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

Anonymous ACL submission

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 flow, enabling more accurate predictions of code behavior. Our experiments demonstrate that augmenting LLMs with visual CFGs significantly outperforms text-based CFG descriptions in code reasoning tasks. 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.

1 Introduction

Recent advancements in Large Language Models (LLMs) (Hui et al., 2024; Rozière et al., 2024) 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). 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, most existing work focuses on a single-modality input (i.e., plain code) and has yet to explore the potential of multimodal approaches for code execution reasoning. While code can be read in a linear fashion, understanding its full behavior requires focusing on the non-linear structure of its execution, something that is often visualized more clearly through a CFG.

In recent years, Vision-Language Large Models (VLLMs) (OpenAI et al., 2024; Chen et al., 2023; 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 advancements in multimodal LLMs, such as Flamingo (Alayrac et al., 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 code execution reasoning of LLMs by leveraging multimodal reasoning, combining plain code with visual representations of the corresponding CFG. In our preliminary experiments, simply presenting the plain code alongside textual or visual representations of the CFG has poor performance for code 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 method 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 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.

2.2 Reasoning about Program Execution

Research into program execution reasoning has advanced through various approaches. They use execution states from constructed programs (Chen et al., 2021; 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; **?**), often relying on specialized architectures to model flows and data 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 advancements in Large Language Models (LLMs) have shown promise in tasks like code 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*

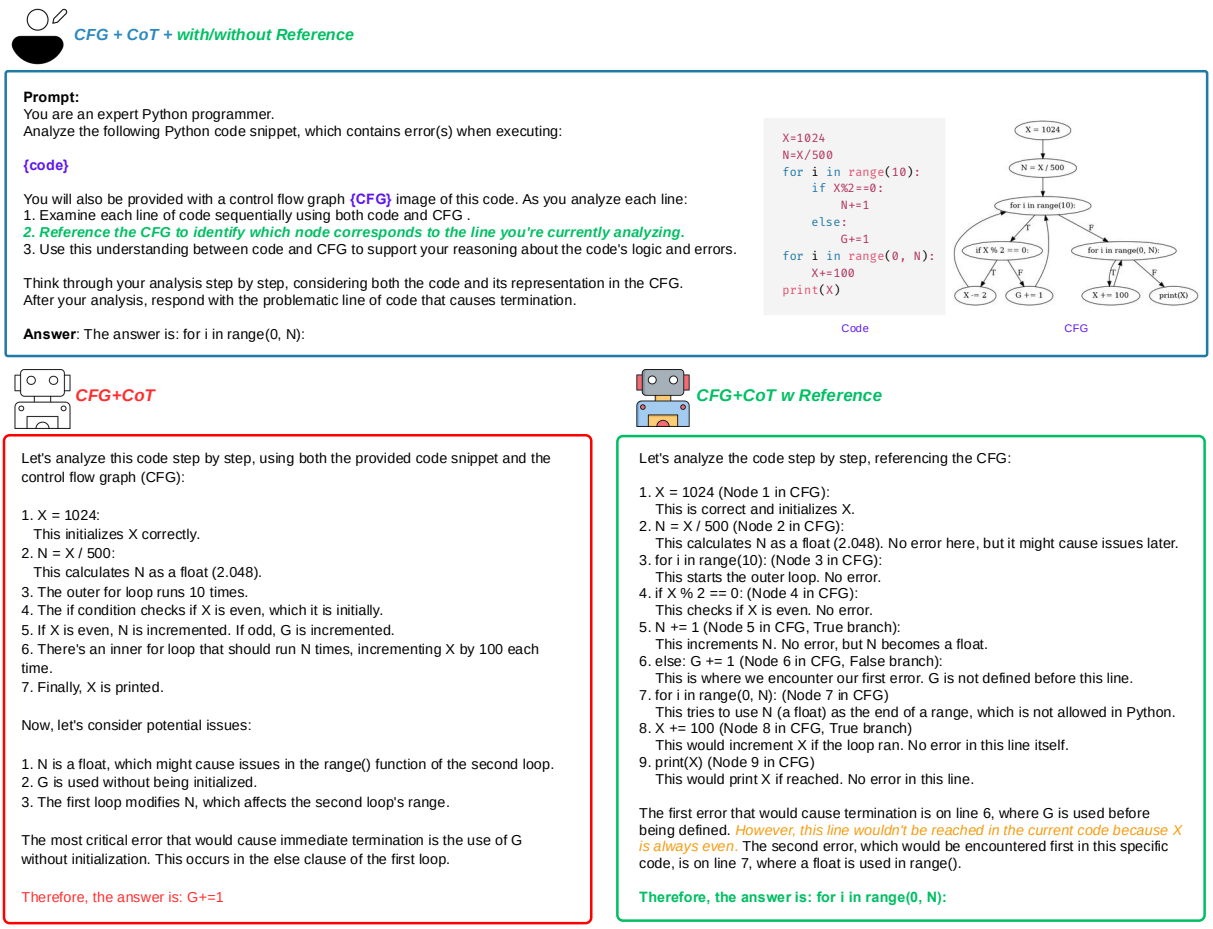

Figure 1: Comparison of Code Execution Reasoning: CFG + CoT w/o Reference vs. CFG + CoT w Reference. With reference, LLM correctly identifies the unreachable node and critical termination point (highlighted in orange).

and branches, which are harder to grasp through sequential textual descriptions alone. This result is also consistent with the one in (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 execute.

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 yellow). 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 code 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.

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 facilitate enhanced code execution reasoning. 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 Python code snippet be represented as a sequence of lines of code:

$$\text{Code} = \{C_1, C_2, \ldots, C_n\} \tag{1}$$

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:

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

where $N = \{N_1, N_2, \ldots, N_m\}$ 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. We define the reasoning process as a recursive function:

$$R(C_i) = f(C_i, \{C_1, C_2, \ldots, C_{i-1}\}) \tag{3}$$

where f is a function that takes the current line of code and the context, iterating through each step of the code while considering the nodes in the CFG.

4.2 Reference Mechanism

The **Reference Mechanism** augments the CoT reasoning by mapping each line of code C_i to its respective CFG node. This mapping is expressed as: $M : C_i \mapsto N_j$, where C_i is represented by node N_j in the CFG. The model now references N_j during reasoning to ensure consistency between the plain code and control flow. This alignment helps the model analyze each line within the control flow, improving its 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:

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

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 $(N_1, N_2, \ldots, N_m)$ 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:

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

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 control flow and the sequential 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 7, demonstrate the effectiveness of our method in enhancing code execution reasoning.

5 Experiments

5.1 Better Code Execution Understanding with Control Flow Graph

In this experiment, we aimed to show that providing the LLM with a CFG improves its understanding of code execution. Using the CRUXEval benchmark (Gu et al., 2024), we tested models on predicting execution outputs. We compared the performance of three state-of-the-art VLM models—Claude Sonnet 3.5 (Anthropic, 2024), Gemini-1.5-Flash (Reid et al., 2024), and InterVL2-8B (Chen et al., 2023)—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 Accuracy@1 metric, indicating if the models' first predictions were correct.

Task	Settings	Models	Accuracy@1
Output Prediction	Plain code	Claude Sonnet 3.5	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 Prediction	Plain code	Claude Sonnet 3.5	75.2
	Plain code + CFG image	Claude Sonnet 3.5	**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

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 behaviors more accurately. This result is consistent with the one reported by (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-Sonet	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 instead of code. 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 A.3). For Pro-

gram 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 (Odena et al., 2021). We sampled six solutions using Claude 3.5 Sonnet and Haiku with a 3:1 ratio, ensuring varied difficulty levels. After filtering out fully correct or incorrect solutions, we ensure the direction of code is correct but with some mistakes. We finalized 384 solutions for 173 problems. For **Fault Localization**, we used the FixEval dataset (Haque et al., 2022), which has about 210 programs with diverse runtime errors. The prompt used was listed in A.4.

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 Claude Sonnet 3.5 improved from 64.1% (plain code) to 63.0% (with CoT), and GPT-4o increased from 38.7% to 40.1%. However, combining CoT with CFG images caused a notable performance drop across all models. For instance, Claude Sonnet 3.5'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 (e.g., plain code without CoT for Claude at 64.1% or GPT-4o at 40.1%), it significantly boosted performance over the plain code + CFG w/ CoT setting. It raised Claude Sonnet 3.5 from 55.5% to 62.9%,

Tasks	Settings	Claude Sonet 3.5	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	38.7	6.3
	Multimodal-CoT + VISUALCODER	60.1	38.2	**10.7**
Fault Localization	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

and GPT-4o from 37.6% to 38.7%. 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 Sonnet 3.5 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.

The real improvement came with VISUALCODER. As shown in Table 3, VISUALCODER achieved the highest accuracy for both Claude Sonnet 3.5 (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 significantly 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.

6 Qualitative Analysis

Figure 2 presents two examples of buggy code alongside their corresponding CFGs and the rea-

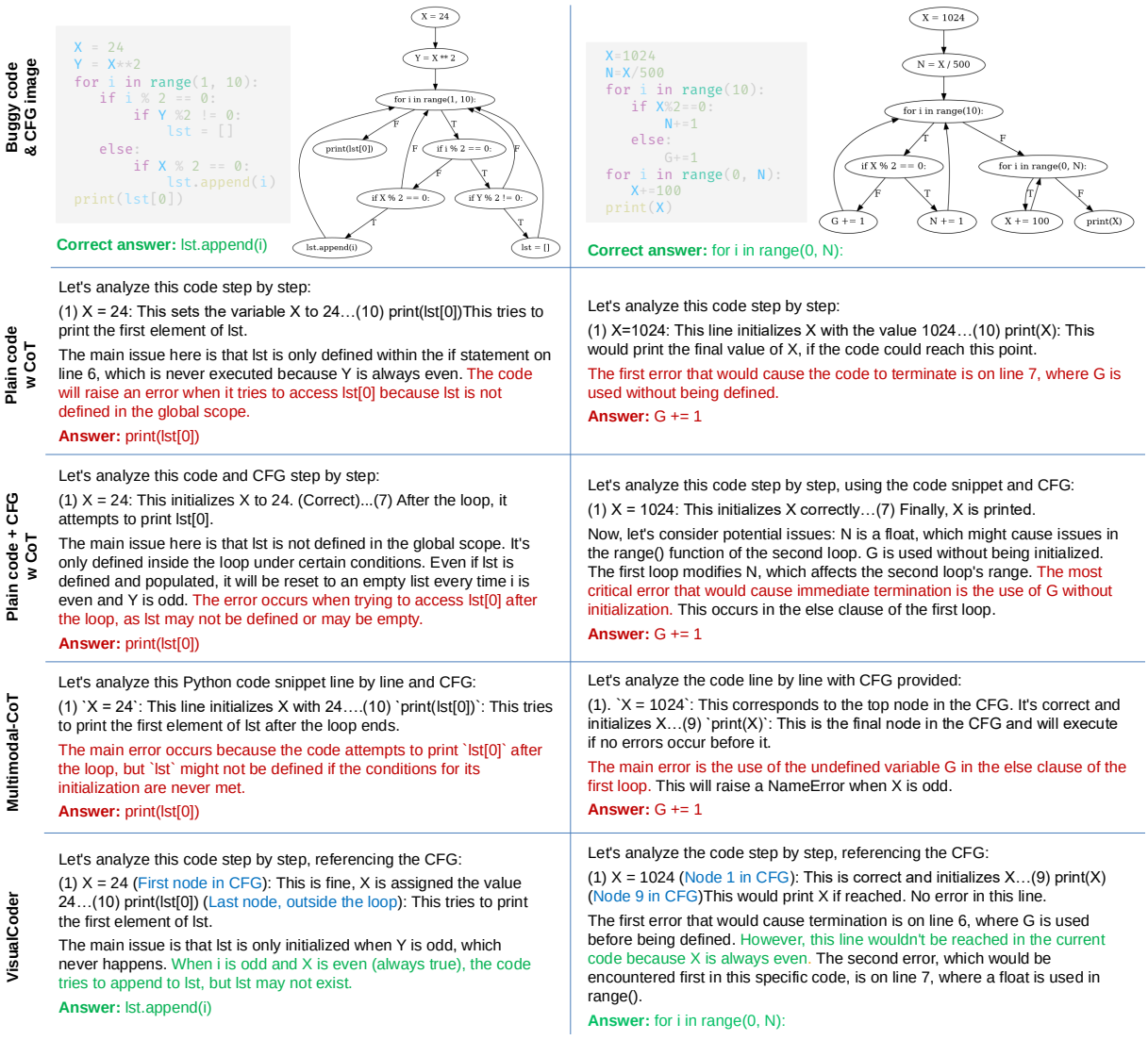

Figure 2: Qualitative comparison of reasoning outputs for buggy code using different prompt settings in Claude Sonet 3.5. Red text indicates where the reasoning fails, green text highlights correctly identified critical points, and blue text in VISUALCODER shows the referencing from the plain code to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

soning outputs of Claude Sonet 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 wrongly 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 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

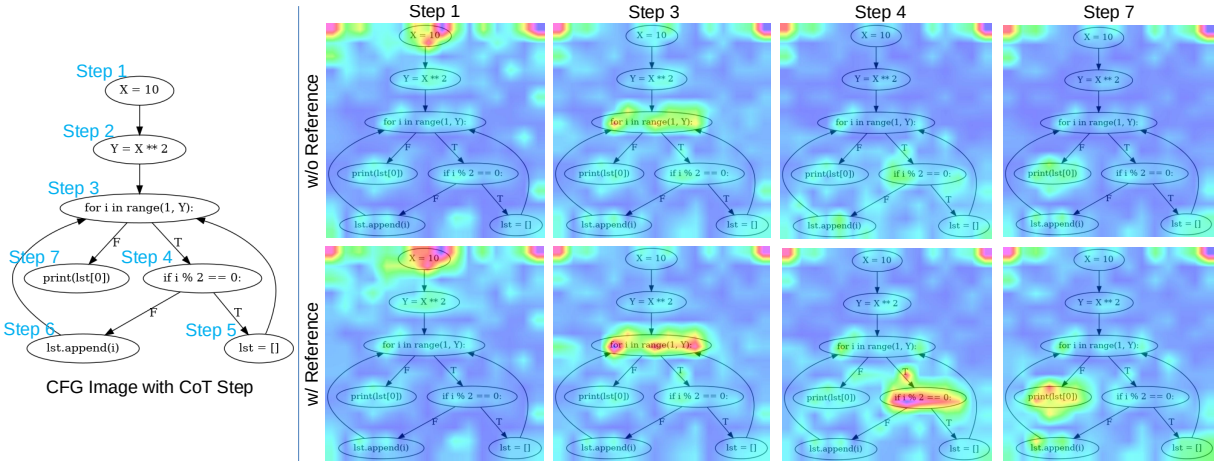

Figure 3: Attention Heat Map in CFG Image for each CoT reasoning step.

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

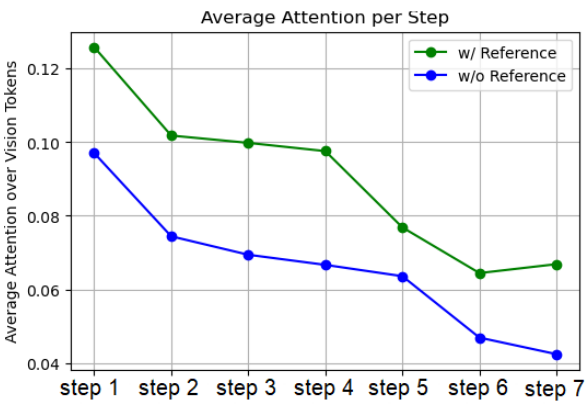

Figure 4: Average Attention Score over Vision Token in CFG Image for each CoT reasoning step.

In this section, we aim to analyze to determine whether the Vision-Language Model effectively 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 (CoT) step, facilitating more precise reasoning.

8 Conclusion

In conclusion, VISUALCODER enhances LLMs' understanding of code execution by integrating multimodal inputs, specifically control flow graph (CFG) visualizations. While traditional LLMs handle static code well, they struggle with dynamic behaviors. Our Reference CoT technique links code lines with CFG elements, reducing errors and providing a clearer representation of execution flow. Our results show that visual CFGs improve LLMs' reasoning capabilities on code execution.

9 Limitations

Our method has not yet been extensively tested on large-scale codebases or repository-level projects with numerous files. While it performs well on individual code snippets, generating a full CFG for an entire repository presents significant challenges due to the complexity and scale involved. Constructing CFGs for large repositories with many interconnected files can be computationally expensive and difficult to manage efficiently.

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.

Finally, our method has been primarily evaluated on Python-based benchmarks, leaving its generalizability across other programming languages in question. Since different languages have distinct control structures and syntax, VISUALCODER's ability to generate accurate CFGs and reason effectively may vary. Future work should explore its applicability to a wider range of programming languages and paradigms to assess its broader utility.

References

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millicah, 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.

Anthropic. 2024. Claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet.

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.

Xinyun Chen, Dawn Song, and Yuandong Tian. 2021. 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. 2023. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. *arXiv preprint arXiv:2312.14238*.

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.

Md. Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2022. Fixeval: Execution-based evaluation of program fixes for competitive programming problems. *ArXiv*, abs/2206.07796.

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.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *CoRR*, abs/2403.07974.

Tiankai Jiang. 2023. Cfg-generator. https://github.com/Tiankai-Jiang/CFG-Generator. GitHub repository.

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.

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.

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.

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.

Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. 2021. Program synthesis with large language models. In *n/a*, page n/a, n/a. N/a.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, 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, Rajeev 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.

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.

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 T. Kwok, and Yu Zhang. 2024b. Gita: Graph to visual and textual integration for vision-language graph reasoning. *Preprint*, arXiv:2402.02130.

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 <img> and </img> 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.

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

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}

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