A Matter of Representation: Towards Graph-Based Abstract Code Generation

Nyx Iskandar
University of California, Berkeley
nyx@berkeley.edu

Hisham Bedri
Ramen VR
hisham@ramenvr.com

Andy Tsen
Ramen VR
andy@ramenvr.com

Abstract

Most large language models (LLMs) today excel at generating raw, sequential code with minimal abstractions and custom structures. However, there has been little work on graph-based abstract code generation, where significant logic is encapsulated in predefined nodes and execution flow is determined by edges. This is relevant for visual programming languages, and in cases where raw source code is inaccessible to users and LLM training sets. In this work, we propose and evaluate JSON representations for graphs to enable high accuracy graph-based abstract code generation. We evaluate these representations on ScratchTest, a mini-benchmark based on our custom Python re-implementation of Scratch, which tests the LLM in code graph space. Our findings demonstrate that LLMs can indeed perform the aforementioned generation task in a single pass without relying on specialized or complex pipelines, given the correct graph representations. We also show that different representations induce significantly different accuracies, highlighting the instrumental role of representations in this generation task. All in all, this work establishes the first steps towards representation learning for graph-based abstract code generation.

1 Introduction

Large language models (LLMs) [5] have increased in capability significantly over recent years in not only natural language tasks, but also the more syntactically rigorous task of code generation [6, 2]. These models excel not only in code completion [20, 43, 47], but also in instruction-following to generate code in natural-language-to-code (NL2Code) tasks [2, 3, 53]. Indeed, many LLMs today, be they open-weight or otherwise, feature a code fine-tuning of them, such as Qwen3-Coder [49, 38], or have their coding capabilities highlighted by model developers and evaluators through blog highlights and benchmark performance reporting, such as those for GPT-5 [33], Claude Opus 4.1 [31], and DeepSeek-R1 [9]. This is in no small part due to the abundance of benchmarks that specifically evaluate the code generation, understanding, and/or problem-solving capabilities of these models, such as SWE-bench Verified [34] and Weapons of Mass Destruction Proxy - Cyber [25]. The success of agentic tools in Cursor [8], Codex [32], and ClaudeCode [1] is a testament to how capable these models are in coding, and how rapidly these capabilities have developed in the past two years of their massive popularity and widespread adoption.

However, these coding tasks are restricted to the domain of raw, sequential code, which is the style of code that software engineers are most familiar with and thus have produced an abundance of

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Deep Learning for Code.

data of that is available online for training: neatly arranged line-by-line in a file as a string in an intuitively linear fashion. Of course, code need not necessarily be written in this form; there are many graph-based languages like Scratch [29], Unreal Engine Blueprints [13], and n8n software [30] that are widely used by beginner programmers for learning how to think programmatically, game developers and designers for rapid prototyping, and software engineers for implementing agentic AI workflows, respectively. Despite its prevalence, the domain of graph-based abstract code has thus far been sidelined, which is evident in the lack of benchmarks related to it. There is also no unified representation of these graph-based code fragments. As such, LLMs today lack the ability to reliably generate logically sound graph-based code that accomplishes the task or goal detailed in the user query, or even graph-based code that compiles.

Furthermore, research on graph-based abstract code generation has greater implications in supporting LLMs in cases where there are minimal code implementation examples available, such as in newer libraries. This is because inherent in this flavor of code is the fact that implementation details are abstracted away from the user and thus the LLM. While it can be argued that any programming language abstracts implementation details away, the abundance of available data of such abstractions in, for instance, GitHub repositories that is exposed to these LLMs during pre-training and fine-tuning eliminates that barrier [24, 27]. However, for lesser-known or newer libraries or frameworks, this is hardly the case. In fact, we argue that these libraries or frameworks face the same challenge as graph-based coding languages: abstraction. Furthermore, the existence of Scratch, Blueprints, and other similar languages shows that raw code can be represented as graphs, and thus the code underlying these lesser-known or newer libraries or frameworks can also be represented as graphs. While in-context learning [11] seems like a potential solution, it is more suited for familiarizing LLMs with new syntax, not logic flows; graph-based languages introduce interesting non-linear relationships between code fragments in the form of nodes, posing a challenge in-context learning is ill-equipped to solve. As such, advancing graph-based code generation also advances the general capability of LLMs to generate code using the libraries or frameworks that are disproportionately underrepresented in their training data.

As such, **our contributions** are as follows:

- We propose a simple JSON representation of **nodes** that enables current LLMs to generate the most syntactically and logically accurate code graphs when given a list of those representations as **reference** compared to other JSON representations.
- We propose a JSON representation of **code graphs** that further enables current LLMs to generate the most syntactically and logically accurate code graphs when **outputting** that representation compared to other JSON representations.
- We propose a new **mini-benchmark** based on a Python re-implementation of Scratch to evaluate graph-based abstract code generation capabilities of LLMs.

Our work focuses on improving the performance of a one-agent LLM framework without fine-tuning [28, 10], in-context learning [11], or other additions to the base model. In other words, we focus on showing how integrating the correct node and graph representations into the LLM's prompt increases the LLM's generation accuracy. Code is available in the GitHub repository.

2 Related Work

2.1 Graph Understanding and Generation

Past work on graph generation includes GraphRNN [52] and GraphGAN [42]. Though leveraging distinct approaches, both are suited for generating generic nodes and edges by learning a particular graph distribution represented in the ground truth set. However, these approaches do not translate well for our focus on graph-based code generation; these past approaches focus on instantiating graph models based on characteristics or properties of a set of observed graphs, while our focus is on generating graphs which fulfill a functional objective as we work with rich nodes that encode information and rich edges that encode logic and data flow. The same limitation exists in Fatemi et al. [15], Wang et al. [41], Ye et al. [50], and Zhang [54] as the foci of these papers are on graphs as mathematical objects instead of representations of logic, and on evaluating LLM graph understanding (e.g. whether two nodes are connected to each other) instead of on completion based on the encoded logic within each graph.

More recently, there has been work on Graph Foundation Models (GFMs) [26], which are built on either graph neural networks (GNNs) [39] or LLMs. While GFMs are designed to handle non-Euclidean data for graphs, we argue that it is unnecessary, at least for our focus, as graphs can be represented as plain string or stringified JSON, both being Euclidean. As to the backbones of GFMs, it has been shown that GNNs do not scale well with current hardware [44, 40, 48], while the representation used by the proposed LLM-based GFMs relies too heavily on brittle natural language rather than structured representations like JSON. The limitations of GFMs also apply to the work by Jin et al. [22] as they leverage GNNs and/or natural language translations of graphs. Instead, we show that with a correct structured representation of code graphs, vanilla LLMs alone suffice in generating syntactically and logically sound graphs.

2.2 LLM Coding Agents

The other side of our work is code generation, which today's LLMs are extremely capable of, as aforementioned. Recent work in reinforcement learning on chain-of-thought has increased LLM capabilities in verifiable domains, including mathematics and coding [45, 51, 9]. Prior to that, methods like fill-in-the-middle (FIM) [4] have proved effective in training code models like CodeGemma [7], and data processing methods in the form of file-level and repo-level pre-training have also proven effective in increasing the accuracy of code models like Qwen2.5-Coder [21]. LLM code capabilities have also been enhanced by constrained decoding for structured output generation [19, 12, 46, 16], as much of software engineering relies on reliable structures for reliable data flows.

However, it is evident that LLMs, be they foundation models or fine-tunes or agents, show greater capability in generating certain languages, libraries, frameworks, etc. than others [37, 2, 36]. This discrepancy is due to the different levels of availability of data online for different domains. We thus hypothesize that having LLMs generate graph-based code in an ubiquitous format like JSON will increase their accuracy.

3 ScratchTest

Before discussing our proposed representations, it is valuable to first explain the mini-benchmark we used to run the evaluations, and to ground what we mean by graph-based abstract code in real examples. This section thus discusses ScratchTest, a set of test prompts and correct behaviors that assesses the accuracy of an LLM in implementing nodes of a re-implementation of Scratch [29] written in Python.

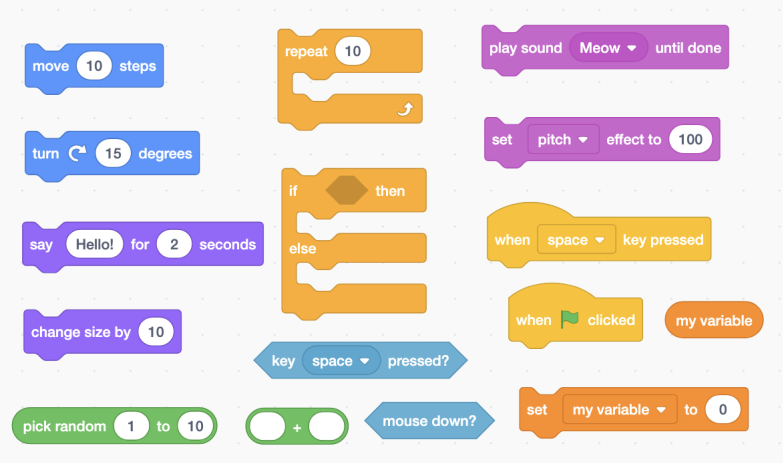

Figure 1: Two Scratch blocks from each built-in block type in the Scratch online editor.

Scratch is a beginner-friendly, high-level programming language that is based on the concept of blocks, which can be thought of as nodes in a graph (we thus use "block" and "node" interchangeably in this paper). Programs in Scratch primarily manipulate a sprite using built-in blocks of various

types: motion, looks, sound, event, control, sensing, operator, and variable [29]. A subset of these blocks is found in Figure 1. Scratch blocks abstract away many implementation details, hence are used by users through connecting them together (as indicated by the indents and outdents of each block) and modifying the default parameters (e.g. 10 in "move 10 steps"), rather than writing lines of code that are more akin to Python or C++ to manipulate primitives and keywords. ScratchTest includes Python re-implementations of 53 out of the 107 built-in Scratch blocks, chosen by which blocks can be feasibly implemented in a command line interface (CLI). The full list of blocks that are included in ScratchTest is found in Appendix A.

To preserve simplicity in implementation and output analysis, and given the constraints of a CLI, the blocks in ScratchTest do not directly manipulate a sprite nor any visual element. Instead, we rely on each block returning a behavior log (e.g. a "move n steps" block would output "moved n steps" when called). We also persist a dictionary of sprite attributes, such as its x-position, y-position, and direction (i.e. rotation). Furthermore, each complete and valid graph as the final output of our proposed methodology outputs a Python file that directly interfaces with the classes and functions of the Scratch re-implementation. As such, assessing both the behavior logs outputted by the graphs generated by LLMs and the output Python file is how we evaluate the accuracy of these LLMs.

Note that our mini-benchmark currently does not have a canonical set of ground-truth behavior logs nor Python files due to the high variability in approaches to satisfy the functionality requested in a prompt. While we can curate more straightforward prompts to circumvent this issue, this risks asking the LLM to implement functionalities which are too elementary, preventing the tests from being sensitive enough to evaluate the proposed representations and their ablations or alternatives. Hence, to ensure reliability and accuracy in the evaluation, we opt to manually evaluate both the behavior logs and Python files of sufficiently complex prompts. A correct implementation is one which outputs a reasonable behavior log and a logically accurate Python file based on the requested functionality in the prompt.

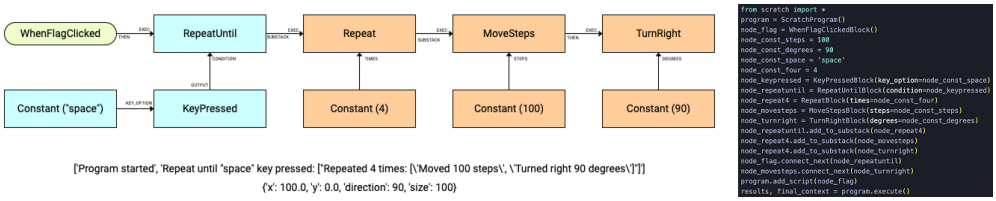

Figure 2: Correct output graph, behavior log, sprite state, and corresponding Pythonic Scratch for prompt "When the green flag is clicked, continuously move in a square pattern until the user presses the space key."

ScratchTest includes 20 unique prompts that describe a certain functionality; for example, "When the green flag is clicked, continuously move in a square pattern until the user presses the space key." and "When the 's' key is pressed, say a secret password made of two random letters and three random numbers." Each prompt's requested functionality can be accomplished by implementing at least one permutation of the nodes available in ScratchTest. Figure 2 shows a sample correct graph of Scratch nodes (with connected ports specified) with the corresponding behavior log, sprite state, and Python file for the prompt specified in the caption. Each prompt is run five separate times; the resulting behavior log and Python file corresponding to each prompt are thus also evaluated five separate times.

4 Representations

This section explains our proposed JSON representations to ensure the highest accuracy in graph generation for a vanilla one-agent LLM framework compared to other JSON representations. These include the node representation we provide the LLM as reference, as well as the graph representation we steer the LLM to output. We include a **reference node representation** to provide the LLM with context on what nodes (i.e. functions) it can use, which is crucial as it is not fine-tuned on the particular graph-based library or framework. This is distinct from in-context learning; performing in-context learning would be providing the LLM with correct prompt-graph pairs. We also include an **output graph representation** to guide the LLM to generate a standardized structured output format

for the post-processing step that translates the generated graph into its equivalent in the Pythonic Scratch implementation.

For our main experiments, we use OpenAI's gpt-oss-120b [35] on Groq [18] as is. The exact model configuration is found in Appendix B. We also ran experiments on other LLMs, namely qwen3-32b [49], deepseek-r1-distill-llama-70b [9], and llama-3.3-70b-versatile [17] on Groq; their respective configurations are also found in Appendix B. The system prompt can be found in Appendix C.

It is important to note that the LLM is exposed only to the representations detailed in this section. Notably, the Python implementation of Scratch is not at all provided to the LLM, which is to say that the LLM is not aware of what logic underlies the JSON representations it is actually provided with. This simulates real-world environments where the user (or LLM) is only given the function names without the underlying logic that has been abstracted away. As such, the LLM is unable to use or infer from the actual implementation to assist its generation, which is desired.

4.1 Reference Node Representation

We propose a node representation with the following format to be provided to the LLM as a reference list of canonical nodes:

```
[NODENAME]: {
    inPorts: {id: string, type: string}[],
    fields: {id: string, type: string}[],
    outPorts: {id: string, type: string}[]
}
```

More specifically:

- **NODENAME**: The name of the node. Each name corresponds to that of the actual block in Scratch (e.g. WhenFlagClicked corresponds to the "when green flag clicked" block in Scratch).
- **inPorts**: The input ports of the node, which are ports that correspond to an incoming edge. These include ports that represent parameters, and EXEC ports which are connected to an edge that is also connected to the node that immediately precedes the current node in execution (the visual indent of a Scratch block is EXEC).
- **fields**: Parameters of the node that must be chosen from a predefined list. For instance, the MathFunction node has a field called OPERATOR which takes in only the following values: abs, floor, ceiling, sqrt, sin, cos, tan, asin, acos, atan, ln, log, e^, 10^. These are virtually indistinguishable from **inPorts** during generation.
- **outPorts**: The output ports of the node, which are ports that correspond to an outgoing edge. These include ports that represent return values, THEN ports which are connected to an edge that is also connected to the node that immediately succeeds the current node in execution (the visual outdent of a Scratch block is THEN), and SUBSTACK ports which are connected to edges that are also connected to the nodes that exist within the loop/control block (only relevant for control nodes).
- **id**: The unique identifier of each port (e.g. EXEC).
- **type**: The type that is to be connected to this port (e.g. number, string, Node).

We find that the addition of **type** to each port increases the accuracy of the graph generation, as we observe that the LLM may hallucinate node-node or constant-node connections of incompatible types (e.g. connecting a port of type number to a port of type string). Interestingly, we find that adding natural language descriptions to both the nodes and the ports does not increase the accuracy of the LLM in constructing the graph by any significant margin. We hypothesize that this is because the node and port names are sufficiently descriptive for the LLM such that the addition of these descriptions only increases the token count for a comparable accuracy. We will show in Section 5 the quantitative effects of these additions on the accuracy of the generation through ablations.

The inclusion of **id** for each port helps in the post-processing step that translates the output graph generated by the LLM to the equivalent Pythonic Scratch implementation. This is because each port can then be uniquely identified and thus be well-assembled after the translation. Port IDs generally

correspond to parameter names (e.g. STEPS for the number of steps to move in a MoveSteps nodes), but we also have the specially defined ports THEN and EXEC to facilitate the reconstruction of execution flow such that the THEN port of some node is connected to the EXEC port of the next node in the main graph. Another specially defined port is SUBSTACK, which is relevant only for control nodes (e.g. Repeat, IfElse); the SUBSTACK port of a control node is connected to the EXEC port of the nodes to be repeated, looped over, branched into, etc. Some nodes like IfElse have multiple SUBSTACK ports to handle logic branches (i.e. one for the case where the condition is true, and another for the case where the condition is false).

4.2 Output Graph Representation

We propose a graph representation with the following format to be outputted by the LLM:

```
{
    nodes: {
        [key: string]: {
            name: string,
            value: any|null
        }
    },
    edges: {
        outNodeID: string,
        outPortID: string,
        inNodeID: string,
        inPortID: string
    }[]
}
```

More specifically:

- **nodes**: A map of **nodeIDs** to node data. nodeIDs are unique string identifiers such that no two nodes in one graph can have the same ID. The node data dictionary has the node name and the node value. The value field is always $null$ except for when it is for a constant, in which case its value is the value it is supposed to hold (e.g. 10, "Hello", True). Note that unique nodeIDs are necessary as nodes of the same name may be instantiated more than once in one graph (e.g. the graph may have multiple Add nodes).
- **edges**: An array of dictionaries representing edges. **outNodeID** belongs to the node whose outPort is used for this edge (**outPortID** is the ID of said port). **inNodeID** belongs to the node whose inPort is used for this edge (**inPortID** is the ID of said port). A **field** of a node can be considered an inPort for the purposes of an edge.

As will be shown in our experiments detailed in the subsequent section, this particular output graph representation achieves the highest generation accuracy compared to an alternative representation which is equally as intuitive. We hypothesize that the separation of concerns between generating the nodes and the edges reduces the margin of error for the LLM, and the generation itself is more linear in the sense that connectable nodes are defined first and edges connecting those nodes are defined later. This is in contrast to a representation where nodes and edges are defined within the same atomic JSON object, where nodes that are defined later are referenced earlier, and one edge must be defined twice (first from the perspective of the outNode via the outPin, and second from the perspective of the inNode via the inPin), resulting in duplicative and more error-prone effort.

The proposed output graph representation will then undergo a post-processing step to translate the graph into the Python implementation of Scratch. The reconstruction first orders the nodes and edges using topological sort, as the graph must be a directed acyclic graph for it to have a valid execution sequence; the existence of for loops and other similar logic is handled by the respective nodes, which prevents instances of actual loops in the graph representation. After the sort, the post-processing step constructs a Python file which contains the Pythonic Scratch methods obtained after translating the nodes and edges in the graph representation. The translation process includes instantiating Scratch block objects bound to variables named the respective nodeIDs, defining constants, passing in nodes or constants as arguments to the block objects whenever necessary, adding nodes to the respective substacks of control nodes, and connecting the nodes sequentially based on the THEN and EXEC ports.

5 Evaluation

We conduct ablation studies to evaluate the impact of including and omitting certain components from both the reference node representation and the output graph representation. As a reminder, a correct implementation is one that outputs a reasonable behavior log and a logically accurate Python file; an incorrect implementation is either logically flawed or results in an error during the post-processing step or during the execution of the constructed Pythonic Scratch implementation. The results shown in Sections 5.1 and 5.2 are for gpt-oss-120b; results for other models can be found in Appendix G.

5.1 Reference Node Representation

Breaking down the ablations, we have the following three reference node representations:

- **No Types**: Our proposed representation, but without types for both nodes and ports. This representation is essentially the minimal baseline representation.
- **Extra Description**: Our proposed representation, but with description fields for both nodes and ports (e.g. the description for the MoveSteps node is "Move sprite forward by specified number of steps", and that for the EXEC port is "Previous block that triggers this block").
- **Proposed**: Our actual proposed representation described in Section 4.1.

Evaluating the ablations on 20 ScratchTest prompts five independent times each, we obtain the results shown in Table 1. For a granular per-prompt breakdown of the results, please refer to Appendix E.

Table 1: ScratchTest results for reference node representation ablations.

Name	Run 1	Run 2	Run 3	Run 4	Run 5
No Types	12/20	14/20	12/20	15/20	12/20
Extra Description	14/20	**16**/20	14/20	**17**/20	13/20
Proposed	**16**/20	14/20	**15**/20	16/20	**14**/20

We conclude that **Proposed** has the highest mean accuracy (albeit only slightly more accurate than **Extra Description**) and is the most consistent across runs. The results summary is shown in Table 2.

Table 2: Summary results for reference node representation ablations.

Name	Mean	Standard Deviation
No Types	0.65	0.071
Extra Description	0.74	0.082
Proposed	**0.75**	**0.050**

Calculating the two-sided p-values with Welch's t-test, and setting the significance level $\alpha = 0.05$, the difference in accuracy between **Proposed** and **No Types** is significant as $p = 0.036 \leq \alpha$; however, that between **Proposed** and **Extra Description** is not significant as $p = 0.82 > \alpha$. This shows that while adding types increases accuracy by a statistically significant amount, adding descriptions does not; descriptions are unnecessary, cost more in terms of tokens, and do not scale as robustly due to the extra tokens per node. We posit that descriptions must be well-thought out and sufficiently verbose to result in a significant increase in accuracy (but at the cost of tokens) as otherwise accuracy may instead decrease due to misleading or vague descriptions; a more scalable alternative is to name the nodes and ports sufficiently descriptively, and if necessary create a mapping between the more descriptive names to the actual names provided by the abstracted library or package. The exact descriptions used are all included in the GitHub repository, where one can see the difference in token counts between **Proposed** and **Extra Description**.

5.2 Output Graph Representation

Breaking down the ablations, we have **Proposed**, which is our actual proposed representation described in Section 4.2 (this is identical to **Proposed** in Table 1), and **Alternative**, which is further described below:

```
{
    [key: string]: {
        nodeName: string,
        value: any|null,
        edges: {
            portID: string,
            otherNodeID: string
        }[]
    }
}
```

More specifically (refer to Section 4.2 for more explanation of the respective attributes):

- **key**: The keys of the JSON object are **nodeIDs**.
- **nodeName**: The name of the node.
- **value**: The value of the node.
- **edges**: An array of dictionaries representing edges. **portID** is the ID of the port on this current node that is connected to this edge. **otherNodeID** is the ID of the other node that is connected to this edge (i.e. is paired with this current node by this edge).

Given the different output graph representation, the system prompt for **Alternative** is also different. It is outlined in Appendix D.

Evaluating the ablations on 20 ScratchTest prompts five independent times each, we obtain the results shown in Table 3. For a granular per-prompt breakdown of the results, please refer to Appendix F.

Table 3: ScratchTest results for output graph representation ablations.

Name	Run 1	Run 2	Run 3	Run 4	Run 5
Alternative	10/20	9/20	10/20	11/20	9/20
Proposed	**16**/20	**14**/20	**15**/20	**16**/20	**14**/20

The mean accuracy of **Alternative** is 0.49 and its standard deviation is 0.042; **Proposed** has a significantly higher mean accuracy and a slightly lower consistency (refer to Table 2). Calculating the two-sided p-value with Welch's t-test, and setting the significance level $\alpha = 0.05$, the difference in accuracy between **Proposed** and **Alternative** is significant as $p = 0.000024 \leq \alpha$. With 95% confidence, **Proposed** improves accuracy by 23–29$\%$ compared to **Alternative**.

Common errors faced by the alternative representation include connecting ports of the same direction together (i.e. connecting an outPort with another outPort, or an inPort with another inPort) and forgetting to define the ports for the two nodes that are connected by the particular edge (recall how **Alternative** requires one edge to be defined twice). These are fundamental and thus critical errors. Hence, we conclude that our proposed output graph representation indeed enables the LLM to generate more accurate graphs.

6 Limitations

Our proposed representations do not enable the LLM to achieve 100% accuracy. We primarily observe these two common errors: failing to create a valid directed acyclic graph, and referencing an undeclared variable. It is to be noted that this work establishes the first steps towards graph-based abstract code generation, and we encourage and will ourselves embark on further research to find more effective representations and/or tokenization and encoding methods to increase the accuracy of the generated code graphs. Furthermore, we intentionally limit this work to the vanilla

one-agent framework, recognizing that fine-tuned models as well as multi-agent frameworks that include planning and error-correction will likely boost the accuracy, especially for more complex long-horizon tasks, prompts which are exceedingly high-level (e.g. create a platformer game using only Scratch blocks), or frameworks or libraries with hundreds or thousands of nodes. Our goal in this paper is to propose representations that outperform other similar representations to act as a baseline for further refinement to the model or pipeline architecture.

7 Conclusion

It has been observed that the code generation capability of LLMs is a research focus prioritized by leading labs and companies [6, 2, 20, 1, 38], most advancing these capabilities through post-training methods like chain-of-thought reasoning for foundation models [9], or relying on multi-turn methods leveraging planning and/or error-correction [14, 23] for downstream applications. However, the current focus is heavily on raw, sequential code with minimal abstractions, while minimal work has been done on graph-based abstract code. We argue that the latter is also an important direction in code generation, as not only do many software tools today leverage visual graph-based programming languages [29, 13, 30], but also this flavor of code essentially represents niche or specialized libraries or frameworks that abstract away implementation details from downstream users and thus from training data for LLMs [37, 2, 36]. Our work acts as an early step towards graph-based abstract code generation, and we posit that finding the optimal representations of these graphs allows even current LLMs that are not specially trained to understand and generate non-sequential graphs to achieve a relatively high accuracy in this task.

A key idea of our proposed representations, both for reference nodes and output graphs, is that current LLMs are already frequently exposed to and thus well-capable of generating JSON or structured outputs. The question we aim to answer is thus "what JSON representation will yield the highest accuracy consistently for graph-based abstract code generation using today's LLMs?" Through evaluating ablations of our proposed representations on our new mini-benchmark ScratchTest, we conclude that our proposed representations are indeed the JSON representations that would yield the highest accuracy in the most consistent and resource-conservative manner, particularly for the one-agent framework without fine-tuning, in-context learning, multi-turn interactions, etc. Adding the aforementioned techniques and using frontier models will likely further increase the generation accuracy; what is valuable from our research for this future work is the baseline accuracy guarantees as demonstrated in the ScratchTest results. Given that our work proposes structures for reference nodes and output graphs, it can easily be adapted to larger and more complex graph-based domains beyond Scratch, as long as the context window of the LLM can accommodate the representations.

Furthermore, in attempting to answer the aforementioned question, we observe a behavior that has implications beyond graph-based abstract code generation. We learn that the capabilities of an LLM depend not only on its architecture and training data, but also the representation of information that it is provided within its context window during inference, and the representation of the output it is steered to generate. In other words, the same LLM can generate outputs of varying qualities given the same prompt structure but different representations. While we do not have definitive conclusions nor hypotheses as to why this occurs for the general case beyond the particular ablations evaluated in this work, it is nevertheless an important idea to bear in mind when working with these and similar models.

All in all, we hope that our work will spur more research into graph-based abstract code generation, either through searching for even better representations (that may not necessarily be JSON-based) or through innovating new algorithms and generative models that specialize in code graph space. We believe that these are promising directions for future work in deep learning for code.

Acknowledging societal impacts: We acknowledge that there are broader societal impacts of our work. Improving graph-based abstract code generation will increase the productivity of users of the tools which rely on graph-based code, yet it may also cause job loss in particular fields where automation becomes predominant. It is our hope that the tools our work inspires will be designed to assist and empower human users, not replace them.

Acknowledgments and Disclosure of Funding

The first author conducted this research during her internship with Ramen VR. All of the authors' sincerest thanks to our dedicated and talented colleagues for their trust and support. All funding for this work is received from Ramen VR.

References

[1] Anthropic (2025). Claude code: Deep coding at terminal velocity | anthropic.

[2] Athiwaratkun, B., Gouda, S. K., Wang, Z., Li, X., Tian, Y., Tan, M., Ahmad, W. U., Wang, S., Sun, Q., Shang, M., Gonugondla, S. K., Ding, H., Kumar, V., Fulton, N., Farahani, A., Jain, S., Giaquinto, R., Qian, H., Ramanathan, M. K., Nallapati, R., Ray, B., Bhatia, P., Sengupta, S., Roth, D., and Xiang, B. (2023). Multi-lingual evaluation of code generation models.

[3] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. (2021). Program synthesis with large language models.

[4] Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J., and Chen, M. (2022). Efficient training of language models to fill in the middle.

[5] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners.

[6] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. (2021). Evaluating large language models trained on code.

[7] CodeGemma, Zhao, H., Hui, J., Howland, J., Nguyen, N., Zuo, S., Hu, A., Choquette-Choo, C. A., Shen, J., Kelley, J., Bansal, K., Vilnis, L., Wirth, M., Michel, P., Choy, P., Joshi, P., Kumar, R., Hashmi, S., Agrawal, S., Gong, Z., Fine, J., Warkentin, T., Hartman, A. J., Ni, B., Korevec, K., Schaefer, K., and Huffman, S. (2024). Codegemma: Open code models based on gemma.

[8] Cursor (2025). Cursor - the ai code editor.

[9] DeepSeek-AI (2025). Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

[10] Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. (2023). Qlora: Efficient finetuning of quantized llms.

[11] Dong, Q., Li, L., Dai, D., Zheng, C., Ma, J., Li, R., Xia, H., Xu, J., Wu, Z., Chang, B., Sun, X., Li, L., and Sui, Z. (2024). A survey on in-context learning. In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N., editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 1107–1128, Miami, Florida, USA. Association for Computational Linguistics.

[12] Dong, Y., Ruan, C. F., Cai, Y., Lai, R., Xu, Z., Zhao, Y., and Chen, T. (2025). Xgrammar: Flexible and efficient structured generation engine for large language models.

[13] EpicGames (2025). Blueprints visual scripting.

[14] Erdogan, L. E., Lee, N., Kim, S., Moon, S., Furuta, H., Anumanchipalli, G., Keutzer, K., and Gholami, A. (2025). Plan-and-act: Improving planning of agents for long-horizon tasks.

[15] Fatemi, B., Halcrow, J., and Perozzi, B. (2023). Talk like a graph: Encoding graphs for large language models.

[16] ggml (2023). Ggml-org/llama.cpp: Llm inference in c/c++.

[17] Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Wyatt, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Guzmán, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Thattai, G., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I., Misra, I., Evtimov, I., Zhang, J., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Prasad, K., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Lakhotia, K., Rantala-Yeary, L., van der Maaten, L., Chen, L., Tan, L., Jenkins, L., Martin, L., Madaan, L., Malo, L., Blecher, L., Landzaat, L., de Oliveira, L., Muzzi, M., Pasupuleti, M., Singh, M., Paluri, M., Kardas, M., Tsimpoukelli, M., Oldham, M., Rita, M., Pavlova, M., Kambadur, M., Lewis, M., Si, M., Singh, M. K., Hassan, M., Goyal, N., Torabi, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Zhang, N., Duchenne, O., Çelebi, O., Alrassy, P., Zhang, P., Li, P., Vasic, P., Weng, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., Xu, P., He, Q., Dong, Q., Srinivasan, R., Ganapathy, R., Calderer, R., Cabral, R. S., Stojnic, R., Raileanu, R., Maheswari, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Hosseini, S., Chennabasappa, S., Singh, S., Bell, S., Kim, S. S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Shen, S., Wan, S., Bhosale, S., Zhang, S., Vandenhende, S., Batra, S., Whitman, S., Sootla, S., Collot, S., Gururangan, S., Borodinsky, S., Herman, T., Fowler, T., Sheasha, T., Georgiou, T., Scialom, T., Speckbacher, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Gonguet, V., Do, V., Vogeti, V., Albiero, V., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X., Wang, X., Tan, X. E., Xia, X., Xie, X., Jia, X., Wang, X., Goldschlag, Y., Gaur, Y., Babaei, Y., Wen, Y., Song, Y., Zhang, Y., Li, Y., Mao, Y., Coudert, Z. D., Yan, Z., Chen, Z., Papakipos, Z., Singh, A., Srivastava, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Teo, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Dong, A., Franco, A., Goyal, A., Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., Paola, B. D., Paranjape, B., Liu, B., Wu, B., Ni, B., Hancock, B., Wasti, B., Spence, B., Stojkovic, B., Gamido, B., Montalvo, B., Parker, C., Burton, C., Mejia, C., Liu, C., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Gao, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Adkins, D., Xu, D., Testuggine, D., David, D., Parikh, D., Liskovich, D., Foss, D., Wang, D., Le, D., Holland, D., Dowling, E., Jamil, E., Montgomery, E., Presani, E., Hahn, E., Wood, E., Le, E.-T., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Kokkinos, F., Ozgenel, F., Caggioni, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Inan, H., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Zhan, H., Damlaj, I., Molybog, I., Tufanov, I., Leontiadis, I., Veliche, I.-E., Gat, I., Weissman, J., Geboski, J., Kohli, J., Lam, J., Asher, J., Gaya, J.-B., Marcus, J., Tang, J., Chan, J., Zhen, J., Reizenstein, J., Teboul, J., Zhong, J., Jin, J., Yang, J., Cummings, J., Carvill, J., Shepard, J., McPhie, J., Torres, J., Ginsburg, J., Wang, J., Wu, K., U, K. H., Saxena, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Jagadeesh, K., Huang, K., Chawla, K., Huang, K., Chen, L., Garg, L., A, L., Silva, L., Bell, L., Zhang, L., Guo, L., Yu, L., Moshkovich, L., Wehrstedt, L., Khabsa, M., Avalani, M., Bhatt, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, M., Liu, M., Seltzer, M. L., Valko, M., Restrepo, M., Patel, M., Vyatskov, M., Samvelyan, M., Clark, M., Macey, M., Wang, M., Hermoso, M. J., Metanat, M., Rastegari, M., Bansal, M., Santhanam, N., Parks, N., White, N., Bawa, N., Singhal, N., Egebo, N., Usunier, N., Mehta, N., Laptev, N. P., Dong, N., Cheng, N., Chernoguz, O., Hart, O., Salpekar, O., Kalinli, O.,

Kent, P., Parekh, P., Saab, P., Balaji, P., Rittner, P., Bontrager, P., Roux, P., Dollar, P., Zvyagina, P., Ratanchandani, P., Yuvraj, P., Liang, Q., Alao, R., Rodriguez, R., Ayub, R., Murthy, R., Nayani, R., Mitra, R., Parthasarathy, R., Li, R., Hogan, R., Battey, R., Wang, R., Howes, R., Rinott, R., Mehta, S., Siby, S., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Mahajan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., Patil, S., Shankar, S., Zhang, S., Zhang, S., Wang, S., Agarwal, S., Sajuyigbe, S., Chintala, S., Max, S., Chen, S., Kehoe, S., Satterfield, S., Govindaprasad, S., Gupta, S., Deng, S., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Koehler, T., Robinson, T., Li, T., Zhang, T., Matthews, T., Chou, T., Shaked, T., Vontimitta, V., Ajayi, V., Montanez, V., Mohan, V., Kumar, V. S., Mangla, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wu, X., Wang, X., Wu, X., Gao, X., Kleinman, Y., Chen, Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu, Wang, Zhao, Y., Hao, Y., Qian, Y., Li, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., Zhao, Z., and Ma, Z. (2024). The llama 3 herd of models.

[18] Groq (2025). Groq is fast inference for ai builders.

[19] Guidance-AI (2024). Guidance-ai/guidance: A guidance language for controlling large language models.

[20] Guo, D., Xu, C., Duan, N., Yin, J., and McAuley, J. (2023). Longcoder: A long-range pre-trained language model for code completion.

[21] Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., Dang, K., Fan, Y., Zhang, Y., Yang, A., Men, R., Huang, F., Zheng, B., Miao, Y., Quan, S., Feng, Y., Ren, X., Ren, X., Zhou, J., and Lin, J. (2024). Qwen2.5-coder technical report.

[22] Jin, B., Liu, G., Han, C., Jiang, M., Ji, H., and Han, J. (2024). Large language models on graphs: A comprehensive survey.

[23] Kim, S., Moon, S., Tabrizi, R., Lee, N., Mahoney, M. W., Keutzer, K., and Gholami, A. (2024). An llm compiler for parallel function calling.

[24] Kocetkov, D., Li, R., Allal, L. B., Li, J., Mou, C., Ferrandis, C. M., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., Bahdanau, D., von Werra, L., and de Vries, H. (2022). The stack: 3 tb of permissively licensed source code.

[25] Li, N., Pan, A., Gopal, A., Yue, S., Berrios, D., Gatti, A., Li, J. D., Dombrowski, A.-K., Goel, S., Phan, L., Mukobi, G., Helm-Burger, N., Lababidi, R., Justen, L., Liu, A. B., Chen, M., Barrass, I., Zhang, O., Zhu, X., Tamirisa, R., Bharathi, B., Khoja, A., Zhao, Z., Herbert-Voss, A., Breuer, C. B., Marks, S., Patel, O., Zou, A., Mazeika, M., Wang, Z., Oswal, P., Lin, W., Hunt, A. A., Tienken-Harder, J., Shih, K. Y., Talley, K., Guan, J., Kaplan, R., Steneker, I., Campbell, D., Jokubaitis, B., Levinson, A., Wang, J., Qian, W., Karmakar, K. K., Basart, S., Fitz, S., Levine, M., Kumaraguru, P., Tupakula, U., Varadharajan, V., Wang, R., Shoshitaishvili, Y., Ba, J., Esvelt, K. M., Wang, A., and Hendrycks, D. (2024). The wmdp benchmark: Measuring and reducing malicious use with unlearning.

[26] Liu, J., Yang, C., Lu, Z., Chen, J., Li, Y., Zhang, M., Bai, T., Fang, Y., Sun, L., Yu, P. S., and Shi, C. (2025). Graph foundation models: Concepts, opportunities and challenges. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 47(6):5023–5044.

[27] Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., Liu, T., Tian, M., Kocetkov, D., Zucker, A., Belkada, Y., Wang, Z., Liu, Q., Abulkhanov, D., Paul, I., Li, Z., Li, W.-D., Risdal, M., Li, J., Zhu, J., Zhuo, T. Y., Zheltonozhskii, E., Dade, N. O. O., Yu, W., Krauß, L., Jain, N., Su, Y., He, X., Dey, M., Abati, E., Chai, Y., Muennighoff, N., Tang, X., Oblokulov, M., Akiki, C., Marone, M., Mou, C., Mishra, M., Gu, A., Hui, B., Dao, T., Zebaze, A., Dehaene, O., Patry, N., Xu, C., McAuley, J., Hu, H., Scholak, T., Paquet, S., Robinson, J., Anderson, C. J., Chapados, N., Patwary, M., Tajbakhsh, N., Jernite, Y., Ferrandis, C. M., Zhang, L., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. (2024). Starcoder 2 and the stack v2: The next generation.

[28] Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. (2025). Wizardcoder: Empowering code large language models with evol-instruct.

[29] MIT (2019). Scratch: Imagine, program, share.

[30] n8n (2025). Ai workflow automation tool.

[31] Naruse, K., Wang, J., Rodriguez, M., Truell, M., Catasta, M., Palmer, J., Wu, S., Tang, H., Gultekin, B., Axen, B., and et al. (2025). Claude opus 4.1.

[32] OpenAI (2025a). Introducing codex | openai.

[33] OpenAI (2025b). Introducing gpt-5 | openai.

[34] OpenAI (2025c). Introducing swe-bench verified | openai.

[35] OpenAI (2025d). Openai/gpt-oss: Gpt-oss-120b and gpt-oss-20b are two open-weight language models by openai.

[36] Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. (2023). Gorilla: Large language model connected with massive apis.

[37] Peng, Q., Chai, Y., and Li, X. (2024). Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization.

[38] Qwen (2025). Qwen3-coder: Agentic coding in the world.

[39] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80.

[40] Sypetkowski, M., Wenkel, F., Poursafaei, F., Dickson, N., Suri, K., Fradkin, P., and Beaini, D. (2024). On the scalability of gnns for molecular graphs. In Globerson, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J., and Zhang, C., editors, *Advances in Neural Information Processing Systems*, volume 37, pages 19870–19906. Curran Associates, Inc.

[41] Wang, H., Feng, S., He, T., Tan, Z., Han, X., and Tsvetkov, Y. (2024). Can language models solve graph problems in natural language?

[42] Wang, H., Wang, J., Wang, J., Zhao, M., Zhang, W., Zhang, F., Xie, X., and Guo, M. (2017). Graphgan: Graph representation learning with generative adversarial nets.

[43] Wang, Y. and Li, H. (2021). Code completion by modeling flattened abstract syntax trees as graphs.

[44] Wang, Z., Wang, Y., Yuan, C., Gu, R., and Huang, Y. (2021). Empirical analysis of performance bottlenecks in graph neural network training and inference with gpus. *Neurocomputing*, 446:165–191.

[45] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. (2023). Chain-of-thought prompting elicits reasoning in large language models.

[46] Willard, B. T. and Louf, R. (2023). Efficient guided generation for large language models.

[47] Wu, D., Ahmad, W. U., Zhang, D., Ramanathan, M. K., and Ma, X. (2024). Repoformer: Selective retrieval for repository-level code completion.

[48] Xue, R., Han, H., Zhao, T., Shah, N., Tang, J., and Liu, X. (2024). Largescale graph neural networks: Navigating the past and pioneering new horizons.

[49] Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., Zheng, C., Liu, D., Zhou, F., Huang, F., Hu, F., Ge, H., Wei, H., Lin, H., Tang, J., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Zhou, J., Lin, J., Dang, K., Bao, K., Yang, K., Yu, L., Deng, L., Li, M., Xue, M., Li, M., Zhang, P., Wang, P., Zhu, Q., Men, R., Gao, R., Liu, S., Luo, S., Li, T., Tang, T., Yin, W., Ren, X., Wang, X., Zhang, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Zhang, Y., Wan, Y., Liu, Y., Wang, Z., Cui, Z., Zhang, Z., Zhou, Z., and Qiu, Z. (2025). Qwen3 technical report.

[50] Ye, R., Zhang, C., Wang, R., Xu, S., and Zhang, Y. (2024). Language is all a graph needs.

[51] Yeo, E., Tong, Y., Niu, M., Neubig, G., and Yue, X. (2025). Demystifying long chain-of-thought reasoning in llms.

[52] You, J., Ying, R., Ren, X., Hamilton, W. L., and Leskovec, J. (2018). Graphrnn: Generating realistic graphs with deep auto-regressive models.

[53] Zan, D., Chen, B., Zhang, F., Lu, D., Wu, B., Guan, B., Wang, Y., and Lou, J.-G. (2023). Large language models meet nl2code: A survey.

[54] Zhang, J. (2023). Graph-toolformer: To empower llms with graph reasoning ability via prompt augmented by chatgpt.

A All ScratchTest Blocks

1. WhenFlagClicked
2. WhenKeyPressed
3. MoveSteps
4. TurnRight
5. TurnLeft
6. GoToRandom
7. GotoXY
8. GlideToRandom
9. GlideToXY
10. PointInDirection
11. ChangeXBy
12. SetXTo
13. ChangeYBy
14. SetYTo
15. XPosition
16. YPosition
17. Say
18. SayForSecs
19. Think
20. ThinkForSecs
21. ChangeSizeBy
22. SetSizeTo
23. Wait
24. Repeat
25. Forever
26. If
27. IfElse
28. WaitUntil
29. RepeatUntil
30. Stop
31. KeyPressed
32. MouseDown
33. Add

34. Subtract
35. Multiply
36. Divide
37. Random
38. GreaterThan
39. LessThan
40. Equals
41. And
42. Or
43. Not
44. Join
45. LetterOf
46. LengthOf
47. Contains
48. Mod
49. Round
50. MathFunction
51. SetVariable
52. ChangeVariableBy
53. GetVariable

B LLM Configurations

B.1 gpt-oss-120b

Attribute	Value
Model	openai/gpt-oss-120b
Temperature	1
Max Completion Tokens	8192
Top P	1
Reasoning Effort	Medium

B.2 qwen3-32b

Attribute	Value
Model	qwen/qwen3-32b
Temperature	1
Max Completion Tokens	8192
Top P	1
Reasoning Effort	Default

B.3 deepseek-r1-distill-llama-70b

Attribute	Value
Model	deepseek-r1-distill-llama-70b
Temperature	1
Max Completion Tokens	8192
Top P	1

B.4 llama-3.3-70b-versatile

Attribute	Value
Model	llama-3.3-70b-versatile
Temperature	1
Max Completion Tokens	8192
Top P	1

C LLM System Prompt

Goal

You are an expert software engineer extremely proficient in programming using a graph-based, visual programming language. You will be provided with the following information:

User Query: The overall functionality that you are required to fulfill through calling pre-declared functions.

Your task is to analyze the **User Query** and generate a connected graph that fulfills the requested functionality in **User Query**. All necessary nodes and connections MUST be included in your output. Each node can be instantiated in the graph more than once.

Node Reference (Complete Form)

Every possible node in their complete form:

{reference_nodes}

Output Format

Your output must be a JSON object in the following format: {“nodes”: Dict[str, Node], “edges”: List[Edge]}

The keys of the ‘nodes’ dictionary are NODEIDs, which are unique string identifiers such that no two nodes can have the same NODEID. A NODEID must contain only letters (a-z, A-Z) and be prefixed with “node_”.

The Node type is {“name”: NODENAME, “value”: any|None} The value field of a Node type is always None except for when it is for a constant, in which case its NODENAME is “Constant” and its value is the value it is supposed to hold (e.g. 10, “Hello”, True).

The Edge type is {“outNodeID”: str, “outPortID”: str, “inNodeID”: str, “inPortID”: str} outNodeID is the NODEID of the node of which you are using one of the outPorts. inNodeID is the NODEID of the node of which you are using one of the inPorts. Connecting constants to a node should be done like the following example: “moving 10 steps” means having the edge {“outNodeID”: IDOfAConstantNode, “outPortID”: “”, “inNodeID”: “IDOfAMoveStepsNode”, “inPortID”: “STEPS”}. A “field” of a node can be considered an inPort for the purposes of an Edge. An outPort connects to an inPort; an outPort cannot connect to another outPort, and an inPort cannot connect to another inPort.

D LLM System Prompt: Alternative Output Graph Representation

Goal

You are an expert software engineer extremely proficient in programming using a graph-based, visual programming language. You will be provided with the following information:

User Query: The overall functionality that you are required to fulfill through calling pre-declared functions.

Your task is to analyze the **User Query** and generate a connected graph that fulfills the requested functionality in **User Query**. All necessary nodes and connections MUST be included in your output. Each node can be instantiated in the graph more than once.

Node Reference (Complete Form)

Every possible node in their complete form:

{reference_nodes}

Output Format

Your output must be a JSON object in the following format: {[key: NODEID]: “nodeName”: NODENAME, “value”: any|None, “edges”: List[Edge]}

NODEIDs are unique string identifiers such that no two nodes can have the same NODEID. A NODEID must contain only letters (a-z, A-Z) and be prefixed with “node_”. The value field of a Node type is always None except for when it is for a constant, in which case its NODENAME is “Constant” and its value is the value it is supposed to hold (e.g. 10, “Hello”, True).

The Edge type is {“portID”: str, “otherNodeID”: OTHERNODEID}

OTHERNODEID is the NODEID this current node is connected to via the port with ID that matches the one specified in portID. An outPort connects to an inPort; an outPort cannot connect to another outPort, and an inPort cannot connect to another inPort. A Constant node has only one port, and it is an outPort: VALUE. A “field” of a node can be considered an inPort for the purposes of an Edge. Port connections must be defined in the “edges” list for both the to and from nodes.

E Granular Per-Prompt Results Breakdown: Reference Node Representation

The following results are ordered by prompt number. The prompts are:

1. When the green flag is clicked, continuously move in a square pattern until the user presses the space key.
2. When the “r” key is pressed, set size to a random number between 50 and 150.
3. When the “s” key is pressed, say a secret password made of two random letters and three random numbers.
4. When the green flag is clicked, simulate a loading bar by incrementally increasing a progress variable and displaying it.
5. When the green flag is clicked, repeat until key “q” is pressed: move randomly.
6. When the green flag is clicked, move forward by 50 steps and rotate right 45 degrees, repeating forever.
7. When the “down arrow” key is pressed, decrease the sprite’s size by 10.
8. When the green flag is clicked, repeat until X is greater than 150: move 10 steps.
9. When the green flag is clicked, join “X is ” with current X position and say it.
10. When the “d” key is pressed, set X to 30 and Y to 6.
11. When the green flag is clicked, simulate a spiral by increasing steps and turning each time.
12. When the green flag is clicked, if size is greater than 100, say “Too big!”.
13. When the green flag is clicked, say a sentence constructed from 3 random words.
14. When the “r” key is pressed, simulate a die roll and say the result.
15. When the green flag is clicked, count up for 5s from 0 to 100, saying each number.
16. When the green flag is clicked, simulate a heartbeat by growing and shrinking repeatedly.
17. When the “left arrow” key is pressed, change X by -20 and say “Going left!”
18. When the green flag is clicked, wait until the “a” key is pressed, then jump to (0, 0).
19. When the green flag is clicked, alternate between saying “Tick” and “Tock” forever.
20. When the “x” key is pressed, say a randomly selected word from a list.

The results are logged using the following symbols: ✓ means correct, × means incorrect logic, E means the generation resulted in an error.

E.1 Run 1

Name	Results
No Types	×✓✓E✓ ✓✓✓✓✓ ×E×✓× E✓✓✓E
Extra Description	✓✓✓×✓ ✓✓✓✓✓ ×E✓✓E E✓✓✓×
Proposed	×✓✓E✓ ✓✓✓✓✓ ✓E✓✓✓ ✓✓✓✓E

E.2 Run 2

Name	Results
No Types	×✓✓✓✓ ✓✓✓✓✓ ✓E✓××E✓✓✓×
Extra Description	✓✓✓✓✓ ✓✓✓✓✓ ✓E✓✓✓ E✓✓×E
Proposed	✓✓✓×✓ ✓×✓✓✓ ✓E✓××E✓✓✓

E.3 Run 3

Name	Results
No Types	E✓×✓✓ ✓✓✓✓✓ ×EE✓× E✓✓✓E
Extra Description	✓✓×✓✓ ✓✓✓✓✓ ×EE✓E ✓✓✓✓E
Proposed	E✓✓×✓ ✓✓✓✓✓ ✓E✓✓× ✓✓✓✓E

E.4 Run 4

Name	Results
No Types	✓✓✓×✓ ✓E✓✓✓ ✓E✓×✓ E✓✓✓✓
Extra Description	✓✓E✓✓ ✓✓✓✓✓ ✓E✓✓✓ E✓✓✓✓
Proposed	✓✓✓×✓ ✓E✓✓✓ ✓✓✓✓× ✓✓✓✓E

E.5 Run 5

Name	Results
No Types	E✓✓×✓ ✓✓✓✓✓ ×E×✓× E✓✓✓E
Extra Description	✓✓✓×✓ ✓✓✓✓✓ ×EE✓✓ E✓✓×E
Proposed	✓✓✓×✓ ✓E✓✓✓ ×E✓✓× E✓✓✓✓

F Granular Per-Prompt Results Breakdown: Output Graph Representation

The following results are ordered by prompt number. The prompts are the same as in Appendix E.

The results are logged using the following symbols: ✓ means correct, × means incorrect logic, E means the generation resulted in an error.

F.1 Run 1

Name	Results
Alternative	E✓×E✓ ✓✓E✓× ×E✓E✓ E✓✓✓E
Proposed	×✓✓E✓ ✓✓✓✓✓ ✓E✓✓✓ ✓✓✓✓E

F.2 Run 2

Name	Results
Alternative	E✓✓EE E✓E✓✓ EE✓✓E ✓✓×EE
Proposed	✓✓✓×✓ ✓×✓✓✓ ✓E✓××E✓✓✓✓

F.3 Run 3

Name	Results
Alternative	E✓EE✓ ✓✓E✓✓ EE✓✓× E✓×E✓
Proposed	E✓✓×✓ ✓✓✓✓✓ ✓E✓✓× ✓✓✓✓E

F.4 Run 4

Name	Results
Alternative	E✓✓EE ✓✓✓✓✓ EE✓✓E E✓✓EE
Proposed	✓✓✓×✓ ✓E✓✓✓ ✓✓✓✓× ✓✓✓✓E

F.5 Run 5

Name	Results
Alternative	E✓✓EE E✓✓✓✓ ×E✓××E✓✓EE
Proposed	✓✓✓×✓ ✓E✓✓✓ ×E✓✓×E✓✓✓✓

G Evaluation for Other LLMs

The following results are ordered by prompt number. The prompts are the same as in Appendix E.

The results are logged using the following symbols: ✓ means correct, × means incorrect logic, E means the generation resulted in an error.

G.1 qwen3-32b

Table 4: ScratchTest results for reference node representation ablations.

Name	Results
No Types	E×EEE E×✓✓× EE××E E×✓E×
Extra Description	EE×E✓ E×E✓× ×EE×E ×EEE×
Proposed	E×✓E× ××✓✓× EE✓×E E×✓×E

Table 5: ScratchTest results for output graph representation ablations.

Name	Results
Alternative	EEEEE EEEE× EEEEE EEEEE
Proposed	E×✓E× ××✓✓× EE✓×E E×✓×E

G.2 deepseek-r1-distill-llama-70b

Table 6: ScratchTest results for reference node representation ablations.

Name	Results
No Types	E×EE× ××E✓✓ EE✓EE E×✓EE
Extra Description	EE×E✓ EE✓✓× EEE×E ××✓EE
Proposed	E×EE✓ ✓×✓✓E EEEEE ✓×✓EE

Table 7: ScratchTest results for output graph representation ablations.

Name	Results
Alternative	EEEEE EEE✓E EEEEE EEEEE
Proposed	E×EE✓ ✓×✓✓E EEEEE ✓×✓EE

G.3 llama-3.3-70b-versatile

Table 8: ScratchTest results for reference node representation ablations.

Name	Results
No Types	E✓×EE ✓EE✓✓ EEE✓E EE✓E×
Extra Description	E✓EEE EE✓EE EE×EE E✓EE×
Proposed	E✓EE× E✓E✓× E✓✓E✓ E✓✓EE

Table 9: ScratchTest results for output graph representation ablations.

Name	Results
Alternative	EEEEE EEEE× ✓EEEE EEEEE
Proposed	E✓EE× E✓E✓× E✓✓E✓ E✓✓EE

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We make sure to acknowledge the progress that has been made by the LLM and Deep Learning For Code community in raw sequential code generation, while highlighting that graph-based abstract code generation is an overlooked vertical in the field. We also condition our approach to a one-agent LLM framework and JSON representations. We also highlight our evaluation method, which is a new mini-benchmark based on our own Python reimplementation of Scratch.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We have a dedicated Limitations section which highlights the errors our approach faces as well as the self-imposed restrictions we placed on our work, which when lifted will likely increase the accuracy of the approach.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: The paper does not include theoretical results.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We have specified the model configurations, the platform we run it on, and the system prompt (and its variations). We have also specified the prompts we used to test our approach. These are found in Section 4 and Appendices B, C, D, and E.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility.

In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Since the OpenReview page does not have the Supplementary Material field, we will be releasing the relevant materials via GitHub once we are able to unanonymize our paper. Note that since we used Groq in our experiments, API credits must be purchased in order to run them.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We have specified the model configurations, the platform we run it on, and the system prompt (and its variations). We have also specified the prompts we used to test our approach. These are found in Section 4 and Appendices B, C, D, and E.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We have included standard deviations and p-values for statistical significance. These are found in Section 5.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We specify the platform (Groq) which was used to run the experiments.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed the NeurIPS Code of Ethics and our paper conforms fully.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

Justification: We have described societal impacts in our paper in Section 7.

Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper does not pose such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We use open-source models and respect open-source licenses. We also use our own dataset for benchmarking.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.

- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: Since the OpenReview page does not have the Supplementary Material field, we will be releasing the relevant materials via GitHub once we are able to unanonymize our paper. Documentation for these assets will be well-written and included alongside the assets.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: The core method development in this research does not involve LLMs as any important, original, or non-standard components.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.