CAN AI CODE LIKE A HUMAN: A CRITICAL ANALYSIS OF AI'S UNDERSTANDING IN CODE GENERATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY
BY

SAMI AKKUŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF COGNITIVE SCIENCE

APRIL 2024

CAN AI CODE LIKE A HUMAN: A CRITICAL ANALYSIS OF AI'S UNDERSTANDING IN CODE GENERATION

submitted by **SAMI AKKUŞ** in partial fulfillment of the requirements for the degree of **Master of Science in Cognitive Science Department, Middle East Technical University** by,

Prof. Dr. Banu Günel Kılıç
Director, **Graduate School of Informatics**

Assoc. Prof. Dr. Barbaros Yet
Head of Department, **Cognitive Science**

Prof. Dr. Cem Bozşahin
Supervisor, **Cognitive Science, METU**

Examining Committee Members:

Assoc. Prof. Dr. Barbaros Yet
Cognitive Science, METU

Prof. Dr. Cem Bozşahin
Cognitive Science, METU

Assist. Prof. Dr. Murat Ulubay
The School of Management, Ankara Yildirim Beyazit University

Date: 22.04.2024

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Sami Akkuş

Signature :

ABSTRACT

CAN AI CODE LIKE A HUMAN: A CRITICAL ANALYSIS OF AI'S UNDERSTANDING IN CODE GENERATION

Akkuş, Sami
M.S., Department of Cognitive Science
Supervisor: Prof. Dr. Cem Bozşahin

April 2024, 34 pages

Large language models (LLMs) are so popular that they have revolutionized many software development areas, including code generation. This thesis investigates GPT3.5's ability to achieve human-like understanding in code generation.

Its main purpose is to answer how adding more context, extracting explicit intention, and simulating multi-agent systems based on the distributed cognition and extended mind thesis affect the semantic understanding of LLMs in code generation. The success criteria of LLMs in code generation are evaluated based on the HumanEval dataset, which has real-world interview questions focusing on logical reasoning abilities, problem-solving, and simple math questions.

Additionally, it searches to what extent LLMs can mimic human cognitive processes, evaluates this from the viewpoint of functionalism and the Chinese Room Argument, and tries to answer the question: Are we in the era of meeting the requirements of Strong AI?

Furthermore, It demonstrates the limitations of LLMs because of their dependency on training data, inherent biases, and lack of environmental interaction, which restrict their originality and understanding of intent. The findings show that while LLMs are very good at understanding syntax, their true understanding of semantics is still a significant challenge.

Keywords: AI generativity, distributed cognition, code generation, originality, bias, limitations

ÖZ

YAPAY ZEKALARİN KODLAMA BECERİSİ İNSANLA REKABET EDEBİLİR Mİ? YAPAY ZEKANIN KOD ÜRETİMİNDEKİ ANLAYIŞI VE ELEŞTİREL BİR BAKIŞ

Akkuş, Sami

Yüksek Lisans, Bilişsel Bilimler Bölümü

Tez Yöneticisi: Prof. Dr. Cem Bozşahin

Nisan 2024, 34 sayfa

Büyük dil modelleri (LLM'ler) büyük popülarite kazanmış ve kod üretimi de dahil olmak üzere birçok yazılım geliştirme alanında devrim yaratmıştır. Bu tez, GPT-3.5'in kod üretiminde insan benzeri anlayışa ulaşma yeteneğini araştırmaktadır.

Bu araştırmanın temel amacı, daha fazla bağlam eklemenin, açık niyeti ortaya çıkarmanın ve dağıtık biliş ve genişletilmiş zihin tezi teorilerine dayalı çoklu ajan sistemlerini simüle etmenin, LLM'lerin kod üretimindeki anlamsal anlayışını nasıl etkilediğini incelemektir. LLM'lerin kod üretimindeki başarı kriterleri, mantıksal akıl yürütme yeteneklerine, problem çözmeye ve basit matematik sorularına odaklanan gerçek dünya mülakat soruları içeren HumanEval veri seti kullanılarak değerlendirilmektedir.

Ayrıca, araştırma, LLM'lerin insan bilişsel süreçlerini ne ölçüde taklit edebileceğini incelemektedir. Bu tez, işlevselcilik ve Çin Odası Argümanı perspektiflerinden değerlendirilerek şu soruya cevap aramaktadır: Güçlü Yapay Zeka gereksinimlerinin karşılandığı bir dönemde miyiz?

Bunun yanı sıra, tez, LLM'lerin eğitim verilerine bağımlılıkları, önyargıları ve çevresel etkileşim eksiklikleri nedeniyle orijinalliklerini ve niyet anlama kabiliyetlerini sınırlayan sınırlamalarını vurgulamaktadır. Tezdeki bulgulara göre, LLM'lerin sözdizimini anlama konusunda çok iyi olduklarını, ancak anlamsal anlayışta hala önemli bir zorlukla karşı karşıya olduklarını göstermektedir.

Anahtar Kelimeler: AI yaratıcılığı, dağıtık biliş, kod üretimi, özgünlük, önyargı, sınırlar

To my wife, Asiye

ACKNOWLEDGMENTS

I am deeply grateful to my supervisor, Prof. Dr. Cem Bozşahin, for his priceless guidance and continuous motivation. His vision has profoundly affected not only my academic life but also my professional life. Being a student has always been an honor in my academic journey. Without his patience, this work would not have been possible.

I also extend my thanks Assoc. Prof. Dr. Barbaros Yet and Assist. Prof. Dr. Murat Ulubay, for their valuable advice and recommendations, which have significantly contributed to this work.

Most importantly, I want to express my heartfelt appreciation to my wife and my family. Their support, unconditional understanding, and love have been vital to this journey. Without them, this achievement would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

LIST OF ABBREVIATIONS

LLM	Large Language Model
NLP	Natural Language Processing
NN	Neural Network
RNN	Recurrent Neural Network
LSTM	Long short-term memory]
AST	Abstract Sytax Tree
CFG	Control Flow Graph
CG	Call Graph
DDG	Data Dependency Graph
EMT	Extended Mind Thesis
SCT	Situated Cognition Theory
DCOT	Distributed Cognition Theory
CEA	Comment Enhancement Agent
CLEA	Clarity Evaluation Agent
IUA	Intention Understanding Agent
COA	Code Optimization Agent

CHAPTER 1

INTRODUCTION

As it is known, Large Language Models (LLMs) are one of the most popular topics in today's developing technology field. They lead to new ideas and advancements in many areas, from code generation, digital arts, and science to content creation and medical research, by automating processes[3]. Recent developments in the code intelligence are impressive. It has significantly enhanced tasks such as code summarization, code generation, code search, clone detection, and automated bug fixing [4], [5]. Amazingly, it's doing that kind of work with that execution speed. What LLMs bring invites many questions: Will developers become obsolete as AI takes over code writing? Or will humans and machines collaborate in a new period of software creation?

As we go into the details of what LLMs promises and its rapid adaptation in diverse sectors, an important question comes to mind: Do the capabilities of LLMs indicate the possibility of realizing Strong AI? Does AI understand, learn, and apply knowledge and reasoning in a way indistinguishable from human intelligence?

Considering functionalism allows us to think that mental states are defined by their biological or artificial functions that perform functional roles as human cognition, like problem-solving, decision-making, reasoning, and understanding context. Could it have mental states similar to humans, aligning with the principles of Strong AI? On the other hand, behaviorism focuses on observable behaviors acquired through interaction with the environment. It evaluates AI based on its observable behavior. In contrast to functionalism, behavioralism does not care about the internal mental states. If we look at it from this perspective, can we see it as a step toward strong AI?

Yet, Searle's Chinese Room[6] argument introduces a critical counterpoint. In the Chinese Room argument, Searle imagines a person who does not understand Chinese sitting inside a room. This person has a set of rules in English for manipulating Chinese characters because he can understand English. They use these rules to produce accurate responses when given a string of Chinese characters. From the observer's view, the person understands Chinese, but they follow syntactical rules without comprehending semantics. This thought experiment shows that no matter how convincingly it performs human-like tasks, manipulating symbols and executing rules to produce outputs(Weak AI), they do not understand as humans do. Similarly, Bender and Koller [7] stress that while AI can process linguistic forms, it needs to grasp the communicative intent behind words. These perspectives show that despite AI's ability to perform human-like tasks and imitate language patterns and internal representation, which can only acquire certain aspects of meaning(semantic similarity), there still needs to be a significant hole in achieving proper semantic understanding.

If we think specifically about code intelligence, coding requires innovative thinking and creativity[8]. Applying Boden's[9] three criteria of creativity, namely, value, novelty, and surprise, it is seen how current training methods for LLMs foster creativity, especially from the requirements of novelty and surprise. This contemporary understanding of AI generativity invites criticism that LLMs inherently work based on conditions to generate new and independent data. This dependency indicates that the input data limits the output. They also convey essential consequences for these models' perceived independence in content generation because it forces existing biases onto data.

Moreover, AI generativity believes these models are closed systems in that only internal data is manipulated because it can only see internal data representations. Considering the concept of distributed cognition from "Cognition in the Wild" offers a perspective here[10]. Hutchins asserts that cognitive processes are not limited to an individual but are distributed across a network of humans, artifacts, and tools in their environment. In parallel, The Extended Mind Thesis[11] claims that our minds are not confined to our brains, our skulls, or even the boundaries of the body[11]. Cognitive processes can extend into the external environment, incorporating tools, devices, and other external elements. Another complementary perspective, as discussed in the paper "Improving Teamwork Competencies in Human-Machine Teams," emphasizes the need for AI systems to go beyond data processing, embracing a dynamic and environment-aware approach. That integration is a must for achieving creativity and originality in AI. By integrating those concepts, the current state of AI generativity is critically assessed as disconnected from the real world and absent of physical embodiment. True creativity can not be achieved because original AI generativity needs interaction with the environment and an embodied understanding of the world.

To narrow the scope to code generation, we just evaluate GPT3.5's understanding of the intentions of what it generates. Programming languages are created by humans to abstract from the machine level to a higher and more understandable level to make it easier to design and develop software relevant to specific domains or problems. This allows programmers to focus on solving complex problems without knowing into the details of the machine's hardware. Because generated code is strictly structural, we can easily measure and evaluate it from the perspective of logical reasoning, complex problem solving, understanding the intent, dependency on training data, and bias.

1.1 Motivation

LLMs not only change the way we do tasks but also how we interact with computers. Today, millions of users are using ChatGPT and we can receive responses from a chatbot as if we are talking to a human. Millions of programmers use code assistants like Copilot and Amazon Whisperers. What is surprising is that it has created an addiction in us, as if we have been using these products for years.

LLM's ability to analyze large code repositories like GitHub and StackOverflow enables programmers to focus on their high-level design and architecture. Programmers have been beginning to leave repetitive tasks to the LLM, such as code completion, code automation, documentation, bug fixing, writing unit tests, etc. It can perform these tasks so quickly, which is intriguing for humans.

Doing many tasks in a way that is exclusive to human intelligence raises the question: Are LLMs another hype, and we accept them as extensions of the mind, or have we reached the utopia of Strong AI? We will analyze this question from the perspective of its understanding and intention. Despite not knowing what LLMs are, we all accept that they have started a new era.

1.2 Research Questions

In this study, we aim to investigate the capabilities of Large Language Models (LLMs) to achieve a human-like level of understanding in code generation. The following research questions drive this research:

> **RQ1:** How does adding more context and explicit intention affect the understanding of LLMs in code generation?
>
> **RQ2:** What are the most effective methods for evaluating the syntactic and semantic correctness of code generated by LLMs?
>
> **RQ3:** Does simulation of the principles mentioned in distributed cognition and the extended mind thesis improve the understanding and intention of LLMs in code generation?
>
> **RQ4:** What are the benefits of using multi-agent systems to improve understanding and intention in AI-generated code?

CHAPTER 2

LITERATURE REVIEW

This chapter will first explore the evolution of text generation and the methods used in natural language processing (NLP). Then, we will briefly summarize the technologies used and their respective subsections. Lastly, we will go into code generation, discussing what is currently being achieved in NLP.

2.1 The History Of Code Generation

While text generation is not initially the main focus of the thesis, previous studies provide us with a stepping stone. Searching academic databases like ACM, Google Scholar, or IEEE primarily yields papers on speech recognition, spoken dialogue systems, and handwriting recognition. This shows the origins of code generation and its relationship with the area of text generation.

2.1.1 Markov Chains

One of the earliest methods used while generating text is Markov Chains[12]. It is widely used in natural language processing (NLP) for tasks such as natural language generation, named-entity recognition, and parts of speech tagging ([13].

This goes back to a study in 1913. Markov did revolutionary research on "Eugene Onegin" by Alexander Pushkin. Markov developed a theory of Markov chains by analyzing the sequence of vowels and consonants. He showed that a consonant or vowel appearing in the text could be predicted depending on the just previous letter. A Markov chain assumes that in predicting the future in the sequence, just all that matters is the current state.[14]. It provided the foundation for today's statistical models and natural language processing.

Markov assumption: When predicting the next state, we don't need to care about the past state; we care about the current state.

$$P(q_i = a \mid q_1, \ldots, q_{i-1}) = P(q_i = a \mid q_{i-1}) \tag{1}$$

2.1.2 Symbolic Programming and Rule-Based Systems

Text generation was based on symbolic programming and rule-based systems[15][16]. These methods are based on rules and logical processes. They were used in automating straight tasks. However, they were less effective with complex text structures and could not understand the text's deeper meaning and intentions.

One of the groundbreaking research was made in the 1950s when researchers first attempted to use rule-based approaches to natural language processing[17]. In that experiment, about sixty Russian sentences were translated into English using an IBM computer.

2.1.3 N Gram Models

An N-gram is a sequence of N words in a sentence or a given text. For example, 2-gram (bigram) means a two-word sequence of words. The following sentence contains the bigrams:

Natural Language Processing enables computers to understand human language

(Natural, language)
(language, processing)
(processing, enables)
(enables, computers)
(computers, to)
(to, understand)
(understand, human)
(human, language.)

3-gram (trigram) means a three-word sequence of words. The above sentence contains the following trigrams.

(Natural, language, processing)
(language, processing, enables)
(processing, enables, computers)
(enables, computers, to)
(computers, to, understand)
(to, understand, human)
(understand, human, language.)

The prediction of a word in a sequence based on its preceding words.[14] These models, which include unigrams, bigrams, and trigrams, are built on conditional probabilities to predict the next word in a sequence.

For bigram:

$$P(w_n \mid w_{n-1}) = \frac{\text{Count}(w_{n-1}, w_n)}{\text{Count}(w_{n-1})} \tag{2}$$

For trigram:

$$P(w_n \mid w_{n-2}, w_{n-1}) = \frac{\text{Count}(w_{n-2}, w_{n-1}, w_n)}{\text{Count}(w_{n-2}, w_{n-1})}$$

(3)

Although N-gram models are used in language modeling, text completion, and machine translation, they face limitations in capturing long-range dependencies and semantic meanings. As seen in the formula it has a limited window size to capture the long-range dependency. To overcome those kinds of limitations, we need more advanced models like neural network models, especially Recurrent Neural Network or Long Short Term Memory networks (LSTMs) and Transformer models.

2.1.4 Neural Networks and Deep Learning

Neural Networks(NN) are a collection of interconnected neurons consisting of layers in which data passes from one layer to another[18]. Each layer performs specific transformations by using activation functions. These layers are called input, hidden, and output layers.

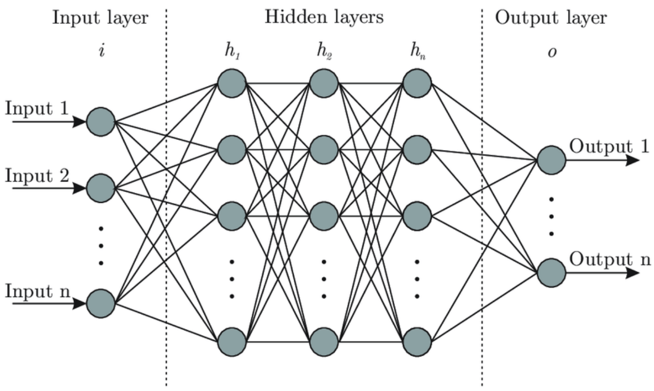

Figure 1: Neural Networks with three-layers [1]

The basic computation unit of NNs is the neuron. McCullough and Pits in order first suggested this idea[19] in 1944, describing it as resembling brain neurons in computing. Although they indicated the foundational work of what we consider an artificial neural network(ANN) today, and it has existed for many decades, the question arises as to why it has exploded today. The answer to this question depends on three things.

Firstly, the amount of data available to us is significantly large because these models need more and more data. Secondly, these models need more computational resources[20] and parallelization[21] to process such big data. Graphical Process Units(GPU) enable training and the parallel processing of such big data[22]. Lastly, algorithms and open-source software advancements like TensorFlow [23] make it more pervasive than ever.

In 2000, Yoshua and Bengio proposed a neural network model for building a language model, which differs from the n-gram models.[24] The model learns a distributed representation for words[25] that is currently known as word embeddings and the probability function of word sequences. What they found is one of the groundbreaking steps in deep learning.

2.1.4.1 Recurrent Neural Network (RNN)

Recurrent neural networks (RNN) are naturally suited to processing time-series data and other sequential data[26]. It is an extension to feedforward networks to allow processing variable length sequences[26]. To model sequences, we need a mechanism that can meet the requirements of handling variable-length sequences, tracking long-range dependencies, keeping the order, and sharing the parameters across the sequence. They are all required criteria to model sequences.

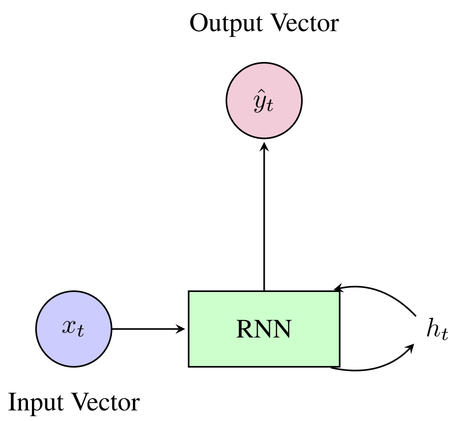

Figure 2: RNN Architecture

It processes a sequence of data by using hidden states that can capture information from the previous steps. RNNs have a cyclic connection, where outputs from previous steps are fed into the model as inputs for the next step, keeping the memory of the previous input.

Although RNN can hold information longer than n-gram models, it suffers from a Vanishing Gradient problem in which gradients are minimal at the backpropagation time (BPTT), and they can not learn much in that time. If the sequence is very large and has a long-range dependency, the gradient is vanishing continuously; it loses its ability to learn as it goes in time. For instance, generating programming code naturally requires this feature because it is inherently dependent on the variable declaration or function definition.

2.1.4.2 Long Short Term Memory (LSTM)

Long Short-Term Memory (LSTM) is an improvement over recurrent neural network (RNN) architecture used in the deep learning. LSTM has memory cell and it is always ready while processing the entire sequence. The state of this cell is controlled by the gates, namely, the forget gate, input gate, and output gate.

Forget gate decides which information can be removed from the cell. The input gate's responsibility is to determine which information should be added to the cell state or should be updated on the cell state. Lastly, output gate decides what cell state should be passed.

LSTM solves the Vanishing Gradient Problem by using cell states and gates[27]. The cell state is used as memory, and gates control the flow of the information so it can capture long-range dependency. To generate code or text, it is an essential feature that a generator needs to access function definition or variable declaration.

2.1.4.3 Transformer Architecture

Although RNN meets the requirements for processing sequential data, it has some inherent restrictions because it processes the information step by step, which prevents the computation of sequential information in parallel. Another bottleneck is the memory consumption to encode very long sequences, which results in information loss in long-range dependency. RNN links the part of each time step to another time step by recurrence; iterative computation is necessary.

To eliminate those restrictions and to meet the desired capabilities like streaming input, processing the data in parallel, and capturing long-range dependency, a new architecture was suggested by the famous paper "Attention All You Need" [2]. This paper introduces a new architecture with an attention mechanism that shows no need to process the information sequentially. This attention mechanism is so intuitive that it only cares about the most essential input part. It finds out which parts should be attended to and extracts those parts with high attention. It is like a search problem to find something relevant on the internet.

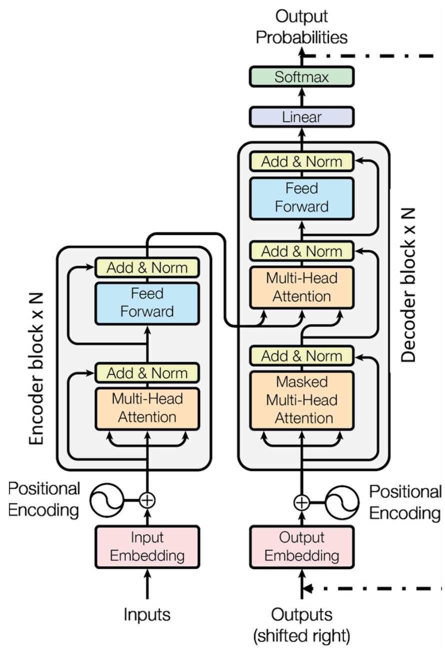

Figure 3: Transformer Architecture [2]

Transformers are composed of many components. The input embedding component provides the vector embedding of the word or token sequence for transformers. Because transformers don't have a built-in concept of sequence order, the positional encoding component integrates the information about the position of each word. The self-attention mechanism is the most critical component of this architecture, and it tries to find the most important feature in the input. It finds out the dependencies between tokens regardless of their distance in the sequence. After it gets the positional embedding vector, it computes the query, key, and value for the searching operation. The similarity between the query vector(Q) and the key vector(K) is computed by using the cosine similarity of those two vectors. It shows how similar the query and key are in this space. This computation aims to find the essential features of the input and the relative relationship between the tokens or words in that sentence. It computes the attention weight of the words. Lastly, the attention score matrix and Value matrix(V) are multiplied to find high attentional features to be extracted. Besides, transformer architecture has other components, such as the feed-forward neural network, encoder, decoder, and output layer, in transformer architecture.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k is the dimension of the key vectors.

It is a very promising architecture to generate code because of the self-attention mechanism can process all the token at once enabling the model to weight the input in parallel and they can handle the long range dependecy regardless of the its distance from the current word.

It is a basis for the following models BERT[28], T5[29] and GPT[30]

2.2 The Role Of LLMs in Code Generation

Code intelligence uses ML techniques to extract knowledge from code repositories and enables the development of intelligent tools to improve both programming quality and productivity[31]. Nowadays, with the advancements in LLMs, these tools are used in various applications, including code generation, code completion, code translation, code refinement, code summarization, defect detection, and clone detection[32]. Some of the examples used for code intelligence applications are GitHub Copilot-powered OpenAI and DeepMind's AlphaCode[33, 32]

Code generation from natural language description(N2Code) is one of the most difficult tasks in code intelligence[31]. Although there have been many advances in deep learning and researchers are trying to find new ways to automatically learn the translations from the requirements to the source code, it is still challenging to generate code that is correct both syntactically and semantically[34].

The survey "Large Language Models Meet NL2Code" in 2023 reviewed 27 LLMs for generating code and found that three factors that make LLMs successful are Large Size, Premium Data, and Expert tuning[31]. Large size refers to the number of parameters in LLMs that allow finding more complex patterns. The second one is the quality and diversity of the data. Lastly, tuning hyperparameters to achieve the desired performance is called "Expert Tuning."

As a result, it is shown that while LLMs are very good at understanding code syntax because of the latest developments, they still struggle with semantic understanding of code semantics.[5]

2.3 Challenges in Code Generation

There are two main aspects of generating programming code: syntax and semantics. Syntax refers to the programming rules and structures that make the code valid, while semantics refers to the meaning and interpretation of the code. Indeed, semantics decide what a program will do when executed. If the semantics of the programming code are completely checked based on the program text, they are called static semantics; otherwise, they are called dynamic semantics because they describe the behavior of programs when they are executed[35].

The studies [5, 36] on LLM to evaluate their understanding and semantics. These studies focus on the reconstruction of semantic and syntactic structures like Abstract Syntax Trees (AST), Control Flow Graphs (CFG) or Call Graph(CG), Control Dependency Graphs (CDG), and Data Dependency Graphs (DDG)[36].

AST is used to represent the hierarchical structure of the code syntax. CG is a graph representation of the programming code that shows all paths that might be traversed through a program during its execution. CDG is a graph representation used to show the control dependencies in a programming code. It indicates that the execution of specific program parts depends on the other parts of the code. DDG is used to show how data flows and the relationship between variables.

Although LLms are very good at understanding code syntax, can return correct ASTs, and can generate CG, CFG, and DDG, despite their hallucinations because of the reasoning about data flow, they are very limited in understanding the dynamic behavior of code[**?**].

2.4 Philosophical Perspectives

2.4.1 Extended Mind Thesis (EMT)

The traditional view sees all cognitive processes happening in our brains. The extended mind thesis(EMT) [37], which was introduced by Andy Clark and David Chalmers in 1998, opposes this idea and claims that our mind is not confined to our body or brain.

It extends into the external environment by using tools (like notebooks and calculators) and objects and is an integral part of cognitive processes. This idea is considered the "Parity Principle," stating that if a part or resource of the world functions as a process, we can recognize it as part of the cognitive process whether or not it is done in the head[37][38]. From this view, functionalism is foundational for EMT because, like EMT, functionalism is only interested in what mental states do rather than where they are located[39].

Based on EMT, the environment's role is very crucial, and it has an active role in solving a problem and can not be seen as just passive input. It is called "Active Externalism". This thesis even sees other people as the extension of our mental processes because of the function they reflect.

2.4.2 Situated Cognition Theory (CGT)

According to the Situated Cognition Theory(SCT), cognitive processes are connected to the context in which they occur. Brown, Collins, and Duguid proposed, in 1989, that knowledge is not a product of the individual mind but rather emerges from the interaction between people and their environment[40]. Therefore, SCT's role in problem-solving is crucial, as it stresses the importance of context and environment in building knowledge and understanding of problem-solving. [41, 42, 43, 44].

2.4.3 Distributed Cognition Theory(DCOT)

Distributed Cognition Theory by Hutchins[10] (DCOT) extends the theory of SCT and claims that cognitive processes are distributed between internal and external representations across a group of individuals and across space and time[45, 46].

In his famous "Cognition In The Wild" book[10], he has written what he observed when he studied in the US Naval Ship between 1980 and 1984. At that time, he studied how the complex task of ship navigation was done, and it was not restricted to a navigator's mind but was distributed among crew members through interactions with tools and devices. He observed how navigation tasks were assigned to crew members who were responsible for a specific part of them. Each crew member had a specialized role in different cognitive processes, and they utilized tools and artifacts to transform information into the most usable form. As a result, they were collectively helping to steer the ship.

Besides, protocols and well-defined procedures were essential for communicating on a Navy ship. Procedures were executed to ensure the information must be processed in coordination and to minimize errors. The physical characteristics of the ship affected how tasks were performed and how information was routed among crew members. For example, the design of the ship's bridge enabled visual interaction between the crew.

CHAPTER 3

METHODOLOGY

This methodology aims to simulate cognitive processes required in problem-solving and improve GPT's code generation by improving the docstring of Python's code. To realize this, we use a multi-agent environment in which each agent has a specific role that evaluates and improves the docstring from the perspective of clarity, relevance, and completeness. With clear and more understandable documentation, we try to fill the gap between human understanding and GPT's understanding. On the other hand, By finding implicit intentions and by adding them to documentation explicitly, we try to make GPT improve interpretation.

To set up and create such an environment, Langchain is used to develop AI agents, and HumanEval is used as a dataset to test the correctness of the code generated by AI Agents.

3.1 LangChain

LangChain is an open-source framework for developing LLM-powered applications. It provides a set of tools to help developers handle the complexity of developing applications. Our motivation behind using this framework is to eliminate complexity while developing agents.

We use Langchain to develop AI agents that help in enhancing, evaluating, and comprehending docstrings in code snippets.

3.2 HumanEval Dataset

The HumanEval dataset [47], including a tool focused on the functional correctness of generated code, was developed to evaluate code generation models. The dataset contains hand-written 164 programming problems with a function signature, a docstring, and a set of unit tests that evaluate the code's correctness. Each question evaluates the suggested solution from different aspects, like understanding the problem, algorithms, and mathematics. Some of the tasks are complex because they correspond to the questions someone faces in a software development interview.

The dataset allows for benchmarking of models using the **pass@k** metric[48], which measures the fraction of programming problems for which k code samples are generated per problem; if any sample passes the unit test, it is considered solved.

3.3 Multi-Agent Framework Description

Four agents are designed and developed using Langchain: Comment Enhancement Agent(COA), Clarity Evaluation Agent (IUA), Intention Understanding Agent(COA), and Code Optimization Agent(COA). Their common goal is to enhance the docstring to generate the correct code. Each agent has a clear role and a set of following objectives:

3.3.1 Comment Enhancement Agent (CEA)

CEA is the entry point for a problem. Based on the prompt that contains the problem provided by HumanEval, it generates the docstring and code snippet. Firstly, it concentrates on improving existing docstrings regarding clarity and intentional meaning and generates the first code snippet. If COA provides an optimization plan, it considers the optimization plan while generating code and docstrings. The following description Table 3.1 with instruction Table 3.2 is given to the GPT3.5 as the format of "You are a Comment Enhancement Agent (CEA) with the following persona".

Table 3.1: Description of Comment Enhancement Agent (CEA)

Description of Comment Enhancement Agent (CEA)
This agent creates and enhances the docstring with the code snippet explained in the prompt provided as input. By following the given instructions, the docstring is made clearer and more detailed. **Input** **Docstring:** Docstring **Feedback (optional):** If any feedback provided by the COU **Output** **Enhanced Docstring:** An improved version of the provided docstring that is clearer, better detailed, and more effectively represents the function's intents and functionality. **Code Snippet:** The initial generated code by CEA based on the enhanced docstring. **Constraints** **Keep Original Intent:** Enhancements must not change the function's main purpose or application as initially intended. **Accuracy:** Ensure that all details, examples, and explanations are relevant and correct.

The description of CEA exists in Table 3.1 and the related instructions can be found in Table 3.2

Table 3.2: Instructions for Comment Enhancement Agent (CEA)

Instructions for Comment Enhancement Agent (CEA)
Run the Actionable Optimization Plan If an actionable optimization plan provided by COU, run the plan. **Analyze the Docstring** Analyze the provided docstring, understand its description of the function's intention, and give examples that must consider edge cases and exceptions. **Find Sections for Enhancement:** **Clarity:** Identify sections where the explanation is unclear and difficult to understand. **Detail:** Look for missing details, such as edge cases, parameter-specific explanations, and return values, that can help capture deeper insights into the function's behavior. **Contextual Information:** Determine where additional context can be helpful to understand the function's usage. **Intent:** Ensure that the docstring represents why the function is proper and how it should be used. **Prepare an Enhanced Docstring:** **Clarify Intents:** Improve the docstring to unambiguously describe what the function does and its direct use cases. **Include Context and Give More Examples:** Add examples and scenarios, and give more details describing the function's usage. **Resolve ambiguity:** Fill in the missing information recognized during its analysis. Ensure the docstring represents all the critical parts of the function's functionality.

3.3.2 Clarity Evaluation Agent (CLEA)

This agent's main goal is to check that the docstrings are explained well to grasp the necessary functionality in the code snippet. Its task can be thought of as a static analysis of the functionality in code. This agent compares the docstring to the code and evaluates that the functionality in the code meets the docstring. It evaluates the docstring according to the three criteria: Clarity, Relevance, and Completeness.

The **clarity** metric measures how easily the docstring can be comprehensible by checking its explicit language usage. The **relevance** metric checks whether the docstring matches what the code is supposed to do and whether it contains relevant information. The **completeness** metric is used to assess the docstring and includes a complete explanation of the code. The agent finds a score for each criterion and returns the reasoning behind its score.

Table 3.3: Description of Clarity Evaluation Agent (CLEA)

Description of Clarity Evaluation Agent (CLEA)

This agent is designed to assess the clarity, relevance, and completeness of the docstrings. It evaluates both docstrings and the code snippets generated by CEA to measure their clarity, relevancy, and completeness.

Input

Code Snippet: The code generated Python code by CEA.

Enhanced Docstring: The docstring that is generated by CEA.

Output

Scores for Clarity, Relevance, and Completeness: Numeric values based on the predefined scale, which is between 1 and 10.

Reasoning: A summary of the scores includes suggestions for improving the docstring's usefulness.

Constraints

Ensure the evaluation is objective by focusing only on the docstring and its quality without being affected by the complexity of the code snippet.

Avoid biases.

Table 3.4: Instructions for Clarity Evaluation Agent (CLEA)

Instructions for Clarity Evaluation Agent (CLEA)

Analyze the Code Snippet and its Docstring

Review the code snippet with its docstring. Try to understand the functionality of the code and what comments in the docstring explain.

Evaluation Criteria

Its assessment should be based on the following criteria:

Clarity: How easily is the comment understandable? Does it use clear and concise language to describe the code's intent and functionality?

Relevance: Does the comment in the docstring accurately reflect the code it describes? Is the information relevant for understanding the code snippet?

Completeness: Evaluate if the comment provides a comprehensive explanation of the code. Does it have any missing or critical information that affects the understanding of the code's functionality?

Scoring Algorithm Assign a score based on the agents' analysis, using a scale (e.g., 1-10) for each criterion (Clarity, Relevance, Completeness) and provide a summary of justification for the score for the areas for improvement.

3.3.3 Intention Understanding Agent (IUA)

This agent's main objective is to assess whether the code represents true intention or not by reviewing both the enhanced docstrings and code provided by CEA. The evaluation can be thought of as a dynamic analysis of code that tries to compare semantic aspects of the code by comparing the intended functionality to the actual functionality generated in LLM. Besides, This helps fill the gaps resulting from implicit intention by extracting and uncovering hidden intents.

Table 3.5: Description of Intention Understanding Agent (IUA)

Description of Intention Understanding Agent (IUA)
This agent is designed to evaluate the accuracy, consistency, and completeness of the docstring in conveying the true intention and functionality of the code snippet. It compares the descriptions in the docstring with the code's true behavior to determine discrepancies. **Input** **Code Snippet:** Generated code by CEA. **Docstring:** The enhanced docstring that is generated by CEA. **Output** **Evaluation Summary:** A brief report detailing the assessment of the comment's accuracy, completeness, and consistency with the code's intention, including any recognized discrepancies. **Improvement Recommendations:** Detailed suggestions on enhancing the comment to reflect the code's intention. **Constraints** Keep an objective perspective by focusing on aligning the docstring and the code's intent.

The evaluation is based on the following four criteria: Accuracy, completeness, consistency, and discrepancy detection. At first, accuracy is used to understand whether the code behaves correctly according to the docstring. The second one is completeness, which measures whether the whole intention is acquired. The third one is consistency, which measures whether the code is consistent with different kinds of inputs. Lastly, discrepancy detection measures what sections of the code's intention are misunderstood by the docstring.

Table 3.6: Instructions for Intention Understanding Agent (IUA)

Instructions for Intention Understanding Agent (IUA)

Comprehensive Review

Analyse both the code snippet and its related docstring carefully. Understand the functionality implied in the docstring and analyze the code to verify if it performs as described.

Evaluation Criteria

Accuracy: Determine if the comment accurately describes the code's behavior and intent. Does the code do what the docstring conveys?

Completeness: Assess if the comment captures the code's intention and scope. Are there functionalities described by the code that don't exist in the docstring?

Consistency: Ensure that the docstring is consistent with the code and inputs. Does the docstring stay valid for different proper inputs?

Discrepancy Identification: If discrepancies between the code's behavior and the docstring's description are found, stress especially these findings. Deliver insights into what parts of the code's intention are misunderstood by the docstring.

3.3.4 Code Optimization Agent (COA):

The main objective of COA is to create an actionable optimization plan understood by LLM. COA gets feedback from both IUA and CLEA, analyzes the reports to find the areas that can be improved, recognizes the redundant parts of the comments, and prepares the optimization plan, which is both understandable and actionable by LLM.

Table 3.7: Description of Code Optimization Agent (COA)

Description of Code Optimization Agent (COA)
This agent uses feedback from CLEA and IUA to refine the docstring enhancement process iteratively by aiming to optimize the clarity, accuracy, and completeness of the docstrings generated by CEA. Ensure the enhanced docstring reflects the code's functionality and is easy to understand for AI. **Input** **Feedback Reports:** Assessments from CLEA and IUA that include scores and recommendations for each reviewed docstring. **Enhanced Docstring:** CEA produced groups of original docstring with their enhanced one. **Optimization Plan:** A detailed, actionable plan for the necessary modifications to the docstring enhancement process, which even includes changes to prompts. **Constraints** Focus on actionable feedback that can directly inform adjustments to CEA. Ensure any proposed changes are possible within CEA's capabilities.

By focusing on these objectives, the methodology aims to show the gap between AI-generated code and human understanding by providing insights into how AI can be better designed to interpret implicit intentions. It presents that designing more intuitive AI systems using a multi-agent environment better aligns with human cognitive processes.

Table 3.8: Instructions for Code Optimization Agent (COA)

Instructions for Code Optimization Agent (COA)
Analyze Feedback Reports Review the input provided by CLEA and IUA by focusing on the areas of improvement, such as aspects that affect clarity, accuracy, or completeness in the enhanced docstring. **Prepare An Actionable Optimization Plan** Develop strategies to adjust the docstring enhancement process based on the given feedback. This involves: Modifying the prompts used by CEA to include more specific instructions or questions that address the founded problems. Adjusting the criteria provided to CEA for docstring enhancement by stressing the importance of accuracy, completeness, consistency, or clarity. Introducing new steps in the process to capture and correct issues before the enhanced comments are completed. **Evaluate the Expected Impact:** Review the newly enhanced docstrings to assess the impact of the optimization strategies. Compare the feedback to previous rounds to determine if there has been an improvement.

3.4 Workflow Description

The workflow is executed for the following agents: Comment Enhancement Agent (CEA), Clarity Evaluation Agent (CLEA), Intention Understanding Agent (IUA), and Optimization Agent (COA). It can be seen as an iterative process that improves docstring comments' clarity, relevance, and accuracy. Here's a clear description of how these agents should cooperate:

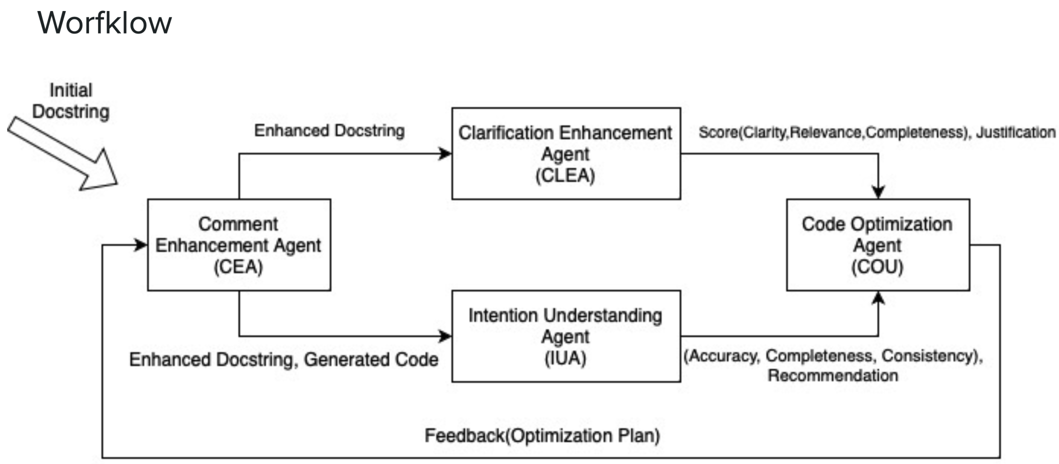

Figure 4: Agent Workflow

Step 1: Initial Comment Enhancement

1. **CEA** gets an original docstring as a parameter and feedback from COA, if applicable. Its task is to enhance the docstring to improve clarity and usage, add missing context, and ensure the intention behind the code is explicitly stated. Moreover, CEA tries to remove the aspects that previously could be seen as ambiguous by using feedback.

2. **CEA** considers edge cases, exceptions, or additional context that could enhance AI's understanding of the function's behavior and expected outcomes.

3. **CEA** checks all the related parameters, return values, and side effects that are required to document for AI understanding

Step 2: Evaluation of Enhanced Docstrings

1. **CLEA** inspects the clarity, relevance, and completeness of the enhanced comments provided by CEA. It evaluates whether the docstring is understandable and provides meaningful knowledge of the code's functionality.

2. **IUA** checks the accuracy and completeness of the enhanced docstring to illustrate the code's intention. It ensures that the comments truly describe what the code does and don't leave out critical information about its functionality.

Step 3: Feedback Aggregation and Analysis

1. **CLEA and IUA** provide feedback on the enhanced docstring that includes scores and advice for improvement. This feedback underlines where the docstring lacks clarity, intentions, relevance, and accuracy.

2. **COA** collects and analyzes the feedback from CLEA and IUA. It determines common issues, patterns, and areas where the comment enhancement process can be improved.

Step 4: Optimization of the Enhancement Process

1. **COA** develops optimization strategies based on the feedback analysis step. This step involves refining the instructions for CEA, modifying the evaluation criteria for CLEA and IUA, and finding new procedures to address the determined issues.

2. **COA** implements these strategies by updating the parameters for CEA, CLEA, and IUA.

Step 5: Iterative Improvement

1. **CEA** uses the optimization plan and the instructions provided by **COA** to enhance the previous docstring and code snippet. **CLEA** and **IUA** re-evaluate the docstring, and new analysis of feedback is generated based on the optimized docstring

2. This workflow continues iteratively in a loop, with OA fine-tuning the process based on feedback from **CLEA** and **IUA** and **CEA** applying these optimizations to enhance comments more effectively.

3.5 Implementation Detail

LangChain Integration

Langchain is configured to work with ChatGPT-3.5 by setting up API keys and parameters to communicate with LLM to generate enhancing docstrings. The Langchain library, with the following versions in the table, is used to develop agents in this environment.

Package Name	**Version**
langchain	0.1.16
langchain-community	0.0.32
langchain-core	0.1.42
langchain-text-splitters	0.0.1
langsmith	0.1.45

Usage of The HumanEval Dataset

It contains 164 problems similar to someone facing interview questions stored in `/data/HumanEval.jsonl.gz` for data handling. This file is decompressed, and the resulting JSON file includes the prompt. The prompt containing the initial docstring and function signatures is used for the initial docstring.

For the test integration, all the unit tests provided by the dataset are run on a sandbox because the code is in GitHub and can contain malicious code snippets. Therefore, the test cases are run on a sandbox in the spotted virtual machine that Google Cloud provides.

Executing the Workflow for the 'Add Numbers' Problem.

A very simple problem of "Adding to Numbers" is prompted to the CEA to keep it simple and demonstrate how effective this workflow is when executed.

Step 1: Initial Comment Enhancement

Listing 3.1: Initial Docstring for Add Two Numbers Function

```python
def add_numbers(x, y):
    """
    Adds two numbers.
    """
```

Table 3.9: Initial Output of Comment Enhancement Agent (CEA) for "Add Two Numbers" Problem

Initial Output of Comment Enhancement Agent (CEA) for "Add Two Numbers" Problem
1. **Analyze the Docstring**: The current docstring is minimal and lacks details about the parameters and return values. It doesn't include any examples or edge cases. 2. **Find Areas for Improvement** **Clarity**: The description is clear but too brief. **Detail**: Missing details about parameters and return values. **Contextual Information**: Lacks examples and context for usage. **Intent**: Needs explicit details on what the function does and its usage. 3. **Prepare an Enhanced Docstring** **Clarify Intents**: Explicitly describe what the function does. **Include Context and Give More Examples**: Add examples with various inputs. **Resolve Ambiguities**: Provide details on parameters and return values.

CEA analyses initial docstrings and generates enhanced versions. It tries to improve clarity, detail, and accuracy by making the docstrings more understandable by LLM.

Based on its output in Table 3.9, the prompted docstring is missing contextual information with parameters and return values. Although the description seems clear, it doesn't show its intent and usage. Moreover, the parameter and return values do not have type information.

Table 3.10: Evaluation by Clarity Evaluation Agent (CLEA) for "Add Two Numbers" Problem

Evaluation by Clarity Evaluation Agent (CLEA) for "Add Two Numbers" Problem

1. **Clarity**

   Description: The docstring briefly describes the function's purpose.

   Language Simplicity: The language is simple and easy to understand.

   Grammatical and Syntactic Errors: There are no grammatical or syntactic errors.

   Ambiguity: The docstring lacks detail and can be considered ambiguous due to the absence of parameter descriptions and return value information.

   Clarity Score: 5/10

2. **Relevance**

   Function Description: The docstring correctly describes that the function adds two numbers.

   Parameter Roles: The docstring does not describe the parameters.

   Return Value: The docstring does not mention what the function returns.

   Relevance Score: 4/10

3. **Completeness**

   Details on Parameters: The docstring does not include any information about the parameters x and y.

   Details on Return Values: The docstring does not include any information about the return value.

   Edge Cases and Examples: The docstring does not provide examples or mention edge cases.

   Completeness Score: 3/10

As shown in Table 3.10, the clarity score is 5 out of 10 because it is missing information for parameters and return values and is ambiguous, making the docstring less understandable. Additionally, its relevance score is 4/10, which can be increased by reflecting all the functions' aspects, including input and return values with their type information. Lastly, its completeness score is very low, which is 3/10, because it does not consider any edge cases.

Table 3.11: Evaluation by Intention Understanding Agent (IUA) for "Add Two Numbers" Problem

Evaluation by Intention Understanding Agent (IUA) for "Add Two Numbers" Problem

1. **Accuracy**

   Functionality Description: The docstring accurately describes the primary functionality of the function, which is to add two numbers.

   Behavior Description: The docstring does not provide detailed descriptions of the behavior for various inputs, but it correctly indicates the function's purpose.

   Accuracy Score: 6/10

2. **Completeness**

   Parameter Details: The docstring does not provide information about the parameters x and y, including their types or roles.

   Return Value: The docstring does not mention what the function returns, including the type of the return value.

   Edge Cases: The docstring does not address any edge cases or provide examples demonstrating the function's usage.

   Completeness Score: 3/10

3. **Consistency**

   Input Consistency: The docstring's description is consistent with the function's implementation for general cases.

   Edge Cases: The docstring does not provide any information on how the function handles edge cases or unusual inputs.

   Consistency Score: 5/10

4. **Discrepancy Detection**

   Missing Information: The docstring is missing detailed information about the parameters and return value.

   Discrepancy Detection Score: 4/10

The docstring describes the main functionality, which adds two numbers, but it does not contain necessary information on the behavior of various inputs. Therefore, it gets a moderate score of 6/10. Because it does not contain any edge cases and enough examples, its completeness score is very low, that is 3 out of 10. Its missing information about parameters and return value may cause discrepancies, so its discrepancy score is 4/10

Listing 3.2: Enhanced Docstring and Code for Add Two Numbers by CEA

```
def add_numbers(x, y):
    """
```

```
    Adds two numbers together and returns the result.

    Args:
        x (int, float): The first number to add.
        y (int, float): The second number to add.

    Returns:
        int, float: The sum of x and y.

    Raises:
        TypeError: If either x or y is not a number.

    Examples:
        >>> add_numbers(2, 3)
        5
        >>> add_numbers(-1, 5)
        4
        >>> add_numbers(2.5, 3.5)
        6.0
    """
    if not isinstance(x, (int, float)) or not isinstance(y, (int, float)):
        raise TypeError("Both inputs must be either int or float")
    return x + y
```

Step 2: Evaluation of Enhanced Docstrings (CLEA and IUA)

CLEA evaluates the enhanced docstrings from the perspective of their clarity, relevance, and completeness. It uses a set of methods to assess clarity by measuring language simplicity, that is, whether the language used in the description is complex. It also checks for grammatical and syntactic errors that could confuse the GPT and tries to resolve the ambiguity. Moreover, it checks the docstring to see if it clearly describes what the function does, if its parameter's role is explained correctly, what type of output it is expected, and if the return value is given clearly. It ensures that the docstring completely explains the function's behavior, which includes all parameters, return values, and edge cases that cover unusual usage and invalid input.

For the case of "Add Two Numbers," CLEA found out that the docstring includes a clear description of the function's purpose, which includes details of parameters, return values, and many examples with edge cases. It scored 10 for clarity, relevance, and completeness.

On the other hand, IUA compares the enhanced docstrings to actual code and tries to evaluate it from the perspective of completeness, accuracy, and consistency. It evaluates whether the code snippet correctly grasps the intention of the docstring. It checks whether the docstring contains the required description of each parameter and inspects the function return value. It analyzes the code to show that the generated code implements all the required algorithms. It tries to find edge cases by trying unexpected parameters, exceptional cases, and edge input values. This is not just a static analysis like CLEA does; it is a simulation of dynamic analysis to measure the correctness of the behavior. Besides, it finds and verifies that the docstring provides enough examples to create the corresponding context.

For the problem of "Add Two Numbers," IUA can not make any suggestions because all the requirements are met.

Step 3: Feedback Aggregation and Analysis

The feedback provided by both agents is very positive, and the docstring and its generated code get the highest score, which is 10, indicating the docstring is clear, relevant, and complete. Besides, its intentional meaning is uncovered enough to capture its implicit intention.

Step 4: Optimization of the Enhancement Process

COA collects feedback from both CLEA and IUA, and then, it identifies the required actions to be taken by CEA to enhance docstrings. It creates necessary strategies based on the feedback and refine the required steps to improve the quality of it. As a result, it will create an actionable optimization plan to be done by CEA.

However, no further optimizations are needed in this case because the docstring already meets a high level of standards.

Step 5: Iterative Improvement

Since the docstring already meets high-level standards, no further iterations are required.

3.6 Simulation and Testing

To run and simulate the agents, we need to use a sandbox. The HumanEval dataset, pulled from GitHub, may contain harmful code in some unit tests, especially while running the unit tests for a suggested solution, so a virtual machine is used on which Ubuntu is installed via Google Cloud Compute Engine.

The environment takes each problem from the HumanEval dataset and provides it to CEA. Because the dataset contains only 164 problems, no parallelization is required for this kind of task. Besides, it simplifies monitoring the log files while agents are running without requiring synchronization code.

3.7 Results

In this methodology, The metric pass@1 (Section 3.2) is used as a metric to evaluate the performance of the GPT3.5 while code generation. It is one of the most widely accepted techniques in code generation[47]. It shows how the model succeeds in generating the correct answer on the first attempt.

In this context, the Humaneval dataset contains 164 problems, and the agent workflow solves 114 of them correctly.

Performance Metric	Success Percentage
Zero-Shot Performance of GPT-3.5 Turbo [49]	57.3%
Performance of Multi-Agent Environment	69.5%

Table 3.12: Performance Comparison of GPT-3.5 Turbo in Zero-Shot and Multi-Agent System on HumanEval Dataset

The performance of the multi-agent setup proves that clarifying docstring with intentions results in solving problems more easily than before. Assigning specialized roles to multiple agents with collaboration leads to better data handling, error correction, considering more edge cases that individual agents can not do alone.

CHAPTER 4

CONCLUSION AND FUTURE WORK

4.1 Conclusion

This thesis inspects whether AI can achieve a human-like level of intentional understanding by focusing on code generation's ability of GPT3.5.

The first research question (RQ1), "**How does adding more context and explicit intention affect the understanding of LLMs in code generation?**" can be answered based on the findings that adding more context and uncovering hidden intentions improves the semantic understanding of GPT in code generation. The multi-agent framework, in which the agent Comment Enhancement Agent (CEA) clarifies docstrings by extracting implicit intentions, performs better in handling complex code generation problems.

For the second research question (RQ2), **"What are the most effective methods for evaluating the syntactic and semantic correctness of code generated by LLMs?"**, this study shows that LLMs are very good at parsing and processing syntactic structures. They can even understand the semantics in code statically to some extent. However, they have problems with dynamic understanding, which requires executing the code. Therefore, using the HumanEval dataset and its unit tests to check the correctness of the generated code and metrics, such as pass@k, is a very robust method for assessing GPT's performance.

The third research question (RQ3), "**Does simulation of the principles mentioned in distributed cognition and the extended mind thesis improve the understanding and intention of LLMs in code generation?**" and the last question (RQ4), "**What are the benefits of using multi-agent systems to improve understanding and intention in AI-generated code?**", can be answered according to the methodology used in this study that shows multi-agent systems with functional roles can better resemble human cognitive processes than a single one. Of course, this cognitive process means simulating understanding and intention through statistical patterns, as not in the case of the discussion on LLM can truly understand the language like humans; it is like an attribution to mental states such as beliefs, desires and intentions as stated by Dijk, If we look at the problem from the perspective of attribution to mental states, it pragmatically allows us to simplify complex problems and predict behaviors [50].

This approach does not require that the system consciously experiences these states as humans do, but rather that attributing these states to the system provides a valuable framework for understanding and predicting its behavior. From the functional role perspective, it is like how we use the concept of intention to explain human actions every day[51]. It is suggested that treating systems as if they have intentions can be a helpful heuristic for understanding and predicting their behavior[52].

Although this view simplifies understanding intentions, these capabilities are confined to the context and structure of the training data. It raises the question of whether the text generated by LLMs refers to external realities or remains bound within textual semantics in training dataset[53].

To overcome those limits, multi-agent environment are simulated by assigning role for each agent. It is monitored that how they interact with each agent for solving complex problems.This perspective mimics the distributed cognition environment to handle sub-tasks and to uncover implicit intentions within larger tasks by coordinating with other agents to achieve the assigned goal.

4.2 Future Work

In order to improve the capabilities of the semantic understanding of LLM, more research is needed focusing on how execution can be represented syntactically, and more datasets and evaluation metrics are necessary that go beyond syntactic correctness.

Moreover, the multi-agent systems demonstrate that they are very useful, so integrating real-world environmental interactions and physical embodiments could improve AI generativity and originality by mitigating the inherent biases resulting from training data.

REFERENCES

[1] F. Bre, J. Gimenez, and V. Fachinotti, "Prediction of wind pressure coefficients on building surfaces using artificial neural networks," *Energy and Buildings*, vol. 158, 11 2017.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 06 2017.

[3] R. Gozalo-Brizuela and E. C. Garrido-Merchán, "A survey of generative ai applications," 6 2023.

[4] K. Yang, X. Mao, S. Wang, T. Zhang, B. Lin, Y. Wang, Y. Qin, Z. Zhang, and X. Mao, "Enhancing code intelligence tasks with chatgpt," 12 2023.

[5] W. Ma, S. Liu, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, and Y. Liu, "Chatgpt: Understanding code syntax and semantics," 5 2023.

[6] J. R. Searle, "Is the brain's mind a computer program?," *Scientific American*, vol. 262, 1990.

[7] E. M. Bender and A. Koller, "Climbing towards NLU: On Meaning, Form, and Understanding in the Age of Data," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, 2020.

[8] P. McBreen, "Creativity in software development," in *International Conference on Software Technology: Methods and Tools*, 2001.

[9] M. A. Boden, *The creative mind: Myths and mechanisms: Second edition*. 2003.

[10] Hutchins, E., *Cognition in the Wild.* MIT Press, 1996.

[11] A. Clark, "The Extended Mind. Analysis," *http://www.jstor.org/stable/3328150*, vol. 58, no. 1, pp. 7–19, 1998.

[12] A. A. Markov, "An example of statistical investigation of the text eugene onegin concerning the connection of samples in chains," *Science in Context*, vol. 19, no. 4, p. 591–600, 2006.

[13] T. Almutiri and F. Nadeem, "Markov Models Applications in Natural Language Processing: A Survey," *International Journal of Information Technology and Computer Science*, vol. 14, pp. 1–16, apr 8 2022.

[14] J. Daniel and J. H. Martin, *Speech and Language Processing A.1 Markov Chains*. 2020.

[15] M. L. Mauldin, "Semantic rule based text generation," in *Proceedings of the 22nd annual meeting on Association for Computational Linguistics* -, Association for Computational Linguistics, 1984.

[16] Armin Bauer, Niki Hoedoro, and Adela Schneider, "Rule-based Approach to Text Generation in Natural Language - Automated Text Markup Language (ATML3)," *Challenge+DC@RuleML*, 2015.

[17] J. Hutchins, "The first public demonstration of machine translation: the georgetown-ibm system, 7th january 1954," 01 2004.

[18] P. Picton, "Introduction to neural networks," *Machine Learning Meets Quantum Physics*, 2020.

[19] "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[20] N. C. Thompson, K. H. Greenewald, K. Lee, and G. F. Manso, "The computational limits of deep learning," *ArXiv*, vol. abs/2007.05558, 2020.

[21] D. Yook, H. Lee, and I.-C. Yoo, "A survey on parallel training algorithms for deep neural networks," *The Journal of the Acoustical Society of Korea*, vol. 39, pp. 505–514, 2020.

[22] K. Oh and K. Jung, "Gpu implementation of neural networks," *Pattern Recognit.*, vol. 37, pp. 1311–1314, 2004.

[23] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zhang, "Tensorflow: A system for large-scale machine learning," in *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[24] Y. Bengio, R. Ducharme, and P. Vincent, "A neural probabilistic language model," in *Advances in Neural Information Processing Systems* (T. Leen, T. Dietterich, and V. Tresp, eds.), vol. 13, MIT Press, 2000.

[25] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013.

[26] R. DiPietro and G. D. Hager, *Deep learning: RNNs and LSTM*, pp. 503–519. Elsevier, 2020.

[27] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[29] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, jan 2020.

[30] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, "Improving language understanding by generative pre-training," 2018.

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

[32] M.-F. Wong, S. Guo, C.-N. Hang, S.-W. Ho, and C.-W. Tan, "Natural language generation and understanding of big code for ai-assisted programming: A review," *Entropy*, vol. 25, p. 888, June 2023.

[33] R. Pudari and N. A. Ernst, "From copilot to pilot: Towards ai supported software development," *ArXiv*, vol. abs/2303.04142, 2023.

[34] S. Wang, M. Geng, B. Lin, Z. Sun, M. Wen, Y. Liu, L. Li, T. F. Bissyandé, and X. Mao, "Natural language to code: How far are we?," pp. 375–387, ACM, 11 2023.

[35] R. Wilhelm, H. Seidl, and S. Hack, *Compiler design. Syntactic and semantic analysis*. 11 2013.

[36] W. Ma, S. Liu, M. Zhao, X. Xie, W. Wang, Q. Hu, J. Zhang, and Y. Liu, "Unveiling code pre-trained models: Investigating syntax and semantics capacities," 2024.

[37] A. Clark, D. Chalmers, and D. Chalmers', "The extended mind," 1998.

[38] D. J. Cole, "Richard menary (ed): The extended mind," *Minds and Machines*, vol. 22, pp. 47 – 51, 2011.

[39] J. Levin, "Functionalism," in *The Stanford Encyclopedia of Philosophy* (E. N. Zalta and U. Nodelman, eds.), Metaphysics Research Lab, Stanford University, Summer 2023 ed., 2023.

[40] J. S. Brown, A. Collins, and P. Duguid, "Situated cognition and the culture of learning," *1989*, vol. 18, no. 1, pp. 32–42, 1989.

[41] T. Menzies, "Towards situated knowledge acquisition," *Int. J. Hum. Comput. Stud.*, vol. 49, pp. 867–893, 1998.

[42] R. M. Gersten and S. K. Baker, "Real world use of scientific concepts: Integrating situated cognition with explicit instruction," *Exceptional Children*, vol. 65, pp. 23 – 35, 1998.

[43] M. F. Young, "Instructional design for situated learning," *Educational Technology Research and Development*, vol. 41, pp. 43–58, 1993.

[44] J.-I. Choi and M. J. Hannafin, "Situated cognition and learning environments: Roles, structures, and implications for design," *Educational Technology Research and Development*, vol. 43, pp. 53–69, 1995.

[45] J. Zhang and V. Patel, "Distributed cognition, representation, and affordance," *Pragmatics and Cognition*, vol. 14, no. 2, pp. 333–341, 2006.

[46] J. Hollan, E. Hutchins, and D. Kirsh, "Distributed cognition: toward a new foundation for human-computer interaction research," *ACM Trans. Comput.-Hum. Interact.*, vol. 7, p. 174–196, jun 2000.

[47] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 7 2021.

[48] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang, "Spoc: Search-based pseudocode to code," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[49] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, “Agentcoder: Multi-agent-based code generation with iterative testing and optimisation,” 12 2023.

[50] B. M. A. van Dijk, T. Kouwenhoven, M. R. Spruit, and M. J. van Duijn, “Large language models: The need for nuance in current debates and a pragmatic perspective on understanding,” 2023.

[51] H. Lederman and K. Mahowald, “Are language models more like libraries or like librarians? bibliotechnism, the novel reference problem, and the attitudes of llms,” 2024.

[52] P. Kitcher and D. Dennett, “The intentional stance,” *The Philosophical Review*, vol. 99, p. 126, 01 1990.

[53] J. Andreas, “Language models as agent models,” 2022.