Code Language Models

Speaker: Jiawei Shen   Eric Liu   Yiwen Lu
Presentation date: 2024/10/17

Popular code language models

OpenAI Codex

Contents

1. Code Llama: Open Foundation Models for Code
2. Planning with Large Language Models
3. Teaching Large Language Models to Self-Debug
4. SELFEVOLVE: A Code Evolution Framework via Large Language Models

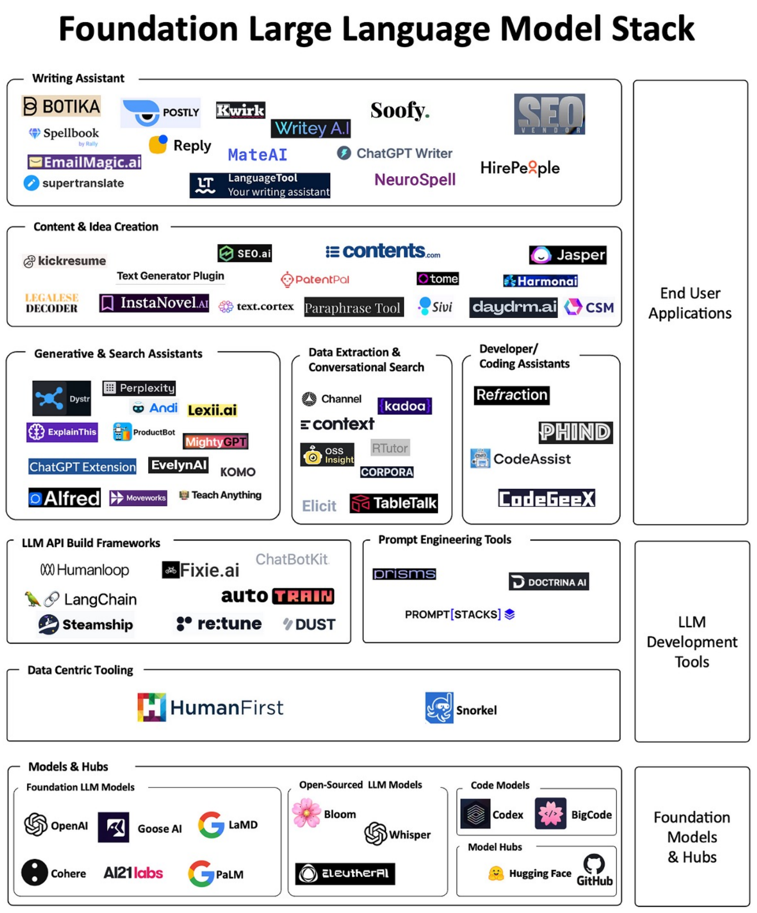
Code Llama: Open Foundation Models for Code

https://arxiv.org/abs/2308.12950

1. **Code Llama: Open Foundation Models for Code**
2. Planning with Large Language Models
3. Teaching Large Language Models to Self-Debug
4. SELFEVOLVE: A Code Evolution Framework via Large Language Models

1. Background

Large language models (LLMs) power a rapidly increasing number of applications, having reached a proficiency in natural language that allows them to be commanded and prompted to perform a variety of tasks.



2. Contribution

Different variants of Code Llama: Three main variants are provided, each with three sizes (7B, 13B, and 34B parameters):

Code Llama: Basic code generation model.

Code Llama - Python: A version customized for Python.

Code Llama - Instruct: A version that combines human instructions and self-generated code synthesis data.

2. Contribution

This paper thoroughly evaluates the model on major code generation benchmarks such as HumanEval, MBPP, APPS, and the multi-language version of HumanEval (MultiPL-E). Code Llama performs well in these tests and sets a new standard for open source LLMs.

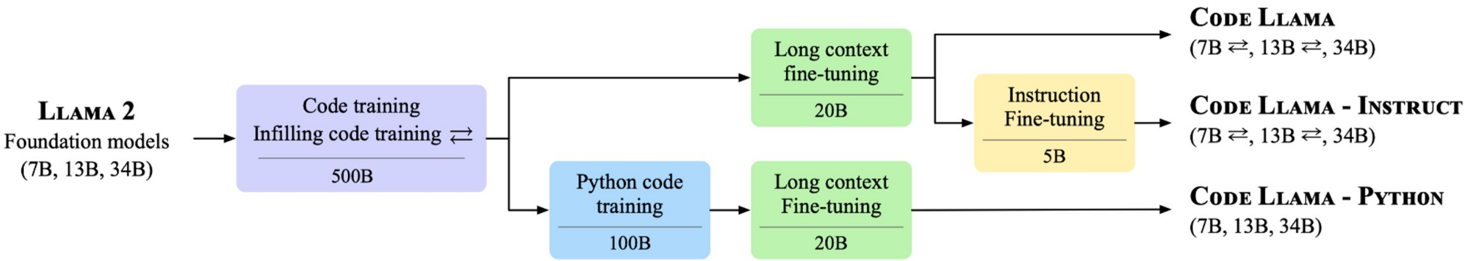
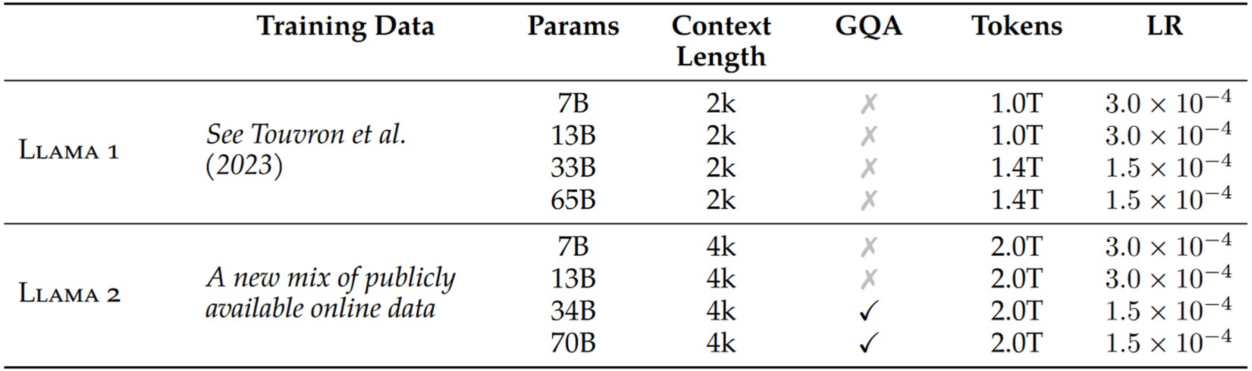


Figure 2: **The Code Llama specialization pipeline.** The different stages of fine-tuning annotated with the number of tokens seen during training. Infilling-capable models are marked with the ⇄ symbol.

3. Llama2

Most of the pre-training settings and model architecture from Llama 1 are adopted. The standard Transformer architecture is used, pre-normalization is applied using RMSNorm, SwiGLU activation function and rotated position embeddings are used. The main architectural differences from Llama 1 include increased context length and grouped query attention (GQA).



	Training Data	Params	Context Length	GQA	Tokens	LR
Llama 1	*See Touvron et al. (2023)*	7B	2k	✗	1.0T	3.0×10^{-4}
		13B	2k	✗	1.0T	3.0×10^{-4}
		33B	2k	✗	1.4T	1.5×10^{-4}
		65B	2k	✗	1.4T	1.5×10^{-4}
Llama 2	*A new mix of publicly available online data*	7B	4k	✗	2.0T	3.0×10^{-4}
		13B	4k	✗	2.0T	3.0×10^{-4}
		34B	4k	✓	2.0T	1.5×10^{-4}
		70B	4k	✓	2.0T	1.5×10^{-4}

3. Llama2: Grouped Query Attention (GQA)

Before understanding what GQA is, we need to know two more concepts: MHA and MQA

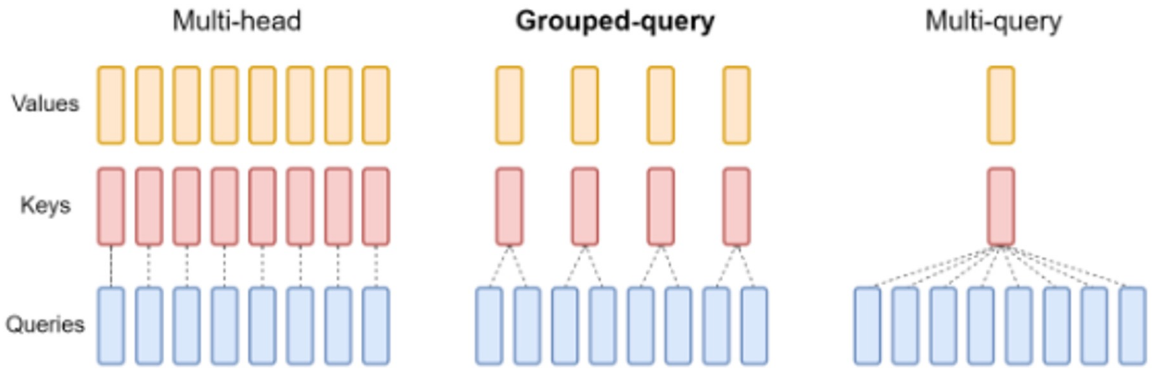


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

MHA

Multi-Head Attention (MHA) splits input data into multiple heads, each independently performing attention calculations with distinct weight matrices to capture different features. The Query (Q), Key (K), and Value (V) components align across heads, and their outputs are summed to generate the final resu

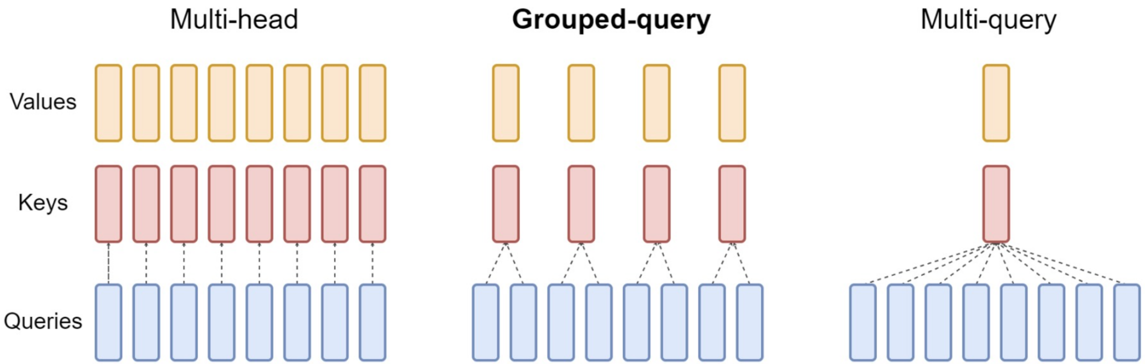


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

MQA

Multi-Query Attention (MQA) simplifies attention by keeping Q multi-headed while sharing K and V across all heads within each layer, reducing the number of K and V matrices to one per layer. This approach, as seen in models like ChatGLM2-6B, enhances computational efficiency without compromising performance.

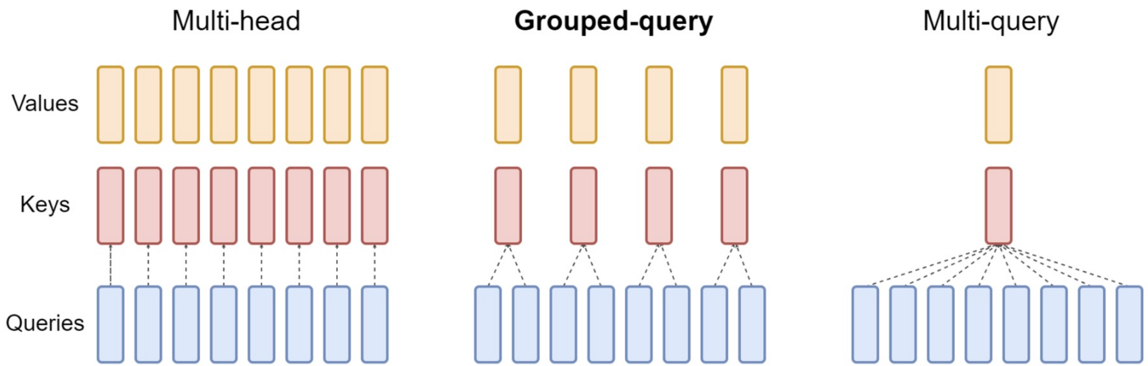


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

GQA

Although MQA can minimize the cache space required for KV Cache, it is conceivable that the reduction of parameters means a decrease in accuracy. Therefore, in order to make a trade-off between accuracy and calculation, GQA (Group Query Attention) came into being. That is, Q is still multi-head, but K, V are shared in groups, which not only reduces the cache space required for K, V cache, but also exposes most parameters without serious loss of

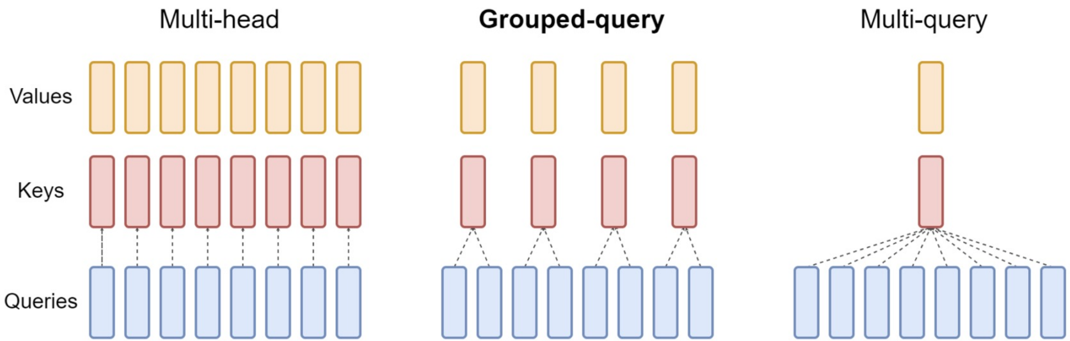


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

4.1 Codellama technique: Dataset

Available code, of which 8% of the sample data comes from natural language datasets related to code. These datasets contain code discussions and code snippets, which help the model understand natural language.

Dataset: The training data mainly comes from publicly available code data, as well as natural language data related to code.

Tokenization: Tokenization is performed using the same tokenizer as Llama 2.

4.2 Codellama technique: Infilling

Filling is the task of predicting the missing parts of a program, and the Code Llama model achieves this function through a specific training method. The training uses causal masking technology to move parts of the training sequence to the end of the sequence and autoregressively predict the rearranged sequence.

Segmentation: The training documents are segmented into prefix, middle part and suffix at the character level.

Masking: The masking transformation is applied with a certain probability, and only operates on documents that do not exceed the model context length.

4.3 Codellama technique: Long context fine-tuning

The long context fine-tuning stage is specifically proposed to improve the model's ability to handle long sequences. By modifying the parameters of the RoPE position embedding, the maximum context length of the model is extended from 4,096 tokens to 100,000 tokens.

Sequence length: The sequence length used during training is 16,384 tokens.

RoPE adjustment: The frequency of the rotation embedding is adjusted to accommodate longer sequences

4.4 Codellama technique: Instruction fine-tuning

Code Llama - Instruct The model is fine-tuned on Code Llama with three additional data to better answer questions.

Three different types of data:

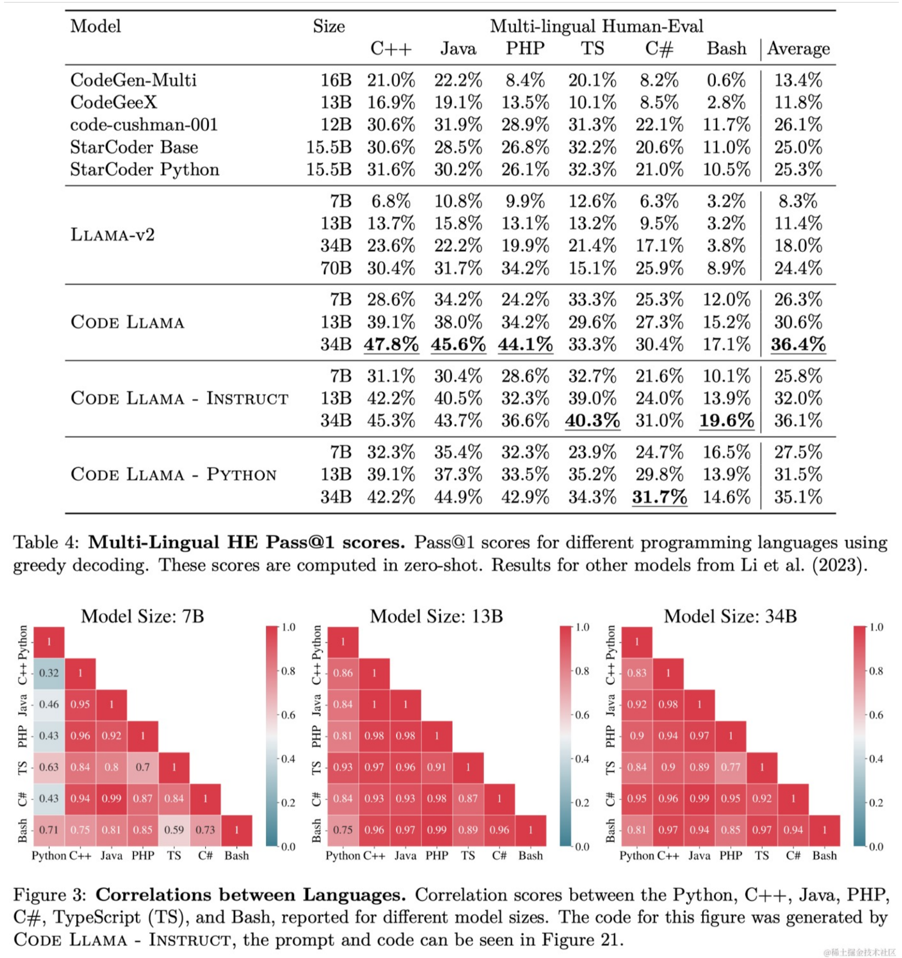
Proprietary dataset: Fine-tuning dataset using RLHF V5 instructions from Llama 2. Improve the security of model output and enhance the model's responsiveness to user instructions.

Self-guided dataset: Select data through execution feedback and build a self-guided dataset. Use llama2 70B to produce instruction questions, coda llama 7B to answer questions, and select the correct ones as the final dataset.

Recap: Prevent the model from regressing in general coding and language understanding capabilities, and use a small part of the code dataset and natural language dataset for training

4.5 Evaluation

Different Models



Model	Size	Multi-lingual Human-Eval C++	Java	PHP	TS	C#	Bash	Average
CodeGen-Multi	16B	21.0%	22.2%	8.4%	20.1%	8.2%	0.6%	13.4%
CodeGeeX	13B	16.9%	19.1%	13.5%	10.1%	8.5%	2.8%	11.8%
code-cushman-001	12B	30.6%	31.9%	28.9%	31.3%	22.1%	11.7%	26.1%
StarCoder Base	15.5B	30.6%	28.5%	26.8%	32.2%	20.6%	11.0%	25.0%
StarCoder Python	15.5B	31.6%	30.2%	26.1%	32.3%	21.0%	10.5%	25.3%
LLAMA-v2	7B	6.8%	10.8%	9.9%	12.6%	6.3%	3.2%	8.3%
	13B	13.7%	15.8%	13.1%	13.2%	9.5%	3.2%	11.4%
	34B	23.6%	22.2%	19.9%	21.4%	17.1%	3.8%	18.0%
	70B	30.4%	31.7%	34.2%	15.1%	25.9%	8.9%	24.4%
CODE LLAMA	7B	28.6%	34.2%	24.2%	33.3%	25.3%	12.0%	26.3%
	13B	39.1%	38.0%	34.2%	29.6%	27.3%	15.2%	30.6%
	34B	**47.8%**	**45.6%**	**44.1%**	33.3%	30.4%	17.1%	**36.4%**
CODE LLAMA - INSTRUCT	7B	31.1%	30.4%	28.6%	32.7%	21.6%	10.1%	25.8%
	13B	42.2%	40.5%	32.3%	39.0%	24.0%	13.9%	32.0%
	34B	45.3%	43.7%	36.6%	**40.3%**	31.0%	**19.6%**	36.1%
CODE LLAMA - PYTHON	7B	32.3%	35.4%	32.3%	23.9%	24.7%	16.5%	27.5%
	13B	39.1%	37.3%	33.5%	35.2%	29.8%	13.9%	31.5%
	34B	42.2%	44.9%	42.9%	34.3%	**31.7%**	14.6%	35.1%

Table 4: **Multi-Lingual HE Pass@1 scores.** Pass@1 scores for different programming languages using greedy decoding. These scores are computed in zero-shot. Results for other models from Li et al. (2023).

Model Size: 7B

	Python	C++	Java	PHP	TS	C#	Bash
Python	1						
C++	0.32	1					
Java	0.46	0.95	1				
PHP	0.43	0.96	0.92	1			
TS	0.63	0.84	0.8	0.7	1		
C#	0.43	0.94	0.99	0.87	0.84	1	
Bash	0.71	0.75	0.81	0.85	0.59	0.73	1

Model Size: 13B

	Python	C++	Java	PHP	TS	C#	Bash
Python	1						
C++	0.86	1					
Java	0.84	1	1				
PHP	0.81	0.98	0.98	1			
TS	0.93	0.97	0.96	0.91	1		
C#	0.84	0.93	0.93	0.98	0.87	1	
Bash	0.75	0.96	0.97	0.99	0.89	0.96	1

Model Size: 34B

	Python	C++	Java	PHP	TS	C#	Bash
Python	1						
C++	0.83	1					
Java	0.92	0.98	1				
PHP	0.9	0.94	0.97	1			
TS	0.84	0.9	0.89	0.77	1		
C#	0.95	0.96	0.99	0.95	0.92	1	
Bash	0.81	0.97	0.94	0.85	0.97	0.94	1

Figure 3: **Correlations between Languages.** Correlation scores between the Python, C++, Java, PHP, C#, TypeScript (TS), and Bash, reported for different model sizes. The code for this figure was generated by CODE LLAMA - INSTRUCT, the prompt and code can be seen in Figure 21.

4.5 Evaluation

Safety

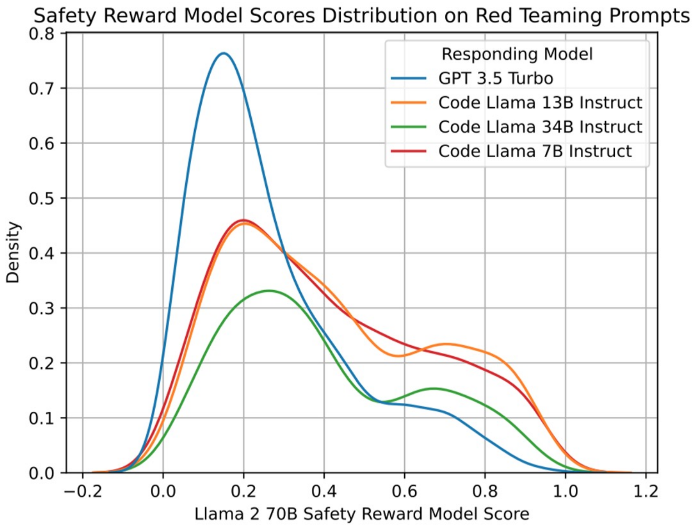


Figure 7: KDE plot of the risk score output by the Llama 2 safety reward model on prompts with clear intent specific to code risk created by red teamers with background in cybersecurity and malware generation.

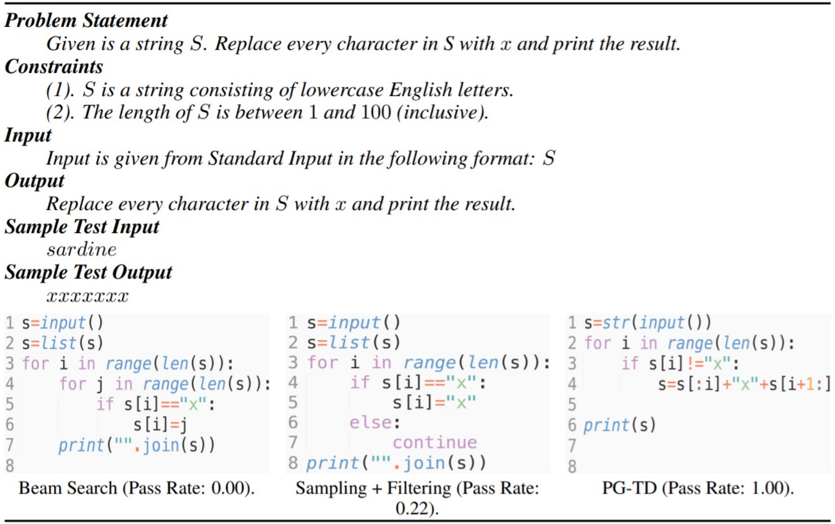
Planning with Large Language Models

https://arxiv.org/abs/2303.05510

1. Code Llama: Open Foundation Models for Code
2. **Planning with Large Language Models**
3. Teaching Large Language Models to Self-Debug
4. SELFEVOLVE: A Code Evolution Framework via Large Language Models

1. Problem/Background

- Code generation methods like beam search or sampling often generate programs that are incorrect
- Previous methods don't test the code until it is fully generated



Problem Statement

Given is a string S. Replace every character in S with x and print the result.

Constraints

(1). S is a string consisting of lowercase English letters.
(2). The length of S is between 1 and 100 (inclusive).

Input

Input is given from Standard Input in the following format: S

Output

Replace every character in S with x and print the result.

Sample Test Input

$sardine$

Sample Test Output

$xxxxxxx$

```
s=input()
s=list(s)
for i in range(len(s)):
    for j in range(len(s)):
        if s[i]=="x":
            s[i]=j
    print("".join(s))
```

Beam Search (Pass Rate: 0.00).

```
s=input()
s=list(s)
for i in range(len(s)):
    if s[i]=="x":
        s[i]="x"
    else:
        continue
print("".join(s))
```

Sampling + Filtering (Pass Rate: 0.22).

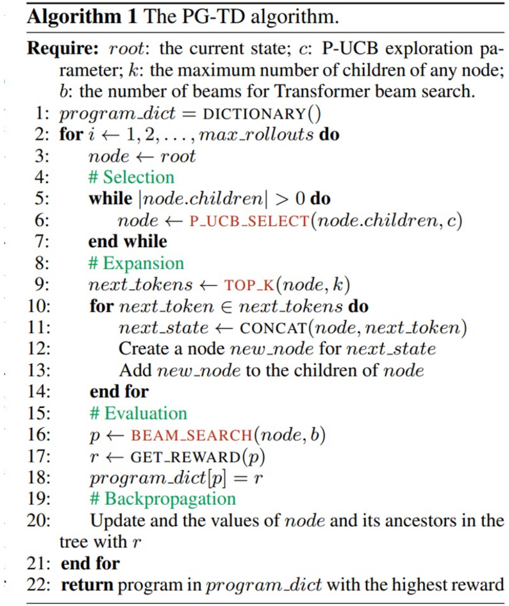
```
s=str(input())
for i in range(len(s)):
    if s[i]!="x":
        s=s[:i]+"x"+s[i+1:]

print(s)
```

PG-TD (Pass Rate: 1.00).

Figure 1: A code generation example for competitive programming, with the problem description (top) and the programs generated by baseline algorithms and our PG-TD algorithm (bottom).

2. Planning-Guided Transformer Decoding

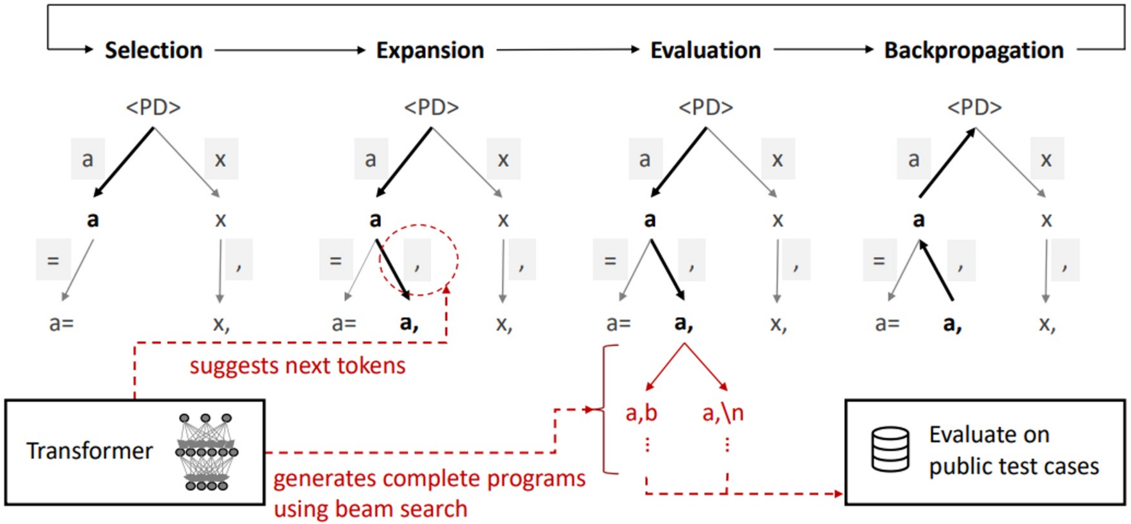
- Also a Transformer-based generation model
- Integrates the Monte Carlo Tree Search (MCTS)
- Four steps:
  - Selection
  - Expansion
  - Evaluation
  - Backpropagation



Algorithm 1 The PG-TD algorithm.

```
Require: $root$: the current state; $c$: P-UCB exploration parameter; $k$: the maximum number of children of any node; $b$: the number of beams for Transformer beam search.
$program\_dict = \text{DICTIONARY}()$
for $i \leftarrow 1, 2, \ldots, max\_rollouts$ do
    $node \leftarrow root$
    # Selection
    while $|node.children| > 0$ do
        $node \leftarrow \text{P\_UCB\_SELECT}(node.children, c)$
    end while
    # Expansion
    $next\_tokens \leftarrow \text{TOP\_K}(node, k)$
    for $next\_token \in next\_tokens$ do
        $next\_state \leftarrow \text{CONCAT}(node, next\_token)$
        Create a node $new\_node$ for $next\_state$
        Add $new\_node$ to the children of $node$
    end for
    # Evaluation
    $p \leftarrow \text{BEAM\_SEARCH}(node, b)$
    $r \leftarrow \text{GET\_REWARD}(p)$
    $program\_dict[p] = r$
    # Backpropagation
    Update and the values of $node$ and its ancestors in the tree with $r$
end for
return program in $program\_dict$ with the highest reward
```

2.1 Planning-Guided Transformer Decoding



2.2 Caching/Information Sharing

- Tree Structure Caching
  - Stores the search tree built during Monte Carlo Tree Search (MCTS) to avoid recomputing the same partial programs
- Sequence Caching
  - Saves complete programs generated during evaluation so that future iterations can reuse these sequences if the same prefix is encountered

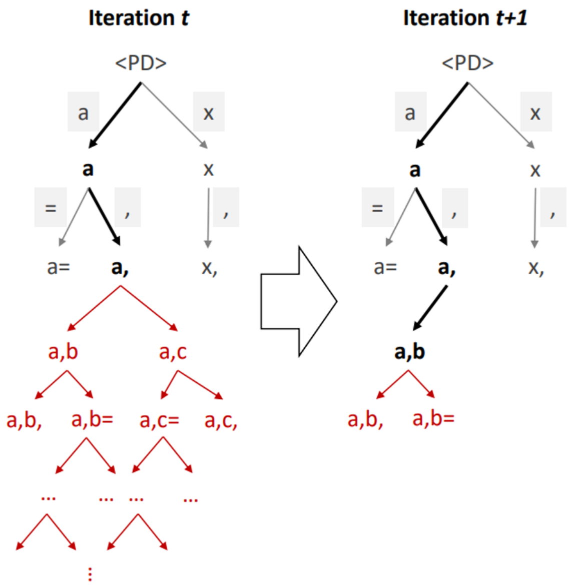
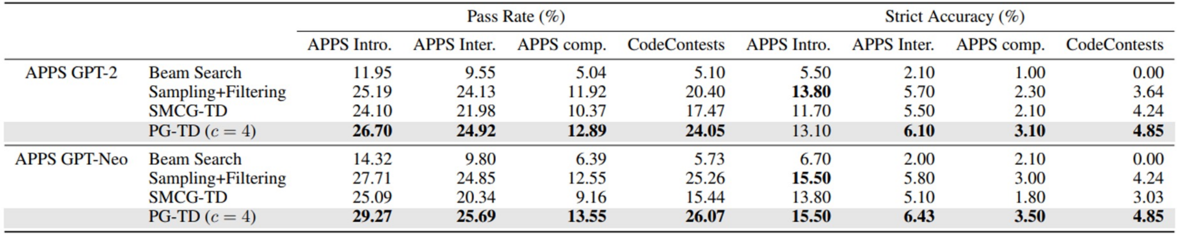


Figure 3: Illustration for caching in the PG-TD algorithm. The tree search part is visualized in black color and the Transformer beam search part is in red color.

3. Results

- APPS, CodeContests are benchmark coding datasets
- Pass Rate: How many test cases were passed
- Strict Accuracy: How many problems all tests cases were passed



		Pass Rate (%)				Strict Accuracy (%)			
		APPS Intro.	APPS Inter.	APPS comp.	CodeContests	APPS Intro.	APPS Inter.	APPS comp.	CodeContests
APPS GPT-2	Beam Search	11.95	9.55	5.04	5.10	5.50	2.10	1.00	0.00
	Sampling+Filtering	25.19	24.13	11.92	20.40	**13.80**	5.70	2.30	3.64
	SMCG-TD	24.10	21.98	10.37	17.47	11.70	5.50	2.10	4.24
	PG-TD ($c = 4$)	**26.70**	**24.92**	**12.89**	**24.05**	13.10	**6.10**	**3.10**	**4.85**
APPS GPT-Neo	Beam Search	14.32	9.80	6.39	5.73	6.70	2.00	2.10	0.00
	Sampling+Filtering	27.71	24.85	12.55	25.26	**15.50**	5.80	3.00	4.24
	SMCG-TD	25.09	20.34	9.16	15.44	13.80	5.10	1.80	3.03
	PG-TD ($c = 4$)	**29.27**	**25.69**	**13.55**	**26.07**	**15.50**	**6.43**	**3.50**	**4.85**

Table 1: Results of PG-TD and other algorithms. The maximum number of Transformer generations for all algorithms is 256.

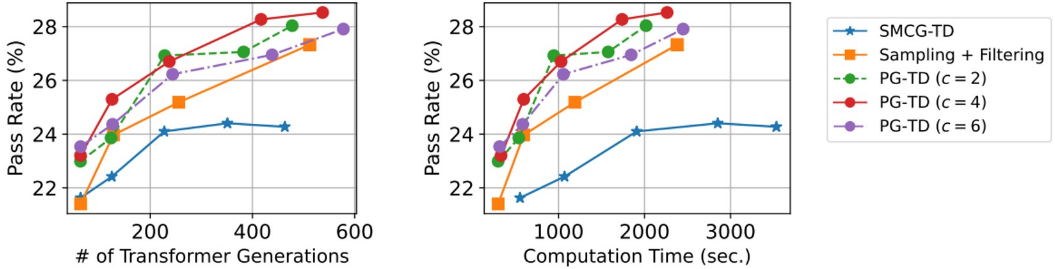


Figure 4: Pass rates of PG-TD and baseline algorithms vs. the number of Transformer generations (left) and the computation time (middle) on the introductory problems in the APPS test dataset (1000 problems), using the APPS GPT-2 Transformer model.

3.1 Results

- Having the two caching mechanism reduces computation time almost twofold
- Finetuning the transformer models using the solutions created by PG-TD improves their performance
- The model is able to perform other objectives without sacrificing pass rate significantly
- The model is able to use automatically generated test cases to still outperform other methods

4. Limitations

- Reliance on test cases:
  - PG-TD depends a lot on the availability of test cases for evaluating generated programs.
- Computational Cost:
  - Although the caching mechanisms improve efficiency, PG-TD still requires more computational resources than standard beam search because of the many calls to the Transformer during the planning process.

5. Key Takeaways/Contributions

- Planning
  - PG-TD adds planning to code generation so the Transformer can generate better programs by looking ahead during the decoding process
- Caching
  - The algorithm design includes caching techniques that reduce repetitive computations
- Model Flexibility
  - PG-TD works with any Transformer-based code generation model without requiring more training

Teaching Large Language Models to Self-Debug

https://arxiv.org/abs/2304.05128

1. Code Llama: Open Foundation Models for Code
2. Planning with Large Language Models
3. **Teaching Large Language Models to Self-Debug**
4. SELFEVOLVE: A Code Evolution Framework via Large Language Models

1. Introduction

1. **Problems:**
   - LLMs perform well in code generation, but struggle to generate **attempt**, especially for complex tasks.
2. **Idea:**
   - Programmers debug by reviewing execution and explaining code, often using **rubber duck debugging**.

Programmers explain their code line-by-line to identify mistakes. By explaining it simply, errors become more obvious, helping to debug the code without external guidance.

3. **SELF-DEBUGGING:**
   - Teaches a large language model to debug its predicted program via **few-shot** demonstrations
   - Without any human feedback on the code correctness or error messages, the model is able to identify its mistakes by investigating the execution results and explaining the generated code in natural language.

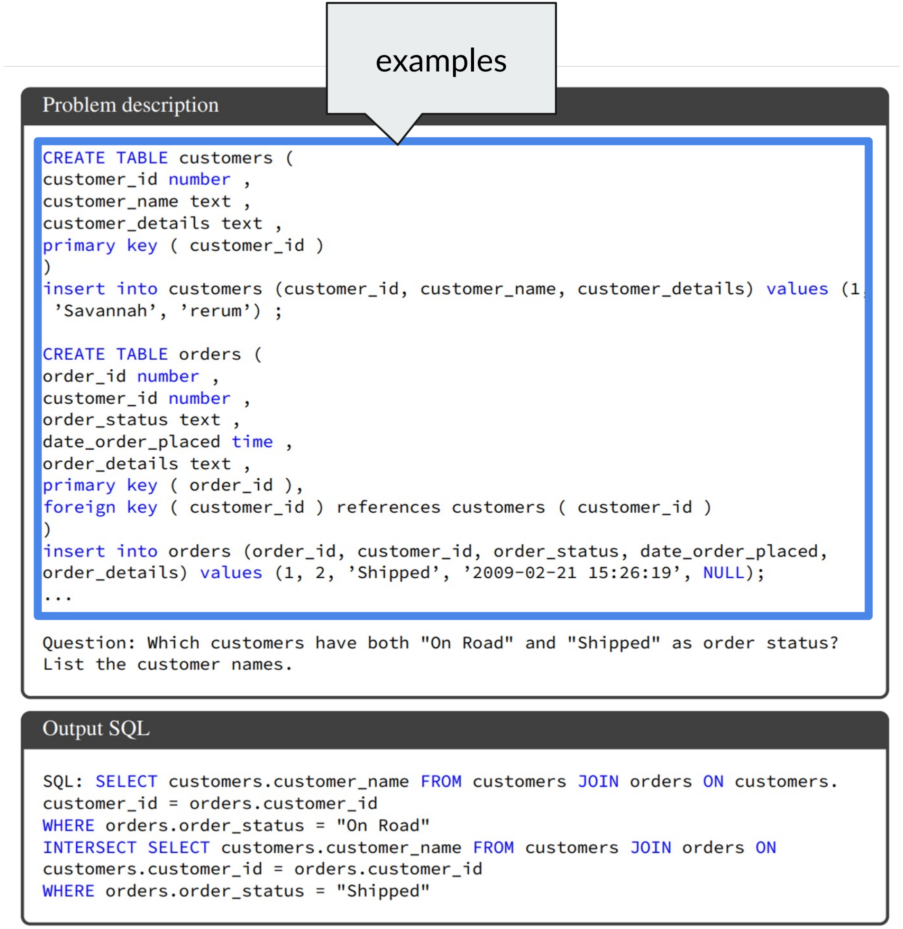
2. Background

Previous Researches:

- Prior works propose deep learning techniques to repair the predicted code, which require **additional training** (Gupta et al., 2020; Wang et al., 2018; Fu et al., 2019; Chen et al., 2023a).
- Recent studies show LLMs can generate **feedback messages** to refine outputs, mainly for **natural language** and **reasoning tasks** (Shinn et al., 2023; Madaan et al., 2023b; Kim et al., 2023; Bai et al., 2022).
- Current LLMs **struggle to self-correct** code without external feedback or human input (Chen et al., 2023a).

Prompting for Code Generation::

- **Few-shot prompting:** Instructs the language model to solve a task with several input-output demonstrations (Brown et al., 2020).
- Besides the demonstrations, we can optionally add an instruction in the prompt to provide a high-level task description (Ouyang et al., 2022; Sanh et al., 2022; Suzgun et al., 2022).



examples

Problem description

```
CREATE TABLE customers (
customer_id number ,
customer_name text ,
customer_details text ,
primary key ( customer_id )
)
insert into customers (customer_id, customer_name, customer_details) values (1,
 'Savannah', 'rerum') ;

CREATE TABLE orders (
order_id number ,
customer_id number ,
order_status text ,
date_order_placed time ,
order_details text ,
primary key ( order_id ),
foreign key ( customer_id ) references customers ( customer_id )
)
insert into orders (order_id, customer_id, order_status, date_order_placed,
order_details) values (1, 2, 'Shipped', '2009-02-21 15:26:19', NULL);
...
```

```
Question: Which customers have both "On Road" and "Shipped" as order status?
List the customer names.
```

Output SQL

```
SQL: SELECT customers.customer_name FROM customers JOIN orders ON customers.
customer_id = orders.customer_id
WHERE orders.order_status = "On Road"
INTERSECT SELECT customers.customer_name FROM customers JOIN orders ON
customers.customer_id = orders.customer_id
WHERE orders.order_status = "Shipped"
```

Execution-Based Code Selection

- Previous Work:
    a. Decoding multiple samples can significantly improve LLM performance (Wang et al., 2023; Shi et al., 2022).
    b. Code execution is used to select the final prediction from multiple generated codes . The most frequent execution result among successful runs is selected as the final code (Chen et al., 2019; Li et al., 2022).
- **When there are multiple predictions,** select the predicted code with the **most frequent execution result** among those that do not encounter execution errors, then apply SELF-DEBUGGING to the code.

3. Contribution

SELF-DEBUGGING Framework

- Introduces a novel framework where LLMs **debug their own code** without external feedback.

Rubber Duck Debugging for LLMs

- Adapts the human strategy of explaining code line-by-line to help LLMs identify and fix errors.

State-of-the-Art Performance

- Achieves **2-3% improvement** on Spider (text-to-SQL)
- **Up to 12% improvement** on TransCoder and MBPP benchmarks.

Better Handling of Complex Tasks

- Improves accuracy on the hardest tasks by **9%**.

Increased Sample Efficiency

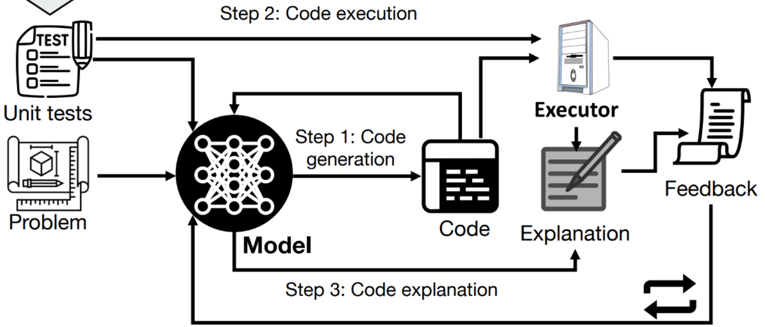
- Reduces the number of samples needed to achieve high accuracy, improving efficiency.

4. Self-Debugging Framework

The process **repeats** until the code is debugged or reaches the maximum iterations:

1. **Generation**
   - The model generates **candidate code** based on the input problem.
2. **Execution:**
   - The generated code is **executed** to observe the results or identify any errors.
3. **Explanation**
   - The model explains the **generated code** in natural language, identifying potential errors.
4. **Feedback**
   - The model uses **the explanation and execution results** to provide feedback on the correctness of

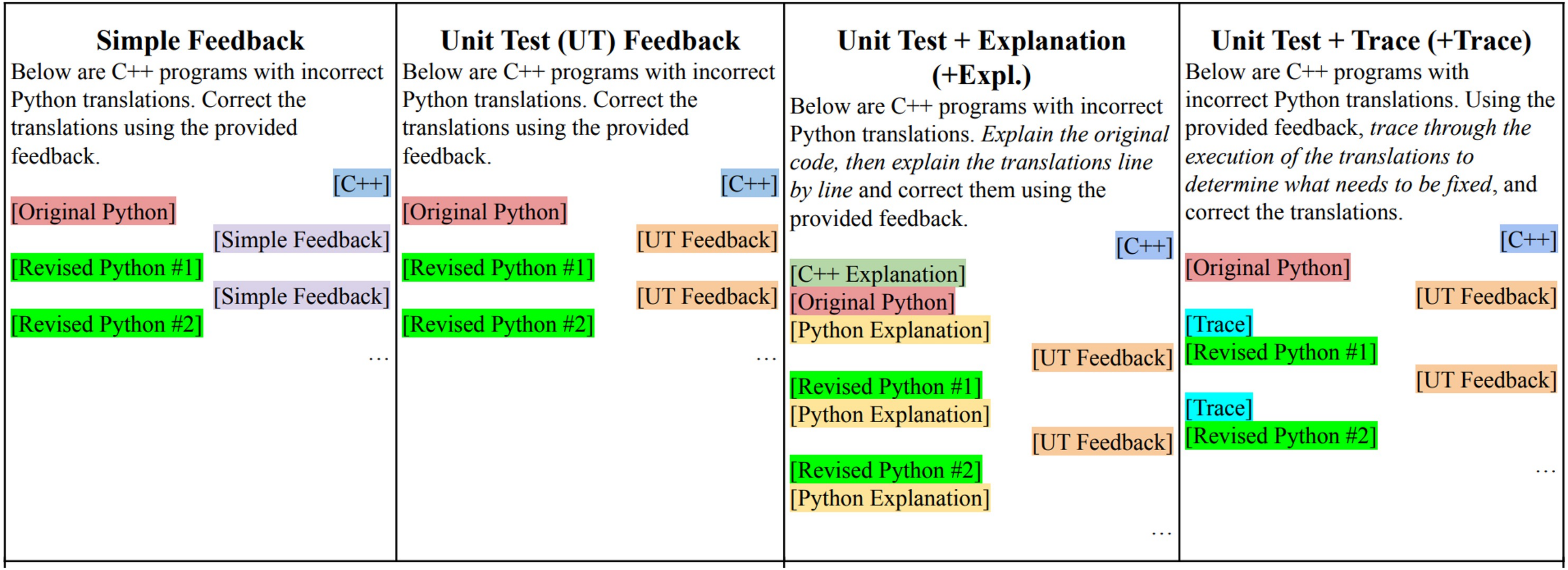
Unit tests provide predefined input-output pairs that the code needs to pass.



- If no unit tests are available, the feedback can rely purely on the code explanation.

4.1 Types of feedback in Self-Debugging

1. **Simple Feedback**
   - A basic message indicating **code correctness** (e.g., "The SQL prediction is correct" or "Please fix the SQL"), which omits the Explanation step.
2. **Unit Test Feedback (UT)**
   - Incorporates **unit test results** into the feedback, providing more detailed information based on code execution.
   - Helps identify **runtime errors** and failed test cases.
3. **Code Explanation Feedback (Expl)**
   - The model explains its **generated code**, similar to **rubber duck debugging**, where it describes and compares code behavior to the problem description. Useful when no unit tests are available.
4. **Execution Trace Feedback (Trace)**
   - Prior work on code repair has demonstrated that training the repair model on execution traces improves the debugging performance (Wang et al., 2018; Gupta et al., 2020).
   - When unit tests are available, the model explains the execution steps **line-by-line** as it runs through the code.



Simple Feedback

Below are C++ programs with incorrect Python translations. Correct the translations using the provided feedback.

[C++]

[Original Python]

[Simple Feedback]

[Revised Python #1]

[Simple Feedback]

[Revised Python #2]

. . .

Unit Test (UT) Feedback

Below are C++ programs with incorrect Python translations. Correct the translations using the provided feedback.

[C++]

[Original Python]

[UT Feedback]

[Revised Python #1]

[UT Feedback]

[Revised Python #2]

. . .

Unit Test + Explanation (+Expl.)

Below are C++ programs with incorrect Python translations. *Explain the original code, then explain the translations line by line* and correct them using the provided feedback.

[C++]

[C++ Explanation]
[Original Python]
[Python Explanation]

[UT Feedback]

[Revised Python #1]
[Python Explanation]

[UT Feedback]

[Revised Python #2]
[Python Explanation]

. . .

Unit Test + Trace (+Trace)

Below are C++ programs with incorrect Python translations. Using the provided feedback, *trace through the execution of the translations to determine what needs to be fixed*, and correct the translations.

[C++]

[Original Python]

[UT Feedback]

[Trace]
[Revised Python #1]

[UT Feedback]

[Trace]
[Revised Python #2]

. . .

5. Experiments

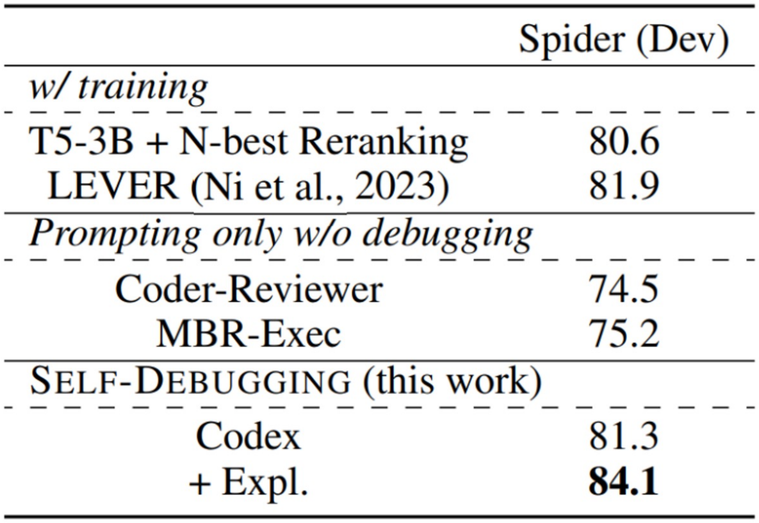
We evaluate **SELF-DEBUGGING** across multiple code generation tasks:

- **Spider Benchmark** (text-to-SQL generation)
- **TransCoder Benchmark** (code translation)
- **MBPP** (text-to-Python generation)
- **Models Used**: Codex, GPT-3.5, GPT-4, StarCoder
- **Decoding Strategy:**
    i. Greedy decoding for initial code generation (temperature τ = 0).
    ii. Sampling multiple programs with temperature τ = 0.7 followed by execution-based selection.
    iii. Maximum debugging turns: 10 (successful debugging mostly ends in 3 turns).

5.1.1 Spider Benchmark (text-to-SQL generation)

- **Task**: Generate SQL queries from natural language.
- **Unit tests: No** unit tests are available.
- **Results:**
    - SELF-DEBUGGING improves accuracy by **2-3%** over baseline.
    - On the hardest queries, accuracy improves by **9%**.
- **Feedback Type**: Code explanation without unit tests.
- **Comparison:**
    - Compared to T5-3B + N-best Reranking, which is trained specifically for text-to-SQL.
    - SELF-DEBUGGING performs without any additional training.

(a) Results on the Spider development set.



	Spider (Dev)
w/ training	
T5-3B + N-best Reranking	80.6
LEVER (Ni et al., 2023)	81.9
Prompting only w/o debugging	
Coder-Reviewer	74.5
MBR-Exec	75.2
SELF-DEBUGGING (this work)	
Codex	81.3
+ Expl.	**84.1**

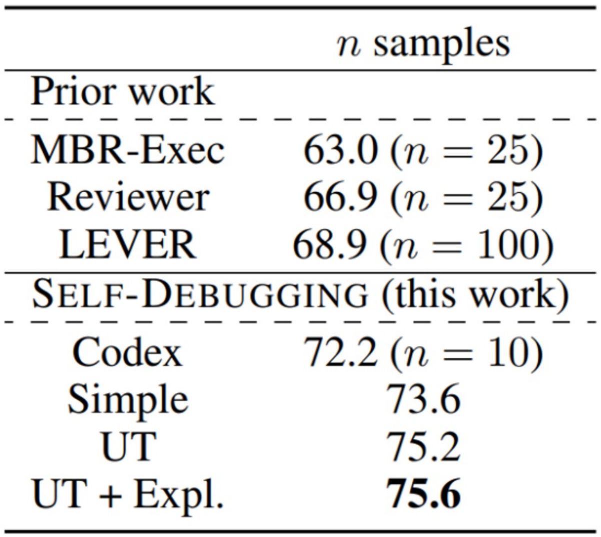
5.1.2 TransCoder Benchmark (Code Translation)

- **Task**: Translate code from one language to another (C++ to Python).
- **Unit Tests**: Available for execution feedback.
- **Results**:
  - SELF-DEBUGGING boosts accuracy by up to **12%**.
  - Performance improves with **unit test feedback** and **code explanations**

5.1.3 MBPP (Text-to-Python Generation)

- **Task**: Generate Python code from text descriptions.
- **Unit Tests**: Only a subset is provided for the problem.
- **Results**:
  - SELF-DEBUGGING improves the baseline accuracy by **8%**.
  - Code explanation and unit test feedback further enhance performance.
- Comparison:
  - MBR-Exec selects programs based on the most common execution output.
  - Coder-Reviewer uses both code likelihood and problem description likelihood for selection.

(b) Results on MBPP dataset.



	n samples
Prior work	
MBR-Exec	63.0 ($n = 25$)
Reviewer	66.9 ($n = 25$)
LEVER	68.9 ($n = 100$)
SELF-DEBUGGING (this work)	
Codex	72.2 ($n = 10$)
Simple	73.6
UT	75.2
UT + Expl.	**75.6**

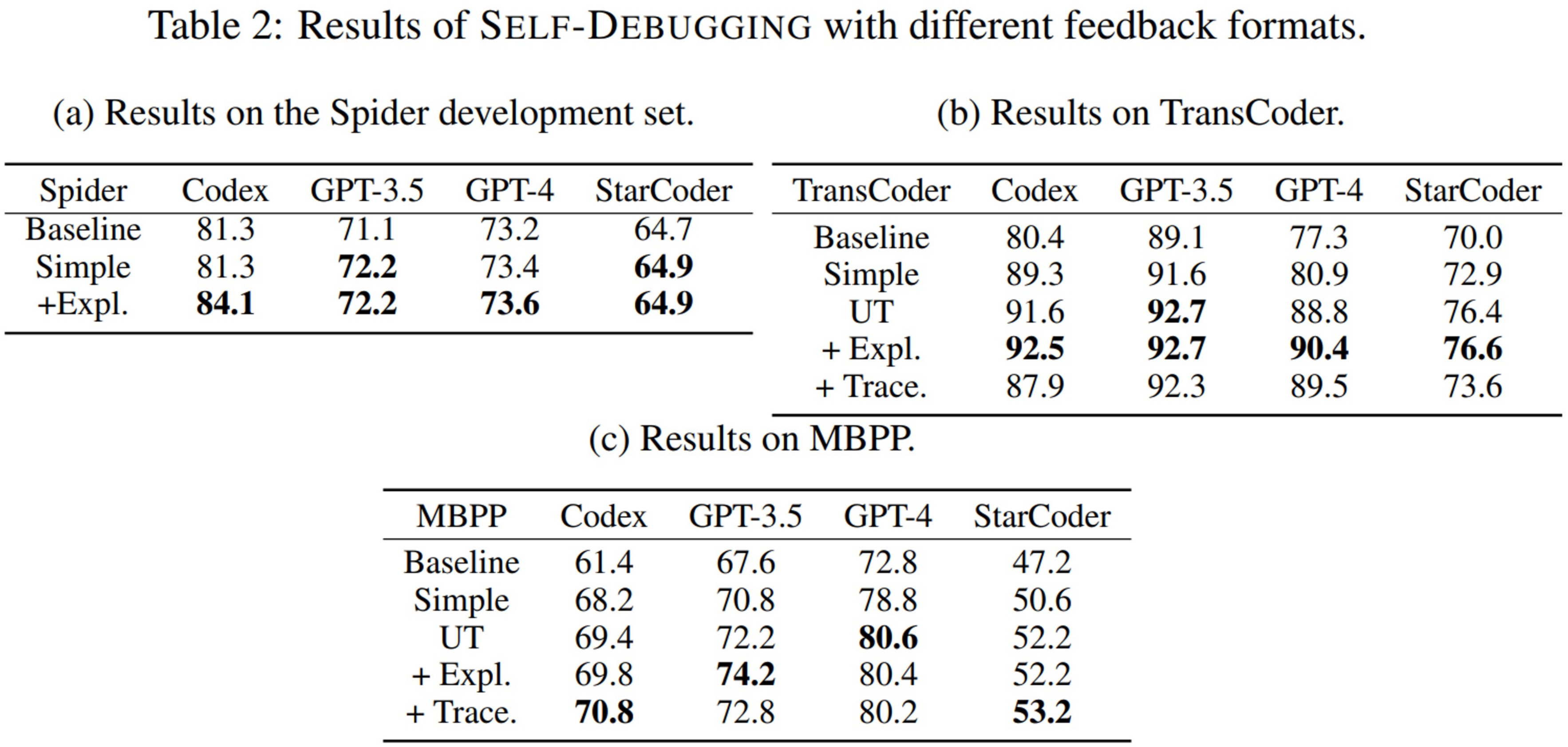


Table 2: Results of SELF-DEBUGGING with different feedback formats.

(a) Results on the Spider development set.

Spider	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	81.3	71.1	73.2	64.7
Simple	81.3	**72.2**	73.4	**64.9**
+Expl.	**84.1**	**72.2**	**73.6**	**64.9**

(b) Results on TransCoder.

TransCoder	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	80.4	89.1	77.3	70.0
Simple	89.3	91.6	80.9	72.9
UT	91.6	**92.7**	88.8	76.4
+ Expl.	**92.5**	**92.7**	**90.4**	**76.6**
+ Trace.	87.9	92.3	89.5	73.6

(c) Results on MBPP.

MBPP	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	61.4	67.6	72.8	47.2
Simple	68.2	70.8	78.8	50.6
UT	69.4	72.2	**80.6**	52.2
+ Expl.	69.8	**74.2**	80.4	52.2
+ Trace.	**70.8**	72.8	80.2	**53.2**

5.2 Ablation studies

To understand the effectiveness of SELF-DEBUGGING from different perspectives.

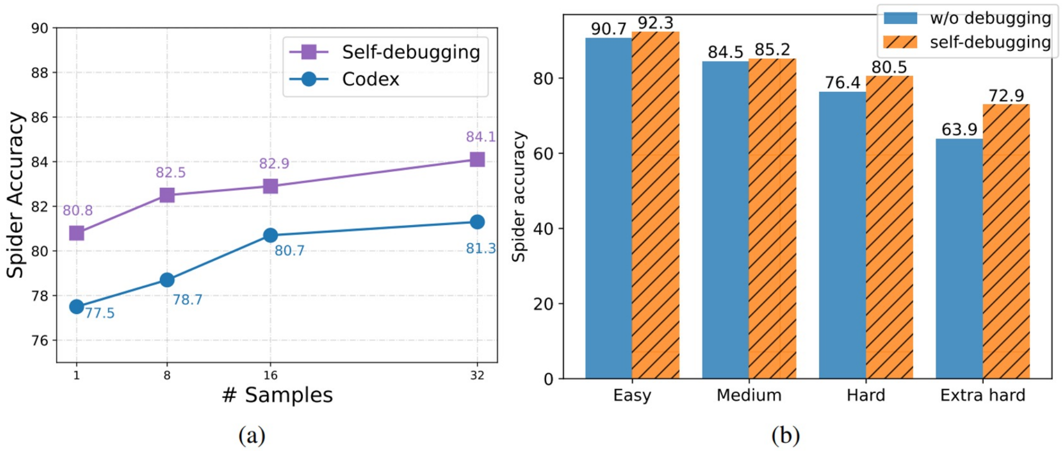


Figure 6: Ablation studies on the Spider development set with Codex. (a) Accuracies with different numbers of initial samples. (b) Breakdown accuracies on problems with different hardness levels.

Figure 6a shows that SELF-DEBUGGING significantly improves sample efficiency. On the Spider benchmark:

- **Greedy decoding with SELF-DEBUGGING** matches the baseline accuracy with 16 samples.
- **8 samples with SELF-DEBUGGING** outperform the baseline with 32 samples.
- Typically, one debugging turn is enough, with further improvements being minimal (~0.1%). This efficiency gain is observed across other benchmarks as well.

5.3 Importance of Code Execution

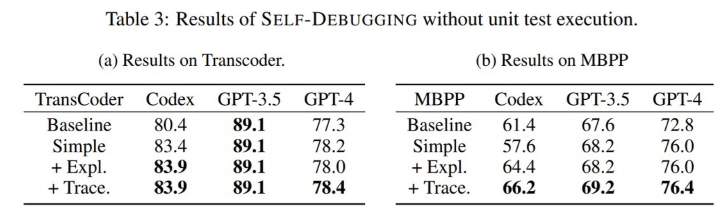


Table 3: Results of SELF-DEBUGGING without unit test execution.

(a) Results on Transcoder.

TransCoder	Codex	GPT-3.5	GPT-4
Baseline	80.4	**89.1**	77.3
Simple	83.4	**89.1**	78.2
+ Expl.	**83.9**	**89.1**	78.0
+ Trace.	**83.9**	**89.1**	**78.4**

(b) Results on MBPP

MBPP	Codex	GPT-3.5	GPT-4
Baseline	61.4	67.6	72.8
Simple	57.6	68.2	76.0
+ Expl.	64.4	68.2	76.0
+ Trace.	**66.2**	**69.2**	**76.4**

Table 3 examines performance without code execution for Transcoder and MBPP, where models rely solely on internal feedback (like in Spider). Key findings:

- **Codex**: SELF-DEBUGGING improves performance by up to **5%**, with execution trace feedback outperforming simple feedback.
- **GPT-4**: Accuracy improves by **3.6%** on MBPP and up to **1%** on other benchmarks without unit test execution.
- **GPT-3.5 vs GPT-4**: Both rely on internal code knowledge without unit tests. GPT-4 performs better but tends to be overconfident in initial predictions.

Conclusion: While unit test execution is important, LLMs can still improve through self-generated feedback.

5.4 Error Types Fixed by SELF-DEBUGGING

- **Syntax Errors**: Incorrect code structure preventing execution.
- **Logical Errors**: Code runs but produces wrong results.
- **Missing Conditions**: Omitted important clauses (e.g., WHERE clauses in SQL).
- **Incorrect Joins**: Mistakes in data relationships (e.g., wrong JOIN clauses in SQL).
- **Function/Variable Misuse**: Incorrect usage of functions or variables.
- **Incomplete Code**: Code lacking necessary elements to execute fully.

6. Conclusion

SELF-DEBUGGING Overview

- **SELF-DEBUGGING** enables LLMs to debug code they generate.
- Empowers models to perform **rubber duck debugging**, identifying and fixing bugs **without human instructions**.
- Achieves **state-of-the-art performance** across several code generation tasks.
- **Improves sample efficiency** significantly.

Performance Highlights

- **Text-to-SQL (No Unit Tests)**:
  - SELF-DEBUGGING improves baseline by **2-3%**.
  - Performance boost of **9%** on the hardest problems.
- **Code Translation & Text-to-Python (With Unit Tests)**:
  - Accuracy increases by up to **12%** with SELF-DEBUGGING.

Key Insights

- **Improved Coding Performance**: Models iteratively debug their own predictions instead of generating correct code from scratch.
- **SELF-DEBUGGING Process**: Understand the code → Identify errors → Follow error messages to fix bugs.

Future Work

- **Better Code Explanation**: Improve models' ability to describe the **high-level semantic meaning** and **implementation details** in their explanations.
- **Informative Error Messages**: Explore techniques to predict more **useful error messages** beyond line-by-line code explanations.
- Preliminary findings show **semantic error feedback** doesn't add much benefit, suggesting more research is needed in this area.

SELFEVOLVE: A Code Evolution Framework via Large Language Models

https://arxiv.org/abs/2306.02907

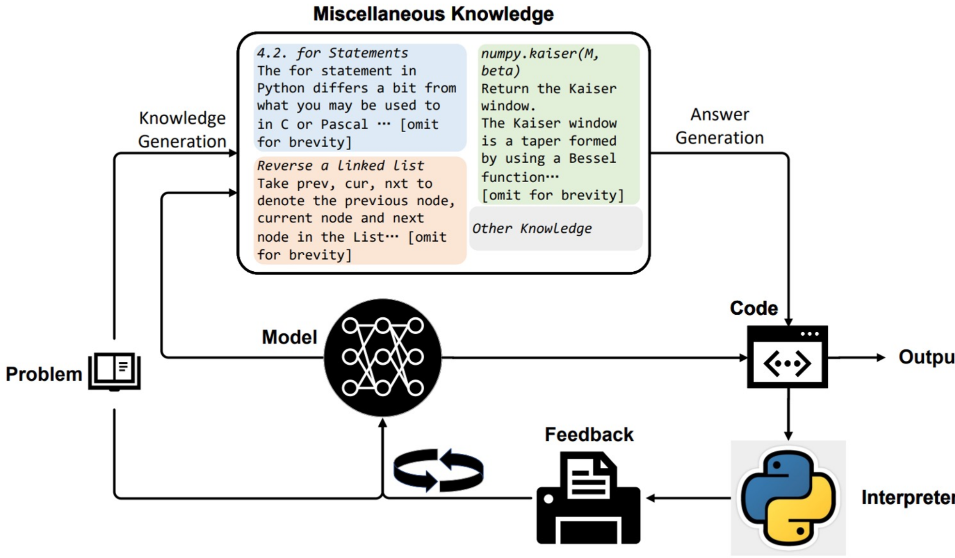
1. Code Llama: Open Foundation Models for Code
2. Planning with Large Language Models
3. Teaching Large Language Models to Self-Debug
4. **SELFEVOLVE: A Code Evolution Framework via Large Language Models**

1. Problem/Background

- Traditional code generation techniques will generate the code then evaluate its performance
  - It will stop when done even if the result is incorrect
- Previous/Proposed methods also retrieve information from external sources to enhance performance
  - Domain mismatch is common (retrieved data doesn’t apply)
  - Bad data is retrieved

2. SELFEVOLVE: A Two-step Pipeline

- Knowledge Generation:
  - LLMs generate knowledge based on problem descriptions or trial solutions rather than external sources. This knowledge could be API documentation, code examples, or problem-specific details.
- Code Refinement:
  - The generated code is executed in a sandbox environment, and any resulting errors are caught. The LLM then refines the code iteratively until it passes all test cases or resolves errors.



4.2. for Statements
The for statement in Python differs a bit from what you may be used to in C or Pascal ⋯ [omit for brevity]

Reverse a linked list
Take prev, cur, nxt to denote the previous node, current node and next node in the List⋯ [omit for brevity]

numpy.kaiser(M, beta)
Return the Kaiser window.
The Kaiser window is a taper formed by using a Bessel function⋯
[omit for brevity]

Other Knowledge

3. Knowledge Generation

- Equation depicts probability of the next token in a sequence conditioned on the previous token in the sequence.
- Product of probabilities of each token given it’s previous tokens and in the input X

$$P(Y) = \prod_{i=1..n} p_\theta(Y_i | Y_{<i}, X), Y_{<1} = \emptyset$$

3.1 Knowledge Generation

$$P(Y|K) = \prod_{i=1..n} p_\theta(Y_i|Y_{<i}, X, K), Y_{<1} = \emptyset$$

$$K := \arg\max_{K \subset B} P(K|X, B)$$

$$p(K) = \prod_{i=1..k} p_\theta(K_i|X, K_{<i}), K_{<1} = \emptyset$$

- We are given m knowledge items
- Knowledge K can be retrieved using a sparse or dense retriever
- Problem: retrievers may be unreliable
    - Use the LLM as the knowledge sources

3.2 Self-Refinement

- Once the results are generated we move on to self-refinement
- Process:
  - Test code against sample test cases in sandbox environment
  - Receive error information then prompt LLM to revise the buggy program
- Below is the joint probability of the refined sequence Y’
- The process is repeated until a working solution is generated or some other criterion is reached

$$P(Y'|X, Y, K, e) = p_\theta(Y'|X, Y, e) \cdot p_\theta(Y|X, K)$$

4. Results

- Three baselines:
  - DocPrompting: Utilizes problem-relevant documentation with a fine-tuned retriever
  - Self-Debugging: Python interpreter to teach LLM to revise python code with bugs
  - SELFEVOLVE: Uses ChatGPT as the knowledge generator and code refiner
- Three data sets:
  - DS-1000: Data set for data science code generation
  - HumanEval: Benchmark for general software engineering code generation
  - TransCoder: C++ to Python code Translation

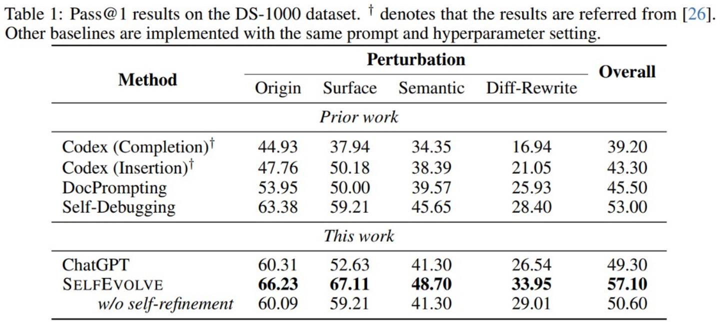


Table 1: Pass@1 results on the DS-1000 dataset. † denotes that the results are referred from [26]. Other baselines are implemented with the same prompt and hyperparameter setting.

Method	**Perturbation**				**Overall**
	Origin	Surface	Semantic	Diff-Rewrite	
		Prior work			
Codex (Completion)†	44.93	37.94	34.35	16.94	39.20
Codex (Insertion)†	47.76	50.18	38.39	21.05	43.30
DocPrompting	53.95	50.00	39.57	25.93	45.50
Self-Debugging	63.38	59.21	45.65	28.40	53.00
		This work			
ChatGPT	60.31	52.63	41.30	26.54	49.30
SELFEVOLVE	**66.23**	**67.11**	**48.70**	**33.95**	**57.10**
w/o self-refinement	60.09	59.21	41.30	29.01	50.60

4.1 Results

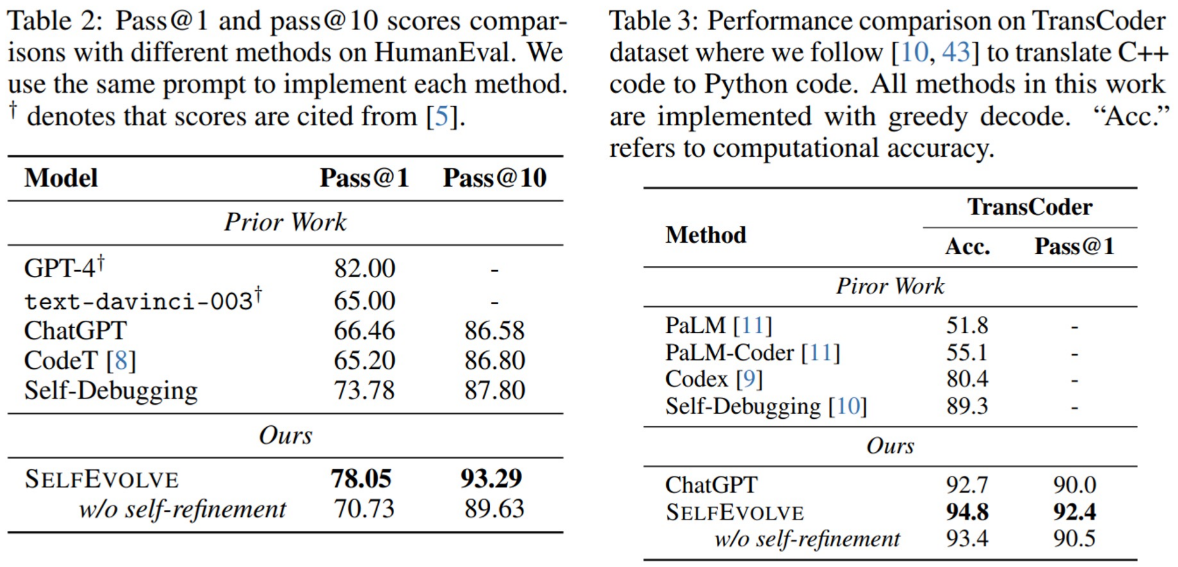


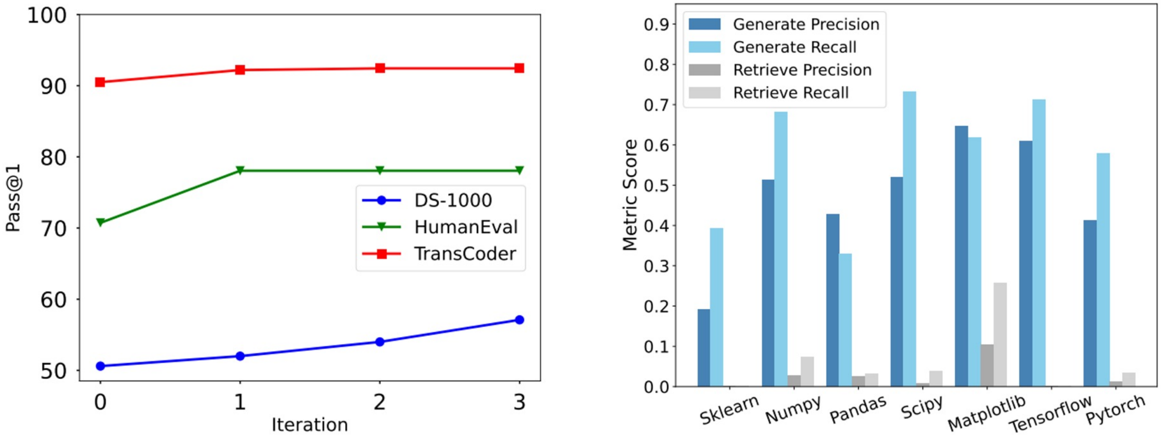
Table 2: Pass@1 and pass@10 scores comparisons with different methods on HumanEval. We use the same prompt to implement each method. † denotes that scores are cited from [5].

Model	Pass@1	Pass@10
	Prior Work	
GPT-4†	82.00	-
`text-davinci-003`†	65.00	-
ChatGPT	66.46	86.58
CodeT [8]	65.20	86.80
Self-Debugging	73.78	87.80
	Ours	
SELFEVOLVE	**78.05**	**93.29**
w/o self-refinement	70.73	89.63

Table 3: Performance comparison on TransCoder dataset where we follow [10, 43] to translate C++ code to Python code. All methods in this work are implemented with greedy decode. "Acc." refers to computational accuracy.

Method	TransCoder	
	Acc.	Pass@1
	Piror Work	
PaLM [11]	51.8	-
PaLM-Coder [11]	55.1	-
Codex [9]	80.4	-
Self-Debugging [10]	89.3	-
	Ours	
ChatGPT	92.7	90.0
SELFEVOLVE	**94.8**	**92.4**
w/o self-refinement	93.4	90.5

4.2 Results



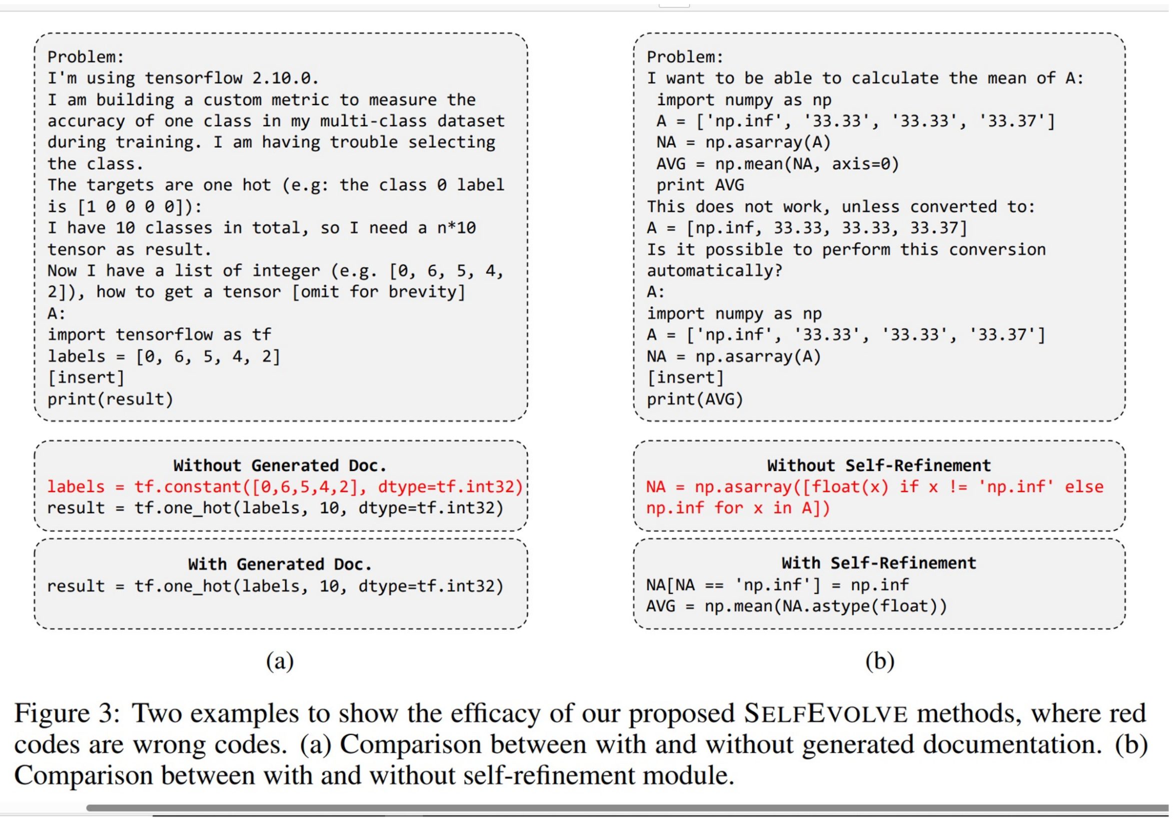
(a)

(b)

Figure 2: (a) Performance-iteration curves of SELFEVOLVE on DS-1000, HumanEval and TransCoder datasets. (b) Precision and recall comparisons between generated knowledge and retrieved one.

4.4 Case Study

- First example is overtuned on the given problem
- Second example forgot to return the average



```
Problem:
I'm using tensorflow 2.10.0.
I am building a custom metric to measure the
accuracy of one class in my multi-class dataset
during training. I am having trouble selecting
the class.
The targets are one hot (e.g: the class 0 label
is [1 0 0 0 0]):
I have 10 classes in total, so I need a n*10
tensor as result.
Now I have a list of integer (e.g. [0, 6, 5, 4,
2]), how to get a tensor [omit for brevity]
A:
import tensorflow as tf
labels = [0, 6, 5, 4, 2]
[insert]
print(result)
```

```
                 Without Generated Doc.
labels = tf.constant([0,6,5,4,2], dtype=tf.int32)
result = tf.one_hot(labels, 10, dtype=tf.int32)
```

```
                 With Generated Doc.
result = tf.one_hot(labels, 10, dtype=tf.int32)
```

(a)

```
Problem:
I want to be able to calculate the mean of A:
 import numpy as np
 A = ['np.inf', '33.33', '33.33', '33.37']
 NA = np.asarray(A)
 AVG = np.mean(NA, axis=0)
 print AVG
This does not work, unless converted to:
A = [np.inf, 33.33, 33.33, 33.37]
Is it possible to perform this conversion
automatically?
A:
import numpy as np
A = ['np.inf', '33.33', '33.33', '33.37']
NA = np.asarray(A)
[insert]
print(AVG)
```

```
                 Without Self-Refinement
NA = np.asarray([float(x) if x != 'np.inf' else
np.inf for x in A])
```

```
                 With Self-Refinement
NA[NA == 'np.inf'] = np.inf
AVG = np.mean(NA.astype(float))
```

(b)

Figure 3: Two examples to show the efficacy of our proposed SELFEVOLVE methods, where red codes are wrong codes. (a) Comparison between with and without generated documentation. (b) Comparison between with and without self-refinement module.

5. Limitations

- Dependence on hand-written prompts
    - Crafting prompts manually is not fully automated. For each new use case, there might be a need to adapt the prompts, limiting the scalability and flexibility of the framework.
- Suitability of generated knowledge
    - While the generated knowledge can be helpful, it may not always be perfectly suited for every task or context. The generated knowledge might contain irrelevant or less accurate information.

6. Key Takeaways/Contributions

- Introduction of a Two-Step Self-Evolving Framework
  - SELFEVOLVE is a novel two-step pipeline where large language models (LLMs) first act as knowledge providers to generate relevant information and then be self-reflective programmers, using error messages to iteratively refine and correct code without external retrieval or predefined test cases.
- Scalability and Generalization
  - The framework is shown to be scalable to more advanced models, such as GPT-4, and is adaptable to a variety of datasets and problem domains without needing domain-specific fine-tuning.

What is more……

Challenges and Future Directions (https://arxiv.org/pdf/2406.00515)

Challenges

- **Complex Code Reasoning**: Difficulty in understanding and generating long, complex code.
- **Multi-Language Limitations**: Struggles to support tasks involving multiple programming languages.
- **Real-Time Collaborative Development**: Challenges in incorporating real-time code changes.
- **Software Engineering Practices**: Misalignment with standard practices in the field.

Future Directions

- **Enhanced Reasoning**: Improving LLMs’ capability to handle complex, multi-step code generation tasks.
- **Efficient Learning**: Incorporating user feedback to refine and optimize code generation.
- **Cross-Domain Integration**: Extending LLMs to handle multi-language programming environments and tasks.
- **Robustness and Transparency**: Ensuring models are reliable and explainable in real-world coding scenarios.

Q&A