2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

LitterBox^{+}: An Extensible Framework for LLM-enhanced Scratch Static Code Analysis

Benedikt Fein
University of Passau
Passau, Germany

Florian Obermüller
University of Passau
Passau, Germany

Gordon Fraser
University of Passau
Passau, Germany

***Abstract*—Large language models (LLMs) have become an essential tool to support developers using traditional text-based programming languages, but the graphical notation of the block-based *Scratch* programming environment inhibits the use of LLMs. To overcome this limitation, we propose the *LitterBox*$^{+}$ framework that extends the *Scratch* static code analysis tool *LitterBox* with the generative abilities of LLMs. By converting block-based code to a textual representation suitable for LLMs, *LitterBox*$^{+}$ allows users to query LLMs about their programs, about quality issues reported by *LitterBox*, and it allows generating code fixes. Besides offering a programmatic API for these functionalities, *LitterBox*$^{+}$ also extends the *Scratch* user interface to make these functionalities available directly in the environment familiar to learners. The framework is designed to be easily extensible with other prompts, LLM providers, and new features combining the program analysis capabilities of *LitterBox* with the generative features of LLMs. We provide a screencast demonstrating the tool at https://youtu.be/RZ6E0xgrIgQ.**

***Index Terms*—Scratch, Block-based Programming, LLM, Automated Feedback**

I. Introduction

Scratch [1] is a popular block-based programming environment frequently used to introduce beginners to programming [2]. While the block-based nature simplifies coding, beginners often lack a deep understanding of required concepts and introduce issues into their code [3]. LLMs have recently become popular to help programmers of text-based languages to fix and improve their code, but the graphical nature of block-based code so far hampered the application of LLMs to *Scratch*.

To bridge the gap between *Scratch* programs and the promises of LLM support, we introduce the *LitterBox*$^{+}$ framework: It uses the *LitterBox* static code analysis framework [4] to convert *Scratch* code to a textual representation suitable for LLMs, and provides a convenient API to query LLMs about *Scratch* programs. The framework is designed to be easily extensible with different types of prompts and queries, as well as LLM providers. It automatically provides the context for a given query using the target code in question, offers convenient access to common types of queries, and can build on the existing linting features of *LitterBox* to allow users to gain further insights and even fix suggestions for the otherwise generic issue descriptions delivered by the static analysis. In addition, *LitterBox*$^{+}$ extends the *Scratch* user interface to enable learners to directly make use of these features (cf. Fig. 1). This includes a custom parser for LLM responses such that suggested fixes can be directly applied to *Scratch* projects.

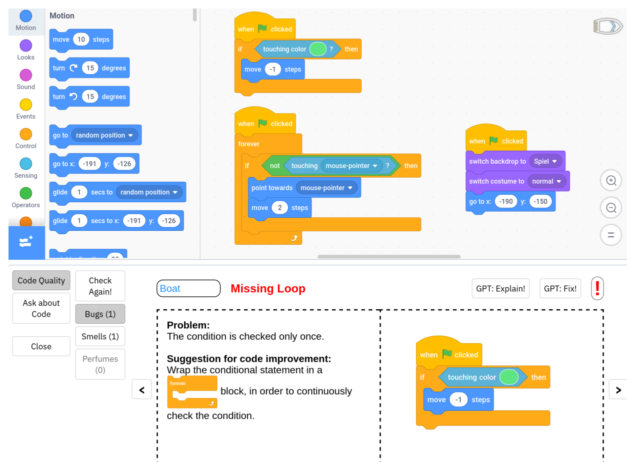


Figure 1: A buggy program displayed in the extended *Scratch*-GUI, together with LLM controls to explore the bug.

LitterBox^{+} is also intended to be an extensible framework upon which researchers can build creative new *Scratch*+LLM integrations, tapping into both the potential of LLMs as well as the established static analysis features of *LitterBox*. To support both users and researchers alike, *LitterBox*$^{+}$ with its command-line and Java APIs as well as the extended *Scratch* user interface are available as open-source code. The extension to *LitterBox* is available at https://github.com/se2p/litterbox and our extensions to the *Scratch* user interface can be found at https://github.com/se2p/NuzzleBug.

II. Background

The different shapes of *Scratch* blocks determine how they can be combined into programs, thus simplifying programming. However, learners may still struggle to correctly apply programming concepts and produce buggy code. While the process of finding and fixing bugs is an inherent part of learning to program, automated tools can offer support. Since *Scratch* code is block-based, dedicated analysis tools are required. For example, *LitterBox* [4] is a linter that automatically detects bugs based on recurring patterns that are common manifestations of misconceptions. Figure 1 shows such a bug pattern in the popular ‘BoatRace’ program[1] where a misconception caused a

[1] https://scratch.mit.edu/projects/63957956/, accessed 2025-07-21

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00357

'Missing Loop' bug pattern, resulting in a collision detection only being executed once at the start of program execution. Such bug patterns are common in *Scratch* [3], and feedback on them has been shown to support learners [5]. However, classic static analysis tools like *LitterBox* can only provide generic help, and learners have to rely on other sources for clarifications or suggestions on how to fix their code.

While LLMs are capable of providing exactly this help for text-based languages, their application to *Scratch* code is hampered by the graphical block-based notation. Consequently, existing work on using LLMs for *Scratch* tends to focus more on creative aspects, like for example using the LLM chat to collect ideas for program features or using the model to generate sprite images [6], [7]. These existing approaches integrate less targeted context, e.g., generic *Scratch* code patterns from other programs [6], when students ask for support in the LLM chat [6], [7]. In contrast, in this paper we focus on providing specific context to the LLM in the form of the program's code, as well as detected issues therein to help the user to solve these issues.

Compared to prior work, we also support more extensive LLM-based modifications of existing programs. In particular, we do not rely on heuristic text matching semantics to collect *Scratch* blocks from the LLM's output [6], but instead developed a parser for the widely used textual *scratchblocks*[2] *Scratch* code representation as part of *LitterBox*+, which allows us for example to merge fixed scripts returned by the LLM into the abstract syntax tree of the surrounding program.

III. User Interfaces

The main user interface is integrated directly into the *Scratch* web interface. This direct integration ensures a low barrier to entry by not requiring context switches away from the familiar programming environment. We added a toggleable panel that shows by default issues found by the regular *LitterBox* static code analysis in the current program (cf. Fig. 1). For each issue, the user can request an extended LLM-generated explanation, or request a fix for the issue, which will be automatically integrated into the active program. Additionally, our GUI extension offers a chat-like input where a user can ask arbitrary questions about the current sprite or program.

While early trials of our tool showed that LLMs can understand and reason about *Scratch* code, the model may still occasionally produce misleading or wrong explanations and fixes. Since especially learners often cannot discern such cases from helpful ones, we show a prominent hint about the potentially wrong nature in every location where LLM-generated content is presented. For example, the red exclamation mark in Fig. 1 shows this hint as tooltip when hovered over. Even though our tool is LLM-agnostic, we call all LLMs '*GPT*' in the user interface, since students are likely more familiar with *GPT* as a general term for all kinds of LLMs.

By requesting a different output language in the prompts, *LitterBox*+ can be customised to support any natural language. While the *scratchblocks* code in the prompts remains English,

[2]https://en.scratch-wiki.info/wiki/Block_Plugin/Syntax, accessed 2025-07-22

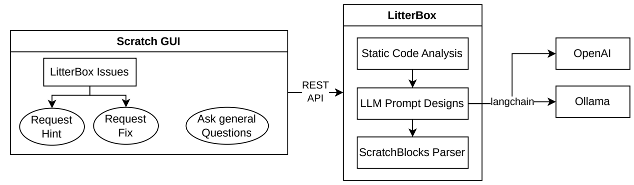


Figure 2: *LitterBox*+ architecture overview.

the *Scratch*-GUI can still automatically display the code in the editor area in the user's chosen UI language.

For teachers or other more experienced users, we also make all functionality of our *LitterBox*+ extension available via the command-line interface (CLI). Like regular *LitterBox*, this for example additionally allows for batch-processing of multiple programs or automatically downloading and processing specific programs from the official *Scratch* website. The extended *LitterBox* can be started via the CLI like a regular Java program:

```
java -Dlitterbox.llm.openai.api-key=KEY
    -jar LitterBox.jar llm --help
```

For example, to ask the LLM to analyse the 'Boat' sprite of the program and report new issues missed by *LitterBox*:

```
[...].jar llm analyze --path=program.sb3
    --target=Boat --new-issues
```

All subcommands of the CLI accept the `--help` option which explains the available subcommands and their options.

Finally, to allow future research to extend upon our framework, all *LitterBox* functionality is available via its Java API so that it can also be integrated as a library into other tools. This API allows for future extension in various directions: (1) Our prompt designs can be replaced by customised prompt templates. (2) While the user can already configure *LitterBox*+ to use either externally hosted (*OpenAI*) or self-hosted (*Ollama*) models, additional model providers could be added.[3] (3) New LLM-based features that make use of the existing static code analysers and the *scratchblocks* parser, or ones based on entirely new analysers could be introduced.

For example, to change the default prompts, only a single 'prompt provider' class has to be overridden. It can then be selected as the default prompt provider by setting a system property when starting the *LitterBox* application. Using a similar selection mechanism, another LLM provider can be supported by adding a new concrete implementation of our generic LLM-provider interface to *LitterBox*. The same system properties are used by both *LitterBox* itself (i.e., when using it as a command-line tool) and by our REST-API extension (i.e., when using it as a backend for the *Scratch*-GUI).

IV. System Architecture

To support the communication between *LitterBox* running as a Java application on a server and the website in the browser, we extend *LitterBox* with a REST-API. When we receive a request relating to an issue found by *LitterBox* from the user interface,

[3]Our implementation uses *langchain4j*, https://docs.langchain4j.dev/, accessed 2025-07-14.

we combine information from the issue itself and the program to construct a prompt for the LLM. From the issue itself, we extract its type (e.g., Missing Loop) and the default description added by *LitterBox*. To embed the issue in the relevant context, we also extract the source code of the *Scratch* program. To provide this code to the LLM, we convert the program into the textual *scratchblocks* format. We use the same format as is used for example on the *Scratch* community forums and only extend it slightly to include special comments adding unique IDs to the start of sprites and scripts. The *scratchblocks* format was likely seen by the LLM during training as it is common on forums and uses a sequential structure for code, unlike *Scratch*'s internal JSON program representation, thus resulting in improved performance. Unlike taking screenshots of the workspace, this removes the visual component from the code. This avoids the problem that code blocks are allowed to overlap in the *Scratch* coding area; by using the textual format instead of a screenshot of the code, we therefore have more control over which part of the code is included in our prompt.

Depending on the task, we prompt the LLM to respond either with an explanation suitable for beginner programmers or respond with a code snippet in *scratchblocks* format (e.g., when asking for a fix for an issue). To integrate such a code response into the existing program, we add a parser for the *scratchblocks* format to *LitterBox*. In case the parsing of a script fails, we extract the script from the response and ask the LLM to fix the syntax. If the LLM fails to repair the *scratchblocks* code, the user is informed that the LLM returned something unuseful. Since the LLM is instructed to retain our special ID-comments for modified scripts, we can correlate such code snippets back to the full original program and update the code there accordingly.

Depending on the request, we finally send the response as text, an updated issue, or the program with the LLM-proposed fix back to the browser where it can be displayed to the user.

V. Features

This section describes the three main features available via the *Scratch*-GUI and other features only available in the CLI.

A. Questions About the Program

Users might have general questions about the behaviour of their program, or they might want to find ideas on how to continue. For this use-case, our framework allows users to ask arbitrary questions about the program in the *Scratch* user interface. Even though we do not use the static code analysis features in this step, *LitterBox* is still used to convert the program code into the *scratchblocks* format. We then combine the user question with this code as relevant context to create the overall prompt for the LLM. Depending on the user choice whether the question is specific only to the currently selected sprite or more general about the whole program, only the relevant code context is included in the query to the LLM.

B. Explanation for LitterBox Issues

By default, our extension shows the *LitterBox* bugs and smells as generated by the static code analysis in the user interface. Their description is mostly generic depending on the bug type, with only some customisation placeholders that are filled for example with the relevant variable names. These generic hints might make it harder for students to come up with reproduction steps or possible fixes since they need to mentally map the generic information to their specific program first. The generic description lacks instructions on how to reproduce the issue, since such steps are always program-specific.

To obtain a more detailed hint, we add a 'GPT: Explain!' button to the user interface (cf. Fig. 1). This sends the issue context together with the program to *LitterBox*. The prompt includes the original issue explanation and asks the LLM to (1) explain how the faulty code affects the observable behaviour of sprites and (2) to generate a list of steps required to observe this faulty behaviour.

Since the response is an extended explanation rather than a modified program, it is enough to append the LLM's explanation to the original issue description and send the updated issue back to the browser for display as addition to the previous description (cf. left half of the issue display in Fig. 1). In this example, *GPT 4.1*[4] correctly explains the issue as follows:

1. How the faulty code affects observable behavior of sprites:

The static code analysis tool pointed out that a certain condition is only checked once, instead of being checked continuously.

(a) This script will only check *once*, immediately after the green flag is clicked, if the sprite is touching the color .

(b) If at that exact moment the sprite is NOT touching this color, nothing will happen, *even if the sprite touches it later*.

(c) As a result, the response (move -1 steps) won't trigger when the sprite later comes into contact with the specified color, making the code seem unresponsive.

(d) *Observable Effect:* The Boat sprite will NOT react (move -1 steps) when it touches the color at any point after the game starts, unless it happens to be touching it *exactly* as the green flag is clicked.

2. List of steps to observe the faulty behavior:

1. *Press the green flag* to start the program.

2. *Move the Boat sprite* (using code or manually, depending on your setup) so that it touches the color *after* the game has started.

3. *Observe:* The Boat sprite does *not* move backwards (-1 steps), despite touching the color.

4. *(Optional)* Restart and have the sprite start on top of color. Now, you might see it move -1 steps at game start, since the condition was true at that moment.

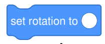
C. Automatic Issue Fixing

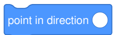
To extend the support beyond providing a more specific description of the issue, our framework can also propose fixed versions of the code and integrate them directly into the current program. Similar to the explanation, we added a 'GPT: Fix!' button into the user interface next to the shown *LitterBox* issue.

When requesting the LLM to propose a suitable fix, we again prompt it with the current issue description and the

[4]https://openai.com/index/gpt-4-1/, accessed 2025-07-21

relevant code context in *scratchblocks* format. As opposed to the previously described features, the LLM now has to produce code in its output rather than only a natural language description. Like for the LLM input, we rely on the *scratchblocks* format and extend *LitterBox* with a suitable parser. Our parser can process all standard *Scratch* blocks, custom block definitions, and blocks of the *Scratch* ‘Pen’ extension. In the future, its grammar could be extended to support further *Scratch* extensions.

While we observed that the LLMs can in most cases produce valid *scratchblocks* code, they still often repeat similar issues, like for example using a non-existing block 

instead of the actual 

or using braces to scope inner blocks of control statements instead of using the correct `then` and `end` markers. Sometimes the LLM also includes both the buggy and the fixed scripts in its output with the script marker comments (cf. Section IV) modified with an additional ‘original version’ or ‘modified version’ suffix, even though our prompts explicitly request the LLM to not modify the special comments. To mitigate such frequent problems, we introduced a postprocessing step to the LLM’s output that applies heuristic rules fixing common syntactical issues before passing the *scratchblocks* code to the actual parser. In case this parsing still fails, we prompt the LLM with only the unparseable scripts again in up to three attempts to request a syntactically correct script instead. If the code continues to be incorrect afterwards, we remove it and do not process it further.

To integrate the parsed scripts into existing programs, we rely on the special comments marking the beginning of sprites and scripts (cf. Section IV). Using the sprite names and block IDs contained in these comments, we can align the parsed code snippets with the original code and make the required replacements. These replacements are performed on the internal program representation of *LitterBox* (i.e., the abstract syntax tree), which is substantially more flexible and reliable compared to working directly on the textual *scratchblocks*- or JSON-formats. Any sprites or scripts with unseen IDs are added as new code to the program. When adding a new sprite, we use the default image of the *Scratch* cat mascot.The updated program is finally converted by *LitterBox* back into the *Scratch*-native JSON format and loaded into the *Scratch*-GUI where the user can inspect it in the native code editor (upper half of Fig. 1) and continue to edit it. In case the proposed fix is not satisfactory, the user can choose to revert the change.

D. Other Features

We also implemented additional features that are currently only available via the CLI and the Java API.

1) New Issue Finder: The user can prompt the LLM to find additional issues not detected by the *LitterBox* heuristics in the program. This can also be used to find code ‘perfumes’ [8], i.e., parts of the code following correct programming practices.

2) Code Completion: The user can select a specific script of the program by providing its ID and request the LLM to extend it with suitable code. The prompt automatically adds the surrounding sprite code as related context to guide the LLM to provide useful code that does not duplicate existing parts.

VI. Future Work

Since we introduce an initial version of our extensible *LitterBox*+ framework in this paper, this leaves many opportunities for future work. Our framework could be extended to other visual languages (e.g., *Snap!*), but suitable linters and code converters from/into text need to be developed first. For use in a classroom setting where the target solution of the program is known, additional agentic capabilities could be integrated that automatically run *Whisker* [9] or block-based [10] test suites so that either LLM-proposed code analysis issue fixes can be checked and improved automatically via a follow-up prompt to the LLM, or that fixes for failing tests can be requested directly by the user. Since the *Scratch*-GUI is lacking any kind of code completion feature at the moment, our LLM-based completion could be a valuable further extension to the framework and its users. Finally, all features proposed as part of *LitterBox*+ should be evaluated with children to understand how such LLM-based tools are used [6], [7] to then adjust prompts and user interface, and develop new features based on the findings.

VII. Conclusions

In this paper we introduced our *LitterBox*+ framework aiming to simplify the integration of LLMs into *Scratch* programming tasks. By providing this framework as open-source, we encourage the development of new *Scratch*+LLM integrations going beyond the features presented in this paper.

Acknowledgments

We would like to thank Patric Feldmeier for his contributions to the *scratchblocks* parser implementation. This work is supported by DFG project FR 2955/3-3 ‘Pedagogy and Technology for Feedback in Block-Based Programming’ and FR 2955/5-1 ‘TYPES4STRINGS: Types For Strings’.

References

[1] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, “The Scratch programming language and environment,” *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, Nov. 2010.

[2] M. M. McGill and A. Decker, “Tools, languages, and environments used in primary and secondary computing education,” in *Conf. on Innovation and Technology in Computer Science Education*. ACM, 2020.

[3] C. Frädrich, F. Obermüller, N. Körber, U. Heuer, and G. Fraser, “Common bugs in Scratch programs,” in *Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 2020.

[4] G. Fraser, U. Heuer, N. Körber, F. Obermüller, and E. Wasmeier, “Litterbox: A linter for Scratch programs,” in *ICSE-SEET*. IEEE, 2021.

[5] L. Greifenstein, F. Obermüller, E. Wasmeier, U. Heuer, and G. Fraser, “Effects of hints on debugging Scratch programs: An empirical study with primary school teachers in training,” in *WiPSCE*. ACM, Oct. 2021.

[6] L. Chen, S. Xiao, Y. Chen, Y. Song, R. Wu, and L. Sun, “ChatScratch: An AI-augmented system toward autonomous visual programming learning for children aged 6-12,” in *CHI Conference on Human Factors in Computing Systems*. ACM, May 2024.

[7] S. Druga and A. J. Ko, “Scratch Copilot: Supporting youth creative coding with AI,” in *Interaction Design and Children*. ACM, Jun. 2025.

[8] F. Obermüller, L. Bloch, L. Greifenstein, U. Heuer, and G. Fraser, “Code perfumes: Reporting good code to encourage learners,” in *WiPSCE*. ACM, Oct. 2021.

[9] A. Deiner, P. Feldmeier, G. Fraser, S. Schweikl, and W. Wang, “Automated test generation for Scratch programs,” *Empirical Software Engineering*, vol. 28, no. 3, May 2023.

[10] P. Feldmeier, G. Fraser, U. Heuer, F. Obermüller, and S. Steckenbiller, “A block-based testing framework for Scratch,” in *Koli Calling*. ACM, Nov. 2024.