Generating Data Engineering Code Using LLMs

Jialin Yang*, Bart Maciszewski*, Saviour Owolabi*, Ahmad Abdellatif, Henry Leung, Steve Drew
Department of Electrical and Software Engineering, *University of Calgary*, Calgary, Canada
{jialin.yang, bart.maciszewski, saviour.owolabi, ahmad.abdellatif, leungh, steve.drew}@ucalgary.ca

***Abstract*—Data engineering is a complex and time-consuming part of data science, critical for transforming raw data into actionable insights. This complexity stems from diverse and large data sources, data-dependent logic, and exploratory workflows. We perform an empirical evaluation of the performance of three LLMs (GPT-4o-mini, Claude-3.5-Haiku, and Gemini-2.0-Flash), which have been shown to be effective at code generation, on multi-step data engineering notebooks. Our study considers the impact of prompting with previous execution context, output samples, and the application of iterative refinement on entire notebooks and their individual steps. We benchmark performance against the ARCADE dataset and introduce a new benchmark derived from Spider 2.0 (Spider2-intents) to mitigate potential data leakage. Our results show that LLMs generate syntactically and semantically correct code, with output data match scores reaching up to 80%, and BLEU scores of 0.35 on our newly created Spider2-intents benchmark. While generated code trends toward reduced runtime, memory, and CPU usage, these improvements are not statistically significant. Further analysis reveals that ambiguity in user intents is the leading cause of functional correctness issues, accounting for 40.74% of such cases. We also observe that iterative refinement shows a modest but statistically inconclusive trend toward improved output correctness, with gains of up to 2.9% after two rounds of notebook and intent refinement each.**

***Index Terms*—Data Engineering, Large Language Models (LLMs), Code Generation, Multi-step Data Transformation**

I. Introduction

Data engineering in the context of enterprise data science is the process of extracting data from various sources (e.g. databases and APIs), transforming it (e.g. cleaning and aggregating), and loading it into a target system (e.g. data warehouse, flat file, or other data structure). According to Anaconda's State of Data Science 2024 report [1], such data preprocessing remains one of the most time-consuming tasks in data science. 88% of data practitioners list data preparation as the most time-consuming task, followed by data cleaning at 83%. These statistics highlight the persistent bottlenecks in the data analytics pipeline. The emergence of generative AI offers a promising avenue to ameliorate these pain points; 67% of data science practitioners already report using AI in some capacity (an 87% uptick from the previous year, 2023) for data cleansing, visualization, and analytics tasks[1] and we envisage that greater use of Generative AI in data engineering workflows could free up time for data scientists to focus on big-picture tasks. In addition, the use of Generative AI, and LLMs more specifically, could simplify data manipulation for non-technical users, allowing low or no-code interfaces for data analysis, consequently reducing reliance on specialists and supporting faster decision-making.

However, despite the potential and promise LLMs hold, their adoption for data engineering tasks still faces significant challenges with respect to the generation of correct, efficient, and reliable code, particularly in enterprise environments [2]. Existing LLM-based solutions often struggle with complex logic, multi-step operations, and ambiguous user *intents*, resulting in outputs that frequently require manual debugging and repair [3]. Moreover, little is known about the practical application of LLMs in orchestrating iterative data transformation workflows, where correctness and maintainability are critical.

To address the knowledge gap on the performance of modern LLMs on multi-step data engineering tasks, the underlying causes of any performance deficiencies, and the impact of techniques such as *iterative refinement*, we empirically evaluate the performance of three LLMs on the ARCADE dataset and Spider2-intents datasets, which we crafted for this study. Critically, we focus on Pandas-based workflows, whereas most prior work centers on SQL-based workflows. The first aim of our study is to establish the current LLM performance on the multi-step data engineering code generation task. The second goal of our work is to characterize the issues with the generated code. Third, we aim to explore how techniques like *iterative refinement* might be applied to improve the effectiveness of LLMs on the aforementioned task. To this end, we pose and answer the following research questions (RQs):

RQ1: ***How do modern LLMs perform on the multi-step data engineering code generation task?*** We find that LLMs are capable of generating syntactically and semantically correct code, with *output data match* scores reaching up to 80% and BLEU scores up to 0.35, when measured against ground truth code. They also show trends of reduced runtime, memory and and peak CPU usage. **RQ2:** ***What types of issues do we see in the generated code?*** Our findings show that *unsafe operations* (code without appropriate bounds and checks) are the leading cause of runtime errors in generated code (accounting for 50% of such cases), and *ambiguity* is the leading issue associated with functional incorrectness (accounting for 40.74% of such cases). **RQ3:** ***Does increasing the amount of iterative refinement* improve correctness and performance?** We find that iterative refinement shows a modest but statisti-

*Equal contribution. Listing order is random. All source code is available at: https://github.com/llm-data-engineering/llm_etl

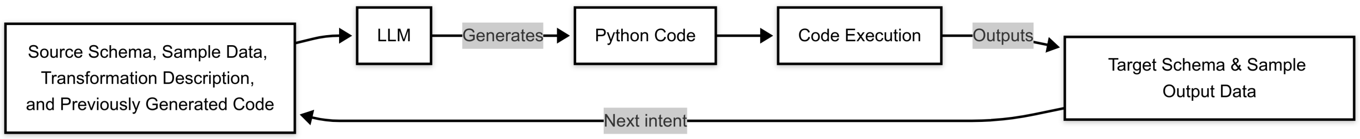

Source Schema, Sample Data, Transformation Description, and Previously Generated Code
LLM
Generates
Python Code
Code Execution
Outputs
Target Schema & Sample Output Data
Next intent

Fig. 1: Concept: LLM generates the next notebook *intent* providing previously generated code and sample input for context

cally inconclusive trend toward improved output correctness, with gains of up to 2.9% after two rounds of notebook and intent refinement. We also note runtime, memory, and CPU utilization improvements.

Achieving satisfactory performance in a real world setting with Pandas based workflows (with data transformations occurring in-memory) would unlock opportunities for faster data scientist experimentation and extracting value from data in today's data driven economy. To this end, we make the following contributions:

- To the best of our knowledge, we are the first to evaluate the key best practices for data engineering code generation; namely, providing notebook context, actual data samples, and execution results as part of the prompt using modern LLMs (Figure 1).
- We investigate and provide a classification on the most common types of errors and issues in the LLM generated code.
- We evaluate the impact of *iterative refinement* on entire notebook and their individual steps.
- We contribute a new, complex real world dataset derived from Spider 2.0 for evaluating LLM performance on data engineering tasks within the Python/Pandas framework.

II. Related works

Text-to-SQL Code Generation. LLMs have shown remarkable capabilities in program synthesis for SQL queries based on natural language descriptions. Yu et al. [4] introduced the Spider dataset, which has become a standard benchmark for evaluating text-to-SQL models. M.Pourreza et al.[5] explored how decomposing the general task from Spider and other datasets into smaller sub-tasks can be effective in improving the performance of LLMs in SQL code generation. Lei et al. [6] recently introduced Spider 2.0, focusing on real-world text-to-SQL tasks. Building on this, Deng et al. [7] proposed techniques like self-refinement, format restriction, and column exploration to boost performance—though achieving strong results on this benchmark remains difficult. Other studies, such as those by Hong et al. [8] and Li et al. [9], have investigated the use of LLMs as database interfaces, evaluating their ability to generate accurate SQL queries across diverse domains using Chain-of-Thought (CoT) reasoning. Despite these advances, few studies have explored decomposing natural language descriptions into Python-based data science workflows, especially for multi-step transformation tasks.

Code Generation for Data Engineering Tasks. LLMs have been proven to be effective at code generation. For example, Austin et al. [10] and Jain et al. [11] studied LLMs for general program synthesis, demonstrating their effectiveness in generating structured code. Nam et al. [12] explored predictive synthesis of API-centric code, which could be particularly relevant for data engineering tasks involving Pandas API calls. Additionally, Ma et al. [13] proposed leveraging LLMs as generic data operators for structured data processing to handle individual data transformations steps.

Recent work has explored fully automating the end-to-end data science and research process; but not specifically the data engineering task. Samuel et al. [14] and Lu et al. [15] included using an LLM to fulfill the data engineer role as part of their efforts to automate the scientific process. Grosnit et al. [16] proposed a model to automate the entire data science life cycle by learning from experience.

A number of benchmarks exist for data science tasks, with a subset of those specifically tailored to data engineering tasks. Lai et al. [17] introduced the DS-1000 benchmark for data science tasks which include data transformations. While the DS-1000 is a useful benchmark, its component tasks are independent of each other and as such, it falls short of being representative of real-world multi-step data transformations. Yin et al. [18] introduced the ARCADE data set that contains notebooks with multiple *rounds* of natural language "user *intents*" and code. They show that a fine tuned 62B parameter model can achieve relatively good *pass@k* performance but only for higher values of *k*. Huang et al. [19] introduced ExeDS, an evaluation dataset for execution evaluation for data science code generation tasks. ExeDS contains a set of 534 problems from Jupyter Notebooks, each consisting of code context, task description, reference program, and the desired execution output. Wen et Al. [20] extended the work of Yin et al. [18] to include synthetic data samples in addition to notebook context passed as part of the prompt to generate the next user *intent* in a multi-step notebook. Quoc et al. [21] explored the use of chain-of-thought reasoning and *iterative refinement* to generate general data science code, evaluating on the DS-1000 data set.

A noteworthy gap in the existing literature is the absence of a method that effectively integrates and leverages the key strengths of previously proposed approaches for generating multi-step data engineering code using LLMs. In addition, many studies (and the approaches they introduce) in the literature are dated and limited, lacking comprehensive evaluations with contemporary LLMs, with said evaluations neglecting runtime performance assessments and failing to reflect realistic enterprise scenarios.

To the best of our knowledge, we are the first to evaluate the performance of various modern LLMs for multi-step data engineering code generation using real data, notebook context, and execution feedback, while also contributing a novel dataset, a classification of common code issues, and an analysis of *iterative refinement* effects.

III. Terminology

Before presenting further details on our empirical study, we provide an overview of the terminology used throughout this paper.

- **Intent**: A natural language description of a specific data transformation task provided by a user. An example is provided in the second cell of Figure 2.
- **Notebook**: An interactive execution environment widely used by data engineers and data scientists for their flexibility and support for exploratory workflows. In the context of this study, 'notebook' also refers to a sequence of related data transformation *intents* and the code that satisfies those *intents*. The ARCADE and our Spider2-intents (see Section IV) contain such intent-code pairs.
- **Iteration**: A prompt-response exchange with an LLM aimed at generating code to fulfill a given *intent*. A single *iteration* refers to one such prompt-response pair, where the model generates code without further self-improvement. In contrast, when self-refinement is applied, previously generated code and outputs are passed back to the LLM to improve or fix the generated code.
- **Round**: A single pass through all *intents* associated with a notebook. During each round, *intents* are addressed sequentially. The LLM is provided with code and output generated in previous *rounds* as context.
- **Memory**: LLM-generated code and the corresponding outputs or errors (following execution) from each *round* and *iteration* that are stored and subsequently included in the prompt of further *iterations* and *rounds* to enable self-improvement.

IV. Case Study Setup

In this section, we present the approach we took to answering our research questions, including the datasets used, prompt structure, experimental setting, and model evaluation criteria.

A. Datasets

In this study, we use two popular datasets, namely **ARCADE** and **Spider 2.0-lite**. Furthermore, we construct a new dataset called **Spider2-intents** with a greater amount of complexity than ARCADE, as measured by the number of required data transformations and complexity of the input data **ARCADE**.

1) ***ARCADE:*** ARCADE (Answer Repository for Computational Analysis and Data Engineering), created by Yin et al. [18], contains 1082 natural language-to-code tasks sourced from both mined and manually authored Jupyter notebooks. It is split into two subsets:

- **ARCADE-Existing:** Tasks collected from repositories like JuICe [22] and BigQuery. These notebooks were filtered to include at least three Pandas API calls, or two API calls preceded by a descriptive markdown cell.
- **ARCADE-New:** Notebooks manually created using datasets released after February 2022 to avoid potential data leakage. These notebooks contain exploratory data analysis (EDA) and transformation tasks with a minimum of five Pandas operations per notebook.

Each task includes a sequence of natural language user *intents*, ground-truth code, and expected output data. Compared to other benchmarks like DS-1000 [17], ARCADE emphasizes execution feasibility and realistic task descriptions: approximately 45% of *intents* are deliberately underspecified, reflecting real LLM usage scenarios.

2) Spider2-intents (Derived Dataset): In addition to ARCADE, we construct *Spider2-intents*, a new dataset based on Spider 2.0-lite [6], a benchmark focused on complex enterprise *text-to-SQL* tasks, with each task characterized by a complex user *intent*, a ground-truth *SQL* query, and its execution output. We expand this dataset to support multi-step evaluation. More specifically, since Spider 2.0-lite includes only *single-intent SQL* tasks, we decompose them into sequential natural language *intents* paired with equivalent Python (Pandas) code using the following procedure:

1) We extract the SQLite-based subset (tasks with directly accessible input data) and export their tables to CSV files.
2) We leverage GPT-4o to automatically generate the instructions (*intents*) and corresponding Pandas code to mirror the logic of the original *SQL*.
3) The first three authors **manually** review and debug the generated code to ensure that the final notebook produces identical output to the original *SQL* result.
4) Finally, we group intent-code pairs into notebooks, with each notebook consisting, on average, of seven such pairs.

As Spider 2.0-lite was released after major LLM pretraining corpora were finalized, and we curated spider2-intents manually, we believe this dataset provides a trustworthy testbed that we can validate results from the *ARCADE* dataset on, and we use it as such.

Table I provides key statistics across the datasets we use in this study. The *original* columns describe the characteristics of the source datasets, while the *retained* columns reflect the subsets of the source datasets we use in our study after applying appropriate *filtering* criteria. In particular , we exclude any *ARCADE* intents for which the ground truth code could not be executed successfully. In addition, we include only tasks derived from *local* subset of *Spider2-lite* that provided ground truth SQL code. Note that **Spider2-intents** contains a comparable number of *intents* per notebook and transformation complexity to ARCADE, while being derived from independently released enterprise *SQL* tasks.

Both datasets feature collections of multiple *intents* (*notebooks*) aimed at achieving an overarching objective. This structure allows us to evaluate the performance of LLMs on multi-step code generation tasks. In addition, they support execution-

TABLE I: Dataset Description

Metric	**ARCADE-Existing**		**ARCADE-New**		**Spider 2.0-lite (local subset)**		**Spider2-intents**
	Original	Retained	Original	Retained	Original	Retained	
Intents	476	323	585	404	135	24	169
Intents per Notebook	7.68	6.10	9.44	6.97	1	1	7.04
Notebooks	62	53	62	58	135	24	24
Lines of code/Intent	2.28	1.99	2.83	2.99	53.3(*SQL*)	52.8(*SQL*)	3.78
Average Number of Transformations	2.32	1.67	5.89	4.02	10.92(*SQL*)	11(*SQL*)	-

driven evaluation by providing ground truth output data either directly (Spider 2.0-intents) or indirectly (ARCADE, where expected output can be obtained by executing the ground truth code). This is valuable for execution-grounded evaluation and helps us avoid relying solely on surface-level metrics like the BLEU score.

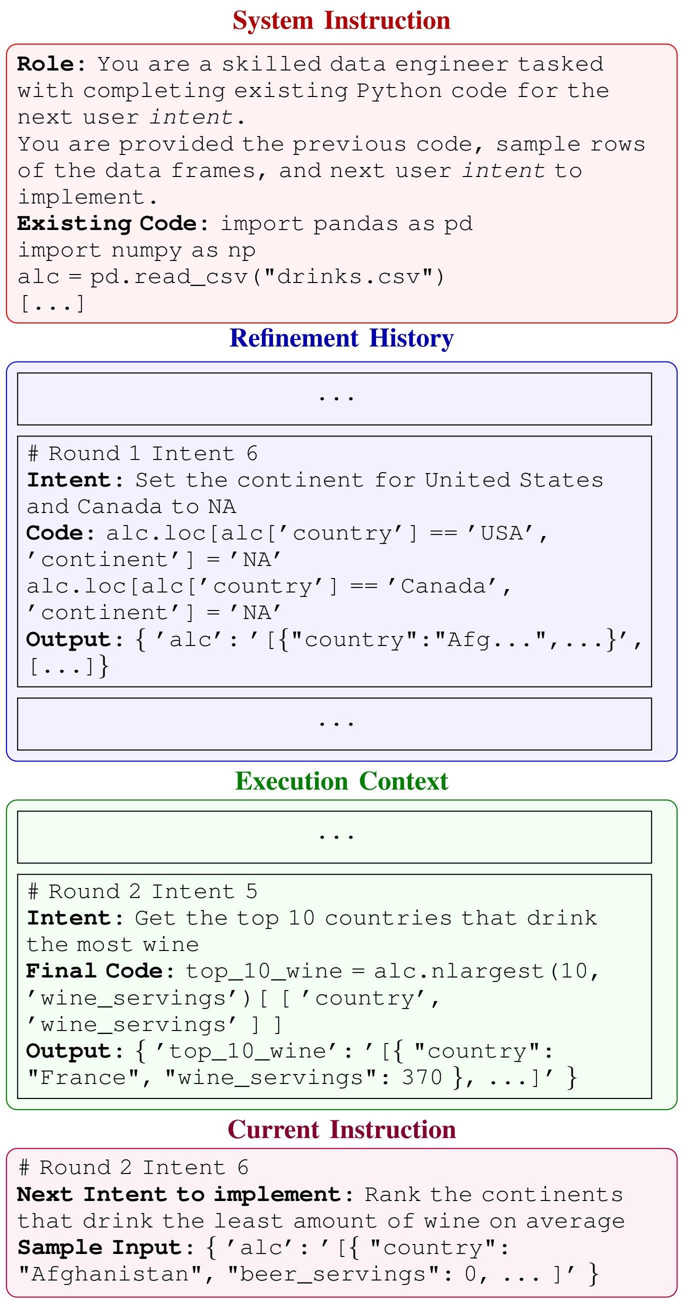
B. Prompt Structure

Effective prompting is essential for generating satisfactory code with LLMs. To support iterative improvement, we include previous code outputs and execution results as context from earlier *rounds* and *intents*. The prompt consists of the following components:

1) **LLM Role Description**: Defines the LLM's task and objectives.
2) **Setup Code from the Original Notebook**: Provides environmental setup code, including imported libraries and predefined functions.
3) **Next User Intent**: Specifies the transformation or task to be performed.
4) **Previous Intents and Code/Output Pairs from the Current Round**: Offers same *round* previous *intents* context for continuity.
5) **Prior Rounds' Intents and Code/Output Pairs**: Provides historical context to facilitate learning from past *rounds*, including the execution results from generated code in those *rounds*.
6) **Output Requirements**: A description of the expected output from the LLM to guide it to generate valid, executable Python code that meets task objectives.
7) **Sample Input**: Provides the first 10 rows of each data frame and series due to compute and context-window constraints.

We designed this prompt structure based on insights from several recent papers on self-improvement using context [5] [23] [24]. These works suggest that including execution history and structured inputs can help LLMs reason more effectively and improve the functional correctness of generated code over multiple *iterations*.

Figure 2 shows the structure of our prompt, exemplifying its form after multiple *intents* have been introduced in a sample second *round*.



System Instruction

```
Role: You are a skilled data engineer tasked
with completing existing Python code for the
next user intent.
You are provided the previous code, sample rows
of the data frames, and next user intent to
implement.
Existing Code: import pandas as pd
import numpy as np
alc = pd.read_csv("drinks.csv")
[...]
```

Refinement History

```
...

# Round 1 Intent 6
Intent: Set the continent for United States
and Canada to NA
Code: alc.loc[alc['country'] == 'USA',
'continent'] = 'NA'
alc.loc[alc['country'] == 'Canada',
'continent'] = 'NA'
Output: {'alc': '[{"country":"Afg...",...}',
[...]}

...
```

Execution Context

```
...

# Round 2 Intent 5
Intent: Get the top 10 countries that drink
the most wine
Final Code: top_10_wine = alc.nlargest(10,
'wine_servings')[['country',
'wine_servings']]
Output: {'top_10_wine': '[{"country":
"France", "wine_servings": 370}, ...]'}
```

Current Instruction

```
# Round 2 Intent 6
Next Intent to implement: Rank the continents
that drink the least amount of wine on average
Sample Input: {'alc': '[{"country":
"Afghanistan", "beer_servings": 0, ...]'}
```

Fig. 2: Multiple *Rounds* and Multiple *Iterations* Prompting Structure.

C. Experimental Setting

In this study, we focus on a practical data science scenario: generating data engineering code based on a user-provided *intent*. Specifically, given a source schema, a sample of source data, and a natural language description of the desired transformations, we prompt an LLM to generate transformation logic in Python using the Pandas library. Alongside this code, the model is also expected to generate the corresponding target schema and sample output data. Furthermore, we evaluate the performance of LLMs with *iterative refinement*. More specifically, at each time step, we prompt the LLM multiple times, each time modifying the context by including the code generated by the LLM in the previous time step, along with any execution outputs/errors generated on execution of said previously generated code. Figure 1 shows how we implement code generation with iterative refinement.

We scope our study to tabular data transformations implemented using Pandas. While many real-world data transformation workflows involve non-tabular data, we believe our study offers insights that can inform progress toward the development of such full-featured LLM-based data transformation systems. In addition, this defined scope ensures that all generated code produces structured, executable outputs in the form of Pandas data frames. This reflects a common real-world use case for data scientists and engineers. While there is potential to extend this approach to other data types (e.g., images), we focus exclusively on textual tabular data in our study due to time constraints and the widespread adoption of Pandas in industry workflows.

D. Model Evaluation

To benchmark the performance of various LLMs on data engineering tasks, we evaluate three prominent models at the time of this study: **GPT-4o-mini**[25] from OpenAI, **Claude 3.5 Haiku**[26] from Anthropic, and **Gemini 2.0 Flash**[27] from Google. These are all models from leading LLM providers, have been used in prior work[28, 29, 30] and have been demonstrated to show good performance in code generation tasks. Although large models may achieve higher absolute correctness, we focus on small models to better reflect the cost constraints and deployment realities of actual production environments.

We assess the code generated by these models through execution-driven criteria, utilizing the following performance metrics averaged over 3 runs:

1) **BLEU3:** Trigram overlap between generated code and the ground truth, calculated on each step. This captures the similarity in syntax between the generated code and code from the oracle.
2) **Output match:** Proportion of generated outputs with an exact match in table structure, column names, and data values relative to the sample output. This metric captures functional correctness, accounting for the possibility of multiple valid solutions.
3) **Performance:** Elapsed runtime, peak memory and peak CPU utilization. All notebooks are executed in a single Kaggle process with the same hardware configuration of 4 virtual CPUs and 30GB of RAM. We use *psutil* to measure peak CPU and *tracemalloc* for peak memory.

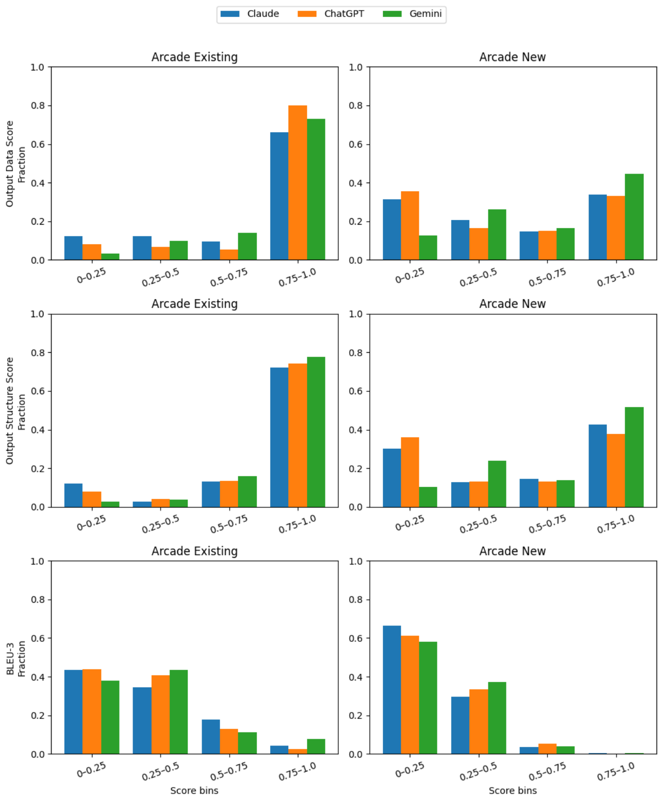


Fig. 3: Correctness scores by LLM and benchmark

V. Results

A. RQ1: How do modern LLMs perform on the multi-step data engineering code generation task?

To answer this research question, we evaluate the performance of the LLMs in our study (GPT-4o-mini, Claude-3.5-Haiku and Gemini-2.0-Flash) on the ARCADE benchmarks without any refinement (i.e. single notebook *round* and single *intent iteration*).

1) Correctness: The results indicate that all LLMs evaluated generate functionally correct and error-free code, albeit with syntax that may be different than the ground truth code, as indicated by the BLEU scores. The results also show that Gemini-2.0-Flash consistently achievs superior performance.

On the ARCADE-Existing benchmark, Gemini-2.0-Flash attained the highest BLEU-3 score (0.35), outperforming GPT-4o-mini (0.30) and Claude-3.5-Haiku (0.32). It also demonstrated the strongest alignment with expected output formats and values, with an *Output Structure Match* of 0.88 and *Output Data Match* of 0.86. Furthermore, Gemini-2.0-Flash produced the fewest errors (3%) on this benchmark, suggesting a high level of reliability.

TABLE II: Comparison of average correctness scores across models and benchmarks. Best benchmark scores in bold.

Benchmark	**ARCADE-Existing**			**ARCADE-New**		
Model	GPT-4o-mini	Claude-3.5-Haiku	Gemini-2.0-Flash	GPT-4o-mini	Claude-3.5-Haiku	Gemini-2.0-Flash
BLEU-3	0.30	0.32	**0.35**	0.22	0.22	**0.24**
Output Structure Match (%)	0.84	0.80	**0.88**	0.52	0.56	**0.69**
Output Data Match (%)	0.85	0.77	**0.86**	0.49	0.51	**0.66**
Generated Error (%)	0.08	0.12	**0.03**	0.32	0.27	**0.06**
Generated Code Lines	5.70	4.74	5.05	7.88	7.61	8.94

Performance differences were more pronounced on the more complex ARCADE-New benchmark. Gemini-2.0-Flash again led with a BLEU-3 score of 0.24, compared to 0.22 for both GPT-4o-mini and Claude. In addition, Gemini-2.0-flash achieved a 0.69 *Output Structure Match* and a 0.66 *Output Data Match*, significantly higher than the other models. Notably, the error rate of code generated by Gemini was 6%, while GPT-4o-mini and Claude exhibited substantially higher error rates of 32% and 27%, respectively.

Initially, Gemini-2.0-Flash exhibited a tendency to repeat code for each intent, inflating output length unnecessarily. After fine tuning the prompts Gemini-2.0-Flash still generated more lines of code on average on both benchmarks (e.g. 8.94 lines on ARCADE-New), however this verbosity did not compromise correctness.

It is important to note that Gemini may have had greater exposure to ARCADE data during pretraining, which could have contributed to its performance advantage. We address this concern in Section VI.

Beyond aggregate scores, a key challenge in evaluating LLMs on multi-step code generation is the high variability in performance across individual notebooks. Figure 3 shows that this variability is significantly greater for ARCADE-New compared to ARCADE-Existing across all models. For example, the proportion of intents with output data match scores below 0.25 increases fourfold for GPT-4o-mini, rising from approximately 0.1 to 0.4.

2) Runtime Performance: Table III shows the difference in mean performance run time, peak memory, and peak CPU utilization between the generated and original *intent* code after removing intents with errors and after fine tuning the prompt so that Gemini did not produce repeated code.

Overall, we find that all models generated more efficient code than the original, demonstrating improvements in runtime, memory usage, and peak CPU load across both ARCADE-Existing and ARCADE-New benchmarks. However, aside from Claude's 54.67% average runtime improvement on ARCADE-New, these gains are not statistically significant ($P > 0.05$). This is likely due to the wide variability in tasks and datasets, as shown by the high standard deviations in runtime, memory, and peak CPU differences.

B. *RQ2: What types of issues do we see in the generated code?*

To understand the limitations of LLM generated code, we analyze cases in which LLM generated code deviates from the ground truth, as determined by the *output data score*. One of the authors performed this categorization through a multi-step process that involved first describing the failure based on the generated code, output and any runtime exceptions; then grouping related descriptions together; and finally assigning tags to them. We conduct this analysis at the notebook level, due to the possibility of intents that follow a failed step being dependent on the output of that step.

More specifically, we consider all intents (and associated generated code) up to, and including, the first *intent* for which an error is encountered or the *output data score* drops below 1.0. These cases are categorized into two types: *Failure with Error* (60 instances), where the generated code leads to a run-time error; and *Failure without Error* (167 instances, 50 sampled for analysis) where the code executes successfully but produces an output that deviates from the ground truth, as measured by the *output structure score*. Table IV summarizes the issues related to *Failure with Error* and *Failure without Error*. The following sections discuss each of these categories.

1) Failure with Error: In cases where the generated code produces a runtime error, the most frequent issue is categorized as *Unsafe Operation*. Accounting for 50% of this group of errors, this category describes cases where the LLM-generated code produces errors due to not correctly accounting for missing values (`NaNs`), incorrectly invoking a method, or makes invalid assumptions about column value consistency based on the limited data available in the prompt (10 rows in our study). GPT-4o-mini and Claude-3.5-Haiku are particularly prone to this category of error (see Table IV). The second most common issue (18% of errors in this group) is *Non-compliant LLM Output*, observed primarily in Claude. In these cases, the model's output does not conform to the expected code format, resulting in syntax errors when running the parsed code. This category of errors are likely caused by limited capacity for *instruction following* rather than an outright failure to generate accurate code. The *Prompt Misinterpretation* category (8% of this group of errors) where the LLM misunderstands the components of the prompt like generated code from previous time steps and sample input data. This leads to cases where the generated code attempts to repeat a non-idempotent operation or uses the sample data passed as the input data for the task (e.g., removing an already-dropped column), leading to execution failures (*Repeated Operation*).

2) Failure without Error: In cases where the code executes without raising an error, many failures are superficial (e.g.

TABLE III: Comparison of the difference between generated and original runtime performance after excluding errors (-ve metric values mean generated code outperformed original). Statistically significant results highlighted in bold.

Benchmark	**ARCADE existing**			**ARCADE new**		
Model	GPT-4o-mini	Claude-3.5-Haiku	Gemini-2.0-Flash	GPT-4o-mini	Claude-3.5-Haiku	Gemini-2.0-Flash
Execution Runtime Metrics						
Mean original execution time (ms)	199.37	190.04	253.77	428.37	439.32	486.53
Mean diff execution time (ms)	-116.67	-94.07	-151.42	-187.25	**-240.17**	-1.73
Std diff execution time (ms)	1780.78	1757.61	2256.26	2787.53	1987.13	4501.91
Execution time improvement (%)	-58.52	-49.50	-59.67	-43.71	-54.67	-0.36
Execution time p value	0.26	0.37	0.24	0.27	0.04	0.99
Peak Memory Metrics						
Mean original peak memory (MB)	4.89	3.28	5.08	7.51	7.51	9.08
Mean diff peak memory (MB)	-1.31	-0.32	1.10	-2.74	-1.72	-3.20
Std diff peak memory (MB)	17.36	7.47	29.18	45.10	18.55	53.45
Peak memory improvement (%)	-26.86	-9.76	21.62	-36.44	-22.93	-35.20
Peak memory p value	0.19	0.47	0.50	0.32	0.12	0.25
Peak CPU Metrics						
Mean original peak cpu (%)	130.34	108.17	115.32	119.61	135.69	132.52
Mean diff peak cpu (%)	-14.36	12.36	5.61	-17.55	-22.39	-6.25
Std diff peak cpu (%)	305.60	103.41	145.82	143.73	216.93	209.15
Peak cpu improvement (%)	-11.02	11.43	4.87	-14.67	-16.50	-4.72
Peak cpu p value	0.51	0.10	0.53	0.08	0.11	0.60

TABLE IV: Issue Categories Across LLMs and Datasets

Issue	**Description**	**Dataset**	**GPT-4o-mini**	**Claude-3.5-Haiku**	**Gemini-2.0-Flash**	**Total**	**%**
		Failure with Error					
Unsafe Operation	Cases where the generated code applies to data that it is unsuited for (for example due to a type-mismatch), leading to an error	ARCADE-Existing	3	3	1	7	50.00%
		ARCADE-New	10	5	3	18	
Non-Compliant LLM Output	Cases where the LLM output does not meet output specification, causing syntax errors or parsing failures.	ARCADE-Existing	0	7	0	7	18.00/%
		ARCADE-New	0	1	1	2	
Prompt Misinterpretation	Cases where the LLM fails to recognize that it has done a non-idempotent operation at a previous step, or misuses data passed in the prompt.	ARCADE-Existing	0	2	0	2	8.00%
		ARCADE-New	1	1	0	2	
Other	Other error-causing cases (e.g., syntax errors, missing imports).	ARCADE-Existing	2	3	0	5	24.00%
		ARCADE-New	3	2	2	7	
		Failure without Error					
Correct	Functionally correct solutions that differ from the reference in trivial ways, such as assigning a result to a variable instead of printing it directly.	ARCADE-Existing	1	2	2	5	—%
		ARCADE-New	4	9	5	18	
Ambiguity	Underspecified intents with multiple plausible interpretations.	ARCADE-Existing	1	1	1	3	40.74%
		ARCADE-New	2	3	3	8	
Information Disadvantage	Cases where the LLM-generated code makes assumptions about the nature of the data, leading to incorrect output following operations like filtering	ARCADE-Existing	0	0	0	0	37.04%
		ARCADE-New	3	1	6	10	
Task Misunderstanding	Cases where the LLM generates code that misinterprets an *intent*. These are deemed to be clear, in contrast with intents in the 'ambiguity' category.	ARCADE-Existing	0	0	0	0	22.22%
		ARCADE-New	2	2	2	6	

printing out the result of an operation rather than storing it in a variable), as represented by the *correct* category. These instances highlight limitations in the evaluation metric rather than actual model failures and are not assigned any percentage score in Table IV. More substantial issues arise from *Ambiguity* in the *intent*, accounting for 40.74% of this category of errors. The generated code in these cases represents a plausible interpretation of the user *intent*, but does not match the reference implementation. Another frequent issue involves incorrect inference of input column data types, leading the model to apply operations that are required by the *intent* on a specified data type. We also identify failures stemming from issues that we dub *Information Disadvantage* (37.04% of this category of errors), where the model makes flawed assumptions based on the limited data it observes, such as applying filtering logic that fails to generalize due to an incomplete view of column values. Finally, some errors are caused by a clear *Misunderstanding* of the *intent*, where the generated code reflects an incorrect or irrelevant interpretation of the user's instruction. These account for 22.22% of errors in this category.

Across both categories, a common theme is the need to generalize generated code to fit data that would be infeasible to include in the LLMs context directly. Errors in the *Prompt Misinterpretation* category occur later in the notebooks, after multiple intents have been processed, suggesting potential dif-

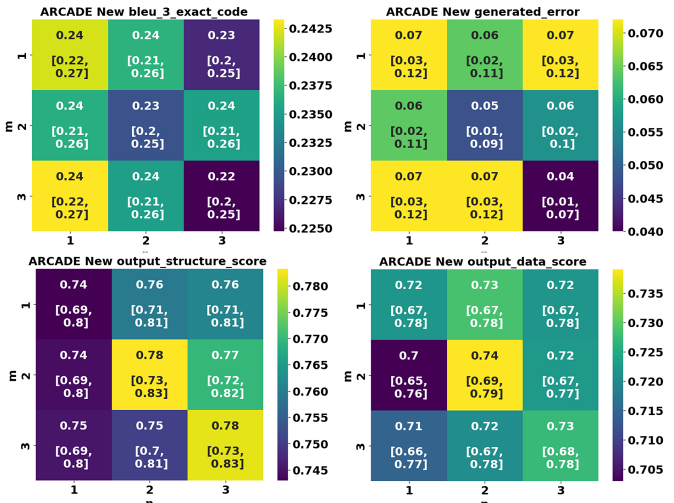


Fig. 4: Refinement correctness scores for Gemini-2.0-Flash on ARCADE-New. Refinement seems to reduce error rate which in turn improves output structure and data scores for increasing (*m*) and (*n*).

ficulty with handling longer contexts. Techniques for managing context windows, especially for incrementally constructed tasks involving both data and code, could be valuable. To address ambiguity, it might be useful to prompt an LLM to interpret an *intent* in different ways, allowing the user to make a choice on what solution best fits their description, given the inherent imprecision of natural language.

C. RQ3: Does increasing the amount of iterative refinement improve correctness and performance?

To address this question, we employ an *iterative refinement* process that allows the LLM being evaluated refine the generated code at both the *intent* and *notebook* levels. We then measure the impact that these refinements have on output correctness and execution performance.

The *iterative refinement* process proceeds through m *rounds* for each notebook. In every *round*, the LLM refines the code generated for each intent n times. During each *iteration*, the prompt is constructed using the prior execution context and output. The system then generates new code, executes it to produce an output, and appends both the code and its result (we restrict this to 10 rows) to the prompt for the next iteration. This process is visually represented in Figure 2.

TABLE V: Output data score Mixed Linear Model regression results for m by n experiments. Improvements in bold.

	Coef.	Std. Err.	z	p-value	[0.025	0.975]
Intercept	0.722	0.027	26.468	0.000	0.669	0.775
m = 2	-0.019	0.017	-1.112	0.266	-0.052	0.014
m = 3	-0.008	0.017	-0.440	0.660	-0.041	0.026
n = 2	**0.007**	0.017	0.384	0.701	-0.027	0.040
n = 3	-0.000	0.017	-0.007	0.994	-0.034	0.033
m = 2, n = 2	**0.029**	0.024	1.223	0.221	-0.018	0.077
m = 3, n = 2	-0.001	0.024	-0.022	0.982	-0.048	0.047
m = 2, n = 3	**0.019**	0.024	0.807	0.420	-0.028	0.067
m = 3, n = 3	**0.013**	0.024	0.537	0.591	-0.034	0.060

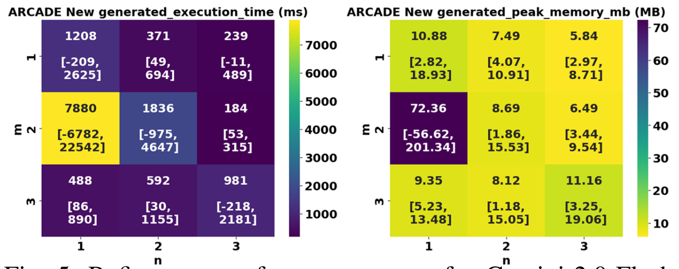


Fig. 5: Refinement performance scores for Gemini-2.0-Flash on ARCADE-New. Mean runtime and memory scores generally improve with more rounds (*m*) and iterations (*n*), but are sensitive to outliers. Intents with errors are excluded.

Figures 4 and 5 show the correctness and runtime performance of the generated code under different levels of *notebook* (m) and *user intent* (n) refinement. We observe that generated error tends to decrease as refinement increases (0.07 for m=n=1 vs 0.04 for m=n=3). We hypothesize this is due to the LLM being able to correct previous errors which are passed back as context during refinement. More error free code in turn improves the *output structure scores* (up to 2% points) and *output data scores* (up to 4% points). The BLEU-3 scores remain relatively stable, suggesting that the refinement process does not increase the semantic similarity of the code to the ground truth. However, these results exhibit high variability due to individual notebooks (see 95% confidence intervals). Table V shows the regression results of a Mixed Linear Model for the same notebook intents under different *m* and *n* treatments. The results indicate that despite the trend towards improvement, these gains are not statistically significant. For example, after adjusting for between-notebook variation, we observe a 2.9% improvement in *output data match* scores (0.029 regression coefficient) after 2 additional *rounds* (m=2) or 2 additional *iterations* (n=2).

We observe, as represented in Figure 5, that increasing iterations (*n*) tends to improve intent execution time and memory use. However, adding a second notebook round (*m*) reduces performance, while a third round brings it back to baseline. This pattern suggests that while refinement has the potential to improve performance, excessive notebook-level refinement may introduce diminishing returns, highlighting the need for a balanced approach.

We conclude that refinement tends to improve correctness through the ability to fix code errors and also tends to improve runtime performance. However, this is highly sensitive to notebook variability, and further research is needed to identify the point at which refinement is no longer beneficial.

VI. Discussion

A. Further Experiments on Spider2-intents

To control for possible pretraining exposure to the ARCADE dataset (released in 2022), we validate our findings on the manually constructed Spider2-intents dataset described in Section IV-A1. This evaluation follow the same process described in Section III (Case Study). The results of this evaluation are presented in Table VI. We observe that LLM performance is comparable between ARCADE and Spider2-intents. Namely, comparing ARCADE-New to Spider2-intents the best scores are: BLEU-3 of 0.24 vs 0.35, output structure of 0.69 vs 0.74, data match of 0.66 vs 0.80, and the same generated error of 0.06. Since ARCADE-New performance is lower than that of Spider2-intents it implies data leakage is not likely a significant factor in ARCADE results as Spider2-intents has not been seen by LLMs and was designed with similar task complexity in mind.

TABLE VI: Spider2-intents Evaluation Results

Model	**GPT-4o-mini**	**Claude-3.5-Haiku**	**Gemini-2.0-Flash**
BLEU-3	0.34	0.33	**0.35**
Output Structure Match	0.58	0.60	**0.74**
Output Data Match	0.60	0.64	**0.80**
Generated Error	0.29	0.24	**0.06**
Generated Code Lines	**3.55**	4.69	4.26

B. Challenges with Complex 'Single-Step' Tasks

In an additional experiment using the tasks collected from the Spider 2.0-lite dataset, we found that while *iterative refinement* improves performance over simple prompting by up to 18% but overall performance remains low. VII shows the results of this experiment, with Gemini-2.0-Flash* indicating the performance of Gemini-2.0-Flash (the best performing model in the zero-shot setting) after *iterative refinement* using the optimal parameters for *round*(2) and *iteration*(1) as determined in RQ3. Compared to the results on *Spider2-intents*, the average output data score is 79% lower, suggesting that automated decomposition of complex tasks into simpler subtasks could be a promising strategy for improving performance on tasks with the level of complexity of those in Spider 2.0-lite.

TABLE VII: LLM Performance on Spider 2.0-lite

Model	**GPT-4o-mini**	**Claude-3.5-Haiku**	**Gemini-2.0-Flash**	**Gemini-2.0-Flash***
Output Structure Match	0.43	0.14	**0.51**	0.51
Output Data Match	0.11	0.06	**0.15**	0.17
Generated Error	0.45	0.67	**0.26**	0.16

VII. Threats to Validity

Our study involves a number of assumptions and simplifications made for practical reasons that may influence the generalizability and correctness of our results.

- **Execution Environment:** We used Kaggle, a cloud platform as our execution environment. To mitigate the impact of the inherent variability of the execution environment on our results, the results reported here are from an average of three notebook runs. However, we acknowledge that it does not completely eliminate the impact of environment variability on our results.
- **Sampling for Qualitative Evaluation:** Taking a limited number of rows of output for correctness evaluation is limited and sometimes generates false positives.
- **Prompt Structure:** We did not fully account for the impact of our prompt structure on the performance of the different models we experimented with. As models are sensitive to prompt structure, this could affect the validity and reproducibility of our results. In particular, little model-specific prompt optimization was performed which may have contributed to non-compliant outputs observed for some models.

VIII. Conclusion

In this study, we performed an empirical study of how LLMs perform on multi-step data engineering code generation tasks; categorized the issues with the generated code, and evaluated the impact of iterative refinement.

Our results reveal that modern LLMs are capable of generating code that satisfies user requirements, reaching *output data match* scores of up to 80% on a newly created benchmark. In terms of performance, the LLMs also showed trends toward reduced runtime, memory, and cpu usage; however improvements were not statistically significant.

The results of our issue categorization (on cases where the LLM-generated code fails) reveal that limitations in data exploration hinder LLM performance, often leading to generated code that does not sufficiently account for variations in the data source, and that ambiguity remains a significant challenge.

Furthermore, our results demonstrate that *iterative refinement* can improve correctness and runtime performance in code generation tasks. Specifically, we find that the optimal configuration involves a **two rounds** of notebook refinement with **two iterations** per user *intent* resulting in 2.9% improvement in output data match scores on average.

Based on the results of our study, we propose further research into prompt optimization for different models, reward signals for *iterative refinement*, and dynamic selection of source data examples. Future work could also investigate the performance overhead and cognitive demands that using LLMs for this task places on developers, as well as develop metrics to identify which data transformation tasks LLMs are most effective at handling. Furthermore, given success on Spider2-intents, future research could extend our methodology to transform Spider 2.0 tasks to multi-step tasks and open up a new avenue for solving this challenging benchmark.

References

[1] Anaconda. *State of Data Science: AI and Open Source at Work*. 2025. URL: https://www.anaconda.com/resources/report/state-of-data-science-report.

[2] Dong Huang et al. *EffiBench: Benchmarking the Efficiency of Automatically Generated Code*. 2025. arXiv: 2402.02037 [cs.SE]. URL: https://arxiv.org/abs/2402.02037.

[3] Sanidhya Vijayvargiya et al. *Interactive Agents to Overcome Ambiguity in Software Engineering*. 2025. arXiv: 2502.13069 [cs.AI].

[4] T. Yu et al. "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task". In: *arXiv preprint arXiv:1809.08887* (2018).

[5] Mohammadreza Pourreza and Davood Rafiei. "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction". In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh et al. Vol. 36. Curran Associates, Inc., 2023, pp. 36339–36348.

[6] F. Lei et al. "Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows". In: *arXiv preprint arXiv:2411.07763* (2024).

[7] Minghang Deng et al. "Reforce: A Text-to-SQL agent with self-refinement, format restriction, and column exploration". In: *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*. 2025.

[8] Z. Hong et al. "Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL". In: *arXiv preprint arXiv:2406.08426* (2024).

[9] J. Li et al. "Can LLM Already Serve as a Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs". In: *Advances in Neural Information Processing Systems*. Vol. 36. 2024.

[10] J. Austin et al. "Program Synthesis with Large Language Models". In: *arXiv preprint arXiv:2108.07732* (2021). URL: https://arxiv.org/abs/2108.07732.

[11] N. Jain et al. "Jigsaw: Large Language Models Meet Program Synthesis". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*. 2022, pp. 219–230. DOI: 10.1145/3510003.3510203.

[12] D. Nam et al. "Predictive Synthesis of API-Centric Code". In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022)*. 2022, pp. 40–49. DOI: 10.1145/3520312.3534866.

[13] L. Ma et al. "LLMs with User-Defined Prompts as Generic Data Operators for Reliable Data Processing". In: *2023 IEEE International Conference on Big Data (BigData)*. 2023, pp. 3144–3148. DOI: 10.1109/BigData59044.2023.10386472.

[14] S. Schmidgall et al. "Agent Laboratory: Using LLM Agents as Research Assistants". In: *arXiv preprint arXiv:2501.04227* (2025).

[15] C. Lu et al. "The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery". In: *arXiv preprint arXiv:2408.06292* (2024).

[16] A. Grosnit et al. "Large Language Models Orchestrating Structured Reasoning Achieve Kaggle Grandmaster Level". In: *arXiv preprint arXiv:2411.03562* (2024). URL: https://arxiv.org/abs/2411.03562.

[17] Y. Lai et al. "DS-1000: A natural and reliable benchmark for data science code generation". In: *International Conference on Machine Learning*. 2022.

[18] Pengcheng Yin et al. "Natural language to code generation in interactive data science notebooks". In: *arXiv preprint arXiv:2212.09248* (2022).

[19] Junjie Huang et al. "Execution-based evaluation for data science code generation models". In: *arXiv preprint arXiv:2211.09374* (2022).

[20] Yeming Wen et al. "Grounding data science code generation with input-output specifications". In: *arXiv preprint arXiv:2402.08073* (2024).

[21] T. T. Quoc et al. "An Empirical Study on Self-Correcting Large Language Models for Data Science Code Generation". In: *arXiv preprint arXiv:2408.15658* (2024). DOI: 10.48550/arXiv.2408.15658.

[22] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. "JuICe: A Large Scale Distantly Supervised Dataset for Open Domain Context-based Code Generation". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Ed. by Kentaro Inui et al. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 5436–5446. DOI: 10.18653/v1/D19-1546. URL: https://aclanthology.org/D19-1546/.

[23] Greta Dolcetti et al. "Helping LLMs Improve Code Generation Using Feedback from Testing and Static Analysis". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2023. DOI: 10.1145/3576915.3616413.

[24] Eric Zelikman et al. "Self-taught optimizer (stop): Recursively self-improving code generation". In: *First Conference on Language Modeling*. 2024.

[25] OpenAI. *GPT-4o-mini Model Overview*. Accessed: 2025-06-28. 2024. URL: https://platform.openai.com/docs/models/gpt-4o-mini.

[26] Anthropic. *Claude 3.5 Haiku Model Card*. Accessed: 2025-06-28. 2025. URL: https://www.anthropic.com/claude/haiku.

[27] Google DeepMind. *Gemini 1.5 and Gemini Flash Models*. Accessed: 2025-06-28. 2024. URL: https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash.

[28] Nathalia Nascimento et al. *LLM4DS: Evaluating Large Language Models for Data Science Code Generation*. 2024. arXiv: 2411.11908 [cs.SE].

[29] Jia Li et al. *EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories*. 2024. arXiv: 2404.00599 [cs.CL].

[30] Jiawei Guo et al. *CodeEditorBench: Evaluating Code Editing Capability of Large Language Models*. 2025. arXiv: 2404.03543 [cs.SE].