participants were distributed uniformly into the four groups.

Section I-D2 explained how we chose 17 tasks. We used the average time spent during the pilot (section I-I) as a proxy for task difficulty.

We used 5 of the easiest commands for participant training. For the general training, we used 2 of the 3 easiest tasks whose Bash one-liner solution contains one command, and 1 of the 2 easiest multi-command tasks. For the Tellina training, we used 1 of the 3 easiest single-command tasks, and 1 of the 2 easiest multi-command tasks. Inclusion of multi-command tasks ensured the participant is familiar with Bash one-liners and made the training representative of the rest of the experiment.

We randomly split the remaining 12 tasks into two tasksets of size 6, and sorted each taskset by difficulty from easiest to hardest. We named the tasks in one taskset A to F, and named the tasks in the other taskset G to L.

H. Analysis procedure

1) Effect: We used a four-way analysis of variance (ANOVA) to verify the effect of independent variables (order, subject, taskset, treatment) on dependent variables SR and TST.

SR_t and SR_c, TST_t and TST_c are calculated per participant. (t subscripts are for the Tellina treatment, and c subscripts are for the control treatment.) We used a mortality analysis for tasks that are abandoned, timed-out, or not completed (i.e., the participant ran out of time in the taskset): the task was counted as taking the maximum time of 6 minutes.

We removed participants who did not complete the experiment and manually reviewed cases where the total time spent is 1.5 times the interquartile range above or below the upper and lower quartile respectively: we removed participants who didn't make serious attempts (i.e., they didn't enter any commands, or skipped the majority of the tasks).

2) RQ1: Assuming a normal distribution (i.e., Shapiro–Wilk) for SR_t and SR_c, TST_t and TST_c, we answered RQ1 by testing the null hypotheses $H1_0$ and $H2_0$.

Specifically, we tested for significant difference between SR_t and SR_c. We rejected $H1_0$ and accepted the alternative hypothesis $H1_a$ (Access to Tellina has a significant effect on SR) if there were a significant difference with $p = 0.05$ as threshold.

Similarly, we tested for significant difference between TST_t and TST_c. We rejected $H2_0$ and accepted the alternative hypothesis $H2_a$ (Access to Tellina has a significant effect in TST) if there were a significant difference with $p = 0.05$ as threshold.

3) RQ2: We performed a qualitative analysis to answer RQ2. We used the data from the *Exit Survey*.

We used questions 3), 4), 5) to analysis the perception of the participants. We used question 2) as a sanity check and open-ended question 7) as supporting evidence.

Questions 3), 4), 5) were Likert-scale questions rated from 1 to 7. A value of 1 represented a *Strong disagreement* and a value of 7 represented a *Strong agreement* with the statement. A value of 4 was neutral.

We looked at the frequency of negative and positive responses centered from the middle of the Likert questions as evidence to answer RQ2.

For example, for question *Sometimes Tellina makes incorrect suggestions. How often did partially correct suggestions help you*, we answered by the affirmative (i.e., Partially correct suggestions are helpful) if most of the answers are above the middle of the Likert scale (i.e., 5).

If most of the questions were affirmative, then it indicates that Tellina suggestions are useful and the incorrect suggestions are not impeding participants.

I. Pilots

Following recommendations from [19], the experiment design went through several rounds of peer reviews and pilot tests.

1) Pilot 1: The two tasksets contained 8 tasks each and the training consisted of 1 general training task at the start and 1 training task before the treatment with Tellina. The tasks had a limit of 10 minutes each and the tasksets had a limit of 40 minutes each. The experiment environment included a custom web interface hosting an emulated shell environment, the Tellina website, and the exit survey. The experiment was conducted on the web interface.

We recruited 39 students in the computer science major to participate (24 graduate students, 15 undergraduates). None were familiar with Tellina. All of them were familiar with Bash. We accepted only graduate students who self-reported to be Bash users, and we accepted only undergraduates who had completed or were enrolled in our department's Linux tools course. We excluded data from 4 of the participants, because 3 of them forgot to switch treatment conditions between the tasksets and 1 of them did not complete the study. The participants were not compensated.

We observed that participants in the Tellina treatment spent on average 22% less time than without Tellina. This seems to indicate that Tellina helps programmers to write bash commands in less time. Furthermore, the survey indicated that Tellina is perceived positively overall and that partially correct suggestion were still helpful.

This pilot shed light on important issues. First, we observed that around 20% participants were not able to complete the experiment or the data generated was lost due bugs in the infrastructure. To address this issue, we changed the experimental infrastructure to work inside a regular Bash environment directly. Additionally, we designed and implemented a recovery feature allowing participants to restart the experiment where they left off. We control for the effect of the recovery feature

by observing whether a participant has to use this feature and check for statistically significant effect.

Second, we learned that due to the small number of participants and dropped data, the experimental results were not always statistically significant ($p = 0.05$), and that with a larger pool of subjects, the results would be more compelling.

2) Pilot 2: The first pilot was performed with a version of Tellina that handles 17 file system utilities. We wished to perform our main experiment with the latest version of Tellina, which handles 102 utilities, so we performed another pilot experiment.

We recruited 6 graduate students and 8 undergraduate students by convenience sampling. The graduate students participated voluntarily and the undergraduate students were compensated $20. All of the participants were familiar with Bash. The experiment environment included a Bash experiment environment, the Tellina website, and the exit survey, as presented in Sections I-C.

We found that the initial 8 tasks each plus 2 training tasks was too much (18 total). The number of tasks incurred frustration and fatigue, causing participants to skip tasks or cheat (e.g., printing the expected output with `echo`), particularly for participants that were not compensated, indicating potential quality issues with the data collected. To mitigate this effect, we decreased the size of each taskset to 6 tasks and reduced the time limits for tasks to 6 minutes and for tasksets to 25 minutes. Additionally, we added 3 more training tasks (Section I-G) to reduce the learning effect during the experiment proper.

Difficult tasks frustrated participants, especially when encountered early in the experiment. To mitigate this, we sorted the tasks by increasing difficulty (based on time taken in the pilots to date). This does not introduce fatigue bias tasks do not change position except between treatments, which is addressed by the factorial design.

Decreasing the number of tasks makes it harder to show statistically significant results. We calculated that at least 60 participants are needed to obtain a power of 0.80 with a moderate effect size (Cohen's $f = 0.25$) and $p = 0.05$.

Similarly to the previous pilot, we noticed that some participants didn't switch treatments. To mitigate this issue, we added a prompt encouraging the participants to pause and read the new instructions indicating that the treatment had changed.

REFERENCES

[1] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 201–228.

[2] J. P Simmons, L. D Nelson, and U. Simonsohn, "Pre-registration: Why and how," *Journal of Consumer Psychology*, vol. 31, no. 1, pp. 151–162, 2021.

[3] Rocky Enterprise Software Foundation. (2022, mar) Rocky linux. [Online]. Available: https://rockylinux.org/

[4] K. Willadsen. (2011, Jan) Meld. Kai Willadsen. [Online]. Available: https://meldmerge.org/

[5] S. McKee, N. Nelson, A. Sarma, and D. Dig, "Software practitioner perspectives on merge conflicts and resolutions," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 467–478.

[6] M. Agarwal, T. Chakraborti, Q. Fu, D. Gros, X. V. Lin, J. Maene, K. Talamadupula, Z. Teng, and J. White, "NeurIPS 2020 NLC2CMD competition: Translating natural language to Bash commands," in *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*, ser. Proceedings of Machine Learning Research, vol. 133, 06–12 Dec 2021, pp. 302–324. [Online]. Available: https://proceedings.mlr.press/v133/agarwal21b.html

[7] Q. Fu, Z. Teng, J. White, and D. C. Schmidt, "A transformer-based approach for translating natural language to Bash commands," in *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2021, pp. 1245–1248.

[8] J. Maene and G. Csatari. (2020, Dec) Hubris. Nokia Bell Labs. [Online]. Available: https://github.com/nokia/nlc2cmd-submission-hubris

[9] D. Litvinov, G. Morgachev, A. Popov, N. Korolev, and D. Orekhov, "NLC2CMD report from JB team," JetBrains, Tech. Rep., December 2020. [Online]. Available: https://github.com/JetBrains/nlc2cmd/blob/master/report.pdf

[10] H. Shin. (2020, Dec) Aicore. Samsung. [Online]. Available: https://github.com/Samsung/NLC2CMD

[11] D. Gros. (2020, Dec) Ainixclaisimple. University of California, Davis. [Online]. Available: https://github.com/DNGros/clai

[12] S. Kang and J. Yoon. (2020) Hierarchical decoding of Bash commands. University of California, Davis. Hierarchical Decoding of Bash Commands. [Online]. Available: https://github.com/greenmonn/nlc2cmd-competition

[13] X. V. Lin, C. Wang, L. Zettlemoyer, and M. D. Ernst, "Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system," in *Proceedings of the Eleventh International Conference on Language Resources and Evaluation, LREC 2018, Miyazaki, Japan, May 7-12, 2018*. European Language Resources Association (ELRA), 2018. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2018/summaries/1021.html

[14] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. Stroudsburg, PA, USA: Association for Computational Linguistics, 2002, pp. 311–318. [Online]. Available: http://dx.doi.org/10.3115/1073083.1073135

[15] V. Lin, C. Wang, and D. Pang. (2018, Jan) Tellina. Tellina Tool. [Online]. Available: https://tellina.rocks/

[16] R. Likert, "A technique for the measurement of attitudes." *Archives of psychology*, 1932.

[17] F. J. Roethlisberger and W. J. Dickson, *Management and the Worker*. Psychology press, 2003, vol. 5.

[18] R. McCarney, J. Warner, S. Iliffe, R. Van Haselen, M. Griffin, and P. Fisher, "The hawthorne effect: a randomised, controlled trial," *BMC medical research methodology*, vol. 7, no. 1, pp. 1–8, 2007.

[19] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.