FEEDBOT: Formative Design Feedback on Programming Assignments

Elaine Zhu
zhu.lae@northeastern.edu
Northeastern University
Boston, MA, USA

Smaran Teja
smaranteja32@gmail.com
Northeastern University
Boston, MA, USA

Chris Coombes
coombes.c@northeastern.edu
Northeastern University
Boston, MA, USA

Daniel Patterson
dbp@dbpmail.net
Northeastern University
Boston, MA, USA

Abstract

This paper describes FEEDBOT, an open-source formative assessment tool leveraging large language models (LLMs) to provide structured, high-level feedback on design-oriented programming assignments. Designed to address the limitations of traditional autograding and overcome scaling challenges of formative assessment, FEEDBOT uses an existing pedagogical framework to provide targeted but limited actionable feedback. Early results demonstrate measurable improvements in student performance in a large introductory computer science course, while avoiding the pitfall of providing too much assistance that more unstructured tools commonly encounter. Although this experience report focuses on a particular implementation, we believe that FEEDBOT (and its general approach) is adaptable to many other contexts.

CCS Concepts

• **Applied computing → Education**; • **Computing methodologies → Artificial intelligence**.

Keywords

formative, autograding, llm, feedback, education

ACM Reference Format:
Elaine Zhu, Smaran Teja, Chris Coombes, and Daniel Patterson. 2025. FEEDBOT: Formative Design Feedback on Programming Assignments. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2025), June 27-July 2, 2025, Nijmegen, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3724363.3729063

1 Introduction

Formative assessment [4] is important for learning and thus reducing education inequities [2, 12]. However, in large classes, providing iterative feedback by hand quickly becomes unmanageable. Existing automated grading systems (primarily unit-test-based) partially mitigate this but restrict assignment variety and feedback granularity. FEEDBOT leverages large language models (LLMs) to offer high-level, structured feedback on diverse, design-oriented programming tasks in an introductory computer science course. By integrating the Design Recipe [6, 16], which structures problems into sequential steps, FEEDBOT identifies the first problematic step and provides limited but actionable hints. This approach preserves student agency by not revealing solutions.

Briefly, the Design Recipe proceeds as follows: data design is accomplished by (1) identifying the values that make up the data, (2) describing (in a comment) how the data will be used in the program, (3) writing down examples of the data, and (4) constructing an example function "template" showing how the data might be used. Function design is done by (1) describing the input and output types of the function, (2) writing down a concise statement (in English) of what the function does, (3) writing example function uses (test cases), and (4) implementing the function.

Since FEEDBOT knows whether a given problem is a data design problem or a function design problem, its task is to: (1) assess whether all four steps have been completed satisfactorily and if not, (2) identify the first step that needs work. Since the steps are intentionally sequential, the first step with issues is the ideal place for a student to begin improving their work. Additionally, since the steps are granular yet general, the mere identification of that step strikes a good balance of giving students guidance without giving them so much assistance that they do not learn from doing the work on their own. While FEEDBOT is not bound to the Design Recipe, its success is certainly due to the sequential and granular nature of this pedagogical framework, and likely without it the tool would have similar struggles to other tools in this space.

1.1 Contributions

This paper presents the following contributions:

(1) **Tool Design:** Introduces FEEDBOT, an open-source LLM-based system for structured feedback on design-oriented assignments.
(2) **Pedagogical Integration:** Demonstrates how the Design Recipe [6] helps guide the feedback process.
(3) **Evaluation of Effectiveness:** Presents initial evidence suggesting that FEEDBOT usage correlates with higher performance.
(4) **Scalability and Reproducibility:** Explores how FEEDBOT scales to large courses, with notes on adoption and adaptation challenges.

1.2 Structure of paper

The remainder of this paper is structured as follows:

- §2: **Related Work** explores existing literature on formative assessments, autograding systems, and the use of LLMs in education, identifying gaps that FEEDBOT addresses.
- §3: **Implementation** outlines the design, methodology, and key features of the tool.

ITiCSE 2025, June 27-July 2, 2025, Nijmegen, Netherlands

ACM ISBN 979-8-4007-1567-9/2025/06
https://doi.org/10.1145/3724363.3729063

- §4: **Findings and Results** presents both quantitative and qualitative analyses of FEEDBOT 's effectiveness.
- §5: **Conclusion** summarizes key contributions, findings, and directions for future work.

FEEDBOT is open source, available at https://github.com/NUFeedBot.

2 Related Work

The potential of generative AI technologies, particularly large language models (LLMs), in enhancing teaching and learning practices has been increasingly recognized. Baidoo-Anu and Owusu Ansah [1] explored the application of ChatGPT in promoting education, highlighting its role in fostering accessibility and engagement in classrooms through personalized interactions and immediate feedback mechanisms.

Mastery learning, introduced by Bloom [4], provides a theoretical foundation for formative assessment approaches, focusing on achieving mastery of topics through iterative improvement. Building upon these principles, Felleisen et al. [6] introduced the Design Recipe, a structured methodology that guides students in problem solving and program design. The Design Recipe's focus on concreteness fading, a method reviewed systematically by Fyfe et al. [7], demonstrates the educational benefits of gradually transitioning from concrete to abstract representations in teaching programming concepts.

Recent research has evaluated the effectiveness of LLMs in supporting students' learning processes. Hellas et al. [9] analyzed how LLMs respond to beginner programmers' help requests, finding potential in their ability to scaffold learning but identifying challenges in addressing misconceptions.

Garcia et al. [8] and Lionelle et al. [14] emphasized the scalability of LLM-driven feedback in large courses, introducing grading frameworks that balance formative and summative assessment needs.

Ren et al. [15] provided insights into the types of help students seek during TA office hours, a study that informed the integration of structured feedback mechanisms in tools like FEEDBOT. Complementing this, Tuson and Hickey [18] proposed mastery-based grading systems with specifications (specs) grading, aligning assessments with well-defined, modular learning outcomes.

There have also been many LLM-powered tools developed. CodeHelp, developed by Liffiton et al. [13, 17], leverages LLMs to provide scalable support with embedded guardrails to ensure pedagogical alignment. This tool was further studied by Denny et al. [5] (in a limited, 13 day period), where they collected qualitative feedback from students. Highlights from the aforementioned tools support the design of FEEDBOT– that AI assistants should guide students where to work but should not solve problems for them. Unlike FEEDBOT, CodeHelp provides open-ended feedback and thus is somewhat prone to assisting too much (partially addressed by introducing keyword blacklists). This is, of course, a trade-off, as FEEDBOT does not have a mechanism to explain concepts, which a more general "AI Assistant" clearly would have.

A similar tool is CodeAid [11], which adopts the strategy of only explaining in terms of pseudocode, but still explains in high levels of detail how to solve problems, and thus possibly stands in for student learning, unlike FEEDBOT. Giving solutions as pseudocode and leaving the only learning task a translation to real code seems a real detriment.

Bassner et al. [3] built Iris, which allows open-ended questions but filters them using extensive prompting via a similar strategy as FEEDBOT to ensure that the output does not contain solutions or too much help. It is a much heavier weight solution, as it requires integration into an interactive learning platform, and unlike FEEDBOT, which suggests a place where a student has made a mistake, Iris doesn't provide assistance outside of what students explicitly ask for.

Jacobs and Jaschke [10] built Tutor Kai, which gives feedback on programming tasks, but with single functions and with extensive feedback. Perhaps partly due to the model (GPT-4) or partly due to the level of detail in the feedback (unlike FEEDBOT, which simply identifies a step in the Design Recipe where a mistake occurs, Tutor Kai describes exactly what is wrong, and if there are multiple mistakes, describes all), the authors ran into problems where the tool produced hallucinations and mistakes.

While these (and many other) tools use LLMs, most provide nearly unaltered output from LLMs, even with extensive prompt engineering, which often means that there are risks of providing too much help to students. This is very different from FEEDBOT, where the feedback is intentionally very limited: while the output from the model is extensive, we include a delimiter and then an extremely short response after it, and only display the latter to students. Because FEEDBOT fits into an existing pedagogical framework that is highly structured, it can refer students back to the step of the design process they should focus on with minimal additional help. Thus, it can use the LLM to identify where a student should focus their work, rather than telling them what to do, or even what exactly is wrong, since the step to focus on should allow them to do their own reasoning.

3 Implementation

FEEDBOT uses OpenAI's `o1-mini` to analyze student submissions and provide feedback aligned with the Design Recipe framework. FEEDBOT has been prototyped using Racket's Teaching Languages, but is not tied to Racket. It does, however, rely upon the structured and sequential approach of the Design Recipe, and in order to be adapted to a different pedagogical approach, a similar structure would have to be identified. This is because the primary feedback is pointing students to which step in the process they should focus their attention on.

3.1 Prototyping

Initial experiments were conducted using the OpenAI API (first with `GPT-4-turbo-preview`, and later with `GPT-4o` and `o1-mini` when those models became available). Experiments assessed the feasibility of using commercial LLMs to provide actionable feedback on student programming assignments. They demonstrated the models' capability to understand student code in the Racket Teaching Languages, identify deviations from reference solutions, and generate feedback aligned with high-level design principles.

The prompt was refined iteratively to provide more accurate and consistent feedback. Many components align with known "best practices" in prompt engineering, including:

- providing a persona for the LLM ("You are FeedBot...")
- providing a clear step-by-step process
- requesting chain-of-thought output (this was particularly effective in improving consistency, since it separated the tasks of code analysis and feedback writing)

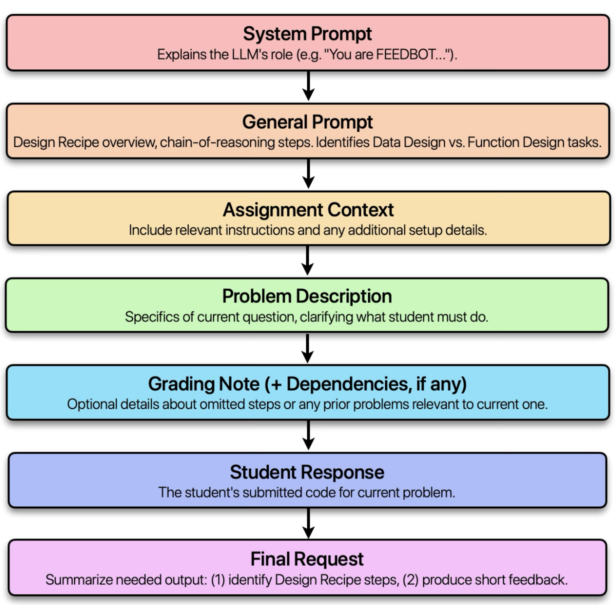
For testing, we designed a system to generate responses with multiple prompts, multiple times per prompt, and present them side-by-side. This allowed direct analysis of differences in quality and replicability between prompt iterations.

These experiments utilized the Design Recipe framework[6]. This structured methodology not only facilitates effective feedback but also aligns with the types of assistance students typically seek from teaching assistants. While the experiments focused on the Beginning Student Language dialect of Racket, the findings suggest that the approach could be extended to other languages like Python or Java, though additional structuring may be required for providing similar pedagogically sound feedback in those contexts.

While the results from closed-source models are variable, we did notice a consistent problem with `GPT-4o` where it would occasionally not notice type signatures despite them being present. This issue, which we identified during the final round of testing right at the beginning of the semester, motivated the switch to `o1-mini` (which was released right at that time), which did not have the same weakness.

3.2 Prompt Structure

For each question, the final prompt was structured as follows:



We found this structure provides reasonably accurate feedback, particularly due to the chain-of-reasoning style prompting, even though the particular LLM we used (OpenAI's `o1-mini`) already claimed to be a "reasoning model". We did make a few minor tweaks/additions to the prompt during the semester as we encountered some misleading feedback, but the structure of the prompt was unchanged. The refinement step asks the model to provide output after a delimiter, and we only provide the post-delimiter output to the student.

3.3 Demonstration

In this subsection, we show an example problem and the raw output that the prompt produces. Note, importantly, that after giving extensive feedback, it inserts a delimiter and provides a very concise summary of only the first issue, which is what is shown to the student. While this was very reliable, we also included additional failsafes like stripping markdown code from the output (as any code it outputs is a bug, and code always appears inside markdown code marks).

This problem, which appeared in HW7 (see §4 for the array of assignments we ran the tool on) occurred during an assignment where students were tasked with manipulating survey data. The task involved converting a list of student responses containing names into an anonymized list without names, adhering to structured design principles. Below is an example of a submission and the feedback generated by FEEDBOT using our standard prompt:

Submission.

```
(define-struct student/v4 [age demographic legacy?])
;; A student/v4 is a:
;; (make-student/v4 Integer String Boolean)
;; - age: The age of the student
;; - demographic: The ethnicity of the student
;; - legacy?: Whether the student is a legacy student or not

(define STUDENT1 (make-student/v4 19 "White" #false))
(define STUDENT2
  (make-student/v4 20 "Black or African American"
                   #true))

;; anonymize: [Listof student/v4] -> [Listof student/v4]
;; Removes names from the list of students
(check-expect (anonymize '()) '())
(check-expect (anonymize (list STUDENT1 STUDENT2))
              (list STUDENT1 STUDENT2))

(define (anonymize students)
  (cond
    [(empty? students) '()]
    [else
     (cons (make-student/v4
             (student/v4-age (first students))
             (student/v4-demographic (first students))
             (student/v4-legacy? (first students)))
           (anonymize (rest students)))]))
```

Feedback. Running FEEDBOT on the above submission, which has a dependency on a prior problem (where the non-anonymized student was defined as `student/v3`) results in the following response, with what comes after the delimiter (======) shown to students:

```
### PART 1:
1. Signature for anonymize
   (a) No
  (b) The signature in [Student Code] does not correctly
```

```
   reflect the required input and output types.
2. Purpose Statement for anonymize
   (a) No
   (b) It is satisfactorily completed in [Student Code].
3. Tests for anonymize
   (a) No
   (b) Only two tests are provided; at least three are
   required.
4. Function Body for anonymize
   (a) No
   (b) It is satisfactorily completed in [Student Code].

======
The signature for `anonymize` does not correctly represent
the function's input and output types as required.
Please revise it to match the problem statement.
```

Before we consider the summary (after the ======), note that in order to avoid issues where the model missed instructor provided parts, part (a) asked the model to locate the step *outside* student code, and only if it was absent, to proceed with (b), feedback on the student code. Since the above problem had students complete all four steps, all part (a) responses are No.

We highlight two important parts of the summary. First, not only is the identification of the problem correct – the type signature is incorrect, since the input should have been a `[Listof student/v3]` (what they defined previously, which had names), rather than the anonymized definition (`[Listof student/v4]`), FEEDBOT did not tell them exactly what was wrong. Instead, it pointed them to the signature step, so the student would still work on their own to figure out exactly what the signature should be. Second, note that while FEEDBOT identified that the student should have included more test cases, it did not provide that feedback at all, since it had already identified an earlier step with an issue.

3.4 Dependencies

A key element that allows FEEDBOT to work on real assignments is that it supports problems that depend on one another. As in the above example, sometimes in one problem students would design a data definition, and then later use it in another problem. While we wanted to provide isolated feedback for each problem to increase precision, problem dependencies need to be dealt with, and simply providing the entire assignment for a single piece of feedback is not a great idea. Not only would this reduce the likelihood of good feedback, as the model would have to reason about much more code at once, but while errors on earlier steps of the Design Recipe should result in no feedback on later steps, we did not want errors on earlier problems to affect the feedback on later problems.

This meant that while we ran one query per problem, we included dependencies where necessary and included in the prompt that only the current problem should receive feedback.

This necessitates splitting the assignment submission files, which were ordinary text files, using particular comment delimiters (arbitrarily chosen as `;;!`, given `;` begins a line comment in Racket) into individual problems, where each problem has a description and student-produced code in response. We then used a second delimiter ";;!! Write your response below" to distinguish problem descriptions from student-produced responses, which allows us to extract just the student code, and prevent student modifications of the assignment, which occasionally happened, from interfering with our feedback.

An example outline of an assignment is shown below:

```
;;!Problem 1
Problem 1 overall description

;;!Part A
Problem 1, part A description
;;!! Write your response below
...student response...

;;!Part B
Problem 1, part B description
;;!! Write your response below
...student response...

;;!Problem 2
Problem 2 description
;;!! Write your response below
...student response...
```

To track problem types and determine which problems would get feedback from FEEDBOT, each assignment has a manually created JSON file that specified, among other things, the problem dependencies for each problem. Problems are described by *paths*, which are ordered sequences of strings following the special `;;!` delimiter. For the example above, the paths for the problems would be `["Problem 1", "Part A"]`, `["Problem 1", "Part B"]` and `["Problem 2"]`. Each problem then has an optional list of dependent problems that the LLM requires to fully understand the problem. For instance, in the example above `["Problem 1", "Part A"]` may be listed as a dependency of `["Problem 1", "Part B"]`.

3.5 Other Metadata

In addition to the problem paths, each problem is categorized as either a Data Design (DD) or Function Design (FD) task. We crafted specific prompts for each category, explaining what each of these "recipes" are and the expected steps that a student is required to complete for each type of problem.

Finally, problems have optional "grading notes", which allow us to include problem-specific feedback instructions. These notes are particularly helpful for problems where certain parts of the recipe are predefined or omitted. For example, if a problem includes a signature and purpose statement, students are only required to complete the tests and implementation steps. The default prompts assume all recipe steps must be completed, leading the LLM to request omitted steps due to how strongly we prompt it to ensure the student includes all steps. Reiterating that certain parts of the recipe can be omitted in the grading note for these types of problems helped resolve this issue. This also helped resolve occasional oddities, like when the model got confused about a struct field name ending in a ? and thought uses of it were for an unimplemented function.

3.6 Integration into Gradescope

While the feedback from FEEDBOT was entirely unrelated to grading, we used the submission & autograder mechanism from Gradescope in order to handle student submissions. When a student submitted an assignment, in addition to ordinary autograder tests that ran before the assignment deadline, we would run the client for FEEDBOT, which was responsible for extracting problems and generating prompts according to §3.2. It would then submit those to the FEEDBOT server, which existed externally in order to defer actually running the LLM queries until students went to view the output. This was a serious cost savings, given that only around half of students viewed the results of FEEDBOT. The server also had buttons that allowed students to give feedback on the quality of response (see §4.1 for results from those). Once the prompt was submitted to the FEEDBOT server, it returned a URL that was displayed in the autograder output. When students visited the link, they would see a loading page while the query was run, and around 10 seconds later would get the feedback.

Partly to control cost, and partly because we wanted students to spend time thinking about the feedback before resubmitting, we implemented both a cooldown period and an overall rate limit. For our implementation, we only ran FEEDBOT if it had not been run in the last hour (other submissions could have occurred), and only ran a total of 5 submissions per assignment.

We tried to implement both using Gradescope metadata, only finding out after the semester ended that the unreliability was due to them not implementing the metadata on past submissions.

4 Findings and Results

The bulk of the development on FEEDBOT was done over the spring and summer preceding the semester whose results we are describing. While that testing was done with other models, the actual use of FEEDBOT that this experience report describes was all done with OpenAI's `o1-mini`. Due to integration details unrelated to FEEDBOT, including being unsure whether we wanted to provide the feedback from the very beginning of semester, we didn't introduce FEEDBOT until partway through the semester. We also ran it for one assignment with the results only visible to TAs in order to identify any remaining bugs in the tool or, more likely, prompt. As a result, we used it fully on seven homework assignments, in an introductory programming course with around 500 students.

4.1 Quantitative Analysis

Since the manual grading that captures the design skills that FEEDBOT was intended to aid with only happened after the assignment deadline (and only a single time), measuring the impact of FEEDBOT is challenging. We can, however, segment students on each assignment based on whether they viewed the output of the tool (at least once) and those that did not.

From that initial analysis, we find (see Table 1; counts of students are in parentheses) that those that used FEEDBOT did notably better in almost every assignment where it was available. The outlier to this phenomenon, HW9, where the difference was only 2.3%, was an assignment where we included signatures & purpose statements (the first two parts of the design process) as part of the assignment, so students only had to write tests & implementations, both of

Table 1: Average Scores of Students

HW	Never Viewed	Viewed At Least Once	Delta
HW6	81.4% (340)	90.7% (179)	+9.3%
HW7	79.3% (232)	87.4% (283)	+8.1%
HW8	86.5% (261)	91.1% (240)	+4.6%
HW9	93.6% (301)	95.9% (211)	+2.3%
HW10	88.5% (230)	93.9% (283)	+5.4%
HW11	83.5% (288)	90.6% (218)	+7.1%
HW12	72.3% (240)	82.2% (212)	+9.9%

which were assessed by a traditional autograder and visible at the same time as the results from FEEDBOT. Thus, we expect that the additional help that students got from FEEDBOT was minimal.

One potential threat to validity in this analysis is that students who use FEEDBOT may be students who are stronger, and thus the assistance from FEEDBOT might be marginal. To try to address this, we used the aggregate performance on HW1-4, where FEEDBOT was not available (HW5 had partial availability, so was eliminated from both sections), to segment students into four quartiles of even size. While usage of FEEDBOT was consistently highest in the top quartile, the improvement of scores showed up across the spectrum, and was indeed often most notable in the lowest quartile, showing that FEEDBOT was beneficial even to lower performing students, provided they actually used it. These results are shown in Table 2. In that table, for each quartile (Quartile 1 is lowest performing, Quartile 4 highest) the average score from those that didn't use FEEDBOT and the average from those that did, and in parentheses the percent of that quartile that used FEEDBOT.

Another potential threat is that students in the lowest quartile may include those who simply did not submit one or more of the early homeworks. Because quartile membership is based on cumulative performance across HW1-4, a student who received zeros on one or two assignments but completed others could still land in Quartile 1 despite showing moderate performance on the assignments they completed. We opted not to exclude such students entirely because their scores and submission patterns are still pertinent to the question of who benefits from FEEDBOT usage. Nevertheless, we acknowledge that outliers of this nature may affect the quartile analyses, and in a more formal study, filtering or separate analyses (e.g., excluding students who submitted no work at all for early homeworks) could help isolate the effect of FEEDBOT from non-submission effects.

A final threat is that usage of FEEDBOT may be correlated to motivated students across all quartiles. Students who had weak early scores may have improved due to hard work unrelated to FEEDBOT. Unfortunately, our data does not allow us to challenge this assumption.

There is one other place we can look for quantitative feedback: debugging feedback built into the tool. In order to identify issues with FEEDBOT, we included a mechanism where users could rate any piece of feedback as "Very Helpful", "Somewhat Helpful", and "Not Helpful". A total of 1357 responses were collected (around 5.3% of the total possible). Of those, 63% chose "Very Helpful", 15% chose "Somewhat Helpful", and 22% chose "Not Helpful".

Table 2: Effect of FEEDBOT Usage by Performance In Class

HW	Quartile 1 Scores (Usage)	Quartile 2 Scores (Usage)	Quartile 3 Scores (Usage)	Quartile 4 Scores (Usage)
HW6	65.5% → 80.0% (14%)	82.5% → 89.5% (30%)	87.6% → 91.2% (40%)	93.0% → 94.5% (46%)
HW7	70.5% → 79.5% (24%)	80.5% → 82.5% (53%)	86.3% → 88.4% (60%)	84.5% → 93.0% (69%)
HW8	73.5% → 76.4% (18%)	82.9% → 86.2% (41%)	88.8% → 94.4% (57%)	92.4% → 95.3% (59%)
HW9	88.6% → 88.1% (16%)	93.8% → 95.3% (33%)	96.6% → 97.1% (50%)	97.0% → 97.5% (54%)
HW10	84.4% → 90.3% (27%)	87.6% → 92.0% (49%)	90.7% → 94.4% (62%)	93.4% → 96.3% (68%)
HW11	72.7% → 86.0% (16%)	84.5% → 84.6% (43%)	87.4% → 93.0% (45%)	91.3% → 94.7% (54%)
HW12	62.3% → 71.7% (22%)	74.5% → 82.7% (38%)	76.9% → 82.2% (44%)	84.3% → 86.6% (51%)

While 22% is notable, manual review of the "Not Helpful" ratings reveals nuance. In particular, we found that 37% of those "useless" responses were associated with errors in the student submission that FEEDBOT correctly identified. This suggests that a portion of the negative feedback was due to students' misunderstandings, rather than issues with the tool's functionality. This is, indeed, one possible risk in the design of FEEDBOT: since the feedback is very limited, and relies upon the student to identify *where* in the identified step there is a mistake, it is possible that students will not understand the feedback that is given. This is obviously compounded by the fact that occasionally the tool was, indeed, wrong! But the same can be true of nearly any intervention, including traditional teaching assistants!

An additional 4% of the "Not Helpful" ratings were due to a bug caused by a misconfiguration of dependencies, which had to be configured before assignment release (see §3.4). We corrected those once we noticed, but any submissions that had already been processed included erroneous complaints by FEEDBOT about references to unbound identifiers. Removing those 41% reduces the total number of unhelpful comments to around 13%.

While there could be similar errors in "Helpful" feedback, without being able to see what students changed in response to the feedback, it's hard to determine if the feedback was indeed erroneously helpful, so we do not attempt to quantify this.

A potential critique of this work is the lack of a direct comparison against a simpler intervention, such as merely pointing students to the Design Recipe and asking them to confirm for themselves that each step was completed. Indeed, that simpler approach characterized the first four homework assignments (HW1-4), where we consistently reminded students to verify the alignment with each step of the Design Recipe on their own. However, we believe that the structured and automatic nature of FEEDBOT's feedback (i.e., diagnosing which step is problematic) offers clearer guidance than self-diagnosis.

4.2 Qualitative Feedback

Around 30% of students gave feedback about the use of FEEDBOT alongside other general anonymous feedback for the course. Of those, 71% reported using FEEDBOT on most assignments, and over 90% used it at least once. Additionally, 83% of those that used it said the feedback was either very or sometimes helpful, and another 5% indicated that FEEDBOT always told them their code was fine, and they didn't get later deductions. This reinforces the notion that FEEDBOT has almost a 90% success rate.

In free response, many students complained about the rate limiting of the tool, which indirectly confirms utility. While the "cooldown" period was an additional mechanism to ensure that students still did independent work, the hard upper limit on the number of submissions was primarily for budgeting reasons. Both, however, were somewhat inflexible, as they did not allow the student to determine on which submissions they wanted feedback. We intend to change how this is done in the future, in order to have the rate limiting be enforced based on the submissions students view, rather than what they submit, which would alleviate many of the complaints (e.g., that by the time they had submitted something they wanted feedback on, they had used up all their submissions).

4.3 Future Work

Expanding Language / Pedagogic Framework Support: We plan on using FEEDBOT in other contexts, to see how tied to the Design Recipe its success is.
Conducting Studies: As an experience report, we clearly are only reporting on our experience with this tool. Among many potential things to study, it would be interesting to know the changes that happened after viewing the responses, and to code them to see if they were connected to FEEDBOT.
Better Rate Limiting: Not only allowing students to choose which submissions to get feedback on (see §3.6), but also exploring an adaptive feedback system that adjusts the frequency of feedback based on individual student performance to improve equity.
Adding Clarifications: While we don't want to turn FEEDBOT into a chatbot, adding some limited ability to ask for clarification on feedback may improve the cases of students not understanding correct feedback.

5 Conclusion

In this experience report, we have described a new open source tool called FEEDBOT that can be used to provide formative feedback on design-oriented programming assignments. It uses LLMs to provide structured, limited, but actionable feedback, addressing key challenges in large-class settings. We have shown that even in this restricted form, it can assist students.

Acknowledgments

We thank all the students and TAs who used FEEDBOT, and also Alessandra Simmons who, in addition to working on an early version, came up with the name "FEEDBOT ".

References

[1] David Baidoo-Anu and Leticia Owusu Ansah. 2023. Education in the Era of Generative Artificial Intelligence (AI): Understanding the Potential Benefits of ChatGPT in Promoting Teaching and Learning. *Journal of AI* 7, 1 (2023), 52–62.

[2] Alice Barana, Marina Marchisio Conte, et al. 2024. Promoting socioeconomic equity through automatic formative assessment. *Journal on Mathematics Education* 15, 1 (2024), 227–252.

[3] Patrick Bassner, Eduard Frankford, and Stephan Krusche. 2024. Iris: An AI-Driven Virtual Tutor for Computer Science Education. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 394–400. https://doi.org/10.1145/3649217.3653543

[4] Benjamin S. Bloom. 1968. Learning for Mastery. *Evaluation Comment* (1968), nil.

[5] Paul Denny, Stephen MacNeil, Jaromir Savelka, Leo Porter, and Andrew Luxton-Reilly. 2024. Desirable Characteristics for AI Teaching Assistants in Programming Education. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 408–414. https://doi.org/10.1145/3649217.3653574

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004. The Structure and Interpretation of the Computer Science Curriculum. *Journal of Functional Programming* 14, 4 (2004), 365–378. https://doi.org/10.1017/s0956796804005076

[7] Emily R. Fyfe, Nicole M. McNeil, Ji Y. Son, and Robert L. Goldstone. 2014. Concreteness Fading in Mathematics and Science Instruction: A Systematic Review. *Educational Psychology Review* 26, 1 (2014), 9–25. https://doi.org/10.1007/s10648-014-9249-3

[8] Dan Garcia, Armando Fox, Solomon Russell, Edwin Ambrosio, Neal Terrell, Mariana Silva, Matthew West, Craig Zilles, and Fuzail Shakir. 2023. A's for All (As Time and Interest Allow). In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. nil. https://doi.org/10.1145/3545945.3569847

[9] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. 2023. Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*. Association for Computing Machinery, 93–105. https://doi.org/10.1145/3568813.3600139

[10] Sven Jacobs and Steffen Jaschke. 2024. Evaluating the Application of Large Language Models to Generate Feedback in Programming Education. In *2024 IEEE Global Engineering Education Conference (EDUCON)*. 1–5. https://doi.org/10.1109/EDUCON60312.2024.10578838

[11] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 650, 20 pages. https://doi.org/10.1145/3613904.3642773

[12] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* 19, 1, Article 3 (Sept. 2018), 43 pages. https://doi.org/10.1145/3231711

[13] Mark Liffiton, Brad Sheese, Jaromir Savelka, and Paul Denny. 2023. CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes.

[14] Albert Lionelle, Sudipto Ghosh, Marcia Moraes, Tran Winick, and Lindsey Nielsen. 2023. A Flexible Formative/Summative Grading System for Large Courses. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. nil. https://doi.org/10.1145/3545945.3569810

[15] Yanyan Ren, Shriram Krishnamurthi, and Kathi Fisler. 2019. What Help Do Students Seek in TA Office Hours?. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. nil. https://doi.org/10.1145/3291279.3339418

[16] R. Benjamin Shapiro, Kayla DesPortes, and Betsy DiSalvo. 2023. Improving Computing Education Research through Valuing Design. *Commun. ACM* 66, 8 (July 2023), 24–26. https://doi.org/10.1145/3604633

[17] Brad Sheese, Mark Liffiton, Jaromir Savelka, and Paul Denny. 2024. Patterns of Student Help-Seeking When Using a Large Language Model-Powered Programming Assistant. In *Proceedings of the 26th Australasian Computing Education Conference* (Sydney, NSW, Australia) *(ACE '24)*. Association for Computing Machinery, New York, NY, USA, 49–57. https://doi.org/10.1145/3636243.3636249

[18] Ella Tuson and Timothy Hickey. 2023. Mastery Learning with Specs Grading for Programming Courses. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. nil. https://doi.org/10.1145/3545945.3569853