Evaluation of generated code edits from LLMs

Pontus Berglund

November 2025

Abstract

This report evaluates the code-editing capabilities of Large Language Models (LLMs) by benchmarking 10 models via the OpenRouter API. The study assesses the models' ability to perform three distinct software engineering tasks: fixing a logic error, resolving a runtime crash, and extending a class with new functionality. To test adherence to strict output constraints, each model was prompted to provide solutions in three specific formats: returning the whole corrected file, generating a standard diff, and producing a unified diff (UDiff). A custom automated testing pipeline was developed to parse these varied response formats and execute the resulting code against simple tests. The results provide comparative insights into model accuracy and performance for the different edit formats. Results indicate a significant performance disparity between output formats. Models achieved 90% accuracy when generating whole files, but success rates dropped to 66.7% for standard diffs and 26.7% for unified diffs. Mistral Small 3.1 and Grok 4.1 emerged as the top performers, demonstrating that while models can correct code, their adherence to a strict output format poses a challenge for automated workflows. All code, prompts, and generated model responses are available at https://github.com/pontaberglund/Evaluaute-Code-Edits

1 Introduction

A Large Language Model (LLM) takes a text prompt and returns a text answer. This report investigates the performance of ten different LLMs when prompted with Python code snippets and asked to either fix the code to make it work or add new features. Three different code snippets and three different edit formats are tested and evaluated.

2 Method

To start, an API offering multiple LLMs was selected. The choice of API was OpenRouter (https://openrouter.ai/). From the list of models, 10 were chosen. This was an iterative process, finding 10 free models that did not get rate-limited. After some attempts, the following ten models were selected.

Model	Access
x-ai/grok-4.1-fast	free
kwaipilot/kat-coder-pro	free
z-ai/glm-4.5-air	free
google/gemma-3-4b-it	free
google/gemma-3-12b-it	free
nvidia/nemotron-nano-12b-v2-vl	free
google/gemma-3-27b-it	free
meta-llama/llama-3.3-70b-instruct	free
openai/gpt-oss-20b	free
mistralai/mistral-small-3.1-24b-instruct	free

Table 1: Models used in this project

After this, three types of Python code snippets were created. After some research into existing benchmark problems, it was hard to find simple snippets to test. Therefore, three types of tasks were defined: logic malfunction, crashing code, and class extension. Then Gemini was asked to generate three code snippets and the corresponding tasks to fix them. The problems and prompts to generate them can be found in Appendix A.

The three edit types to be tested were chosen to be whole, diff, and udiff (https://aider.chat/docs/more/edit-formats.html). After that, the prompts could be created for the different problems and edit types. The prompts are in Appendix B. In general, the prompts were kept simple, with an example of what they were supposed to respond with. It was also made clear that they should not return anything more than the code edit.

Lastly, the models were prompted, and the responses were stored. The different responses were then parsed to apply the edits to the code. The edited code was then executed and subjected to simple tests to determine whether the LLM succeeded. Due to the small size of this experiment, the tests were designed only to test the task described in the prompts, not any other edge cases. Two main results were focused on: overall accuracy by edit format and model accuracy across all edit types. The responses that did not pass were manually inspected to determine the main cause of the failure.

3 Results

The overall accuracy by edit type is shown in Figure 1. It shows the whole edit format performed best with a 90% accuracy, followed by diff and udiff at 66.67% and 26.67% respectively. The model accuracy across all edit types is shown in Figure 2. It shows the Mistral small 3.1 model performed best with 100% accuracy, and the Grok 4.1 fast model came in second place with 88.89% accuracy.

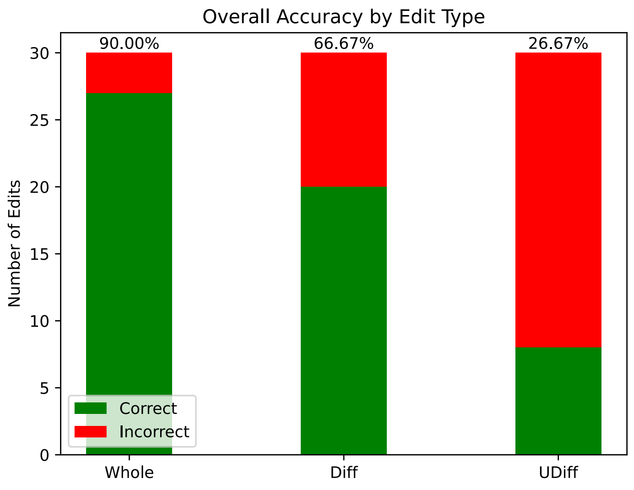

Figure 1: Bar chart of overall accuracy by edit type.

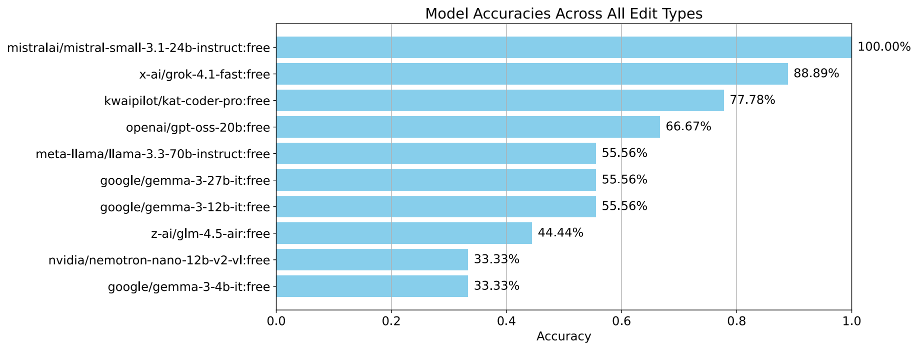

Figure 2: Bar chart of model accuracy across all edit types.

3.1 Example failed responses

In this section, some responses that did not pass the tests will be shown. The overall summary of why responses failed after manual inspections will also be provided.

3.1.1 Whole

Only three responses failed the whole edit format. After manual inspection of the three, it was shown all failed due to an explanation or other text in the response. One example is shown below:

~~~
To fix the function, we need to sort the list before calculating
the median. The median requires the list to be in ascending order
to correctly identify the middle value(s). Here's the corrected
code:

```python
def calculate_median(numbers):
    """
    Calculates the median of a list of numbers.
    """
    if not numbers:
        return None

    sorted_numbers = sorted(numbers)
    n = len(sorted_numbers)
    mid_index = n // 2

    if n % 2 == 1:
        return sorted_numbers[mid_index]
    else:
        return (sorted_numbers[mid_index - 1] + sorted_numbers[mid_index]) / 2
```
~~~

3.2 Diff

Some responses were hallucinated or factually incorrect, searching for non-existent code, like this:

From manual inspection of the failed responses, it was found that many failed due to hallucinations or incorrect format. Some failed because the response search block skipped empty lines that were in the original code block. Therefore, the strict search-and-replace code could not find the search block. Below are examples of hallucination and of indentation failure.

```
-----
Model: google/gemma-3-12b-it:free
-----
Original response:
<<<<<<< SEARCH
    numbers.sort()
=======
    numbers = sorted(numbers)
REPLACE>>>>>>>
-----


-----
Model: nvidia/nemotron-nano-12b-v2-vl:free
-----
Original response:
<<<<<<< SEARCH
for i in range(len(user_ids) + 1):
current_id = user_ids[i]
=======
for i in range(len(user_ids)):
current_id = user_ids[i]
REPLACE>>>>>>>
```

3.3 Udiff

After manual inspection of all failed code blocks, it was found that most failed to respond with the correct format. Some also struggled with the empty line problems mentioned in the Diff section. Below are three examples where the wrong format was returned, and one where the indentation was incorrect, with the else having one space too many.

```
-----
Original response:
@@
    n = len(numbers)
+    numbers = sorted(numbers)
    mid_index = n // 2

-----

-----
Original response:
@@ for i in range(len(user_ids) + 1): @@
- for i in range(len(user_ids) + 1):
+ for i in range(len(user_ids)):
@@

-----

-----
Original response:
@@ -11,4 +11,4 @@
     if not numbers:
         return None

-    n = len(numbers)
-    mid_index = n // 2
+    sorted_numbers = sorted(numbers)
+    return sorted_numbers[mid_index]

-----

-----
Part of original response:
        if item_name in self.inventory:
-            self.inventory[item_name] += quantity
+            self.inventory[item_name] += quantity
         else:
             self.inventory[item_name] = quantity
-----
```

4 Discussion

In this section, the results will be discussed. The main difficulties of the task at hand will also be addressed.

4.1 Edit format performance

The whole edit format was by far the best with 90% accuracy. Worth noting is that the incorrect edits throughout the format were not about the code itself being wrong, but rather that the response contained other text, despite the prompt saying it should not. The reason this format is so good is likely that it does not require any advanced parsing to comprehend the edit. The LLMs know how to write Python code, so they do it well.

The main problem with the diff and udiff formats is that they are hard to parse. Some of the responses were very good at having exactly the original code, including spaces and empty lines in their search blocks (including the - lines in udiff). This way, a strict search-and-replace operation could be used to produce the correct code. The parsing for the diff and udiff formats was first done very strictly and then edited to try to allow even if the model output incorrect exact indentation, but correct relative indentation. You could argue that some of the failed tests are the fault of the parser and not the model.

A manual inspection of the incorrect responses showed some of them produced correct code, but not in the requested format. In the future, this could be improved by using a better parser and seeing whether the results improve. However, due to limited time, a rather strict parser was used, and some models still produced correct edits. A balance is needed for how flexible the parsing should be and how precise the model needs to be in its responses.

The udiff format performed worst, mainly because the output was not in the requested format. Interestingly, despite the prompt saying not to count code lines or edits, some did, probably because they are trained on lots of git diff logs. In the future, this prompt could be clearer about the format and offer more examples. This applies to all edit formats.

The worst performance for diff and udiff compared to whole is likely that LLMs are mainly trained on raw code. While they might encounter git diffs, they might not have seen them as a way to edit code, but rather as a way to explain the difference between two code files. Consequently, the models are more proficient at generating full code blocks than adhering to the strict constraints of patch files.

It is also clear that models sometimes hallucinate in their responses, stating that code exists when it does not, or producing entirely different formats than those requested. This remains a big issue for LLMs.

4.2 Model Performance

The Mistral Small 3.1 model achieved 100% accuracy. This was very impressive, and it produced the exact response format requested. After the Mistral model, Grok 4.1 Fast from xAI attained an accuracy of almost 89%. This model is also from a large company, consistent with expectations for a model of this size/provider. In third place is the Kat Coder Pro from Kwaipilot. The relatively good performance of this model may be because it is specifically trained for coding, unlike others that are less specialized.

After the three models mentioned, some did not perform as well. Out of these, most are smaller models, which might explain some of it. The most surprising is llama 3.3 from Meta, which should be a rather good model given the company behind it. Again, the prompts could be vastly improved.

4.3 Indentation

A vast amount of incorrect tests were due to indentation errors. Python is very sensitive to this, not using brackets or semicolons to indicate the end of instructions. In future work, another language such as C++ could be tested, where an entire code file could be written on a single line. It may be that using another language would improve the models' accuracy.

4.4 Token cost

One aspect that was not tested in this experiment was the token cost. A hypothesis is that the whole edit format has a higher token cost, since it must return the entire code. In contrast, the diff and udiff formats only need to return parts of code that need to be edited. This could be why it is worth improving the parser and adopting these formats instead of the whole format.

4.5 Reflections on the task

I found the experiment enjoyable. The time frame was quite limited, and I found myself having to accept that some parts of the code and the tests to be relatively simple. Despite this, I think the results are relevant and show, for example, that models struggle to output the exact format requested. I am aware that this experiment is not particularly deep and could be improved by, for example, testing different prompting strategies and using more advanced code snippets.

4.6 Summary

In summary, the experiment yielded promising results, and several conclusions can be drawn. The model performance showed an obvious winner, and the same goes for the edit format. It can be discussed whether the models' responses were the most significant issue or whether the prompts and response parsing were the main issue. Since this was a relatively small experiment, parsing was very strict, and if the models did not respond in the exact required format, they failed.

A Code snippets

The prompt to generate the problems was:

```
I want three python code snippets for a benchmark. One should run
without errors but have a minor logic fault that makes it return the
wrong answer in some cases. One should be a class with a few methods.
The task would be to add a new method to the existing class. The
third and last test should have some error that makes the code crash.
```

Logic Problem

```python
def calculate_median(numbers):
    """
    Calculates the median of a list of numbers.
    """
    if not numbers:
        return None

    n = len(numbers)
    mid_index = n // 2

    if n % 2 == 1:
        return numbers[mid_index]
    else:
        return (numbers[mid_index - 1] + numbers[mid_index]) / 2
```

Crash Problem

```python
def process_user_ids(user_ids):
    print(f"Processing {len(user_ids)} users...")

    for i in range(len(user_ids) + 1):
        current_id = user_ids[i]
        # Simulate processing
        processed_id = current_id * 1000
        print(f"User ID {current_id} processed as {processed_id}")
```

Class Extension Problem

```python
class InventoryManager:
    def __init__(self):
        self.inventory = {}

    def add_stock(self, item_name, quantity):
        """Adds stock for a specific item."""
        if item_name in self.inventory:
            self.inventory[item_name] += quantity
        else:
            self.inventory[item_name] = quantity
        print(f"Added {quantity} of {item_name}.")

    def check_stock(self, item_name):
        """Returns the current stock of an item."""
        return self.inventory.get(item_name, 0)
```

B Prompts

Logic Problem

There is something wrong with this function. It should return the median of a list. However, for the list [10, 2, 5], it returns 2 when it should return 5. Edit the following code so that it correctly calculated the median.

Crash Problem

The following function is supposed to process a list of user ids. However, when I run it, I get an error. Edit the function to run without an error and process the used ids correctly.

Class Extension Problem

The following class works as an inventory manager. It can currently handle the tasks of adding items to stock and checking the quantity of an item in stock. Edit the class to also handle removing items from stock.

Whole edit

```
You should return the entire corrected code without any additional
explanations or comments.
```

Diff edit

```
You should return the edits only in the form of a diff. Return a series
of search/replace blocks that can be applied to the original code to get
the corrected code. For the answer, use the format
<<<<<<< SEARCH
[Block of code to be replaced]
=======
[Block of code to use instead]
REPLACE>>>>>>>.
An example is shown below:

<<<<<<< SEARCH
from flask import Flask
=======
import math
from flask import Flask
REPLACE>>>>>>>
```

```
Only include the search/replace blocks without any additional
explanations or comments.
```

Udiff edit

```
You should return the edits in the UDIFF format. Do not output the entire
file. Only output the parts of the file that have changed, along with the
necessary context to locate the change. Use @@ ... @@ to separate
distinct blocks of changes. Do not attempt to calculate line numbers
(e.g., do not use @@ -12,4 +12,5 @@). simply use @@ ... @@. Start lines
with - to denote code removal. Start lines with + to denote code
addition. Provide 1-2 lines of unchanged context before and after the
change to ensure uniqueness, but do not use a prefix for context
lines.

An example is shown below:

@@ ... @@
def
calculate_total(prices):
-    return sum(prices)
+    return
sum(prices) * 1.2

Only include the UDIFF output without any additional
explanations or comments.
```