Neural Integration of Iterative Reasoning (NIR) in LLMs for Code Generation

Soran Ghaderi

Supervisor: Prof. L. Citi

School of Computer Science and Electronic Engineering
University of Essex

This dissertation is submitted for the degree of Master of Science in
Artificial Intelligence

CSEE December 2024

I would like to dedicate this dissertation to my loving parents . . .

Declaration

The author hereby declares that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Soran Ghaderi
December 2024

Acknowledgements

I want to take a moment to sincerely thank the University of Essex for their incredible support and financial assistance during my master's program. Their dedication to creating a vibrant academic environment has truly made a difference in my journey. I'm grateful to my supervisor, Professor Luca, whose guidance and mentorship have helped my research in meaningful ways. The access to GPU resources he provided was invaluable in helping me conduct this dissertation. I also want to acknowledge my fellow students and colleagues; their support and camaraderie have made this journey much more enjoyable. Collaborating with them has not only enriched my experience but also improved the quality of my work. Lastly, I owe a huge debt of gratitude to my family and friends for their unwavering encouragement. Their belief in me has kept me motivated throughout this challenging yet rewarding journey.

Abstract

Despite advances in large language models (LLMs) for code generation, they still struggle to effectively utilize contextual information throughout the generation process. To tackle this challenge, we introduce the Neural Integration of Iterative Reasoning (NIR) framework, which offers a new method for incorporating Context Representation Vectors (CRVs) at multiple levels within LLMs. NIR boosts the ability of these models to generate code without the need for fine-tuning, which allows it to be used across various LLM architectures. We evaluated NIR by testing it with LLaMA 3.1 on the MBPP dataset, focusing on the early, mid- and deep integration stages. Our experiments show that the depth of CRV integration has a notable impact on several aspects of code generation, including response rates, syntactic correctness, and overall code structure. Deeper integration generally improves syntactic accuracy and code conciseness, while mid-layer integration shows optimal performance in semantic tasks. We report detailed evaluation metrics that assess code quality, complexity, and structure. Our findings indicate possible trade-offs among various code quality measures and emphasize the potential of adaptive integration strategies. Although NIR demonstrates promising results, we also identify limitations such as dataset specificity and output inconsistencies. This study contributes to understanding contextual information processing in LLMs and might be useful for future developments in code-LLMs. We outline future research directions, including multilayer integration and dynamic adaptation strategies.

Table of contents

List of figures xv

List of tables xvii

Nomenclature xix

1 Introduction **1**

1.1 Background on LLMs and their limitations . . . 1
1.2 Problem Statement and Research Questions . . . 2
1.3 Scope and Objectives . . . 2
1.4 Contributions of the study . . . 3
1.5 Thesis Structure . . . 4

2 Literature Review **5**

2.1 Traditional Methods Used in LLMs for Reasoning in Code Generation . . . 5
2.1.1 Chain-of-Thought Prompting . . . 5
2.1.2 Self-Reflection and Iterative Refinement . . . 6
2.1.3 Context-Aware Generation Methods . . . 7
2.1.4 Retrieval-Augmented Generation . . . 7

3 Proposed Architecture **9**

3.1 Overview . . . 9
3.2 Theoritical Framework . . . 10
3.2.1 Thinking Stage . . . 10
3.2.2 CRV Generation . . . 10
3.2.3 CRV stacking . . . 14
3.2.4 Dimensionality Reduction . . . 14
3.2.5 Integrating CRVs with Hidden States . . . 15
3.2.6 Generation Stage . . . 21

3.2.7 Overall Process 21

4 Methodology 23

4.1 Dataset Selection and Preprocessing 23

4.1.1 Mostly Basic Python Programming (MBPP) 23

4.2 Quantitative Metrics 23

4.2.1 Response Rate 23

4.2.2 Code Quality Metrics 24

4.2.3 Code Structure and Complexity Analysis 26

4.3 Qualitative Analysis 28

4.3.1 Code Structure and Readability 28

4.3.2 Algorithm Understanding and Implementation 28

4.3.3 Error Patterns and Limitations 29

5 Experiments and Results 31

5.1 Experimental Setup 31

5.1.1 The NIR Framework Configuration 31

5.1.2 Hardware and Software Specifications 32

5.1.3 Intermediate Processing Steps 32

5.2 Quantitative metrics 34

5.2.1 Response Rate 34

5.2.2 Code Quality Metrics 34

5.2.3 Code Structure and Complexity Analysis 36

5.3 Qualitative Analysis 39

5.3.1 Code Structure and Readability 39

5.3.2 Algorithm Understanding and Implementation 40

5.3.3 Error Patterns and Limitations 40

5.3.4 Qualitative Analysis Implications 41

5.4 Ablation Studies 41

5.4.1 Results 41

5.4.2 Analysis 42

5.4.3 Ablation Studies Implications 42

6 Discussion 45

6.1 Interpretation of Results 45

6.1.1 Response Rate and Code Quality 45

6.1.2 Code Complexity and Structure 46

6.1.3 Implications for Model Architecture and Training 46
6.2 Limitations of the Current Approach 47
6.2.1 Model and Dataset Limitations 47
6.2.2 Output Inconsistencies 48
6.2.3 Context Generation Incompleteness 48
6.2.4 Trade-offs in Code Characteristics 48

7 Conclusion and Future Work 49
7.1 Summary of key findings 49
7.2 Implications for LLM development 50
7.2.1 Adaptive Architecture Design 50
7.2.2 Hierarchical Contextual Processing 50
7.3 Potential future research directions 50
7.3.1 Future Research Directions 50

References 53

Appendix A Dataset Structure 61
A.0.1 Example Entry 61
A.1 Model Modifications 62

List of figures

3.1 The generation pipeline of NIR architecture. 9
3.2 An illustration of the proposed architecture. The red circles indicate concatenated elements and the green circles represent the original elements of the hidden states. 12
3.3 Illustration of the Thinking Stage process. 13
3.4 This diagram shows the pre-concatenation process of the new sequence and the original elements of the hidden states. 15
3.5 Updating the causal attention mask with respect to the concatenated sequence 20

4.1 Function Name Consistency 24

5.1 Hardware and Software Specifications 32
5.2 Intermediate Processing Steps 33

List of tables

5.1 NIR Framework Configuration 31
5.2 Sampling Configuration 33
5.3 Response Rates Across Different Layers 34
5.4 Code Quality Metrics Across Different Layers 35
5.5 Code Complexity Metrics Across Different Layers 35
5.6 Code Structure Metrics Across Different Layers 36
5.7 Basic Halstead Metrics Across Different Layers 37
5.8 Derived Halstead Metrics Across Different Layers 38
5.9 Complexity Halstead Metrics Across Different Layers 38

A.1 Example Entry for MBPP Dataset 62

Nomenclature

Roman Symbols

B Batch size

$\mathscr{C}$ Set of CRV stacks

C^j The j-th CRV stack

C_i^j The i-th layer of the j-th CRV stack

CRV Context Representation Vector

CRVreduced Reduced set of selected CRVs

CRVstack Complete stack of CRVs

d Dimensionality of the hidden state

E Number of edges in control flow graph

F Entire model function including thinking, integration, and generation stages

f_g Generation function

f_u Hidden state update function

g Selection function for CRV reduction

H Stack of all hidden states

$H^{(i)}$ Hidden state tensor for layer i

h_i Hidden state at the i-th layer

h' Updated hidden state

h'_i	New hidden state after CRV integration
H_1	Number of unique operators in code
H_2	Number of unique operands in code
$h^{(i)}$	Original hidden state tensor for layer i
h_t	Hidden state at time step t
$\mathscr{I}$	Set of layers where integration occurs
k	Key vector
K'	Updated key matrix
k'	Transformed key vector after RoPE application
k'_i	Updated key vector at position i in extended sequence
K'_l	Updated key matrix for layer l
K_l	Original key matrix for layer l
l	Sequence length
l_0	Original sequence length
$L^{(i)}$	Sequence length at layer i
LLM	Large Language Model
L_o	Original sequence length
L_u	Updated sequence length
$\mathscr{M}$	Set of all model layers
m	Length of context generated from thinking stage
M'	Updated attention mask
M_a	Additional mask for CRVs
M_o	Original attention mask
M_u	Updated attention mask

N Number of nodes in control flow graph

n Total number of layers

N_1 Total number of occurrences of operators

N_2 Total number of occurrences of operands

P Number of connected components in control flow graph

PE' Updated positional encoding

Q Input query

q Input query

q' Transformed query vector after RoPE application

q'_i Updated query vector at position i in extended sequence

q_i i-th element of the input query

$R_{\Theta,n,d}$ RoPE encoding matrix

$R'_{\Theta,i,d}$ Recalculated RoPE encoding matrix for extended sequence

T Thinking stage function

t Output of the thinking stage

V' Updated value matrix

V'_l Updated value matrix for layer l

V_l Original value matrix for layer l

Y Output sequence

y_t Output at time step t

Greek Symbols

Φ Function for updating attention mask

Ψ Integration function

θ Parameters of the thinking stage

θ_d Angle for RoPE encoding at dimension d

Chapter 1

Introduction

The field of artificial intelligence, particularly natural language processing and code generation, has seen remarkable advances in recent years Wang et al. [2021]Black et al. [2022]Brown et al. [2020]Beltagy et al. [2020]. Large language models (LLMs) have demonstrated significant capabilities in understanding and generating natural language and programming code Li et al. [2023]. However, these models often struggle with maintaining a consistent context and applying appropriate reasoning strategies across a diverse coding tasks. This dissertation introduces and explores the concept of neural integration of iterative reasoning, a novel approach to enhance the reasoning abilities of language models in code generation by enhancing LLMs through mid-layer thought injection and investigating its impact.

1.1 Background on LLMs and their limitations

The evolution of language models has led to significant improvements in code generation tasks. Models like StarCoder Li et al. [2023], Codex Chen et al. [2021], and their successors have shown impressive results in translating natural language descriptions into functional code in various programming languages.

However, despite these notable achievements, current language models face several significant challenges that limit their practical applicability and reliability for real-world code generation tasks. One major limitation is the lack of consistent adherence to the provided context and requirements. LLMs often struggle to fully understand and incorporate the nuances and constraints specified in the input prompt, leading to generated code that may be syntactically correct, but semantically inconsistent or irrelevant to the desired functionality Li et al. [2022]. Another critical issue is the limited ability of LLMs to reason about the underlying logic and algorithmic complexity of the code they generate. Although they can

produce code that appears to work for simple test cases, they often fail to consider edge cases, handle complex data structures efficiently, or optimize for performance.

To address these limitations, in this dissertation, we propose a new approach, which aims to enhance LLMs by integrating iterative reasoning through the injection of embeddings at the intermediate layers of the model. By capturing and leveraging the step-by-step reasoning with CoT and self-reflection and processes crucial for effective code generation, the goal is to enable LLMs to more closely mimic the flexible and adaptive thinking of humans.

1.2 Problem Statement and Research Questions

In this research, we aim to introduce an alternative approach to guide the generation process in LLMs using context generated through an inner-monologue stage, which we call neural integration of iterative reasoning, to enhance the reasoning capabilities of LLMs in code generation. Furthermore, we will offer a comprehensive comparison of our method with direct text-level prompting, evaluating several metrics such as code correctness and compilability

The key research questions that drive this dissertation are as follows:

1. How can we enhance LLMs to better capture and emulate the iterative reasoning in a process separate from the main code generation?
2. What is the impact of integrating thought embeddings at intermediate layers of LLMs on their reasoning abilities and the quality of generated code?
3. How does the proposed mid-layer thought injection approach compare to existing code generation techniques in terms of performance, context adherence, and adaptability?
4. What are the potential implications of this research for the future development of LLMs and their application in software engineering tasks?

By addressing these research questions, we aim to contribute to the code generation capability in LLMs and potentially as a part of a larger research for future work that can be utilized in other NLP and multimodal tasks.

1.3 Scope and Objectives

We focus on the designing, implementation, and evaluation of the Neural Integration of Iterative reasoning (NIR) approach, as a technique designed to enhance code generation capabilities. The core of this work involves designing and implementing a differentiable

architecture that enables context manipulation by injecting a proposed solution, thoughts, and general contexts to help with solving the problem. To assess the impact of this approach, specific metrics and evaluation methodologies will be utilized to measure the results in both reasoning abilities and the overall quality of generated code compared with vanila CoT direct prompting on the text level. Moreover, the effectiveness of the NIR approach will be validated through experimentation with a variety of task complexities using Python programming language.

1.4 Contributions of the study

We summarize our contributions as follows:

1. Integrates context representation vectors (CRVs) at various depths within the LLaMA 3.1 model without the need for fine-tuning. This approach provides insights into the connection between contextual information and the model's inherent generating stages.

2. Our findings show a nuanced relationship between integration depth and code quality metrics. Notably, we observe a generally optimal-performing depth at mid-level integration (Layer 10), that results in an optimal balance between consuming CRVs and preserving the model's learned representations.

3. We then highlight the potential need for developing adaptive integration strategies where, the CRV injection layers could be dynamically adjusted based on task complexity.

4. Implications for future LLM development: Insights into the potential impact of the NIR approach on the future development of LLMs and their application in software engineering tasks, considering aspects such as scalability, interpretability, transfer learning potential, and ethical considerations.

5. Comparative analyses, quantitative measurements, and qualitative case studies that demonstrate the effectiveness of the NIR approach.

6. Last but not least, we point out potential trade-offs between syntactic correctness, complexity, and conciseness.

1.5 Thesis Structure

The dissertation proceeds as follows:
in **Chapter 2** We will provide a comprehensive assessment of the literature on both contemporary and conventional approaches that LLMs employ for reasoning in code development.
We then **Chapter 3** introduce the proposed architecture along with the underlying theoretical concepts.
Furthermore in **Chapter 4**, we describe the research methodology, including dataset preparation, preprocessing steps, and the implementation and evaluation of the proposed architecture.
Moving to **Chapter 5**, we cover the experimental setup, datasets, evaluation metrics, and baseline models used for comparison.
In **Chapter 6** we discuss the performance metrics, experimental results, and a detailed analysis of the architectures.
Finally in **Chapter 7** we conclude the dissertation by summarizing key findings, discussing limitations, and proposing future research directions.

Chapter 2

Literature Review

2.1 Traditional Methods Used in LLMs for Reasoning in Code Generation

In recent years, reasoning has emerged as a key area of research in large language models due to the crucial need to enhance the model reliability and interpretability. Therefore, in this section, we will thoroughly examine the methods, approaches, and architectures used to enhance the efficiency of LLMs in reasoning, highlighting the strengths and weaknesses of each in the context of code generation. This analysis will provide insights into the current capabilities of LLMs, the problems they still face, and the future prospects. In addition, we will demonstrate how our research addresses these gaps and introduces a new perspective on reasoning.

2.1.1 Chain-of-Thought Prompting

Large Language Models (LLMs) have achieved significant advances in the NLP field, showing success across various tasks such as text classification, machine translation, and question answering. However, these models face several limitations, with one major shortcoming being their ability to reason effectively. This limitation cannot simply be resolved by increasing the model's size Srivastava et al. [2023].

In order to address this limitation, J. Wei et al. Wei et al. [2023] introduced the Chain of Thoughts (CoT) prompting method. CoT encourages the model to break down tasks into smaller components, imitating human reasoning processes. For instance, given the question: "Soran gained 5 pounds and 3 pounds from his business, and had to pay 2 pounds in taxes. How much did he gain?" The direct answer might be 6, but using CoT, the reasoning process

would be: "Soran initially earned 5 pounds from his business, then gained an additional 3 pounds, totaling 8 pounds. After paying 2 pounds in taxes, his net gain is 6 pounds."

CoT has demonstrated strong potential in reasoning tasks, but its application was initially limited to a single line of reasoning. The Google Brain team addressed this by introducing self-consistency Wang et al. [2023], which generates multiple reasoning paths. This method mimics the variety in human thinking when solving problems, with each path providing an answer. The final answer is determined by majority voting among the different paths, which boosts confidence and reduces errors. This enhancement has made CoT with self-consistency more robust, leading to its integration into several LLMs when dealing with reasoning problems OpenAI et al. [2024]Dubey et al. [2024].

While CoT has led to notable improvements in LLM reasoning, it has primarily been tested on mathematical tasks. Studies like Zhou et al. [2022]Fu et al. [2022]Chae et al. [2024]Xu et al. [2024]Qi et al. [2024] have mainly reported results within the mathematical domain, leaving its performance in other areas unexplored. Other research, such as Kambhampati et al. [2024]Wang et al. [2024]Sprague et al. [2024], suggests that CoT may be less effective or even counterproductive in various non-mathematical domains. For example, reasoning in code generation remains a significant challenge for LLMs.

2.1.2 Self-Reflection and Iterative Refinement

N. Shinn et al. Shinn et al. [2023] introduced the concept of Reflexion, a novel framework for enhancing language agents through verbal reinforcement learning with a self-reflection mechanism. Their framework consists of four components: actor, evaluator, self-reflection, and memory, each playing a crucial role in the system. The actor, which is an LLM, generates text based on observed states from the environment, while the evaluator assesses the quality of this output. The self-reflection model, also an LLM, generates verbal feedback to guide the decision-making process. Finally, Reflexion agents rely on both short-term and long-term memory.

This framework has shown remarkable results on the HumanEval coding benchmark, surpassing GPT-4's 80% with a score of 91% Shinn et al. [2023]. However, a limitation of the framework is that it only operates on the final solution directly, rather than on the intermediate stages (thinking process), which reduces its reasoning ability. Additionally, the framework only works with text and does not support other modalities. The framework could be optimized by integrating self-reflection into the prompt rather than using a separate block for it.

2.1.3 Context-Aware Generation Methods

The context window is the maximum length of sequences that an LLM can process at once, determined during the model's training process, and represents a key limitation of pre-trained models. To overcome this limitation, many researchers have begun exploring ways to extend the context window by fine-tuning models on small datasets (or without fine-tuning), with positional encoding becoming a focal point of attention.

The improvement began with the introduction of learnable absolute position encoding Gehring et al. [2017] rather than the original Transformer's absolute sinusoidal position encoding Vaswani et al. [2017]. Additionally, relative positional encoding methods have been developed Shaw et al. [2018], increasing performance and leading to more widely adopted techniques such as T5 Relative Bias Raffel et al. [2023], RoPE Su et al. [2023], XPos Sun et al. [2023], and ALiBi Press et al. [2022].

However, these methods still face the constraint of being unable to reason beyond the context window. Studies such as Chen et al. [2023] modified RoPE by fine-tuning on small datasets and using Position Interpolation (PI). NTK-Aware was also introduced bloc97 [2023], accounting for the loss of high frequencies, which led to further advancements such as "Dynamic NTK" interpolation emozilla [2023] and "NTK-by-parts" interpolation bloc97 [2023]. These improvements culminated in YaRN Peng et al. [2023], a state-of-the-art method that extended the context window after fine-tuning on less than 0.1% of the original pre-training data Peng et al. [2023].

While these studies have made remarkable progress in expanding the context window, allowing LLMs to process more information, the challenge remains to improve reasoning abilities by feeding the model with relevant, structured knowledge to enhance critical thinking and decision-making, rather than simply increasing the amount of information processed.

2.1.4 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) Lewis et al. [2020] is an innovative approach in natural language processing (NLP) that combines two core components: retrieval and generation. The retrieval phase is for finding documents that match the query and feed it to the large language model (LLM). This approach have many advantages, since it helps the model to extract accurate answers and reduces hallucinations by connecting responses to references. Moreover, it eliminates the need to annotate documents with metadata Barnett et al. [2024].

Even though RAG models offer several benefits, they still fail when presented with questions that cannot be answered using the available documents. Furthermore, the models

might fail to extract the correct answer from the context due to noise or contradicting information Barnett et al. [2024]. Most importantly, in the context of reasoning, RAG systems operate at the text level and rely on external tools such as databases and search engines, which may not directly enhance reasoning. In addition, they are challenging to evaluate due to the different components with varying functionalities.

Chapter 3

Proposed Architecture

3.1 Overview

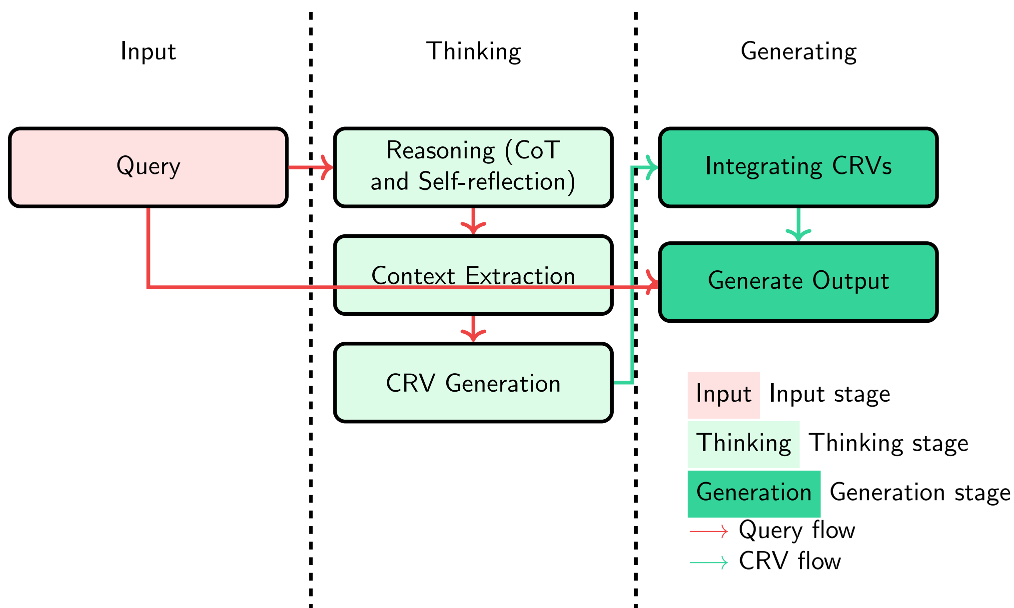

Fig. 3.1 The generation pipeline of NIR architecture.

In NIR framework, we divide the code generation into three main steps including:

1. Querying the model
2. Thinking stage

3. Generation stage

In figure 3.1, we illustrate the overall NIR pipeline for code generation:

3.2 Theoritical Framework

In this section, we will present the theoretical framework for the NIR architecture illustrated in Figure 3.2 and elucidate the foundational concepts.

3.2.1 Thinking Stage

The thinking stage can be formulated as a function T that generates the query t:

$$t = T(q, \theta) \tag{3.1}$$

where:

- $q \in \mathbb{R}^k$ is the initial input query
- θ are the parameters of the thinking stage
- $T : \mathbb{R}^k \times \Theta \to \mathbb{R}^m$ is the thinking stage function

The Algorithm 1 presents the algorithmic process taking place in this stage. Furthermore, Figure 3.3 illustrates this stage visually.

3.2.2 CRV Generation

Let the CRVs generated from the thinking stage be defined as:

$$\text{CRV}_i = f_i(t, h_i), \quad i = 1, 2, \ldots, n \tag{3.2}$$

where:

- $\text{CRV}_i \in \mathbb{R}^{d \times l}$ is the i-th Context Representation Vector
- $f_i : \mathbb{R}^m \times \mathbb{R}^{d \times l} \to \mathbb{R}^{d \times l}$ is the function representing the transformation at the i-th layer
- $t \in \mathbb{R}^m$ is the query generated by the thinking stage
- $h_i \in \mathbb{R}^{d \times l}$ is the hidden state at the i-th layer

Algorithm 1 Thinking Stage Algorithm

```
Require:
    question: Input question or problem statement
Ensure:
    context: Generated context for the solution
    pythonCode: Final Python code solution
 1: procedure ThinkingStage(question)
 2:     analysis ← Analyze(question)
 3:     hints ← GenerateHints(analysis)
 4:     for all hint ∈ hints do
 5:         rationale, answer ← GenerateRationaleAndAnswer(hint)
 6:         repeat
 7:             reflection ← ReflectOnRationale(rationale)
 8:             if reflection is Confirmed then
 9:                 break
10:             end if
11:             hint ← RejectedHint(hint)
12:             rationale, answer ← GenerateRationaleAndAnswer(hint)
13:         until reflection is Confirmed
14:         bestRationale ← SelectBestRationale(rationale)
15:         repeat
16:             pythonCode ← SuggestPythonCode(bestRationale)
17:             review ← ReviewCode(pythonCode)
18:             if review is not Good then
19:                 RefineCode(pythonCode)
20:             end if
21:         until review is Good
22:         context ← GenerateContext(pythonCode)
23:         Output(context, pythonCode)
24:     end for
25:     return context, pythonCode
26: end procedure
```

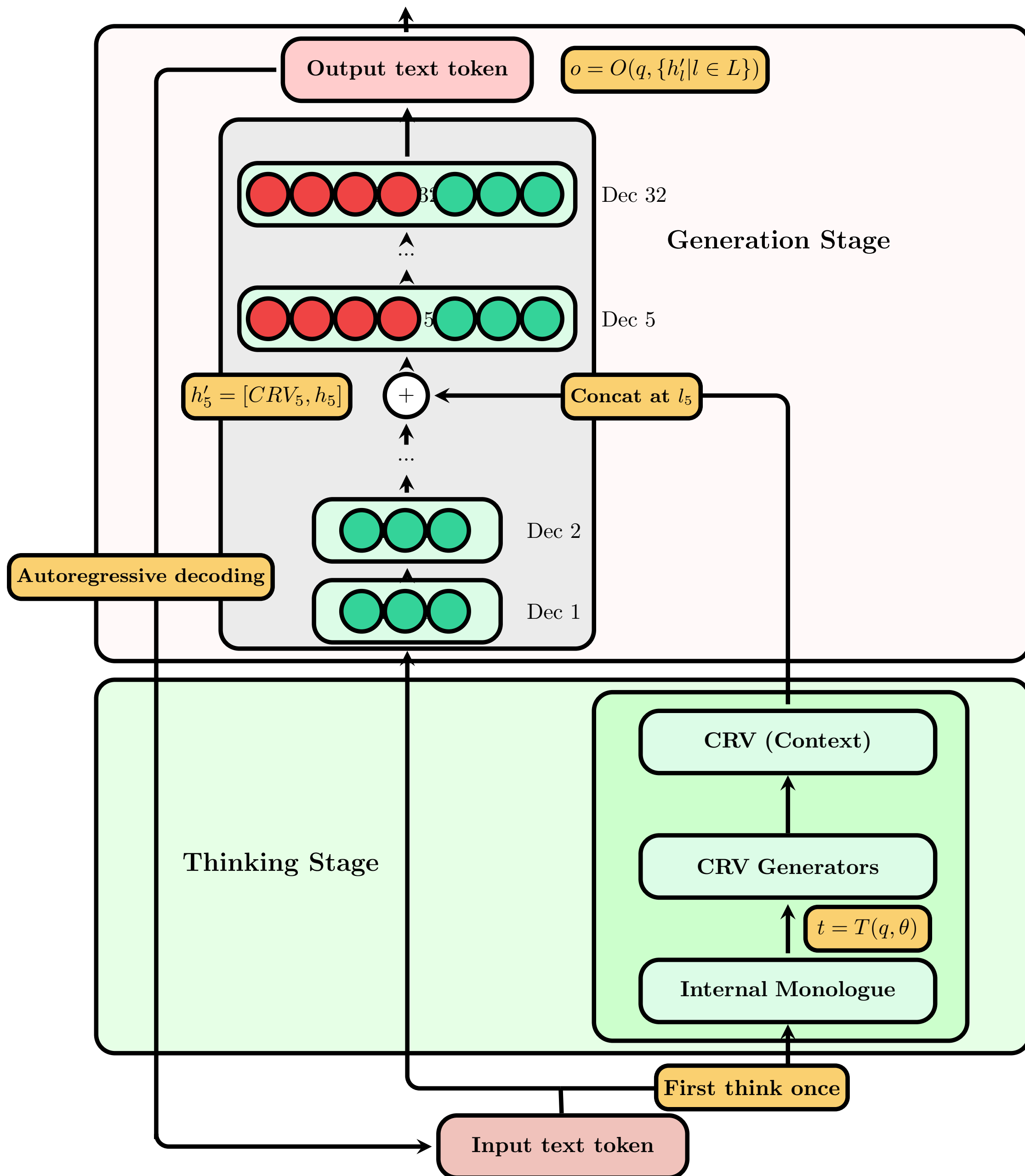

Fig. 3.2 An illustration of the proposed architecture. The red circles indicate concatenated elements and the green circles represent the original elements of the hidden states.

The complete CRV stack for input t can be represented as a tensor:

$$\mathrm{CRV}_{\mathrm{stack}} = [\mathrm{CRV}_1; \mathrm{CRV}_2; \ldots; \mathrm{CRV}_n] \in \mathbb{R}^{n \times d \times l} \tag{3.3}$$

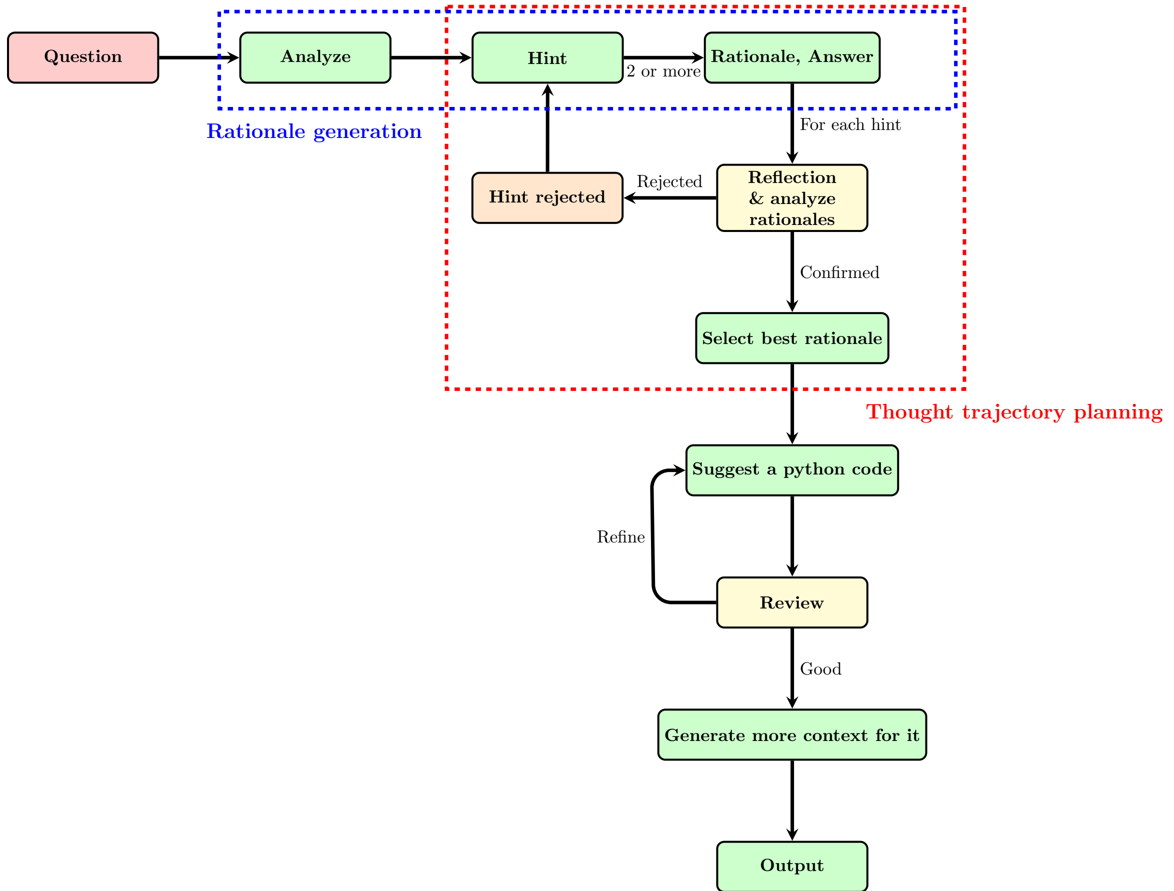

Fig. 3.3 Illustration of the Thinking Stage process.

where:

- n is the total number of layers
- d is the dimensionality of the hidden state
- l is the sequence length
- $[;]$ denotes vertical stacking.

Combining equations 3.1 and 3.3

The generation of the CRV stack can then be expressed as a composition of functions:

$$\text{CRV}_{\text{stack}} = F(T(q,\theta),H) \tag{3.4}$$

where:

- $F : \mathbb{R}^m \times \mathbb{R}^{n\times d\times l} \rightarrow \mathbb{R}^{n\times d\times l}$ is the overall CRV generation function
- $H = [h_1; h_2; \ldots; h_n] \in \mathbb{R}^{n\times d\times l}$ is the stack of all hidden states

Each CRV_i can be extracted from the stack using an indexing operation:

$$\text{CRV}_i = \text{CRV}_{\text{stack}}[i, :, :] \tag{3.5}$$

3.2.3 CRV stacking

The stacking of CRVs vertically is represented as:

$$\text{CRV} = \begin{bmatrix} \text{CRV}_1 \\ \text{CRV}_2 \\ \vdots \\ \text{CRV}_n \end{bmatrix} \tag{3.6}$$

3.2.4 Dimensionality Reduction

Furthermore, a reduction in dimension can be applied. Here, we maintain a specified number of CRVs as representatives of the context:

$$\text{CRV}_{\text{reduced}} = g(\{\text{CRV}_i\}_{i=1}^{L}) \tag{3.7}$$

where:

- g is a selection function that chooses a subset of CRVs
- $\{\text{CRV}_i\}_{i=1}^{L}$ is the set of all CRVs from L layers of the model
- $\text{CRV}_{\text{reduced}}$ is the reduced set of selected CRVs

The function g could be implemented in various ways, such as:

- Selecting a stack of arbitrary CRVs from the decoder layers
- Selecting CRVs from specific layers (e.g., every n-th layer)
- Choosing CRVs based on a relevance metric
- Applying a dimensionality reduction technique (e.g., PCA) to the entire CRV stack

We use the first option as the g function to select the early, middle, and late layers of the model.

3.2.5 Integrating CRVs with Hidden States

The integration with the input embedding is formulated as:

$$h'_i = [\text{CRV}_i; h_i] \tag{3.8}$$

where:

- CRV_i is the i-th CRV of the generated context from the thinking stage
- h_i is the i-th hidden state for the generation stage
- h'_i is the new hidden state after integration
- $[;]$ denotes concatenation

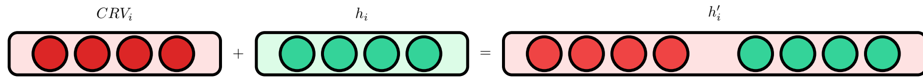

Fig. 3.4 This diagram shows the pre-concatenation process of the new sequence and the original elements of the hidden states.

In the event more than one CRV layers were used, we can generalize the integration approach, and the final hidden states will be represented as follows:

Let $\mathscr{M}$ be the set of all model layers, where $|\mathscr{M}| = n$.

We define $\mathscr{I} \subseteq \mathscr{M}$ as the set of layers where integration occurs.

Let $\mathscr{C} = \{C^1, C^2, ..., C^k\}$ be the set of CRV stacks, where each C^j is a tensor of shape (n, d, l_j), with d being the hidden state dimension and l_j the sequence length of the j-th context.

We define the integration function Ψ for layer i as:

$$\Psi_i : \mathbb{R}^{d \times l_0} \left(\biguplus_{j=1}^{k} \mathbb{R}^{d \times l_j} \right) \to \mathbb{R}^{d \times (l_0 + \Sigma_{j=1}^{k} l_j) 1} \tag{3.9}$$

where l_0 is the original sequence length and $\biguplus_{j=1}^{k}$ represent the concatenation.

Let $\mathscr{I} = \{i_1, i_2, ..., i_n\}$ be the set of indices where CRV integrations occur, ordered such that $i_1 < i_2 < ... < i_n$. Now, we can define the hidden state tensor $H^{(i)}$ for layer i as:

[1]The symbol $\biguplus$ denotes the concatenation of hidden states and CRVs. This is particularly appropriate since sequences are essentially positionally encoded sets that are organized by indices and may include repeated elements. Concatenation functions as the union of these sets, maintaining the original order while permitting duplicates.

$$H^{(i)} = \begin{cases} h^{(i)} & \text{if } i \notin \mathscr{I} \text{ and } i < \min(\mathscr{I}) \\ \Psi_i(\{C_i^j : j \in [1,k]\}, h^{(i)}) & \text{if } i \in \mathscr{I} \\ f\left(h^{(i-1)}\right) & \text{otherwise} \end{cases} \tag{3.10}$$

where:

- $h^{(i)}$ is the original hidden state tensor for layer i
- C_i^j is the i-th layer of the j-th CRV stack
- $[\cdot,\cdot]$ denotes tensor concatenation along the sequence dimension

The sequence length $L^{(i)}$ at layer i is given by:

$$L^{(i)} = \begin{cases} l_0 & \text{if } i < \min(\mathscr{I}) \\ l_0 + \sum_{j=1}^{k} l_j & \text{otherwise} \end{cases} \tag{3.11}$$

The final shape of the hidden states of the model for the first round of integration can be represented as follows:

The sequence of hidden states can be represented as

$$(h_1^{(0)}, h_2^{(0)}, ..., h_{i_1-1}^{(0)}, h_{i_1}^{(1)}, h_{i_1+1}^{(1)}, ..., h_{i_2-1}^{(1)}, h_{i_2}^{(2)}, ..., h_{i_n}^{(n)}, h_{i_n+1}^{(n)}, ...) \tag{3.12}$$

For instance, if integrations happen at $\mathscr{I} = \{5,7\}$, we can represent the sequence of hidden states as

$$(h_1, h_2, h_3, h_4, h_5^{(1)}, h_6^{(1)}, h_7^{(2)}, h_8^{(2)}, \dots) \tag{3.13}$$

where the superscript indicates the number of CRV integrations that have occurred up to that position. Algorithm 2 shows the algorithmic process of integrating CRVs.

Query Reception

The LLM receives the query Q:

$$Q = (q_1, q_2, ..., q_n) \tag{3.14}$$

where n is the length of the query.

Algorithm 2 Integrate CRVs

Require:

h: Current hidden states for all layers
I: Set of indices for layers with full updates
C: Context vectors for all layers
l: Length of the prefix to be preserved

Ensure:

H: Updated hidden states for all layers

```
1:  procedure IntegrateCRVs(h, I, C, l)
2:      H ← h
3:      for all i ∈ [1, num_layers] do
4:          if i ∉ I and i < min(I) then
5:              H_i ← h_i
6:          else if i ∈ I then
7:              H_i ← IntegrateCRVAtLayer(C_i, h_i)
8:          else
9:              prefix ← IntegrateCRVAtLayer(C_i, h_i[: l])[l :]
10:             updatedSuffix ← H_{i−1}[: l]
11:             H_i ← Concatenate(prefix, updatedSuffix)
12:         end if
13:     end for
14:     return H
15: end procedure
```

Positional Encoding Recalculation

The positional encoding is recalculated for the new sequence from layer l onward: For a given position m and dimension d, the RoPE encoding is defined as:

$$R_{\Theta,n,d} = \begin{pmatrix} \cos(n\theta_d) & -\sin(n\theta_d) \\ \sin(n\theta_d) & \cos(n\theta_d) \end{pmatrix} \tag{3.15}$$

where $\theta_d = 10000^{-2d/D}$, and D is the total number of dimensions.

Application to Query and Key Vectors

For a query vector q and a key vector k at position m, RoPE is applied as follows:

$$q' = (q_1, q_2, \ldots, q_D) \cdot R_{\Theta,m} \tag{3.16}$$

$$k' = (k_1, k_2, \ldots, k_D) \cdot R_{\Theta,m} \tag{3.17}$$

where $\cdot$ denotes element-wise multiplication, and $R_{\Theta,m}$ is applied to each pair of dimensions.

Recalculation for Extended Sequence

After CRV integration, for the extended sequence of length $n+m$, where n is the original sequence length and m is the length of the new input:

$$R'_{\Theta,i,d} = \begin{pmatrix} \cos(i\theta_d) & -\sin(i\theta_d) \\ \sin(i\theta_d) & \cos(i\theta_d) \end{pmatrix}, \quad i = 0, \ldots, n+m \tag{3.18}$$

Updated Query and Key Computation

For the new positions in the extended sequence:

$$q'_i = (q_{i,1}, q_{i,2}, \ldots, q_{i,D}) \cdot R'_{\Theta,i} \tag{3.19}$$

$$k'_i = (k_{i,1}, k_{i,2}, \ldots, k_{i,D}) \cdot R'_{\Theta,i} \tag{3.20}$$

where $i = 0, \ldots, n+m$.

Attention Mask Recalculation

Let $M_o \in \{0,1\}^{B\times L_o}$ be the original attention mask, where B is the batch size and L_o is the original sequence length. After concatenation with the CRVs, we define the updated attention mask M_u as follows:

$$M_u = \Phi(M_o, H) \in \{0,1\}^{B\times L_u\times L_u} \tag{3.21}$$

where $H \in \mathbb{R}^{B\times L_u\times D}$ represents the hidden states after concatenation, L_u is the updated sequence length, and D is the hidden state dimension. The function Φ is defined as:

$$\Phi(M_o, H) = \begin{cases} M_a \oplus M_o & \text{if not causal} \\ (M_a \oplus M_o) \odot C & \text{if causal} \end{cases} \tag{3.22}$$

Here:

- $M_a \in \{1\}^{B\times(L_u-L_o)}$ is the additional mask for CRVs
- $\oplus$ denotes concatenation along the sequence dimension
- $\odot$ represents element-wise multiplication
- $C \in \{0,1\}^{L_u\times L_u}$ is the causal mask defined as:

$$C_{ij} = \begin{cases} 1 & \text{if } i \geq j \\ 0 & \text{otherwise} \end{cases} \tag{3.23}$$

The dimensions of the tensors involved are:

$$M_o \in \{0,1\}^{B\times L_o} \tag{3.24}$$
$$M_a \in \{1\}^{B\times(L_u-L_o)} \tag{3.25}$$
$$M_a \oplus M_o \in \{0,1\}^{B\times L_u} \tag{3.26}$$
$$C \in \{0,1\}^{L_u\times L_u} \tag{3.27}$$

The final updated mask M_u as shown in 3.5 is obtained by:

$$M_u = \begin{cases} (M_a \oplus M_o) & \text{if not causal} \\ (M_a \oplus M_o)^T \odot C & \text{if causal} \end{cases} \tag{3.28}$$

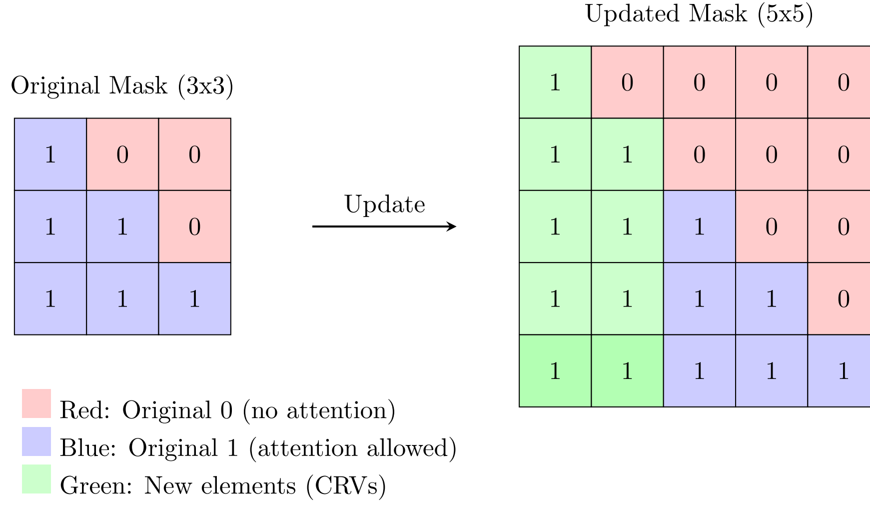

Fig. 3.5 Updating the causal attention mask with respect to the concatenated sequence

where ^{T} denotes tensor transposition to match the dimensions for element-wise multiplication with C.

This formulation ensures that:

1. The CRVs (represented by M_a) are attended to.
2. The original attention pattern (represented by M_o) is preserved.
3. In the causal case, the mask maintains the property that position i can only attend to positions $j \leq i$.

Key-Value Cache Update

The cached key-values are updated accordingly:

$$K_l' = [K_l; f_k(h'_{0:n+m})] \tag{3.29}$$

$$V_l' = [V_l; f_v(h'_{0:n+m})] \tag{3.30}$$

where:

- K_l, V_l are the original key and value matrices for layer l
- K_l', V_l' are the updated key and value matrices
- f_k, f_v are the key and value projection functions
- n is the length of the original sequence

- m is the length of the context generated from t (thinking stage)

3.2.6 Generation Stage

The generation stage can be described by:

$$y_t = f_g(h_t, \text{CRV}_{\text{reduced}}) \tag{3.31}$$

$$h_{t+1} = f_u(h_t, y_t, \text{CRV}_{\text{reduced}}) \tag{3.32}$$

where:

- f_g is the generation function
- f_u is the hidden state update function
- y_t is the output at time step t
- h_t is the hidden state at time step t

3.2.7 Overall Process

The entire process can be summarized as:

$$Y = F(Q, \{\text{CRV}_i\}_{i=1}^{n}) \tag{3.33}$$

where:

- F represents the entire model including thinking, integration, and generation stages
- Q is the input sequence
- Y is the output sequence

Pseudocode

The pseudocode of the entire process to implement NIR algorithm is described in Algorithm 3.

Updated Generation Process

The generation process now incorporates these changes:

$$y_t = F(Q, h', PE', M', K', V') \tag{3.34}$$

where F represents the entire updated model including the integration stage and subsequent adjustments. We can see the complete algorithmic process in Algorithm 3.

Algorithm 3 NIR Process Algorithm

Require:
- Q: Input query
- θ: Model parameters
- h: Initial hidden state
- g: CRV reduction function
- I: Set of layers for CRV integration
- M: Initial attention mask
- l: Sequence length
- K, V: Initial key and value matrices

Ensure:
- Y: Generated output sequence

```
1:  procedure NIR(Q, θ, h, g, I, M, l, K, V)
2:      t ← T(Q, θ)
3:      CRV ← GENERATE_CRV(t, h)
4:      CRV_stack ← CREATE_CRV_STACK(CRV)
5:      CRV_reduced ← REDUCE_CRV_DIMENSION(CRV_stack, g)
6:      H ← INTEGRATE_MULTIPLE_CRVS(CRV_reduced, H, I)
7:      R ← RECALCULATE_POSITIONAL_ENCODING(n, d)
8:      M_prime ← UPDATE_ATTENTION_MASK(M, l)
9:      Y ← ∅
10:     for t = 1 to max_sequence_length do
11:         (y, H) ← GENERATE(CRV_reduced, h)
12:         Y.append(y)
13:         if IS_END_OF_SEQUENCE(y) then
14:             break
15:         end if
16:     end for
17:     return Y
18: end procedure
```

Chapter 4

Methodology

4.1 Dataset Selection and Preprocessing

4.1.1 Mostly Basic Python Programming (MBPP)

The MBPP (Mostly Basic Python Programming) has been used for benchmarking the proposed framework. The dataset is designed to assess the ability of the model to generate Python code based on natural language descriptions. This dataset is the subset dataset that is part of the evaluation results for the Meta-Llama-3.1-8B-Instruct model. The details of the dataset are described in Appendix A and the example entry is represented in Table A.1 under Appendix A.0.1.

4.2 Quantitative Metrics

In this part, we will explore the important metrics that help us evaluate how well the Neural Integration of Iterative Reasoning (NIR) framework performs. These metrics provide a comprehensive way to measure the accuracy, structure, and functionality of the generated code.

4.2.1 Response Rate

When evaluating large language models (LLMs) for code generation, one important metric we look at is the response rate. This metric tells us the percentage of tasks where the model successfully generated a full response without running into problems like timeouts or internal errors. It's important to note, however, it does not mean that the generated code is correct; therefore, it is always paired with other metrics.

$$\text{Response Rate} = \frac{\text{Number of Responses Generated}}{\text{Total Number of Prompts}} \times 100$$

Where:

- *Number of Responses Generated*: This refers to how many code responses the LLM produced, regardless of whether they were correct or complete.
- *Total Number of Prompts*: The total number of prompts or tasks the LLM was given to generate code for.

4.2.2 Code Quality Metrics

Function Name Consistency

Function name consistency refers to using clear, meaningful, and consistent names for functions in the code. A good function name should describe the task or purpose of the function, making it easy for others (and yourself) to understand what the function does just by looking at its name. Following common naming conventions, such as using ‘snake_case‘ for Python function names, is also important for readability and maintainability.

Inconsistent and unclear function names

```
# Bad Example: The function name is unclear and does not
follow Python's naming conventions.
def DoSTUFF(a, b):
    return a + b
def f(x):
    print(x)
```

Consistent and meaningful function names

```
# Good Example: The function names are descriptive and follow
Python's naming conventions.
def add_numbers(a, b):
    return a + b
def print_message(message):
    print(message)
```

Fig. 4.1 Function Name Consistency

Syntactic Correctness

It refers to whether the generated code follows the rules and structure of the programming language it is written in. Essentially, it checks if the code is "written correctly" according to the syntax of the language. If the code has missing semicolons, misused parentheses, or other syntax errors, it would fail to be syntactically correct. It is crucial to understand that even if the logic of our code is correct, it will not run properly if there are syntax errors.

$$\text{Syntactic Correctness} = \begin{cases} 1 & \text{if there are no syntax errors} \\ 0 & \text{if there are syntax errors} \end{cases}$$

Cyclomatic Complexity

Cyclomatic complexity is an important software metric that helps us understand how complex a program really is. It does this by counting the number of linearly independent paths through the code. In simpler terms, it shows us how many decisions or branches exist in our code, giving us insight into how challenging it might be to test, debug, or maintain.

To calculate cyclomatic complexity, we use something called a control flow graph, where:

- E: The number of edges (or transitions between nodes).
- N: The number of nodes (which are decision points or instructions).
- P: The number of connected components (independent parts of the code).

The formula for cyclomatic complexity looks like this:

$$\text{Cyclomatic Complexity} = E - N + 2P$$

When we see a higher cyclomatic complexity value, it indicates that there are more branches in our code. This usually means the code is more complex and could be harder to maintain. A value of 1 suggests there's just one straightforward path through the code, while values over 10 might signal that it's time for some refactoring. Understanding cyclomatic complexity is crucial because it helps us gauge how testable our code is—each independent path needs testing to ensure everything works as intended.

4.2.3 Code Structure and Complexity Analysis

Basic Code Structure Metrics

Comment Ratio The comment ratio is an important metric which allows us to evaluate how well-commented the code is. By dividing the number of comment lines by the total number of lines, it calculates the proportion of comment to LoC in the generated sample. Generally, a higher comment ratio suggests that our code is well-documented and easier to maintain, however, sometimes if the ratio is too high it negates its effect and makes the code difficult to understand especially when the solution is straightforward.

The formula for calculating comment ratio is:

$$\text{Comment Ratio} = \frac{\text{Number of Comment Lines}}{\text{Total Lines of Code}} \times 100$$

where:

- *Number of Comment Lines*: This counts all the lines that contain comments.
- *Total Lines of Code*: This includes both executable lines and comment lines.

Lines of Code The LoC number is another metric that includes both actual codes and comments. Although, it can roughly indicate the size of the solution, it is generally better to be used in conjunction with other metrics to provide a more reliable measurement. Understanding this metric is crucial as it gives us a sense of how efficient and clear our model's output is.

Furthermore, shorter code that effectively accomplishes its task typically indicates greater efficiency compared to longer code that does the same job. However, it is important to highlight the fact that extremely short code snippets could potentially lead to complexity and lack of clarity. So we need to find an appropriate level where our code is concise enough to be readable, but not so compact that it becomes hard to maintain.

Number of Characters The next metric we calculate is the number of characters which counts all the characters in the generated code snippet. Generally, its characteristics are similar to those of LoC.

Halstead Metrics

Another metric that we will evaluate our framework based on it, is a stack of a range of various related metrics that measure the program's complexity and maintainability called

Halstead Metrics. These metrics include vocabulary, length, volume, difficulty, and effort which are calculated based on the number of operators and operands in the code snippet.

As mentioned above the main components of Halstead Metrics are as follows:

- H_1 - The number of unique operators in the code.
- H_2 - The number of unique operands in the code.
- N_1 - The total number of occurrences of operators.
- N_2 - The total number of occurrences of operands.

From these components, the following Halstead metrics can be derived:

Vocabulary The vocabulary of the program is simply the sum of the number of unique operators and operands:

$$\text{Vocabulary} = H_1 + H_2$$

Length The length of the program is calculated as the total occurrences of operators and operands:

$$\text{Length} = N_1 + N_2$$

Volume The volume of the program represents the size of the implementation and is calculated as:

$$\text{Volume} = \text{Length} \times \log_2(\text{Vocabulary})$$

Difficulty The difficulty of understanding the code is given by:

$$\text{Difficulty} = \frac{H_1}{2} \times \frac{N_2}{H_2}$$

Effort From the above calculated metrics, we can then calculate the effort which is a measure of the cognitive effort required to understand or write the code:

$$\text{Effort} = \text{Volume} \times \text{Difficulty}$$

These metrics provide quantitative insights about the complexity of the code, which is an estimation of the effort required to maintain, and assessing the overall readability and maintainability of the codebase.

4.3 Qualitative Analysis

Qualitative metrics provides a crucial insights regarding the performance of NIR approach by providing a more detailed understanding of the generated code's quality, the model's reasoning process and its limitations. There are multiple ways to judge and evaluate the qualitative metrics like its code structure and readability, its understanding and implementation and errors patterns and limitations.

4.3.1 Code Structure and Readability

This metric is primarily utilized to assess the structure and overall readability of the code generated by the LLM using NIR approach.

This includes proper indentation, that plays a crucial role in Python where it impacts the logic. Secondly, it also assesses the quality of the comments by the generated code, on how well the comments are documents according to its relevance, information and placement. Additionally, code readability is also assessed by the size of functions; as long functions are harder to understand and maintain. Lastly, well-named variables improves code readability and structure, this metric evaluates whether variable name follow a proper consistent and meaningful naming convention.

4.3.2 Algorithm Understanding and Implementation

This section focuses on evaluation of the correct understanding and implementation of algorithms, to determine if LLM's are able to generate a logically sound, complex and efficient code.

To measure this, the correctness of algorithms is a must. Whether the generated algorithm correctly implements the task specified in the prompt; moreover efficiency plays an important role where it is required to measure how well the algorithm optimizes time and space complexity. LLM's must be able to handle edge cases, where there are empty inputs or boundary conditions. Lastly, it should avoid unnecessary complexity and measure whether LLM chooses to generate an optimal algorithm for the required problem.

4.3.3 Error Patterns and Limitations

Another way to evaluate the performance of the generated code is by identifying and analyzing common error patterns and limitations in the code using NIR approach.

Detecting error includes, the measure of frequency of syntax, and logical errors in the generated code; logic errors may occur where the code is syntactically correct but does not perform the desired task. Additionally, to track the instances where LLMs generates an incomplete code, and stops mid-function or miss critical elements of the desired task. Lastly, to evaluate if the LLMs are overfitting, this can be measured if the generated code closely resembles examples from the training set, and fails to generalize.

Chapter 5

Experiments and Results

5.1 Experimental Setup

This section details the configuration of our Neural Injection Reasoning (NIR) framework, the hardware and software specifications used for our experiments, and the intermediate processing steps employed in our study.

5.1.1 The NIR Framework Configuration

Our implementation of the NIR framework is based on the LLaMA 3.1 model, specifically the 8 billion parameter version. Table 5.1 summarizes the key configuration details.

Table 5.1 NIR Framework Configuration

Parameter	**Value**
Base Model	LLaMA 3.1
Model Size	8 billion parameters
Architecture	Decoder-only Transformer
Max New Tokens (Reasoning)	1,000
Max New Tokens (Generation)	250
CRV Integration Layers	1, 10, 23
CRV Max Length	4,096 tokens
Precision	16-bit

We chose this configuration for its balance between computational efficiency and performance. The integration of Context Representation Vectors (CRVs) at layers 1, 10, and 23 allows us to examine the impact of CRV injection at early, middle, and late stages of the model's processing pipeline.

5.1.2 Hardware and Software Specifications

Our experiments were conducted on a Linux-based system accessed via SSH. Figure 5.1 illustrates our hardware and software setup.

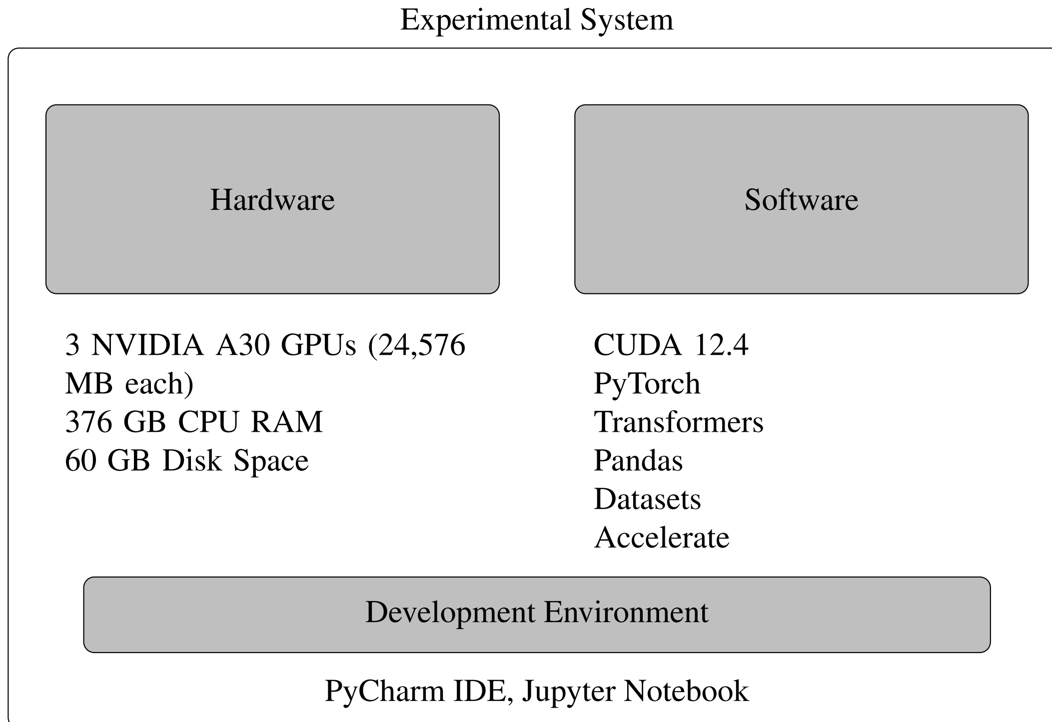

Fig. 5.1 Hardware and Software Specifications

5.1.3 Intermediate Processing Steps

Our experimental pipeline involves several key processing steps, as illustrated in Figure 5.2.

For our benchmarking experiments, we used a subset of 250 examples from our dataset. The dataset includes instructions used by Meta for testing LLaMA 3.1, modified and augmented by the Meta LLaMA 3.1 team. We made several modifications to the base implementation of the Transformer library, affecting various classes as listed in Appendix A.1. These modifications were implemented one at a time throughout our experimentation process. The pretrained weights for our model were obtained from Meta through the Hugging Face platform. Our sampling configuration is detailed in Table 5.2.

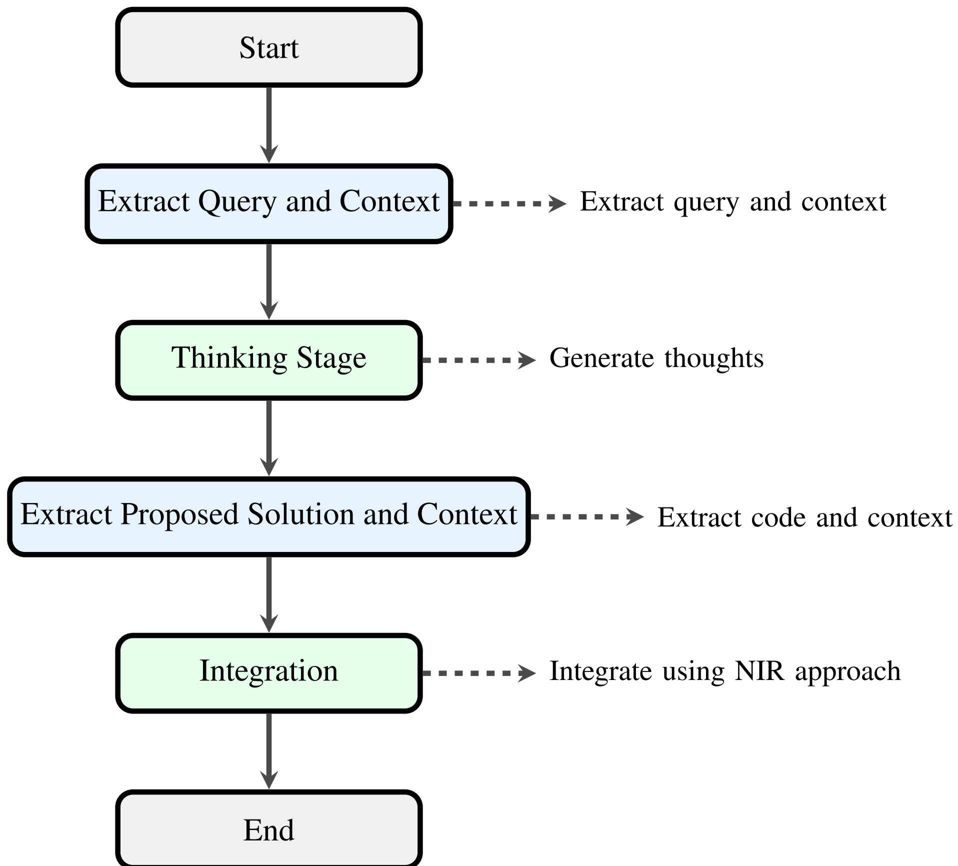

Fig. 5.2 Intermediate Processing Steps

The sampling configuration for our experiments is outlined in Table 5.2. We set the temperature to 0.8, which balances creativity and coherence in the generated outputs. The do_sample parameter is set to True, enabling the model to generate diverse responses. We generate one sequence per input to maintain consistency across our evaluations.

Table 5.2 Sampling Configuration

Parameter	**Value**
Temperature	0.8
do_sample	True
Sequences per input	1

5.2 Quantitative metrics

5.2.1 Response Rate

The percentage of the model's ability to generate outputs for the given queries is indicated by the response rate where higher values are generally considered better. Table 5.3 presents the response rates produced throughout different layer integrations of the NIR framework, as well as the original model configuration.

Table 5.3 Response Rates Across Different Layers

Metric	Layer 1	Layer 10	Layer 23	Original
Response Rate	0.5799	0.9941	0.9941	0.9941
Sample Size	169.0000	169.0000	169.0000	169.0000

Several patterns could be observed from the table 5.3 such as a notably lower response rate for Layer 1 at 57.99%. Layers 10 and 23, as well as the original configuration, all demonstrate a higher response rate of 99.41%. The sample size remains consistent at 169 across all configurations. These findings indicate a significant difference in response rate between Layer 1 and the other configurations. The higher layers of the NIR framework (10 and 23) match the performance of the original model configuration in terms of response rate.

However, it is important to note that, while a high response rate is desirable, it does not necessarily mean a high quality or correctness of the generated code. Therefore, it is crucial to take this metric into consideration in conjunction with other code quality metrics to thoroughly understand the performance of the model.

The following section presents the code quality metrics to assess the quality of the generated code.

5.2.2 Code Quality Metrics

Function Name Consistency

Function name consistency measures how well the generated function names align with the given query and test cases. Higher scores indicate a better understanding of the task. From Table 5.4, we can see that integrating CRVs at Layer 1 achieves a function name consistency of 95.74%. Layer 10 demonstrates perfect consistency with a score of 100%. Layer 23 shows a slight decrease to 98.21%, while the original configuration achieves 99.40% consistency.

These findings indicate that the NIR framework maintains high function name consistency across all layers, with Layer 10 showing the highest performance in this metric.

Syntactic Correctness

An important part of code quality is syntactic correctness which is a core concept for generating compilable code. Table 5.4 provides details about the syntactic correctness of the generated code by NIR framework as well as the outputs by the model with origianl configurations.

Table 5.4 Code Quality Metrics Across Different Layers

Metric	Layer 1	Layer 10	Layer 23	Original
Syntactic Correctness	0.1915	0.8690	0.9762	0.9762
Function Name Consistency	0.9574	1.0000	0.9821	0.9940

From Table 5.4, we can observe that integrating CRVs at Layer 1 results in a syntactic correctness of 19.15%. This score increases significantly to 86.90% when integrating CRVs at Layer 10. The syntactic correctness increases to 97.62% at Layer 23, which is identical to the score achieved by the original model configuration which is the standard prompting technique in the text space. The results show a clear progression in syntactic correctness from earlier to later layers of the model. The significant improvement from Layer 1 to Layer 10, and the further increase to Layer 23, indicates a progressive performance to generate syntactically correct code through the model's layers.

Next, we will present various other metrics to help understanding the performance of the NIR architecture.

Cyclomatic Complexity

Table 5.5 presents the average cyclomatic complexity scores for code generated by the NIR framework at different layers, as well as the original model configuration.

Table 5.5 Code Complexity Metrics Across Different Layers

Metric	Layer 1	Layer 10	Layer 23	Original
Cyclomatic Complexity	0.4468	2.0714	2.5952	2.3571

The results show that integrating CRVs at Layer 1 produces code with an average cyclomatic complexity of 0.4468. When integrating CRVs at Layer 10, we observe a substantial increase in complexity to 2.0714. At Layer 23, the complexity further increases to 2.5952. The original model configuration yields an average complexity of 2.3571. These findings indicate variations in code complexity across different layers of the model. The lower complexity observed at Layer 1 suggests that this layer may generate structurally

simpler code compared to other layers. The complexity scores for Layer 23 and the original configuration are relatively similar, with Layer 23 showing a slightly higher complexity.

Lower values for cyclomatic complexity generally indicate simpler, more maintainable code. However, it is important to note that extremely low values, as seen in Layer 1, might suggest overly simplistic solutions that may not fully address the given tasks. Higher values, while potentially indicating more complex logic, might not be totally unrelated to a comprehensive solution to complex problems.

In the next section we provide insights about the structural characteristics of the generated code using a range of code structure metrics and Halstead complexity measures.

5.2.3 Code Structure and Complexity Analysis

Basic Code Structure Metrics

Table 5.6 presents the average values of various code structure metrics for the code generated by the NIR framework at different layers, as well as the original model configuration.

Table 5.6 Code Structure Metrics Across Different Layers

Metric	Layer 1	Layer 10	Layer 23	Original
Lines of Code	8.2234	11.5893	6.0952	15.7262
No. Characters	217.1596	335.9464	198.0119	535.0417
Comment Lines	0.0745	1.2262	0.3750	3.0714
Comment Ratio	0.0140	0.0852	0.0256	0.1295

The results show variations in code structure across different layers of the model. The average lines of code (LoC) range from 6.0952 in Layer 23 to 15.7262 in the original configuration. Layer 10 produces code with an average of 11.5893 LoC, while Layer 1 generates an average of 8.2234 LoC.

Average comment lines and ratios also vary across layers. The original configuration shows the highest average of comment lines (3.0714) and comment ratio (0.1295), indicating longer code documentation. Layer 10 follows with an average of 1.2262 comment lines and a comment ratio of 0.0852. Layers 1 and 23 show lower values for both metrics.

Overall, the results show a significant variation in code structure and documentation level across different depths of integration. The original configuration tends to produce longer, longer comments, while the NIR framework at different layers generates more code with varying levels of documentation.

In the following sections we will present additional complexity metrics to provide a more comprehensive understanding of the code generated by our proposed framework.

Halstead Complexity Measures

Basic Halstead Metrics Halstead complexity measures provide quantitative metrics for assessing various aspects of code complexity. Table 5.7 presents the averages of basic Halstead metrics for code generated by the NIR framework at different layers, as well as the original model configuration.

Table 5.7 Basic Halstead Metrics Across Different Layers

Metric	Layer 1	Layer 10	Layer 23	Layer orig
h1 (Distinct Operators)	2.0319	3.0357	2.8036	3.7143
h2 (Distinct Operands)	11.3511	20.6726	17.1250	32.7917
N1 (Total Operators)	4.9894	6.5774	6.2500	7.6786
N2 (Total Operands)	30.2128	49.1786	42.2321	83.3274

The results show variations in the basic Halstead metrics across different layers of the model. The number of distinct operators (h1) ranges from 2.0319 in Layer 1 to 3.7143 in the original configuration. Layer 10 and Layer 23 show intermediate values of 3.0357 and 2.8036 respectively. This suggests that the original configuration tends to use a more diverse set of operators in the generated code.

For distinct operands (h2), we observe a similar pattern with the original configuration having the highest value (32.7917), followed by Layer 10 (20.6726), Layer 23 (17.1250), and Layer 1 (11.3511). This indicates that the original configuration and Layer 10 generate code with a richer vocabulary of variables and constants.

The total number of operators (N1) and operands (N2) follow a similar trend, with the original configuration showing the highest values, followed by Layer 10, Layer 23, and Layer 1. This aligns with our previous observations on code length, suggesting that the original configuration generates longer and potentially more complex code.

We should remember that higher values in these metrics don not necessarily mean better code quality. They merely suggest that the code is more diverse in terms of operators and operands used. The interpretation of these metrics should be done in conjunction with other code quality measures.

To further understand the quality of the generated codes based on Halstead metrics, we will examine derived and complexity Halstead metrics in the next two sections.

Derived Halstead Metrics

Table 5.8 presents the derived Halstead metrics for code generated by the NIR framework and the original model configuration to point out additional insights about the complexity and size of the generated code.

Table 5.8 Derived Halstead Metrics Across Different Layers

Metric	Layer 1	Layer 10	Layer 23	Layer orig
Vocabulary	13.3511	23.5655	19.9286	36.5000
Length	43.7660	61.7381	54.9464	97.1905
Volume	170.4025	291.6987	252.0961	511.6619

The vocabulary metric, which is the sum of distinct operators and operands, shows a similar trend to the basic Halstead metrics. The original configuration has the highest vocabulary (36.5000), followed by Layer 10 (23.5655), Layer 23 (19.9286), and Layer 1 (13.3511). This suggests that the original configuration generates code with the most diverse set of operators and operands, while Layer 1 produces the least diverse code.

The length metric, representing the total number of operators and operands, follows a similar pattern. The original configuration generates the longest code (97.1905), followed by Layer 10 (61.7381), Layer 23 (54.9464), and Layer 1 (43.7660). This aligns with our previous observations on code length from the basic code structure metrics.

The volume metric, which combines vocabulary and length to estimate the size of the implementation, shows the most significant differences between layers. The original configuration has the highest volume (511.6619), more than three times that of Layer 1 (170.4025). Layers 10 and 23 show intermediate values of 291.6987 and 252.0961 respectively.

Table 5.8 provides more nuanced insights into the code complexity and it shows that the original configuration of the model generally generates more verbose and potentially more complex codes which is not unlikely to negatively affect the quality of the generated code.

In the next part, we examine the complexity Halstead metrics to provide estimates of the difficulty, effort, and time required to understand and implement the generated code

Complexity Halstead Metrics Table 5.9 presents the complexity Halstead metrics for code generated. These metrics provide estimates of the difficulty and effort required to understand and implement the generated code.

Table 5.9 Complexity Halstead Metrics Across Different Layers

Metric	Layer 1	Layer 10	Layer 23	Layer orig
Difficulty	2.6944	3.8036	3.7718	4.8205
Effort	732.6550	1431.2053	1712.3707	2925.4783

The difficulty metric, which estimates the complexity of the program in terms of error-proneness and understanding, shows a general increase from Layer 1 to the original configuration. Layer 1 has the lowest difficulty (2.6944), while the original configuration has the

highest (4.8205). Layers 10 and 23 show similar intermediate values of 3.8036 and 3.7718 respectively.

The next metric which is the effort, estimates the effort required to develop or maintain the program and follows a similar trend. The original configuration shows the highest effort (2925), which is nearly four times that of Layer 1 (732). Layers 10 and 23 show intermediate values of 1431 and 1712 respectively. This substantial difference suggests that the code generated by the original configuration might require significantly more mental effort to develop or maintain compared with integration of CRVs at different layers in NIR framework.

5.3 Qualitative Analysis

5.3.1 Code Structure and Readability

Examining the code generated by Layer 10 of our NIR framework for the task of checking distinct elements in a tuple, we observe:

```
1 def check_distinct(tup):
2     return len(tup) == len(set(tup))
```

This solution demonstrates a concise and elegant approach to the problem. The use of Python's built-in `set` data structure showcases an understanding of efficient data manipulation. The one-line implementation is both readable and efficient, directly comparing the length of the original tuple with the length of its set representation. Comparing this to the original model's output:

```
1 def check_distinct(numbers):
2     """ Checks if all numbers in the tuple are distinct.
3     Args:
4     numbers (tuple): A tuple of integers.
5     Returns:
6     bool: True if all numbers are distinct, False otherwise.
7     """
8     return len(numbers) == len(set(numbers))
```

We notice that while the core logic remains identical, the original model includes a detailed docstring. This addition enhances code readability and maintainability, especially in a collaborative or long-term project setting. However, for simple functions like this, the NIR framework's output might be preferred in scenarios where conciseness is valued over extensive documentation.

5.3.2 Algorithm Understanding and Implementation

The NIR framework demonstrates a solid grasp of the problem and implements an efficient solution. The thought process revealed in the `trajectories_and_context_10` shows a systematic approach to problem-solving:

1. Problem understanding: "Write a function that checks if all elements in a tuple are distinct."
2. Algorithm consideration: Initially considered a counting-based approach, which was later optimized.
3. Edge case consideration: Empty tuples and single-element tuples were explicitly considered.
4. Solution refinement: The initial counting-based solution was replaced with a more efficient set-based approach.

This thought process indicates that the NIR framework is capable of considering multiple approaches and refining its solution, mimicking human-like problem-solving strategies.

5.3.3 Error Patterns and Limitations

While the generated code is correct and efficient, we observed some limitations in the overall process:

- Output parsing: There were challenges in consistently extracting code snippets from the generated output, which could lead to difficulties in automated evaluation processes.
- Context generation incompleteness: Due to token limitations, sometimes the context generation remained incomplete. This could potentially impact the quality of solutions for more complex problems that require a fuller context understanding.
- Inconsistent output format: The model sometimes deviated from expected output patterns, which could complicate automated testing and integration processes.

These issues, while not directly affecting the code quality in this case, highlight areas for potential improvement in the NIR framework's output consistency and completeness.

5.3.4 Qualitative Analysis Implications

We can point out different key points as a result of the above analysis and evaluations in the previous sections:

1. However the proposed framework appears to generate functional and accurate codes in the samples, to generalize the results of the functionality of the framework, further investigation would be necessary across a range of programming languages and difficulty levels.
2. We observe a potential trade-off between code conciseness and documentation and therefore it might require more in-depth study to find the optimal balance or devising a dynamic approach to selectively decide whether the code needs documentation or not.
3. The framework shows the adherence to the thought processes in both thinking and generation stages. However, additional research might be necessary to thoroughly assess its performance across different tasks.
4. We see certain inconsistencies in the output format and completeness. Addressing these issues could potentially improve the framework's performance, although the extent of this improvement would need to be empirically verified.

It is important to note that these observations are based on a limited set of examples and would require more extensive testing to confirm their generalizability.

5.4 Ablation Studies

Throughout our experiments, we conducted a series of ablation studies that focused on the impact of CRV integration at different layers of the LLaMA 3.1 model. Our Ablation studies evaluate the code quality metrics by integrating CRVs at three different layers: 1, 10, and 23. The original model without CRV integration served as our baseline. All experiments used the MBPP dataset for consistency.

5.4.1 Results

We have provided the results of this study in 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, and 5.9 tables, including the key performance metrics for each ablation condition.

5.4.2 Analysis

The results suggest that CRV integration at different layers has varying effects on model performance:

- Response Rate: As shown in Table 5.3, Layer 1 integration exhibits a notably lower response rate (0.5799) compared to other layers and the original model (all at 0.9941). This suggests that early integration may impede the model's ability to generate responses consistently.
- Code Quality: Table 5.4 indicates that syntactic correctness improves as we move from Layer 1 (0.1915) to Layer 23 (0.9762), matching the original model's performance. Function name consistency is highest at Layer 10 (1.0000), slightly outperforming the original model (0.9940).
- Code Complexity: The cyclomatic complexity (Table 5.5) increases from Layer 1 (0.4468) to Layer 23 (2.5952), with Layer 10 (2.0714) being closest to the original model (2.3571). This suggests that deeper integration may lead to more complex code structures.
- Code Structure: Table 5.6 shows varying trends across layers. Layer 23 generates the most concise code (6.0952 lines), while the original model produces the longest (15.7262 lines). Comment ratios are generally lower in the NIR framework compared to the original model.
- Halstead Metrics: The basic Halstead metrics (Table 5.7) show a general increase from Layer 1 to the original model, indicating more diverse and numerous operators and operands. This trend is shown in the derived Halstead metrics (Table 5.8), with vocabulary, length, and volume all increasing.
- Complexity Halstead Metrics: Table 5.9 shows that difficulty and effort generally increase from Layer 1 to the original model, suggesting that deeper integration and the original model produce more complex code that may require more effort to understand and maintain.

5.4.3 Ablation Studies Implications

These ablation studies reveal several important points:

1. The depth of CRV integration significantly impacts various aspects of code generation, from response rates to code complexity.

2. Layer 1 integration generally underperforms across most metrics, indicating that very early integration may not be optimal for capturing and utilizing contextual information effectively.
3. Layer 10 integration often strikes a balance between the extremes, sometimes outperforming both Layer 23 and the original model in certain metrics (e.g., function name consistency).
4. While deeper integration (Layer 23) and the original model tend to produce more complex code, they also achieve higher syntactic correctness.

These findings suggest that the optimal layer for CRV integration may depend on the specific requirements of the task at hand. For applications prioritizing code simplicity and conciseness, earlier integration (around Layer 10) might be preferable. However, for tasks requiring high syntactic correctness and potentially more complex solutions, later integration or the original model might be more suitable. It's important to note that these results are based on the MBPP dataset and may not generalize to all types of programming tasks. Further research across diverse programming challenges and larger datasets would be necessary to draw more definitive conclusions about the optimal CRV integration strategy.

In the next chapter, we discuss the results and provide insights on the quantitative metrics calculated for our proposed framework on the sample dataset.

Chapter 6

Discussion

6.1 Interpretation of Results

Insights about the performance of our proposed framework across a range of metrics are provided in the experimental results. The integration of Context Representation Vectors (CRVs) at different layers of the LLaMA 3.1 model yielded varying outcomes, which might indicate the necessity for careful interpretation.

6.1.1 Response Rate and Code Quality

The response rate remained consistently high (99.41%) for Layer 10, Layer 23, and the original model configuration, while Layer 1 showed a notably lower rate of 57.99%. This suggests that early integration of CRVs might result in the model's inability to generate responses consistently. This disruption could be attributed to the premature injection of high-level contextual information before the model has had the chance to form its own basic representations of the input. However, the high response rates for deeper layers show that the NIR framework can maintain the model's generative capabilities in these configurations.

Regarding code quality, syntactic correctness improved progressively from Layer 1 (19.15%) to Layer 23 (97.62%), matching the original model's performance. This trend suggests that deeper integration of CRVs might enhance the model's ability to generate syntactically correct code. Function name consistency reached its peak at Layer 10 (100%), slightly outperforming the original model (99.40%), which could indicate an optimal balance for this particular aspect of code generation. This suggests that mid-level integration reflect a balance between leveraging contextual information and preserving the model's inherent language generation capabilities.

The results regarding the Layer 10 integration may be explained by the hierarchical nature of neural network representations. At this depth, the model has likely formed robust intermediate representations of the input, making it more receptive to the injection of high-level contextual information without overwhelming its processing pipeline.

6.1.2 Code Complexity and Structure

The cyclomatic complexity increased from Layer 1 (0.4468) to Layer 23 (2.5952), with Layer 10 (2.0714) being closest to the original model (2.3571). This trend suggests that deeper CRV integration may lead to more complex code structures, potentially reflecting a more sophisticated understanding of the programming tasks.

Interestingly, as CRV integration moved deeper into the network, the code structure metrics revealed that Layer 23 generated the most concise code (6.0952 lines on average), while the original model produced the longest (15.7262 lines). This could indicate that the NIR framework, especially at deeper layers, allows the model to leverage more sophisticated reasoning patterns.

6.1.3 Implications for Model Architecture and Training

These findings have profound implications for the design and training of language models for code generation:

Adaptive Integration Strategies

The varying performance across layers suggests that an adaptive CRV integration strategy could be beneficial. For instance, a model could dynamically adjust the depth of integration based on the complexity of the programming task at hand. Simple tasks might benefit from shallower integration, while more complex problems could leverage deeper integration for more sophisticated reasoning.

Hierarchical Context Processing

The performance of mid-layer integration (Layer 10) hints at the potential for hierarchical context processing in language models. Future architectures could incorporate explicit mechanisms for processing and integrating contextual information at multiple levels of abstraction, potentially leading to more robust and versatile code generation capabilities.

Balancing Complexity and Conciseness

The inverse relationship between code complexity and conciseness observed in deeper layers (particularly Layer 23) raises interesting questions about the nature of "optimal" code generation. It suggests that models might benefit from training objectives that explicitly balance structural complexity with expression conciseness, potentially leading to more elegant and maintainable code outputs.

6.2 Limitations of the Current Approach

While these results can show us promising hints toward exploring more novel ideas to improve LLMs, there are still several limitations that must be acknowledged:

6.2.1 Model and Dataset Limitations

Dataset

The study primarily used the MBPP dataset, which, while valuable, might not encompass the full range of real-world programming scenarios. To fully assess the NIR framework's effectiveness, it needs to be tested on a wider variety of programming problems and languages.

Generalization to Other Models

The study's results are specific to the LLaMA 3.1 model. To understand how broadly applicable the NIR framework is, further research is needed to explore its effectiveness with other model architectures. This could reveal if the observed patterns hold true across different language models.

Practical Applications

This research mainly focuses on immediate performance metrics. To understand how CRV integration impacts model behavior over time, more studies are necessary to investigate more practical applications. This includes exploring potential fine-tuning strategies that could optimize CRV integration for practical use cases.

6.2.2 Output Inconsistencies

The analysis revealed some inconsistencies in output format and completeness, particularly in parsing code snippets from generated output. This could potentially impact automated evaluation processes and practical applications of the framework.

6.2.3 Context Generation Incompleteness

Due to token limitations, the context generation sometimes remained incomplete. This could affect the quality of solutions for more complex problems that require a fuller context understanding.

6.2.4 Trade-offs in Code Characteristics

The results indicate potential trade-offs between code conciseness, complexity, and documentation. While the NIR framework often generated more concise code, it also tended to produce less documentation compared to the original model. The optimal balance between these characteristics may vary depending on specific use cases and preferences.

In the next chapter we provide a conclusion over this research and mention potential future research directions to further investigate this approach.

Chapter 7

Conclusion and Future Work

7.1 Summary of key findings

We propossed a new LLM framework called Neural Integration of Iterative Reasoning (NIR) that integrates context representation vectors (CRV) at different layers of the LLaMA 3.1 model to guide the generation process of the model and potentially enhance reasoning. We summarize our findings as follows.

- The clear performance in the response rates between Layer 1 (57.99%) and deeper layers (99.41%) suggests a critical threshold in early processing stages where contextual information can be effectively integrated without disrupting the model's generative capabilities.
- The constant improvement shown in the evalutation of syntactic correctness from Layer 1 (19.15%) to Layer 23 (97.62%) indicates a potential correlation between deeper integration and the model's ability to adhere to language-specific syntax rules.
- The peak in function name consistency at Layer 10 (100%), which slightly outperforms the original model, hints at an optimal depth for semantic understanding in the context of naming conventions. This is probably due to the fact that naming convention representations could be captured in the early layers of the model as they require less cognitive effort to understand and are easier to capture.
- The inverse relationship between code conciseness and integration depth, with Layer 23 producing the most concise code (6.0952 lines on average) while maintaining other metrics rather good or roughly equal compared to other layers, suggests that deeper integration might enable more abstract reasoning and efficient solution expression.

Based on these results, we find that the depth of CRV integration significantly impacts various aspects of code generation, from response rates to code complexity and structure.

7.2 Implications for LLM development

The results of this study offer potential useful insights that could be considered in the future development of Large Language Models in code generation:

7.2.1 Adaptive Architecture Design

By dynamically adjusting integration depth of CRVs based on task complexity, LLMs could become more efficient and context-aware. This adaptive approach would allow models to optimally allocate computational resources for different tasks.

7.2.2 Hierarchical Contextual Processing

The model's performance during mid-layer integration, particularly at Layer 10, hints that it is worth stuying the explicit hierarchical contextual processing modeling in LLMs. Future architectures might use distinct processing stages for different levels of abstraction, such as syntactic details and higher-level semantic concepts.

7.3 Potential future research directions

7.3.1 Future Research Directions

In section 5.2 we calculated quantitative metrics for our proposed framework which in turn raises some interesting research questions. Here we summarize the most important ideas for further exploration.

Qeustions

- Multi-layer Integration: Exploring the effects of simultaneous CRV integration at multiple layers could uncover more sophisticated integration strategies.
- Study the effects of integrating the CRVs in multiple rounds.
- Dynamic Integration: Developing mechanisms for dynamically adjusting the depth and intensity of CRV integration based on task complexity and other contextual factors.

- Explore the outcomes of maintaining a memory manager which can integrate or remove CRVs at arbitrary iterations and generation steps.
- How would the pooling over the neighboring CRV layers affect the model performance?
- How can we change the thinking stage and integrate it into the model architecture rather than mere prompt engineering?
- Assessing the framework's performance based on pass@1
- A more comprehensive evaluation of the NIR framework across a diverse range of programming tasks.
- Apply the framework to other pre-trained LLMs such as Qwen-72B-Instruct, Phi-3, and similar open-source LLMs.

We conclude our study by reiterating the core concepts of our findings that show potential for improving code generation, but also highlights the need for balance among different performance metrics. Our findings may contribute to understanding context use in LLMs and could lead to developing more advanced coding assistants.

References

Barnett, S., Kurniawan, S., Thudumu, S., Brannelly, Z., and Abdelrazek, M. (2024). Seven Failure Points When Engineering a Retrieval Augmented Generation System. arXiv:2401.05856 [cs] version: 1.

Beltagy, I., Peters, M. E., and Cohan, A. (2020). Longformer: The Long-Document Transformer. arXiv:2004.05150 [cs].

Black, S., Biderman, S., Hallahan, E., Anthony, Q., Gao, L., Golding, L., He, H., Leahy, C., McDonell, K., Phang, J., Pieler, M., Prashanth, U. S., Purohit, S., Reynolds, L., Tow, J., Wang, B., and Weinbach, S. (2022). GPT-NeoX-20B: An Open-Source Autoregressive Language Model. arXiv:2204.06745 [cs].

bloc97 (2023). NTK-Aware Scaled RoPE allows LLaMA models to have extended (8k+) context size without any fine-tuning and minimal perplexity degradation.

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.

Chae, H., Kim, Y., Kim, S., Ong, K. T.-i., Kwak, B.-w., Kim, M., Kim, S., Kwon, T., Chung, J., Yu, Y., and Yeo, J. (2024). Language Models as Compilers: Simulating Pseudocode Execution Improves Algorithmic Reasoning in Language Models. arXiv:2404.02575 [cs].

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., 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. arXiv:2107.03374 [cs].

Chen, S., Wong, S., Chen, L., and Tian, Y. (2023). Extending Context Window of Large Language Models via Positional Interpolation. arXiv:2306.15595 [cs].

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, 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., 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., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., 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., 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., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, 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., 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., 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., 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., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X., Tan, X. E., 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., Grattafiori, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Vaughan, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Franco, A., Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., De Paola, B., 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., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Wyatt, 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., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Ozgenel, F., Caggioni, F., Guzmán, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Thattai, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Damlaj, I., Molybog, I., Tufanov, I., Veliche, I.-E., Gat, I., Weissman, J., Geboski, J., Kohli, 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., Prasad, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Huang, K., Chawla, K., Lakhotia,

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., Tsimpoukelli, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, 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., Laptev, N. P., Dong, N., Zhang, 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., Li, R., Hogan, R., Battey, R., Wang, R., Maheswari, R., Howes, R., Rinott, R., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., 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., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Kohler, 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., Albiero, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wang, X., Wu, X., Wang, X., Xia, X., Wu, X., Gao, X., Chen, Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu, Wang, Hao, Y., Qian, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., and Zhao, Z. (2024). The Llama 3 Herd of Models. arXiv:2407.21783 [cs].

emozilla (2023). Dynamically Scaled RoPE further increases performance of long context LLaMA with zero fine-tuning.

Fu, Y., Peng, H., Sabharwal, A., Clark, P., and Khot, T. (2022). Complexity-Based Prompting for Multi-step Reasoning.

Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. (2017). Convolutional Sequence to Sequence Learning. arXiv:1705.03122 [cs].

Kambhampati, S., Valmeekam, K., Guan, L., Verma, M., Stechly, K., Bhambri, S., Saldyt, L. P., and Murthy, A. B. (2024). Position: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. In *Proceedings of the 41st International Conference on Machine Learning*, pages 22895–22907. PMLR. ISSN: 2640-3498.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.-H., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D.,

Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. (2023). StarCoder: may the source be with you! arXiv:2305.06161 [cs].

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., d'Autume, C. d. M., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. (2022). Competition-Level Code Generation with AlphaCode. *Science*, 378(6624):1092–1097. arXiv:2203.07814 [cs].

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., Bello, I., Berdine, J., Bernadett-Shapiro, G., Berner, C., Bogdonoff, L., Boiko, O., Boyd, M., Brakman, A.-L., Brockman, G., Brooks, T., Brundage, M., Button, K., Cai, T., Campbell, R., Cann, A., Carey, B., Carlson, C., Carmichael, R., Chan, B., Chang, C., Chantzis, F., Chen, D., Chen, S., Chen, R., Chen, J., Chen, M., Chess, B., Cho, C., Chu, C., Chung, H. W., Cummings, D., Currier, J., Dai, Y., Decareaux, C., Degry, T., Deutsch, N., Deville, D., Dhar, A., Dohan, D., Dowling, S., Dunning, S., Ecoffet, A., Eleti, A., Eloundou, T., Farhi, D., Fedus, L., Felix, N., Fishman, S. P., Forte, J., Fulford, I., Gao, L., Georges, E., Gibson, C., Goel, V., Gogineni, T., Goh, G., Gontijo-Lopes, R., Gordon, J., Grafstein, M., Gray, S., Greene, R., Gross, J., Gu, S. S., Guo, Y., Hallacy, C., Han, J., Harris, J., He, Y., Heaton, M., Heidecke, J., Hesse, C., Hickey, A., Hickey, W., Hoeschele, P., Houghton, B., Hsu, K., Hu, S., Hu, X., Huizinga, J., Jain, S., Jain, S., Jang, J., Jiang, A., Jiang, R., Jin, H., Jin, D., Jomoto, S., Jonn, B., Jun, H., Kaftan, T., Kaiser, , Kamali, A., Kanitscheider, I., Keskar, N. S., Khan, T., Kilpatrick, L., Kim, J. W., Kim, C., Kim, Y., Kirchner, J. H., Kiros, J., Knight, M., Kokotajlo, D., Kondraciuk, , Kondrich, A., Konstantinidis, A., Kosic, K., Krueger, G., Kuo, V., Lampe, M., Lan, I., Lee, T., Leike, J., Leung, J., Levy, D., Li, C. M., Lim, R., Lin, M., Lin, S., Litwin, M., Lopez, T., Lowe, R., Lue, P., Makanju, A., Malfacini, K., Manning, S., Markov, T., Markovski, Y., Martin, B., Mayer, K., Mayne, A., McGrew, B., McKinney, S. M., McLeavey, C., McMillan, P., McNeil, J., Medina, D., Mehta, A., Menick, J., Metz, L., Mishchenko, A., Mishkin, P., Monaco, V., Morikawa, E., Mossing, D., Mu, T., Murati, M., Murk, O., Mély, D., Nair, A., Nakano, R., Nayak, R., Neelakantan, A., Ngo, R., Noh, H., Ouyang, L., O'Keefe, C., Pachocki, J., Paino, A., Palermo, J., Pantuliano, A., Parascandolo, G., Parish, J., Parparita, E., Passos, A., Pavlov, M., Peng, A., Perelman, A., Peres, F. d. A. B., Petrov, M., Pinto, H. P. d. O., Michael, Pokorny, Pokrass, M., Pong, V. H., Powell, T., Power, A., Power, B., Proehl, E., Puri, R., Radford, A., Rae, J., Ramesh, A., Raymond, C., Real, F., Rimbach, K., Ross, C., Rotsted, B., Roussez, H., Ryder, N., Saltarelli, M., Sanders, T., Santurkar, S., Sastry, G., Schmidt, H., Schnurr, D., Schulman, J., Selsam, D., Sheppard, K., Sherbakov, T., Shieh, J., Shoker, S., Shyam, P., Sidor, S., Sigler, E., Simens, M., Sitkin, J., Slama, K., Sohl, I., Sokolowsky, B., Song, Y., Staudacher, N., Such, F. P., Summers, N., Sutskever, I., Tang, J., Tezak, N., Thompson, M. B., Tillet, P., Tootoonchian, A., Tseng, E., Tuggle, P., Turley, N., Tworek, J., Uribe, J. F. C., Vallone, A., Vijayvergiya, A., Voss, C., Wainwright, C., Wang, J. J., Wang, A., Wang, B., Ward, J., Wei, J., Weinmann, C. J., Welihinda, A., Welinder, P., Weng, J., Weng, L., Wiethoff, M., Willner, D., Winter, C., Wolrich, S., Wong, H., Workman, L., Wu, S., Wu, J., Wu, M., Xiao, K., Xu, T., Yoo, S., Yu, K., Yuan, Q., Zaremba, W., Zellers, R.,

Zhang, C., Zhang, M., Zhao, S., Zheng, T., Zhuang, J., Zhuk, W., and Zoph, B. (2024). GPT-4 Technical Report. arXiv:2303.08774 [cs].

Peng, B., Quesnelle, J., Fan, H., and Shippole, E. (2023). YaRN: Efficient Context Window Extension of Large Language Models. arXiv:2309.00071 [cs].

Press, O., Smith, N. A., and Lewis, M. (2022). Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. arXiv:2108.12409 [cs].

Qi, Z., Ma, M., Xu, J., Zhang, L. L., Yang, F., and Yang, M. (2024). Mutual Reasoning Makes Smaller LLMs Stronger Problem-Solvers. arXiv:2408.06195 [cs].

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2023). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs, stat].

Shaw, P., Uszkoreit, J., and Vaswani, A. (2018). Self-Attention with Relative Position Representations. In Walker, M., Ji, H., and Stent, A., editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana. Association for Computational Linguistics.

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., and Yao, S. (2023). Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366 [cs].

Sprague, Z., Yin, F., Rodriguez, J. D., Jiang, D., Wadhwa, M., Singhal, P., Zhao, X., Ye, X., Mahowald, K., and Durrett, G. (2024). To CoT or not to CoT? Chain-of-thought helps mainly on math and symbolic reasoning. arXiv:2409.12183 [cs] version: 1.

Srivastava, A., Rastogi, A., Rao, A., Shoeb, A. A. M., Abid, A., Fisch, A., Brown, A. R., Santoro, A., Gupta, A., Garriga-Alonso, A., Kluska, A., Lewkowycz, A., Agarwal, A., Power, A., Ray, A., Warstadt, A., Kocurek, A. W., Safaya, A., Tazarv, A., Xiang, A., Parrish, A., Nie, A., Hussain, A., Askell, A., Dsouza, A., Slone, A., Rahane, A., Iyer, A. S., Andreassen, A., Madotto, A., Santilli, A., Stuhlmüller, A., Dai, A., La, A., Lampinen, A., Zou, A., Jiang, A., Chen, A., Vuong, A., Gupta, A., Gottardi, A., Norelli, A., Venkatesh, A., Gholamidavoodi, A., Tabassum, A., Menezes, A., Kirubarajan, A., Mullokandov, A., Sabharwal, A., Herrick, A., Efrat, A., Erdem, A., Karakaş, A., Roberts, B. R., Loe, B. S., Zoph, B., Bojanowski, B., Özyurt, B., Hedayatnia, B., Neyshabur, B., Inden, B., Stein, B., Ekmekci, B., Lin, B. Y., Howald, B., Orinion, B., Diao, C., Dour, C., Stinson, C., Argueta, C., Ramírez, C. F., Singh, C., Rathkopf, C., Meng, C., Baral, C., Wu, C., Callison-Burch, C., Waites, C., Voigt, C., Manning, C. D., Potts, C., Ramirez, C., Rivera, C. E., Siro, C., Raffel, C., Ashcraft, C., Garbacea, C., Sileo, D., Garrette, D., Hendrycks, D., Kilman, D., Roth, D., Freeman, D., Khashabi, D., Levy, D., González, D. M., Perszyk, D., Hernandez, D., Chen, D., Ippolito, D., Gilboa, D., Dohan, D., Drakard, D., Jurgens, D., Datta, D., Ganguli, D., Emelin, D., Kleyko, D., Yuret, D., Chen, D., Tam, D., Hupkes, D., Misra, D., Buzan, D., Mollo, D. C., Yang, D., Lee, D.-H., Schrader, D., Shutova, E., Cubuk, E. D., Segal, E., Hagerman, E., Barnes, E., Donoway, E., Pavlick, E., Rodola, E., Lam, E., Chu, E., Tang, E., Erdem, E., Chang, E., Chi, E. A., Dyer, E., Jerzak, E., Kim, E., Manyasi, E. E., Zheltonozhskii, E., Xia, F., Siar, F., Martínez-Plumed, F., Happé, F., Chollet, F., Rong, F., Mishra, G., Winata, G. I., de Melo, G., Kruszewski, G., Parascandolo,

G., Mariani, G., Wang, G., Jaimovitch-López, G., Betz, G., Gur-Ari, G., Galijasevic, H., Kim, H., Rashkin, H., Hajishirzi, H., Mehta, H., Bogar, H., Shevlin, H., Schütze, H., Yakura, H., Zhang, H., Wong, H. M., Ng, I., Noble, I., Jumelet, J., Geissinger, J., Kernion, J., Hilton, J., Lee, J., Fisac, J. F., Simon, J. B., Koppel, J., Zheng, J., Zou, J., Kocoń, J., Thompson, J., Wingfield, J., Kaplan, J., Radom, J., Sohl-Dickstein, J., Phang, J., Wei, J., Yosinski, J., Novikova, J., Bosscher, J., Marsh, J., Kim, J., Taal, J., Engel, J., Alabi, J., Xu, J., Song, J., Tang, J., Waweru, J., Burden, J., Miller, J., Balis, J. U., Batchelder, J., Berant, J., Frohberg, J., Rozen, J., Hernandez-Orallo, J., Boudeman, J., Guerr, J., Jones, J., Tenenbaum, J. B., Rule, J. S., Chua, J., Kanclerz, K., Livescu, K., Krauth, K., Gopalakrishnan, K., Ignatyeva, K., Markert, K., Dhole, K. D., Gimpel, K., Omondi, K., Mathewson, K., Chiafullo, K., Shkaruta, K., Shridhar, K., McDonell, K., Richardson, K., Reynolds, L., Gao, L., Zhang, L., Dugan, L., Qin, L., Contreras-Ochando, L., Morency, L.-P., Moschella, L., Lam, L., Noble, L., Schmidt, L., He, L., Colón, L. O., Metz, L., Şenel, L. K., Bosma, M., Sap, M., ter Hoeve, M., Farooqi, M., Faruqui, M., Mazeika, M., Baturan, M., Marelli, M., Maru, M., Quintana, M. J. R., Tolkiehn, M., Giulianelli, M., Lewis, M., Potthast, M., Leavitt, M. L., Hagen, M., Schubert, M., Baitemirova, M. O., Arnaud, M., McElrath, M., Yee, M. A., Cohen, M., Gu, M., Ivanitskiy, M., Starritt, M., Strube, M., Swędrowski, M., Bevilacqua, M., Yasunaga, M., Kale, M., Cain, M., Xu, M., Suzgun, M., Walker, M., Tiwari, M., Bansal, M., Aminnaseri, M., Geva, M., Gheini, M., T, M. V., Peng, N., Chi, N. A., Lee, N., Krakover, N. G.-A., Cameron, N., Roberts, N., Doiron, N., Martinez, N., Nangia, N., Deckers, N., Muennighoff, N., Keskar, N. S., Iyer, N. S., Constant, N., Fiedel, N., Wen, N., Zhang, O., Agha, O., Elbaghdadi, O., Levy, O., Evans, O., Casares, P. A. M., Doshi, P., Fung, P., Liang, P. P., Vicol, P., Alipoormolabashi, P., Liao, P., Liang, P., Chang, P., Eckersley, P., Htut, P. M., Hwang, P., Miłkowski, P., Patil, P., Pezeshkpour, P., Oli, P., Mei, Q., Lyu, Q., Chen, Q., Banjade, R., Rudolph, R. E., Gabriel, R., Habacker, R., Risco, R., Millière, R., Garg, R., Barnes, R., Saurous, R. A., Arakawa, R., Raymaekers, R., Frank, R., Sikand, R., Novak, R., Sitelew, R., LeBras, R., Liu, R., Jacobs, R., Zhang, R., Salakhutdinov, R., Chi, R., Lee, R., Stovall, R., Teehan, R., Yang, R., Singh, S., Mohammad, S. M., Anand, S., Dillavou, S., Shleifer, S., Wiseman, S., Gruetter, S., Bowman, S. R., Schoenholz, S. S., Han, S., Kwatra, S., Rous, S. A., Ghazarian, S., Ghosh, S., Casey, S., Bischoff, S., Gehrmann, S., Schuster, S., Sadeghi, S., Hamdan, S., Zhou, S., Srivastava, S., Shi, S., Singh, S., Asaadi, S., Gu, S. S., Pachchigar, S., Toshniwal, S., Upadhyay, S., Shyamolima, Debnath, Shakeri, S., Thormeyer, S., Melzi, S., Reddy, S., Makini, S. P., Lee, S.-H., Torene, S., Hatwar, S., Dehaene, S., Divic, S., Ermon, S., Biderman, S., Lin, S., Prasad, S., Piantadosi, S. T., Shieber, S. M., Misherghi, S., Kiritchenko, S., Mishra, S., Linzen, T., Schuster, T., Li, T., Yu, T., Ali, T., Hashimoto, T., Wu, T.-L., Desbordes, T., Rothschild, T., Phan, T., Wang, T., Nkinyili, T., Schick, T., Kornev, T., Tunduny, T., Gerstenberg, T., Chang, T., Neeraj, T., Khot, T., Shultz, T., Shaham, U., Misra, V., Demberg, V., Nyamai, V., Raunak, V., Ramasesh, V., Prabhu, V. U., Padmakumar, V., Srikumar, V., Fedus, W., Saunders, W., Zhang, W., Vossen, W., Ren, X., Tong, X., Zhao, X., Wu, X., Shen, X., Yaghoobzadeh, Y., Lakretz, Y., Song, Y., Bahri, Y., Choi, Y., Yang, Y., Hao, Y., Chen, Y., Belinkov, Y., Hou, Y., Hou, Y., Bai, Y., Seid, Z., Zhao, Z., Wang, Z., Wang, Z. J., Wang, Z., and Wu, Z. (2023). Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models. arXiv:2206.04615 [cs, stat].

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., and Liu, Y. (2023). RoFormer: Enhanced Transformer with Rotary Position Embedding. arXiv:2104.09864 [cs].

Sun, Y., Dong, L., Patra, B., Ma, S., Huang, S., Benhaim, A., Chaudhary, V., Song, X., and Wei, F. (2023). A Length-Extrapolatable Transformer. In Rogers, A., Boyd-Graber, J., and Okazaki, N., editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14590–14604, Toronto, Canada. Association for Computational Linguistics.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, u., and Polosukhin, I. (2017). Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. (2023). Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171 [cs].

Wang, Y., Ma, X., Zhang, G., Ni, Y., Chandra, A., Guo, S., Ren, W., Arulraj, A., He, X., Jiang, Z., Li, T., Ku, M., Wang, K., Zhuang, A., Fan, R., Yue, X., and Chen, W. (2024). MMLU-Pro: A More Robust and Challenging Multi-Task Language Understanding Benchmark. arXiv:2406.01574 [cs].

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. H. (2021). CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv:2109.00859 [cs].

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. arXiv:2201.11903 [cs].

Xu, X., Tao, C., Shen, T., Xu, C., Xu, H., Long, G., and Lou, J.-g. (2024). Re-Reading Improves Reasoning in Large Language Models. arXiv:2309.06275 [cs].

Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q. V., and Chi, E. H. (2022). Least-to-Most Prompting Enables Complex Reasoning in Large Language Models.
[heading=bibintoc, title=References]

Appendix A

Dataset Structure

The dataset contains information about the performance of the model on the MBPP task. Each entry in the dataset includes the following columns:

1. **task_type**: This is "Generative" for the MBPP task, as it requires the model to generate code.

2. **task_name**: This is "mbpp" which describes this specific dataset.

3. **input_question**: The natural language description of the programming task.

4. **input_final_prompt**: The complete prompt provided to the model, including additional context or instructions.

5. **input_correct_responses**: The expected correct Python code solution(s) for the given task.

6. **output_prediction_text**: The actual code generated by the Meta-Llama-3.1-8B-Instruct model in response to the prompt.

7. **is_correct**: A boolean value indicating whether output of the model matches the expected solution.

A.0.1 Example Entry

Field	Value
task_type	Generative
task_name	mbpp
input_question	Write a Python function to find the sum of all even numbers in a list.
input_final_prompt	Write a Python function to find the sum of all even numbers in a list. Your function should take a list of integers as input and return the sum of all even numbers in that list.
input_correct_responses	`def sum_even_numbers(numbers): return sum(num for num in numbers if num % 2 == 0)`
output_prediction_text	`def sum_even_numbers(numbers): return sum(num for num in numbers if num % 2 == 0)`
is_correct	true

Table A.1 Example Entry for MBPP Dataset

A.1 Model Modifications

In our implementation of the NIR framework, we made several modifications to the base Transformer library. These changes affected various classes and functions within the LLaMA model architecture. The following list details the specific components that were modified:

1. **LLaMA SDPA Attention**: We adjusted the scaled dot-product attention mechanism to make it compatible with the integration of Context Representation Vectors (CRVs).
2. **LLaMA Attention**: The general attention mechanism was modified to handle the additional context after concatentation of CRVs during the forward pass.
3. **LLaMA Rotary Embedding**: We adapted the rotary embedding function to be updated after CRV injection process.

4. **Apply Rotary Positional Encoding Function**: This function was updated to ensure the correctness of positional encoding after concatenating CRVs.

5. **LLaMADecoderLayer Class**: We modified this class to include the CRV integration step at arbitrary layers and made it compatible with all other modified classes.

6. **LLaMAModel Class**: The main model class was updated to handle the concatenation of CRVs through the network and manage the process.

7. **LLaMAForCausalLM Class**: We adjusted this class to ensure proper handling of CRVs during the causal language modeling task.

These modifications were implemented incrementally while experimenting the NIR framework. The changes allow for the seamless integration of CRVs at different depths of the model, which allows us to study the impact of context injection on the model's performance in code generation tasks.