VISUALCODER: GUIDING VISION LANGUAGE MODELS IN CODE EXECUTION WITH FINE-GRAINED CHAIN-OF-THOUGHT REASONING

Anonymous authors
Paper under double-blind review

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 novel approach that enhances code reasoning by integrating multimodal Chain-of-Thought (CoT) reasoning with visual Control Flow Graphs (CFGs). 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 logical problem. There are diverse approaches aimed at enhancing a model's 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 state changes that evolve during execution. They do not inherently simulate code execution, which is necessary for understanding how variables and control flow evolve at runtime. Furthermore, LLMs lack the ability to track mutable state or anticipate runtime-specific conditions, leading to difficulties in predicting behavior that depends on dynamic, 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 control flow representations.

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 (VQA), 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. (2023) 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. (2024).

In this work, we propose enhancing the code execution reasoning of Large Language Models (LLMs) by leveraging multimodal reasoning, combining plain code with visual representations of the corresponding control flow graph (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 (Sections 5). Recent work by (Zhang et al., 2023) focuses on improving multimodal reasoning in LLMs using the prominent Chain-of-Thought (CoT) prompting technique (Wei et al., 2023) in which the solution has two separate steps: rationale generation and reasoning to produce answers. However, when applied to our multimodal setup of plain code and CFG, their CoT prompting approach suffers from cascading errors, where inaccuracies in rationale generation negatively impact the reasoning and final answers. To address this, 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 & PROGRAM REPAIR

Recent deep learning-based fault localization (FL) techniques such as GRACE Lou et al. (2021), DeepFL Li et al. (2019), CNNFL Zhang et al. (2019), and DeepRL4FL (Li et al., 2021) have achieved significant advancements in FL performance. GRACE, for instance, employs a novel graph-based representation for methods and ranks potentially faulty methods more effectively. Earlier ML-based approaches, including MULTRIC (Xuan & Monperrus, 2014), TrapT (Li & Zhang, 2017), and Fluccs (Sohn & Yoo, 2017), laid the foundation for these improvements. Neural network-based FL methods initially relied heavily on test coverage data (Zheng et al., 2016; Briand et al., 2007; Zhang et al., 2017; Wong & Qi, 2009; Li & Zhang, 2017), but they faced challenges in differentiating between elements executed by failed tests and truly faulty components (Li & Zhang, 2017). To address these shortcomings, more advanced techniques such as TRANSFER (Meng et al., 2022), which leverages deep semantic features and transferred knowledge from open-source projects, and FixLocator (Li et al., 2022a), which detects co-fixing locations, were introduced. Additionally, CodeT5-DLR (Bui et al., 2022) presents an end-to-end approach using large language models (LLMs) to detect, localize, and repair bugs sequentially. Automated program repair tools (Li et al., 2022b) focus on both identifying and fixing buggy hunks, while other approaches (Li et al., 2022b) emphasize the integration of FL and repair. Several works in program repair have leveraged execution information such as traces (Gupta et al., 2020) or test diagnostics (Ye et al., 2022).

2.2 REASONING ABOUT PROGRAM EXECUTION

Research into reasoning about program execution has progressed through various approaches, particularly in the domain of program synthesis. These systems frequently use execution states from partially constructed programs Chen et al. (2021); Ni et al. (2024b); Shin et al. (2018), or predict intermediate execution subgoals to improve search strategies in sequence-to-sequence models Shi et al. (2023). Another prominent approach involves training neural networks to simulate program execution, functioning like a learned interpreter Bieber et al. (2020); Nye et al. (2021). These efforts often rely on customized neural architectures to model execution flows and handle data dependencies.

Our work diverges from these approaches by concentrating on large language models (LLMs) that reason about execution in natural language, eliminating the need for specialized architectures. Prior works such as Scratchpad and Self-Debugging have explored LLMs in this space, focusing on generating reasoning chains that incorporate execution details, including variable states or their natural language explanations. NExT (Ni et al., 2024b) utilizes real execution traces generated at runtime. This method allows for more focused and succinct rationales that are better suited for specific downstream tasks.

3 MOTIVATION

Recent advancements in Large Language Models (LLMs) have demonstrated their potential in addressing complex tasks such as code execution prediction, e.g., combined with Chain-of-Thought (CoT) reasoning Dhulipala et al. (2024). However, LLMs still encounter significant challenges in fully understanding the execution flows inherent in complex code structures such as iterations and conditions. Plain code provides a linear view of logic, which often overlooks deeper relations between different segments of the code. Our experimental results in Table 1 (see details later), show that incorporating **Control Flow Graph (CFG)** along with the code significantly improves model performance across the tasks. CFG images offer a visual structure that captures the flow of execution, highlighting important control mechanisms such as branches, loops, and conditional dependencies. This additional layer of information enables the model to better grasp the interaction between code blocks, and better understand the code's non-linear execution paths, which are crucial for reasoning about program runtime behavior more effectively.

Choosing the appropriate type of data representation for the CFG plays a critical role in determining how effectively LLMs understand code execution. To motivate the use of visual representation, we conducted an experiment to compare the effectiveness of the textual representation and the visual image of the CFG. 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, CFG images potentially offer a rich, intuitive visualization of execution paths, making the intricate relationships between different code blocks more apparent. 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 one in Wei et al. (2024), 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 for tasks that require deep structural reasoning, particularly in non-linear and complex code scenarios during predictive code execution.

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 address the specific challenges of code reasoning.

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

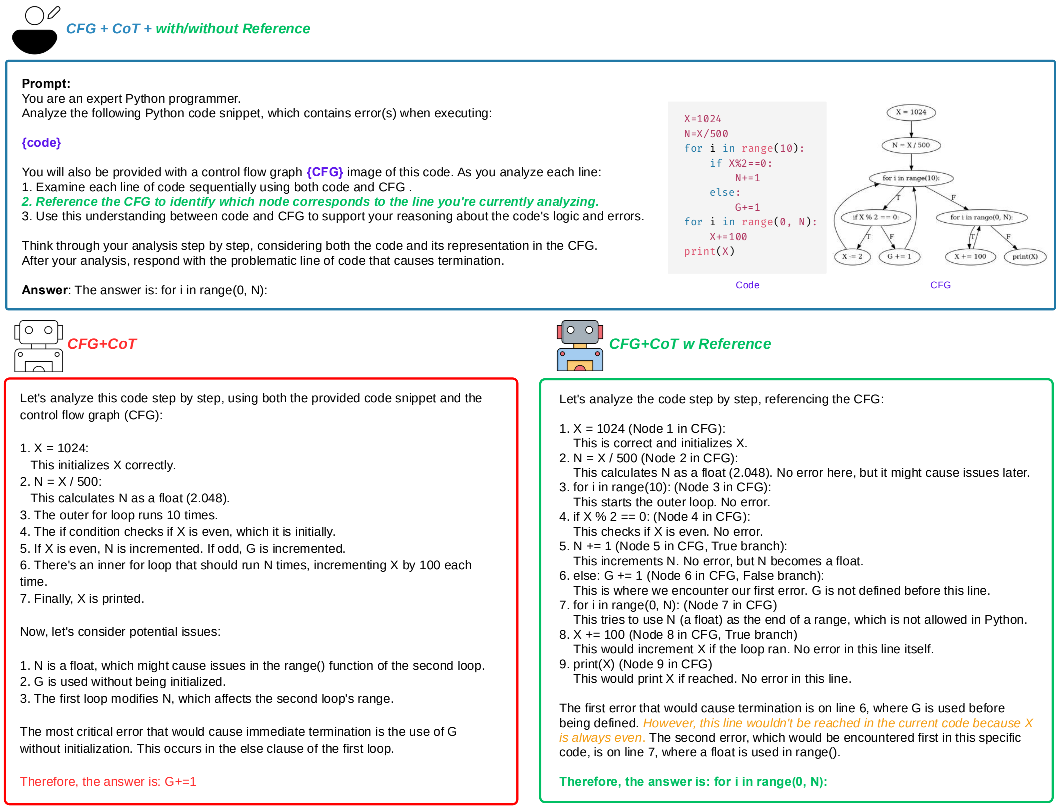

Figure 1: Comparison of Code Execution Reasoning: CFG + CoT w/o Reference vs. CFG + CoT w Reference. The reference-based method correctly identifies the unreachable node and critical termination point (highlighted in orange).

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 predictions and reasoning.

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

4 APPROACH: REFERENCE MECHANISM

In this section, 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.

4.1 Overview

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

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

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 $N = \{N_1, N_2, \dots, N_m\}$ 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 corresponding node N_i in the CFG, and utilize this to perform stepwise reasoning.

4.2 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:

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

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

4.3 Reference Mechanism

The **Reference Mechanism** augments the CoT reasoning by mapping each line of code C_i to its corresponding node in the CFG. This mapping can be expressed as:

$$M : C_i \mapsto N_j, \quad \text{where } C_i \text{ is represented by node } N_j \text{ in the CFG}$$

The model now references N_j during the reasoning process to ensure consistency between the flow of plain code and control flow structure. This alignment ensures that the model not only analyzes the code line by line but also understands how each line fits into the overall control flow of the program. By referencing the CFG, the model gains a clearer view of execution paths, transitions, and dependencies between different statements in the same block and between different blocks, improving its ability to reason about the entire code structure rather than treating each line in isolation. Currently, we achieve this by adding a simple sentence instructing the model to reference the CFG during code analysis (the line in prompt highlighted by green color in Figure 1).

4.4 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:

$$p_{\text{no-ref}}(Y|C_1, \dots, C_n, \text{CFG}) = \prod_{i=1}^{n} \mathcal{P}(Y_i|C_1, \dots, C_i, \text{CFG}) \tag{4}$$

$$= \prod_{i=1}^{n} \mathcal{P}(Y_i|C_1, \dots, C_i, (N_1, N_2, \dots, N_m), E) \tag{5}$$

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.

4.5 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:

$$p_{\text{ref}}(Y|C_1,\ldots,C_n,\text{CFG}) = \prod_{i=1}^{n} \mathcal{P}(Y_i|(C_1,M(C_1)),\ldots,(C_i,M(C_i)),E) \tag{6}$$

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 lines of code, improving reasoning accuracy.

4.6 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 this work, 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 6. 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, 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 aim to demonstrate that by providing the LLM with a CFG, we can improve its ability in understanding code execution. We performed our experiment on the CRUXEval benchmark Gu et al. (2024), where models were tested on their ability to predict code execution outcomes. We compared 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. (2024)—in two settings: 1) plain code only, and 2) plain code with its CFG image. The task involved both **output prediction** (predicting the result of running the code) and **input prediction** (predicting which inputs would lead to a specific outcome).

For consistency and to ensure a direct comparison with prior work, we used the same prompt format as described in the original CRUXEval paper Gu et al. (2024). This prompt provided the models with the code and, where applicable, a visual CFG representation, guiding them through a step-by-step reasoning process. The Accuracy@1 metric was used to measure performance, capturing whether the models' first predictions were correct—an important indicator of their immediate understanding of code execution. The diverse range of code structures in CRUXEval ensured that the models were tested on realistic, complex code scenarios.

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: Comparison of models with single and multiple modalities on code execution prediction.

The results in Table 1 demonstrate 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 significantly improve performance on code coverage prediction.

5.2 Rich Information Encoded in CFG Images vs. Text-Based Descriptions

To evaluate the impact of visual representations in coding tasks, particularly in *Code Execution Prediction*, we conducted another experiment in which various LLM models were provided with either CFG in Mermaid format (text-based CFG) or CFG images of the code, along with the input, and tasked with predicting the code's output. The prompt remained the same as used in the previous experiment, but instead of code, the models were given either the text-based or image-based CFG of the original code. The results in Table 2 demonstrate that CFG images significantly improve the performance in reasoning tasks involving code execution flow, highlighting the value of visual representations in enhancing Multimodal LLMs' reasoning abilities.

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.

5.3 Challenges in Multi-modal Reasoning with Control Flow Graphs and Chain of Thought

5.3.1 Experiment setting

This experiment involved two tasks: **Program Repair** and **Fault Localization**. For the **Program Repair** task, we generated our own dataset by selecting instances from LiveCodeBench Jain et al. (2024), focusing on challenging cases requiring complex reasoning and control/data flow analysis. From 400 instances, we sampled six solutions using Claude Sonnet 3.5 (75%) and Haiku models (25%). We excluded solutions that either passed or failed all test cases, retaining only partially correct solutions. After further filtering, we finalized 384 solutions for 173 problems. This dataset emphasized debugging solutions where intricate control and data flow graphs play a critical role in repairing the code.

For the **Fault Localization** task, we used the FixEval dataset Haque et al. (2022), consisting of approximately 210 programs with diverse runtime errors. This task focused on identifying the faulty code segments responsible for the errors, making it an excellent benchmark to assess the models' ability to detect and localize errors in real-world code scenarios.

We evaluated the models in multiple configurations: plain code (with and without Chain-of-Thought reasoning), plain code combined with CFGs, plain code with execution in-line comment (NeXT Ni et al. (2024a)), **Multimodal-CoT** from Zhang et al. (2023) and our method VisualCoder (combined with Multimodal-CoT). For the **VisualCoder + Multimodal-CoT** setting, we incorporated our method by applying a reference mechanism during the first stage of Rationale Generation, where each line of code was linked to the corresponding part of the CFG. The second stage, Answer Inference, remained the same as in the original Multimodal-CoT framework. This allowed us to compare how well the models reasoned about execution flows in each configuration.

5.3.2 Experiment result and Analysis

When we introduced a CFG image to the vanilla prompt (containing buggy code but no CoT reasoning), we observed a notable increase in performance compared to the vanilla setting. This confirms our earlier findings that CFG images provide valuable structural information, enabling the model to better understand the code's execution flow and the dependencies between code blocks. The

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	37.2	**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: Preliminary Experiment Results Showing the Impact of CFG and CoT on Code Understanding Tasks.

improvement in this setting highlights how visual representations like CFGs can enhance code comprehension by offering insights that are not easily extracted from plain text.

However, when we combined the CFG image with Chain-of-Thought reasoning (CoT) in the prompt (**+ CFG + CoT**), performance unexpectedly dropped compared to using CoT reasoning alone and the vanilla setting. This suggests that the model struggled to effectively integrate the visual information from the CFG with its CoT reasoning process. This aligns with challenges highlighted in the work of Zhang et al. (2023), which points out that combining CoT with multimodal inputs often leads to hallucinations or misaligned reasoning steps, as the model is unable to fuse the textual and visual modalities coherently. Due to insufficient training on such multi-modal inputs, the model generated reasoning steps that did not match the actual execution flow represented by the CFG. Instead of enhancing its reasoning, the additional modality caused confusion, leading to reduced accuracy despite the richer input.

In the **Program Repair** task, the results indicate that plain code settings, with or without CoT, show limited improvement in performance. For instance, the plain code without CoT setting results in an accuracy of 64.1% for Claude Sonnet 3.5, while using CoT slightly decreases the performance to 63.0%. This trend is consistent across GPT-4o and InternVL2, suggesting that applying CoT alone in this task does not significantly help the models' reasoning. In contrast, when CFGs are introduced alongside the plain code, even without CoT, there is a notable performance drop in some cases (e.g., 61.2% for Claude Sonnet 3.5). However, when combining CFGs with CoT reasoning, the models show modest improvements in some cases, but the results remain suboptimal, especially in the case of InternVL2-26B, which only reaches 2.1% accuracy.

The real improvement is observed when applying our method, particularly when combined with Multimodal-CoT. This task is mainly about logical Our method, which integrates a reference mechanism during the first stage of Rationale Generation, shows substantial gains, especially for InternVL2-26B, where the accuracy rises to 6.3% when using our method alone and further increases to 10.7% when combined with Multimodal-CoT. This indicates that our approach significantly enhances the model's ability to reason about program repair, especially for models like InternVL2, which previously struggled in this task.

The Fault Localization task results demonstrate a consistent trend where models perform better across all settings compared to program repair. In the plain code without CoT setting, Claude Sonnet 3.5 achieves a high accuracy of 90.4%, with GPT-4o reaching 87.1%. Introducing CoT slightly improves performance for GPT-4o (89.5%) but shows minimal change for Claude Sonnet 3.5.

When CFGs are added, either with or without CoT, the results are somewhat mixed. While there is a performance dip in some cases (e.g., 86.1% for Claude Sonnet 3.5 with plain code and CFG without CoT), the models generally maintain high performance levels. However, when we apply Multimodal-CoT and combine it with our method, the improvements are more pronounced.

Our method alone achieves the highest accuracy for Claude Sonnet 3.5 and GPT-4o at 91.4% and 90.4%, respectively. When Multimodal-CoT is combined with our method, the performance reaches new heights: Claude Sonnet 3.5 achieves an accuracy of 92.8%, and GPT-4o reaches 91.9%. Notably, InternVL2-26B, which struggled in other settings, shows a dramatic improvement, rising from 41.0% (CoT with CFG) to 53.6% when our method is applied in combination with Multimodal-CoT. This confirms that the integration of CFGs with CoT reasoning and our reference mechanism significantly boosts fault localization performance.

6 QUALITATIVE ANALYSIS

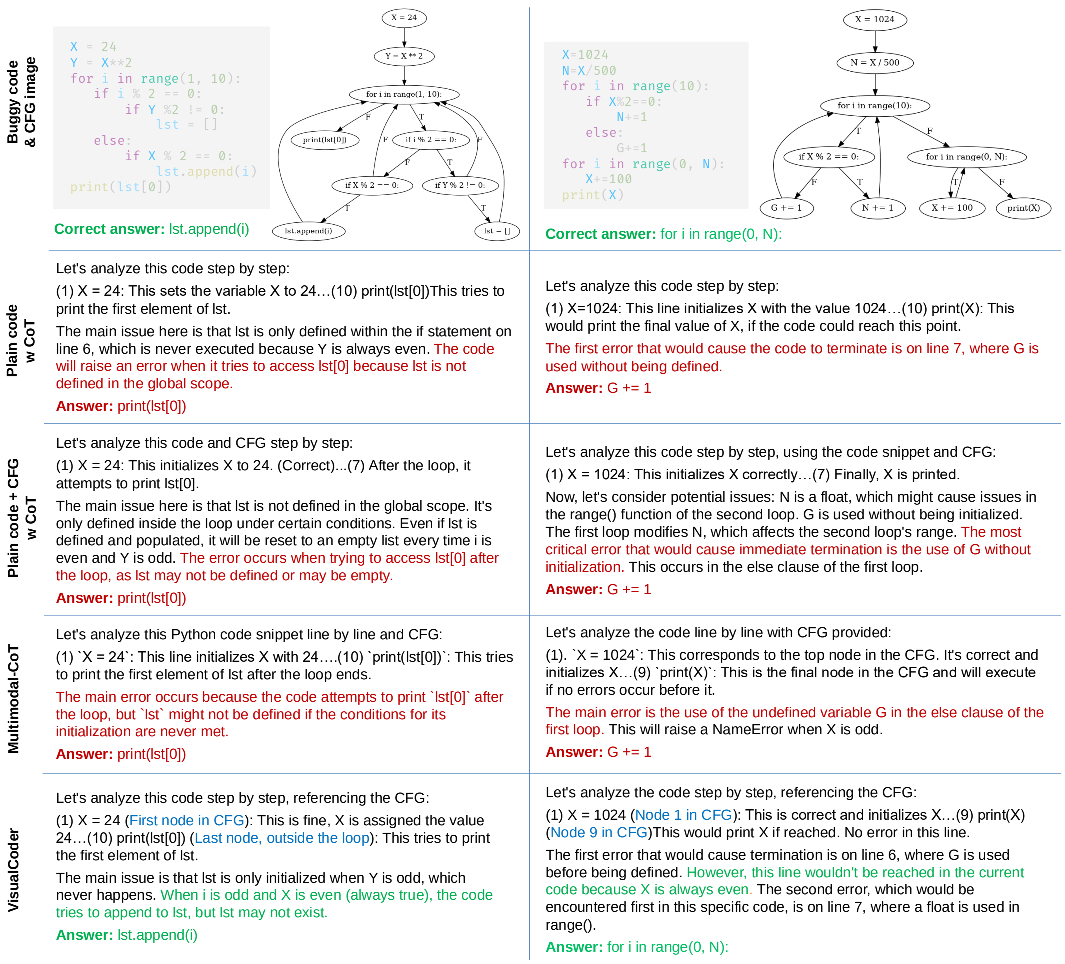

Figure 2: Qualitative comparison of reasoning outputs for buggy code using different prompt settings in Claude Sonet 3.5. Red text indicates where the reasoning fails, green text highlights correctly identified critical points, and blue text in VISUALCODER shows the referencing from the plain code to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

Figure 2 presents two examples of buggy code alongside their corresponding Control Flow Graphs (CFGs) and the reasoning 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). These examples are used to qualitatively compare various methods and highlight the effectiveness of our proposed method, VISUALCODER, which integrates CFG, Chain-of-Thought (CoT) reasoning, and a reference mechanism.

The first three rows of Figure 2 show the outputs of Claude Sonet 3.5 under the different prompt settings. In all these settings, the model fails to fully understand the complexity of the code. In the left example, which involves a use-before-initialization error, the model in these settings incorrectly identifies the issue as related to accessing `lst[0]`, highlighted in red, because it does not properly account for the control flow dependencies that affect when `lst` is initialized. Similarly, in the right example, which contains unreachable code, the model misinterprets the error, highlighting `G += 1` as the cause, but fails to recognize that the actual issue is the float value `N` being used in the `range` function. These failures highlight the limitations of reasoning based on plain code, even when aided by CFG or CoT individually. Without a deeper understanding of how the code executes dynamically, the model cannot pinpoint the true source of the errors.

In contrast, the final row shows the performance of VISUALCODER. In the example on the left side, VISUALCODER captures the critical error by analyzing the CFG and identifying that the node for `lst`'s initialization and the node for `lst.append(i)` do not connect. As a result, when the code tries to append to `lst`, the initialization never occurs in the current control flow, leading to a `NameError` due to `lst` being undefined. This critical point (captured by VISUALCODER) is highlighted in green. Other methods mistakenly assume that the list `lst` is reinitialized during each iteration of the `for` loop, causing them to incorrectly conclude that `lst[0]` raises an `IndexError` because the list is empty. In fact, the error arises because `lst` is never initialized before being used.

Additionally, VISUALCODER utilizes a reference mechanism, shown in blue in the output, which refers to the key CFG nodes during the reasoning process. This mechanism helps the model explicitly link the 'execution' steps to the corresponding control flow nodes, which is a major departure point from other methods lacking such explicit referencing.

In the example on the right side, VISUALCODER again demonstrates its advantage by leveraging the CFG to understand the non-linear control flow. While the previous methods struggled to detect that the float value `N` is used incorrectly in the `range` function, VISUALCODER's reference to the CFG allows the model to recognize the true cause of the error: the unreachable branch of the code. The CFG shows that the `else` block involving `G += 1` is never executed because `X` is always even, allowing the model to focus on the correct error related to the float value in the `range` function. As a result, VISUALCODER correctly identifies `for i in range(0, N)` as the solution.

These qualitative comparisons clearly demonstrate the advantage of VISUALCODER. The red turning points in previous methods indicate where the model's reasoning breaks down, whereas the green critical points in VISUALCODER 's output show how our method resolves the errors by aligning the code with its CFG during the reasoning process. By maintaining a structured alignment between code lines and their CFG nodes, our approach ensures that the model grasps the control flow, avoids mistakes, and accurately identifies both use-before-initialization and unreachable code errors.

7 CONCLUSION & FUTURE WORK

In conclusion, our work explores the potential of enhancing Large Language Models (LLMs) in understanding and reasoning about code execution by leveraging multimodal inputs, specifically integrating control flow graph (CFG) visualizations. While traditional LLMs excel in recognizing static code patterns, they struggle with dynamic program behaviors, especially those that require an understanding of execution context. Our proposed approach, VISUALCODER, introduces the Reference CoT prompting technique, which directly links lines of code with their corresponding CFG elements to improve reasoning about code execution. This method not only addresses limitations in existing CoT techniques by reducing cascading errors but also provides a more grounded and intuitive representation of the code's execution flow. Our preliminary results suggest that the inclusion of visual CFG representations enhances the model's ability to reason about code, and we believe that further refinement of this technique could significantly improve LLM performance in tasks involving complex program analysis.

Future work stemming from this research holds several promising directions. First, expanding VISUALCODER 's approach to diverse programming languages could help evaluate its scalability across different code structures and paradigms, including functional and declarative languages. Additionally, integrating real-time feedback from execution environments could enable LLMs to simulate dynamic program behaviors, such as runtime error detection or variable state tracking, which are currently challenging for these models. Optimizing multimodal prompts, such as refining Reference CoT prompting to better handle larger and more complex control flow graphs, could further improve performance, potentially through selective focus on critical execution paths using attention mechanisms.

Moreover, building interactive code debugging agents that leverage visualizations of control flow in real time could empower developers by providing automated debugging and repair suggestions. Exploring more complex graph representations, such as abstract syntax trees (ASTs) or data flow graphs (DFGs), could also deepen VISUALCODER 's multimodal reasoning capabilities. Lastly, incorporating human feedback into the reasoning process—creating human-in-the-loop systems—could allow VISUALCODER to learn dynamically from corrections, improving adaptability in practical coding scenarios.

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. 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, 2024. Curran Associates Inc. ISBN 9781713871088.

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

David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. Learning to execute programs with instruction pointer attention graph neural networks. *Advances in Neural Information Processing Systems*, 33:8626–8637, 2020.

Lionel C Briand, Yvan Labiche, and Xuetao Liu. Using machine learning to support debugging with tarantula. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*, pp. 137–146. IEEE, 2007.

Nghi DQ Bui, Yue Wang, and Steven Hoi. Detect-localize-repair: A unified framework for learning to debug with codet5. *arXiv preprint arXiv:2211.14875*, 2022.

Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems*, 34: 22196–22208, 2021.

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. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks, 2024. URL `https://arxiv.org/abs/2312.14238`.

Hridya Dhulipala, Aashish Yadavally, and Tien N. Nguyen. Planning to guide llm for code coverage prediction. In *2024 IEEE AI Foundation Models and Software Engineering*. IEEE, 2024.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16568–16621. PMLR, 21–27 Jul 2024. URL `https://proceedings.mlr.press/v235/gu24c.html`.

Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. *Advances in Neural Information Processing Systems*, 33:17685–17695, 2020.

Md. Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. Fixeval: Execution-based evaluation of program fixes for competitive programming problems. *ArXiv*, abs/2206.07796, 2022. URL `https://api.semanticscholar.org/CorpusID:249712458`.

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. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/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. Livecodebench: Holistic and contamination free evaluation of large language models for code. *CoRR*, abs/2403.07974, 2024. URL https://doi.org/10.48550/arXiv.2403.07974.

Cuong Chi Le, Hoang Nhat Phan, Huy Nhat Phan, Tien N. Nguyen, and Nghi D. Q. Bui. Learning to predict program execution by modeling dynamic dependency on code graphs, 2024. URL https://arxiv.org/abs/2408.02816.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. 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, 2023.

Xia Li and Lingming Zhang. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 169–180. ACM, 2019.

Yi Li, Shaohua Wang, and Tien N. Nguyen. Fault localization with code coverage representation learning. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE'21. IEEE, 2021.

Yi Li, Shaohua Wang, and Tien N. Nguyen. 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, pp. 659–671, New York, NY, USA, 2022a. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549137. URL https://doi.org/10.1145/3540250.3549137.

Yi Li, Shaohua Wang, and Tien N. Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE'22. ACM Press, 2022b.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 34892–34916. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/6dcf277ea32ce3288914faf369fe6de0-Paper-Conference.pdf.

Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning, 2024. URL https://arxiv.org/abs/2310.03744.

Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 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*, pp. 664–676, 2021.

Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 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, pp. 1169–1180, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510147. URL https://doi.org/10.1145/3510003.3510147.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. NExT: Teaching large language models to reason about code execution. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 37929–37956. PMLR, 21–27 Jul 2024a. URL https://proceedings.mlr.press/v235/ni24a.html.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*, 2024b.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

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. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/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. Learning transferable visual models from natural language supervision. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 8748–8763. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/radford21a.html.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

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. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

Kensen Shi, Joey Hong, Yinlin Deng, Pengcheng Yin, Manzil Zaheer, and Charles Sutton. Exedec: Execution decomposition for compositional generalization in neural program synthesis. *arXiv preprint arXiv:2307.13883*, 2023.

Eui Chul Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. *Advances in Neural Information Processing Systems*, 31, 2018.

Jeongju Sohn and Shin Yoo. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 273–283, 2017.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Yanbin Wei, Shuai Fu, Weisen Jiang, Zejian Zhang, Zhixiong Zeng, Qi Wu, James T. Kwok, and Yu Zhang. Gita: Graph to visual and textual integration for vision-language graph reasoning, 2024. URL https://arxiv.org/abs/2402.02130.

W Eric Wong and Yu Qi. BP neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597, 2009.

Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, pp. 191–200. IEEE, 2014.

He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–13, 2022.

Zhuo Zhang, Yan Lei, Qingping Tan, Xiaoguang Mao, Ping Zeng, and Xi Chang. Deep learning-based fault localization with contextual information. *Ieice Transactions on Information and Systems*, 100(12):3027–3031, 2017.

Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 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)*, pp. 445–455. IEEE, 2019.

Zhuosheng Zhang, Aston Zhang, Mu Li, Hai Zhao, George Karypis, and Alexander J. Smola. Multimodal chain-of-thought reasoning in language models. *Trans. Mach. Learn. Res.*, 2024, 2023. URL https://api.semanticscholar.org/CorpusID:256504063.

Wei Zheng, Desheng Hu, and Jing Wang. Fault localization analysis based on deep neural network. *Mathematical Problems in Engineering*, 2016, 2016. doi: https://doi.org/10.1155/2016/1820454.

Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. Minigpt-4: Enhancing vision-language understanding with advanced large language models, 2023. URL https://arxiv.org/abs/2304.10592.