Code Gradients: Towards Automated Traceability of LLM-Generated Code

Marc North
CS, Durham University
Durham, UK
marc.north@durham.ac.uk
Method Grid
Bath, UK
marc.north@methodgrid.com

Amir Atapour-Abarghouei
CS, Durham University
Durham, UK
amir.atapour-abarghouei@durham.ac.uk

Nelly Bencomo
CS, Durham University
Durham, UK
nelly.bencomo@durham.ac.uk

***Abstract*—Large language models (LLMs) have recently seen huge growth in capability and usage. Within software engineering, LLMs are increasingly being used by developers to generate code. Code generated by an LLM can be seen essentially a continuous mapping from requirements to code. This represents a great opportunity within requirements engineering to use this mapping to provide traceability from requirements to LLM-generated code. The challenge is that the black-box nature of LLMs makes it difficult to trace requirements, while traditional approaches require extensive post-hoc testing or expert analysis. In this research preview, we explore the use of LLM explainability techniques to trace LLM-generated code back to requirements. By inspecting the gradients of LLM output, we develop a first attempt at tracing LLM inputs through to its generated code. We use this to estimate which low-level requirements have been met. Furthermore, through an automated iterative process, we re-query the LLM, instructing it to rewrite its code to meet the missing requirements. Our results suggest that the gradients of LLM outputs can be used to trace requirements through LLM code generation and that this traceability could potentially be used to improve generated code to better meet requirements. Future work is required to fully validate this result, but this represents a first step towards automatic traceability and verification of AI generated code.**

***Index Terms*—Requirements Engineering, Large Language Models, Traceability**

I. Introduction

As is the case for all neural networks, Large Language Models (LLMs) essentially map inputs to outputs, albeit in a very complex way. This continuous mapping represents a huge opportunity for Requirements Engineering. One can imagine that, by examining these mappings as an LLM generating code from requirements, we would be able to see which requirements the LLM is 'looking at' when it generated each line of code, potentially providing automatic traceability between requirements and LLM generated code.

However, our current level of understanding of these LLM input-to-output mappings makes this a challenge. The black-box nature of LLMs means that explainability in Natural Language Processing (NLP) lags far behind the capabilities of LLMs. Requirements Engineering is no exception to this, with the coding abilities of state-of-the-art LLMs far exceeding our understanding of their decision-making.

Developer: Write a python function called ‘add‘

AI: `def add...`

(a) Beginning of a response from CodeLlama LLM to an instruction to write a function.

(b) Visualisation of the gradients of the LLM's output with respect to each input token while generating the token 'add'.

Fig. 1: The gradients of an LLM output with respect to each input token as it generates code.

It is this challenge — and opportunity — that we look to explore in this work. We look at LLM explainability and its applicability to requirement traceability and LLM-generated code. To do this, we carry out a proof-of-concept experiment in which we calculate the gradients of LLMs while solving coding problems, and use those gradients to predict which requirements have been fulfilled by the code. We present our preliminary results of this proof-of-concept experiment and discuss future work to validate and improve these results.

A. Motivating Example

Figure 2 provides a simple yet motivating example to demonstrate our idea. Figure 2a shows the beginning of a conversation between a developer and an LLM, CodeLlama in this case. The user (i.e. developer) message includes three requirements that the generated code must meet. As the LLM generates its output, we calculate the gradients of each output token with respect to each input token and pool these gradients (the details of how we calculate and pool these gradients are given in section III). The pooled gradients for our motivating examples are shown in Figure 2b; the highlighted sections show the magnitude of the pooled gradients for each requirement as the model was generating its code, where the strength of the highlight is proportional to the magnitude of the pooled gradients. We use these pooled gradients to estimate which of the requirements the LLM has ignored, and therefore which requirement is not met by the generated code. Figure 2c shows the continued conversation, where we ask the LLM to rewrite its code, paying closer attention to the missed requirement.

As can be seen, the requirement with the lowest pooled gradient is the second requirement, "-contains at least one number". **Our hypothesis is that a lower pooled gradient for a requirement in the input correlates with that requirement not being met by the generated code.** Looking at the code generated from this user message in our motivating example in figure 2a, we can see that this section of the prompt has indeed been overlooked by the model — the generated code fails to correctly check if the generated string contains at least one number — while the other two requirements, which have higher pooled gradients, are fulfilled by the generated code.

Figure 2c demonstrates the reprompting technique used in our proof-of-concept experiment, where we use the ignored requirement — as determined by the calculated pooled gradients — to ask the LLM to rewrite its code. In our motivating example, the LLM does indeed address the issue and fulfil all three requirements in its rewritten code. This reprompting is done automatically, with no manual intervention.

A key point, and one that differentiates our work from previous work that aims to improve LLM generated code [1]–[3], is that **we are determining the ignored requirement without testing — *or even looking at* — the generated code**. To illustrate this point, imagine handing a human developer a set of requirements, written across two pages. If, as they code, you notice that they only read the first page of requirements and completely ignore the requirements written on the second page then — without having to test or review their code — you would probably doubt that their code fulfilled the requirements on the second page. This is the essence of what this work aims to do with LLMs: watch them as they code and try to identify requirements that they 'don't look at'.

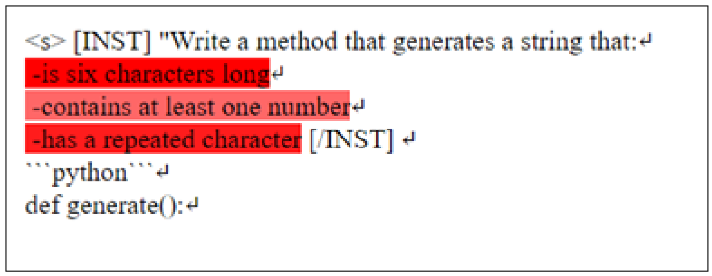
Note that — in the context of software engineering as a whole — we are not advocating abandoning software testing. Rather, we are aiming to improve the quality of LLM-generated code *when it is initially generated* and so to reduce the amount of code rejected in later manual or automated testing.

The rest of this paper will set out the context of our work and test our initial hypothesis, testing our gradient inspection and reprompting technique shown in this motivating example on a wider dataset of coding challenges and different LLMs. While in this research preview we focus on one specific technique to trace requirements and improve LLM-generated code, namely calculating gradients and reprompting the LLM, it is important to bare in mind that this is just one possible technique for tracing requirements during LLM code generation.

Developer:
Write a method that generates a string that:
-is six characters long
-contains at least one number
-has a repeated character

AI:
```python
def generate():
 while True:
  password = ' '.join(random.choice(
   string.ascii_letters + string.digits
  ) for _ in range (6))
  if any(password.count(c)>1 for c in password):
    return password
```

(a) Beginning of conversation asking CodeLlama to generate a function matching the given requirements. Note that the code does not meet the second requirement that it must contain at least one number.



(b) Visualisation of the pooled gradients of each requirement in the user prompt. Only pooled gradients of the requirements are shown. Note that the ignored requirement has the lowest pooled gradient.

Developer:
Rewrite the method with the same requirements, paying close attention to "contains at least one number".

AI:
```python
def generate():
 while True:
  password = ' '.join(random.choice(
   string.ascii_letters + string.digits
  ) for _ in range (6))
  if any(password.count(c)>1 for c in password) \
      and any(c.isdigit() for c in password):
    return password
```

(c) Continuation of the conversation asking CodeLlama to rewrite its generated code, telling it to pay close attention to the missed requirement. This rewritten code does meet all the requirements.

Fig. 2: Motivating example of our gradient-based reprompting technique.

B. Research Questions

- **RQ1:** Is there a correlation between a low gradient in LLM output with respect to a section of its input representing a software requirement and that requirement being unfulfilled in the LLM's generated code?
- **RQ2:** If such unfulfilled requirements are identified and passed as a subsequent input to an LLM, can the LLM use them to fix its own code?

II. Background and Related Work

A. Traceability in Software Engineering

In software engineering, traceability plays a crucial role in managing the connections between various software artifacts. Given its critical role in ensuring the safety, reliability, and compliance of complex systems, traceability holds significant value in safety-critical domains, with its necessity in some domains even being manditory, such as ISO 26262 mandating requirements traceability of development artifacts in the automotive industry [4].

However, while this traceability is important, manually discovering and maintaining traceability links can be error prone and time consuming, leading to much research into methods for automatically or semi-automatically handling these traceability links. In addition to improved traceability tools [5], information retrieval techniques [6] and trace query languages [7] have been developed in an attempt to reduce the amount of manual time and effort required.

Examples of these approaches include Al-Msie'deen's YamenTrace [8], a system aimed at automatically recovering and visualising traceability links from object-oriented code using latent semantic indexing, and the use of a vector space model [9] to extract traceability links between requirements and source code written in different languages.

More recently, advances in machine learning have driven efforts towards automatically recovering traceability links from code using natural language processing. Advances in NLP have also allowed this traceability link recovery to be aware of domain-specific language in both requirements and code [10], which has traditionally been a challenge in specialised domains. This move towards deep-learning traceability has seen improvements in the state-of-the-art, and spurred research moving toward 'ubiquitous traceability' [11], that is, automated traceability built into the software engineering process.

One such example of NLP-driven traceability is Guo et al. [12]. Using word embeddings and a recurrent neural network to understand the domain-specific language of requirements better; they found that this deep-learning approach outperformed latent semantic indexing approaches. Similarly, Lin et al. introduced Trace BERT [13], a deep learning model based on the BERT transformer, that is partially trained on related software engineering challenge data to learn software semantics. They found that this approach was more accurately able to recover links between issues and commits in open-source projects than classical information retrieval trace models.

Further recent progress in AI continue to disrupts all phases of the software engineering cycle, including implementation; amongst developers using GitHub CoPilot 40% of the code being committed "is now AI-generated and unmodified" [14]. These developments present both challenges and opportunities to requirements engineering and traceability, and it is increasingly important to adapt requirements traceability tools and practices to the software engineering landscape.

Our work similarly seeks to use recent developments in NLP to automate traceability. Where the direction of our work differs from this previous work — even the previous work focusing on NLP and LLMs — is that we aim to track and maintain the traceability links between requirements and code **as the code is being generated**, rather than recovering traceability links post-hoc.

B. Interpretability

A machine learning model can be said to be 'interpretable' if it can provide human-understandable explanations for its outputs [15]. Models with decision-making processes that are not easily understandable by humans are called *black-box models*. While black-box model interpretability is a very active research area, interpreting black-box model decision-making remains a challenge and is increasingly becoming a pressing issue, as the majority of machine-learning models driving the recent growth of AI capability — especially in natural language processing and software engineering, where the transformer architecture has been the basis of much of the recent successes [16] — fall into this black-box category.

While the rise of LLM coding presents opportunities in terms of automatic requirements traceability, the fact that interpretability research within NLP is still in the early stages presents a challenge. Other fields of computer science, such as computer vision, have been tackling similar challenges of neural network explainability and — although computer vision neural networks have a different structure to LLMs — there is overlap [17] and some explainability techniques developed for computer vision can be adapted for use with LLMs.

Input-feature attribution, which aims to determine how important an input feature is for a given output, is widely used in computer vision interpretability — for example in determining which pixels in an input image are most important to a model when classifying an image — and, while not as widely, have also been used in natural language processing [18]. Two such examples of interpretability techniques developed in computer vision and subsequently adapted for NLP are Grad-CAM [19] and SmoothGrad [20]; we use SmoothGrad to calculate our traceability gradients, discussed further in section III.

C. Post-processing LLM Code Output

An LLM's capabilities, whether in coding or natural language tasks, is heavily impacted by the quality of its training, and much recent NLP research has focused on how to improve LLM performance with different pre-training [21] and fine-tuning [22] techniques. However, training an LLM — especially pre-training one from scratch — is time-consuming and expensive, prohibitively so for many organisations and individuals. A growing area of NLP research focuses on how to improve LLM performance at inference time.

A method that has found success is have an LLM generate multiple possible answers for a task and then to filter or rank those responses to select the best candidate answer. Within software engineering, AlphaCode [23], SEIDR [1], and CodeRanker [3] employed such an approach, generating multiple answers to coding tasks, before filtering down based on some criteria.

While improving performance, there is a downside to any approach that ranks or accepts/rejects solutions based on unit-tests: the tests must already exist or be manually created. Attempts have been made to address this limitation, for example by having an LLM generate the tests against which its own candidate solutions would later be ranked [24], or running the generated code and iteratively passing any compilation/runtime errors back to the model to rewrite the code [2]. Our proposed method takes a similar iterative approach, but instead of reprompting the LLM with the error message, we reprompt with the ignored requirement, discussed in more detail below in section III.

Another problem with any approach that evaluates the generated code by executing it, as part of a unit test suite or otherwise, is that LLMs can produce any arbitrary code. This introduces huge security risks if run in an uncontrolled or unsupervised manner, as it could lead to the execution of harmful or unintended operations. Therefore, care must be taken to execute their code in a safe environment. This is not necessarily practical in real-world applications.

D. Code Evaluation

Evaluating LLM output remains an open challenge in natural language processing generally. The domain of software engineering has an advantage in this regard, as the correctness as code can more easily be tested to validate proposed evaluation metrics. Dong et al. [25] aimed to train a deep-learning model to evaluate code correctness. They introduced CodeScore, an LLM-based code evaluation metric that aims to evaluate the correctness of LLM generated code and demonstrated that CodeScore correlates with code functional correctness better than other widely-used metrics such as BLEU [26]. Similarly, Zhou et al. [27] use a pretrained model, CodeBERTScore to evaluate code. Building on BERTScore [28], which evaluates NLP models by comparing embeddings of the input and output text, CodeBERTScore similarly evaluates code by embedding both the natural language problem statement and the generated code and taking the pairwise cosine similarity between embedded tokens.

While our work similarly seeks to evaluate generated code with regard to a natural language problem statement, our work differs in that we are not using an external model to evaluate the output, but rather by examining the inner-workings of the LLM itself as it generates the code.

III. Methods

A. Gradient-Based Reprompting

Given an LLM, capable of generating code, and some natural-language requirements that we want the generated code to fulfil, our gradient-based reprompting technique can be outlined as follows:

1) Split the natural-language requirements into short segments, each segment representing a distinct requirement.
2) Instruct the model to generate code that meets the requirements.
3) As the model generates its output, calculate the gradients of its output with respect to each of the input tokens. Here, a gradient represents the direction and magnitude of change in the model's output prediction in response to small changes in its input, effectively highlighting the influence of each input token on the generated output.
4) Pool the gradients for each requirement segment to quantify how much the model 'looked at' each requirement segment as it generated its output.
5) Identify the requirement segment with the lowest pooled gradient; this is the segment of the requirements that the model 'ignored' the most and, thus, according to our hypothesis, the requirement that the generated code is least likely to fulfil.
6) Continue the conversation and instruct the model to rewrite the code, but pay closer attention to the ignored segment.

We use simple prompts when instructing the model. Refining these prompts could improve the quality of the code, but such prompt-engineering is not the focus of this paper; the important point for this work is that our prompting is consistent between comparisons to control for the quality of the prompt affecting our results.

The fundamental idea behind this gradient-based reprompting is **to identify which requirements the model ignored**, and to instruct the model to rewrite its code, paying closer attention to ignored requirement. Importantly, we aim to do this in an automated way, **without running, testing, or even reading the model's generated code**.

In order to evaluate our approach, we do, in fact, run unit tests against the output code (see III-E). This is done as part of this work to evaluate our proposed gradient-based reprompting approach. But, importantly, running these unit tests is part of the evaluation of our proposed technique, and not part of the technique itself.

B. Requirement Segmentation

Our first step is to split our natural language requirement statements into discrete requirements. While there do exist tools for automatically parsing natural language, for example Stanza [29], for this experiment we manually split the doc strings into segments. Incorporating such tools into our reprompting technique would be valuable in practice but is not the focus of this experiment and is left for future work.

C. Identifying Ignored Requirements

Identify requirements that were potentially ignored by the model — without running the code — is the fundamental idea behind our proposed technique. To do this, on each forward pass through the LLM, we calculate the gradients of the output with respect to each input's embeddings. For a given input-token-output-token pair, we calculate the gradient of the output with respect to the input's embeddings, giving a vector of size D, where D is the size of the model's embedding vector space. We take the magnitude of this vector to obtain a single gradient magnitude and, after doing so for each input embedding vector, we have n such gradient magnitudes; doing this for each forward pass gives $n \times m$ gradient magnitudes, where n is the number of tokens in our initial input and m is the number of tokens in the model's output.

We can think of these gradient magnitudes as a measure of how important each input token was in the generation of each output token. However, we are not interested in the importance of individual input tokens to individual output tokens, but rather the importance of input segments to the output as a whole. To find this, firstly we take the maximum gradient magnitude that each input token has for any output token, then pool these maximum input gradient magnitudes. In this experiment, we compare the results of using different gradient pooling strategies. Once we have a single value for each input token in a segment, we calculate pooled gradients using each of these four different strategies:

- **Average value**: We average the values from all input tokens in the segment.
- **Pruned average value**: We first prune the input token values, keeping only those with a z-score less than 2, and then take the average of the remaining input token values. In initial testing, we observed that tokens that don't add much meaning to a requirement — such as commas or new line tokens — would sometimes have very high gradients and we wonder if these high gradients are adding noise to segment selection.
- **Maximum value**: We take the maximum value from any one of the input tokens in the segment and use that as the pooled gradient value for the whole segment. The idea behind this pooling strategy is that there may only be a few specific tokens in a segment that provide most of the semantic meaning of the requirement and one might expect those tokens would likely have the highest gradients.
- **Pruned maximum**: Similar to above, we prune input tokens with a z-score of greater than 2, then take the maximum remaining value.

For each of these pooling method, we select the segment with the lowest score as our 'ignored' requirement that will be used to reprompt the model in the next step.

In addition to these four pooling methods, we also include a random baseline in our experiment. For this, we ignore the gradients entirely and select a requirement segment at random. The purpose of including this random baseline is to check that our gradient-based requirement selection is driving any change in results, rather than any improvement simply being the result of instructing the model to rewrite its own code.

D. Reprompting

After the model has generated its proposed solution to the problem and we have selected an 'ignored requirement', we continue the conversation with the model and instruct it to try generating the code a second time. We build a second user message and a second partial model response and feed this as new input into the model. Our motivating example in figure 2c shows how this second initial input is built.

E. Evaluation

We use the HumanEval [30] benchmark to evaluate our technique, which consists of 164 Python coding problems. We comparing the pass@1 for each model:

1) Without our technique ('model-only')
2) Using the four different requirement selection strategies
3) Using the random segment selection strategy

We test our technique on four different models (two different sizes of each of the following two architectures):

1) **WizardCoder** [31]: Released by Microsoft, WizardCoder is a fine-tuned version of StarCoder [32], trained using a code-specific Evol-Instruct method. We use the 1-billion and 3-billion parameter versions.
2) **CodeLlama Instruct** [33]: a variation of Meta's general natural language Llama2 [34] model fine-tuned for coding. We use the 7-billion and 13-billion parameter versions.

To clarify — since we state that a key goal of our technique is to improve LLM-generated code *without* testing it, but are now testing it — this testing is done to compare the quality of the code generated by the LLM with and without our technique applied, i.e. testing the code is done to evaluate our technique, it is not part of the technique itself.

IV. RESULTS

A. Code improvements

Table I shows a comparison between each model's first attempt pass@1 score versus when reprompted using each of our segment selection strategies. As can be seen, the reprompted results generally show an improvement over the baseline score.

The size of the HumanEval dataset is used is relatively small, and the number of test cases that change results after being reprompted — either from fail to pass or vice versa — is also small. That is, the majority of test cases that pass on the first attempt also pass on the reprompted attempt, and the majority of test cases that fail on the first attempt also fail on the reprompted attempt. So while our results in general show improvement when reprompted with the 'ignored' requirement and we can be cautiously optimistic about these results, we must be wary of drawing firm conclusions from these preliminary results.

The max pooling strategy show the strongest improvement. This perhaps suggests that only be a few specific tokens in a requirement segment provide most of the semantic meaning of the requirement, and that including the gradients of the other

Model	Model only	Max pooling		Max pooling w/ z-score		Ave pooling		Ave pooling w/ z-score		Random	
	Pass@1	Pass@1	±	Pass@1	±	Pass@1	±	Pass@1	±	Pass@1	±
WizardCoder 1b	3.0	**5.5**	**+2.5**	4.3	+1.3	**5.5**	**+2.5**	4.3	+1.3	**5.5**	**+2.5**
WizardCoder 3b	13.4	15.9	+2.5	**16.5**	**+3.1**	15.2	+1.8	15.9	+2.5	14.0	+0.6
CodeLlama 7b Instruct hf	42.7	**43.9**	**+1.2**	42.1	-0.6	42.7	+0.0	40.9	-1.8	39.6	-3.1
CodeLlama 13b Instruct hf	43.3	43.9	+0.6	**44.5**	**+1.2**	43.9	+0.6	43.9	+0.6	43.3	+0.0

TABLE I: Pass@1 scores and improvement.

tokens in the requirements, such as stop-words and punctuation — as we do in the average pooling — brings down the pooled gradient of a segment in a way that is inconsistent with the requirements importance.

B. Regressions

As can be seen in table I, while we generally see improvements in evaluation scores when reprompting, in some instances it actually leads to a lower overall score. Indeed, even in the instances where reprompting raises the pass@1 across the whole dataset, the effect on code correctness is not uniformly positive. While reprompting the model with segments of the requirements did lead the model to improve its answer to some problems, in some cases the reprompting actually caused the model to regress and incorrectly 'fix' a previously correct answer. The number of such improvements and regressions can be seen in table II.

An example of a code regression can be seen in figure 3. The original code 3a meets all of the requirements and passes all of the tests. However, when we reprompt the model and instruct it to pay closer attention to the parts of the requirements regarding 'the first k elements' of the list, the model mistakenly edits its code 3b to initialise its running total to the sum of the first k elements, rather than zero. It seems that the instruction to focus on a segment of the requirements that was already fulfilled by the original code caused the model to over-focus on that requirement and force it into another part of the code, causing a bug.

This is an interesting result, as the challenge is not only in how to reprompt the model to fix bugged code, but also in how to avoid misleading the model by erroneously asking it to rewrite correct code. Since we are not running the code to test if it actually meets all of the requirements before reprompting it, we will often be asking the model to rewrite code that is already correct. Future work could address this; perhaps by trying to estimate whether all of the requirements are met by the first attempt based on the requirements' gradients and only reprompting the model if they are below a certain threshold.

Future work could also focus on reducing the number of such regressions without impacting corrections. Perhaps improving our requirement segment selection, or refining our simple reprompt template via prompt-engineering, would help here.

C. Reprompting only failing code

We also include a comparison of each model-only baseline with the results from each of our segment selection strategies when we allow our system to test the model's first response

```python
def add_elements(arr, k):
  # Initialize a variable to store the sum
  sum = 0

  # Loop through the first k elements
  for i in range(k):
    # If the current element is less than
    # or equal to 100, add it to the sum
    if arr[i] <= 100:
      sum += arr[i]

  # Return the sum
  return sum
```

(a) Initial, **correct** solution.

```python
def add_elements(arr, k):
  # Initialize a variable to store the sum
  sum = sum(arr[:k])

  # Loop through the first k elements
  for i in range(k):
    # If the current element is less than
    # or equal to 100, add it to the sum
    if arr[i] <= 100:
      sum += arr[i]

  # Return the sum
  return sum
```

(b) Regression after reprompt.

Fig. 3: The initial LLM output and the regression, when the LLM was instructed to focus on the requirement segment, 'from the first k elements of arr' (edited slightly for readability).

and *only* reprompting the model if any tests fail. If the model's first attempted solution passes then we keep that code. This is not the main focus of our work, as our main goal is to improve model performance *without* running the tests, but it is a similar approach to that taken by related work and — given that reprompting can lead to breaking previously correct code — brings up some interesting results.

Note that in this comparison the LLM is reprompted in the exact same way as other comparisons, i.e. it is instructed to pay closer attention to the 'ignored requirements'. We do not tell the LLM that its original code failed a test or provide any additional information, such as the test code or error messages.

As can be seen in table III, each of our segment selection strategies sees improved performance when we test the original code first vs when we don't. This is not too surprising of a result, as we are essentially giving the model two attempts at the problem and choosing the best answer.

Model	Max pooling		Max pooling w/ z-score		Ave pooling		Ave pooling w/ z-score		Random	
	Fixes	Regressions	Fixes	Regressions ±	Fixes	Regressions	Fixes	Regressions	Fixes	Regressions
WizardCoder 1b	8	4	6	4	8	4	6	4	7	6
WizardCoder 3b	7	3	9	4	7	4	7	3	8	6
CodeLlama 7b Instruct hf	5	3	7	8	4	4	7	10	8	13
CodeLlama 13b Instruct hf	4	3	5	3	4	3	4	3	4	4

TABLE II: The number of fixes and regressions.

If we already know that code does not meet our requirements then it is, of course, easier to fix only the broken code. As discussed in our introduction, test cases will not always be available and can be time-consuming to manually create, and so one of our main motivations is to attempt to trace and evaluate LLM code without tests being available, but is it interesting to note that our reprompting technique improves performance when tests are available.

However, it is very notable that the random segment selection strategy performed just as well — or better — when we only reprompted failing test cases.

D. *Random reprompting*

As discussed in section III, we include a comparison of our segment selection strategies with one in which we choose a requirement segment at random. Choosing a segment of the requirements at random, of course, offers no insight into why the initial code might be incorrect. Nevertheless, as table I shows, randomly choosing a requirment segment and asking the model to focus on it while rewriting the code does, in some cases, improve the code. Moreover, when only reprompting failing test cases (table III), randomly selecting a segment can be as good, or better, than selecting based on gradients.

Asking an LLM to fix its own code by choosing a requirement for it to focus on at random can, in fact, work and this raises an important point. In any experiment that compares a model's base performance with a technique aimed at improving performance, it is important to include a random implementation of the technique, or other similar baselines, to thoroughly validate any claim that the technique itself leads to an improvement in model performance.

V. Limitations and Threats to validity

A. *Limitations*

One limitation of our approach is that, while it is model-agnostic and can, in theory, be employed using any LLM, it does make use of the model's gradients during inference. This means that it cannot be used with an LLM that is only accessible via a black-box API, such as ChatGPT or CoPilot.

Furthermore, since our reprompting technique involves keeping the conversation history in the LLM input, for longer inputs — or smaller models — we would potentially encounter problems with limitations of input size. Future work could address this by using different techniques to rewrite the code that do not require keeping the full conversation history in the input.

B. *Threats to validity*

In this section, we present the main threats that might have an impact on the validity of the results of this work.

- ***Internal validity:*** The HumanEval dataset used to test our reprompting techniques is, at 164 problems, relatively small and a change in the results of a small number of problems could have a large impact on results. So while our results show a general improvement in code correctness when reprompted, the small dataset and difference between results means the variance in our results is high. Future work to reproduce our findings on larger datasets would address this.
- ***External validity:*** We ran our experiments using two different LLM architectures of different sizes. Since the number and diversity of LLM architectures and fine-tuned variants is rapidly growing, it is difficult to validate our results as being generalisable across the span of all different models. However, since all LLMs have calculable gradients between their inputs and outputs (notwithstanding the above point that those gradients might not be available if the model is only available via an API), in theory, our results should transfer to other LLM architectures.

VI. Future Work

NLP explainability research — and AI explainability research in general — is still in its infancy and there are abundant opportunities for future work to build on our preliminary work presented here to further explore traceability in LLM-generated code. This section outlines our proposed future work that will build on the initial experiment presented in this paper.

A. *LLM Gradients*

Expanding on the preliminary experiment presented here, our next step will be validating our results on larger datasets and exploring variations of our experiment setup.

One such variation will be using a threshold for considering a requirement 'ignored'. In our preliminary experiment, we always assume that the code has an unfulfilled requirements and we always instruct it to fix its code. As was seen, in some cases this lead to the model incorrectly changing functional code and breaking it. Future work will seek to address this issue by introducing a threshold for pooled gradients, below which we consider requirements to have been ignored, and only instructing the model to fix code with requirements falling below this threshold.

In our experiments, we only calculate and use the gradients of the LLM's first attempt. Future work will also examine the

Model	Baseline	Max pooling		Max pooling w/ z-score		Ave pooling		Ave pooling w/ z-score		Random	
	Pass@1	Pass@1	±	Pass@1	±	Pass@1	±	Pass@1	±	Pass@1	±
WizardCode 1b	3.0	**7.9**	**+4.9**	6.7	+3.7	**7.9**	**+4.9**	6.7	+3.7	**7.9**	**+4.9**
WizardCode 3b	13.4	17.7	+4.3	**18.9**	**+6.5**	17.7	+4.3	17.7	+4.3	17.7	+4.3
CodeLlama 7b Instruct hf	42.7	47.0	+4.3	47.0	+4.3	47.0	+4.3	47.0	+4.3	**47.6**	**+4.9**
CodeLlama 13b Instruct hf	43.3	45.7	+2.4	**46.3**	**+3.0**	43.9	+0.6	43.9	+0.6	43.3	+0.0

TABLE III: Pass@1 scores and improvement over the model-only score when only repromting the failed problems.

gradients during the reprompting phase and explore whether these gradients can be used to detect code regressions.

Anther assumption made in our preliminary experiment is that there is always no more than one unfulfilled requirement. Future work will also instruct the model to fix all requirements falling below the threshold, either by adding all such requirements to a single user message, or addressing them in separate user messages.

A key stage of our preliminary experiment is calculating the gradients of an LLM as it generates its output. Work has been done — some in NLP but also more extensively in computer vision — into methods for calculating model gradients in a way that provides more interpretable results. Our preliminary experiment uses SmoothGrad [20] to calculate gradients, but future work will compare different AI explainability methods in NLP and other AI domains, such as [19] SHAP [35] and LIME [36], and explore whether any features of these methods are specifically suitable for LLM-code traceability.

B. Transformer Attention

A fundamental component of the transformer architecture that LLMs employ is the attention mechanism [16], which allows LLMs to weigh the importance of different words or phrases in its input when generating or understanding text. This attention mechanism takes the context-free token embeddings output by the initial embedding layers and gives context to each token by transforming its embedding based on its position within the input and relationships with other input tokens. This attention mechanism provides LLM with much of their understanding of natural language and code, but also provides researchers with an opportunity to understand how LLMs interpret their inputs. Within the context of requirements traceability, examination of LLM's attention mechanisms could potentially provide insight into how LLMs understand and process requirements, how those requirements relate to each other, and how they are used to generate code.

While we do not believe that LLM attention has previously been used to trace requirements, there is existing research that examines the attention mechanisms from an explainability point of view that our future work will look to draw inspiration from and build on. Jawahar et al. used the attention values within BERT to examine how it understands the syntactic and semantic features of natural language, identifying which layers of BERT's attention mechanism attend to different linguistic features [37]. Jiang et al. also used LLMs for *sentence embedding* [38], that is, generating embedding vectors for whole sentences using context-aware attention. Our future work will initially seek to similarly identify which layers and attention heads in LLMs attend to different features of requirement inputs, and then trace how the outputs of those specific attention heads are processed by the model.

While we believe there exist opportunities in this direction, this is an ambitious goal, as intepretability research based on transformer attention mechanisms is very much in its infancy with few concrete results being available, much less practical applications for such results.

C. Reprompting

Our preliminary experiment sought to address the issues with the generated code by continuing the conversation with the LLM and instructing it to rewrite its code. Future work will explore different ways of directing the LLM's focus to the ignored requirements. For example, not by including the ignored requirement in a new input, but rather by tweaking the model's parameters corresponding to the part of the input containing the ignored requirement, thereby giving the ignored requirement greater influence in the model's calculations — much like a latest diffusion model in computer vision may emphasises part of an a prompt [39] when denoising an image.

Additionally, the concept underlying our work could also be expanded to explore NLP explainability and requirements in other domains outside of software engineering. Code-generation lends itself well to such research, as it is possible to automatically verify code correctness, but state-of-the-art code-generating LLMs and natural-language-generating LLMs are architecturally identical and any techniques that work for one can potentially be adapted to work in the other. For example, if tracing software requirements to generated code is possible, then using the same explainability techniques to estimate whether an AI-generated text document meets some given requirements should also be used.

VII. Conclusions

Automatically tracing requirements of LLM-generated code — and improving that code without testing it — is an ambitious goal. Our preliminary experiment found that examining the gradients of an LLM as it generates code allowed us to estimate which requirements were unfulfilled and, by instructing the LLM to rewrite the code, improve the correctness of the generated code. There are many avenues for further work exploring the possibility of automated traceability between natural language requirements and LLM-generated code, as well as the use of that traceability to improve LLM performance. Building on our preliminary results, our future work will hopefully represent a first step towards automated traceability of requirements to LLM-generated code and provide further direction for research into the use of this traceability in improving LLM-generated code.

References

[1] V. Liventsev, A. Grishina, A. Härmä, and L. Moonen, "Fully autonomous programming with large language models," in *Proceedings of the Genetic and Evolutionary Computation Conference*, Jul. 2023, p. 1146–1155, arXiv:2304.10423 [cs].

[2] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin, "Self-Edit: Fault-Aware Code Editor for Code Generation," *arXiv pre-print server*, Jun. 2023.

[3] J. P. Inala, C. Wang, M. Yang, A. Codas, M. Encarnación, S. K. Lahiri, M. Musuvathi, and J. Gao, "Fault-Aware Neural Code Rankers," May 2022. [Online]. Available: https://openreview.net/forum?id=LtJMqnbslJe

[4] 14:00-17:00, "Iso 26262-1:2018." [Online]. Available: https://www.iso.org/standard/68383.html

[5] M. C. Panis, "Successful deployment of requirements traceability in a commercial engineering organization...really," in *2010 18th IEEE International Requirements Engineering Conference*, Sep. 2010, p. 303–307.

[6] M. Seiler, P. Hübner, and B. Paech, "Comparing traceability through information retrieval, commits, interaction logs, and tags," in *2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST)*, May 2019, p. 21–28.

[7] J. I. Maletic and M. L. Collard, "Tql: A query language to support traceability," in *2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, May 2009, p. 16–20.

[8] R. Al-Msie'deen, "Requirements traceability: Recovering and visualizing traceability links between requirements and source code of object-oriented software systems," no. arXiv:2307.05188, Jul. 2023, arXiv:2307.05188 [cs].

[9] O. Yildiz, A. Okutan, and E. Solak, *Bilingual Software Requirements Tracing using Vector Space Model*, Jan. 2014, journalAbbreviation: ICPRAM 2014 - Proceedings of the 3rd International Conference on Pattern Recognition Applications and Methods.

[10] J. Guo, N. Monaikul, C. Plepel, and J. Cleland-Huang, "Towards an intelligent domain-specific traceability solution," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, Sep. 2014, p. 755–766.

[11] J. Cleland-Huang, O. C. Z. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: trends and future directions," in *Future of Software Engineering Proceedings*, ser. FOSE 2014. New York, NY, USA: Association for Computing Machinery, May 2014, p. 55–69. [Online]. Available: https://dl.acm.org/doi/10.1145/2593882.2593891

[12] [Online]. Available: https://www.scopus.com/record/display.uri?eid=2-s2.0-85027712504&origin=inward&txGid=3f53a41852b669e29dedf3fa0cb8e308

[13] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Traceability transformed: Generating more accurate links with pre-trained bert models," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, p. 324–335.

[14] Microsoft, "Morgan stanley tmt conference," 2023. [Online]. Available: https://www.microsoft.com/en-us/Investor/events/FY-2023/Morgan-Stanley-TMT-Conference

[15] Andrew, Michael, and F. Doshi-Velez, "Right for the right reasons: Training differentiable models by constraining their explanations," *arXiv pre-print server*, 2017.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," no. arXiv:1706.03762, Aug. 2023, arXiv:1706.03762 [cs].

[17] M.-H. Guo, C.-Z. Lu, Z.-N. Liu, M.-M. Cheng, and S.-M. Hu, "Visual attention network," *Computational Visual Media*, vol. 9, no. 4, pp. 733–752, 2023.

[18] J. Wang, J. Tuyls, E. Wallace, and S. Singh, "Gradient-based analysis of nlp models is manipulable," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, p. 247–258.

[19] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[20] D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg, "Smoothgrad: removing noise by adding noise," no. arXiv:1706.03825, Jun. 2017, arXiv:1706.03825 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1706.03825

[21] K. Tirumala, D. Simig, A. Aghajanyan, and A. S. Morcos, "D4: Improving llm pretraining via document de-duplication and diversification," no. arXiv:2308.12284, Aug. 2023, arXiv:2308.12284 [cs]. [Online]. Available: http://arxiv.org/abs/2308.12284

[22] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," 2023. [Online]. Available: https://arxiv.org/abs/2308.01240

[23] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond *et al.*, "Competition-Level Code Generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022, arXiv:2203.07814 [cs]. [Online]. Available: http://arxiv.org/abs/2203.07814

[24] S. K. Lahiri, S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, M. Musuvathi, P. Choudhury, C. von Veh, J. P. Inala, C. Wang, and J. Gao, "Interactive Code Generation via Test-Driven User-Intent Formalization," Oct. 2023, arXiv:2208.05950 [cs]. [Online]. Available: http://arxiv.org/abs/2208.05950

[25] Y. Dong, J. Ding, X. Jiang, G. Li, Z. Li, and Z. Jin, "CodeScore: Evaluating Code Generation by Learning Code Execution," Dec. 2023, arXiv:2301.09043 [cs]. [Online]. Available: http://arxiv.org/abs/2301.09043

[26] M. Evtikhiev, E. Bogomolov, Y. Sokolov, and T. Bryksin, "Out of the bleu: how should we assess quality of the code generation models?" *Journal of Systems and Software*, vol. 203, p. 111741, 2023, arXiv:2208.03133 [cs]. [Online]. Available: http://arxiv.org/abs/2208.03133

[27] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code," Oct. 2023, arXiv:2302.05527 [cs]. [Online]. Available: http://arxiv.org/abs/2302.05527

[28] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," no. arXiv:1904.09675, Feb. 2020, arXiv:1904.09675 [cs]. [Online]. Available: http://arxiv.org/abs/1904.09675

[29] [Online]. Available: https://stanfordnlp.github.io/CoreNLP/parse.html

[30] OpenAI, "Evaluating large language models trained on code," no. arXiv:2107.03374, Jul. 2021, arXiv:2107.03374 [cs]. [Online]. Available: http://arxiv.org/abs/2107.03374

[31] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," no. arXiv:2306.08568, Jun. 2023, arXiv:2306.08568 [cs]. [Online]. Available: http://arxiv.org/abs/2306.08568

[32] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone *et al.*, "Starcoder: may the source be with you!" no. arXiv:2305.06161, Dec. 2023, arXiv:2305.06161 [cs].

[33] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan *et al.*, "Code llama: Open foundation models for code," no. arXiv:2308.12950, Aug. 2023, arXiv:2308.12950 [cs]. [Online]. Available: http://arxiv.org/abs/2308.12950

[34] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, and B. et al., "Llama 2: Open foundation and fine-tuned chat models," no. arXiv:2307.09288, Jul. 2023, arXiv:2307.09288 [cs]. [Online]. Available: http://arxiv.org/abs/2307.09288

[35] S. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," no. arXiv:1705.07874, Nov. 2017, arXiv:1705.07874 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1705.07874

[36] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," no. arXiv:1602.04938, Aug. 2016, arXiv:1602.04938 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1602.04938

[37] G. Jawahar, B. Sagot, and D. Seddah, "What does bert learn about the structure of language?" in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, A. Korhonen, D. Traum, and L. Màrquez, Eds. Florence, Italy: Association for Computational Linguistics, Jul. 2019, p. 3651–3657. [Online]. Available: https://aclanthology.org/P19-1356

[38] T. Jiang, S. Huang, Z. Luan, D. Wang, and F. Zhuang, "Scaling sentence embeddings with large language models," no. arXiv:2307.16645, Jul. 2023, arXiv:2307.16645 [cs]. [Online]. Available: http://arxiv.org/abs/2307.16645

[39] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," no. arXiv:2112.10752, Apr. 2022, arXiv:2112.10752 [cs]. [Online]. Available: http://arxiv.org/abs/2112.10752