Benchmark Dataset Generation and Evaluation for Excel Formula Repair with LLMs

Ananya Singha*
Microsoft
ananyasingha@microsoft.com

Harshita Sahijwani*
Microsoft
hasahijwani@microsoft.com

Walt Williams
Microsoft
walwilliams@microsoft.com

Emmanuel Aboah Boatang
Microsoft
emmanuelab@microsoft.com

Nick Hausman
Microsoft
nhausman@microsoft.com

Miguel Di Luca
Microsoft
migueldiluca@microsoft.com

Keegan Choudhury
Microsoft
kechoudhury@microsoft.com

Chaya Binet
Microsoft
Chaya.Leiser@microsoft.com

Vu Le
Microsoft
levu@microsoft.com

Tianwei Chen
Microsoft
tianweichen@microsoft.com

Oryan Rokeah Chen
Microsoft
orokeahchen@microsoft.com

Sulaiman Vesal
Microsoft
svesal@microsoft.com

Sadid Hasan
Microsoft
sadidhasan@microsoft.com

Abstract

Excel is a pervasive yet often complex tool, particularly for novice users, where runtime errors arising from logical mistakes or misinterpretations of functions pose a significant challenge. While large language models (LLMs) offer promising assistance by explaining formula errors, the automated correction of these semantic runtime errors remains an open problem. A primary challenge to advancing models for such scenarios is the severe lack of high-quality, comprehensive datasets for training and rigorous evaluation. This paper addresses this gap by introducing a novel approach for constructing a benchmark dataset specifically designed for Excel formula repair. We propose a data generation pipeline, which leverages a small set of curated seed samples from online forums to synthetically expand the dataset. Our pipeline integrates few-shot prompting with LLMs and employs a robust *LLM-as-a-Judge* validation framework, combined with execution-based checks to ensure the correctness and semantic fidelity of the generated data. This process produced a benchmark dataset of 618 high-quality samples, covering common runtime errors. Furthermore, we propose a context-aware baseline technique for Excel formula repair that utilizes LLMs to leverage both the faulty formula, and relevant spreadsheet context. We evaluate the performance of various LLMs (GPT-4o, GPT-4.1, Phi-3, Mistral) on our newly generated benchmark using execution-based metrics. Our analysis demonstrates the dataset's quality through manual annotation and provides insights into error and function distributions. The proposed generation methodology is highly scalable and can be readily adapted to create evaluation benchmarks for similar code repair tasks in other low-resource programming languages.

Keywords

Synthetic Data Generation, Large Language Models, Formula Repair

ACM Reference Format:
Ananya Singha, Harshita Sahijwani, Walt Williams, Emmanuel Aboah Boatang, Nick Hausman, Miguel Di Luca, Keegan Choudhury, Chaya Binet, Vu Le, Tianwei Chen, Oryan Rokeah Chen, Sulaiman Vesal, and Sadid Hasan. 2025. Benchmark Dataset Generation and Evaluation for Excel Formula Repair with LLMs. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (KDD '25).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Spreadsheets (e.g., Microsoft Excel, Google Sheets) are among the most widely used end-user programming platforms, with hundreds of millions of users worldwide [2, 16]. They empower users—often without formal programming backgrounds—to manipulate and analyze data through formulas composed on a tabular grid. However, writing correct and robust formulas remains a significant challenge. Small mistakes such as incorrect cell references, missing arguments, or improper function nesting can break computations or lead to incorrect results. These errors may surface as syntax problems, logical bugs, or runtime failures (e.g., `#DIV/0!`, `#REF!`), and diagnosing

*Both authors contributed equally to this research.

KDD '25, Toronto, ON

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

them is often non-trivial for non-programmers [12]. This highlights the need for tools that assist users in automatically detecting and fixing errors in their formulas.

Automated Program Repair (APR) has been extensively studied for general-purpose programming languages (GPLs) like Java, Python, and C++ [9, 20]. These systems typically rely on well-structured code with modular functions and comprehensive test suites, which support static analysis and test-driven repair strategies. In contrast, spreadsheet formulas present unique challenges. They operate over structured data objects (e.g., cell ranges, tables) and often lack modular abstractions or formal specifications. Writing correct formulas requires not just syntactic fluency but also a clear understanding of how to reference and manipulate tabular data. Errors often arise from misinterpreting layouts, incorrect argument choices, or misunderstanding function semantics.

Prior work on APR in spreadsheets (henceforth referred to as *Excel formula repair*) has largely focused on syntactic repair, often without leveraging the spreadsheet context. For example, LaMirage [3] uses grammar-based candidate generation and a neural ranker to fix syntax errors. A more recent system, RING [4] applies prompting techniques and error localization to formula text to suggest potential repairs to formula errors. FLAME [15], a domain-specific fine-tuned model has shown impressive performance on formula synthesis using compact transformers trained on Excel-specific corpora. However, none of these approaches consider tabular data. Also, most of these approaches remain focused on syntax repair, not semantic correctness.

Importantly, our analysis of real-world user queries on forums like Stack Overflow and Excel support channels reveals that a majority of formula issues are semantic rather than syntactic. These issues often manifest as runtime errors and require context-aware reasoning over both formula logic and spreadsheet data for effective resolution. Current Excel formula repair approaches do not address runtime errors. Moreover, existing datasets typically include only isolated formula pairs—incorrect and corrected—but omit the spreadsheet context necessary for modeling these semantic repair tasks. There isn't an existing dataset that can be used to train and evaluate a repair model for runtime errors.

Our Contributions. To address this gap, we introduce FoRepBench (**Fo**rmula **Rep**air **Bench**mark), a new benchmark and dataset for context-aware Excel formula repair. Our main technical contributions are:

(1) **Excel Formula Repair Dataset**: We present FoRepBench, the first large-scale dataset of Excel formula repair examples for runtime errors. Each example includes spreadsheet context (cell values, headers), a broken formula, its corrected version, and a user utterance expressing intent. The dataset contains 618 examples and spans 5 runtime error types: #DIV/0!, #N/A, #NAME?, #REF!, and #VALUE!.
(2) **Synthetic Data Generation and Validation Pipeline**: We introduce a data generation pipeline that bootstraps from a small number of high-quality samples and produces realistic examples for training and evaluating models for Excel formula repair. We validate each repaired formula both via execution (to confirm correctness) and through a chain-of-thought LLM judge (to ensure semantic alignment with intent), resulting in high-quality examples.
(3) **Baseline Approach for Excel Formula Repair**: We propose a method that leverages a large language model and incorporates both formula text and spreadsheet context for error correction. We also report the performance of the baseline approach on FoRepBench and seed dataset using various state-of-art proprietary and open-source models.

We have made FoRepBench available as a resource for further research on Excel formula repair and related tasks. [1].

2 Related Work

2.1 LLMs for Code Generation

Large Language Models (LLMs) have emerged as a powerful paradigm for code generation. Early work such as GPT-3 [5] showed that scaling autoregressive transformers in a few-shot setting can yield impressive results in synthesizing code from natural language prompts. This breakthrough spurred the development of models fine-tuned specifically on code, substantially improving both fluency and correctness. OpenAI's Codex [7] adapts the GPT architecture with fine-tuning on a massive corpus of public code repositories and supports a wide range of programming tasks—from simple completions to complex algorithmic problems. Salesforce's CodeGen [21] and Meta AI's InCoder [18] introduced novel pre-training objectives, including span-masking and infilling, enabling the generation of code that integrates naturally within surrounding context.

Hybrid approaches have also been explored. DeepMind's AlphaCode [17] combines LLMs with search-based techniques, achieving competitive results on programming competitions. Other models, like CodeT5 [24], leverage structural information by incorporating representations of abstract syntax trees (ASTs) [25], improving syntactic accuracy and semantic consistency. In our experiments, we evaluate our formula repair approach using four recent LLMs as the backbone: GPT-4o [13], GPT-4.1 [2], Phi-3 [1], and Mistral [14], through prompt engineering tailored to the Excel formula repair task.

2.2 Excel Formula Generation and Repair

Research on code generation and repair in the context of Excel formulas remains relatively limited. One major line of work focuses on the NL-to-Formula (NL2F) task, which adapts the Text2SQL paradigm to Excel formula generation [26]. SpreadsheetCoder [8] enhances formula prediction by incorporating spreadsheet context, improving the accuracy of generated formulas. FlashFill [11] pioneered example-driven formula synthesis, enabling users to generate formulas via input-output examples. LaMirage [3] targets the "last-mile" repair problem by fixing near-correct formulas using symbolic and neural techniques, but it does not leverage the surrounding spreadsheet data for deeper semantic reasoning.

Recent work such as FLAME [15] introduced a lightweight transformer model for formula completion and repair, trained specifically on Excel formulas. While effective, FLAME operates solely on formula syntax and does not incorporate the spreadsheet context and natural language input, limiting its applicability in user-facing or

[1] https://github.com/microsoft/prose-benchmarks/tree/main/FoRepBench
[2] https://openai.com/index/gpt-4-1/

intent-driven tasks. In contrast, our dataset includes natural language utterances alongside spreadsheet context, faulty formulas, and ground-truth repairs. This enables the study of broader problem settings—ranging from NL-to-formula generation to semantic formula repair—not just syntactic correction or last-mile fixes.

Using LLM-as-a-judge for synthetic data evaluation has emerged as an active research area across text generation [10, 19, 23] and code generation tasks [6, 27]. Complementary to our work, Singh et al. [22] proposed an automated method for validating synthetic NL-to-formula datasets using LLMs. Their pipeline classifies and filters low-quality synthetic annotations to improve fine-tuning performance. We adopt a similar idea in our generation pipeline by incorporating LLM-based validation to ensure both execution correctness and semantic fidelity of generated examples.

3 Methodology

This section describes our proposed methodology for synthetic benchmark generation, which we refer to as Bootstrap Generator. Each data point in the benchmark, which represents a formula repair scenario, must include the following fields:

(1) **Tabular Data**: The spreadsheet context where the user encountered a runtime error.
(2) **Faulty Formula**: A formula that results in a runtime error. In this work, we focus on #N/A, #REF!, #VALUE!, #NAME?, and #DIV/0! errors.
(3) **Correct Formula**: A formula that resolves the runtime error and is also consistent with the user intent expressed in the utterance.
(4) **Utterance**: A natural language query describing a user's problem and/or task that they are attempting to solve with their formula.

(Refer to Figure 3 for an example data point for Excel formula repair.)

Our data generation method relies on a small set of high quality examples to generate a larger dataset. In Section 3.1, we describe how we curated a set of seed samples. Then in Section 3.2, we describe our synthetic data generation approach which creates FoRepBench.

3.1 Seed Data Curation

To create a seed dataset for bootstrap generation, we developed a systematic approach to collect and process data from online forums where users discuss Excel-related problems and solutions. This section describes the methodology used to gather relevant seed data and prepare it for use in our synthetic data generation pipeline.

3.1.1 Dataset Creation. Figure 1 shows an overview of our seed dataset creation approach. We scraped posts from the MrExcel[3] forum, a well-established platform where users frequently seek assistance with Excel formulas and share solutions. Next, we filtered posts to ensure that they had all the required elements for our dataset, i.e. a faulty formula, table context, and the correct formula (we used a reply being marked as "accepted answer" on the forum as an indicator). Once relevant posts were identified, we extracted the table context and formulas. Users often share tabular data and formulas in various formats, including plain text, code blocks, or specialized markup. We employed parsing techniques to accurately extract the exact text of the formulas used in cells, the data values in cells which may be referenced by the formulas, and any error messages or codes displayed by Excel, as reported by the users. Using the extracted data, we reconstructed the structure of the Excel workbooks by identifying different worksheets mentioned in the posts, mapping formulas and values to their corresponding cell addresses, and understanding the relationships between cells, such as which cells are referenced by a formula. Since we want to create a dataset focused on runtime errors, we needed to identify posts where the faulty formula resulted in a runtime error. We simulated the evaluation of the extracted formulas using Calc.ts[4], an Excel formula evaluation engine capable of interpreting and calculating the results of formulas outside of the Excel application environment. This allowed us to detect the type of errors produced by the faulty formulas and test the corrected formulas to confirm that they resolve the errors. We retained posts for 5 runtime error types: #N/A, #REF!, #VALUE!, #NAME?, and #DIV/0!. For every post that passed, we added one sample to our dataset containing: 1) faulty formula, 2) correct formula, 3) table context, 4) user query, 5) runtime error type, and other metadata.

3.1.2 Manual Verification and Correction. Although we applied several automated filtering and validation steps, not all samples met our requirements for high-quality seed samples. This is because 1) a formula that was accepted by a user on the forum as a solution and did not result in a runtime error when executed through Calc.ts could still be *semantically* incorrect, and 2) The table extracted by our scripts could contain data that is not part of the user's intended table context. For example, in one of the samples, a column with the expected outputs was included in the context. A good benchmark sample should test a model's formula repair capability without leaking information.

To address these limitations, we conducted two rounds of reviews to verify and correct the seed data. In the first round, each sample was assigned to one annotator. The annotator was asked to comment on the following:

(1) Does the correct formula meet the requirements expressed in the utterance?
(2) Does the faulty formula produce the required error type?
(3) Is the table accurately extracted from the post?
(4) Is the table consistent with the user utterance?

Only a third of the samples annotated in the first round met all the requirements. We therefore had a second round of annotations to verify the labels in the first round, and if necessary, edit the samples. In this round, the samples were reviewed again by a group of three or more annotators. If they agreed that the sample met the above requirements, it was added to the final seed dataset. If not, the sample was edited based on the comments from the first round. In some instances, the user utterance was edited to make it more explicit. In some cases, the formula crawled automatically was not correct for the user's intent and was manually replaced with a truly correct formula. Some examples were deleted if they

[3]https://www.mrexcel.com/

[4]https://www.microsoft.com/en-us/garage/wall-of-fame/calc-ts-in-excel-for-the-web/?msockid=38b38871134a6f5806f59df512676e0c

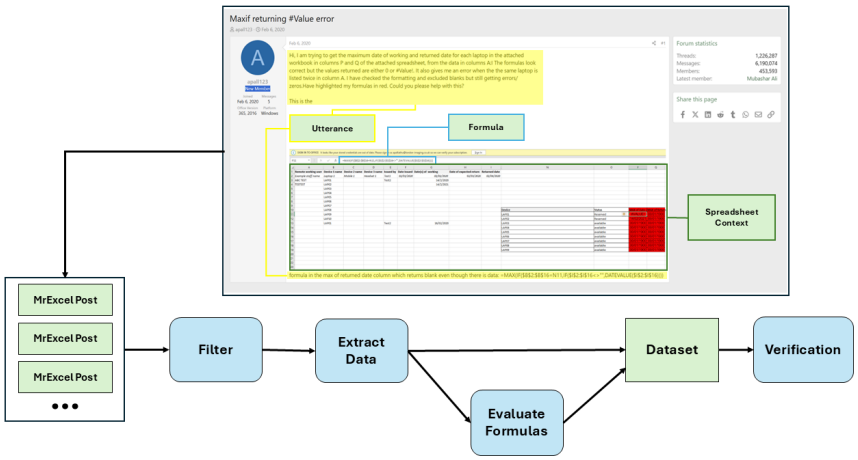

Figure 1: Overall workflow illustrating the sequential steps in the seed data curation process. MrExcel forum is scraped to get posts. Filters are used to identify posts containing table context, faulty formula and correct formula. The page is parsed to extract the required data. The faulty and correct formula are evaluated using Calc.ts. Samples where they execute as expected are added to the dataset. Finally the dataset is manually verified.

were too ambiguous or if the error was too trivial. This process ensured that for the samples in the seed dataset,

(1) The table contains the necessary information for the repair, and not rows with the desired output that might leak information.
(2) The user utterance clearly indicates what the user wants to accomplish.

3.2 Bootstrap Generation

Automatic extraction of data from forums as discussed in Section 3.1.1 led to incomplete and inaccurate samples, and manual validation and correction is not scalable to a large number of samples. In this section, we introduce our Bootstrap Generator approach, using which we created FoRepBench, a large scale benchmark dataset for Excel formula repair focused on runtime errors. Figure 2 shows an overview of the pipeline. We start with a small number of high-quality samples, i.e. seed samples (See Section 3.1), and generate a larger benchmark dataset synthetically.

The development of the synthetic data generation pipeline was guided by three primary objectives:

(1) Ensure proper formatting of all synthetic data.
(2) Verify the correct execution of all synthetic formulas in Excel.
(3) Ensure that the data is semantically consistent
(4) Assess the quality of the data to ensure appropriate difficulty levels, function coverage, etc.

The first step in the pipeline involves generating synthetic samples using few-shot prompting. Following the generation phase, it is essential to validate the synthesized samples to ensure both correctness and quality. This led to the creation of FoRepBench, constructed through the following two-stage validation process. The complete pipeline consists of these three steps, as described below.

3.2.1 ***Data Generation with Few-Shot Prompting.*** We apply *one-shot* prompting to generate synthetic samples utilizing every sample from our seed data. This approach involved injecting each data point from our seed data into its own text prompt and subsequently generating new data based on that prompt. This method proved to be the most effective in our validation experiments which have been discussed in Section 3.1.

Fewshot-learning Setup Optimization. To generate synthetic samples, we initially employed a zero-shot prompt to evaluate the LLM's capability to generate samples without grounding data. We generated 125 data points per error type, and the key observations were as follows:

(1) **Simple Data:** The generated Excel formulas and tables were relatively simple, e.g., dividing an arbitrary cell value by 0 to produce a #DIV/0! error.
(2) **Lack of Diversity:** The resulting synthetic data exhibited minimal semantic and syntactic diversity. For a given error type, the model predominantly produced data points with minor differences to the table data or variables in the formula.

Despite extensive prompt modifications to address the aforementioned issues, the problems persisted. Consequently, the next phase involved incorporating real-world grounding data into our prompts as few-shot examples for generating new data.

Next, we explored few-shot prompting as a more promising approach for creating a dataset with more diversity and complex examples. Our validation experiments showed that 1-shot prompting produced more samples that passed our validation tests (§3.2.2 and §3.2.3) compared to zero-shot. Each data point from the seed dataset was used to produce multiple new samples.

3.2.2 ***Validating Generations executing Excel formulas.*** To verify correctness, we utilized a tool called Calc.ts[5], designed to

[5]https://www.microsoft.com/garage/wall-of-fame/calc-ts-in-excel-for-the-web/

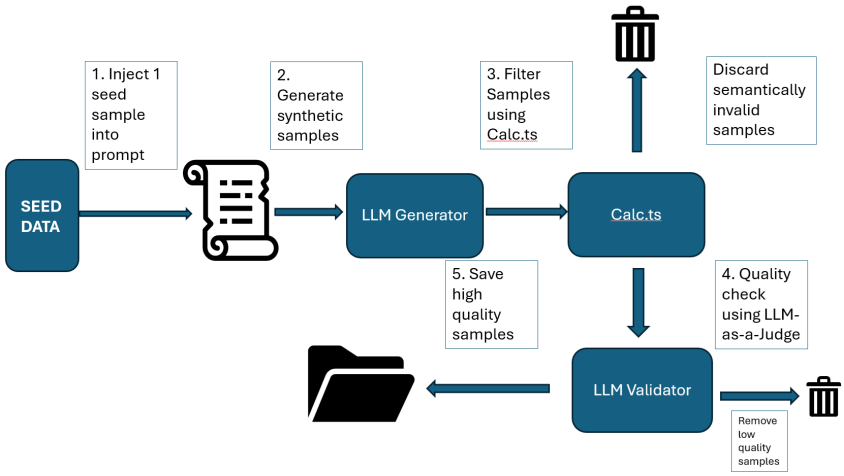

Figure 2: Pipeline overview of the synthetic data generation system.

check Excel formulas against the corresponding spreadsheet data. We ensured that each generated "correct" formula did not result in an error and that faulty formulas produced the appropriate runtime error. After confirming correctness, we evaluated the quality of the generated data using an LLM-as-a-judge framework. Samples that passed both correctness and quality checks were subsequently added to the final dataset.

3.2.3 ***Validating Generations with LLM-as-a-Judge Approach.*** To ensure reliability of the generated synthetic data, the LLM-judge approach we implemented leverages Chain-of-Though (CoT) reasoning for systematic assessment. We refer to this model as LLM Validator. In this work, the LLM-judge systematically analyzes each repaired formula by first determining whether it resolves the original runtime error in the synthetic data. It then assesses whether the formula aligns with the user's intent, considering the spreadsheet context and any provided utterance. It is also prompted to assess and annotate the difficulty level of the repair, which we use for analysis of our proposed benchmark in Section 6.1.

4 Formula Repair

As discussed in Section 1, there has been scarcity of research done on systems capable of Excel formula repair for formulas that result in semantic errors, rather than merely correcting syntax mistakes. A key challenge in Excel formula repair lies in incorporating the relevant spreadsheet context. Unlike general-purpose programming languages, Excel formulas are tightly coupled to tabular data layouts, and the correct repair often depends on values, ranges, headers, or even user-entered text elsewhere in the spreadsheet.

To address this, we propose a baseline solution that not only leverages the buggy formula and any available auxiliary information (such as natural language descriptions), but also uses the context present within the spreadsheet. Our system thereby enables context-aware formula repair that handles both syntactic and semantic errors. We further utilize our baseline repair pipeline to evaluate FoRepBench, thereby demonstrating its practical relevance and showcasing how such controlled benchmarks can effectively approximate real-life formula repair scenarios encountered in production spreadsheets.

4.1 Baseline Repair Technique

The baseline method follows a structured pipeline that makes a single call to an LLM, designed to efficiently process faulty Excel formulas and generate repaired versions along with explanations. Figure 2 illustrates the single-call LLM solution used for evaluating the benchmark. The system takes as input a faulty formula, the corresponding runtime error, and an optional user utterance. Since spreadsheet tables can be large, passing the entire spreadsheet as context is impractical due to LLM token limitations. To address this, we identify the nearest table associated with the faulty formula and extract its header along with a few sample rows to provide as context. This enables the LLM to receive the necessary contextual information to generate accurate repairs.

Once the relevant spreadsheet context is retrieved, a structured prompt is constructed. This prompt consists of four key elements: the extracted spreadsheet data context, the faulty formula, the runtime error, and an optional user utterance (if provided). In this work, a standardized prompt template is used to maintain consistency, and it included instructions that guide the LLM in repairing Excel formulas while ensuring a meaningful explanation is generated. The constructed prompt is then passed to the LLM, which processes the input and attempts to generate a corrected formula along with a natural language explanation of the fix. The repaired formula is subsequently evaluated by comparing it with the ground truth correct formula from the benchmark dataset. This comparison helps determine whether the generated repair successfully resolves the runtime error while maintaining the intended logic of the original formula. This structured pipeline used in this work provides a consistent and repeatable methodology for evaluating formula repair performance.

Tabular Data

	A	B	C
1	695172-Sale Polo Blue XL Shirt	695172	#N/A

Faulty Formula
VLOOKUP(B1,MID(A:A,8,15),1,FALSE)

Correct Formula
SUBSTITUTE(VLOOKUP(B1&"*",A:A,1,FALSE),B1&"-","")

Utterance
I have a number followed by the description in Col A. I have tried to set up a Vlookup to lookup the description pertainingto the number.

Figure 3: An example from the seed dataset where the faulty formula applies substring extraction, while the correct formula uses wildcard matching and string substitution—requiring multiple semantic edits to fix.

Tabular Data

	A	B
1	Item	Price
2	Apple	1.2
3	Banana	0.5
4	Cherry	2
5	Date	3
6	Fig	2.5
7	Grape	2.8
8		
9	Item to Find	Found Price
10	Mango	#N/A

Faulty Formula
VLOOKUP(A10, A2:B7, 2, FALSE)

Correct Formula
IF(COUNTIF(A2:A7, A10) > 0, VLOOKUP(A10, A2:B7, 2, FALSE), "Not found")

Utterance
I am trying to use VLOOKUP to find the price of an item, but I keep getting an #N/A error. What am I doing wrong?

Figure 4: An example from FoRepBench. The faulty formula does not handle the exception when the value A10 is not present in the table. The correct formula fixes this by checking if the value exists, and returning "Not Found" if it doesn't.

5 Experimental Setup

We applied Bootstrap Generator(Section 3.2) to generate our dataset using `GPT-4o` as the LLM. A temperature of 0.64 is used to promote diversity among generated examples. In total, we generated 1095 samples, out of which 618 passed LLM Validator. Table 1 shows the error-wise breakdown. We conducted analysis to assess the characteristics and quality of our dataset. We address the following research questions:

- RQ1 Does the data distribution in the FoRepBench match real world data? How does the generated data look in terms of distribution across errors and functions?
- RQ2 What is the quality of the FoRepBench based on human judgments?
- RQ3 What is the performance of the proposed baseline repair approach across a range of state-of-the-art proprietary (i.e, `Gpt-4.1`, `Gpt-4o`) and open-source LLMs (i.e, `Phi-3`, `Mistral`) on FoRepBench?
- RQ4 What is the cost of generating FoRepBench? How many LLM calls are needed?

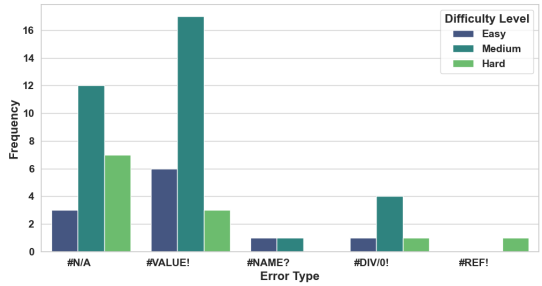

Figure 5: Distribution of sample difficulty levels by Excel error type in the seed dataset. Difficulty levels were assigned by LLM Validator.

To answer RQ1, we present the difficulty and function distributions for both the seed dataset and FoRepBench. The difficulty of each sample was determined by LLM Validator. It was prompted to assign one of 3 difficulty ratings to the sample based on how complex the Excel repair task was - easy, medium, and hard.

To answer RQ2, we recruited 2 annotators to assess the quality of the generated data *before* it was passed through LLM Validator. Due to the complexity of the task and limited resources, two teammates with extensive familiarity with the task and deeper understanding of the nuances served as annotators in this study. They annotated a subset of 24 samples from the synthetic dataset. They were asked to perform the same task as LLM Validator in Section 3.2, i.e. check the correctness and consistency of the table context, faulty formula, correct formula, and utterance.

5.1 Metrics

To evaluate the performance of the baseline repair technique across datasets, we employ the following execution-based metrics:

Syntax Validity: We first check whether the repaired Excel formula can be successfully compiled. If the formula parses without any compilation errors, it is considered syntactically valid; otherwise, it is marked as invalid.

Can Execute: This metric verifies whether the repaired formula can be successfully executed on the spreadsheet without triggering any runtime errors (e.g., `#VALUE!`, `#REF!`, `#DIV/0!`, etc.). A successful execution without runtime errors is considered a success; otherwise, it is considered a failure.

Execution Match: After execution, we compare the output produced by the repaired formula against the output of the ground-truth (correct) formula. If the outputs match exactly the repair is considered correct under this metric.

6 Results

6.1 RQ1: Data Distribution and Comparison with Seed Dataset

Figures 5 and 6 display the distribution of difficulty levels across error types in the seed dataset and FoRepBench, respectively. Compared to the seed dataset, the samples in FoRepBench are skewed

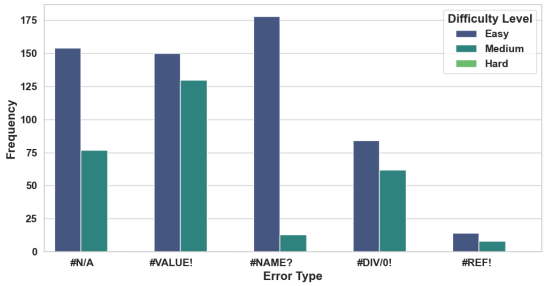

Figure 6: Distribution of sample difficulty levels by Excel error type in FoRepBench. Difficulty levels were assigned by LLM Validator.

Error Type	Samples Generated	Samples passed by LLM Validator
#VALUE!	241	64
#N/A	329	140
#REF!	16	12
#NAME?	222	158
#DIV/0!	287	244

Table 1: The table shows the number of generated samples that remained after LLM Validator filtered out low quality samples. Overall, 56% of all generated samples pass LLM Validator.

toward simpler formula repairs. This suggests that Bootstrap Generator tends to produce less complex scenarios, as it's hard to synthesize examples that both execute successfully and pass the LLM-based quality filter LLM Validator. Figures 3 and 4 illustrate this disparity in complexity. Both examples have faulty formulas with the VLOOKUP function. In the example from the seed dataset in Figure 3, the user wants to extract a part of a string from a cell, a semantically complex scenario. In a simpler (more syntactic) scenario from FoRepBench shown in Figure 4, the user only needs to handle an exception when the value being looked up isn't present in that cell range.

Tables 2 and 3 present the distribution of Excel functions in the seed dataset and FoRepBench respectively, broken down by error type. We include the top 10 functions that appear in the most examples in this dataset. Comparing the distribution with Table 2, FoRepBench frequently has the functions AVERAGE and CONCATENATE which the seed dataset did not. This indicates that Bootstrap Generator creates samples diverse from the fewshot examples. Some trends remain consistent between both datasets, for example VLOOKUP, INDEX, and MATCH being the most common functions for #N/A error.

6.2 RQ2: Synthetic Dataset Quality Based on Human Evaluation

Table 4 reports Cohen's Kappa scores measuring pairwise agreement between the two human annotators and between each annotator and LLM Validator. The results indicate moderate agreement among human annotators (Kappa: 0.60), suggesting some subjectivity in evaluating the quality of generated samples. Annotators

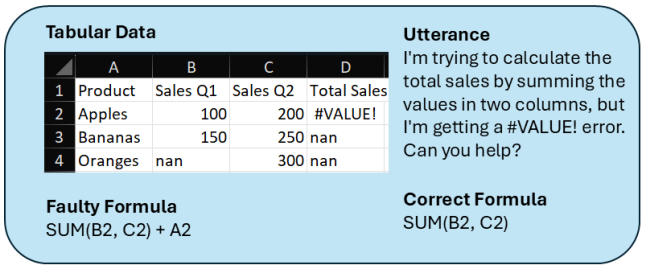

Figure 7: Example from the synthetic dataset *before* being passed through LLM Validator. where annotators disagree on validity. The formula includes cell A2 in a summation, but A2 contains a string. One annotator considers it valid, interpreting it as a possible typo, while the other marks it as invalid.

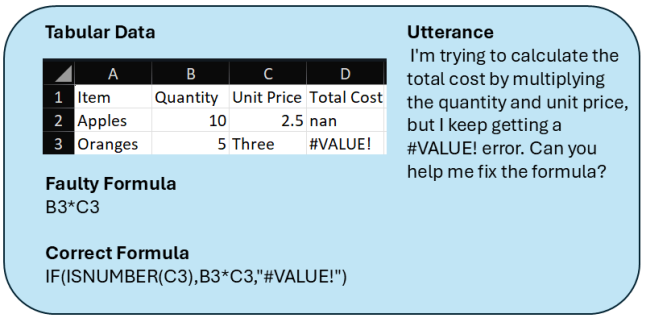

Figure 8: Example from FoRepBench where human annotators disagree with LLM Validator regarding validity. In this case, the "Unit Price" column contains the string "Three" instead of a numeric value. While annotators label the example as invalid, LLM Validator incorrectly accepts it, highlighting its limitations in filtering out unrealistic inputs.

were asked to assess whether the corrected formula appropriately fixes the faulty one and satisfies the intended user operation, and whether the example reflects a realistic spreadsheet scenario. Disagreements often stemmed from differing interpretations of what constitutes a plausible Excel table. For instance, in the example in Figure 7, where a formula attempted to multiply a string by a number, one annotator interpreted it as a plausible user typo, while the other considered it unrealistic. This suggests the need for more concrete annotation guidelines with detailed examples for the annotators.

Agreement between LLM Validator and each annotator was also moderate (0.42 for both annotators) but notably lower than inter-annotator agreement. In most cases, LLM Validator accepted examples that were consistent in formula execution but unrealistic in context. For example, in the sample in Figure 8, the table has textual values like "three" in a numeric column. While LLM Validator deemed such examples valid based on logical consistency, human annotators rejected them due to implausible table semantics. These results imply that while LLM Validator is effective at rejecting invalid or illogical formula pairs, it lacks sensitivity

Error Type	AND	FIND	IF	INDEX	LEFT	MATCH	MID	MIN	SUM	VLOOKUP
#DIV/0!	0	0	1	0	0	0	0	2	1	0
#N/A	1	0	2	8	2	10	1	0	1	7
#NAME?	0	0	1	0	1	0	0	0	0	0
#REF!	0	0	0	0	0	0	0	0	0	0
#VALUE!	2	4	12	2	2	3	3	1	2	0

Table 2: This table reports the frequency of the top 10 most frequent functions in the seed dataset (N=59), split by error type. Darker color signifies higher frequency within each row.

Error Type	AVERAGE	AVRG	CONCATENATE	COUNT	IF	INDEX	MATCH	SUM	VALUE	VLOOKUP
#DIV/0!	3	0	0	4	1	0	0	9	0	0
#N/A	1	0	0	0	1	25	28	1	0	66
#NAME?	2	4	0	0	2	0	0	1	0	0
#REF!	0	0	0	0	0	0	0	1	0	2
#VALUE!	9	0	16	0	10	2	3	11	4	0

Table 3: The table reports the frequency of the top 10 most frequent functions present in FoRepBench (N=618), split by error type. Darker color signifies higher frequency within each row. Comparing the distribution with Table 2, FoRepBench frequently has the functions AVERAGE and CONCATENATE which the seed dataset did not. This indicates that Bootstrap Generator creates samples diverse from the fewshot examples. Some trends remain consistent between both datasets, for example VLOOKUP, INDEX, and MATCH being the most common functions for #N/A error.

to the contextual plausibility of spreadsheet content. As a result, some unrealistic samples may persist in the dataset despite passing automated filtering.

	Ann-1 vs. LLM	Ann-2 vs. LLM	Ann-1 vs. Ann-2
Kappa	0.42	0.42	0.60

Table 4: Cohen's Kappa agreement scores between human annotators and LLM Validator.

6.3 RQ3: Performance of Repair Task on Synthetic and Seed Data

We evaluate the performance of the baseline excel formula repair technique described in Section 4.1 on both FoRepBench and the seed dataset. To ensure diversity in our evaluation, we selected four representative LLMs spanning a range of model families, including state-of-the-art proprietary large language models (i.e, `Gpt-4.1`, `Gpt-4o`) and open-weight models (i.e, `Phi-3`, `Mistral`), as well as different architectural paradigms (transformer-based and mixture-of-experts). Table 5 reports the results across the three execution-based metrics described in Section 5.1, evaluated over both datasets using different LLMs that power the baseline repair technique.

Overall, we observe that both the *Execution Match* and *Can Execute* scores are significantly lower across LLMs on the seed dataset compared to FoRepBench, indicating that the FoRepBench is relatively easier for the repair technique to handle which matches our learning from the RQ 6.1 on difficulty level distribution. Upon further analysis, we identify two primary reasons for this discrepancy: (1) the faulty formulas in the seed dataset are generally more complex, often involving deeper levels of nesting; and (2) the number of edits required to transform the faulty formula into the correct version is substantially higher.

For instance, consider a faulty formula from the manually annotated seed dataset shown in Figure 3. This example contains nesting of depth two and uses both `VLOOKUP` and `MID` functions. To repair this formula, multiple non-trivial edits are required, including altering the internal logic from $MID(A:A, 8, 15)$ to simply $A:A$ within the `VLOOKUP` function, followed by additional transformations where the looked-up value is further modified to $B1\&" - "$. Such repairs demand reasoning about user intent and understanding of higher-level semantics, as the modifications involve significant logic changes rather than isolated token-level corrections.

In contrast, examples from the synthetic dataset often require fewer edits to achieve the correct formula, as illustrated in Figure 3. In these cases, the internal logic embedded within the `VLOOKUP` function typically remains intact, with only minor corrections or additional function wrap-up are needed. Consequently, these repairs are more straightforward for the model to handle, as they involve localized changes rather than substantial semantic rewrites. We also see that bigger GPT-4 series models are better at solving the repair tasks in comparison to Phi-3 and Mistral suggesting more layers and training data can help improve Excel formula repair task.

These findings suggest that while the baseline repair technique is effective on FoRepBench, the Bootstrap Generator pipeline may not fully capture the range of complexity observed in real-world Excel formulas. Specifically, the synthetic instances from FoRepBench tend to exhibit shallower nesting and require fewer semantic transformations compared to the manually curated seed dataset. Building on this insight, we propose that incorporating an additional *reviewer agent* or a *human-in-the-loop component* into the Bootstrap Generator pipeline could help bridge this gap. By comparing generated synthetic samples against real-world instances and providing targeted feedback, the pipeline could iteratively generate more complex and diverse faulty formula instances that better reflect real-world repair challenges.

6.4 RQ4: Cost of Dataset Generation

Table 6 summarizes the cost of generating the FoRepBench dataset in terms of LLM API usage. For data generation, we issue a single

LLM	Dataset Origin	Syntax Valid	Can Execute	Execution Match
GPT-4.1	FoRepBench	**1.00**	**0.96**	**0.80**
	Seed Dataset	**0.98**	**0.65**	**0.41**
GPT-4o	FoRepBench	1.00	0.93	0.73
	Seed Dataset	0.96	0.63	0.35
Phi-3	FoRepBench	0.81	0.77	0.58
	Seed Dataset	0.73	0.41	0.24
Mistral	FoRepBench	0.78	0.76	0.51
	Seed Dataset	0.67	0.37	0.19

Table 5: Performance of Baseline Repair Technique on FoRepBench and seed dataset across various LLMs.

LLM call per prompt to generate N candidate samples (where $N = 25$). From this pool, 1,095 samples contained executable formulas with the correct error type, as automatically verified by Calc.ts. For each of these accepted samples, we issue one additional LLM call to assess semantic validity, resulting in a total of 1,154 LLM calls (59 for generation and 1,095 for validation). This corresponds to an average of approximately 2.09 LLM calls per accepted sample.

On average, each call consumes approximately 2,000 tokens, yielding a per-sample generation cost of approximately \$0.02[6]. These results demonstrate that the Bootstrap Generator pipeline is both scalable and cost-effective for generating large-scale formula repair datasets.

Generation Calls	Validation Calls	Avg. Calls/Valid Sample
59	1,095	2.09

Table 6: LLM call breakdown for generating the FoRepBench dataset. Out of 1,095 executable samples, 618 are accepted by LLM Validator.

7 Discussion and Conclusion

In this work, we introduced a modular, low-supervision pipeline for generating and validating synthetic Excel formula repair data, resulting in the FoRepBench benchmark. Our Bootstrap Generator method combines structured spreadsheet context with prompt-based LLM sampling to produce realistic faulty formulas, which are then filtered through a multi-stage validation process. This process leverages automated execution checks and an LLM-based reviewer agent that evaluates semantic plausibility. We also proposed a prompt-based repair baseline system to evaluate model performance on both synthetic and real (seed) data. Our investigation across four research questions uncovered key insights about the characteristics, challenges, and utility of the generated data.

From RQ1, we found the synthetic dataset covers a broad range of formula categories and function types, achieving greater diversity than the manually curated seed dataset. RQ2 reinforced this, showing that although synthetic data spans multiple function categories, its error types and logical transformations are more localized and less complex than those in seed data. RQ3 examined LLM-based repair performance on both datasets, revealing a stark contrast: models like GPT-4o and GPT-4.1 perform well on synthetic samples (execution match rates up to 0.80), but accuracy drops significantly on seed data, which often requires deeper nesting and multi-step reasoning. This gap highlights a limitation of synthetic generation-despite lexical and structural diversity, synthetic examples lack the semantic complexity of real-world formulas that demand substantial rewrites or ambiguous intent interpretation. Conversely, synthetic examples tend to preserve internal logic, requiring minor corrections. RQ4 addressed the efficiency of data generation. With only 1,154 total LLM API calls—59 for generation and 1,095 for validation—we curated 618 high-quality samples, averaging just over two calls which costs only \$0.02 per accepted sample. This demonstrates that large-scale, diverse datasets can be created with minimal manual effort, adaptable to other structured domains with simulated task-specific errors.

Despite these strengths, limitations remain. The LLM-based reviewer agent, while scalable, diverges from human annotators (Cohen's kappa 0.25), reflecting the challenge of modeling subjective plausibility. Reviewer judgments may misalign with human correctness, especially in ambiguous or multi-step reasoning cases. Additionally, narrow context windows due to token limits can omit globally relevant spreadsheet information like distant dependencies or multi-sheet references, limiting repair accuracy for complex scenarios. Finally, our dataset focuses on single-sheet, English-language spreadsheets, omitting collaborative, localized, and dynamic formula contexts common in practice.

These findings suggest promising future directions: incorporating human-in-the-loop validation to improve fidelity; refining reviewer agents via feedback fine-tuning or preference learning; enhancing context selection with structure-aware retrieval to prioritize semantically relevant spreadsheet regions; and extending the pipeline to multi-sheet, shared, and localized spreadsheets to broaden applicability.

Beyond benchmarking, FoRepBench can serve as a data augmentation tool to improve formula understanding and repair models. By simulating common spreadsheet errors at scale, the pipeline offers a valuable resource for training and evaluating systems that assist users in real-world spreadsheet environments. Our work demonstrates that scalable, semi-automated generation of Excel formula repair data is feasible and effective. While synthetic data cannot fully replace human-curated examples, carefully filtered synthetic samples can enhance model robustness and enable systematic benchmarking. We hope FoRepBench will support future research in end-user programming, intelligent assistants, and robust Excel formula repair.

[6]https://llmpricecheck.com/openai/gpt-4o/

References

[1] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219* (2024).

[2] Daniel W Barowy, Shan Gao, Alvin Cheung, and Brad A Myers. 2014. ExceLint: Automatically detecting spreadsheet formula errors. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 460–470.

[3] Rohan Bavishi, Harshit Joshi, José Cambronero, Anna Fariha, Sumit Gulwani, Vu Le, Ivan Radiček, and Ashish Tiwari. 2022. Neurosymbolic repair for low-code formula languages. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 164 (Oct. 2022), 30 pages. doi:10.1145/3563327

[4] Rohan Bavishi, Harshita Joshi, Jorge Cambronero, Ayesha Fariha, Sumit Gulwani, Vu Le, Ivan Radiček, and Aditya Tiwari. 2022. Neurosymbolic Repair for Low-Code Formula Languages. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022).

[5] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems* 33 (2020).

[6] Binyuan Chen, Qian Liu, Jinjie Jiang, and et al. 2023. CodeLLM: Evaluating Large Language Models on Code Generation. *arXiv preprint arXiv:2305.14335* (2023).

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. 2021. SpreadsheetCoder: Formula Prediction from Semi-structured Context. arXiv:2106.15339 [cs.SE] https://arxiv.org/abs/2106.15339

[9] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, Volume 1*. IEEE, 358–368.

[10] Aniruddh Gudibande, Xisen Li, Ethan Chi, Percy Liang, and Yuxin Wu. 2023. False sense of security: Evaluation misalignment in language models. *arXiv preprint arXiv:2304.09106* (2023).

[11] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. doi:10.1145/1926385.1926423

[12] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2016. Detecting errors in spreadsheets. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, 818–828.

[13] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[14] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL] https://arxiv.org/abs/2310.06825

[15] Harshit Joshi, Abishai Ebenezer, José Cambronero Sanchez, Sumit Gulwani, Aditya Kanade, Vu Le, Ivan Radiček, and Gust Verbruggen. 2024. Flame: A small language model for spreadsheet formulas. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 12995–13003.

[16] Sean Kandel, Andreas Paepcke, Joseph M Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3363–3372.

[17] Xi Li et al. 2022. Competition-Level Code Generation with AlphaCode. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[18] A. Liu et al. 2022. InCoder: A Generative Model for Code Infill. arXiv:2210.00745 [cs.CL]

[19] Fangyu Liu, Yuxian Wu, Yixuan Liu, and et al. 2023. GPTEval: NLG evaluation using GPT-4 as the reference-free evaluator. *arXiv preprint arXiv:2305.04648* (2023).

[20] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. doi:10.1145/3105906

[21] Eric Nijkamp, Christopher Rosin, Antonio Martins, Adam Rogers, Thomas Wolf, Mikel Artetxe, Victor Costa, Sudheer Banerjee, Binh Shih, Emelie Siktberg, et al. 2022. CodeGen: An Open Large Language Model for Code Generation. *arXiv preprint arXiv:2203.13474* (2022).

[22] Usneek Singh, José Cambronero, Sumit Gulwani, Aditya Kanade, Anirudh Khatry, Vu Le, Mukul Singh, and Gust Verbruggen. 2024. An Empirical Study of Validating Synthetic Data for Formula Generation. arXiv:2407.10657 [cs.CL] https://arxiv.org/abs/2407.10657

[23] Alex Wang and Ellie Pavlick. 2023. ChatGPT as a judge: Linguistic acceptability judgments. *arXiv preprint arXiv:2304.03442* (2023).

[24] Shuai Wang, Feng Li, Jia Zhou, Ruo Yan, and Jie Chen. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *arXiv preprint arXiv:2109.00859* (2021).

[25] Pengcheng Yin and Graham Neubig. 2018. Learning to Represent Programs with Graphs. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 195–206.

[26] Wei Zhao, Zhitao Hou, Siyuan Wu, Yan Gao, Haoyu Dong, Yao Wan, Hongyu Zhang, Yulei Sui, and Haidong Zhang. 2024. NL2Formula: Generating Spreadsheet Formulas from Natural Language Queries. arXiv:2402.14853 [cs.CL] https://arxiv.org/abs/2402.14853

[27] Lei Zheng, Xiaowei Wang, Baoxu Peng, Xin Wang, and Minlie Huang. 2023. Judging code generation with large language models: A comparative study. *arXiv preprint arXiv:2305.17951* (2023).