TrumanBench: Profiling LLMs' Ability to Help Non-Programmers Modify a Real-World Code Base

Qian Yang*
qianyang@cornell.edu
Cornell University
Ithaca, New York, USA

J.D. Zamfirescu-Pereira*
zamfi@berkeley.edu
UC Berkeley
Berkeley, CA, USA

Jessie Jia
hj359@cornell.edu
Cornell University
Ithaca, New York, USA

Asad Nabi
an448@cornell.edu
Cornell University
Ithaca, New York, USA

ABSTRACT

Imagine a future where anyone can customize open-source software to suit their own needs simply by requesting changes in natural language. This future empowers non-programmers and amplifies the impact of open-source software. This paper examines the potential of Large Language Models (LLMs) and multi-agent systems in realizing this future. Specifically, we assess the capabilities of Claude- and metaGPT-based systems in fulfilling social scientists' needs to modify Truman, a 20,000-line web application that is a social science experimental platform. Our evaluation suggests that LLMs currently cannot reliably execute the social scientists' requests, and contrary to popular belief, adding more LLM agents did not necessarily help. A key reason is that LLMs struggle with cross-file dependencies. We present these findings and discuss lessons learned, including (1) TrumanBench, a framework for assessing LLMs' abilities to execute non-programmers' code modification instructions, and (2) near-future opportunities in designing LLM-powered end-user code modification tools, and (3) research opportunities in tailoring future open-source codebases for LLM inquiries.

CCS CONCEPTS

• **Human-centered computing** → **Empirical studies in HCI**; *HCI theory, concepts and models*; • **Computing methodologies** → *Artificial intelligence.*

1 INTRODUCTION

Envision a future where anyone can customize open-source software to suit their own needs simply by requesting changes in natural language. This future not only empowers everyone—especially those without formal programming training—in harnessing the power of computing, but also amplifies the impact of open-source software. Large Language Models (LLMs) hold exciting promises for realizing this vision, thanks to their emerging abilities to modify existing code based on natural language instructions. LLM-based multi-agent systems like metaGPT [4] further add to these promises. Multiple LLM agents can divide and conquer complex tasks, coordinate workflows, and perform quality control, thereby executing code modification requests even more effectively.

This paper puts this potential to test in a real-world case study. Specifically, we examine how well state-of-the-art LLMs and multi-agent systems can execute social scientists' requests to modify Truman, a 20,000-line web app and platform created for running social science experiments. Social scientists requested modifications to Truman for their respective experimental needs, for example:

> If a user is in the experimental condition `users: ambiguous` or `users: unambiguous`, display a flag icon beneath the [social media] posts labeled `ambig_flag` or `unambig_flag`, and show a prompt box asking, "*Other users have flagged this comment as harassment. Do you agree?*" (Task no.14)

Such requests differ significantly from those examined in prior research on LLM or multi-agent systems' coding abilities, which focused on requests made by programmers (e.g., "*Set up JanusGraph and run it locally with an HTTP endpoint*" [11]) or by non-programmers' to create simple applications from scratch (e.g., create a chatbot that walks me through this recipe step-by-step [13, 14]). By focusing on non-programmers' code-modification tasks, this work addresses a critical gap in understanding LLMs' code generation capabilities and their potential for end-user programming.

We started by developing a framework for evaluating LLMs' abilities to execute non-programmers' code modification tasks. First, we developed several single-LLM (based on Claude) and multi-agent systems (based on MetaGPT [4]) and iteratively evaluated them on the social scientists' 23 modification tasks for Truman. This iterative process produced over 350 pieces of LLM-generated code. Next, we ran all these pieces of code, manually reviewed results, identified common failure modes, and developed an evaluation framework that can identify these errors. We named this framework TrumanBench. Finally, we formally evaluated three top-performing systems on all 23 tasks across five trials using TrumanBench. Our evaluation revealed three key findings:

1. LLMs were not yet capable of reliably executing non-programmers' requests to modify a large, real-world code-base. The best-performing systems, on average, generated code that interpreted the requests semi-correctly, executed them partially, while altering unrelated parts of the codebase, though not enough to break the entire app;
2. Contrary to popular belief, adding more LLMs or more agents did not necessarily help. In this case study, reducing the number of agents in our multi-agent system improved its performance. Moreover, our top-performing multi-agent system completed fewer tasks than our best-performing single LLM;

*Both authors contributed equally to this research.

HEAL@CHI'25, April 26, 2025, Yokohama, Japan

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

3. The LLMs and multi-agent systems struggled with handling cross-file dependencies even at similar levels of task complexity.

This paper makes three contributions. First, it presents TrumanBench, the first framework for evaluating intelligent systems' ability to execute non-programmers' code modification requests. Second, it offers a profile of three LLM-based systems' performance in executing non-programmers' requests to modify a real-world, 2,000-line codebase. Third, it identifies opportunities for future work, including near-future opportunities in designing end-user code modification tools using today's LLMs, and research opportunities around tailoring future open-source codebases for end-user inquires via LLMs.

2 RELATED WORK

2.1 Evaluating LLM on Code Modification

LLMs such as GPT and Claude hold exciting promises for end-user programming, as they possess an unprecedented ability to generate code based on natural language instructions—a paradigm sometimes referred to as Malleable Software [8]. Multi-agent systems built upon these LLMs further add to this promise. For example, architectures like MetaGPT [4] and CHATDEV [10] assemble a set of LLM-based agents, each assigned specific roles such as project managers, developers, and quality testers, in accordance with Standardized Operating Procedures (SOPs). These architectures have demonstrated stronger software engineering abilities than singular LLMs, as measured by popular benchmarks.

However, no existing benchmarks or evaluative frameworks have utilized end-user code modification requests to assess LLMs' code generation capabilities [11, 12]. Instead, they focused on requests made by programmers, such as GitHub pull requests [2, 5–7, 11].

One likely reason for this gap is that it is very difficult to assess LLMs' ability to execute code-modification requests from non-programmers *at scale*. Non-programmers' requests can be ambiguous, hence lack clear criteria for success. In contrast, programmers' code-modification requests, such as GitHub pull requests, are often framed to highlight specific bugs or outcomes; Some even provided their own unit or Oracle tests. As a result, the success of these tasks can be easily or automatically measured. As prior research prioritized evaluating LLMs' code-modification abilities at scale, they almost always carefully chose such programmers' tasks [5, 11].

We wanted to better understand LLMs' ability in executing non-programmers' code-modification requests. To this end, we made some compromises on scalability and concentrated on a single real-world codebase and non-programmers' needs to modify it. We hope this case study will provide initial insights into LLMs' capabilities in end-user code modification, and that future research can build on this work to make this type of LLM evaluation more scalable.

2.2 The Truman Codebase and Social Scientists' Needs to Modify It

In this paper, we focus on Truman, a 20,000-line web app that is a simulated social media platform created for social media research [3]. It is primarily written in JavaScript and uses frameworks like Node.js and Pug. Since its launch in 2018, social scientists have increasingly turned to Truman for hosting human-subject experiments, manipulating the interface, algorithms, and policy designs of the simulated social media platform in various ways and analyzing its users' responses. Truman's popularity further grew in recent years, as real-world social media platforms increasingly restricted access to their data and research APIs.

The social scientists, most of whom lack programming training, needed to customize Truman to meet their experimental needs. For instance, they often wanted the platform to exhibit different behaviors for control and experimental groups. Many hired computer scientists to make these customizations, enabling them to conduct their experiments and produce many publications (e.g., [1, 9]). However, many struggled and were unable to make the necessary customizations. This paper explores the extent to which existing LLMs can help.

3 METHOD

We wanted to investigate how effectively state-of-the-art LLMs and multi-agent systems can assist social scientists in customizing Truman for their diverse experimental needs. We hope this investigation will serve as a case study for understanding LLMs' potential for end-user code-modification more broadly.

We undertook a four-step process to achieve this goal. First, we collected 23 modification requests for Truman that social scientists had previously described in their publications. Next, we developed several single-LLM and multi-agent systems, iteratively evaluating them on the 23 tasks. This process resulted in over 350 pieces of LLM-generated code. We then ran all these pieces of code, manually reviewed results, identified common failure modes, and developed an evaluation framework for identifying these errors. We named this evaluative framework TrumanBench. Finally, we formally evaluated three top-performing systems on all 23 tasks across five trials using TrumanBench.

3.1 Collecting Codebase Modification Tasks

We started by collecting social scientists' requests for modifying Truman; requests that reflect real-world non-programmers' needs to modify open-source software. To achieve this, we manually reviewed publications that referenced Truman and identified 23 modifications that social scientists had made to the platform across 11 studies. These tasks fall into five broad categories with respect to the changes in platform behavior they aim to achieve:

- *Adding a feature*, such as task 9: If the user is in the experimental group `view: large`, `view:small`, or `view:none`, then when they scroll past each post, display an opaque overlay over the post. This overlay should have the following: a large eye icon, the text "*You've read this!*", and a black button "Read Again?" [...];
- *Removing a feature*, such as task 18: Remove the functionality to flag (social media) comments;
- *Making an requirement optional*, such as task 19: Make submitting an image optional when creating a post;
- *Assigning experimental conditions*, such as task 2: When a user creates an account, randomly assign them to one of 4 experimental conditions: `none:view`, `empathy:view`, `none:none`, `empathy:none`. This information should remain hidden from the user;

- *Reordering social media feed in a certain way*, such as task 13: Show the post that has the comment labeled `ambig_flag` or `unambig_flag` at the top of the timeline each day.

Regarding difficulty levels, we estimated that for a competent programmer—such as a senior in a computer science undergraduate program who is already familiar with the codebase—12 of the 23 tasks would take only minutes to complete, 10 tasks would require hours, and 1 task would take about a day.

3.2 Creating LLM and Multi-Agent Systems

We underwent an iterative process of developing different variations of a single-LLM system (based on Claude-3-5-Sonnet) and a multi-agent system (based on GPT-4-Turbo and MetaGPT) and evaluating them on the 23 tasks.

Let us illustrate this iterative process by showing how we enhanced the multi-agent system's ability to identify the right file to modify. We initially implemented this system using the SOP framework from MetaGPT [4]. The system consisted of many LLM agents, each assigned roles such as project manager, spec writer, tech lead, developer, quality assurance (QA) agent, and more. However, we found that the system struggled with most tasks due to difficulties in identifying the specific lines of code that required modification. To address this, we experimented with several approaches, listed below, and ultimately adopted 2, 3, and 4 because their combination yielded the best performance.

1. Adding a Retrieval-Augmented Generation (RAG) engine, which dynamically selects the most relevant files based on the embedding similarity between the task instruction and the stored document embeddings;
2. Adding a "file identifier" LLM agent;
3. Removing non-essential agents, specifically the "project manager", "spec writer", and "tech lead". We found that these agents often added unnecessary details to codebase-modification request, making the system more error-prone in identifying relevant files;
4. Adding to the codebase a file directory, which includes two new manually written files: `file_structure.json`, which outlines the locations of each file within the codebase, and `file_description.json`, which offers a two-sentence summary of each file's function;

In addition to this example, we also improved the multi-agent system by experimenting with various implementations of the QA agent for quality control before eventually removing it, by adding a "file replacer" agent to ensure the system generates descriptions of changes to existing code (rather than standalone new code), along with many other improvements. We performed similar experimentation with the single LLM system.

After all this experimentation, we identified three top-performing systems for final evaluation.

- "*The Claude-based system*", which prompts Claude-3-5-Sonnet using a three-part prompt: the evaluation task described in natural language, the entire code base serialized into a string, and a template for the output:
```
=== /path/to/modified_file1 ===
[Complete content of modified_file1]
===/path/to/modified_file2 ===
[Complete content of modified_file2]
[etc...];
```
- "*Claude with file directory*", which includes the Claude-based system and the file directory files;
- "*MetaGPT with file directory*," which includes a system of three agents (a file identifier, a developer, and a file replacer) using the MetaGPT architecture, along with two file directory files[1].

3.3 Analyzing System Performance

We evaluated each work-in-progress system by feeding it one task description at a time[2] and manually reviewing the outcomes. We then made improvements to the system accordingly. This iterative process produced over 350 different LLM-generated code changes, the analysis of which revealed common system failure modes. We synthesized these findings into an evaluative framework, namely TrumanBench. For the final three systems, we tested each one on all 23 tasks five times to rigorously evaluate both their performance and consistency. We manually graded the code modifications generated by each system during each trial using TrumanBench. The next chapter first provides an overview of TrumanBench and then presents the results of this formal evaluation.

4 FINDINGS

4.1 TrumanBench, an Evaluative Framework

We present TrumanBench, an initial framework for evaluating AI systems' ability to execute non-programmers' code modification requests (Table 1.) It includes three components:

- *A set of questions that categorize the request into types*, which helps stratify the request's difficulty level. These questions assess, for example, whether fulfilling the request requires changes to multiple files or just one, and whether it involves front-end changes, back-end changes, or both;
- *A set of rubrics for grading the system's output in response to the request.* These rubrics assess: (1) Does the generated code modification correctly interpret and follow the request? (2) How well does it fulfill the request? (3) To what extent did it alter or break unrelated parts of the codebase? Each criterion is scored as follows: 8 indicates complete correctness, meaning the system executed the task fully and accurately without unintentionally impacting the rest of the codebase; 5 indicates partial correctness; 2 indicates complete incorrectness; and 0 indicates that the generated code does not run, such as when it includes an emoji. Additionally, each system can earn 2 bonus points if the generated code adheres to software engineering best practices. For example, if a system could complete the task by simply importing another file from the codebase but instead rewrites that file, it receives 8 points. If it imports the file, it receives 10 points. If it fails to complete the task, it receives 5 points or lower.

[1]Note that there is no "metaGPT without file directory" system. Unlike Claude, the MetaGPT system cannot function without the file directory. The Truman codebase exceeds MetaGPT's context window size and cannot fit into its prompt. We address this issue by relying on the file identifier agent and the file directory files to dynamically identify relevant files and include only these files in the prompt.

[2]We created a Python pipeline to do so more efficiently. Taking inspiration from Jimenez et al. [5], this pipeline feeds one code-modification task into one system at a time, collects the code change the system proposes, then clones a new copy of the Truman code base, applying the code changes to this copy.

How difficult is the codebase modification task?	
Func Count	Does the task require modifications to multiple functions? Yes | No
File Count	Does the task require modifications to multiple files? Yes | No
Front/Backend	Does the task require modifications to the front end, the backend, or both? Front end | backend | both
Effort Level	How much time would this task take a component programmer who is familiar with this code base to accomplish? Minutes | hours | a day | days | weeks
How capable is the system in executing this task?	
Instruction Interpretation & Following	**8 pts** The code generated indicates that the system interpreted the instruction correctly. **5 pts** The code generated is relevant to the instruction, but indicates that the system misinterpreted it partially. **2 pts** The code generated is entirely irrelevant to the instruction. **0 pt** The code generated doesn't run.
Task Completion	**10 pts** The generated code not only accomplishes the task, but also adheres to software engineering best practices **??**. These best practices include: efficiency (e.g., if a task can be accomplished by modifying a single line of code, it doesn't rewrite an entire file), readability, maintainability, etc. **8 pts** The generated code accomplishes the task, but does not adhere to software engineering best practices. **5 pts** The generated code runs, but only partially accomplishes the task. **2 pts** The generated code runs, but does not perform the task at all. **0 pt** The code generated doesn't run.
Impact on the Rest of the Codebase	**8 pts** The code generated does not unintentionally impact the rest of the codebase. **5 pts** The code generated creates an unintended impact on the rest of the codebase, but the impact is not severe enough to jeopardize its user-facing functionalities. **2 pts** The code generated unintentionally jeopardizes other user-facing functionalities of the codebase. **0 pt** The code generated doesn't run.
How consistent is the system?	
Consistency	**Standard deviation of the scores above.** Provide the same codebase modification task to the same system multiple times. How consistent is the system's performance on the rubrics above?

Table 1: TrumanBench, a framework for evaluating AI systems' ability to execute natural langauge code modification requests.

- *Consistency measure.* This measures the stability of the grades across multiple runs.

We developed this evaluative framework based on the errors we observed while assessing various LLM systems on social scientists' Truman modification tasks. Thus, the framework can also be viewed as a taxonomy of errors that LLMs might make in handling code modification requests.

4.2 Evaluation Result Overview

Our evaluation suggests that state-of-the-art LLM-based systems are not yet capable of reliably executing non-programmers' requests to modify a large, real-world code-base. Three key results led to this claim.

Firstly, the three best-performing systems generated code modifications that, on average, scored between 6.3 and 7.0 on the three output quality measures (Figure 1). It means that the modifications interpreted social scientists' requests semi-correctly, executed them partially, while altering unrelated parts of the codebase, though not enough to break the entire web app.

Secondly, each LLM-based system shows highly inconsistent performance, as evidence in the high standard deviation of their output quality scores across all three metrics. For the two criteria with a maximum score of 8 points—instruction interpretation and impact on the rest of the codebase—the standard deviation of each system's performance ranges from 2.33 to 3.03. For the criterion with a maximum score of 10 points, the standard deviations range from 3.64 to 4.13. This indicates significant performance variability, encompassing nearly half of the full range of possible scores.

The inconsistency of the LLM systems is also evident in the unpredictability of when and where severe errors occur. 11.7-18.7%

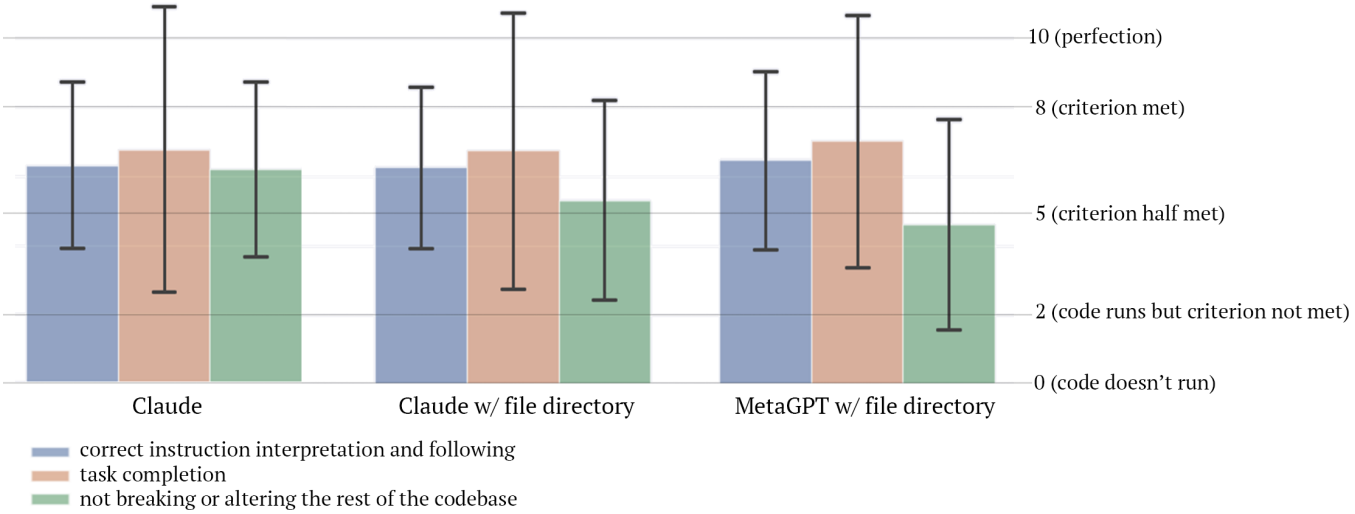

Figure 1: The performance of three state-of-the-art LLMs and multi-agent systems in executing social scientists' various requests for modifying Truman, a 2000-line web app and research software. These systems are not yet capable of reliably executing these tasks.

of the time, these systems generated code that doesn't run, indicating the lowest possible output quality. Importantly, such errors occurred across all types of tasks, rather than being concentrated in a few difficult ones. No single task or task type (e.g., adding versus removing features) consistently caused any of the systems to fail at this level. This was particularly true for single LLM systems, which completed more tasks reliably but performed worse on other tasks.

Thirdly, the most common and severe failure mode of these LLM systems is their tendency to break the rest of the Truman codebase. Among the three output quality measures, these systems scored the lowest on the third criterion: their impact on the rest of the codebase, with an average of 4.6-5.3 points out of eight.

To summarize, the three state-of-the-art LLM systems failed to execute tasks correctly, did not excel in any specific subset of tasks, and often broke the codebase when they made errors instead of merely failing to add new features. As they currently stand, these systems are not ready for real-world use.

4.3 More LLMs and More Agents Did Not Help

Contrary to popular belief, adding more LLMs or agents did not improve system performance (Figure 1). In this case study, while our best-performing single LLM successfully and consistently executed eight out of 23 tasks, the best-performing multi-agent system managed to do so with only two tasks. Furthermore, this top-performing multi-agent system comprised just three agents (file identifier, developer, and file replacer), outperforming systems with more agents.

One key reason for this is that adding an additional LLM instance or agent introduces new types of errors, and the errors made by different agents often compounded. For instance, the original metaGPT architecture [4] included a project manager agent, which is designed to coordinate other agents to divide and conquer the code-modification tasks. However, in this case study, the project manager agent often added new requirements to the social scientists' requests, derailing the work of downstream agents. To identify and address these errors, we included a Quality Assessment (QA) agent and experimented with various implementations. However, the QA agent itself also made errors, further complicating the workflow without yielding measurable improvements in outcomes.

4.4 Struggles with Cross-File Dependencies

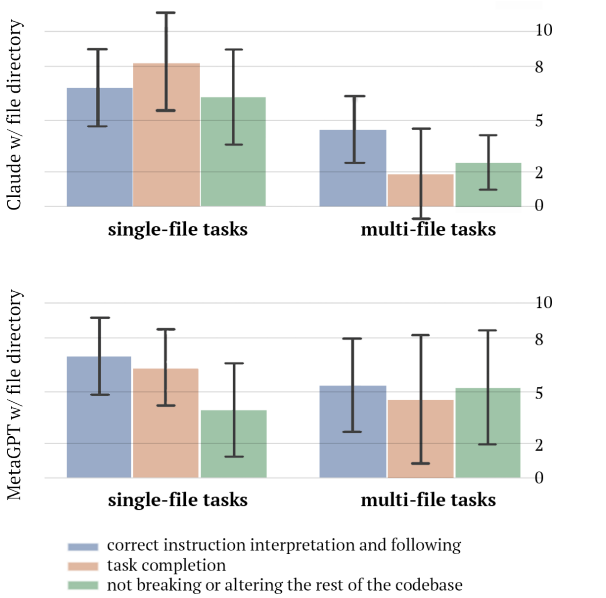

Figure 2: The performance of two top-performing LLM systems. They both were significantly better at tasks involving the modification of a single file compared to those involving multiple files.

A key reason for the LLM systems' suboptimal performance of these is their struggle with handling cross-file dependencies. All three best-performing LLM systems performed significantly better on tasks involving the modification of a single file than on those involving multiple files. In executing tasks that involved modifying a single file, Claude-generated code almost scored 8 points on average, indicating consistent, successful task completion (Figure 2, top left figure, red bar). However, when faced with multi-file tasks, these systems struggled. They often failed to identify the multiple files to modify (Figure 2, figures on the right, blue bars) and sometimes attempted to create a new file from scratch to compensate for a file they could not locate in the codebase. At other times, they had difficulty making the necessary modifications across these files (Figure 2, figures on the right, red bars) or avoiding the disruption of unrelated files (green bars).

This limitation of LLM-based systems significantly impacted overall performance because most code modification tasks involved multiple files. This issue arises not only from the inherent nature of the tasks (e.g., modifying both the front end and the back end) but also from the design of the Truman codebase. On the one hand, Truman took a PUG-based template approach, which seperates the presentation layer (HTML) and the underlying logic of the web app in different files. Many modification requests from social scientists involved changes to both. On the other hand, Truman kept the variables that social scientists frequently needed to modify into a single .ENV file. While this setup makes it easier for social scientists to make manual edits, it complicates matters for LLMs. When performing any task involving these variables, LLMs must understand the cross-file dependencies between the .ENV file and the other files from which the variables originate—a task that LLMs often struggled with.

We attempted to improve Claude's understanding of cross-file dependencies by adding a file directory to its prompt, but this change led to minimal improvement overall (Figure 1 left and middle subfigures). It did enhance Claude's performance on tasks where it was already fairly successful. Among the 23 modification tasks across five trials, Claude with the file directory consistently completed 15 tasks, one more than Claude without the file directory. However, the file directory also led to an increase in severe errors: Among the 115 outputs generated (23 tasks x 5 trials), Claude with a file directory produced code that failed to run 16.5% of the time, worse than Claude without file directory (11.7%) or MetaGPT (18.7%.)

5 CONCLUSION

This paper aims to profile the capabilities of state-of-the-art LLMs and multi-agent systems in executing social scientists' requests to modify Truman, a 20,000-line web app that is a social science experimental platform. LLMs have significant potential in end-user code modification, as they can empower non-programmers and greatly amplify the impact of open-source software. However, these capabilities remain under-studied. We hope that this case study can serve as an initial step toward understanding and improving LLMs' abilities in end-user code modification more broadly.

Our case study shows that LLMs currently cannot reliably execute the social scientists' requests, and contrary to popular belief, adding more LLM agents did not necessarily help. A key reason for this is that LLMs struggle with cross-file dependencies, making them capable of completing the few tasks that involve modifying only one file, but not yet proficient in handling multi-file tasks.

Nevertheless, our research process yielded several important insights that can be valuable for future HCI and NLP research. The first is TrumanBench, a framework for assessing LLM systems' abilities to execute non-programmers' code modification instructions. While this case study shows that current LLMs and multi-agent architectures may not yet be ready for real-world end-user code-modification, we hope that this framework can help researchers benchmark and track the rapidly evolving capabilities of future LLMs and multi-agent architectures. As such, they can seize opportunities and build no-code code modification tools once the technology is ready.

Second, we see opportunities in designing end-user code modification tools for tasks that involve understanding or modifying a single file. While simple from a programming perspective, single-file tasks can be highly valuable for end users. In the case of Truman, many essential needs of its users—such as randomly assigning participants to experimental groups and adding LLM agents to the simulation—can be accomplished by modifying just one CSV file in the backend. We suspect such needs exist for many other open-source software and their users. We encourage designers of end-user programming tools to identify these needs, as they present immediate opportunities for current LLMs to make an impact.

Lastly, we see exciting new research opportunities in designing open-source codebases specifically for end-user modifications via LLMs. In this case study, we observed that LLM systems struggled with modifying the codebase, because it was designed for easier human modifications (e.g., aggregating frequently modified variables into one .env file) rather than LLM interactions. This raises the question: What might an open-source codebase look like if it were designed for LLMs? Even more fundamentally, how might we design open-source software and end-user code-modification tools in tandem, such that they together can maximally harness LLMs' code-generation capabilities while mitigating their limitations? These are important questions to investigate, as we move toward a future where end users may increasingly modify code through LLMs instead of manually. This study offers one initial insight into these questions: Open-source codebases may become easier for LLMs to modify if they use simpler file structures, such as REACT instead of PUG, and avoid aggregating commonly modified variables (a common practice in current open-source software development). These simple changes can reduce the need for LLMs to understand and modify multiple files simultaneously, a task they currently struggle with. We hope future research can build upon these emergent insights and join us in investigating how to harness LLMs best for end-user programming.

ACKNOWLEDGMENTS

We thank Beichen Ma and Winice Hui for their contributions to earlier incarnations of the LLM systems described in this paper. We thank Professor Natalie Bazarova and Professor Dominic DiFranzo for offering insights about Truman and its social scientist users.

This work is supported by Google Research gift <Real-Time AI for Domain Experts>. This material is also based upon work

supported by the National Science Foundation under Grant No. 2313078. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Aparajita Bhandari, Marie Ozanne, Natalya N Bazarova, and Dominic DiFranzo. 2021. Do you care who flagged this post? Effects of moderator visibility on bystander behavior. *Journal of Computer-Mediated Communication* 26, 5 (2021), 284–300.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/arXiv.2107.03374 arXiv:2107.03374 [cs] version: 2.

[3] Dominic DiFranzo and Natalie Bazarova. 2018. The Truman Platform: Social Media Simulation for Experimental Research. In *ICSWM Workshop on Bridging the Lab and the Field*. https://socialmedialab.cornell.edu/the-truman-platform

[4] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *ICLR*. https://openreview.net/forum?id=VtmBAGCN7o

[5] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[6] Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, Grusha Prasad, Amanpreet Singh, Pratik Ringshia, Zhiyi Ma, Tristan Thrush, Sebastian Riedel, Zeerak Waseem, Pontus Stenetorp, Robin Jia, Mohit Bansal, Christopher Potts, and Adina Williams. 2021. Dynabench: Rethinking Benchmarking in NLP. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 4110–4124.

[7] Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. 2024. DevBench: A Comprehensive Benchmark for Software Development. *arXiv preprint arXiv:2403.08604* (2024).

[8] Geoffrey Litt. 2023. Malleable software in the age of LLMs. https://www.geoffreylitt.com/2023/03/25/llm-end-user-programming.html

[9] Philipp K Masur, Dominic DiFranzo, and Natalie N Bazarova. 2021. Behavioral contagion on social media: Effects of social norms, design interventions, and critical media literacy on self-disclosure. *Plos one* 16, 7 (2021), e0254670.

[10] Cheng Peng, Kaige Xue, Yunfan Shao, Xiyang Zhang, Yilun Du, Wenchang Ma, Tong Zhang, Yong Jiang, Chao Yang, Zhouchen Lin, and Yuandong Tian. 2023. ChatDev: Developing Software via LLM-based Multi-Agent Collaboration. In *Proceedings of the 2023 Conference on Neural Information Processing Systems (NeurIPS)*. https://arxiv.org/abs/2307.07924

[11] Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. 2024. TheAgentCompany: Benchmarking LLM Agents on Consequential Real World Tasks. arXiv:2412.14161 [cs.CL] https://arxiv.org/abs/2412.14161

[12] Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. 2025. Survey on Evaluation of LLM-based Agents. arXiv:2503.16416 [cs.AI] https://arxiv.org/abs/2503.16416

[13] J.D. Zamfirescu-Pereira, Eunice Jun, Michael Terry, Qian Yang, and Bjoern Hartmann. 2025. Beyond Code Generation: LLM-supported Exploration of the Program Design Space. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*.

[14] J.D. Zamfirescu-Pereira, Heather Wei, Amy Xiao, Kitty Gu, Grace Jung, Matthew G Lee, Bjoern Hartmann, and Qian Yang. 2023. Herding AI cats: Lessons from designing a chatbot by prompting GPT-3. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference*. 2206–2220.