

Reasoning About Bugs in Learners' Scratch Programs Using Large Language Models

Benedikt Fein*

University of Passau
Passau, Germany

benedikt.fein@uni-passau.de

Gordon Fraser

University of Passau
Passau, Germany

gordon.fraser@uni-passau.de

Patric Feldmeier

University of Passau
Passau, Germany

patric.feldmeier@uni-passau.de

Florian Obermüller

University of Passau
Passau, Germany

florian.obermueller@uni-passau.de

Abstract

Program analysis tools for the block-based programming environment *Scratch* support learners by identifying blocks indicative of bugs, which may be evidence of misconceptions and likely lead to faulty program behaviour. However, this information may not be sufficient to guide learners towards fixing both their misunderstandings and their programs. In this paper we therefore propose to use large language models (LLMs) as a means to explain the issues found by program analyses, and to help learners understand how to debug and fix these issues. While the power of modern LLMs has resulted in a surge of new support tools for text-based programming languages, *Scratch* programs pose additional challenges for LLMs: The visual, block-based code first needs to be converted to textual prompts for the LLMs, and even then is it unclear whether the LLMs can exhibit similar performance as for text-based programming languages with their abundant training data. We determine the best way of generating prompts from *Scratch* programs and their bugs, and use these prompts with a dataset of buggy *Scratch* programs to evaluate the ability of LLMs to reason about these bugs. Our experiments demonstrate that LLMs can reason well about bugs in *Scratch* programs, thus setting the scene for more powerful tools that assist young learners in debugging their programs.

CCS Concepts

• **Social and professional topics** → K-12 education; Software engineering education; • **Software and its engineering** → Visual languages.

Keywords

Scratch, Block-based Programming, LLM, Automated Feedback

ACM Reference Format:

Benedikt Fein, Patric Feldmeier, Gordon Fraser, and Florian Obermüller. 2026. Reasoning About Bugs in Learners' Scratch Programs Using Large Language Models. In 2026 *IEEE/ACM 48th International Conference on Software*

*Authors listed alphabetically.



This work is licensed under a Creative Commons Attribution 4.0 International License. *ICSE-SEET '26, Rio de Janeiro, Brazil*

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2423-7/2026/04

<https://doi.org/10.1145/3786580.3786949>

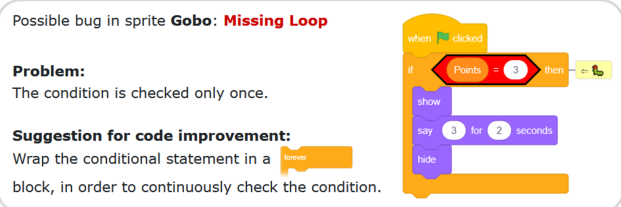


Figure 1: *Missing Loop* pattern inferred and explained by the *LitterBox* static code analysis tool.

Engineering (ICSE-SEET '26), April 12–18, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3786580.3786949>

1 Introduction

Block-based programming languages such as *Scratch* are often used as an entry point to programming education [24]. Even though they can reduce complexity, especially young learners require assistance due to their often limited prior experience [14]. However, providing this assistance may quickly overwhelm teachers because they (1) lack time available per student [14, 31], (2) need to deal with a variety of student problems (especially for open-ended programming tasks) [31], and (3) often lack subject knowledge themselves [14].

To improve the support provided to learners, code analysis tools can automatically identify and report issues such as code smells or common bug patterns [11]. Tools such as *LitterBox* [10] are easy to use, they require no context information and any program can simply be uploaded and analysed. This convenience, however, is mainly due to their limited output: They flag individual bits of code and provide generic descriptions of the underlying patterns. Learners then need to reason about their bugs in order to correct their programs and misunderstandings. For example, they need to understand the relevance of the incorrect code, how it affects their specific program, how the underlying programming construct is used correctly, and ultimately how to fix the program. Code analysis tools rarely provide guidance on these aspects.

Figure 1 exemplifies this gap between what code analysis tools can provide and what learners may need: The figure shows a hint generated by the static code analysis tool *LitterBox* for the buggy *Scratch* program in which the player collects points, but unlike intended, the sprite 'Gobo' does not react if three points are reached.

While *LitterBox* provides an explanation that the bug consists of the use of a conditional statement outside a loop construct, it does not give any contextual information: It does not tell the programmer whether and how the bug affects the program’s behaviour, leaving it unclear how resulting problems can be observed and debugged. While a generic hint on adding a loop-block is given, it is unclear whether this is sufficient to remove a possible misconception on if-statements among students.

Recently, large language models (LLMs) have demonstrated astonishing ability to reason about code, and they can now be found in modern programming environments, for example to explain or synthesise code. This ability provides an opportunity to bridge the gap between what code analysis tools can provide and what learners actually need: Given an issue detected by a code analysis tool, LLMs could explain how that issue changes the program behaviour, how that change can be observed, what the learner may have misunderstood, and how to fix the issue. For our example (Fig. 1), *OpenAI*’s GPT-4 model helpfully explains: ‘*This means that if the player has not scored three points within that very short time frame, Gobo will not check the score again and therefore will not appear and say the message even if the player later reaches 3 points.*’

However, while LLMs can deal well with textual programming languages, it is unclear how well they perform with visual, block-based programming languages like *Scratch*. On the one hand, this is an issue of training: There simply is so much more textual code available on the internet for LLMs to be trained on. On the other hand, this is also an issue of representation: How does one provide visual code as input to an LLM? These are important concerns, since a hallucinating LLM would confuse rather than support learners.

In this paper, we investigate whether LLMs are able to reason about bugs in *Scratch* code, such that future research can integrate them into code analysis tools or interactive tutorial systems. We use 24 *Scratch* programs containing different bugs reported by the static analysis tool *LitterBox*, and compare different methods to encode these programs and *LitterBox*’s output into prompts for *OpenAI*’s GPT-4o-mini model and a locally hosted Gemma 3 instance. To choose the best representation we compare the rate of detecting false-positive warnings and the ability to produce fixes for actual warnings. Using the best representation, we compare the base prompt with prompts using (1) chain-of-thought and (2) few-shot learning. Selecting the best prompt we then investigate how well the LLMs can reason about bugs in *Scratch* in terms of (1) describing the faulty program behaviour observable by the user, (2) explaining the underlying programming concept that may be misunderstood and (3) naming possible user inputs to the program that replicate the faulty program behaviour. Our key contributions are:

- Determining the best way to represent a *Scratch* program in a textual prompt for LLMs.
- Determining the best prompt strategy for detecting bug patterns and receiving fixes.
- Providing a way to automatically apply fix suggestions made by the LLMs.

The best results are achieved using GPT-4o-mini with few-shot learning for fixing bugs (22/24) and GPT-o4-mini with chain-of-thought for detecting false-positive warnings (24/24). GPT-4o-mini and Gemma 3 demonstrate that they can reason about bugs in

Scratch programs and excel in describing which bug-causing programming concept the learner misunderstood for at least 21/24 programs. This opens up new possibilities for producing improved learning environments that provide better support to learners.

2 Background

We propose combining *Scratch* analysis tools with LLMs to assist learners in programming with *Scratch*.

Scratch. *Scratch* [22] aims to make programming more approachable for novices by emphasising recognition over recall, distinguishing different categories of blocks by colour, featuring intuitive interactivity via an event-driven paradigm and offering simple media integration. Syntactical errors are prevented by shapes determining how blocks can be assembled. Whereas programs in regular programming languages are simply represented as text files, *Scratch* programs are zip-archives containing image and sound files, as well as a JSON encoding of the visual code. While JSON is a textual format, these JSON files are not human-readable: They represent blocks using collections of attributes such as opcodes and unique IDs, and rather than reflecting the sequential nature of code they may occur in any order, linked implicitly through their IDs.

Code Analysis Tools. There are different types of analysis tools, but in this paper we focus on *linters* due to their widespread use in programming education [17, 21, 25, 29]. Linters for *Scratch* (e.g., *LitterBox* [10]) can identify *code smells* [2, 30] (i.e., code idioms that increase the likelihood of bugs during future changes) and *bug patterns* [11] (i.e., aspects of code that are likely to lead to undesired behaviour). *LitterBox* does so by statically generating an abstract syntax tree (AST) from the JSON file in a *Scratch* program and looking for the block combinations that define code smells and bug patterns utilising a visitor pattern on the AST. Since *LitterBox* is a static linter, it does not have information about what functionalities a program should have or how a bug pattern occurrence influences the execution of a program. However, it can at least generate feedback explaining a bug pattern in the local context of the program code, such as in Fig. 1. This corrective feedback is intended to help novices fix bugs [13, 15]. To represent the code addressed in the feedback, *LitterBox* can convert any part of the AST representation into *scratchblocks*¹ notation, which is a commonly used textual representation on *Scratch* discussion boards on the internet.

Large Language Models (LLMs). Large language models are huge neural networks that were trained on vast amounts of data to predict the expected answer to a given prompt [9], where both prompt and response are interpreted as chunks of letters (tokens). These models can be used to support various coding tasks in mature programming languages, including refining the results of static analysis tools [20]. In computer science education, recent work shows the ability of LLMs to create programming exercises [28], to answer student questions [18], to explain introductory programming code examples [19], or to assist with auto-completion [16]. However, to the best of our knowledge all prior work focuses on text-based programming languages, except for a pre-print and its follow-up user study indicating promising performance of *OpenAI*’s GPT-4

¹<https://github.com/scratchblocks/scratchblocks>, accessed May 2025

LLM when explaining *Scratch* scripts, or suggesting how to extend or debug them [4, 5]. The best way to encode *Scratch* programs in textual LLM-prompts and how to use LLMs to reason about the bugs in these programs remains unclear.

3 Reasoning about Bugs using LLMs

Figure 2 describes a proposed framework for combining LLMs with *Scratch* analysis tools to obtain context-specific bug explanations. Although we will refer to the *LitterBox* analysis tool in the following, the framework generally applies to any tool capable of locating bugs in *Scratch* programs. The input is a *Scratch* program in its JSON representation, which contains all the information describing the code of *Scratch* programs, such as a list of sprites with their scripts and blocks. The JSON file is then sent to *LitterBox* to be checked for bugs. If a bug is found, *LitterBox* generates a JSON file containing a description of the bug, its location in terms of the responsible block's ID, and a general hint on how to fix bugs of this type. The report can be rendered by a browser as in Fig. 1.

We infer an LLM's basic reasoning ability by its ability to detect false-positive *LitterBox* warnings given the bug description and the relevant code snippet. In order to allow LLMs to process *Scratch* code, the JSON file describing the code has to be converted to text, for which there are multiple options, amongst them the original JSON or the *scratchblocks* notation.

The overall prompt for the LLM is generated by combining the bug description, the converted program and the question for the respective bug into a single prompt as shown in Fig. 3a. The *LitterBox* output can be localised in different languages, and the framework can therefore also be configured to produce prompts in any language, eliciting explanations in that language. Although *Scratch* code can be localised in the user interface, the internal representation using opcodes implies that the framework always produces English language textual representation of code except for user-defined strings such as sprite- and variable names, messages, and string constants which remain as-is.

Providing a suitable fix for a bug detected by *LitterBox* requires a deeper level of understanding of the program structure. For this, the LLM needs to produce either (1) a valid JSON file representing the fixed *Scratch* program or (2) valid code in *scratchblocks* syntax (cf. prompt in Fig. 3b) that then needs to be converted to JSON. Since *LitterBox* can already parse and analyse *Scratch* JSON files, we only need to extend *LitterBox* to automatically handle the second case.

As Fig. 2 shows, we extended *LitterBox* to parse the *scratchblocks* code and integrate it back into the original program, and then run the static code analysis again to check whether the previous issue was fixed. When integrating the scripts in *scratchblocks* format returned by the LLM, the new scripts need to be matched to their counterpart in the existing *Scratch* code. For this, we prefix each script with a comment containing the *script_id* in the prompt. Since the LLM retains the prefixed comments even when generating modified code and generates full scripts, we completely replace the old version of the script by its presumably fixed version. Variables and messages passed between scripts are not defined locally per script in *Scratch*, but are part of global program metadata. Therefore, we do not require the LLM to specifically define any new variables or messages it introduces. Instead, we resolve unknown references by

adding the required definitions in the integration step in *LitterBox*. The initialisation of messages only requires their name as known from the parsed code. We assume a default value of zero for regular variables and an empty list for lists. Custom blocks are declared within the sprite similar to regular scripts but with a special header block. If the LLM defines a new custom block, we use this definition during the integration. If the code calls an unknown custom block without defining it, we do not need to create a definition since *Scratch* ignores such blocks during execution. Further details of our *LitterBox* extension can be found in a dedicated tool paper [7].

4 Evaluation

In the first three of the following research questions, we deduce the best prompt design to ask LLMs about *Scratch* programs. We then study the ability of LLMs to reason about *Scratch* bugs in the fourth.

- RQ1** Which textual representation of *Scratch* program performs best when querying the LLM?
- RQ2** Does chain of thought prompting improve the LLM's ability to detect and fix valid static analysis issues?
- RQ3** Does few-shot prompting improve the LLM's ability to detect and fix valid static analysis issues?
- RQ4** Can LLMs reason about *Scratch* programs?

4.1 Dataset

To answer our research questions we require a dataset of buggy programs with corresponding fixes. While deriving such datasets for textual programming languages can be done by mining abundantly available open-source code and bugfixes from *GitHub*, this is not possible for *Scratch*: Although projects can be publicly shared on the *Scratch* website, there is neither a change history nor is there issue tracking. The dataset for our experiments is therefore derived from two sources: The first half is taken from a prior study on debugging [15], where each program contains an instance of one of the most common bug patterns in *Scratch* [11]: *Comparing Literals*, *Forever Inside Loop*, *Message Never Received*, *Message Never Sent*, *Missing Clone Initialisation*, *Missing Loop*. We omit *Stuttering Movement* despite it being used before [15], as it is no longer considered a bug pattern in more recent versions of *LitterBox*.² For each bug the dataset contains two buggy programs together with their fixed, bug-free versions. The programs are of typical sizes for *Scratch* courses: There are 3–13 scripts, 2–6 sprites, and the weighted method count ranges from 8 to 23. In addition, each program is accompanied by a clear description of the program's actual and desired output as well as a *Whisker* [3] test suite checking whether the specific *LitterBox* issue found in the buggy program variants is fixed and that the program behaviour does not differ from the example solution. These allow us to systematically cross-check the LLM's answers.

We additionally sampled programs shared publicly on the *Scratch* website containing the same bug patterns. From a dataset of about 8 million mined programs we considered all programs that contain exactly one of the above bug types, resulting in 117 765 programs. From these, we randomly chose two programs for each of the above six bug patterns and also implemented a fixed version for each program. The dataset size is limited by the required manual fix

²https://github.com/se2p/LitterBox/tree/v1.9.2/src/main/java/de/uni_passau/fim/se2/litterbox/analytics/smells, version 1.9.2, accessed September 2025

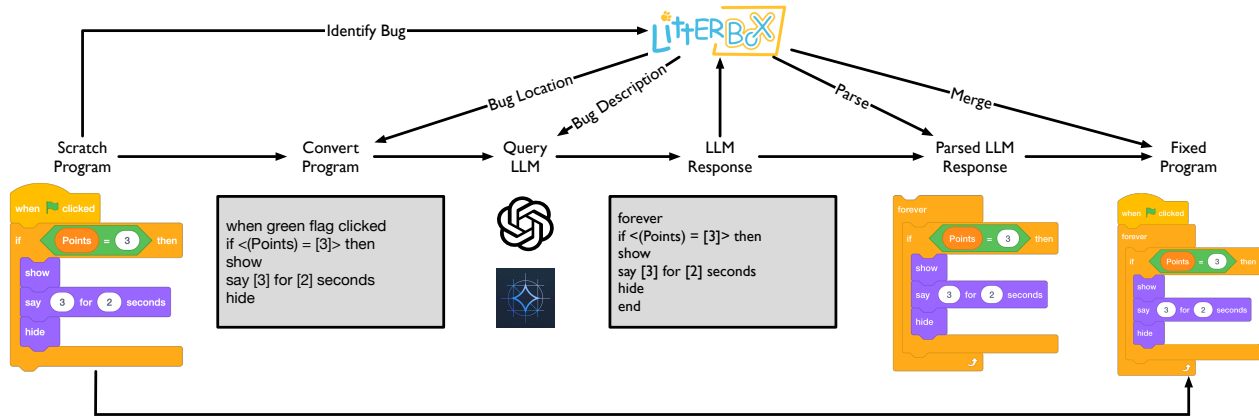


Figure 2: Overview of the bug reasoning and fixing framework using *LitterBox* as a paradigmatic *Scratch* analysis tool.

You are given the following Scratch program: {program}
 A static code analysis tool identified an issue in sprite {sprite_name}, script {script_id}, and provides the following explanation: {explanation}
 Does the given program actually contain the issue described by the static code analysis tool?
 Answer the question with @@ yes @@ or @@ no @@.

(a) Detection of false-positive static code analysis warnings.

You are given the following Scratch program: {program}
 The code contains the following bugs and code smells: {explanation}
 Create a version of the program where these issues are fixed.
 {{ if scratchblocks_format }}
 Only output the ScratchBlocks code and nothing else. Include sprite and script ids in the ScratchBlocks code.
 {{ else if json_format }}
 Only output a JSON in the same format as the Scratch program above and nothing else.

(b) Requesting a fix for a static code analysis warning.

Figure 3: Base prompt templates.

implementation for each of these buggy programs, as well as the successive manual analysis of LLM responses. The sampled programs have a weighted method count of 3–33 and contain up to 22 scripts that are distributed across 1–7 sprites. In total, the combined dataset thus consists of 24 pairs of buggy and fixed programs.

4.2 Methodology

For our experiments, we prompted GPT-4o-mini (2024-07-18) and GPT-o4-mini (2025-04-16) [27] via their *OpenAI* API and a locally hosted instance of Gemma 3 [12] as available on *ollama*.³ For Gemma 3 we used a context window of up to 16 384 tokens. To account for randomness in the outputs, we repeated each query 5 times and use the most frequently returned answer by extracting the yes/no verdict enclosed in ‘@@’ signs (cf. Fig. 3a). We used the default temperature of 1 for all models.

RQs 1–3 deduce the best prompt design to ask LLMs about *Scratch* programs. We evaluate the LLM performance according to two levels of code understanding. At the basic level, we ask the model to determine whether a *LitterBox*-generated warning is a false-positive or a true warning. We always include the bug description in the prompt shown in Fig. 3a, but then create two alternative overall

³<https://ollama.com/library/gemma3:27b-it-qat>, hash 29eb0b9aeda3

Table 1: Conversion approaches from *Scratch* to text.

Conversion	Description
JSON	Transforms JSON file into string
JSON-Extraneous	JSON + removes extraneous JSON attributes
JSON-Sprite	Extraneous + removes sprites without buggy script
JSON-Script	Extraneous + removes all non-affected scripts
Blocks-Program	Transforms JSON into <i>scratchblocks</i> notation
Blocks-Sprite	Sprite + transforms JSON into <i>scratchblocks</i> notation
Blocks-Script	Script + transforms JSON into <i>scratchblocks</i> notation

prompts using the pairs of buggy and fixed program versions. In the first prompt variant, we add the actually buggy program to the prompt, i.e., we test whether the LLM correctly agrees with the true-positive static analysis. In the second prompt variant, we instead place the fixed program in the prompt, i.e., we pretend that the code analysis tool found the warning in the actually correct program. Assuming the LLM has an understanding of *Scratch* code, we would expect it to identify this mistake and classify it as a false-positive warning report. We query the LLMs with these two variants of the prompt for each combination of model and conversion method.

To check for a deeper level of understanding, we evaluate the LLMs’ ability to provide fixes for actual warnings as part of RQ 1–3. We measure the number of correct fixes by running the *LitterBox* analysis again after integrating the proposed change into the program. We check whether the original bug no longer exists, and using the *Whisker* tests ensure no new bugs have been introduced.

4.2.1 RQ1: Program Representation. To evaluate the impact of the format on the LLMs’ performance in detecting and fixing issues, we queried the LLMs with various formats used in the *program* placeholder in the prompt template (see Fig. 3a). We identified two main textual formats to represent a *Scratch* program: the original JSON and the *scratchblocks* notation. From these we derived the seven different conversions summarised in Table 1.

Since *scratchblocks* only represents individual scripts, we extend the notation to represent whole programs by grouping scripts of the same sprite and then adding comments that denote which sprite the following scripts belong to. Furthermore, since scripts in *Scratch*

do not have individual names or signatures like methods in textual programming languages we add comments to the start of each script marking them with the ID of their starting block (*script_id*). With this, each script can be referenced individually.

As the performance of an LLM may be affected by the presence of superfluous information, the *JSON-Extraneous* conversion uses the JSON representation but filters attributes that are irrelevant for reasoning about the code (e.g., the block locations on the *Scratch* workspace; see replication package for details). To further reduce the information provided (and tokens consumed), the conversion can also retain only the scripts of the bug-containing sprite (*JSON-Sprite* for JSON and *Blocks-Sprite* for *scratchblocks*), or only the buggy script itself (*JSON/Blocks-Script*). We use the affected block's ID supplied by *LitterBox* to identify sprites and scripts to be removed.

To benefit from the large LLM training dataset, the *Scratch* code could alternatively be converted to a Python-like syntax. However, we dismissed this format, since the LLM then could output code using 'pythonic' patterns like classes or list comprehensions that can neither be converted back to *Scratch* nor are familiar to students.

4.2.2 RQ2: Chain of Thought. Using the best program representation, we extended the base prompt (Fig. 3a) by requesting the model to *think in steps*. This chain-of-thought prompting strategy has been shown previously to improve the model's ability to determine false-positive warnings [20]. By prompting GPT-o4-mini we can evaluate whether a dedicated reasoning language model, that has been specifically trained to use a chain of thought process when answering [1, 27], results in improved responses.

4.2.3 RQ3: Few-Shot Learning. For the first two RQs we used a zero-shot approach which requires the model to already have sufficient knowledge about *Scratch* code, static analysis warnings, and the *scratchblocks* format. In this RQ we investigate whether prompts providing examples of the relevant issues and their fixes, i.e. few-shot prompting, can help the model to understand warnings and fix them. Using this approach the model can use the given examples as a template from the local context that only needs to be adapted to match the code location for the actual issue rather than having to rely on similar training data. Using the best program representation format (RQ1) and chain-of-thought prompting according to the results of RQ2, we then additionally prefix the base prompt shown in Fig. 3a with either one pair (one-shot prompting) or three pairs of example scripts from other programs containing the same *LitterBox* warning type and their fixes each.

4.2.4 RQ4: Reasoning. To analyse the bug reasoning capability we use three tasks: describing how the bug affects the observable program behaviour (Task 1), explaining a possible underlying misconception of the developer writing the faulty code (Task 2), and generating a list of steps to reproduce the faulty behaviour (Task 3). For this RQ we use the best prompt regarding program representation (RQ1) and chain of thought prompting (RQ2) based on the prior results. We do not consider one-/few-shot learning (RQ3) as examples could only very generically describe what is considered an explanation; examples of the actual effects of each different type of bug would be very specific and different for each program. While examples for false positives or fixes of bug patterns as used in RQ3 can be generated automatically, examples of how different bugs

affect individual programs would need to be manually constructed. Consequently, when students query LLMs about explaining the problems in their specific programs no other examples of the bug pattern in a similar program would be available for using few-shot learning.

RQ4 is answered by manually reporting for each task and program whether the LLM's prediction is correct, incorrect or partially correct. Given the lack of research on the feasibility and appropriateness of LLM-generated feedback for *Scratch* we consider using children (i.e., the target audience of *Scratch*) as raters to be unethical, and instead the raters are two of the authors of this paper. In the case of disagreement between their independent ratings, they discussed to reach an agreement on how to rate the task.

4.3 Threats to Validity

External Validity. Results of our explorative study may not generalise to other programs, bugs, or LLMs. We mitigate this partially by evaluating our prompts on multiple LLMs. We chose the GPT-mini variants over the newer GPT-4.1 due to their faster responses and considerably lower cost. GPT-5 was not yet available when we conducted the experiments. Gemma 3 represents a state of the art self-hosted model that can run on a single GPU. *OpenAI* might make future undisclosed changes to their proprietary GPT model and thus change the results when replicating the experiments.

We acknowledge the limited dataset size resulting from both the need to manually implement fixed versions of buggy programs and the manual analysis of RQ4. However, a human analysis is required since only then we can judge whether the LLM responses and descriptions are understandable and appropriate for children. By choosing a manual analysis instead of an automated one on a larger dataset, we accept the threat to external validity to mitigate a potential threat to internal validity.

We chose *Scratch* for our dataset since it is one of the most common introductory programming languages for children, having about 135 million registered users.⁴ We did not extend our dataset with other text-based programming languages, since *Scratch* programs contain inherently different bug patterns [11]. Experiments on such a mixed dataset would therefore not be able to draw meaningful conclusions on whether an observable difference is caused by the choice of programming language or by different bug types.

Internal Validity. Since the output of LLMs is not deterministic, repeating queries might yield different results even with the same prompts. We mitigate this by repeating every query to the LLM multiple times before taking the majority vote of these responses as final answer. A further threat to validity arises in case the LLM has seen the *Scratch* programs already during training. However, the dataset originating from the debugging study [15] was shared directly by the original authors rather than a public replication package. It is also unlikely that the programs in our dataset mined from the *Scratch* website are part of the GPT training dataset, since the download of programs is a multistep process unlikely to be implemented in a general web-crawler and since the *Scratch* API enforces a rate limit of ten requests per second. I.e., even if implemented in the training data crawlers, it is unlikely they randomly

⁴<https://scratch.mit.edu/statistics/>, accessed September 2025

Table 2: RQ1: Number of programs where the LLM agrees with the true *LitterBox* analysis. Best-performing conversion variant for each model highlighted in bold.

Conversion	Buggy programs (n=24)			Non-buggy programs (n=24)		
	4o-mini	o4-mini	Gemma	4o-mini	o4-mini	Gemma
JSON	17	23	23	10	24	0
JSON-Extraneous	17	22	24	10	24	1
JSON-Sprite	20	24	24	10	20	3
JSON-Script	22	22	24	9	17	2
Blocks-Program	23	22	24	13	24	1
Blocks-Sprite	22	18	24	7	19	1
Blocks-Script	21	20	24	8	17	0

crawled the exact same 12 programs out of over 164 million publicly available programs on the *Scratch* website.

Construct Validity. Our dataset contains no cases of actual false-positive *LitterBox* warnings. Instead, we give the fixed program version to the LLM and treat it in the prompt like a buggy program. Therefore, our experiments might report a better than actual performance in false-positive detection capability, since it is likely easier to detect the full absence of an issue than detecting more subtle cases where the *LitterBox* analysis does not work precisely. Furthermore, the experiment procedure of RQ4 involves two human raters, but an inter-rater reliability of 86.81% confirms reliable results. Extensive discussions were held between the two evaluators in order to reach a final verdict on the few discrepancies.

4.4 RQ1: Program Representation

As Table 2 shows, the Gemma 3 model in all program representations nearly always assumes the warning produced by the analysis tool is correct. Even in the best configuration, it can only detect the false-positive warning in 3 of the 24 cases. Therefore, we assume it does not actually understand the code. Using the *Blocks-Program* conversion, GPT-4o-mini can at least detect the false-positives more than half of the time (13 of 24). Interestingly, the representation seems to impact the performance of GPT-o4-mini less, resulting in it always classifying the programs correctly for all representations that include the full program. Only when reducing the available context to single sprites or scripts the performance decreases. While the context from other scripts and sprites does not help much when confirming the *LitterBox* analysis for all models, it results in a clear improvement for the detection of false-positive warnings in non-buggy programs. Other scripts may contain similar code examples which support the LLM in making its decision.

For the JSON-based program representations, the removal of unnecessary data benefits GPT-4o-mini to detect that the program is indeed buggy (JSON: 17, *Script*: 22). Conversely, the addition of more context helps when using *scratchblocks*-based inputs, both when detecting truly buggy programs and when finding false-positives. Since the blocks in the JSON format are in no specific order, we conjecture that the removal of irrelevant blocks helps the model to extract relevant information from the remaining blocks. In case of the *scratchblocks* format, however, the code is structured similar to a regular programming language so that logically related blocks (e.g., within a script) remain closely together in the textual representation.

Table 3: RQ1: Number of programs fixed by the LLM by conversion method. JSON-based conversions omitted since no fixes were produced for any of them by any of the models.

Conversion	GPT-4o-mini	GPT-o4-mini	Gemma 3
Blocks-Program	20	10	6
Blocks-Sprite	19	9	6
Blocks-Script	14	7	6

This makes it easier for the LLM to reason about the problematic code in its local context. The addition of other parts of the program may help the LLM to compare the code at the issue location to similar patterns appearing in the program, which may lead to an improved judgement whether the warning is valid or not.

For the task of fixing true warnings, the textual representation of the program is even more important. As Table 3 shows, none of the three models were able to provide a valid fix for any of the JSON-based inputs. Gemma 3 could in most cases (JSON: 20 of 24) at least generate an updated JSON that was parseable by *LitterBox*, i.e., representing a valid *Scratch* program. When given the JSON representation, the GPT variants on the other hand could only rarely generate a valid *Scratch* program-JSON (4o-mini: 1, o4-mini: 5). We expected the JSON-format to be challenging for LLMs, since the already mentioned unordered storage of blocks as a linked list requires multiple consistent changes to next-pointers when inserting or deleting blocks. In case of the *scratchblocks*-based format, we could always parse and integrate the LLM output into the original program, even if it contained syntax errors. For example, the LLMs often used brackets surrounding variables instead of parentheses (cf. Fig. 2). However, our parser implementation converts non-parseable single statements into custom block calls without effect during runtime. Therefore, the program can still be shown in the *Scratch* user interface and since we add the piece of unparseable text as name of the custom block, the original intention of the block can in most cases still be inferred so that a manual fix is possible.

Similar to the false-positives, the inclusion of context from the whole program helped both GPT variants to create a fixed program (cf. Table 3). The script-local context does not seem to be enough for the LLM to fully understand some bugs. The addition of sprite-local context seems to be more relevant than code from other sprites, since the addition of the sprite’s code led to five additional fixes in case of GPT-4o-mini, while the program-context only resulted in one more fix to a total of 20 successfully fixed programs. While for Gemma 3 the *scratchblocks*-based input also is the only one that resulted in fixed programs, we did not observe a similar improvement with additional context. Instead, the results indicate that this model is not able to reason about the program in most cases, thus resulting in only 6 fixed programs for any of the *scratchblocks*-based inputs.

Summary (RQ1) The *Blocks-Program* conversion of whole programs into *scratchblocks* notation should be used when querying an LLM, since this format clearly showed the best performance for both tasks when using the GPT-4o-mini model. For Gemma 3, the results are not as conclusive, which we attribute to a general lack of program understanding rather than the representation.

Table 4: RQ2, RQ3: Number of programs where the LLM and LitterBox agree. Prompts are zero-, one-, or three-shot.

RQ	Model	Shots	Buggy Programs	Non-buggy Programs
2	GPT-4o-mini	0	23	15
2	GPT-o4-mini	0	20	24
2	Gemma 3	0	24	14
3	GPT-4o-mini	1	24	6
3	GPT-4o-mini	3	24	5
3	Gemma 3	1	23	14
3	Gemma 3	3	24	3

4.5 RQ2: Chain of Thought

Since both detection of real bugs and false-positives as well as creating fixes requires reasoning about the program, a chain-of-thought process could lead the LLMs to a more detailed examination and understanding of the given program. As seen in Table 4, for both Gemma 3 and GPT-4o-mini the number of buggy programs where the LLM and *LitterBox* agree stays the same compared to RQ1 with 24 and 23 out of 24 cases respectively. However, for the detection of false-positive linter warnings the chain-of-thought prompting results in an improvement for both LLMs. While there is only a slight increase for GPT-4o-mini (from 13 to 15), Gemma 3 profits heavily. Here the false-positive detection increases from 1 (for the *Blocks-Program* encoding also used in this RQ) to 14 of 24 cases. This suggests that triggering a more active reasoning process can help the LLMs in detecting false-positives instead of just agreeing with what is contained in the prompt.

Unexpectedly, the reasoning model GPT-o4-mini has the lowest agreement with *LitterBox* on the buggy programs with only 20 cases. When looking at the false-positive detection rate however, this model stands out by detecting all 24 false linter warnings. This further supports the insight that an active reasoning process leads to a more critical examination of the user input, detecting statements which do not fit the program, and highlights that this model has been specifically trained for this style of decision-making.

For the task of fixing true warnings, the reasoning via chain-of-thought seems to be less important than for discerning false-positives. Using GPT-4o-mini the number of 20 programs fixed by integrating the LLM given *scratchblocks* code stays the same as when using the normal prompt (see Table 5). Looking at Gemma 3, there is only a slight increase from 6 to 7 fixes. This may be rooted in the fix generation just needing to find the broken part of the code and to produce an altered version without critical reasoning if the program really is broken like in the task to detect false-positives. The reasoning focus GPT-o4-mini was trained for even seems to hinder this model compared to the normal GPT-4o-mini as it only produced 8 fixes. As with the other models, this task does not seem to target the reasoning ability, which results in GPT-o4-mini not being able to utilise its core feature.

Summary (RQ2) Chain-of-thought prompting should be used for detecting false-positive linter warnings as it shows very promising results compared to the basic prompt in RQ1, especially when using the dedicated reasoning model. However, it seems to have no effect on creating fixed programs.

Table 5: RQ2, RQ3: Number of programs fixed by the LLM.

RQ	Shots	GPT-4o-mini	GPT-o4-mini	Gemma 3
2	0	20	8	7
3	1	22	—	18
3	3	22	—	21

4.6 RQ3: Few-Shot Learning

The chain-of-thought process may be supported by giving the LLM some examples of similar cases as part of the prompt. Since the LLMs were already able to nearly always confirm true bugs in the programs, we do not observe an improvement when adding examples to the prompt (cf. Table 4). More interestingly, the additional examples lead to a worse performance in detecting false-positive warnings. For example, GPT-4o-mini with zero-shot prompting can correctly classify 15 instances, but only 6 when also having an example as part of the prompt. Only Gemma 3 with one example can retain the previous performance of detecting 14 false-positives.

Conversely, when tasked with fixing programs, the examples of similar fixes help the LLMs (cf. Table 5). For GPT-4o-mini we observe a slight improvement to the already high number of successful fixes (20 to 22), but for Gemma 3 the performance considerably improves from 7 fixed programs when using zero-shot prompting to 18 fixes for one-shot and 21 for three-shot prompting. As highlighted in the previous research questions, Gemma 3 does not seem to be able to reason about the code as well as GPT-4o-mini. Therefore, the examples might help this model specifically to re-use parts thereof when fixing the actual bug without requiring a further understanding of the underlying program semantics.

Such a re-use of code examples without deeper understanding might also explain the worse performance in the false-positive detection task. When given no examples, the LLMs can only make their decision based on the actual code and the description of the bug as obtained by *LitterBox*. However, when given additional examples of similar cases of false-positive reports, the LLM might now try to match patterns between the examples and the actual code. Since the examples stem from a different program, the patterns do not sufficiently match and the LLM subsequently decides that the program has the described bug. Since the LLMs therefore tend to err towards the decision that the program has a bug, the performance for the actually buggy programs remains high.

Compared to RQ1 and RQ2, both models can achieve a high number of fixed programs when given examples (cf. Tables 3 and 5). However, as Fig. 4 shows, only between 16.33% (RQ1, o4-mini) and 74.42% (RQ2, 4o-mini) of the programs still pass all functional tests. While for GPT the ratio of functionally correct programs increases when using chain-of-thought prompting and decreases again when adding examples, Gemma 3 only improves in the last case. This decrease for RQ3 might again stem from the re-use of code examples in the different program without fully adapting them to the new requirements, i.e., the examples are misleading the model.

As explained in Section 4.4, our *scratchblocks* parser integrates unparseable statements as no-op custom call statements into the

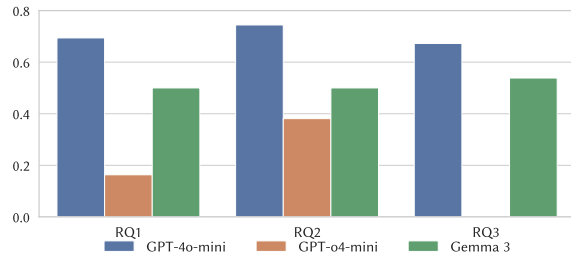


Figure 4: Proportion of functionally correct programs after applying the LLM-suggested fix. Shows only the best-performing variant, i.e., RQ1 Blocks, RQ2 chain-of-thought, RQ3 with one example for GPT and three for Gemma.

fixed program. The improved test results suggest that for the worse-performing Gemma 3 the examples might help the model to respond in valid *scratchblocks* syntax more often.

As parse failures are handled as no-op blocks, we expect that programs that no longer contain the *LitterBox* issue but also do not pass all tests in most cases do not contain superfluous code which breaks the program logic, but are rather missing functionality that would have been present if all blocks could have been parsed.

Summary (RQ3) The best prompt design for both tasks and models uses the whole program in *scratchblocks* format in a chain-of-thought prompt. GPT-4o-mini works best without examples when detecting false-positives and with one example when fixing bugs. For Gemma 3, one example is best for detecting false-positives and three for fixing bugs. Not only the task but also the model must be considered when re-using or adapting prompts.

4.7 RQ4: Reasoning

Since RQ1 and RQ2 demonstrated that using the *Blocks-Program* representation and prompting the LLMs to *think in steps* worked best, RQ4 is answered by queries to GPT-4o-mini and Gemma 3 to reason about *Scratch* programs, using *scratchblocks* to represent programs and require them to follow the *Chain-of-Thought* paradigm. We refrain from providing examples as in RQ3 because providing such examples for the three tasks analysed in RQ4 is not feasible in a real-world application setting, as described in Section 4.2.4.

4.7.1 Task 1. In the first task, we asked the LLMs to predict how the provided *LitterBox* warning affects the program behaviour. As shown by Table 6, GPT-4o-mini solves the first task 11 times, produces a partially correct answer 7 times, and fails to solve the task 6 times. In contrast, Gemma 3 produces a satisfying, partially correct and incorrect answer for 12, 4 and 8 programs, respectively.

We observe that in two-thirds of all programs, both LLMs achieve the same result. However, even if the results for a given program align, the LLM answers tend to vary in the depth of the provided explanations. For instance, both LLMs produce a satisfying answer for the *Treasure* program, in which the developer apparently confused a `say for seconds` block with a `broadcast` block. While the former has a purely visual effect on the program, the latter actually changes the program flow by triggering an event to which other listeners can react. As the answers of both LLMs, shown in Fig. 5, correctly mention,

Table 6: Classified predictions for reasoning about bugs in *Scratch* programs. ✓, ✗, and ◐ mark if the task was solved, not solved, or partially solved by GPT-4o-mini/Gemma 3.

Program	Issue	Dataset	Task 1	Task 2	Task 3
HappyNewYear	Comparing Literals	Study	✗/✗	✓/✗	✓/✓
Pong	Comparing Literals	Study	✓/✓	✓/✗	✓/✓
Balloon	Comparing Literals	Sample	◐/◐	✓/✓	✓/✓
Basketball	Comparing Literals	Sample	✗/✗	✓/✗	✗/✗
Parrot	Forever Inside Loop	Study	✓/◐	✓/✓	✓/✗
Winter	Forever Inside Loop	Study	✓/✗	✓/✓	◐/✗
IceCream	Forever Inside Loop	Sample	✗/✗	✓/✓	✗/✗
MagicCat	Forever Inside Loop	Sample	✓/✓	✓/✓	◐/✓
Labyrinth	Message Never Rec	Study	◐/✓	✓/✓	◐/◐
TicTacToe	Message Never Rec	Study	✓/✓	✓/✓	✓/✓
ExplodingCar	Message Never Rec	Sample	✓/✓	✓/✓	◐/✓
Storm	Message Never Rec	Sample	◐/✓	✓/✓	✓/✓
Garden	Message Never Sent	Study	◐/◐	✓/✓	✗/◐
Treasure	Message Never Sent	Study	✓/✓	✓/✓	✓/✓
BallRace	Message Never Sent	Sample	◐/◐	✓/✓	◐/◐
Race	Message Never Sent	Sample	✓/✓	✓/✓	✗/✓
Catch	Missing Clone Init	Study	◐/✓	✓/✓	✗/✓
Snake	Missing Clone Init	Study	✗/✗	✓/✓	✗/✗
EvilCat	Missing Clone Init	Sample	✓/✓	✓/✓	✓/✓
Points	Missing Clone Init	Sample	✓/✗	✓/✓	✓/✗
Ghosts	Missing Loop Sensing	Study	✗/✓	✓/✓	✗/✓
Stars	Missing Loop Sensing	Study	◐/✗	✓/✓	◐/✗
Sheep	Missing Loop Sensing	Sample	✗/✗	✓/✓	✗/✗
SoundCat	Missing Loop Sensing	Sample	✓/✓	✓/✓	✓/✓
Sum Solved			11 / 12	24 / 21	10 / 13

The faulty code affects the observable behavior of sprites because the script associated with the 'Girl' sprite will never execute. This is due to the fact that the message 'Happy End!' is never broadcasted after it is said by the 'Pirate' sprite. As a result, the 'Girl' sprite, which is supposed to respond to that message, remains hidden and does not enlarge or appear on the screen. This means that when the program runs, the audience will not see the 'Girl' sprite's expected behavior, leading to an incomplete storytelling experience.

(a) GPT-4o-mini.

The problem is that the message 'Happy End!' is *displayed* by the Pirate sprite using 'say' blocks, but it's *never broadcast* using a 'broadcast' block. In Scratch, 'say' simply shows text on the screen; it doesn't trigger other scripts. The scripts in the Girl sprite, and potentially others (though only Girl's script is shown to be affected), are waiting to *receive* this message using 'when I receive [Happy End! v]'. Because the message is never broadcast, those scripts will *never* run. Specifically, this means the Girl sprite will remain hidden throughout the entire execution of the program. Its script, which makes it appear and grow in size, will never be activated. The rest of the program *will* run as expected – the Pirate will shrink, glide around, and say its lines, the Treasure Map will position itself, the Volcano Island will appear when the Pirate gets close, and the Treasure Chest will show up. However, the effect involving the Girl will not happen, so the user will not see the Girl sprite at all.

(b) Gemma 3.

Figure 5: Task 1 answers of GPT-4o-mini and Gemma 3 for the *Treasure* program. To avoid confusion, we translated the sprite and message names to English.

this confusion leads to the *Girl* sprite not appearing on the screen, which significantly alters the shown animations in the program. However, only Gemma 3 explains that the two block types have been confused. From a pedagogical point of view, the response of Gemma 3 is preferable as it reveals a fundamental misunderstanding about programming in the *Scratch* environment.

Both LLMs produce fairly satisfying results for the issue categories *Message Never Received* and *Message Never Sent*, with all

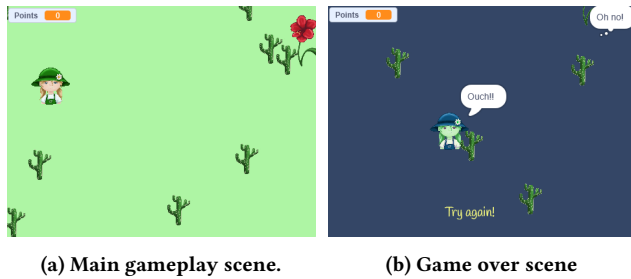


Figure 6: The Garden program containing a Message Never Sent bug.

answers deemed correct or partially correct. Partially correct results often result from the LLMs producing a proper answer but hallucinating further details that are either confusing or simply wrong. For example, in the *Garden* program, depicted in Fig. 6a, both LLMs correctly explain that the faulty code affects the behaviour of the *Cactus* sprite if the player touches one of these sprites. However, both models hallucinate that the given *LitterBox* issue also leads to a missing indication that the game ends, which is wrong since the program changes the background scene and prompts the player to *Try again!* as shown in Fig. 6b, regardless of the issue’s presence.

For the programs *Basketball*, *IceCream* and *Snake*, both models produce unsatisfying answers. In all three programs, the LLMs cannot infer the program behaviour, leading to false assumptions about the control flow. Ultimately, these faulty assumptions cascade into a wrong prediction of how a given *LitterBox* warning affects the program. Since each of the three programs belongs to a different issue category, we conjecture that the LLMs struggle with understanding the behaviour of these programs, regardless of their errors.

4.7.2 Task 2. The second task queries the LLMs to explain which programming concept the developer of the buggy program might have misunderstood. Both models perform exceptionally well on this task, with GPT-4o-mini producing satisfying answers for all 24 programs of the dataset and Gemma 3 solving the task for 21 programs. The good results in this task are particularly interesting for teaching, as this promises an automated generation of hints that help students eliminate their misunderstandings without forcing them in a specific direction for solving the task [13] and not giving away the solution to a problem straight away [23, 26].

Gemma 3 fails to produce satisfying answers for the programs *HappyNewYear*, *Pong* and *Basketball*, which all contain *Comparing Literals* issues. For *HappyNewYear*, Gemma 3 wrongly attributes the issue to a `wait until condition` block, while the issue actually originates from confusing the use of variables and strings in an equals condition. Even worse, the LLM argues that the `wait until condition` block does not continually monitor whether the respective condition is satisfied. This is incorrect since repeatedly evaluating the respective condition is the exact purpose of this block. For *Pong*, Gemma 3 incorrectly predicts the player’s desired program behaviour, which ultimately leads to a wrong assumption of the underlying misconception. Finally, when prompted for the *Basketball* program, the model hallucinates the presence of a bug different from the one given by *LitterBox*, and thus, responds with a nonexistent misconception.

The underlying programming concept that may be misunderstood by the person who wrote the code is the use of variable comparisons. It appears that the programmer intended to compare the variable ‘current year’ to 2021 in order to make the script progress after waiting. However, they mistakenly referenced a static value rather than using the variable itself that holds the changing year. This misunderstanding can happen if the programmer does not fully grasp how to use and update variables in conditional checks. Proper understanding of variable manipulation and comparison logic is essential to ensure that the code behaves as intended.

(a) HappyNewYear

The underlying programming concept that may have been misunderstood is the use of variables in conditional statements. In Scratch, a comparison statement typically involves checking a variable’s value against a constant. The original code seems to lack the proper variable within the comparison brackets, which suggests that the programmer might not have understood how to reference and utilize variables correctly. The proper structure is to insert a variable (like ‘baloons’) in the comparison to check its value, rather than leaving it empty, leading to the always-FALSE condition.

(b) Balloon.

Figure 7: Task 2 answers of GPT-4o-mini for the *HappyNewYear* and *Balloon* programs. We translated sprite and message names to English for the purpose of this figure.

While Gemma 3 tends to struggle with bugs that fall into the *Comparing Literals* category, GPT-4o-mini manages to make a fine distinction about the developer’s misunderstanding. As an example, consider the answers given by GPT-4o-mini for the programs *HappyNewYear* and *Balloon* shown in Fig. 7, which both suffer from incorrectly referencing variable values. GPT-4o-mini explains that for the game *HappyNewYear*, a string expressing the name of a variable was used as an argument in a comparison operator instead of the actual variable. In contrast, for the program *Balloon*, the model recognises that the developer did not insert an argument in the first place and left the left-hand side of the comparison blank. If an LLM is to be applied in an educational setting, these fine distinctions are crucial as they help students in overcoming their misunderstandings about the given programming environment.

Although Gemma 3 may occasionally produce bad answers (e.g., for the *HappyNewYear* program), the reviewers frequently felt that the output of Gemma 3 was easier to understand and more accessible for young students, which is the primary target group of *Scratch* and block-based programming environments in general. For example, when queried for the *TicTacToe* program, which contains a `broadcast` block but no matching message handler, the LLM explains the concept of event broadcasting via the following analogy: *The concept is analogous to speaking on a radio—broadcasting a message doesn’t automatically mean someone is listening. You need a receiver tuned to the correct frequency to hear and respond to the signal.*

4.7.3 Task 3. In the final task, we prompted the LLMs to generate a list of steps to reproduce the faulty behaviour indicated by the *LitterBox* warning. GPT-4o-mini and Gemma 3 solved this task 10 and 13 times, respectively. Surprisingly, even though the prompt does not explicitly ask for it, the LLMs also add to each user input the behaviour users should observe after executing the input action, which is not only useful for debugging, but also provides an opportunity for dynamic program analyses to overcome hallucinations. Besides the immediate use for debugging the faulty program, the suggested steps to reproduce the behaviour and their results offer an exciting opportunity for future work for automated testing of *Scratch* programs [3, 8] by interpreting the suggested steps as test inputs and the expected behaviour as test assertions.

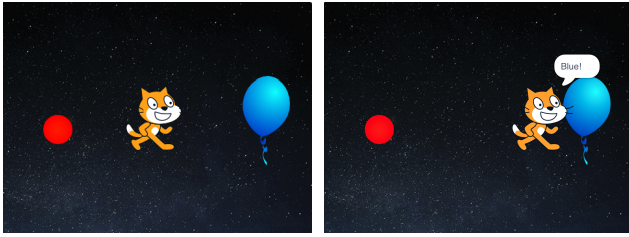


Figure 8: The Soundcat program containing a Missing Loop Sensing bug.

Both LLMs failed to produce adequate steps to reproduce the bugs in 8 programs. In the case of GPT-4o-mini, 6 answers were deemed wrong since the model failed to infer the program behaviour resulting from executing the provided steps. The remaining two answers were considered not solved as the LLM prompts the user to perform actions to which the program does not react. Regarding the faulty answers provided by Gemma 3, 7 are due to inferring incorrect programming behaviour as well, while one was deemed incorrect as the model suggested using a tooltip feature that does not exist. We assume that the model refers to the *Tips Window*⁵ feature, which was part of a previous *Scratch* version but has been removed since. We observe a high overlap between failures for task three and failures for the other two tasks, as both LLMs fail to reason about the expected program behaviour after executing the provided inputs if they already failed to provide satisfying answers to the previous tasks. This observation underlines the hypothesis that for some programs, the LLMs struggle to understand the provided *Scratch* code. In the future, this lack of understanding could be tackled by fine-tuning LLM models to reason about *Scratch* programs.

Similar to *Task 2*, the reviewers agreed that Gemma 3 produces answers with more details and rated them to be more accessible, especially for young students. For instance, when prompted for the *SoundCat* program (cf. Fig. 8), Gemma 3 provides the instructions shown in Fig. 9. The LLM correctly explains that some sound effects are only played if the sprite overlaps with blue or red colours directly after it has moved to the right or left, respectively. Furthermore, the model invites the user to experiment with varying starting positions of the moving sprite in order to experience the potential effects of the bug. Such answers are very valuable from a pedagogical perspective, since experimenting with the program based on the provided scenarios helps learners to overcome potential misunderstandings and teaches them how to test their programs.

Overall, we observe that both LLMs manage to solve the tasks to a varying degree. They both perform exceptionally well in *Task 2*, promising exciting future avenues in automatically generating programming hints for students. However, both models sometimes struggle to comprehend the given *Scratch* code, leading to faulty answers for tasks 1 and 3. In the future, we hope to improve the LLMs' reasoning about *Scratch* programs by fine-tuning the LLMs to predict the behaviour of *Scratch* programs. Nevertheless, the results of task 3 indicate that LLMs could be used to assist students in debugging their programs in the future. Finally, the good results of Gemma 3 might be particularly interesting for educators, since

⁵https://en.scratch-wiki.info/wiki/Tips_Window, accessed September 2025

Steps to observe the faulty behaviour:

Here are the steps you can perform to observe the bug in action:

1. *Start the program:* Press the *green flag*. 'Sprite1' will move forward 10 steps 10 times.
2. *Initial Blue Check:* Observe if 'Sprite1' initially overlaps with the blue colour area. If it does, you will hear the 'Doorbell v' sound and see the 'Blue!' message.
3. *Move and subsequent Blue check:* If 'Sprite1' *does not* initially overlap with blue, but *moves* into a blue area *after* the first 'repeat' loop, observe that the 'Doorbell v' sound *does not* play. This is because the blue check only happened once after the first 'repeat' loop finished.
4. *Move and subsequent Red check:* Similarly, if 'Sprite1' moves forward 27 steps 10 times and *then* overlaps with red, observe if the 'G Bass v' sound plays. If it does not, this is due to the same issue.
5. *Test multiple scenarios:* Repeat the experiment several times, starting 'Sprite1' in different positions relative to the blue and red areas, to confirm the behaviour. You'll see that the sounds only play if the Sprite happens to overlap with the colours immediately after the 'repeat' loops. If the Sprite enters the coloured areas *between* the loops, it will not react.
6. *Observe the Sprite's movement:* As the sprite moves forward, check whether it remains inside of the coloured areas. If it does, it will not respond because the conditional statement is only checked once. These steps will clearly demonstrate that the conditional statements are not continuously monitoring the touching color condition, as the user likely intended. This confirms the bug identified by the static analysis tool.

Figure 9: Task 3 answer of Gemma for the Soundcat program.

the model provides answers with great detail and can be hosted locally on computing clusters in schools.

Summary (RQ4) Both studied LLMs excel at explaining possible misunderstandings of the programmer. However, both models sometimes struggle explaining how bugs influence program behaviour, and suggesting steps to reproduce this faulty behaviour.

5 Conclusions

Although program analysis tools can identify bugs in *Scratch* programs, learners may require more help to resolve their misunderstandings and fix their bugs. Our investigation demonstrates how to best represent *Scratch* programs in LLM prompts to provide this help. In conjunction with a program analysis tool, LLMs are able to reason about the consequences of bugs, infer steps to reveal faulty behaviour, and even provide suggestions on how to fix bugs.

These results enable many opportunities for future work, such as enriching automated feedback, for example by providing not just entire fixes, but next-step hints, related examples, explanations of misunderstood concepts, or other context information instead. Given that our baseline results showed that LLMs can reason about *Scratch* programs without certain bounds, it might now be ethically reasonable to continue future evaluations involving children in studies on the perceptions and effects of the LLM responses. Our results also imply necessary future research on identifying hallucinations, for example by trying to automatically execute suggested sequences of interactions. We envision an agentic framework that can for example ask the LLM to fix the *scratchblocks* syntax in its output and run the static analysis on the updated program to automatically and iteratively improve the LLM responses. Our experiments are based on three vanilla LLMs, and it would be interesting to experiment with models fine-tuned for *Scratch*. Finally, we also envision integrating the framework into the user interface of *Scratch* for a seamless interaction with the LLM.

The source code of our framework including our extensions to *LitterBox* are publicly available together with the dataset, all used prompts, scripts for reproducing the experiment and all obtained results in our replication package [6].

Acknowledgments

This work is supported by DFG project FR 2955/5-1 'Types4Strings'.

References

- [1] Maciej Besta, Julia Barth, Eric Schreiber, Ales Kubicek, Afonso Catarino, Robert Gerstenberger, Piotr Nyczyk, Patrick Iff, Yueling Li, Sam Houliston, Tomasz Sternal, Marcin Copik, Grzegorz Kwaśniewski, Jürgen Müller, Lukasz Flis, Hannes Eberhard, Hubert Niewiadomski, and Torsten Hoefler. 2025. Reasoning Language Models: A Blueprint. (2025). arXiv:2501.11223 [cs.AI]
- [2] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. 2013. Hairball: Lint-inspired static analysis of Scratch projects. In *Technical Symposium on Computer Science Education (SIGSCE)*. ACM, 215–220. doi:10.1145/2445196.2445265
- [3] Adina Deiner, Patric Feldmeier, Gordon Fraser, Sebastian Schweikl, and Wengran Wang. 2023. Automated Test Generation for Scratch Programs. *Empirical Software Engineering* 28, 3 (2023). doi:10.1007/s10664-022-10255-x
- [4] Stefania Druga and Amy J. Ko. 2025. Scratch Copilot: Supporting Youth Creative Coding with AI. In *Interaction Design and Children (IDC)*. ACM. doi:10.1145/3713043.3727051
- [5] Stefania Druga and Nancy Otero. 2023. Scratch Copilot Evaluation: Assessing AI-Assisted Creative Coding for Families. (2023). arXiv:2305.10417 [cs.HC]
- [6] Benedikt Fein, Patric Feldmeier, Gordon Fraser, and Florian Obermüller. 2026. Reasoning About Bugs in Learners' Scratch Programs Using Large Language Models. <https://doi.org/10.5281/zenodo.18096428> Replication Package.
- [7] Benedikt Fein, Florian Obermüller, and Gordon Fraser. 2025. LitterBox+: An Extensible Framework for LLM-enhanced Scratch Static Code Analysis. In *International Conference on Automated Software Engineering (ASE)*. IEEE. arXiv:2509.12021 [cs.SE]
- [8] Patric Feldmeier and Gordon Fraser. 2023. Neuroevolution-Based Generation of Tests and Oracles for Games. In *International Conference on Automated Software Engineering (ASE)*. ACM, 1–13. doi:10.1145/3551349.3556939
- [9] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its Nature, Scope, Limits, and Consequences. *Minds and Machines* 30 (2020), 681–694. doi:10.1007/s11023-020-09548-1
- [10] Gordon Fraser, Ute Heuer, Nina Körber, Florian Obermüller, and Ewald Wasmeier. 2021. Litterbox: A Linter for Scratch Programs. In *International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 183–188. doi:10.1109/ICSE-SEET52601.2021.00028
- [11] Christoph Frädriich, Florian Obermüller, Nina Körber, Ute Heuer, and Gordon Fraser. 2020. Common Bugs in Scratch Programs. In *Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 89–95. doi:10.1145/3341525.3387389
- [12] Gemma Team and Google DeepMind. 2025. Gemma 3 Technical Report. (March 2025). arXiv:2503.19786 [cs.CL]
- [13] Luisa Greifenstein, Markus Brune, Tobias Fuchs, Ute Heuer, and Gordon Fraser. 2023. Impact of Hint Content on Performance and Learning: A Study with Primary School Children in a Scratch Course. In *Conference on Primary and Secondary Computing Education Research (WiPSCE)*. ACM, 1–10. doi:10.1145/3605468.3605498
- [14] Luisa Greifenstein, Isabella Graßl, and Gordon Fraser. 2021. Challenging but full of opportunities: Teachers' perspectives on programming in primary schools. In *Koli Calling International Conference on Computing Education Research*. ACM, 1–10. doi:10.1145/3488042.3488048
- [15] Luisa Greifenstein, Florian Obermüller, Ewald Wasmeier, Ute Heuer, and Gordon Fraser. 2021. Effects of Hints on Debugging Scratch Programs: An Empirical Study with Primary School Teachers in Training. In *Workshop in Primary and Secondary Computing Education (WiPSCE)*. ACM, 1–10. doi:10.1145/3481312.3481344
- [16] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *CHI Conference on Human Factors in Computing Systems*. ACM, 1–23. doi:10.1145/3544548.3580919
- [17] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2019. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Transactions on Computing Education* 19 (2019), 1 – 43. doi:10.1145/3231711
- [18] Natalie Kiesler, Dominic Lohr, and Hieke Keuning. 2023. Exploring the potential of large language models to generate formative programming feedback. In *Frontiers in Education Conference (FIE)*. IEEE, 1–5. doi:10.1109/FIE58773.2023.10343457
- [19] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. 2023. Using large language models to enhance programming error messages. In *Technical Symposium on Computer Science Education (SIGSCE)*. ACM, 563–569. doi:10.1145/3545945.3569770
- [20] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2107–2111. doi:10.1145/3611643.3613078
- [21] David Liu and Andrew Petersen. 2019. Static analyses in Python programming courses. In *Technical Symposium on Computer Science Education (SIGSCE)*. ACM, 666–671. doi:10.1145/3287324.3287503
- [22] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education* 10 (11 2010), 16. doi:10.1145/1868358.1868363
- [23] Samiha Marwan, Anay Dombé, and Thomas W. Price. 2020. Unproductive Help-Seeking in Programming: What It is and How to Address It. In *Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 54–60. doi:10.1145/3341525.3387394
- [24] Monica M. McGill and Adrienne Decker. 2020. Tools, Languages, and Environments Used in Primary and Secondary Computing Education. In *Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 103–109. doi:10.1145/3341525.3387365
- [25] Marcus Messer, Neil CC Brown, Michael Kölling, and Miaoqing Shi. 2024. Automated grading and feedback tools for programming education: A systematic review. *ACM Transactions on Computing Education* 24, 1 (2024), 1–43. doi:10.1145/3636515
- [26] Florian Obermüller, Luisa Greifenstein, and Gordon Fraser. 2023. Effects of Automated Feedback in Scratch Programming Tutorials. In *Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 396–402. doi:10.1145/3587102.3588803
- [27] OpenAI. 2024. Learning to reason with LLMs. (Sept. 2024). <https://openai.com/index/learning-to-reason-with-llms/>
- [28] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Conference on International Computing Education Research (ICER)*. ACM, 27–43. doi:10.1145/3501385.3543957
- [29] Michael Striwe and Michael Goedicke. 2014. A review of static analysis approaches for programming exercises. In *International Computer Assisted Assessment Conference*. Springer, 100–113. doi:10.1007/978-3-319-08657-6_10
- [30] Peeratham Techapalokul and Eli Tilevich. 2017. Quality Hound — An online code smell analyzer for scratch programs. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 337–338. doi:10.1109/VLHCC.2017.8103498
- [31] Aman Yadav, Sarah Gretter, Susanne Hambrusch, and Phil Sands. 2016. Expanding computer science education in schools: understanding teacher experiences and challenges. *Comput. Sci. Educ.* 26, 4 (2016), 235–254. doi:10.1080/08993408.2016.1257418