

VISUALCODER: Guiding Large Language Models in Code Execution with Fine-grained Multimodal Chain-of-Thought Reasoning

Cuong Chi Le¹, Hoang-Chau Truong-Vinh¹, Huy Nhat Phan¹,
Dung D. Le², Tien N. Nguyen³, Nghi D. Q. Bui^{1*}

¹FPT Software AI Center, Viet Nam
{cuonglc4, chautvh, huyphn16, nghibdq}@fpt.com

²VinUniversity, Vietnam
dung.ld@vinuni.edu.vn

³University of Texas at Dallas, USA
tien.n.nguyen@utdallas.edu

Abstract

Predicting program behavior and reasoning about code execution remain significant challenges in software engineering, particularly for large language models (LLMs) designed for code analysis. While these models excel at understanding static syntax, they often struggle with dynamic reasoning tasks. We introduce VISUALCODER, a simple yet effective approach that enhances code reasoning by *integrating multimodal Chain-of-Thought (CoT) reasoning* with a visual Control Flow Graph (CFG). By aligning code snippets with their corresponding CFGs, VISUALCODER provides deeper insights into execution flows. We address challenges in multimodal CoT integration through a reference mechanism, ensuring consistency between code and its execution path, thereby improving performance in program behavior prediction, error detection, and output generation. Our implementations are available at <https://github.com/FSoft-AI4Code/VisualCoder>.

1 Introduction

Recent advances in Code-related Large Language Models (LLMs) (Hui et al., 2024; Rozière et al., 2024; Wang et al., 2023; Bui et al., 2021; Nijkamp et al., 2023; Lozhkov et al., 2024; Stallone et al., 2024; To et al., 2023; Guo et al., 2024; Wei et al., 2024c; Manh et al., 2024; Huang et al., 2024; Muenighoff et al., 2023) have pushed the boundaries of complex reasoning tasks, extending to the domains that require an understanding of code and its intricacies. There are diverse approaches aimed at enhancing LLMs’ ability. LLMs, while excellent at capturing static patterns and syntax from large code corpora, primarily rely on learned associations rather than direct interaction with the program’s execution environment. They struggle with

tasks involving dynamic behaviors of programs, such as predicting execution traces, variable values, or runtime errors, because these tasks require precise understanding of runtime context and program states change during execution. They do not inherently simulate code execution, which is necessary for understanding how variables’ values vary along program’s execution flow. Moreover, LLMs lack the ability to track mutable state or anticipate runtime conditions, leading to difficulties in predicting dynamic behaviors that depends on context-sensitive execution paths.

Recent work has been proposed to enhance the capability of the models in understanding code execution by incorporating Control Flow Graph (CFG) in their reasoning step (Le et al., 2024; Bieber et al., 2020, 2023). It demonstrates that incorporating CFG of given code can significantly improve performance on the code coverage prediction task. However, it utilizes CFGs through graph neural networks rather than directly integrating them into LLM-based reasoning. Despite these advances, the state-of-the-art approaches focus on a single-modality input (i.e., plain code) and has yet to explore the potential of multimodal frameworks for code execution reasoning. While code can be read in a linear fashion, understanding its behavior requires focusing on the non-linear flow of execution, which is visualized more clearly via a CFG.

In recent years, Vision Language Large Models (VLLMs) (OpenAI et al., 2024; Chen et al., 2024; Liu et al., 2024), have made significant progress, showing their potential across a wide range of tasks that involve both visual and textual inputs. These models, which integrate information from multiple modalities, have been successfully applied to tasks like image captioning, visual question answering, and multimodal retrieval. Recent advances in multimodal LLMs, such as Flamingo (Alayrac et al.,

*Corresponding author.

2024), CLIP (Radford et al., 2021), and BLIP-2 (Li et al., 2023), highlight the benefits of combining visual and textual inputs for enhanced reasoning. Models like LLaVA (Liu et al., 2023) and MiniGPT-4 (Zhu et al., 2024) show improved performance in multimodal tasks by integrating both visual and textual inputs. Studies have shown that combining visual representations with text significantly improves reasoning, especially in tasks involving complex structures (Wei et al., 2024b).

In this work, we propose enhancing the program execution reasoning of LLMs by *leveraging multimodal reasoning*, combining plain code with visual representations of the corresponding CFG. In our experiments, simply presenting the plain code alongside textual or visual representations of the CFG has poor performance for program execution-related tasks (Section 5). Recent work by (Zhang et al., 2023) focuses on improving multimodal reasoning in LLMs using the prominent Chain-of-Thought prompting (Wei et al., 2024a) with two separate steps: rationale generation and reasoning to produce answers. However, when applied to our multimodal setup of plain code and CFG, that approach suffers from cascading errors, where inaccuracies in rationale generation negatively impact the reasoning and final answers.

We introduce VISUALCODER, a simple yet effective **Reference CoT prompting** technique that explicitly links individual lines of code to their corresponding visual elements in the CFG. By making these detailed references, our approach *encourages the model to focus on specific connections between the code and its execution flow during multimodal reasoning process*. This technique is expected to improve the LLM’s performance by guiding it to reason more effectively and grounding its reasoning process with more intuitive and informative representation of code graph via imaging, utilizing both the code structure and its execution dynamics.

2 Related Work

2.1 Code Large Language Models

Large Language Models (LLMs) have been widely applied to various code-related tasks, including code understanding, reasoning, and analysis (Chen et al., 2021a; Li et al.; Jiang et al., 2024; Touvron et al., 2023; Rozière et al., 2024; Xu et al., 2022; Allal et al.; Nijkamp et al., 2022; Phan et al., 2024a; To et al., 2023; Manh et al., 2024; Phan et al., 2024b). Early benchmarks (Yin et al.,

2018; Iyer et al., 2018; Nguyen et al., 2023; Chen et al., 2021a; Austin et al., 2021; Hendrycks et al., 2021) primarily assessed model performance using match-based similarity metrics, which fail to capture deeper reasoning and functional correctness (Chen et al., 2021a). Some benchmarks emphasize domain diversity (Yin et al., 2018; Iyer et al., 2018; Nguyen et al., 2023), while others, such as HumanEval (Chen et al., 2021a), MBPP (Austin et al., 2021), and APPS (Hendrycks et al., 2021), focus on specific tasks like function completion or competitive programming. More recent efforts have sought to expand the scope of evaluation, such as ExeDS (Huang et al., 2022), which targets data science workflows, and ODEX (Wang et al., 2022), an open-domain evaluation suite. However, these benchmarks primarily assess static code properties and standalone function reasoning, with limited emphasis on execution flow analysis and dynamic behavior prediction—critical aspects for improving LLMs’ ability to reason about code execution.

2.2 ML-based Fault Localization

Recent deep learning-based fault localization (FL) techniques like GRACE (Lou et al., 2021), DeepFL (Li et al., 2019), CNNFL (Zhang et al., 2019), and DeepRL4FL (Li et al., 2021) have significantly advanced FL. GRACE uses a graph-based representation to rank faulty methods effectively. Earlier ML-based FL approaches relied heavily on test coverage (Zheng et al., 2016), but struggled to differentiate between failed tests and faulty ones (Li and Zhang, 2017). Advanced methods like TRANSFER (Meng et al., 2022) and FixLocator (Li et al., 2022) address this by leveraging semantic features and co-fixing detection. CodeT5-DLR (Bui et al., 2022) uses LLMs for end-to-end bug detection, localization, and repair.

2.3 Reasoning about Program Execution

Research into program execution reasoning has advanced through various approaches. They use execution states from constructed programs (Chen et al., 2021b; Ni et al., 2024; Shin et al., 2018) or predict intermediate subgoals to improve search strategies in sequence-to-sequence models (Shi et al., 2024). Another approach trains neural networks to simulate execution, acting as learned interpreters (Bieber et al., 2020, 2023; Le et al., 2024), often relying on specialized architectures to model flows and dependencies. Other works like Scratchpad and Self-Debugging explored LLM-generated

reasoning chains, while NExT (Ni et al., 2024) uses runtime traces for task-specific rationales.

3 Motivation

Recent advances in Large Language Models (LLMs) have shown promise in tasks like program execution prediction, especially with Chain-of-Thought (CoT) reasoning (Dhulipala et al., 2024). However, LLMs still struggle with understanding complex execution flows, such as iterations and conditions. Our results in Table 1 (see details later) demonstrate that incorporating **Control Flow Graphs (CFG)** with source code significantly boosts performance. *CFG images provide a visual structure of execution flow*, capturing key control structures like branches and loops. This helps LLMs better understand non-linear execution paths, improving program behavior reasoning.

Choosing the right data representation for CFGs is crucial for helping LLMs understand code execution. To support the use of visual representations, we conducted an experiment comparing the effectiveness of textual vs. visual CFGs. As highlighted in Table 2 on our experimental results, *the models that utilized visual CFG images consistently outperformed those relying on text-based CFG representation*. Our results demonstrate that when models are exposed to CFG images rather than text-based descriptions, their reasoning and prediction accuracy improves substantially.

Since text-based representations only provide a linear and sequential description of control flow in textual format, they often fall short in capturing the structural complexity of code execution which requires forward-backward reasoning continuously. In contrast, *the visual modality provides an additional layer of information, allowing the model to better comprehend non-linear code flows, such as loops and branches, which are harder to grasp through sequential textual descriptions alone*. This result is also consistent with the research by Wei et al. (Wei et al., 2024b), which emphasizes that incorporating visual representations significantly enhances the reasoning capabilities of multimodal LLMs. Importantly, this result motivates us on the adopting of visual representations that requires deep, non-linear flow of execution reasoning.

Despite the advantages of CFG images, we found that *incorporating CoT reasoning into multimodal models is not trivial* and introduces new challenges. Surprisingly, our results in Table 3 show

that adding CoT reasoning alongside CFG images often leads to performance degradation. As seen in Table 3, when CoT reasoning was applied to tasks like **bug detection**, performance dropped for models such as **Sonnet 3.5** and **InternVL2-26B**. The models suffer *hallucinations*, leading to incorrect reasoning steps. Existing methods, such as the two-stage multimodal Chain-of-Thought (multimodal-CoT) by (Zhang et al., 2023), attempt to separate rationale generation from answer inference but fail to reason execution on complex code structures.

Let us use an example for illustration. As shown in Figure 1, the **CFG + CoT** approach fails to capture the critical point in reasoning. As with this approach (see red section), the model incorrectly identifies the termination point within the *else* block ($G += 1$), missing the fact that this branch is unreachable. Since X is always even, the *else* block will never be executed.

We hypothesize that the key issue is the model’s inability to align the code with its corresponding CFG image during reasoning. Without proper alignment with the CFG, the model misinterprets this unreachable path as a valid termination point, focusing on an irrelevant error. Therefore, we guide the model to refer to each line of code to the corresponding element in the CFG as shown in Figure 1 (highlighted in green). Let us call it **CFG + CoT + Reference** approach, which correctly identifies the unreachable node and termination point. Our results (Section 5) also show that the two-stage multimodal-CoT approach in (Zhang et al., 2023) is also insufficient for complex coding tasks that involve intricate execution flows.

As illustrated in Figure 1, the **CFG + CoT + Reference** approach (green section) allows the LLM to correctly identify the critical point: the unreachable nature of the *else* branch. By *explicitly referencing the CFG during reasoning*, the model avoids errors in unreachable branches and focuses on the actual critical error—the float N being used in the `range()` function. *This reference mechanism helps the model maintain proper alignment between the visual CFG and the code, leading to more accurate reasoning on program execution at runtime*.

Next, we will provide a detailed explanation of our proposed method, demonstrating how the combination of **Control Flow Graphs**, **Chain-of-Thought reasoning**, and a **Reference Mechanism** (**CFG + CoT + Reference**) addresses these challenges and significantly improves code execution reasoning. We formulate our solution in Section 4.



Prompt:

You are an expert Python programmer.
Analyze the following Python code snippet, which contains error(s) when executing:

{code}

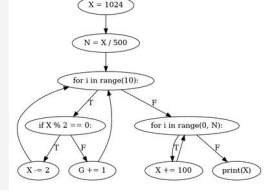
You will also be provided with a control flow graph (CFG) image of this code. As you analyze each line:
1. Examine each line of code sequentially using both code and CFG.
2. *Reference the CFG to identify which node corresponds to the line you're currently analyzing.*
3. Use this understanding between code and CFG to support your reasoning about the code's logic and errors.

Think through your analysis step by step, considering both the code and its representation in the CFG.
After your analysis, respond with the problematic line of code that causes termination.

Answer: The answer is: for i in range(0, N):

```
X=1024
N=X/500
for i in range(10):
    if X%2==0:
        N+=1
    else:
        G+=1
for i in range(0, N):
    X+=100
print(X)
```

Code



CFG



CFG+CoT

Let's analyze this code step by step, using both the provided code snippet and the control flow graph (CFG):

1. X = 1024:
This initializes X correctly.
2. N = X / 500:
This calculates N as a float (2.048).
3. The outer for loop runs 10 times.
4. The if condition checks if X is even, which it is initially.
5. If X is even, N is incremented. If odd, G is incremented.
6. There's an inner for loop that should run N times, incrementing X by 100 each time.
7. Finally, X is printed.

Now, let's consider potential issues:

1. N is a float, which might cause issues in the range() function of the second loop.
2. G is used without being initialized.
3. The first loop modifies N, which affects the second loop's range.

The most critical error that would cause immediate termination is the use of G without initialization. This occurs in the else clause of the first loop.

Therefore, the answer is: G+=1



CFG+CoT w Reference

Let's analyze the code step by step, referencing the CFG:

1. X = 1024 (Node 1 in CFG):
This is correct and initializes X.
2. N = X / 500 (Node 2 in CFG):
This calculates N as a float (2.048). No error here, but it might cause issues later.
3. for i in range(10): (Node 3 in CFG):
This starts the outer loop. No error.
4. if X % 2 == 0: (Node 4 in CFG):
This checks if X is even. No error.
5. N += 1 (Node 5 in CFG, True branch):
This increments N. No error, but N becomes a float.
6. else: G += 1 (Node 6 in CFG, False branch):
This is where we encounter our first error. G is not defined before this line.
7. for i in range(0, N): (Node 7 in CFG):
This tries to use N (a float) as the end of a range, which is not allowed in Python.
8. X += 100 (Node 8 in CFG, True branch):
This would increment X if the loop ran. No error in this line itself.
9. print(X) (Node 9 in CFG):
This would print X if reached. No error in this line.

The first error that would cause termination is on line 6, where G is used before being defined. *However, this line wouldn't be reached in the current code because X is always even.* The second error, which would be encountered first in this specific code, is on line 7, where a float is used in range().

Therefore, the answer is: for i in range(0, N):

Figure 1: Comparison of Program Execution Reasoning: CFG + CoT w/o Reference vs. CFG + CoT with Reference. With reference, LLM correctly identifies the unreachable node and critical termination point (highlighted in orange).

4 VISUALCODER: Reference Mechanism

We propose a method that combines **Control Flow Graphs (CFG)** with **Chain-of-Thought (CoT)** reasoning, augmented by a **Reference Mechanism**, to enhance reasoning on program execution. This approach enables step-by-step evaluation of the code while also cross-referencing control flow points, thereby improving error detection and identifying unreachable or erroneous code paths.

Let the given Python code snippet be represented as a sequence of lines of code:

$$\text{Code} = \{C_1, C_2, \dots, C_n\} \quad (1)$$

where C_i represents the i -th line or block of code. Along the code, we provide the corresponding *Control Flow Graph (CFG)*, which is defined as:

$$\text{CFG} = (N, E) \quad (2)$$

where $N = \{N_1, N_2, \dots, N_m\}$ is the set of nodes, each corresponding to a specific code block, and

$E \subseteq N \times N$ is the set of directed edges representing control flow between nodes.

The goal is to condition the Vision Large Language Model that semantically maps each line C_i of the code to its node N_i in the CFG, and utilize this to perform stepwise reasoning.

4.1 Chain-of-Thought Reasoning (CoT)

Chain-of-Thought reasoning is implemented by analyzing each instruction on C_i while considering its logical dependencies and its corresponding control flow in the CFG. We define the reasoning process as a recursive function:

$$R(C_i) = f(C_i, \{C_1, C_2, \dots, C_{i-1}\}, N_j) \quad (3)$$

where f is a function that takes the current line of code, its execution context, and its corresponding CFG node N_j .

4.2 Reference Mechanism

The **Reference Mechanism** enhances CoT reasoning by mapping each line of code C_i to its corresponding CFG node, expressed as $M : C_i \mapsto N_j$,

where C_i is represented by node N_j in the CFG. To establish this mapping, we guide the model to focus on the relevant CFG node while reasoning about each line of code. This is achieved by reinforcing attention on the corresponding node in the CFG image whenever the model processes its associated code line, as demonstrated in Section 7. By aligning each line of code with its node in the control flow representation, this mechanism improves the model’s understanding of execution paths, transitions, and dependencies across statements and blocks, rather than treating lines in isolation.

4.3 CFG + CoT (Without Reference)

In the **CFG + CoT** approach, the model reasons about the logic purely based on the sequential structure of the plain code. It analyzes each line and attempts to infer potential errors based solely on the textual content, without actively cross-referencing the CFG. This reasoning process can be defined as:

$$\begin{aligned} p_{\text{no-ref}}(Y|C_1, \dots, C_n, \text{CFG}) \\ &= \prod_{i=1}^n \mathcal{P}(Y_i|C_1, \dots, C_i, \text{CFG}) \\ &= \prod_{i=1}^n \mathcal{P}(Y_i|C_1, \dots, C_i, (N_1, \dots, N_m), E) \end{aligned} \quad (4)$$

Here, the probability of generating the correct reasoning Y for the code is determined by the cumulative probabilities of the reasoning steps at each line of code. However, this method is prone to inefficiency, as it includes all CFG nodes (N_1, N_2, \dots, N_m) in each reasoning step, even when many of those nodes are not directly relevant to the current line of code.

4.4 CFG + CoT + Reference

In contrast, the **CFG + CoT + Reference** approach introduces a structured reference to the CFG during each reasoning step. The reasoning at each line C_i is conditioned not only on the previous code lines but also on the corresponding node in the CFG:

$$\begin{aligned} p_{\text{ref}}(Y|C_1, \dots, C_n, \text{CFG}) \\ &= \prod_{i=1}^n \mathcal{P}(Y_i|(C_1, M(C_1)), \dots, (C_i, M(C_i)), E) \end{aligned} \quad (5)$$

Where $M(C_i)$ is the mapped node in the CFG corresponding to the current line C_i . By analyzing and referencing the corresponding CFG block for every

line of code, the model can maintain consistency between the CFG and the source code.

4.5 VISUALCODER

There are several ways to achieve the behavior outlined in the CFG + CoT + Reference process, such as fine-tuning, one-shot or few-shot prompting, and more. In our current implementation, we propose a straightforward yet effective approach that can be integrated into any Chain-of-Thought framework without the need for fine-tuning. By introducing a simple instruction, as shown in Figure 1 (green line in the prompt), we expect to guide Vision Language Models to follow the formulation described in Equation 5. This approach ensures that the model focuses its reasoning on the relevant CFG node for each line of code, thereby improving its alignment with the control flow. The experimental results in Section 5, along with the qualitative analysis in Section 6 and attention heat map in Section 7, demonstrate the effectiveness of our method in enhancing program execution reasoning.

5 Empirical Evaluation

5.1 Better Code Execution Understanding with Control Flow Graph Images

In this experiment, we aimed to show that providing the LLM with CFG images (*no references*) improves its code execution reasoning. Using the CRUXEval benchmark (Gu et al., 2024), we tested models on predicting execution outputs. We compared the accuracies of three state-of-the-art VLM models—Claude 3.5 Sonnet (Anthropic, 2024), Gemini-1.5-Flash (Reid et al., 2024), and InterVL2-8B (Chen et al., 2024)—in two settings: 1) plain code, and 2) plain code with its CFG image. The task involved both **output prediction** (predicting the execution’s result) and **input prediction** (predicting the inputs leading to a specific output).

For direct comparison with prior work, we used the same prompt format from the original CRUXEval paper (Gu et al., 2024). The prompt provided the code and, when applicable, a visual CFG, guiding step-by-step reasoning. Performance was measured using the pass@1 metric, indicating if the models’ first predictions were correct.

The results in Table 1 show that incorporating a CFG image improves model accuracy in two settings. This improvement is consistent across models, showing that CFG enhances the LLMs’ ability to reason about execution flow and predict program

Task	Settings	Models	pass@1
Output Pred.	Plain code	Claude 3.5 Sonnet	79.6
	Plain code + CFG image	Claude 3.5 Sonnet	82.3
	Plain code	Gemini 1.5 Flash	68.5
	Plain code + CFG image	Gemini 1.5 Flash	70.0
	Plain code	InterVL2-8B	40.8
	Plain code + CFG image	InterVL2-8B	44.0
Input Pred.	Plain code	Claude 3.5 Sonnet	75.2
	Plain code + CFG image	Claude 3.5 Sonnet	84.0
	Plain code	Gemini 1.5 Flash	58.4
	Plain code + CFG image	Gemini 1.5 Flash	68.4
	Plain code	InterVL2-8B	43.6
	Plain code + CFG image	InterVL2-8B	44.4

Table 1: Execution Prediction Performance Comparison

behaviors more accurately. This result is consistent with the one reported by Le *et al.* (Le *et al.*, 2024) in which incorporating CFG of given code can improve performance on code coverage prediction.

5.2 CFG Images vs Text-Based Descriptions

Model	CFG (Text)	CFG (Image)
Claude 3.5 Sonnet	60.5	74.0
Gemini 1.5 Flash	65.3	74.1
InternVL2-8B	23.2	36.4

Table 2: Comparison of pass@1 results for CFG in text-based description vs. CFG as Image.

To assess the impact of visual representations in coding tasks, specifically in *Code Execution Prediction*, we conducted an experiment where LLM models were provided with either Mermaid-format (text-based) (see one example of Mermaid-format in A.1) or image-based CFGs, along with the input, and tasked with predicting the code’s output. The prompt remained the same as the previous experiment, but the models received CFGs as images instead of as texts. Results in Table 2 show that CFG images significantly boost performance in reasoning tasks, underscoring the value of visual aids in enhancing Multimodal LLMs’ reasoning.

5.3 VISUALCODER Multi-modal Reasoning

Experimental Setting. This experiment involved two tasks: **Program Repair** and **Fault Localization** (further details are provided in Section A.3). For **Program Repair**, we generated a dataset from LiveCodeBench (Jain *et al.*, 2024), selecting 400 instances to avoid the saturation seen in simpler benchmarks like MBPP-S (Austin *et al.*, 2021). We sampled six solutions for each instance using Claude 3.5 Sonnet and Haiku with a 3:1 ratio, ensuring varied difficulty levels. After filtering out fully correct solutions (*those passing all test cases*)

and completely incorrect ones (*those failing all test cases*), we retained solutions that have the correct direction—passing a subset of test cases but containing some errors. We finalized 384 solutions for 173 problems. For **Fault Localization**, we used the FixEval dataset (Anjum Haque *et al.*, 2023), which has about 210 programs with various runtime errors. The prompt used was listed in Section A.4.

Unlike the previous sections, we selected models with stronger code reasoning abilities to handle the increased complexity of Program Repair and Fault Localization task. Specifically, Claude 3.5 Sonnet was retained for its robust performance, GPT-4o replaced Gemini 1.5 Flash due to its superior capabilities and wider adoption, and InterVL2-26B replaced its 8B version, which struggled with coding tasks, often producing generic or incorrect answers.

We evaluated the models in several configurations: plain code (with/without CoT reasoning), plain code with CFGs (with/without CoT), plain code with execution in-line comments (NeXT (Ni *et al.*, 2024)), the two-stage **Multimodal-CoT** method from (Zhang *et al.*, 2023), and VISUALCODER. To assess our method’s adaptability and efficiency, we integrated it with **Multimodal-CoT** in the first stage of Rationale Generation. The second stage, Answer Inference, remained unchanged. NeXT is excluded from the Fault Localization task, as it relies on code execution, unsuitable for tasks requiring bug detection without execution.

Experimental Results. Table 3 provides a detailed comparison of VISUALCODER with other baseline methods across multiple settings. Chain-of-Thought (CoT) reasoning generally improved model performance, as seen in the Program Repair task where GPT-4o improved from 38.7% (plain code) to 40.1% (with CoT), and InternVL2 increased from 0.4% to 4.0%. However, combining CoT with CFG images caused a notable performance drop across all models. For instance, Claude 3.5 Sonnet’s accuracy dropped from 63.0% to 55.5%, GPT-4o fell from 40.1% to 37.6%, and InternVL2-26B dropped from 4.0% to 2.1%. Similar declines occurred in the Fault Localization task. This suggests that while CFGs offer structural insights, *integrating them with CoT without proper schemes can confuse the models and reduce accuracy*, a finding consistent with (Zhang *et al.*, 2023).

In the Program Repair task, which relies more on logical reasoning than execution-heavy tasks, CFGs proved less useful. Although our method didn’t outperform the highest-performing settings

Tasks	Settings	Claude 3.5 Sonnet	GPT-4o	InternVL2 26B
Program Repair	Plain code w/o CoT	64.1	38.7	0.4
	Plain code w/ CoT	63.0	40.1	4.0
	Plain code + CFG w/o CoT	61.2	36.5	0.9
	Plain code + CFG w/ CoT	55.5	37.6	2.1
	NeXT	57.3	40.7	0.0
	Multimodal-CoT	58.7	35.1	8.2
	VISUALCODER	62.9	41.2	6.3
Fault Localization	Multimodal-CoT + VISUALCODER	60.1	38.2	10.7
	Plain code w/o CoT	90.4	87.1	37.0
	Plain code w/ CoT	90.0	89.5	26.1
	Plain code + CFG w/o CoT	86.1	79.4	22.3
	Plain code + CFG w/ CoT	88.0	85.6	41.0
	Multimodal-CoT	90.9	87.6	52.1
	VISUALCODER	91.4	90.4	47.4
	Multimodal-CoT + VISUALCODER	92.8	91.9	53.6

Table 3: Performance Comparison on Program Repair and Fault Localization Tasks

(e.g., plain code without CoT for Claude at 64.1%), it significantly boosted performance over the plain code + CFG w/ CoT setting. It raised Claude 3.5 Sonnet from 55.5% to 62.9%, and GPT-4o from 37.6% to 41.2%. InternVL2-26B, which struggled with CFG + CoT (2.1%), improved to 6.3% with VISUALCODER and 10.7% when combined with Multimodal-CoT. In some cases, it outperformed methods like NeXT and Multimodal-CoT, with Claude 3.5 Sonnet achieving 62.9% with VISUALCODER, compared to 57.3% with NeXT and 58.7% with Multimodal-CoT, showing its capability, even in tasks where CFGs are less central.

In the Fault Localization task, improvements were consistent across settings. In the plain code w/o CoT setting, Claude reached 90.4%, GPT-4o 87.1%, and InternVL2 37.0%. Introducing CoT improved GPT-4o to 89.5%, while Claude remained at 90.0%. Adding CFGs led to varied results: Claude dropped to 86.1%, GPT-4o to 79.4%, and InternVL2 to 22.3%. These mixed outcomes suggest that while providing structural insights, CFGs complicate reasoning without proper integration.

As shown in Table 3, VISUALCODER achieved the highest accuracy for both Claude 3.5 Sonnet (91.4%) and GPT-4o (90.4%). When combined with Multimodal-CoT, performance further improved, with Claude reaching 92.8% and GPT-4o 91.9%. The biggest gain was for InternVL2-26B, which increased from 41.0% (CoT with CFG) to 53.6% with VISUALCODER and Multimodal-CoT. These results show that *integrating CFGs with CoT reasoning and the Reference Mechanism boosts fault localization, especially when paired with Multimodal-CoT. This also highlights the effectiveness of generating rationale that efficiently leverages both plain code and CFG images.*

VISUALCODER with Complex Java Reposito-

ries. To further evaluate VISUALCODER’s effectiveness on more complex, real-world software defects, we extended our experiments to the Defects4J v1.0 (Just et al., 2014) benchmark. Defects4J comprises 245 real-world Java bugs from five open-source projects: Chart, Closure, Lang, Math, and Time. For this setting, we used GPT-4o and evaluated its fault localization performance using the acc@k metric, which quantifies the number of bugs where the actual buggy location appears among the top k predictions generated by a tool. Table 4 presents the results, comparing Plain Code (Vanilla), Plain Code + CFG, and Plain Code + CFG + Reference Mechanism as VISUALCODER.

Model	acc@1	acc@4	acc@10
Plain code	47	73	90
Plain code + CFG	54	78	95
VISUALCODER	59	80	97

Table 4: Fault localization accuracy on Defects4J v1.0.

As shown in Table 4, VISUALCODER consistently outperforms the baselines across all metrics, demonstrating its effectiveness in complex fault localization tasks. The integration of CFGs and the Reference Mechanism significantly enhances localization accuracy, particularly at acc@1 , where VISUALCODER correctly identifies 12 more bugs than the baseline. These results highlight VISUALCODER’s robustness in large-scale Java projects, reinforcing its potential for software debugging.

6 Qualitative Analysis

Figure 2 presents two examples of buggy code alongside their corresponding CFGs and the reasoning outputs of Claude Sonnet 3.5 under different prompt settings: *plain code with CoT*, *plain code + CFG image with CoT*, and *2-stage prompt of Multimodal-CoT* in (Zhang et al., 2023).

The first three rows of Figure 2 show Claude Sonnet 3.5’s outputs under different prompt settings, all failing to fully grasp the code’s complexity. In the left example (a use-before-initialization error), the model incorrectly identifies `lst[0]` as the issue, missing the control flow dependencies affecting `lst`’s initialization. In the right example (unreachable code), it highlights `G += 1` but overlooks the actual problem: using a float `N` in the range function. These errors highlight the limitations of plain code reasoning, even with CFG or CoT.

The final row shows our approach’s result. In the left side, VISUALCODER correctly identifies the

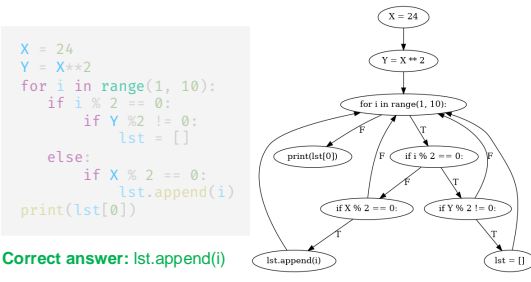
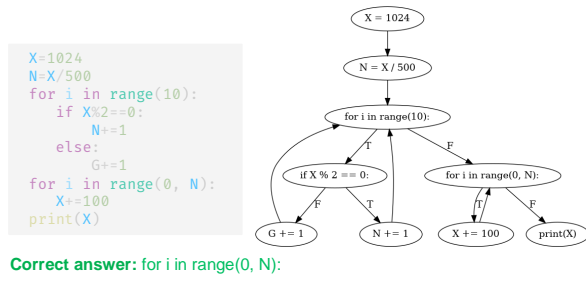
Buggy code & CFG image	 <p>Correct answer: <code>lst.append(i)</code></p>	 <p>Correct answer: <code>for i in range(0, N):</code></p>
Plain code w CoT	<p>Let's analyze this code step by step:</p> <p>(1) <code>X = 24</code>: This sets the variable <code>X</code> to 24...(10) <code>print(lst[0])</code> This tries to print the first element of <code>lst</code>.</p> <p>The main issue here is that <code>lst</code> is only defined within the <code>if</code> statement on line 6, which is never executed because <code>Y</code> is always even. The code will raise an error when it tries to access <code>lst[0]</code> because <code>lst</code> is not defined in the global scope.</p> <p>Answer: <code>print(lst[0])</code></p>	<p>Let's analyze this code step by step:</p> <p>(1) <code>X=1024</code>: This line initializes <code>X</code> with the value 1024...(10) <code>print(X)</code>: This would print the final value of <code>X</code>, if the code could reach this point.</p> <p>The first error that would cause the code to terminate is on line 7, where <code>G</code> is used without being defined.</p> <p>Answer: <code>G += 1</code></p>
Plain code + CFG w CoT	<p>Let's analyze this code and CFG step by step:</p> <p>(1) <code>X = 24</code>: This initializes <code>X</code> to 24. (Correct)...(7) After the loop, it attempts to print <code>lst[0]</code>.</p> <p>The main issue here is that <code>lst</code> is not defined in the global scope. It's only defined inside the loop under certain conditions. Even if <code>lst</code> is defined and populated, it will be reset to an empty list every time <code>i</code> is even and <code>Y</code> is odd. The error occurs when trying to access <code>lst[0]</code> after the loop, as <code>lst</code> may not be defined or may be empty.</p> <p>Answer: <code>print(lst[0])</code></p>	<p>Let's analyze this code step by step, using the code snippet and CFG:</p> <p>(1) <code>X = 1024</code>: This initializes <code>X</code> correctly...(7) Finally, <code>X</code> is printed.</p> <p>Now, let's consider potential issues: <code>N</code> is a float, which might cause issues in the <code>range()</code> function of the second loop. <code>G</code> is used without being initialized. The first loop modifies <code>N</code>, which affects the second loop's range. The most critical error that would cause immediate termination is the use of <code>G</code> without initialization. This occurs in the <code>else</code> clause of the first loop.</p> <p>Answer: <code>G += 1</code></p>
Multimodal-CoT	<p>Let's analyze this Python code snippet line by line and CFG:</p> <p>(1) <code>`X = 24`</code>: This line initializes <code>X</code> with 24...(10) <code>`print(lst[0])`</code>: This tries to print the first element of <code>lst</code> after the loop ends.</p> <p>The main error occurs because the code attempts to print <code>`lst[0]`</code> after the loop, but <code>`lst`</code> might not be defined if the conditions for its initialization are never met.</p> <p>Answer: <code>print(lst[0])</code></p>	<p>Let's analyze the code line by line with CFG provided:</p> <p>(1). <code>`X = 1024`</code>: This corresponds to the top node in the CFG. It's correct and initializes <code>X</code>...(9) <code>`print(X)`</code>: This is the final node in the CFG and will execute if no errors occur before it.</p> <p>The main error is the use of the undefined variable <code>G</code> in the <code>else</code> clause of the first loop. This will raise a <code>NameError</code> when <code>X</code> is odd.</p> <p>Answer: <code>G += 1</code></p>
VisualCoder	<p>Let's analyze this code step by step, referencing the CFG:</p> <p>(1) <code>X = 24</code> (First node in CFG): This is fine, <code>X</code> is assigned the value 24...(10) <code>print(lst[0])</code> (Last node, outside the loop): This tries to print the first element of <code>lst</code>.</p> <p>The main issue is that <code>lst</code> is only initialized when <code>Y</code> is odd, which never happens. When <code>i</code> is odd and <code>X</code> is even (always true), the code tries to append to <code>lst</code>, but <code>lst</code> may not exist.</p> <p>Answer: <code>lst.append(i)</code></p>	<p>Let's analyze the code step by step, referencing the CFG:</p> <p>(1) <code>X = 1024</code> (Node 1 in CFG): This is correct and initializes <code>X</code>...(9) <code>print(X)</code> (Node 9 in CFG): This would print <code>X</code> if reached. No error in this line.</p> <p>The first error that would cause termination is on line 6, where <code>G</code> is used before being defined. However, this line wouldn't be reached in the current code because <code>X</code> is always even. The second error, which would be encountered first in this specific code, is on line 7, where a float is used in <code>range()</code>.</p> <p>Answer: <code>for i in range(0, N):</code></p>

Figure 2: Qualitative comparison of reasoning outputs for buggy code using different prompt settings in Claude Sonet 3.5. **Red** text indicates where the reasoning fails, **green** text highlights correctly identified critical points, and **blue** text in VISUALCODER shows the referencing from the plain code to the corresponding nodes in the CFG.

error by analyzing the CFG and noting the missing connection between `lst`'s initialization and `lst.append(i)`. As a result, when the code attempts to append to `lst`, it triggers a `NameError` since `lst` was never initialized, highlighted in **green**. Other approaches mistakenly assume `lst` is reinitialized in each loop iteration, leading to the incorrect conclusion that `lst[0]` raises an `IndexError`. Moreover, it uses a reference mechanism (highlighted in **blue**) to link key CFG nodes during reasoning. This helps the model connect execution steps to control flow nodes, a key advantage over methods lacking this explicit referencing.

In the example on the right, VISUALCODER again shows its advantage by using the CFG to grasp the non-linear control flow. While previous methods failed to identify the incorrect use of the

float value `N` in the range function, it recognizes that the error stems from an unreachable branch of code. The CFG reveals that the `else` block with `G += 1` is never executed because `X` is always even, allowing the model to pinpoint the correct error related to the float value in range. Thus, it accurately identifies `for i in range(0, N)` as the solution.

These qualitative comparisons highlight our advantage. The red turning points in previous methods indicate breakdowns in reasoning, while the green critical points in our approach's output show how it resolves errors by aligning code with CFG.

7 Attention Pattern Analysis

In this section, we aim to analyze to determine whether the Vision-Language Model effectively

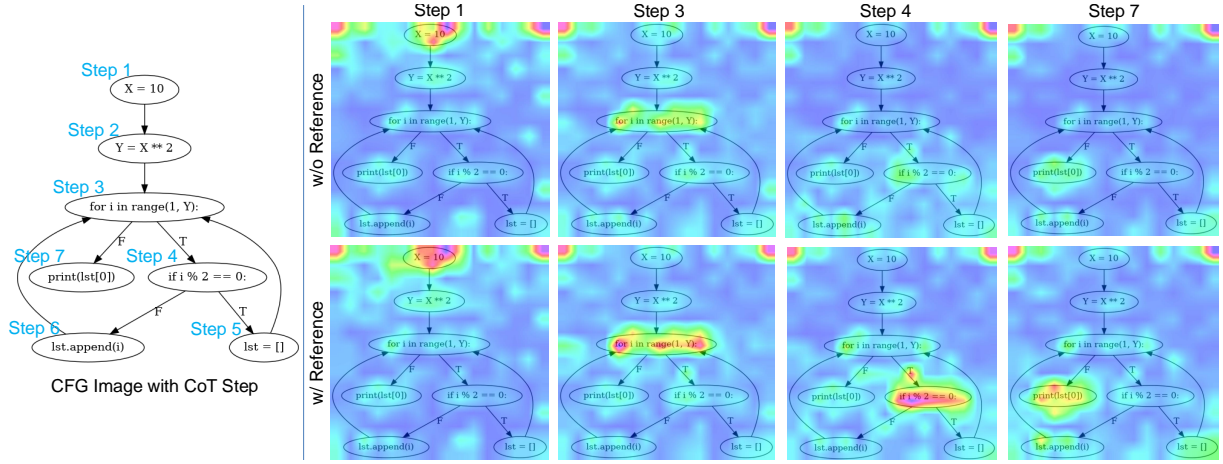


Figure 3: Attention Heat Map in CFG Image for each CoT reasoning step.

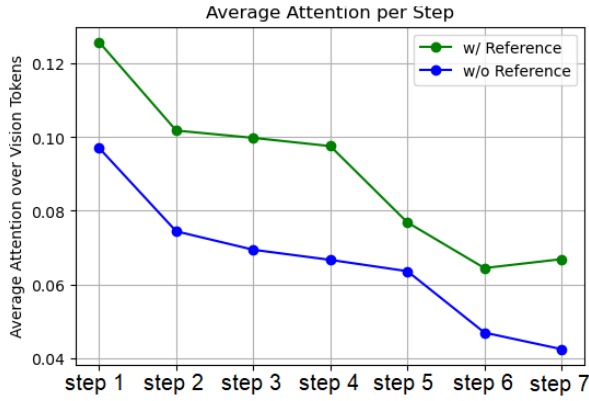


Figure 4: Average Attention Score over Vision Token in CFG Image for each CoT reasoning step.

leverages the CFG images to enhance its reasoning or simply overlooks them during inference. Specifically, we analyze the attention patterns by examining the attention matrices across all heads and layers of the InterVL2-26B model in a specific example (more details in A.2). Our focus lies on the attention weights associated with the generated rationales and their interactions with visual tokens. These weights provide valuable insights into where and to what extent the model attends to visual tokens during code execution reasoning steps.

As shown in Figure 3, the attention maps reveal key differences between the Reference CoT and vanilla CoT approaches. With the Reference CoT, the InterVL2-26B model consistently attends to the relevant nodes in the CFG image at each reasoning step. In contrast, the vanilla approach exhibits a more diffuse attention pattern, occasionally focusing on irrelevant regions, which could contribute to poorer performance in debugging tasks. Figure 4 also shows that with Reference Mechanism, VLM

more focuses on vision tokens in CFG image for reasoning, leading to better capturing of the alignment between code and CFG image. These findings also corroborate our intuition behind Equation 5. By incorporating reference mapping, the model adopts a “more focused” attention mechanism for each Chain-of-Thought step, facilitating more precise reasoning on program execution.

8 Conclusion

In conclusion, VISUALCODER enhances LLMs’ reasoning about code execution by incorporating multimodal inputs, specifically control flow graph (CFG) visualizations. Traditional LLMs, while effective at processing static code syntax, struggle to capture dynamic execution behaviors, leading to incorrect predictions and limited reasoning about program flow. By introducing the Reference CoT technique, VISUALCODER establishes explicit connections between source code lines and CFG elements, ensuring a structured and interpretable representation of execution logic. This approach reduces reasoning errors, improves alignment between textual and visual execution cues, and enables more accurate program behavior predictions. Our experimental results show that augmenting LLMs with visual CFGs significantly improves performance over text-based CFG descriptions alone, validating our multimodal approach.

Acknowledgments

Tien N. Nguyen was supported in part by the US National Science Foundation (NSF) grant CNS-2120386 and the National Security Agency (NSA) grant NCAE-C-002-2021.

9 Limitations

The quality of the CFG plays a crucial role in the performance of our method. If the CFG is incomplete or inaccurate, it can lead to flawed reasoning and missed execution paths. Additionally, we have not tested how well VISUALCODER performs when the CFG contains a large number of nodes, which could affect the graph’s resolution and the model’s ability to process fine-grained details. These factors may influence the effectiveness of the reasoning process, particularly in complex programs with extensive control flows.

Another limitation is the lack of clarity on which type of code graph (e.g., CFG, abstract syntax tree, or repository graph) is most suitable for specific coding tasks. While our work focuses on CFGs, other graph representations may be more effective for different types of code reasoning, such as syntax-based analysis or structural relationships in repositories. Identifying the optimal graph type for each task is an area requiring further exploration.

References

- Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, Andrew Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karen Simonyan. 2024. Flamingo: a visual language model for few-shot learning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS ’22*, Red Hook, NY, USA. Curran Associates Inc.
- Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. [San-tacoder: don’t reach for the stars!](#) *Deep Learning for Code (DLAC) Workshop*.
- Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2023. [Fixe-val: Execution-based evaluation of program fixes for programming problems](#). In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 11–18.
- Anthropic. 2024. Claude 3.5 sonnet. <https://www.anthropic.com/news/claude-3-5-sonnet>.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *Preprint*, arXiv:2108.07732.
- David Bieber, Rishab Goel, Dan Zheng, Hugo Larochelle, and Daniel Tarlow. 2023. [Static prediction of runtime errors by learning to execute programs with external resource descriptions](#). In *The Eleventh International Conference on Learning Representations*.
- David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. 2020. Learning to execute programs with instruction pointer attention graph neural networks. *Advances in Neural Information Processing Systems*, 33:8626–8637.
- Nghi Bui, Yue Wang, and Steven C.H. Hoi. 2022. [Detect-localize-repair: A unified framework for learning to debug with CodeT5](#). In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 812–823, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.
- Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1186–1197. IEEE.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Xinyun Chen, Dawn Song, and Yuandong Tian. 2021b. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems*, 34:22196–22208.
- Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, Bin Li, Ping Luo, Tong Lu, Yu Qiao, and Jifeng Dai. 2024. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 24185–24198.
- Hridya Dhulipala, Aashish Yadavally, and Tien N. Nguyen. 2024. Planning to guide llm for code coverage prediction. In *2024 IEEE AI Foundation Models and Software Engineering*. IEEE.

- Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. [CRUXEval: A benchmark for code reasoning, understanding and execution](#). In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 16568–16621. PMLR.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.
- Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin Clement, and Nan Duan. 2022. [Execution-based evaluation for data science code generation models](#). In *Proceedings of the Fourth Workshop on Data Science with Human-in-the-Loop (Language Advances)*, pages 28–36, Abu Dhabi, United Arab Emirates (Hybrid). Association for Computational Linguistics.
- Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, et al. 2024. Open-coder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. [Qwen2.5-coder technical report](#). *Preprint*, arXiv:2409.12186.
- Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. [Live-codebench: Holistic and contamination free evaluation of large language models for code](#). *CoRR*, abs/2403.07974.
- Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.
- Tiankai Jiang. 2023. Cfg-generator. <https://github.com/Tiankai-Jiang/CFG-Generator>. GitHub repository.
- René Just, Darioush Jalali, and Michael D. Ernst. 2014. [Defects4j: a database of existing faults to enable controlled testing studies for java programs](#). In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA. Association for Computing Machinery.
- Cuong Chi Le, Hoang Nhat Phan, Huy Nhat Phan, Tien N. Nguyen, and Nghi D. Q. Bui. 2024. [Learning to predict program execution by modeling dynamic dependency on code graphs](#). *Preprint*, arXiv:2408.02816.
- Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. 2023. Blip-2: bootstrapping language-image pre-training with frozen image encoders and large language models. In *Proceedings of the 40th International Conference on Machine Learning*, ICML’23. JMLR.org.
- R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliakhko, N. Gontier, N. Meade, A. Randy, M-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, S. Gunasekar, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries. [Starcoder: May the source be with you!](#) *Transactions on machine learning research*.
- Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180. ACM.
- Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30.
- Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault localization with code coverage representation learning. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE’21. IEEE.
- Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. [Fault localization to detect co-change fixing locations](#). In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 659–671, New York, NY, USA. Association for Computing Machinery.

- Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. 2024. Improved baselines with visual instruction tuning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 26296–26306.
- Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. [Visual instruction tuning](#). In *Advances in Neural Information Processing Systems*, volume 36, pages 34892–34916. Curran Associates, Inc.
- Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676.
- Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.
- Dung Nguyen Manh, Thang Phan Chau, Nam Le Hai, Thong T Doan, Nam V Nguyen, Quang Pham, and Nghi DQ Bui. 2024. Codemmlu: A multi-task benchmark for assessing code understanding capabilities of codellms. *arXiv preprint arXiv:2410.01999*.
- Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. [Improving fault localization and program repair with deep semantic features and transferred knowledge](#). In *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*, page 1169–1180, New York, NY, USA. Association for Computing Machinery.
- Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.
- Dung Nguyen, Le Nam, Anh Dau, Anh Nguyen, Khanh Nghiem, Jin Guo, and Nghi Bui. 2023. The vault: A comprehensive multilingual dataset for advancing code understanding and generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 4763–4788, Singapore. Association for Computational Linguistics.
- Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. [NExT: Teaching large language models to reason about code execution](#). In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 37929–37956. PMLR.
- Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.
- OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rameez Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O’Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov,

- Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. [Gpt-4 technical report](#). *Preprint*, arXiv:2303.08774.
- Huy N Phan, Hoang N Phan, Tien N Nguyen, and Nghi DQ Bui. 2024a. Repohyper: Better context retrieval is all you need for repository-level code completion. *arXiv preprint arXiv:2403.06095*.
- Huy Nhat Phan, Tien N Nguyen, Phong X Nguyen, and Nghi DQ Bui. 2024b. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*.
- Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. [Learning transferable visual models from natural language supervision](#). In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR.
- Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. [Code llama: Open foundation models for code](#). *Preprint*, arXiv:2308.12950.
- Kensen Shi, Joey Hong, Yinlin Deng, Pengcheng Yin, Manzil Zaheer, and Charles Sutton. 2024. [Exedec: Execution decomposition for compositional generalization in neural program synthesis](#). In *The Twelfth International Conference on Learning Representations*.
- Eui Chul Shin, Illia Polosukhin, and Dawn Song. 2018. Improving neural program synthesis with inferred execution traces. *Advances in Neural Information Processing Systems*, 31.
- Matt Stallone, Vaibhav Saxena, Leonid Karlinsky, Bridget McGinn, Tim Bula, Mayank Mishra, Adriana Meza Soria, Gaoyuan Zhang, Aditya Prasad, Yikang Shen, et al. 2024. Scaling granite code models to 128k context. *arXiv preprint arXiv:2407.13739*.
- Hung Quoc To, Nghi DQ Bui, Jin Guo, and Tien N Nguyen. 2023. Better language models of code through self-improvement. *arXiv preprint arXiv:2304.01228*.
- Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.
- Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.
- Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2024a. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS '22*, Red Hook, NY, USA. Curran Associates Inc.
- Yanbin Wei, Shuai Fu, Weisen Jiang, Zejian Zhang, Zhixiong Zeng, Qi Wu, James Kwok, and Yu Zhang. 2024b. [GITA: Graph to visual and textual integration for vision-language graph reasoning](#). In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.
- Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024c. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning*.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*, pages 476–486.

Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. Cnn-fl: An effective approach for localizing faults using convolutional neural networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 445–455. IEEE.

Zhuosheng Zhang, Aston Zhang, Mu Li, Hai Zhao, George Karypis, and Alexander J. Smola. 2023. *Multimodal chain-of-thought reasoning in language models*. *Trans. Mach. Learn. Res.*, 2024.

Wei Zheng, Desheng Hu, and Jing Wang. 2016. *Fault localization analysis based on deep neural network*. *Mathematical Problems in Engineering*, 2016.

Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. 2024. *MiniGPT-4: Enhancing vision-language understanding with advanced large language models*. In *The Twelfth International Conference on Learning Representations*.

A Appendix

A.1 Control Flow Graph Representation

Definition 1 (Control Flow Graph - CFG)

A *Control Flow Graph (CFG)* is a graphical representation of the control flow within a program. Nodes in the CFG correspond to basic blocks of code, which may include individual statements or groups of statements that are executed sequentially. The edges between nodes represent the possible transitions or flow of control between these blocks, typically influenced by control structures such as loops, conditional statements (e.g., if-else), or function calls.

In VISUALCODER, the CFG serves as a crucial component for visualizing and reasoning about a program’s execution flow. By aligning each code segment with its corresponding node in the CFG, we provide the model with a more structured and intuitive understanding of the dynamic behavior of the program. This enhanced alignment helps in improving code execution reasoning, error detection, and prediction of execution outcomes.

To generate the Control Flow Graphs (CFGs) used in VISUALCODER, we adapted code from an open-source repository by (Jiang, 2023). The modifications made to the original code focused on improving clarity and reducing unnecessary information in the CFG. Specifically, we removed certain function call nodes that did not correspond to any specific line of code, thus eliminating extraneous details that could distract the model. Additionally, we simplified the labels on the edges of conditional branches by replacing the full conditional statements with "T" (True) and "F" (False).

In addition to the visual representations of Control Flow Graphs (CFGs), we utilize the Mermaid language to provide a text-based representation. The following Mermaid code corresponds to the CFG depicted in Figure 1:

```
graph TD
  A["X = 1024"] --> B["N = X / 500"]
  B --> C["for i in range(10):"]
  C --> D["if X % 2 == 0:"]
  C --> E["for i in range(0, N):"]
  D --> F["N += 1"]
  D --> G["G += 1"]
  E --> H["X += 100"]
  E --> I["print(X)"]

  D -->|T| F
  D -->|F| G
  E -->|T| H
  E -->|F| I
```

A.2 VLM: Patching and Vision Token

In this section, we describe how we process input contain both text and image using VLM. We use InternVL2-26B to handle the Control Flow Graph (CFG) images. The images are first resized to a resolution of 448x448, then patched and tokenized into 16x16 vision tokens <IMG_CONTEXT>. These vision tokens are inserted between the language tokens, enclosed by and tags, allowing the model to incorporate the visual information seamlessly into the multimodal input.

After generating the reasoning steps through Chain of Thought (CoT), we map each step to its corresponding language token. To further analyze the model’s interaction with the visual context, we calculate the attention scores between each reasoning step and the vision tokens. This process helps us track how much attention the model allocates to the visual tokens at each step. We then plot a

heatmap to visualize these attention patterns and compute the average attention score over the vision tokens, providing insight into the model’s focus on visual information during the reasoning process.

A.3 Coding task details

This section describes the three coding tasks used to evaluate our approach: **Input/Output Prediction**, **Program Repair**, and **Fault Localization**. These tasks were selected to assess the model’s ability with the help of Control Flow Graphs (CFGs).

In **Input/Output Prediction**, the model predicts the output of a Python code snippet given specific inputs, or vice versa. This task tests the model’s understanding of execution flow, including variable assignments, loops, and conditionals. CFGs are crucial here as they provide a visual representation of the control flow, helping the model trace execution paths more effectively and make accurate predictions.

In the **Program Repair** task, the model is given a buggy code and must generate a corrected version. CFGs assist the model by highlighting the control flow paths that lead to errors, allowing it to focus on areas where the logic may have broken down. The use of CFGs helps the model better understand the code’s intended execution, leading to more accurate fixes.

The **Fault Localization** task requires the model to pinpoint the exact lines of code responsible for failures. By leveraging CFGs, the model gains a structured view of the execution flow, enabling it to trace how different parts of the code are interconnected. This visual representation helps the model pinpoint problematic lines more effectively by clarifying control paths and dependencies, offering a deeper understanding of the error’s source.

A.4 Prompt

In this section, we present the different prompt configurations used to guide the Vision-Language Model (VLM) in analyzing Python code execution for **Fault Localization** task. These prompts are designed to test different reasoning approaches, including plain code analysis, Chain-of-Thought (CoT) reasoning, and the integration of Control Flow Graphs (CFGs) to enhance the model’s understanding of the code execution flow. The prompt configurations range from basic setups to more sophisticated ones, as shown in Figure 5, 6, 7, 8. Furthermore, we implemented a prompt with the Reference Mechanism, as shown in Figure 9, which

explicitly links the reasoning steps with corresponding CFG nodes, thereby grounding the model’s understanding of the control flow. Finally, Figure 13 demonstrates a two-stage prompt that incorporates the Reference Mechanism during the Rationale Generation phase, significantly improving the model’s capability in error detection and reasoning.

You are an expert Python programmer.
Analyze the following Python code snippet, which contains error(s) when executing:

{code}

Respond with only the problematic line of code that causes termination.

Figure 5: Plain code w/o CoT prompt

You are an expert Python programmer.
Analyze the following Python code snippet, which contains error(s) when executing:

{code}

As you analyze each line:

1. Examine each line of code sequentially.
2. Use this understanding to support your reasoning about the code's logic and potential errors.

Think through your analysis step by step, and then respond with only the problematic line of code that causes termination.

Figure 6: Plain code w/ CoT prompt

You are an expert Python programmer.
Analyze the following Python code snippet, which contains error(s) when executing:

{code}

You will also be provided with a control flow graph (CFG) image of this code. Respond with only the problematic line of code that causes termination.

Figure 7: Plain code + CFG w/o CoT prompt

You are an expert Python programmer.

Analyze the following Python code snippet, which contains error(s) when executing:

{code}

You will also be provided with a control flow graph (CFG) image of this code. As you analyze each line:

1. Examine each line of code sequentially.
2. Use this understanding to support your reasoning about the code's logic and potential errors.

Think through your analysis step by step, considering both the code and its representation in the CFG image. After your analysis, respond with only the problematic line of code that causes termination.

Figure 8: Plain code + CFG w/ CoT prompt

You are an expert Python programmer.

Analyze the following Python code snippet, which contains error(s) when executing:

{code}

You will also be provided with a control flow graph (CFG) image of this code. As you analyze each line:

1. Examine each line of code sequentially.
2. Reference the CFG to identify which node corresponds to the line you're currently analyzing.
3. Use this alignment to support your reasoning about the code's logic and potential errors.

Think through your analysis step by step, considering both the code and its representation in the CFG image. After your analysis, respond with only the problematic line of code that causes termination.

Figure 9: VisualCoder prompt

Stage 1:

You are an expert Python programmer.

Analyze the following Python code snippet, which contains error(s) when executing:

{code}

You will also be provided with a control flow graph (CFG) image of this code. As you analyze each line:

1. Examine each line of code sequentially.
3. Use this understanding to support your reasoning about the code's logic and potential errors.

After your analysis, provide a detailed rationale explaining what might be wrong with the code.

Figure 10: Rationale Generation prompt

You are an expert Python programmer.

Analyze the following Python code snippet, which contains error(s) when executing:

{code}

You will also be provided with a control flow graph (CFG) image of this code. As you analyze each line:

1. Examine each line of code sequentially.
2. Reference the CFG to identify which node corresponds to the line you're currently analyzing.
3. Use this alignment to support your reasoning about the code's logic and potential errors.

After your analysis, provide a detailed rationale explaining what might be wrong with the code.

Figure 11: Rationale Generation w/ Reference Mechanism prompt

Stage 2:

You have a Python code snippet containing error(s) and a rationale for the error(s).Code:

{code}

Rationale: {rationale}

Using this rationale, please identify the specific line of code that causes termination. Respond with only the problematic line of code that causes termination."""

Figure 12: Answer Inference prompt

Figure 13: Multimodal-CoT two-stage prompt