



## **The Impact of Task Specification on Code Generated via ChatGPT**

Bachelor Degree Project in Information Technology  
Basic level 30 ECTS  
Spring 2023

Jonathan Lundblad, Edwin Thörn and Linus Thörn

Supervisor: András Márki

Examiner: Juhee Bae

# Abstract

ChatGPT has made large language models more accessible and made it possible to code using natural language prompts. This study conducted an experiment comparing prompt engineering techniques called task specification and investigated their impact on code generation in terms of correctness and variety. The hypotheses of this study focused on whether the baseline method had a statistically significant difference in code correctness compared to the other methods. Code is evaluated using a software requirement specification that measures functional and syntactical correctness. Additionally, code variance is measured to identify patterns in code generation. The results show that there is a statistically significant difference in some code correctness criteria between the baseline and the other task specification methods, and the code variance measurements indicate a variety in the generated solutions. Future work could include using another large language model; different programming tasks and programming languages; and other prompt engineering techniques.

**Keywords:** Code generation, Task specification, Prompt engineering, ChatGPT, Human evaluation, Code similarity

# Table of contents

<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>2</b>
2.1 Natural Language Processing	2
2.2 Machine Learning	2
2.3 Large Language Models	2
2.3.1 Generative Pre-trained Transformer	3
2.3.2 Codex	3
2.3.3 ChatGPT	4
2.3.4 Tokens	4
2.4 Prompt Engineering	5
2.4.1 Direct Task Specification with Signifiers	5
2.4.2 Task Specification by Demonstration	5
2.4.3 Task Specification by Memetic Proxy	6
2.4.4 Task Specification with Constraints	6
2.4.5 Meta Prompt	7
2.4.6 Serializing Reasoning	7
2.5 Evaluation Metrics	8
2.5.1 Token Match	9
2.5.2 Dynamic Analysis	9
2.5.3 Static Analysis	9
<b>3 Problem</b>	<b>10</b>
3.1 Aim	10
3.2 Motivation	10
3.3 Research Questions	11
3.4 Hypotheses	11
3.5 Objectives	12
3.6 Method	12
3.6.1 Experiment	12
3.6.2 Alternative Methods	13
3.7 Approach	14
<b>4 Related Work</b>	<b>16</b>
<b>5 Implementation</b>	<b>18</b>
5.1 Initial Investigation	18
5.2 Limitations of ChatGPT	19
5.3 Task Specification	20
5.4 Prompt Templates	21
5.5 Data Collection	22
5.6 Data Evaluation	23
<b>6 Results</b>	<b>25</b>
6.1 Experiment Data	25
6.2 Analysis	26
6.3 Code Correctness	28

6.3.1 Data.....	28
6.3.2 Hypothesis A.....	31
6.3.3 Hypothesis B.....	31
6.3.4 Hypothesis C.....	32
6.3.5 Hypothesis D.....	32
6.3.6 Conclusion.....	33
6.4 Comparing Programming Languages.....	34
6.4.1 Data.....	34
6.4.2 Hypothesis E.....	36
6.4.3 Conclusion.....	37
6.5 Code Similarity.....	38
6.5.1 Data Analysis.....	39
6.5.2 Hypothesis F.....	47
6.5.3 Conclusion.....	47
<b>7 Discussion.....</b>	<b>48</b>
7.1 General Discussion.....	48
7.2 Generalization.....	49
7.3 Contributions.....	51
7.4 Ethics & Threats to Validity.....	52
7.4.1 Threats to Validity.....	52
7.4.2 Ethics.....	54
7.5 Future Work.....	55
<b>References.....</b>	<b>57</b>
<b>Appendix A - Prompts.....</b>	<b>1</b>
<b>Appendix B - Software Requirements Specification.....</b>	<b>5</b>
Tic-Tac-Toe.....	5
Blackjack.....	6
<b>Appendix C - Luckycard.....</b>	<b>8</b>
Java.....	8
Python.....	10
<b>Appendix D - Hangman.....</b>	<b>13</b>
Java.....	13
Python.....	14
<b>Appendix E - ConnectFour.....</b>	<b>15</b>
Java.....	15
Python.....	18
<b>Appendix F - Meta Prompts.....</b>	<b>21</b>
Java Tic-Tac-Toe.....	21
Java Blackjack.....	22
Python Tic-Tac-Toe.....	23
Python Blackjack.....	24
<b>Appendix G - Lines of Code Figures.....</b>	<b>27</b>

# 1 Introduction

Code generation using Large Language Models (LLMs) has become more accessible with the release of ChatGPT. The user sends a prompt in the form of a text input in natural language, and the language model generates a response based on it. The design and creation of prompts are known as prompt engineering. The prompt engineering techniques of task specification are methods of using narratives and cultural anchors to encode nuanced intentions (Reynolds and McDonell, 2021). To evaluate code, a software requirements specification can be used to manually or automatically test whether generated code passes the requirement criteria.

There has been active research on the topic of LLMs in recent years, and new developments and technological breakthroughs have been made during the writing of this thesis. Prompt engineering is an important research topic related to LLMs, where a considerable amount of research has been done on using prompt engineering to effectively generate text. This report investigates whether existing research in prompt engineering could be applied to code generation using the LLM, ChatGPT.

To investigate the effectiveness of prompt engineering in code generation, an experiment was conducted comparing the generated solutions of different programming tasks, programming languages, and task specification methods. The study also investigated the code similarity between solutions to identify patterns and support the conclusions. This research is intended to be used by programmers and researchers that use LLMs to generate code. It also gives insight into how prompts can be created and what benefits and drawbacks there are to using the investigated task specification methods.

## 2 Background

This chapter introduces the relevant technical background needed to comprehend the report.

### 2.1 Natural Language Processing

Natural Language Processing (NLP) is, according to Chowdhary (2020), several different computational techniques for automatically analyzing and interpreting human language, i.e., natural language. NLP can vary greatly in complexity depending on how many of these computational techniques are utilized. NLP models excel in tasks such as information retrieval, text categorization, spell-checking, and word-level text analysis. But NLP models struggle with paragraph- and sentence-level text analysis, limiting the models' capabilities in extracting and interpreting information (Chowdhary, 2020).

Challenges with developing capable NLP models are caused by problems of semantics and pragmatics because most natural languages are large and complex, and many words or sentences can have different meanings depending on context, causing ambiguity. Another challenge for NLP is understanding internal representation, i.e., understanding how a human being perceives the world as reflected in natural language (Chowdhary, 2020).

### 2.2 Machine Learning

Machine Learning (ML) is a subsection of Artificial Intelligence (AI), which is the combination of computer algorithms and robust datasets used to accurately make predictions and classifications from input data by mimicking human learning (IBM, 2023a). ML requires mediation from human experts for the algorithm to learn where the difference and meaning of data must be built into the algorithm. Deep machine learning, or deep learning, on the other hand, can learn on its own by recognizing and learning patterns through supervised or unsupervised learning. Supervised learning uses labeled data to learn patterns; labeled data is data given a value by human experts, which the algorithm then assigns relationships to other data based on its given value (IBM, 2023b). Unsupervised learning is a deep learning model that, without labeled instructions or a desired outcome, finds patterns and relationships in data through context, such as affiliated words through a neural network (Salian, 2018). Since unsupervised deep learning algorithms learn without human intervention, it has enabled the use of larger datasets (IBM, 2023b).

### 2.3 Large Language Models

Large Language Models (LLMs) are deep learning language processing algorithms trained using unsupervised learning on large datasets gathered from the internet. LLMs can recognize, summarize, translate, predict, and generate text and code based on their training data (Lee, 2023). LLMs are extensions of language models with the distinction of being trained on larger datasets. Language models assign a probability to a sequence of text based on the model's training data (Hiemstra, 2009). For example, if the training data contains English text, certain words will appear more frequently, such as "the" than a word like "thy", and therefore will be assigned a higher probability. This can be used to perform natural language processing tasks.

LLMs use a transformer-based architecture. Transformer models are neural networks that learn context by applying the mechanism of self-attention, which is a set of mathematical equations that detect and assign relationships between words, which allows the LLM to interpret the meaning of sentences (Vaswani et al., 2017).

### **2.3.1 Generative Pre-trained Transformer**

A Generative Pre-trained Transformer (GPT) is an LLM trained on a large unlabeled dataset based on the deep-learning architecture transformer. The models are generative because of their natural language text-generation capabilities (Radford et al., 2018; Vaswani et al., 2017; Karpathy et al., 2016).

The first GPT model was created to improve the performance of NLP tasks. GPT uses a semi-supervised approach with unlabeled data in unsupervised pre-training and supervised fine-tuning of the model's parameters and weights for specific tasks using labeled data. GPT is a framework for creating task-agnostic models through generative pre-training and discriminative fine-tuning capable of achieving strong natural language understanding (Radford et al., 2018). An LLM being task-agnostic means that the model is not specialized on specific tasks but on every possible task (Brown et al., 2020).

GPT-2 is a more advanced version of GPT with more parameters and training data. GPT-2 was evaluated on language modeling datasets in a zero-shot setting and achieved better results on 7 out of 8 tests compared to comparable models at the time (Radford et al., 2019). Zero-shot is when a task is solved by only using a task description, and few-shot is when a task is solved using a "few" examples (Kojima et al., 2022).

GPT-3 scales up the language models to improve task-agnostic, and few-shot performance. GPT-3 is an Autoregressive Language Model, meaning that the model will predict the next word based on the previous words in the context (Brown et al., 2020). The GPT-3.5 series is a newer version of GPT-3 with additional capabilities such as editing and inserting content into the existing text. (Bavarian et al., 2022).

The newest OpenAI GPT model is GPT-4, which has been benchmarked to be on par with human-level performance in multiple academic and professional fields. When the task complexity reaches a certain threshold, GPT-4 shows greater performance than GPT-3.5, and it can handle more nuanced prompts. Additional features introduced in GPT-4 were visual input with image processing capabilities, which presented a wide range of new possible prompting techniques. GPT-4 with visual input shows similar performance to text-only input (OpenAI, 2023a).

### **2.3.2 Codex**

Codex is a GPT-based large language capable of generating code developed by OpenAI (Chen et al., 2021). One version of Codex is responsible for powering GitHub Copilot, which is a coding assistant tool available for developers in IDEs such as Visual Studio Code (GitHub, 2022). To train Codex, a dataset of Python code was collected from public software repositories hosted on GitHub and filtered to a size of 159 GB (Chen et al., 2021).

To evaluate the model, a dataset called HumanEval was established, which consisted of programming problems with unit tests. The dataset contained 164 original programming

problems that assessed language comprehension, algorithms, and simple mathematics comparable to software interview questions. (Chen et al., 2021).

### **2.3.3 ChatGPT**

ChatGPT is an AI chatbot created by OpenAI based on an LLM in the GPT-3.5 series, which was fine-tuned using Reinforcement Learning from Human Feedback (OpenAI, 2022). ChatGPT gained the ability to replicate the subtleties and intricacies of the human language, and its wide range of training data allows it to engage in dialogue and impart information on a wide range of questions (Susnjak, 2022). Since the model of GPT-3.5 used in ChatGPT was trained in early 2022, it is limited to knowledge from the time of data collection, which would have taken place in 2021 or early 2022 (Hughes, 2023). Though ChatGPT gives well-written answers, there is no fact-checking to determine whether the given answers are truthful. ChatGPT is non-deterministic, which means that given answers can vary when using the same prompt multiple times (OpenAI, 2022).

ChatGPT tends to hallucinate, causing the LLM to fabricate information and present it as truth in responses. These hallucinations can be extrinsic, meaning there is a lack of data to provide correct information, which causes the model to give a fabricated answer based on context and related background knowledge. Hallucinations can also be intrinsic, meaning the model generates a response with false information that contradicts its training data; this is usually caused by flaws in the LLM's design, such as imperfect encoding, erroneous decoding, exposure bias, or parametric knowledge bias (Ji et al., 2023).

Having complex and well-defined prompts can harm ChatGPT's ability to understand and respond (Vasylykiv, 2023). According to Hughes (2023), having complex prompts with many different concepts may lead to ChatGPT ignoring parts of the prompt. This could be because the language model uses in-context learning, which makes it difficult for the language model to perform tasks where it needs to consider previous statements in a prompt (Brown et al., 2020).

### **2.3.4 Tokens**

ChatGPT represents input and output text through tokens; a token can be a word, symbol, blank space, or subword; generally, 100 tokens represent 75 words. Because of technical limitations, ChatGPT has an upper limit on the number of tokens that can be present in the input prompt and output response (Khan, 2023). The token limit for ChatGPT is 4096, counting both input prompts and output responses from the conversation context (OpenAI, 2023b). Each response has a limited number of tokens; when reached, stop text generation.



## 2.4 Prompt Engineering

Prompt engineering is the method of creating prompt functions that result in the most effective performance for the intended task. Prompt template engineering is a common approach in prompt engineering, where a human or an algorithm designs templates for the prompts (Liu et al., 2023). Prompts are text strings used as inputs for a language model to generate responses, and prompt templates are functions that use variables to make reproducible text prompts. The templates can contain instructions, examples, and questions for the language model. In the case of prompt templates being used in an NLP model, the templates are structured using natural language (Harrison, 2022).

Prompt programming is an emerging paradigm where natural language is used to prompt responses from LLMs. Prompt engineering for an NLP model can be conceived as programming in natural language; as such, the term “prompt programming” is used. How a prompt is written has a large impact on the response the model produces (Reynolds and McDonell, 2021).

### 2.4.1 Direct Task Specification with Signifiers

Signifiers are patterns that guide the model to perform a known task. A signifier can be the name of the task, such as “translate” or a compound description, such as “Create a Java application”. The signifier does not explain how to complete the task, rather, it relies on the language model correctly interpreting what it should do (Reynolds and McDonell, 2021). A simple example of this is to use the word “write” as a signifier to ask a question.

---

**Prompt:** *Write the sound a dog makes*

**Response:** *The sound a dog makes is commonly referred to as "bark". It can be written as "woof", "ruff", or "arf" depending on the specific sound the dog is making.*

---

### 2.4.2 Task Specification by Demonstration

Task demonstration consists of two components: context and completion, this is commonly called an example when paired. When writing a demonstration prompt, the question that the prompter wants to be answered is written following the format of the example (Brown et al., 2020).

---

**Prompt:** *Cat: meow. Dog:*

**Response:** *Woof!*

---

In the example, the context is the animal “cat” and the completion is the sound the animal makes “meow”. The question is written as the context “dog” and the response from the model is the completion “woof!”.

Task specification by demonstration can be used for tasks such as translating text from one language to another by providing an example of how a French sentence is translated into English and then specifying the task by writing a French sentence and an English context with the model expected to provide the completion (Brown et al., 2020).

Reynolds and McDonell (2021) argue that it is possible to communicate some tasks to an LLM more effectively by using examples. These tasks include situations where a certain format or language is required, which is easier to demonstrate than to explain in detail.

### 2.4.3 Task Specification by Memetic Proxy

Memetics imitates the concepts of proxies or analogies in human communication. A memetic proxy could be in the form of well-known characters or characteristic situations; these characteristic situations could be, for example, imitating a student, expert, or politician. Using memetic proxies when specifying tasks will provide biases and assumptions related to the proxy, which would be exceedingly difficult to imitate in the form of descriptions and demonstrations (Reynolds and McDonell, 2021).

---

**Prompt:** *As a dog what sound would you make when greeting someone?*

**Response:** *If I were a dog greeting someone, I would likely make a series of high-pitched barks or yips, which are common sounds that dogs make when they are excited or happy. Additionally, I might wag my tail vigorously, jump up to lick the person's face, or wiggle my body with excitement. However, it's important to note that different dogs have different personalities and communication styles, so the exact sounds and behaviors that I would exhibit when greeting someone may vary based on my individual temperament and breed characteristics.*

---

The memetic proxy method is specifically effective for specifying complex open-ended tasks because of GPT-3's inherent ability to effectively use cultural information (Reynolds and McDonell, 2021).

The persona pattern is similar to memetic proxies in that it manipulates the output of an LLM to fit a specific point of view or perspective (White et al., 2023). This can be useful in cases where the user wants a specific behavior, such as conducting a code review with a security expert. By using a persona, the user specifies how the LLM should respond to input in future prompts while in the context where the persona was initiated.

### 2.4.4 Task Specification with Constraints

Constraints can be used in prompts to influence how a model responds. There are various methods to constrain the behavior, such as enclosing the sentence, adding a line break after the sentence, or creating a syntactical constraint where the only reasonable behavior is the intended behavior (Reynolds and McDonell, 2021). Another more direct method is to write out the constraints as a rule, such as "only respond with one word". The example below uses the standard prompt from the signifier example with an added rule to constrain the response, and it can be observed that the rule had the desired effect.

---

**Prompt:** *Write the sound a dog makes. Ensure that the response does not contain any text except the sound the animal makes.*

**Response:** *Woof.*

---

Reynolds and McDonell (2021) argue that the reason LLMs such as GPT-3 fail is that their probability distribution of responses reflects how any person could respond to a prompt, rather than how a specific person would respond. This means that the response could become incoherent as if multiple people tried to respond at the same time. Therefore, it can be helpful to use constraining behavior to force the language model to respond to the intended behavior.

### 2.4.5 Meta Prompt

Meta prompts are proposed by Reynolds and McDonell (2021) as a solution to some of the limitations of prompt programming, such as the difficulty and time investment needed to design prompts for a specific type of task and the lack of automated methods to create prompts. Meta prompts use the language model itself to create specific prompts to solve tasks by combining task details with a so-called “seed” which encapsulates the general intention. The seed is usually quite simple, such as “this problem asks us to” or “write a prompt” followed by the task-specific details.

---

**Prompt 1:** *Write a prompt that can be used by ChatGPT to generate the sound a dog makes.*

**Response 1:** *Sure, here's a prompt:*

*Can you generate the sound a dog makes? Please write the onomatopoeic representation of a dog's bark.*

**Prompt 2:** *Can you generate the sound a dog makes? Please write the onomatopoeic representation of a dog's bark.*

**Response 2:** *Sure, the onomatopoeic representation of a dog's bark is commonly written as "woof" or "arf" in English.*

---

The example above uses the meta prompt concept to first ask the model to create a prompt for a task. The user can then use the response as a second prompt to solve the task.

### 2.4.6 Serializing Reasoning

Certain tasks may be difficult to complete in a single prompt, but solvable if broken up into smaller solvable problems, these tasks often require reasoning. Reynolds and McDonell (2021) assert that when humans try to solve complex close-ended tasks, they will often rephrase the question, outline a procedure, eliminate answers, and reconstruct implicit information into explicit information inside their heads or on paper. When the LLM is

prompted for an answer, the space for reasoning is immediately depleted. An example of a close-ended task could be a math equation or a multiple-choice question. This is what the method of serializing reasoning implies, the prompt is structured in a way that forces the LLM to disclose how an answer to the prompt was determined (Reynolds and McDonell, 2021).

---

**Prompt:** *On a multiple choice questionnaire the question reads "Which of these is least commonly used to describe the sound a dog makes"*

*A: Woof*

*B: Bark*

*C: Ruff*

*D: Arf*

*Explain the reasoning process for answering this question.*

**Response:** *To answer this question, we need to know the meanings of the four options and their usage frequency in the English language.*

- 1. Woof: It is a deep sound that a dog makes, usually associated with larger breeds like Great Danes and Mastiffs. This sound is commonly used to describe a dog's bark.*
- 2. Bark: It is a sharp, explosive sound that a dog makes. This is the most common sound associated with a dog's vocalization.*
- 3. Ruff: It is a low-pitched, rough sound that a dog makes. This sound is less common in dog vocalizations and is often used to describe the sound made by larger animals like lions and tigers.*
- 4. Arf: It is a short, sharp sound that a dog makes. This sound is less common than "bark" but more common than "ruff".*

*Based on this information, we can conclude that the correct answer is likely either "Ruff" or "Arf", as these sounds are less commonly associated with dog vocalizations than "woof" or "bark". However, we need to determine which of the two is the least commonly used.*

*Since "ruff" is a sound that is also associated with other animals like lions and tigers, it is likely that "arf" is the least commonly used sound among the four options provided. Therefore, the correct answer is likely option D: "Arf".*

---

The example above is still solvable in a single prompt but follows the reasoning process outlined. A use case closer to code generation is to ask the language model to create a single class or function, rather than asking it to complete an entire program in a single prompt.

## 2.5 Evaluation Metrics

Various evaluation methods have been used in code generation, as outlined in the systematic review by Dehaerne et al. (2022). The systematic review identifies three categories that evaluation methods can be classified as: token match, dynamic analysis, or static analysis.

### 2.5.1 Token Match

Token match evaluates code by comparing it to a known correct solution called ground truth at the token level. This is done either through NLP-based algorithms or other metrics such as exact match and token accuracy, the score can be affected by the processes used to tokenize the text. Code that is more similar to the ground truth receives a higher score (Dehaerne et al., 2022). One problem is that these metrics do not account for code semantics, which is a problem outlined by both Chen et al. (2021) and Mukherjee et al. (2021). This can result in functionally correct solutions that are written differently than the ground truth receiving a lower score than incorrect solutions which are closer to the ground truth.

Another method of using token match is to compare the similarity between code files. Ragkhitwetsagul et al. (2018) conducted a study comparing code similarity analyzers and their ability to detect code similarities in a range of experimental scenarios. One of the analyzers tested was FuzzyWuzzy<sup>1</sup> which uses the Python library difflib or Levenshtein to perform string matches. The library can handle out of order strings by tokenizing the input and then sorting it alphabetically in order to perform a ratio comparison (Cohen, 2011).

### 2.5.2 Dynamic Analysis

Dynamic analysis evaluates code based on unit tests or functional Software Requirements Specification (SRS). One of the methods of testing functional correctness is using unit tests, where code samples must pass a set of unit tests in order to be considered correct (Dehaerne et al., 2022). Evaluating code correctness using unit tests resembles the process used in test-driven development, where humans write unit tests first and then create code that passes the tests (Chen et al., 2021).

When using SRS to evaluate functional correctness, the requirements should be unambiguous, complete, correct, understandable, and verifiable (Davis et al., 1993). Problems with evaluating functional correctness are that the code needs to be syntactically correct, which is not always guaranteed when generating code. Additionally, it is not feasible to cover every functional requirement when using unit tests or SRS for large code bases.

### 2.5.3 Static Analysis

Static analysis evaluates code based on whether it is syntactically correct and has the advantage of not requiring ground truth or unit tests since the code is not evaluated at runtime (Dehaerne et al., 2022). The problem with this method is that it does not validate whether the code performs the desired functionality. Syntactical correctness is used by Mukherjee et al. (2021) where a code sample is evaluated based on whether it passes the proposed syntactical and semantic criteria. The second type of static analysis is human evaluation, which is a holistic evaluation method that requires experienced programmers and is time-consuming.

---

<sup>1</sup> <https://github.com/seatgeek/thefuzz>

## 3 Problem

This chapter outlines the study's aim followed by the motivation, the research questions, the hypothesis of each research question, and the objectives of the study. Then the method and alternative methods that were considered for the study are presented. Finally, the chapter ends with the approach.

### 3.1 Aim

The aim is to assess prompt engineering techniques, specifically task specifications' effect on code generated with ChatGPT.

### 3.2 Motivation

The release of GPT-3 led to the development of LLMs trained on code such as Codex, which is used in GitHub Copilot to assist programmers by providing machine-generated code suggestions (Brown et al., 2020). Code generation models have shown the ability to solve simple programming tasks in previous studies (Finnie-Ansley et al., 2022; Chen et al., 2021; Denny et al., 2022). The development of LLMs highlights the importance of research into prompt techniques to both evaluate and use models. Much research into prompt engineering is centered around how to write prompts to receive a desired natural language response, such as the “Prompt Programming” methods presented by Reynolds and McDonnell (2021).

LLMs are rapidly improving and are more accessible to the general public with the release of ChatGPT. Previous code generation models such as Codex used docstrings to generate code, the newer ChatGPT can create programs based on natural language prompts, meaning prompt engineering techniques typically used in the generation of natural language can be used in code generation. It would also be interesting to investigate if ChatGPT can solve programming tasks that require more time to solve by a human programmer than those used in previous studies.

The study focuses on whether the chosen task specification methods are effective for generating solutions that are on average more correct than those created using the baseline task specification method of using signifiers. The findings provide insight into how different prompt engineering techniques could be used in the evaluation of code generation models. Another area of focus is the similarity between the solutions generated in the experiment. This is useful for understanding what effects task specification methods may have and the possible limitations of the technology.

### 3.3 Research Questions

To answer the aim of this study, the following research questions have been formulated.

**RQ1:** Is there a difference in the correctness of code generated using the chosen task specification methods compared to direct task specification with signifiers?

**RQ2:** Comparing the programming languages Java and Python, is there a difference in the correctness of code generated using the chosen task specification methods?

**RQ3:** Do solutions generated using the chosen task specification methods result in syntactically varied solutions?

### 3.4 Hypotheses

The null hypotheses of the expected result from the research questions are:

**RQ1:**

**H<sub>0</sub>A:** There is no significant difference in code correctness between solutions generated using task specification with **demonstration** compared to direct task specification with **signifier**.

**H<sub>0</sub>B:** There is no significant difference in code correctness between solutions generated using task specification with **memetic proxy** compared to direct task specification with **signifier**.

**H<sub>0</sub>C:** There is no significant difference in code correctness between solutions generated using task specification with **constraints** compared to direct task specification with **signifier**.

**H<sub>0</sub>D:** There is no significant difference in code correctness between solutions generated using task specification **meta prompt** compared to direct task specification with **signifier**.

**RQ2:**

**H<sub>0</sub>E** There is no significant difference in code correctness between solutions generated in Java compared to Python.

**RQ3:**

**H<sub>0</sub>F:** There is no distinguishable pattern of syntactical variance for solutions generated using the task specification methods.

## 3.5 Objectives

The following objectives are defined to properly answer the three research questions:

1. Research code generation and prompt engineering in LLMs.
2. Investigate ChatGPT.
3. Create a software requirements specification.
4. Prompt ChatGPT using the different task specification methods:
  - a. Signifier.
  - b. Demonstration.
  - c. Memetic proxy.
  - d. Constraints.
  - e. Meta prompt.
5. Evaluate the solutions' code correctness using the software requirements specification.
6. Evaluate the syntactical variance of the solutions using token match.
7. Compare and analyze the results.

## 3.6 Method

The following section details the methods chosen to answer the previously formulated research questions. Alternative methods are also mentioned and are argued against using.

### 3.6.1 Experiment

An experiment was conducted to measure the code correctness of responses generated by ChatGPT using different task specification methods. To answer the first research question, the study used two programming tasks Tic-Tac-Toe and Blackjack to compare direct task specification with signifiers to the other task specification methods. These methods are demonstration, memetic proxy, constraints, and meta prompt. Additionally, the differences between ChatGPT's implementations in Java and Python were compared to answer the second research question. The code similarity was calculated by comparing the tokenized source code using FuzzyWuzzy, the results were then analyzed to answer the third research question.

An experiment was performed as it allowed the researchers to control the environment; the selection, and manipulation of treatments; and measure the effects on outcome variables (Wohlin et al, 2012). Additionally, the experiment is replicable, by using the same environment and treatments. This is important for the study since the LLM, used in the experiment is non-deterministic, meaning a controlled and replicable process is a requirement for testing the hypothesis and answering the research questions.



**Table 1: Table of independent variables**

<b>Independent variable</b>	<b>Values</b>
Programming language	(J) Java (P) Python
Task specification method	(S) Signifier (D) Demonstration (M) Memetic proxy (C) Constraints (R) Meta prompt
Programming task	(T) Tic-Tac-Toe (B) Blackjack

The experiment has three independent variables which are the two programming languages, five task specification methods, and two programming tasks (see Table 1). The combination of independent variables results in 20 different treatments.

### **3.6.2 Alternative Methods**

#### **Case study**

A case study is a method to investigate a phenomenon in a real-life context; it is often difficult to distinguish the phenomenon from its environment (Wohlin et al., 2012). In a case study, researchers collect detailed information using a variety of data collection procedures and different perspectives for analysis. The choice of method between case study and experiment can depend on the scale of the evaluation, the ability to isolate variables, and the ability to use randomization (Wohlin et al., 2012).

The variables for this study are compatible with a case study as they can be isolated from each other, and randomization can be applied. Because the time frame for this study is limited to a couple of weeks, the scale of the evaluation would be rather small, which case studies generally should not be. Another consideration is that case studies only look at variables that represent a typical situation (Wohlin et al., 2012). This means that only some of the values for the variables will be investigated, unlike an experiment where all values are investigated. Therefore, a case study is not the best option for this particular study.

#### **Survey**

An interview is a data-gathering method where a researcher asks human subjects questions that will answer the research questions (Wohlin et al., 2012). Performing an interview was considered in the early stages of the study. The interview would be investigating if the subjects could, after looking at two programs, classify which of the two programs is generated by ChatGPT and created by a human programmer. This method was opted against as the time required to conduct interviews would reduce the size of the experiment.

A survey can also be conducted with a questionnaire which is typically done to receive human reactions to a technique or tool. A questionnaire could be conducted by gathering responses from subjects who are asked to generate solutions for the programming tasks using ChatGPT. These responses can then be measured and compared to the results of the experiment; this would make the conclusions regarding the effectiveness of the task

specification methods more convincing. Instead of conducting a questionnaire, the experiment's sample size was expanded to include the task specification method meta prompt as well as a third research question to investigate code similarity.

### **Systematic Literature review**

Literature analysis is a research method where a problem is comprehensively and systematically examined by analyzing other published research literature. (Berndtsson et al., 2008). This method was ultimately deemed unviable, as Reynolds and McDonell (2021) state in their research, the current prompt engineering methods for controlling LLM are severely limited in the current research literature; there have only been a few years of research since their paper was published. The number of articles available would not be sufficient to conduct a comprehensive systematic literature review, in particular for articles about ChatGPT.

## **3.7 Approach**

The research started by looking into what other studies in the research field of code generation with LLMs had done so far and how their work can be built upon. The previous studies in code generation focused on whether the model could create correct Python code from program descriptions (Finnie-Ansley et al., 2022; Chen et al., 2021; Denny et al., 2022). Similarly, this study could be used to determine whether code generation has an impact on computer science education through students using these models to solve assignments in programming courses. This led to the decision to base the experiment around the University of Skövde's (2020) object-oriented programming course assignment, where students create a program for a simple card or board game in Java. This is the basis for the independent variables programming task and programming language. The programming course assignment also had grading criteria, which is an inspiration for the dependent variable correctness that is used to evaluate the code.

The prompts used to generate code can be written in many ways, which is why techniques from prompt engineering are used to get a more generalizable result. At the start of the study, the researchers could not find a study that had explicitly investigated whether task specification methods used in NLP affected code generation. Thus, the focus of the study was narrowed down to investigating if the choice of task specification methods influences code generation, particularly on the correctness and similarity of the code.

To answer the research questions prompt templates were designed for each of the five task specification methods. The reason for using a template design was so that multiple variations of a prompt could be tested while keeping other variables constant. Prompts created using the prompt templates can be seen in Appendix A.

To generate code, a prompt is sent to ChatGPT inside a new conversation. There is a limited number of tokens received with each response which results in the need to sometimes send a second prompt to receive the remaining code. If ChatGPT crashes during the response it is not possible to send a prompt to generate the missing code, therefore the incomplete code is discarded and the option to regenerate the response is used.

The experiment has three independent variables that can be combined into 20 treatments. Each treatment has three prompt variations, meaning there are a total of 60 prompts. ChatGPT is used to generate ten solutions for each prompt for a total of 600 solutions.

The prompts were sent manually using the free account plan on the February 13 and March 14 versions of ChatGPT. OpenAI states that there were no updates to the free version of ChatGPT in the March 14 update, meaning the solutions should be generated on the same version<sup>2</sup>. The reason for deciding to manually prompt ChatGPT instead of using the API is because prompting in ChatGPT and the API has differences that affect the response, e.g., the use of settings in the API. This study aims to be representative of how ChatGPT may be used in a real-life situation, and the settings used in ChatGPT may differ from those applied in the API. Therefore, ChatGPT was prompted manually instead (OpenAI, 2023b).

The criteria used to evaluate the code correctness were created based on previous work in code evaluation (Chen et al., 2021; Mukherjee et al., 2021). The code generated by ChatGPT is evaluated with human evaluation using the software requirements specification for the two programming tasks Tic-Tac-Toe and Blackjack (see Appendix B). The requirements for the two programming tasks lead to a difference in the functional correctness criteria. Blackjack had more “User interaction” and “Game logic” requirements due to the game’s complexity. Tic-Tac-Toe had a difference in the category “Input validation” due to the input type being integer which requires more validation than Blackjack’s string inputs.

This evaluation data is tabulated in a spreadsheet and is then used to answer the hypotheses related to the first and second research questions by using Fisher’s Exact Tests (FET) to calculate significance values for comparisons determining differences in code correctness between generated code. FET was chosen as the statistical significance test as the gathered evaluation data has two possible values either pass or fail, this is dichotomous data, meaning the data has only two possible categorical values. FET is an exact significance test that is used to determine the contingency correlation between two categorical groups (Khan, 2003). This essentially means that FET determines whether there are dependencies between two dichotomous datasets. If the two datasets are determined to be independent, it means that there is a significant difference in code correctness between the compared datasets. Another reason for choosing FET is that the experiment has a small dataset and, FET is commonly used to test small data samples (Kim, 2017).

To answer the third research question the source code generated by ChatGPT was compared using token match and then analyzed. The comparison works by comparing each file to every other file in the dataset using tokenization. Source code is filtered to remove programming language-specific keywords and characters, leaving only words which are used as tokens. The score is then calculated using the tokens from the reference and candidate files using the Python string-matching algorithm FuzzyWuzzy (Cohen, 2011).

---

<sup>2</sup> <https://help.openai.com/en/articles/6825453-chatgpt-release-notes>

## 4 Related Work

To create the prompt templates for the experiment, the task specification methods specified by Reynolds and McDonell (2021) in their paper, were used. As mentioned in their paper, the application of prompt engineering for LLMs such as GPT-3 is currently in the early stages of research, as these models are relatively new. In their paper Reynolds and McDonell (2021) propose new prompt engineering methods applicable to GPT-3 and their successors. These prompt engineering methods are direct task specification with signifiers, tasks specification by demonstration, task specification by memetic proxy, task specification with constraints, meta prompts, and serializing prompts.

Finnie-Ansley et al. (2022) conducted an experiment where Codex was used to solve introductory Python programming problems from an introductory programming exam. The results from the experiment were compared with student performance on the same exam, which resulted in Codex outperforming most students. Additionally, Finnie-Ansley et al. (2022) investigated how variations in the programming problems affected the results, where Codex could handle most variants investigated. They also investigated the variation of solutions generated from Codex with the same prompts, which resulted in variations in code length, and implementation. The authors also discussed the implications Codex has on computer science education.

The authors Denny et al. (2022) evaluated GitHub Copilot's performance on 166 introductory programming problems, the results of the experiment were that 47.6% of the problems were solved using GitHub Copilot without modifying the programming problems using prompt engineering. And modifying the programming problems resulted in 60.9% of problems being solved with Copilot. The Authors argue that GitHub Copilot in combination with prompt engineering to teach programming will likely have a big influence on the future of programming.

Chen et al. (2021) developed the GPT language model Codex which is trained on code. To evaluate the performance of Codex, an experiment was conducted on 164 programming problems by generating Python code from docstrings which resulted in the model solving 28.8% of the problems. Additionally, repeated code generation with a sample size of 100, resulted in 70.2% of the problems being solved. The authors also investigate the limitations of Codex and discuss the implications of Codex in safety, security, and economics. Chen et al. (2021) also calculated the BLEU score for its generated solutions, comparing them to their HumanEval reference solution counterpart.

White et al. (2023) provides a list of prompt engineering techniques by creating "patterns" also called templates, that can be used to communicate with an LLM. The authors designed a framework for creating prompt patterns, which can be used in many different domains of computer science to generate code with LLMs. The authors also list patterns that have been applied and tested with the LLM ChatGPT. The authors additionally explain how certain patterns can be combined to create better prompts.

Mukherjee et al. (2021) conducted an experiment to evaluate the performance of their neural model using weak supervision from a static program analyzer. The researchers tested the models' performance by giving source code written in Java, with one method removed and having the model generate the missing method, the experiment investigates the model's

understanding of long-distance semantic relationships. The results showed that the researchers' neural model considerably outperforms the compared models that did not use the same training method.

## 5 Implementation

This chapter describes how task specification methods were used for code generation and the process for performing the experiment.

### 5.1 Initial Investigation

Before starting the experiment, several trials were conducted to investigate the feasibility of the prompts, task specification methods, programming tasks, and programming languages.

The choice of prompts and task specification methods was influenced by what was applicable. When it came to selecting methods for task specification, a challenge arose in comprehending and connecting the research articles to code generation. The literature mainly focused on natural language processing rather than code generation, and was centered around GPT-3 rather than ChatGPT. The study aims to implement the task specification methods faithfully to how they were proposed by Reynolds and McDonell (2021).

Multiple programming tasks were tested with a focus on games that students could create during an object-oriented programming course, such as Battleship, Blackjack, Chess (without a computer agent), Connect Four, Hangman, Tic-Tac-Toe, Minesweeper, Pong, and “Luckycard” a card game used in a previous course assignment in University of Skövde’s (2020) Object-Oriented Programming G1F (IT401G) course. The benefit of evaluating games is that they require interactions and have a goal, this makes it easier to determine whether the program fulfills the requirements. To investigate the feasibility of using the games in the experiment, ChatGPT was used to generate code with the signifier prompt. The initial investigation concluded that certain restrictions needed to be placed on the selection of programming tasks. The restrictions used were: The game does not require a GUI. The game has well-established rules. ChatGPT can generate correct solutions for the game. The game is not too simple or too complex.

Battleship was not well-defined or too complex, resulting in games with different goals being created. Chess was also too complex, requiring the prompter to generate the solution in smaller parts. Hangman was too simple and resulted in very few failures. Minesweeper sometimes used GUI, which made comparisons with console-based solutions difficult. Pong uses a GUI. “Luckycard” is not a recognized card game, therefore it requires an explanation of the game rules which makes it a different prompt engineering technique. The decision was made to exclude Connect Four due to it being very similar to Tic-Tac-Toe both games used 2d arrays to represent the game board, both games alternated between players (“X”; ”O”), and both games checked rows, columns and diagonals for a win. This process resulted in Blackjack and Tic-Tac-Toe being the games used as the programming tasks in this study. These games fulfill the restrictions listed above and they are distinct from each other.

The functional correctness criteria are based on the goals, rules, restrictions, and interactions of the two games. The goal of the game Tic-Tac-Toe is to connect three markers in a row, column, or diagonal, for Blackjack it is to have more points than the dealer without going above 21. The complexity of the two games is vastly different. Tic-Tac-Toe is played on a three-by-three board, the players alternate placing markers, the first player to connect three wins, and if the board is filled without a winner it is a tie. Blackjack is a card game with a

player and a dealer. At the start of a round, the player receives two face-up cards; and the dealer has one card face-up, and another hidden. The value of a card is determined by its rank, and for court cards the value is ten; an ace is calculated as 11 points, unless that would make the player go bust, in which case it is counted as one point. The player can hit to take another card or stand to end their round, if the value of the hand exceeds 21 points they go bust and lose. Once the player stands the dealer will reveal their hidden card, and if the value of their hand is below 17 the dealer hits, if it is 17 points or above the dealer stands. The winner is determined by who had the highest value hand without going above 21, if both the player and dealer have the same value it is a push (see Appendix B).

The differences in the games are most noticeable in the functional correctness criteria, since there is a greater number of “User interaction” and “Game logic” requirements for Blackjack compared to Tic-Tac-Toe. There is also a difference in “Input validation” due to the types of user inputs required for the games, Tic-Tac-Toe commonly uses integers to represent the rows and columns which can cause problems if the program does not validate the inputs sent from the user. Blackjack on the other hand has the option to hit or stand and the user inputs their choice, which is typically of the datatype string.

The programming languages Java and Python were chosen and used in the experiment because they are two widely used general-purpose programming languages (Vailshery, L.S., 2022). The programming languages use different syntax, meaning that the same task will be implemented differently which makes it clearer whether the task specification method affects the solution in different scenarios. Another reason for choosing Java is that the University of Skövde’s (2020) Object-Oriented Programming G1F (IT401G) course, which the selection of programming tasks is based on, uses Java. The related work evaluates code generation in Python and to make the results from this study more comparable this study does the same.

## 5.2 Limitations of ChatGPT

One of the previously mentioned limitations of ChatGPT-3.5 was the maximum token limit of 4096 tokens for the prompt and response combined (OpenAI, 2023b). Using the GPT-3 tokenizer tool provided by OpenAI<sup>3</sup>, the number of tokens in prompts and responses from the experiment, reached around half of the token limit. When conducting the demonstration part of the experiment, input prompts above 2000 tokens could not be used to generate code. Usually, ChatGPT would perform a different task such as summarizing the code, but if the demonstration was shortened then code could be generated. This means there is a breaking point for how much code can be included in any given prompt and likely conversation before the token limit is reached.

The cut-off points for answers generated by ChatGPT seemed to be consistent throughout the experiment. This could mean that ChatGPT undershoots the max token limit, as the combined prompt and response reaches only half of the token limit. This could be because OpenAI is trying to reduce the number of potentially long responses, in order to avoid ChatGPT from overloading and increase the response times. Another possibility is that a large portion of the token limit is reserved for providing context from previous prompts and responses. This would explain how the continuation prompts work. The previously generated code is likely provided as context in the continuation prompt, meaning ChatGPT is given the previously generated code and can therefore continue writing the code response.

---

<sup>3</sup> <https://platform.openai.com/tokenizer>

## 5.3 Task Specification

Direct task specification with signifiers uses a signifier, such as “write”, “create”, or “implement” a program with specific functionality, i.e., programming task (see section 2.4.1). The programming task can be a detailed description of the program logic, similar to how a human would convey the task to another person. Another method is to let ChatGPT interpret the program instructions based on the name of the programming task, which requires ChatGPT to have seen the data previously to generate a desired response.

The prompts for task specification with demonstration use context and completion (see section 2.4.2). The implementation of task specification by demonstration is slightly different compared to the literature in that the context explicitly states to write the code instead of simply using the name of the programming task (see Appendix A). Another method of using demonstration would be to give an example of the console output from the program and prompt ChatGPT to write the code based on it. The problem with this method is that it does not demonstrate the task, it instead shows what the expected output of the program is, which does not conform to the principles of the task specification method.

Task specification by memetic proxy can be used to prompt ChatGPT to act as a character when generating code for a specific task (see section 2.4.3). An alternative approach to using task specification with memetic proxy is to combine it with meta prompts to first prompt the language model to mimic a character or characteristic, and then prompt the language model to perform the task (Beurer-Kellner et al., 2022). This alternative was decided against as the approach combines two task specification methods, whereas this study investigates the task specification methods individually.

Task specification with constraints is applied by including a descriptive suggestion in the prompt, constricting the language model’s interpretation of the task and producing a more uniform solution. An example of this is to specify in the prompt to handle “Input validation”, which would consequently affect the answer to the prompt. Initial investigation included a description of the game rules in the prompt, but this was decided against as it was a modification to the variable programming task rather than the task specification method.

Task specification with meta prompt involves using the language model to create a task-specific prompt from an initial seed and then using the response generated by the model as the prompt to solve the task. In code generation, this takes the form of writing a basic prompt that instructs the model to create a prompt that can be used by ChatGPT to generate code for a specific programming task and programming language. The language model then produces a refined prompt which is used in a new conversation to generate code.

Serializing reasoning can be used in code generation to create code by breaking down a complex problem into solvable sub-problems. ChatGPT refused to write a program for the game chess if asked directly, but through serializing it is possible to generate code in smaller steps, such as asking it to write code for the pawn movement. This requires more effort from the prompter compared to the other task specification methods, but it has the advantage of being able to create programs that the other methods cannot. Because the differences between code generation with serializing compared to the other task specification methods differ greatly it was decided against using it in the experiment.



## 5.4 Prompt Templates

Templates were used to ensure that the prompts used in the experiment are consistent across all of the experiments and ensuring that the experiments are reproducible. The templates utilize variables where the different task specification methods, applications, and programming languages are combined into a coherent prompt.

The prompt templates in Table 2 use variables that are annotated as [variable], the values of these variables can be located in Table 3. Some of the variables reference additional variables in their value.

- **Signifier** is the part of the prompt that signifies or indicates to ChatGPT what it should do, the rest of the prompt is specifying details and requirements. Therefore, a signifier is present in every prompt.
- **Code** is the demonstration code that ChatGPT should reference when generating code.
- **Memetic** is the character that ChatGPT should imitate when generating code.
- **Constraint** is a requirement or condition put on the code generated by ChatGPT.
- **Programming Language** is a constraint on the prompt to control which language is used, to have more control over the experiments.
- **Programming Task** is a constraint on the prompt to control what game is implemented, which is also intended to have more control over the experiments.

The prompts are designed to include only the task specification method associated with the experiment and the specific programming language and programming task for the treatment (see Table 2). This is done to ensure that there is a minimal impact from uncontrolled variables and that the comparisons are made on the correct factors for the evaluation of code correctness.

**Table 2:** Table of prompt templates for the different task specification methods.

<b>Task specification method</b>	<b>Prompt template</b>
Signifier	[signifier] [programming task]
Demonstration	Here is the code for the game: [code] Write the code for [programming task]:
Memetic Proxy	[signifier] [programming task] as if you were [memetic]
Constraints	[signifier] [programming task] [constraint]
Meta Prompt	Write a prompt that can be used by ChatGPT to [signifier] [programming task]

**Table 3:** Table of variables and their values, used with the templates in the experiments.

Variable	Values
Signifier	Create a [programming language] application for the game
	Write a program in [programming language] (for/of)
	Using [programming language] implement
Code	Appendix C - [programming language] Luckycard
	Appendix D - [programming language] Hangman
	Appendix E - [programming language] ConnectFour
Memetic	(a) computer science student in their first year
	(a) senior software developer
	(a) computer science teacher giving an example for first year computer science students
Constraint	(,) the application must be object-oriented
	(,) the game must be replayable
	(,) the game handles incorrect user input such as numbers and characters, no code is unused, and the code contains no errors
Programming Language	Java
	Python
Programming Task	Tic-Tac-Toe
	Blackjack

## 5.5 Data Collection

The data collection involved using the prompts seen in Appendix A. To get a representative sample with greater statistical power each treatment has three prompt variations that are used to generate ten solutions each for a total of 30. The code was generated inside a new conversation, meaning no context from previously generated responses was present in the conversation. The generated code was then evaluated using the correctness criteria.

The code generation was occasionally interrupted by ChatGPT's imposed max token response limit. In these cases, ChatGPT was prompted with a second prompt asking the model to continue writing the missing code in its next response. These continuation prompts needed to be executed in the same context as the interrupted code generation response, as the context for the interrupted code was needed to provide the remaining code. The code

generated in ChatGPT's response(s) for each iteration was then fetched and written into Java or Python files depending on the programming language specified in the prompt.

Problems occasionally occurred in the generated code for iterations where a continuation prompt was necessary. These problems were missing semicolons, missing brackets, partially unfinished statements, incorrect spacing, or incorrect indentation. To make sure that the continuation prompts did not affect the results when evaluating the data, small mistakes such as missing semicolons and brackets were manually fixed. Partially unfinished statements needed to be regenerated to show the full statement; this was done by prompting ChatGPT to show the method with the unfinished statement in the context where the problem occurred, and then fetching said statement and replacing the unfinished statement in the file. It is important to note this was only done if there was a problem with the response to the continuation prompt. Spacing and indentation in Python were either fixed manually or by prompting ChatGPT to show parts of the code again. In Java, the files could be reformatted using built-in tools in the Eclipse IDE to fix spacing and indentations.

ChatGPT generated 600 solutions which were saved in an organized file structure. The code files were used to calculate the code similarity to answer the third research question. To answer the first and second research questions, data points for each prompt iteration were gathered and stored in a spreadsheet. The first data points were binary evaluation data of the code evaluated against the correctness criteria, as seen in section 5.6. The code could either pass or fail a correctness criterion. Furthermore, there was a data point for Lines of Code (LOC) and lastly, a section for notes providing brief explanations of notable errors or variations.

## 5.6 Data Evaluation

The Software Requirement Specification (SRS) (see Appendix B) was used to evaluate the correctness of code generated by ChatGPT. The evaluation of code correctness with SRS uses a human evaluation method based on the methods presented by Chen et al. (2021) and Mukherjee et al. (2021). The SRS follows the quality metrics for SRS provided by Davis et al. (1993). The designed SRS was then divided into correctness criteria which cover functional correctness and syntactical correctness.

Two separate SRS were designed for the two programming tasks Tic-Tac-Toe and Blackjack. The SRS for the programming tasks was created using a simple interpretation of the game rules for each of the programming tasks. For Tic-Tac-Toe the definition and rules proposed by The Exploratorium (2000) were used, and for an interpretation of Blackjack, the rules by Bicycle Cards (2023) were followed. As previously mentioned, the requirements for the programming tasks were created while following quality standards (Davis et al., 1993). When code is evaluated against the correctness criteria, the code is compared to the requirements specified in the SRS for the relevant programming task. This means that if the SRS does not have full coverage of the programming task-specific game rules, there is no guarantee that the code fulfills the game rules.

The correctness criteria have three categories for functional correctness: "User interaction", "Game logic", and "Input validation". And two categories for syntactical correctness: "Code errors", and "Unused code". The first functional correctness criterion "User interaction" covers the requirements responsible for displaying the program state to the user. The second

criterion “Game logic” covers the implementation of program logic such that it is accurate to the goals and rules of the game. The third criterion is “Input validation” which covers the requirements for handling user input, which involves parsing the input and gracefully handling wrong input by either using standard values or prompting the user for new input. The first syntactical correctness criterion is “Code errors” which encompasses requirements related to having code free of syntax errors and runtime errors. And the second criterion “Unused code” covers the requirements related to ensuring that there were no unused variables or methods.

## 6 Results

In this chapter, the evaluation data of generated code and source code comparisons are used to answer the research questions by visualizing and analyzing the data using different methods and then concluding the research questions based on the results from the analysis.

### 6.1 Experiment Data

To identify each experiment treatment, the treatments are labeled with abbreviations indicating the values of the independent variables used (see Table 1). An example of an abbreviation is “JST” which stands for the programming language “Java”, the task specification method “Signifier”, and the programming task “Tic-Tac-Toe”.

**Table 4:** Heat map showing the criteria passed for each treatment.

Treatment	User interaction	Game logic	Input validation	Code errors	Unused code
JST	83%	80%	77%	100%	100%
JSB	53%	60%	90%	90%	93%
PST	87%	83%	60%	87%	100%
PSB	73%	60%	97%	97%	93%
JDT	90%	90%	97%	100%	97%
JDB	53%	63%	100%	80%	90%
PDT	97%	90%	80%	90%	100%
PDB	57%	73%	100%	83%	93%
JMT	87%	77%	80%	97%	97%
JMB	63%	47%	97%	77%	57%
PMT	90%	70%	70%	97%	97%
PMB	67%	67%	100%	93%	93%
JCT	80%	83%	100%	100%	100%
JCB	53%	70%	100%	90%	93%
PCT	87%	100%	70%	93%	100%
PCB	80%	73%	100%	87%	100%
JRT	93%	83%	97%	97%	97%
JRB	93%	60%	67%	90%	93%
PRT	90%	77%	73%	97%	100%
PRB	73%	47%	47%	77%	90%

Table 4 displays a heat map of the applied treatments and the correctness criteria passed for each of the treatments. Each treatment includes three prompt variations that were prompted ten times, meaning that the maximum number of criteria passed, in each criterion for each treatment is 30. The colors in the heat map indicate differences in the percentage of criteria

passed where white is the highest percentage, red is the minimum percentage, and yellow is the midpoint between the min and max values.

## 6.2 Analysis

In this section, the statistical analysis test and the statistic for determining sample diversity are presented. These will be used in the upcoming data analysis for the first and second research questions.

Fisher’s Exact Test (FET) is a statistical significance test appropriate to use when dealing with small sample sizes and when the data points are independent of each other. FET is an exact test, meaning that rather than calculating an approximation as many other statistical tests such as the ANOVA test, FET calculates an exact value (Kim, 2017). FET calculates the exact probability p-value of a relationship between two dichotomous variables. If the p-value is less than the significance level, which is commonly 0.05, the relationship shows a statistically significant difference (Khan, 2003). The significance level of 0.05 was chosen as the experiment has a small sample size, a lower significance level (e.g. 0.01) would require a larger sample size.

An alternative to FET is the Chi-square test which calculates an approximation of the significance of the population (Kim, 2017), while FET calculates the exact significance of the sample, an approximation of the population provides higher statistical power than only using the sample. The reason for not using the Chi-square test is that larger sample sizes are needed for an adequate approximation, and Chi-square assumes the sample is normally distributed. Since the sample of this experiment is small and not normally distributed, Chi-square cannot be used.

Another reason for using FET as the statistical test in this experiment is because it is a non-parametric method that does not assume a normal distribution of the experiment data (Khan, 2003). The data from this experiment is non-normally distributed, meaning parametric tests such as the ANOVA and t-test that assume data to be normally distributed cannot be used (le Cessie, 2020).

To calculate the p-value using FET, a contingency table was created to represent the data, the row represents the variables, and the column represents the binary values for the variable (Khan, 2003).

**Table 5:** Contingency table deriving variables used in p-value calculation formula.

	Present (+)	Absent (-)	Total (Row)
Variable 1	$x_1$	$x_2$	$t_1 = x_1 + x_2$
Variable 2	$x_3$	$x_4$	$t_2 = x_3 + x_4$
Total (Column)	$t_3 = x_1 + x_3$	$t_4 = x_2 + x_4$	$X = x_1 + x_2 + x_3 + x_4$

$x_1, x_2, x_3,$  and  $x_4$  is the frequency,  $t_1,$  and  $t_1$  is the rows’ total value,  $t_3$  and  $t_4$  is the column’s

total value, and X is the total value.

After deriving the variables (see Table 5) the p-value can be calculated (Khan, 2003):

$$P = \frac{t_1! \cdot t_2! \cdot t_3! \cdot t_4!}{X! \cdot x_1! \cdot x_2! \cdot x_3! \cdot x_4!} \quad (1)$$

To further analyze the hypotheses, each different experimental treatment was analyzed by calculating the code correctness similarities between the different treatments related to the hypotheses. This comparison was conducted by calculating the Jaccard index (JI) of the code correctness criteria between the compared treatments (Costa, 2021). JI is calculated using the following formula:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \quad (2)$$

Variables A and B in the JI formula represent the datasets of the compared treatments. The formula gives a similarity coefficient between the compared treatments, the similarity coefficients were calculated for every treatment pair relevant to the hypotheses. JI is useful for the analysis because the similarity coefficient between two samples makes differences identifiable.

A drawback with using JI is that the coefficients are non-directional, meaning two different comparisons could have an increase and decrease of the same amount of criterion passed, and they will have the same similarity coefficients. This caused certain treatment comparisons with low similarity coefficients in different directions to cancel out and not be shown in FET. The compared treatments' percentage of criteria passed was compared to determine the direction of the Jaccard similarity coefficients (see Table 4).

## 6.3 Code Correctness

This section answers the first research question by presenting data visualizations showing the code correctness for every task specification method. These visualizations are two bar charts, one showing the code correctness criteria and the other showing the code correctness by functional and syntactical correctness. Analysis of each hypothesis is performed using Fisher's exact test and Jaccard index. The results from the hypothesis analysis are used to conclude the research question.

### 6.3.1 Data

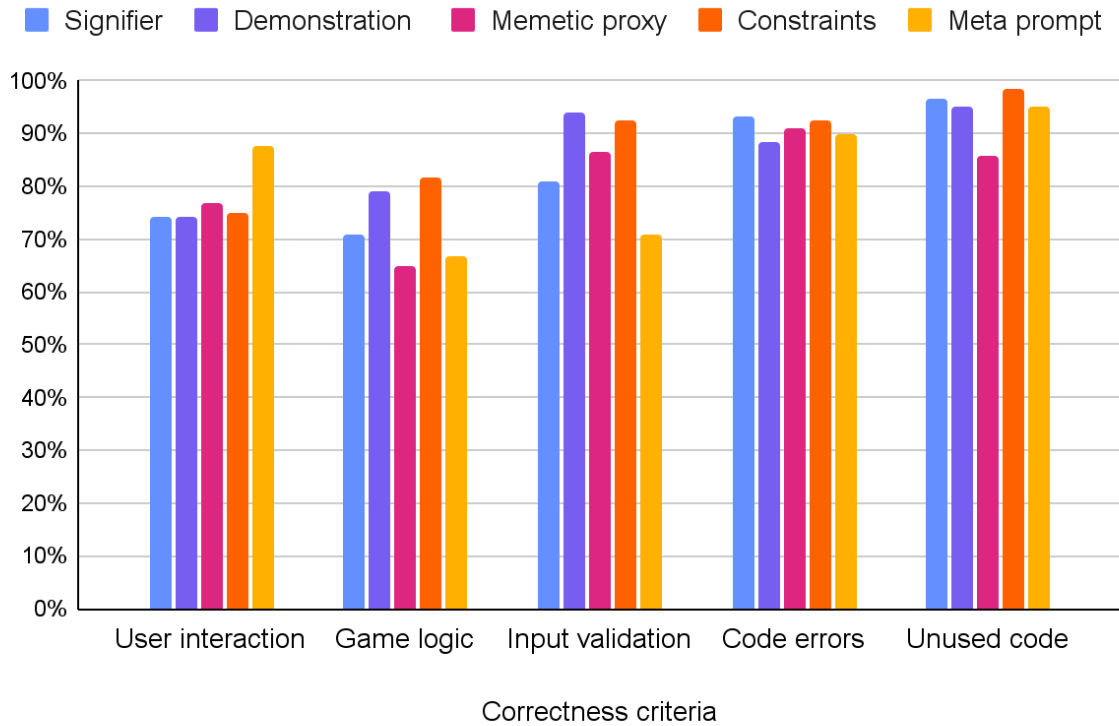
**Table 6:** Fisher's exact test results for the hypotheses, showing significance under 0.05.

Fisher's exact	User interaction	Game logic	Input validation	Code errors	Unused code
(Signifier, Demonstration)	1.0000	0.1794	0.0029	0.2631	0.7486
(Signifier, Memetic proxy)	0.7645	0.4068	0.2937	0.6336	0.0050
(Signifier, Constraints)	1.0000	0.0682	0.0126	1.0000	0.6835
(Signifier, Meta prompts)	0.0133	0.5777	0.0967	0.4844	0.7486

The calculated p-values from FET for the hypotheses related to the first research question can be seen in Table 6. The p-values were calculated for each of the five correctness criteria to answer the hypotheses. The p-values are highlighted in red to show statistical significance when the value is under 0.05, which is the value for a 95% confidence interval. This means there is a significant difference in that specific criterion between the baseline signifier solutions and those from the chosen task specification methods.



## Percentage of correctness criteria passed divided by task specification method



**Figure 1:** Bar chart showing correctness criteria passed for each task specification method.

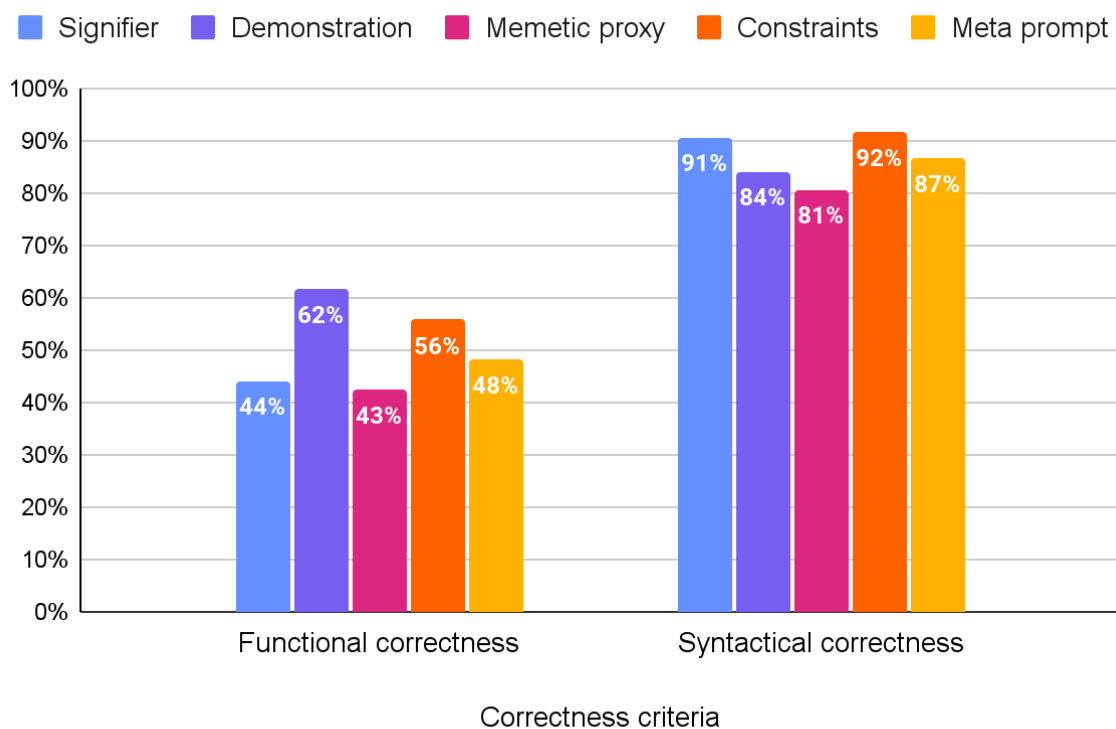
**Table 7:** Average percentage of correctness criteria passed.

Task specification method	User interaction	Game logic	Input validation	Code errors	Unused code
Signifier	74%	71%	81%	93%	97%
Demonstration	74%	79%	94%	88%	95%
Memetic proxy	77%	65%	87%	91%	86%
Constraints	75%	82%	93%	93%	98%
Meta prompt	88%	67%	71%	90%	95%

Figure 1 shows the percentage of code correctness in each correctness criterion divided by which task specification methods were used to generate the code. Table 7 is a complementary table containing the percentage values for each bar in Figure 1. A high code correctness percentage indicates few failures in that criterion for that task specification method, and a low code correctness implies more failures. An interesting observation is that there is higher code correctness for syntactical correctness, “Code errors” and “Unused code”, compared to functional correctness, “User interaction”, “Game logic”, and “Input validation” (see Figure 2).

All of the p-values showing significance in Table 6 are reflected in Figure 1. Solutions generated using signifiers compared to demonstration prompts show a significant difference in the code correctness criterion “Input validation”, with demonstration having higher code correctness. Solutions generated with signifiers compared to memetic proxy prompts show a significance in “Unused code”, where signifier solutions had higher correctness. Signifier solutions compared to constraints solutions show a statistical difference in “Input validation”, and constraints had higher correctness. Signifiers compared to meta prompts are statistically different in “User interaction” with solutions generated with meta prompts having higher code correctness.

### Percentage of correctness criteria passed divided by task specification method



**Figure 2:** Bar chart showing the percentage of functional and syntactical correctness for solutions divided by task specification method.

Figure 2 shows the percentage of functional and syntactical correctness for solutions generated using the different task specification methods. There is overall higher syntactical correctness than functional correctness. There is a higher functional correctness for every task specification method when compared to the baseline signifier, except for memetic proxy. For syntactical correctness, there is a lower correctness for every other method except for constraints when compared to the baseline.

### 6.3.2 Hypothesis A

FET shows that there is a significant difference between direct task specification with signifiers and task specification with demonstration for the correctness criteria “Input validation” (see Table 6). The Jaccard similarity coefficient (JI) for “Input validation” shows that the largest difference comes from the solutions generated for the programming task Tic-Tac-Toe (see Table 8). The difference in the correctness criteria “Input validation” is significant enough to reject the null hypothesis and the following alternative hypothesis was accepted:

*“There is a significant difference in the correctness criterion ‘Input validation’ between solutions generated using Demonstration compared to Signifier.”*

The criteria “Game logic” has a low p-value in the FET table, with task specification method demonstration having fewer failures than signifiers in all the comparisons (see Table 4). However, the difference was not statistically significant enough to reject the null hypothesis. The “User interaction” criteria had a p-value of 1 since there was an equal number of failures for both task specification methods.

**Table 8:** Hypothesis A - Jaccard similarity coefficients.

Treatment comparison	User interaction	Game logic	Input validation	Code errors	Unused code
JDT + JST	0.88	0.82	0.67	1.00	0.94
JDB + JSB	1.00	0.94	0.82	0.82	0.94
PDT + PST	0.82	0.88	0.67	0.94	1.00
PDB + PSB	0.71	0.76	0.94	0.76	1.00

### 6.3.3 Hypothesis B

The FET results show a significant difference in the correctness criteria “Unused code” for prompts using task specification with memetic proxy compared to direct task specification with signifiers (see Table 6). The other correctness criteria did not show sufficient differences in FET when comparing memetic proxy with signifiers. As there was a statistically significant difference in code correctness for one of the correctness criteria, the null hypothesis was rejected, and the following alternative hypothesis was accepted:

*“There is a significant difference in the correctness criterion ‘Unused code’ between solutions generated using Memetic proxy compared to Signifier.”*

From the JI comparing memetic proxy with signifiers (see Table 9), there could be seen that most of the difference in code correctness for the “Unused code” criteria was caused by prompts using memetic proxy compared with signifier to generate Blackjack in Java (JMB + JSB). The dissimilarity in “Unused code” was the result of prompts using memetic proxy generating a high frequency of object-oriented solutions where similar variations of the unused method “*getSuit()*” were repeatedly included in the solutions.

**Table 9: Hypothesis B - Jaccard similarity coefficients.**

Treatment comparison	User interaction	Game logic	Input validation	Code errors	Unused code
JMT + JST	0.94	0.94	0.94	0.94	0.94
JMB + JSB	0.82	0.76	0.88	0.76	0.46
PMT + PST	0.94	0.76	0.82	0.82	0.94
PMB + PSB	0.88	0.88	0.94	0.94	1.00

### 6.3.4 Hypothesis C

The FET results indicate a significant difference in correctness criteria for “Input validation” between task specification with constraints and direct task specification with signifiers. While the other correctness criterion indicates that there is no significant difference (see Table 6). Because at least one of the correctness criteria has a significant difference in code correctness, the null hypothesis is rejected, and the following alternative hypothesis was accepted:

*“There is a significant difference in the correctness criterion ‘Input validation’ between solutions generated using Constraints compared to Signifier.”*

When using JI to compare constraints with signifiers, there was no observed significant variation between the two methods, as the coefficient is above the upper critical value for every treatment comparison (see Table 10), meaning that every comparison for the treatments has a pronounced similarity between them. There is however a treatment comparison that sticks out from the rest, and it is for the prompts that create Tic-Tac-Toe in Java (JCT & JST) for the “Input validation” criteria. It can be observed that there is a sizable difference between the two treatments compared to other comparisons, but not large enough for it to be considered a notable difference.

**Table 10: Hypothesis C - Jaccard similarity coefficients.**

Treatment comparison	User interaction	Game logic	Input validation	Code errors	Unused code
JCT + JST	0.94	0.94	0.62	1.00	1.00
JCB + JSB	1.00	0.82	0.82	1.00	1.00
PCT + PST	1.00	0.71	0.82	0.88	1.00
PCB + PSB	0.88	0.76	0.94	0.82	0.88

### 6.3.5 Hypothesis D

The results from FET show a significant difference in code correctness between meta prompt and signifiers in the “User interaction” criterion (see Table 6). The other correctness criteria did not indicate a significant difference in code correctness between meta prompt and signifiers. As at least one of the correctness criteria showed a significant difference in code correctness, the null hypothesis was rejected, and the following alternative hypothesis was accepted:

“There is a significant difference in the correctness criterion ‘User interaction’ between solutions generated using Meta prompt compared to Signifier.”

When comparing the results from meta prompt and signifiers using JI (see Table 11), it can be seen that JRB and JSB have the most dissimilarities in code correctness for “User interaction”. This dissimilarity is because 93% of Java Blackjack solutions generated by meta prompts passed the “User interaction” criteria, while Java Blackjack solutions generated with signifiers had a 53% pass rate (see Table 4). The JI in Table 11 also shows a high dissimilarity in “Input validation” between treatments PRB and PSB. The reason is that PRB had a score of 47%, whereas PSB had a score of 97% in “Input validation” (see Table 4). The high amount of failures for meta prompts was caused by the inclusion of betting functionality in the solutions, where the program expected the input was an integer without proper validation.

The other correctness criteria show high JI similarity between treatments which indicates low difference in code correctness for the correctness criteria, these criteria do not contribute to the rejection of the null hypothesis.

**Table 11: Hypothesis D - Jaccard similarity coefficients.**

Treatment comparison	User interaction	Game logic	Input validation	Code errors	Unused code
JRT + JST	0.82	0.94	0.67	0.94	0.94
JRB + JSB	0.43	1.00	0.62	1.00	1.00
PRT + PST	0.94	0.88	0.76	0.82	1.00
PRB + PSB	1.00	0.76	0.33	0.67	0.94

### 6.3.6 Conclusion

The hypothesis tests and analysis indicate a significant increase in code correctness for the criterion “User interaction” when comparing meta prompts to signifiers, meaning at least one task specification method showed a notable difference in code correctness for “User interaction”.

There was not a significant difference in code correctness for any of the task specification method comparisons of the criteria “Game logic” and “Code errors”. Therefore, it can be concluded that using different task specification methods compared to the baseline does not yield better or worse results in the aforementioned criteria.

The analysis showed that there was a statistically significant difference in code correctness when comparing memetic proxy with signifier prompts in the correctness criterion “Unused code”. This difference was a decrease in correctness in “Unused code” for memetic proxy compared to signifiers.

Finally, the two task specification methods demonstration and constraints compared with signifiers had a significant difference in correctness for the criterion “Input validation”. Both demonstration and constraints had higher correctness compared to signifiers.

## 6.4 Comparing Programming Languages

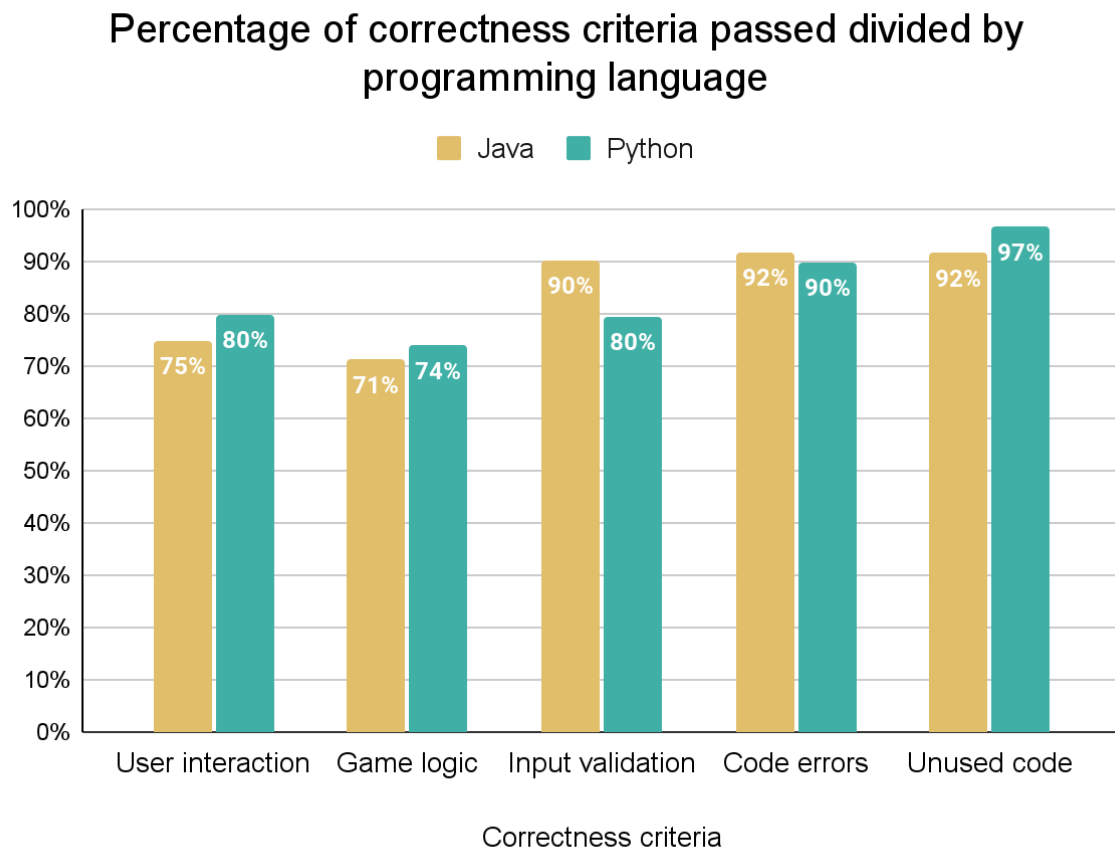
In this section, the second research question is answered. Data visualizations showing the code correctness of the programming languages Java and Python are presented. An analysis is performed on the hypothesis related to the second research question by using FET and JI. With the results from the analysis, a conclusion is formed for the research question.

### 6.4.1 Data

**Table 12:** Fisher's exact test results for the hypotheses, showing significance under 0.05.

Fisher's exact	User interaction	Game logic	Input validation	Code errors	Unused code
(Java, Python)	0.1710	0.5214	0.0004	0.4760	0.0137

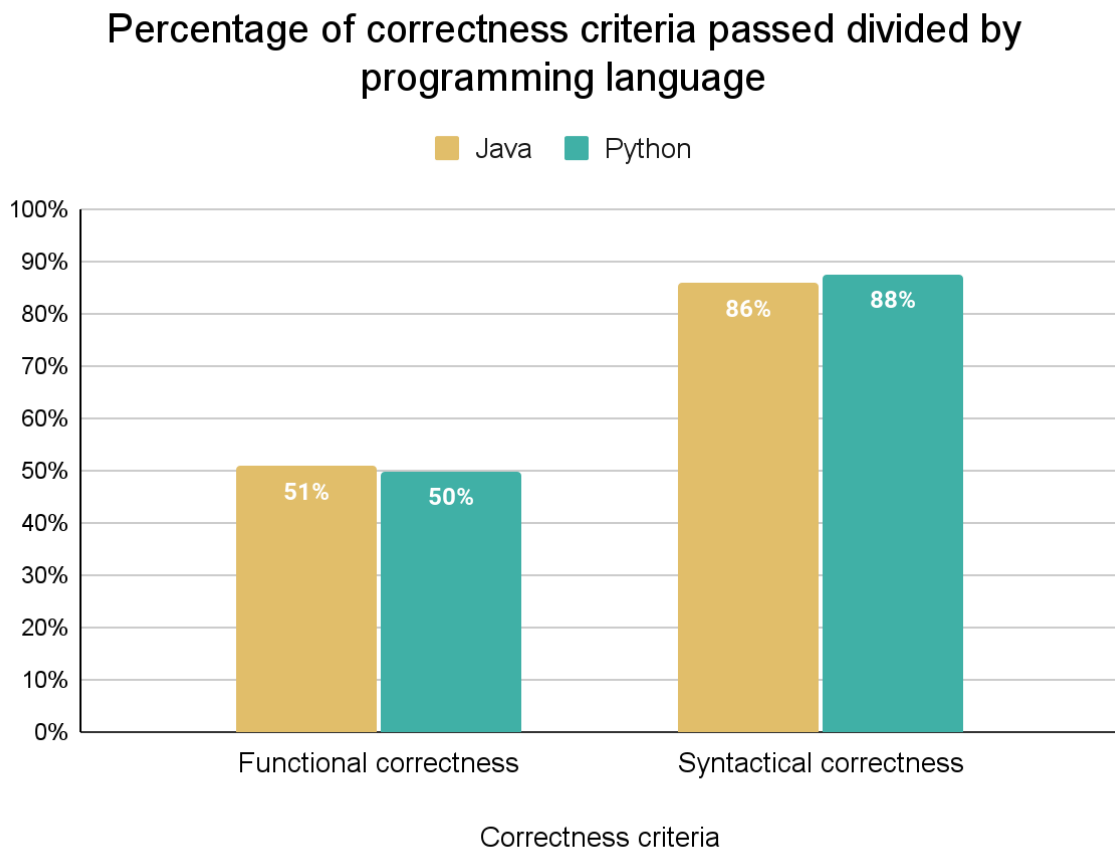
Calculated p-values from FET for the hypothesis related to the second research question can be seen in Table 12. There are five calculated p-values for the hypothesis; one p-value for each correctness criterion. The p-values highlighted in red indicate a statistical significance difference in code correctness with a significance level of 0.05. The significant p-values only indicate a statistically significant difference in that specific code correctness criterion when comparing solutions generated in Java and Python.



**Figure 3:** Bar chart showing correctness criteria passed for Java and Python.

Figure 3 shows the percentage of correctness criteria passed for all solutions generated with the programming languages Java and Python. The results are similar in most criteria with the largest differences being Java having more failures in “Unused code”, and Python having more failures in “Input validation”. The difference in “Unused code” can be largely attributed to the Java Memetic proxy Blackjack (JMB) treatment which had the highest number of “Unused code” failures by a significant margin.

The p-values of the correctness criteria “Unused code” and “Input validation” in Table 12 show a significant difference in code correctness when comparing solutions generated in Java and Python. These significance values can be seen in Figure 3, with Python having higher code correctness in “Unused code”, and Java having higher code correctness in “Input validation”.



**Figure 4:** Bar chart showing the percentage of functional and syntactical correctness for solutions divided by programming language.

Figure 4 shows the percentage of functional and syntactical correctness for solutions in Java and Python. There is a higher syntactical correctness compared to functional correctness for both Java and Python. And there are minimal differences between the programming languages in both functional and syntactical correctness.

## 6.4.2 Hypothesis E

The results from FET determining whether prompts using Java or Python differentiate in code correctness (see Table 12), show that the correctness criteria “Unused code”, and “Input validation” have significant differences. The results show that there is no significant difference in code correctness for the correctness criteria “User interaction”, “Game logic” and “Code errors”. Because more than one of the correctness criteria had statistically sufficient differences in code correctness, the null hypothesis was rejected, and the following alternative hypothesis was accepted:

*“There is a significant difference in the correctness criteria ‘Input validation’ and ‘Unused code’ for solutions generated in Java compared to Python.”*

The JI of code correctness (see Table 13) shows the similarity coefficients between prompts generating solutions for different programming languages with the same task specification method and programming task. JI shows no apparent correlation between the usage of different programming languages, rather it highlights the extreme values that considerably skew the results of FET. The JI of “User interaction” are considerably varied with outliers from Java and Python, which is likely the cause for the insufficient difference in code correctness. The significant difference in “Unused code” is shown by JI to be almost solely because of the difference between JMB and PMB, as the other JI for “Unused code” have high similarities. JMB has more than 6.5 times as many failures in “Unused code” compared to PMB. Lastly the statistically significant difference of “Input validation” is a result of fewer failures in Java compared to Python and is mostly present in the JI coefficients for comparisons of Tic-Tac-Toe solutions, while Blackjack solutions have little to no differences when comparing Java and Python solutions, except for meta prompt Blackjack.

The analysis of JI does not show consistent patterns of differences in code correctness for any of the correctness criteria when comparing programming languages. The JI shows that there exist prompts that skew the results, and the correctness criteria “Input validation” shows a consistent pattern depending on the programming task used rather than the programming language used, which is most likely caused by the difference in input types as Tic-Tac-Toe expects integer values and Blackjack almost only expects string inputs. The solutions of Blackjack which expected both string and integer inputs were solutions that had betting functionality where the user inputs an integer value. In those cases, if the user wrote a string instead of an integer, the program would crash, thus resulting in an “Input validation” failure.



**Table 13:** Hypothesis E - Jaccard similarity coefficients.

Treatment comparison	User interaction	Game logic	Input validation	Code errors	Unused code
JST + PST	0.94	0.94	0.71	0.76	1.00
JSB + PSB	0.67	1.00	0.88	0.88	1.00
JDT + PDT	0.88	1.00	0.71	0.82	0.94
JDB + PDB	0.94	0.82	1.00	0.94	0.94
JMT + PMT	0.94	0.88	0.82	1.00	1.00
JMB + PMB	0.94	0.67	0.94	0.71	0.46
JCT + PCT	0.88	0.71	0.54	0.88	1.00
JCB + PCB	0.58	0.94	1.00	0.94	0.88
JRT + PRT	0.94	0.88	0.62	1.00	0.94
JRB + PRB	0.67	0.76	0.67	0.76	0.94

### 6.4.3 Conclusion

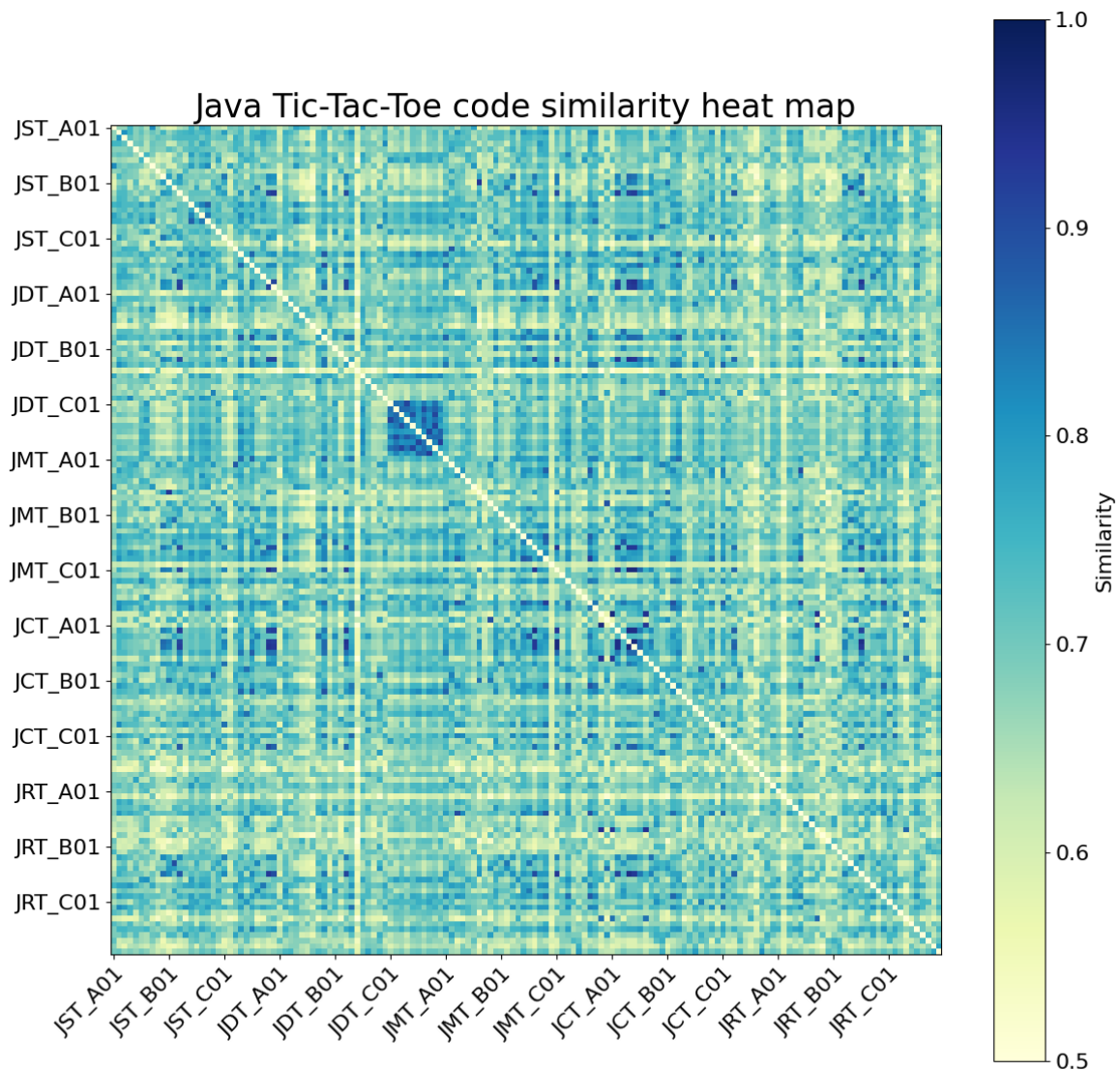
The statistical tests of code correctness between solutions generated in Java and Python indicate a significant difference in code correctness criteria “Unused code”, and “Input validation”. The significant differences in code correctness for these two correctness criteria are caused by separate comparisons. For “Unused code”, Java was the cause for the difference; and for “Input validation”, Python. There were no significant differences for the other three criteria. Because no consistent pattern of differences between the correctness criteria could be observed it cannot be concluded whether the choice of programming language has a positive or negative effect on code correctness as a whole.

## 6.5 Code Similarity

To answer the third research question, the code similarity was calculated using FuzzyWuzzy with the tokenized source code and then presented in two formats: heat map and proximity graph. The heat map shows the code similarity between all the solutions in the form of a correlation matrix. Each pixel represents a pair of solutions, and the ticks are placed on every tenth solution. This means that between the first tick JST\_A01 and the second tick JST\_B01, there are 10 solutions JST\_A01 through JST\_A10 (Figure 5). The heatmap has a range from 0.5 to 1.0, this is to account for the high similarity scores that FuzzyWuzzy produces.

The proximity graph uses nodes and edges to connect solutions with a high level of similarity together. The edge weight and color are based on whether the score exceeds a specific threshold. Strong matches have a similarity of 75% or higher are colored black and have a weight equal to the score, this ensures similar solutions are grouped together. Close matches with a code similarity of 67.5% or higher are colored light gray and have a weight equal to a fourth of the score, ensuring related solutions are closer to each other. All other edges are white and have a reduced weight of 1/20th the score to ensure they are further apart but still form a structured graph with the spring layout from Matplotlib.

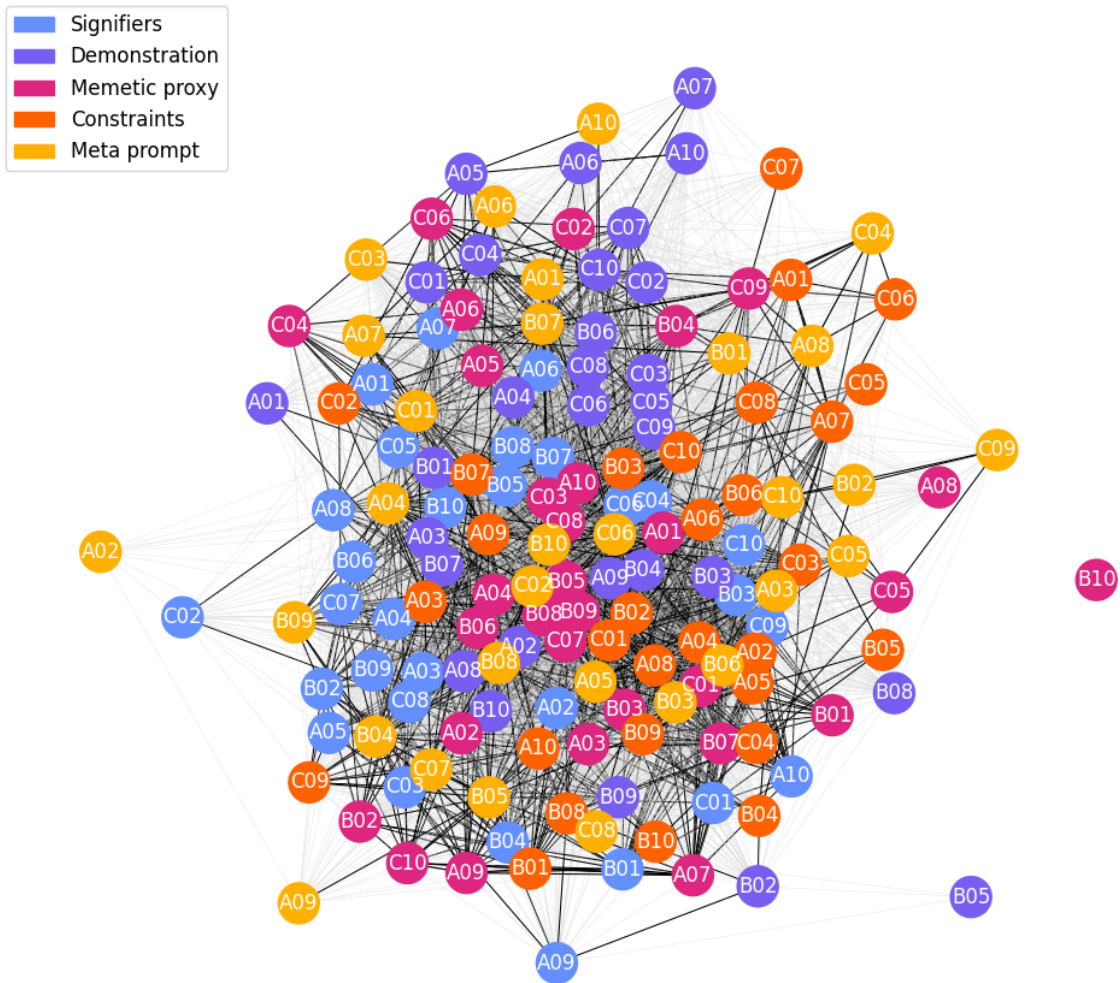
## 6.5.1 Data Analysis



**Figure 5:** Java Tic-Tac-Toe code similarity heat map.

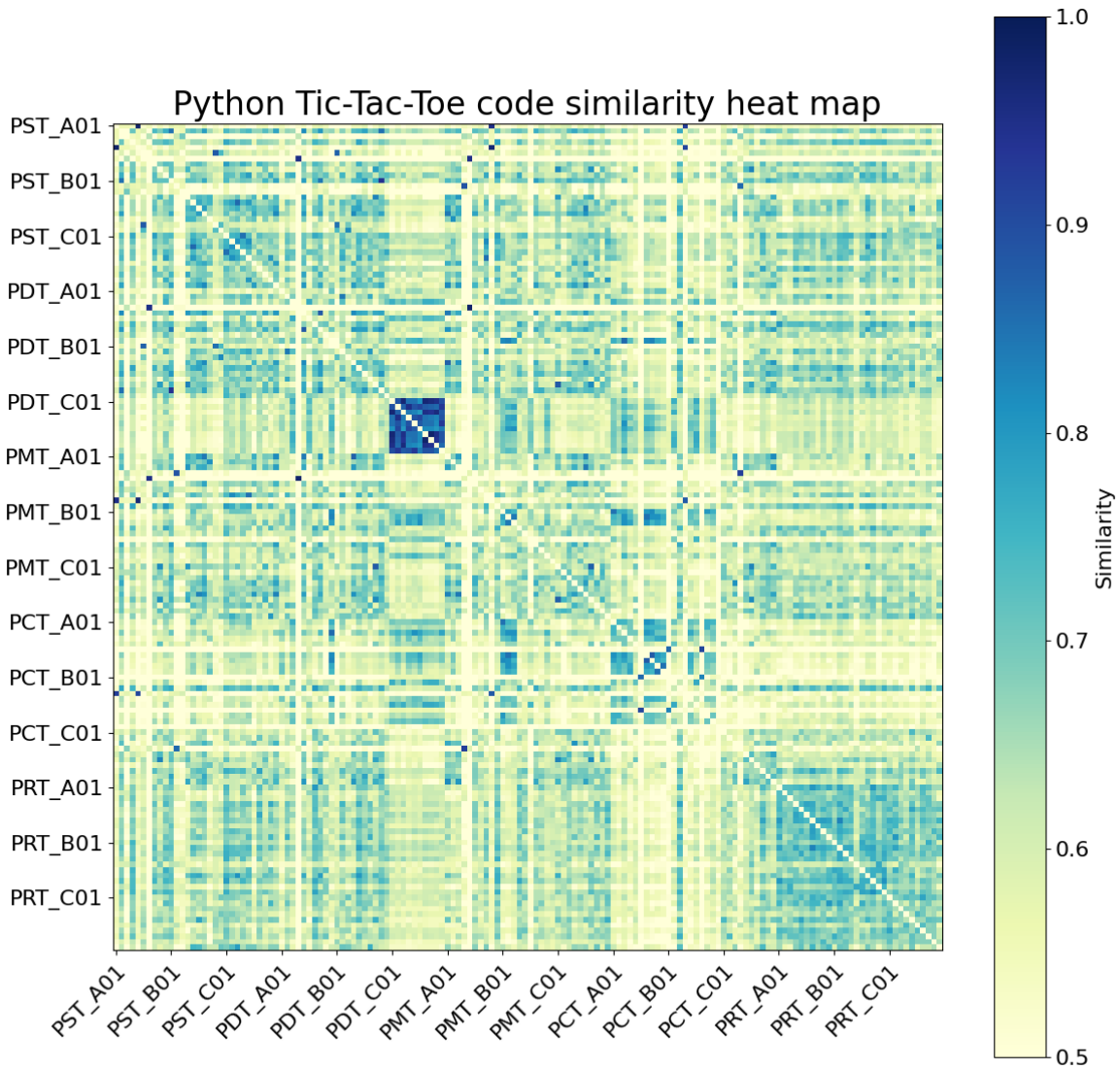
Figure 5 shows how similar the solutions for Java Tic-Tac-Toe were. Solutions with a high level of similarity occurred even when using different task specification methods, as seen by the dark blue dots between JCT\_A01 and JCT\_B01. These solutions had minor changes compared to each other, which suggests that ChatGPT could be replicating a solution it has seen in its training data. Another observation is that the similarity for code solutions generated with the “C” demonstration prompts was high. This is interesting because the used prompt had a demonstration of the game Connect Four which is structured similarly to Tic-Tac-Toe, suggesting that demonstrations of relevant tasks can influence code generation. The solution with the index JDT\_B05 which is between JDT\_B01 and JDT\_C01 has the lowest number of similar solutions and stands out in the figure as the origin of the yellow cross. This is likely due to it being the only solution that does not use “for loops” to check the winner, instead hard coding the values using if statements.

## Java Tic-Tac-Toe proximity graph



**Figure 6:** Java Tic-Tac-Toe proximity graph.

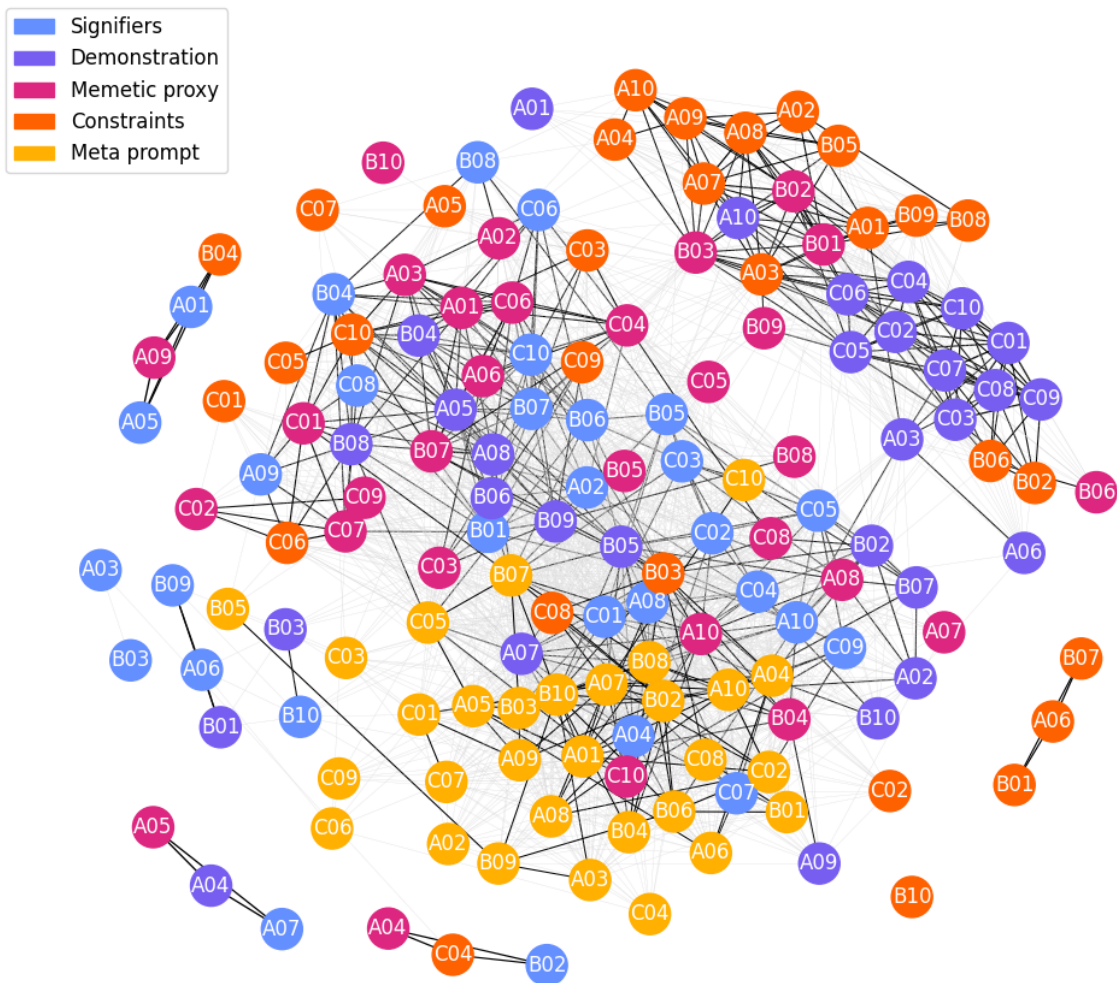
In Java Tic-Tac-Toe, as seen in Figure 6, there is a single large cluster of solutions that have a high level of source code similarity to each other, with some solutions further away with fewer strong matches. One of the outliers is the memetic proxy “B10” solution which has no strong or close matches and can be seen on the right side in the figure. The solution also achieved the lowest score of all the solutions in the experiment by failing all five correctness criteria.



**Figure 7:** Python Tic-Tac-Toe code similarity heat map.

Figure 7 shows how similar the solutions for Python Tic-Tac-Toe were to each other. The figure has many solutions with a low level of similarity, as seen by the yellow lines. It can be observed that the solutions with the highest similarity are those generated using the “C” prompt with a demonstration of the game Connect Four.

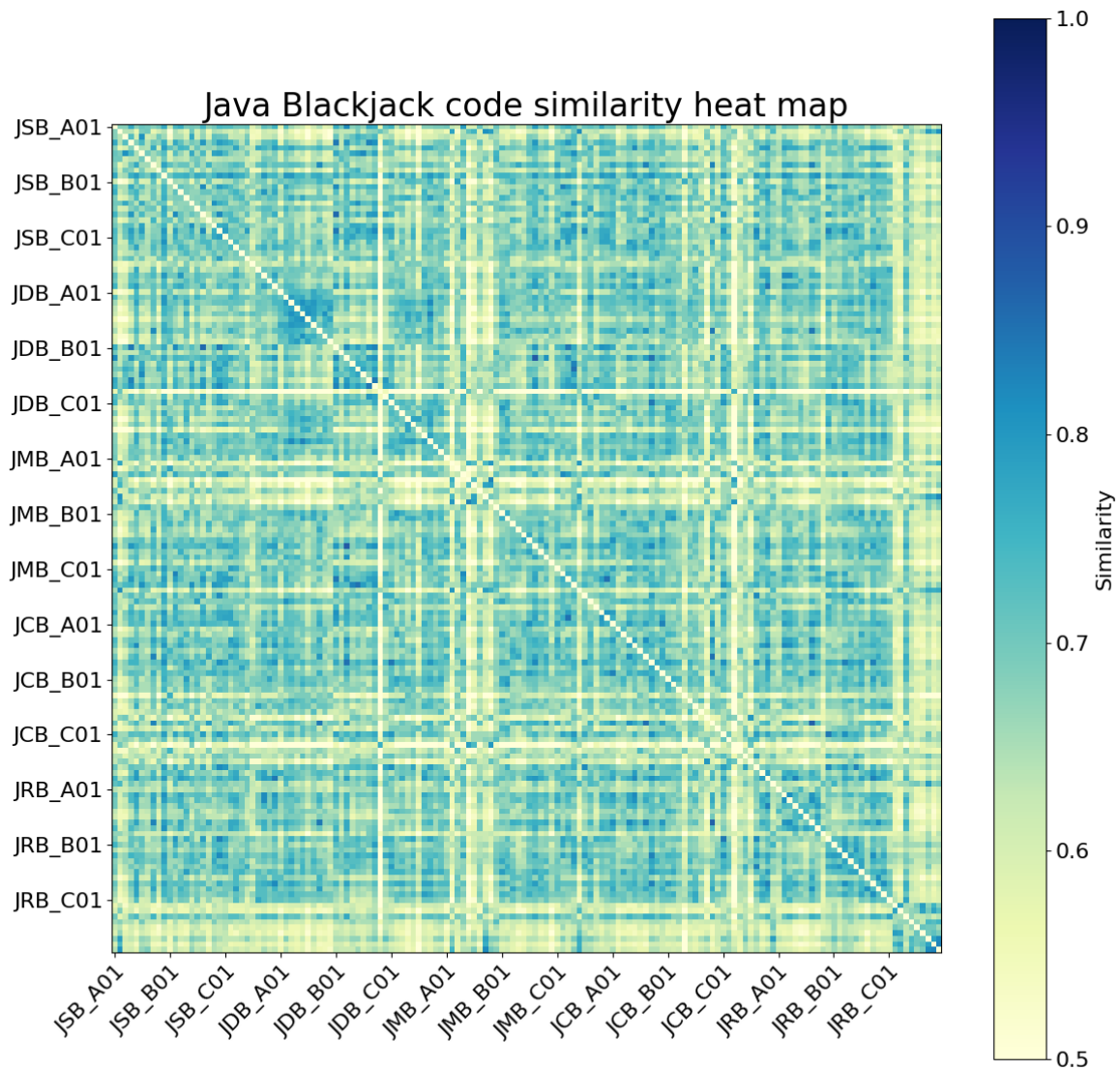
## Python Tic-Tac-Toe proximity graph



**Figure 8:** Python Tic-Tac-Toe proximity graph.

In Python Tic-Tac-Toe, as seen in Figure 8, there is a large cluster that connects the majority of the solutions. It can be observed that the upper right part of the large cluster only contains solutions generated using demonstration, memetic proxy, and constraints and only has a single strong connection to the rest of the cluster through demonstration A03. The lower and larger part is mixed with a higher number of signifier and meta prompt solutions. The same pattern can be observed in Figure 7 which shows that the ten solutions generated using the demonstration “C” prompt between PDT\_C01 and PMT\_A01 have lower similarity to solutions generated using signifiers and meta prompts. With some matches for solutions generated using the demonstration “A” prompt, memetic proxy “B” prompt, and constraints “A” and “B” prompt. There are also some independent clusters with 3 or 4 closely matching solutions. In the lower-left corner of Figure 8, the signifier “A07”, demonstration “A04”, and memetic proxy “A05” solutions have formed a cluster. The solutions in this cluster are almost identical with the same method names and code structure. These solutions correspond to the

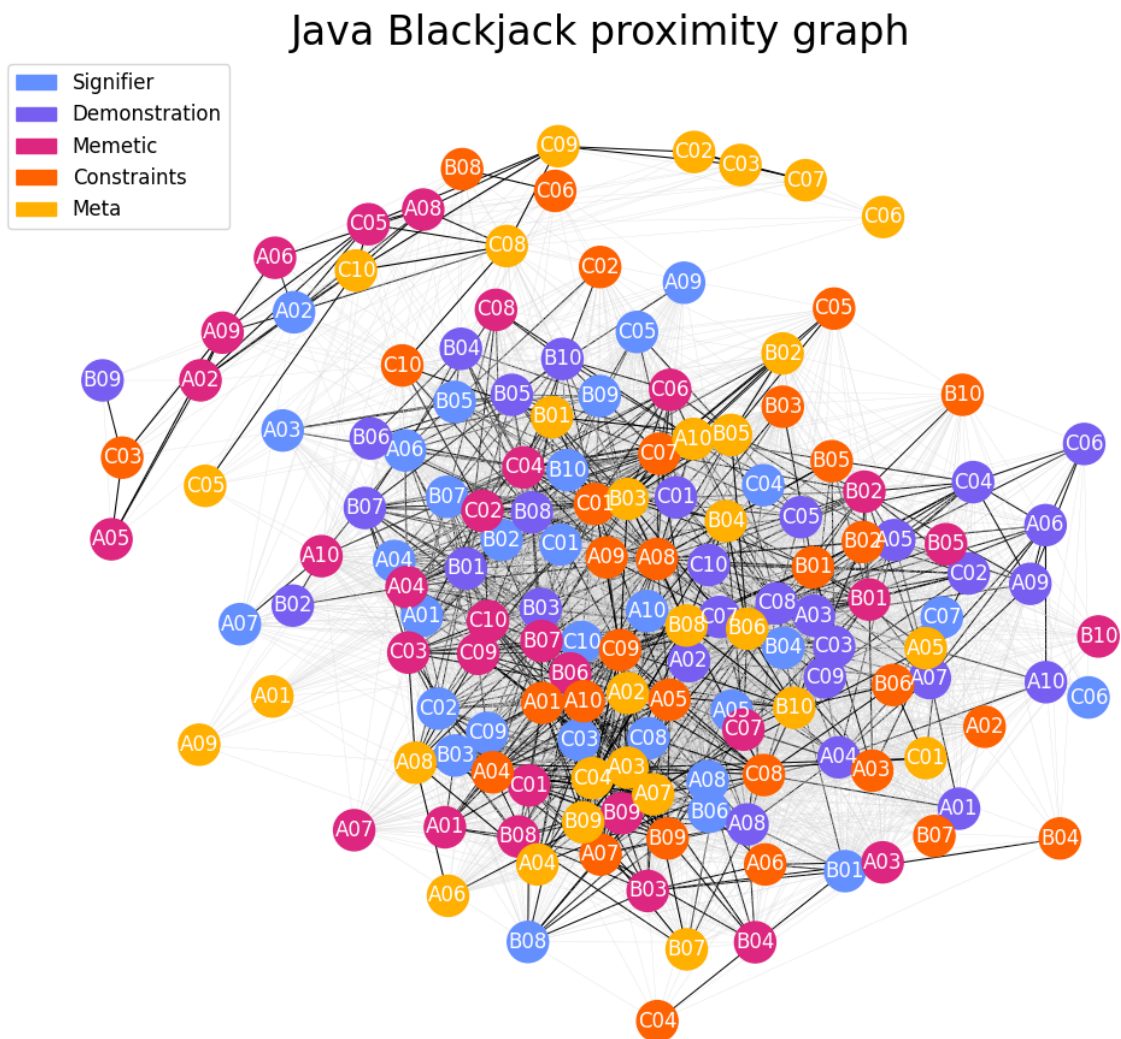
outliers in the LOC boxplot (see Appendix G.3) at 171, 171, and 183 for the respective task specification methods.



**Figure 9:** Java Blackjack code similarity heat map.

Figure 9 shows how similar the solutions for Java Blackjack were to each other. This figure shows many solutions had a high level of similarity. There are some solutions that deviate with a lower similarity score, as indicated by the yellow lines in the figure. This is especially noticeable for meta prompt, where the “C” prompt had a lower similarity compared to most of the solutions.

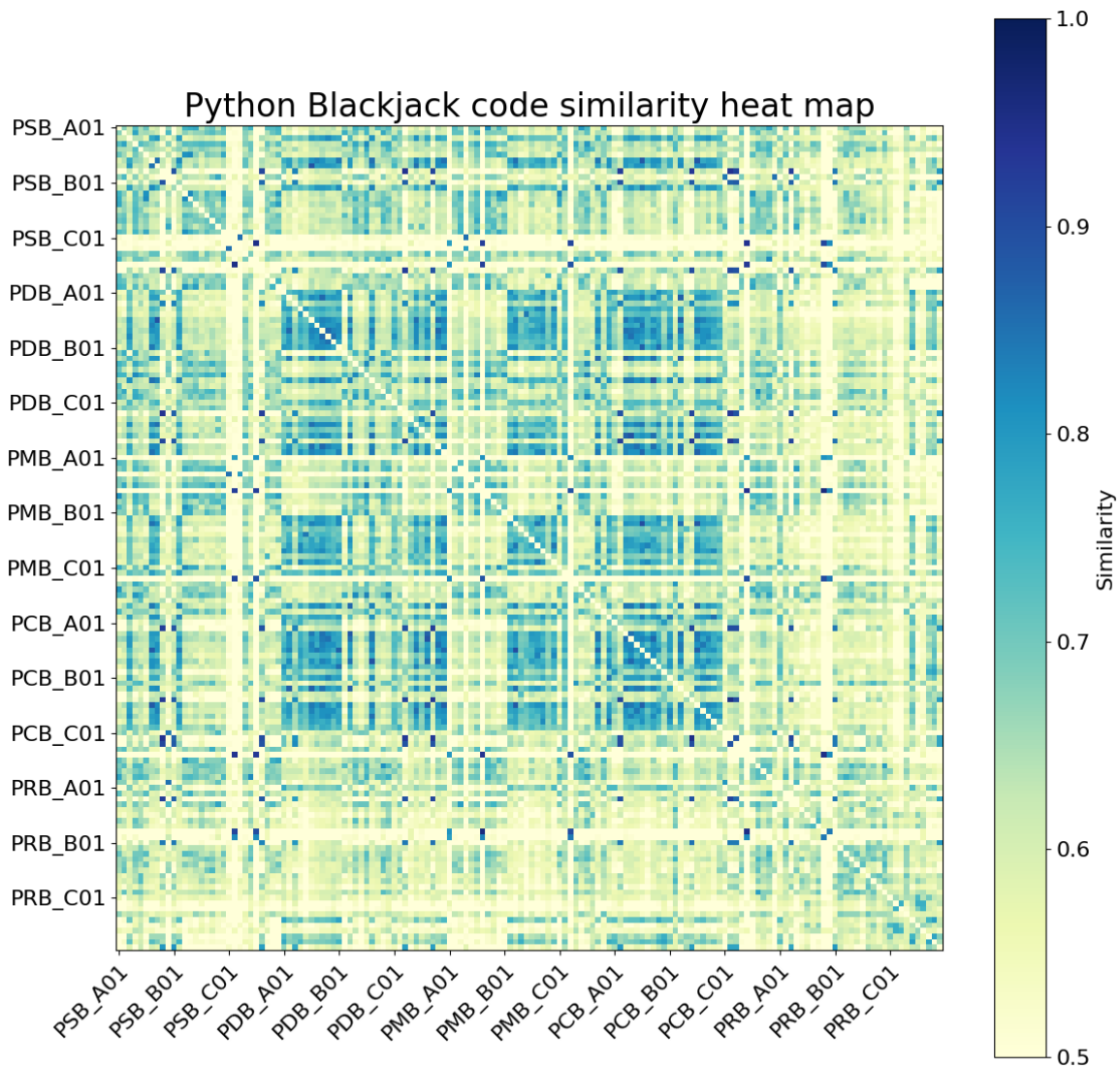




**Figure 10:** Java Blackjack proximity graph.

In Java Blackjack, as seen in Figure 10, there is a large cluster of solutions with a high level of similarity, and solutions with lower similarity are connected at the edges of the cluster. There is a cluster at the top of Figure 10, connected to the larger cluster through the constraints “C10” solution. This resembles the pattern seen in Python Tic-Tac-Toe (Figure 8), and likewise, Figure 9 shows the expected pattern of the majority having a high level of similarity, while there are some solutions that are distinct from the majority.

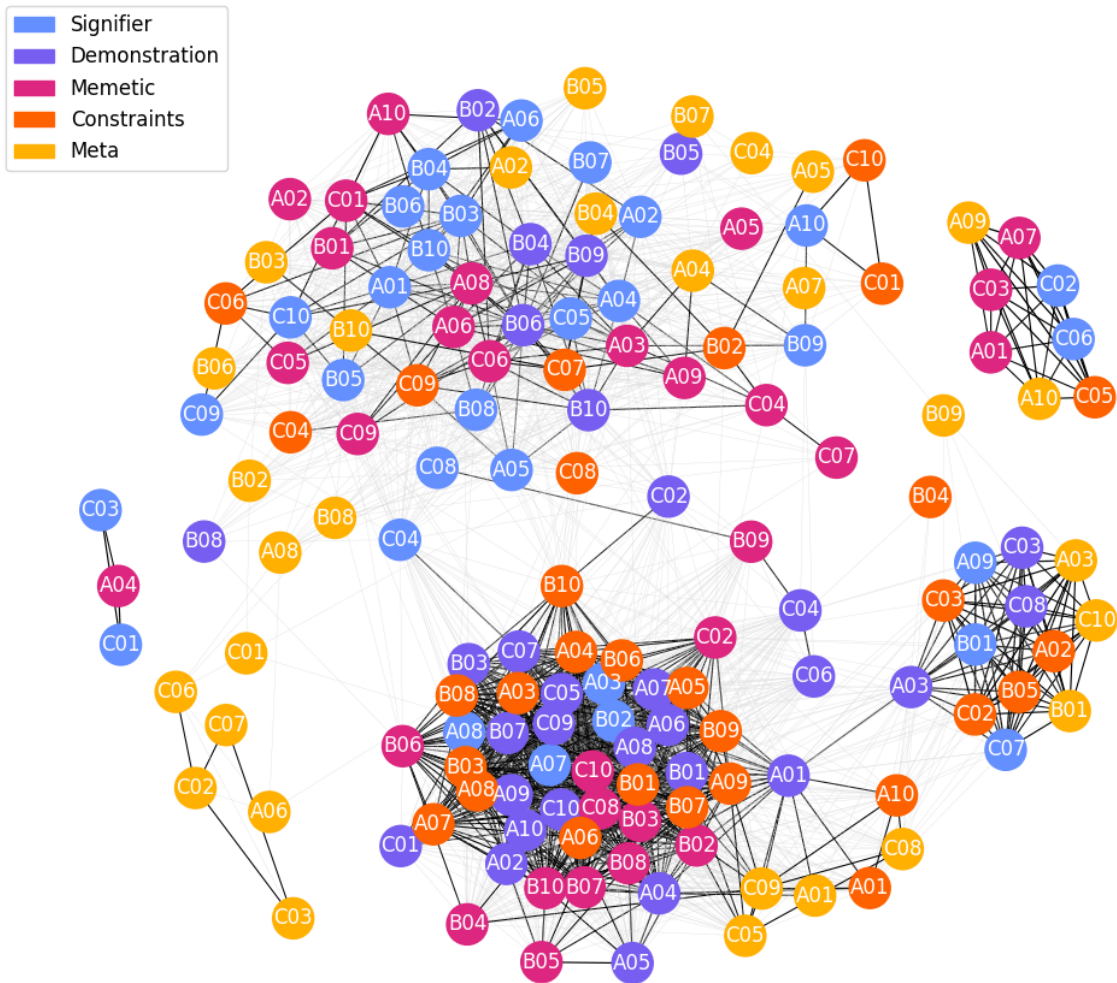




**Figure 11:** Python Blackjack code similarity heat map.

Figure 11 shows how similar the solutions for Python Blackjack were to each other. Solutions generated using the task specification methods demonstration, memetic proxy, and constraints sometimes had a high level of similarity, as indicated by the blue color seen between PDB\_A01 and PDB\_B01. When comparing the code files, a noticeable pattern was that they implemented the game by using the classes: card, deck, hand, and blackjack. The blackjack class implemented the game logic and had the most significant differences between solutions, while the other three classes were very similar in their functionality.

## Python Blackjack proximity graph



**Figure 12:** Python Blackjack proximity graph.

In Python Blackjack, as seen in Figure 12, there are multiple large clusters. The densest cluster is primarily made of demonstration, memetic, and constraints. This large cluster corresponds to the pattern seen in Figure 11, where solutions created using different task specification methods had many similarities. In the lower-left corner of Figure 12, there is a group of eight solutions that have a high level of similarity without any close matches outside their cluster. These solutions were of similar length and structure, with a common trait being that instead of using a deck, a random card function was used.

**Table 14:** Code similarity match types

	Strong match	Close match	Weak match
Java Tic-Tac-Toe	18.1%	40.0%	41.9%
Python Tic-Tac-Toe	4.6%	15.1%	80.3%
Java Blackjack	9.4%	34.7%	55.8%
Python Blackjack	9.1%	12.4%	78.5%

Table 14 shows the percentage of strong, close, and weak matches in the proximity graphs. Strong matches are 75% or above similarity. Close matches are 67.5% similarity or greater but less than 75%. Weak matches are less than 67.5% similarity. These match types correspond to the colors of the edges black, light gray, and white.

Java Tic-Tac-Toe (Figure 6) has the highest number of strong and close matches. Python Tic-Tac-Toe (Figure 8) has the lowest number of strong matches. Python Blackjack (Figure 12) has the fewest number of close matches but a similar number of strong matches as Java Blackjack (Figure 10).

### 6.5.2 Hypothesis F

It can be observed that there is a distinguishable pattern in the variety of the solutions generated for Java Blackjack, Python Tic-Tac-Toe, and Python Blackjack therefore the null hypothesis was rejected, and the alternative hypothesis was accepted:

*“There is a distinguishable pattern of syntactical variety for solutions generated using the task specification methods.”*

### 6.5.3 Conclusion

Based on the analysis, the conclusion is that the solutions generated using task specification methods are syntactically varied. The variety can be seen in Figure 8 Python Tic-Tac-Toe, Figure 12 Python Blackjack, and to a lesser extent, Figure 10 Java Blackjack. One common pattern is that there is a large group of solutions with a high level of similarity connected to a smaller distinguishable group of similar solutions through a single point. This pattern can be observed in the heat maps, where certain solutions have a low similarity compared to the majority of other solutions. Another pattern that can be seen in Python is independent clusters of solutions that are connected but lack strong connections to larger clusters. These independent clusters often have uncommon functionality, such as AI players in Tic-Tac-Toe or random cards instead of decks in Blackjack.

The solutions in Figure 6 Java Tic-Tac-Toe, do not appear to have any syntactic variety, with the majority of solutions being tightly clustered apart from some outliers that lack any strong connections. From the data gathered, the observation can be made that the solutions for Java have a higher level of similarity than those in Python, which could be for numerous reasons, such as the difference in training data, the difference in syntax, or technical limitations.

## 7 Discussion

This chapter discusses the results of the experiment, the generalizability of the results, and the contributions of the study's results. Additionally, threats to validity and ethical concerns are addressed. The chapter ends with a discussion of future work in the field and potential avenues for other research projects.

### 7.1 General Discussion

**RQ1:** Is there a difference in the correctness of code generated using the chosen task specification methods compared to direct task specification with signifiers?

The results show that there is a significant difference in the correctness of code for solutions generated using the chosen task specification methods compared to the baseline direct task specification with signifiers in the correctness criteria “User interaction”, “Unused code”, and “Input validation”.

The results comparing the baseline signifier to demonstration and signifiers to constraints show a difference in the code correctness criteria “Input validation”. Demonstration prompts included examples where input validation was present, which may have led to more of the ChatGPT-generated solutions also having input validation, compared to solutions using signifiers. Constraint prompts used rules to restrict the types of solutions ChatGPT generated, this worked to increase the number of correctness criteria passed.

The results of the baseline compared to memetic proxy show a significant difference in “Unused code”. This is because Blackjack solutions in Java abnormally generated the same unused method “getSuit()” through most of the memetic proxy prompts. This could indicate that the prompts directed ChatGPT towards a specific solution or combination of solutions seen during training that implemented the method but never used it.

A significant difference in code correctness for “User interaction” was seen in the results, comparing the baseline with meta prompts. A possible explanation for the difference is that the meta prompts included more explicit explanations of how the user should interact with the game, resulting in a higher number of correct solutions. Signifier prompts included only the game’s name, inducing ChatGPT to determine how users should interact with the game.

**RQ2:** Comparing the programming languages Java and Python, is there a difference in the correctness of code generated using the different task specification methods?

The comparison of programming languages shows that there is a significant difference between Java and Python in two correctness criteria. Python has higher correctness in “Unused code” but lower in “Input validation” compared to Java.

The difference in the code correctness criterion “Unused code” is likely because most Java solutions were object-oriented and had a tendency to generate “getter” methods that were not used, this behavior was not seen in Python, causing the difference. For the correctness criteria “Input validation” the difference could be because “Input validation” is more commonly included in Java solutions compared to Python. The exact reason why is unclear, but from analyzing the files, what can be observed is that Python, specifically in Tic-Tac-Toe,

does not check whether the input is within the expected range for the game board in as many solutions as Java.

**RQ3:** Do solutions generated using task specification methods result in syntactically varied solutions?

The analysis of the figures shows that the solutions generated using the task specification methods were syntactically varied for Java Blackjack, Python Tic-Tac-Toe, and Python Blackjack. But not for Java Tic-Tac-Toe.

The results are reasonable, based on what is known about ChatGPT and the programming tasks. Generative AI models will create responses similar to the data seen during training, this could be a contributing factor to why Python solutions were different compared to Java solutions. Another factor is the programming tasks used; Tic-Tac-Toe has fewer alternatives to how a feature can be implemented compared to Blackjack. The largest differences seen in Tic-Tac-Toe were solutions with additional features, such as a computer opponent or replayability. Most of the differences were quite small, such as how the board was displayed or how the input for rows and columns was structured. Blackjack had more alternatives, such as how the card deck was implemented or how the program was structured. There were also some solutions with additional functionality, such as betting or replayability.

## 7.2 Generalization

The prompt engineering techniques proposed by Reynolds and McDonell (2021) were designed for generating natural language text in GPT-3, whereas this thesis project applied these proposed methods to code generation with ChatGPT, a successor to GPT-3. The performance results of the used prompt engineering techniques from this report could not be compared with Reynolds and McDonell's (2021) paper, as they did not present any results regarding the performance of the proposed prompt engineering techniques.

Finnie-Ansley et al. (2022) conducted an experiment evaluating Codex's capabilities in solving introductory Python programming assignments. The authors scored the performance of Codex using CS1, and the results were 15.7/20 (78.5%) and 19.5/25 (78.0%). This study however showed that 50.0% of generated solutions in Python had full functional code correctness (see Figure 4). This difference in results is due to different evaluation methods being used.

Finnie-Ansley et al. (2022) evaluated Codex code generation capabilities using a scoring system with different weights and penalties on each problem, while this study used SRS to evaluate the generated code. Because the evaluation methods differ, the results cannot be fairly compared. Additionally, the experiment conducted by Finnie-Ansley et al. (2022) only looked at Python solutions, while this thesis generated both Java and Python solutions, meaning only half of the experiment data from this thesis can be compared to Finnie-Ansley et al. (2022). Another contrast between the compared results is the differentiating programming tasks being investigated, Finnie-Ansley et al. (2022) used CS1 programming tasks, while this thesis looked at the programming tasks Tic-Tac-Toe and Blackjack. These games have a higher complexity compared to CS1 tasks, which makes it more difficult to generate functioning solutions.

Denny et al. (2022) used Copilot to solve 166 Python programming problems from the CodeCheck dataset, where 79 (47.6%) problems were solved on the first attempt. The 87 unsolved problems were modified using prompt engineering and then prompted to Copilot, 53 (60.9%) of the unsolved problems were successfully solved when modified. Comparatively, the results from this study had 50.0% Python solutions that passed all the functional correctness criteria (see Figure 4).

Denny et al. (2022) used an autograder for the CodeCheck dataset to evaluate the generated code, and this study validated the generated code using a human evaluation with SRS. The evaluation methods differ, as automatic evaluation provides more reliable results compared to human evaluation, as the human element is removed from the process. This study used two complex programming tasks, Tic-Tac-Toe and Blackjack. These programming tasks are considerably less than 166 programming exercises. While Denny et al. (2022) used the vague term “prompt engineering” to generate solutions, this study investigated the usage of established prompt engineering techniques provided by Reynolds and McDonnell (2021) to generate code solutions.

Chen et al. (2021) generated code for 164 programming problems from the HumanEval Python dataset with Codex, which resulted in a 28.8% success rate, but when generating each programming problem 100 times, the success rate goes up to 70.2%. This study's results showed that 50% of the generated solutions in Python passed all functional correctness criteria (see Figure 4). However, Chen et al. (2021) evaluated their solutions using pass@k, which implies that a problem is regarded as solved if any solution in k number of generated solutions passes all unit tests. This is different from this study's data evaluation, where the entire sample's pass rate is averaged. Chen et al. (2021) used 164 simple Python problems, while this study used two complex programming tasks for both Java and Python. Chen et al. (2021) prompted each programming problem 100 times using the same prompt, this study however used five task specification methods in two programming languages with three unique prompt templates for each programming task. Each prompt was prompted against ChatGPT ten times, meaning 300 prompts for each programming task. The first percentage of 28.8 is most comparable to this study's result of 50% as it represents the percentage of solutions having full code correctness. ChatGPT greatly outperforms Codex according to this comparison.

The source code comparison data presented by Chen et al. (2021) is limited to four randomly selected solutions, where the BLEU score represents the similarity between the generated code and the human-written solutions. The authors concluded that higher BLEU scores do not indicate higher functional correctness. This study does not have a ground truth, which means BLEU cannot be used, instead, code similarity measurement is used to identify patterns and the effects prompts had on code generation.

White et al. (2023) provide a catalog of prompt engineering techniques, or “patterns” together with examples of these patterns in the form of prompt templates. The patterns are designed to work with all LLMs, but the example prompt templates provided in the paper were created and tested with ChatGPT, the LLM this study investigates. The patterns created by White et al. (2023) focus on multiple different domains in computer science, one of which is code generation, which is used in this study. Some patterns are similar to the task specification methods proposed by Reynolds and McDonnell (2021). The persona pattern proposed by White et al. (2023) is similar to the method of task specification by memetic

proxy presented by Reynolds and McDonell (2021). White et al. (2023) presented no code generation performance results for the patterns and therefore cannot be compared to this study.

Mukherjee et al. (2021) presented a new method of training generative models of source code; this method uses a static-analysis tool that is a weak supervisor. The authors then evaluated the code generation capabilities of the new training method with their developed neural model of source code using “Neurosymbolic Attribute Grammar” (NSG), compared to this study where ChatGPT was evaluated. The studies evaluated different neural models that used different training methods. OpenAI used Reinforcement Learning from Human Feedback (RLHF) (OpenAI, 2022) to train the model on which ChatGPT is based. Mukherjee et al. (2021) evaluated their model by using a variation of task specification by demonstration, as they demonstrated complete Java solutions with removed method bodies, which should be regenerated using the surrounding methods, method name, and parameters as context. This study uses demonstration by providing program code of another similar programming task and a signifier in order to convey the intended task. Comparing the results, NSG passed 86.41% of all static analysis checks in the experiment by Mukherjee et al. (2021). Comparably, ChatGPT in this study passed 86.00% of all the syntactic correctness criteria (see Figure 4). NSG passed 93.84% of the static analysis check “No unused variables”, and comparatively, ChatGPT passed 91.67% of the correctness criteria “Unused code” in Java (see Figure 3). Though the models had similar results, NSG had a slightly higher pass rate. The reason for NSG having a higher pass rate could be because ChatGPT is a general-purpose LLM capable of performing multiple tasks, while NSG is specifically trained for the task that it was tested on.

### 7.3 Contributions

The results from the research show that the choice of task specification method does not guarantee better responses in code generation with ChatGPT. Additionally, the results also show that for certain programming tasks and programming languages, there is a syntactical variety in the solutions generated using ChatGPT. These results can be used when designing prompts and may be relevant for software developers and AI researchers. It may also be relevant for educators in programming courses to understand the capabilities of ChatGPT and the types of programs that are solvable.

The evaluation of the task specification methods does not consider the effort required to utilize them. Direct task specification with signifiers requires the least amount of effort from the user compared to the other task specification methods, the user only needs to state what programming task should be generated.

Task specification by demonstration requires the user to have a demonstration in order to use this technique. This method could be useful for users who already have code that can be used as a demonstration. But for users without a relevant demonstration, this method may be unsuitable since it requires the creation of code that can be used as a demonstration. The benefit of this method compared to the other methods is that code can be used to influence the code generation of new software to a certain extent. However, the drawback is that without a relevant demonstration suited for the desired task, the other methods are most likely more appropriate to use. Demonstration statistically outperforms signifiers in “Input validation” according to the hypothesis test (see Table 6).

Task specification by memetic proxy requires the user to think of a character that could solve the task well and is therefore not that different in the amount of effort compared to direct task specification with signifiers. Even though this method may be easy to use, it does not necessarily mean that it should be used over signifiers. The results from the experiments show that memetic proxy was statistically worse in “Unused code” compared to signifiers (see Table 6). Memetic proxy was the only method with statistically worse performance in a correctness criterion when compared to signifiers and should therefore be avoided when prompting ChatGPT for code generation.

Prompts using task specification with constraints can be used to guide ChatGPT in certain directions. It can be used to constrain what should be included in the code and what should be absent. An example of an inclusive constraint could be that a specific library or design should be used. This gives the user more control over how a program is created, with a low barrier to entry due to the simplicity of the method. Constraints were significantly better than signifiers in the criterion “Input validation” (see Table 6). With refined prompts constraints, could likely be a contender for the best task specification method, though further research would be required to verify.

Meta prompt is in league with direct task specification with signifiers in how easy it is to use. The user simply has to ask the LLM to write instructions for completing a desired task and then use the instructions to generate a solution. Meta prompts perform statistically better than signifiers in the correctness criteria “User interaction” (see Table 6). The instructions generated when using meta prompts vary in both quality and complexity, which makes the comparison between this method and signifiers less reliable. In the experiment, it was observed that high-quality instructions outperformed signifiers consistently, whereas low-quality instructions underperformed.

Demonstration and constraint are the most suitable methods to use in terms of correctness, as they have the highest score in most of the criteria (see Figure 2). When considering which task specification method to use, demonstration becomes less suitable if the prompter has no relevant code to use as an example, in which case task specification with constraints is recommended. If the prompter does have a relevant example, the ease of use and the performance of task specification with demonstration makes it a suitable alternative. Since constraints require the user to write a task-specific prompt, it is more time-consuming than reusing code.

## **7.4 Ethics & Threats to Validity**

### **7.4.1 Threats to Validity**

To determine the validity of the results gathered from the experiment, Threats of conclusion validity, internal validity, external validity, and construct validity are analyzed.

“Fishing” is a conclusion validity threat where the selection of treatments may be influenced by researcher bias towards a certain outcome (Wohlin et al., 2012). This is a concern for this study since an initial investigation was conducted to determine the feasibility of the experiment. Task specification methods had not been investigated in code generation at the time of the pre-study, which meant the researchers had to interpret and decide how to use the methods. This could have resulted in the decision to use the methods in a way that favored a certain conclusion. It could also have resulted in the methods being applied in a



way that is not accurate to how they are used in real-life settings which could result in an external validity threat.

“Low statistical power” is another conclusion validity threat that concerns whether the results are statistically significant enough to answer the research question (Wohlin et al., 2012). This is a large concern in this study since the data collection for the experiment is manually performed which is more time-consuming than automatically performing the experiment, and because of the limited time allocated for the experiment, it is not possible to gather enough data to provide considerable statistical power. To reduce this threat the scope of the research is limited to not overstate the results.

The conclusion validity threat “random irrelevancies in experimental setting” covers elements outside the experimental setting that could disturb the results (Wohlin et al., 2012). This manifested itself in the form of inconsistent behavior from ChatGPT such as the service crashing due to server issues, and continuation prompts having unintended behavior. Certain types of failures can be attributed to the continuation prompts, such as unfinished code or incorrectly nested statements. These failures which only occur at the breakpoint between the first response and the continuation are fixed by the researchers either through regenerating the response or manually fixing the incorrect code. This problem is an element outside the experimental setting that may disturb the result, which is why the researchers resolved the issue.

“Instrumentation” is an internal validity threat concerning the artifacts used in the execution of the experiment that could have potentially negative effects on the experiment (Wohlin et al., 2012). The prompts are created based on the independent variables and were sent to ChatGPT to generate solutions (see Appendix A). The solutions are then evaluated using the SRS to produce the data (see Appendix B). The data collection process relies on three artifacts: the prompts, ChatGPT, and SRS. To address the validity threats prompts are designed based on existing techniques in prompt engineering (Reynolds and McDonell, 2021). ChatGPT is a black box that could affect the independent variables without the researcher's knowledge and is therefore considered a threat to internal validity. And lastly, the artifact SRS is based on research in code evaluation to address potentially negative effects. This research is functional and syntactical evaluation (Chen et al., 2021; Mukherjee et al., 2021).

The “interaction of setting and treatment” is an external validity threat, where the experiment is unrepresentative of industrial practices concerning setting and treatment (Wohlin et al., 2012). In this study, the prompts used in the experiment may be unrepresentative of the practical usage. To circumvent this issue, the prompts used in the experiment were designed using research in prompt engineering applied to LLMs by Reynolds and McDonell (2021). Additionally, the different treatments are varied by using two of the most prevalent programming languages, Java which is statically typed, and Python which is dynamically typed. And two programming tasks, Tic-Tac-Toe and Blackjack, vary in complexity and therefore cover a larger area of the domain. As prompt templates were used to construct the prompts for different treatments, the benefit of replication was included. This made it possible for other researchers to verify the validity of the experiment.

“Mono-operation bias” is a validity threat regarding the underrepresentation of the construct through too few independent variables (Wohlin et al., 2012). To address this threat multiple

programming tasks were used, and multiple prompts in the style of the specific task specification method were tested.

“Mono-method bias” is a construct validity threat, where only a single measuring method was used, potentially causing bias in the results (Wohlin et al., 2012). The study used two methods to investigate the impact of task specification on code generated via ChatGPT, the first method investigated code correctness, and the second method investigated code similarity. By using two methods the construct validity threat was reduced.

## 7.4.2 Ethics

Since ChatGPT is trained on a large dataset containing a multitude of text-based sources, the responses from ChatGPT will be based on those sources. The sources are not referred to nor credited in the response, which raises a concern about copyright infringement. OpenAI, the company behind ChatGPT, acknowledges that the datasets used for training include copyrighted works (Zirpoli, 2023). In the current Terms of Use for OpenAI, it is stated that all rights from ChatGPT’s outputs are assigned to the user of the application (OpenAI, 2023c). Thus, there is a possibility that users of ChatGPT may be liable for copyright infringements on their AI-generated works. Additionally, according to Zirpoli (2023), it is unclear whether training ChatGPT using datasets that include copyrighted works infringes upon copyright. The Copyright Act does not clarify if generative AI programs such as ChatGPT constitute fair use (Zirpoli, 2023). This is a potential threat to the future of LLMs.

The development of LLMs warrants a change to computer science education, as the results presented by Finnie-Ansley et al (2022) show that even ChatGPT’s predecessor Codex outperforms students on introductory computer science exams. Floridi and Chiriatti (2020) believe that people who have jobs that include writing, which encompasses programming, will have to learn how to create prompts and then assemble the results from the prompts rather than copying and pasting text. If the work methods change then education will have to change as well to reciprocate the changing demand of workers. This is considered to be a social concern as the development of LLM is probable to have a large impact on all of education, similar to the effect that the internet had on education. Changing education will affect many aspects of society as education is the backbone of society.

An ethical concern related to LLMs is the validity of the information provided in generated responses. There are concerns about information possibly being opinionated, meaning answers may have a political bias. This opinionated information could be caused by the training data or how the LLM is designed (Bender et al., 2021). The other information validity concern is LLMs tendency of hallucinating, meaning the LLM uses false information in its response. This is either caused by a lack of information in the training dataset or caused by flaws in how the LLM was developed and operated (Ji et al, 2023).

There is an ecological concern regarding the resources required to develop and train advanced language models. According to Bender et al. (2021) over six years the amount of computing power needed to train LLMs has increased by 300,000x. This is a far greater rate of needed computing power than the rate at which microchips energy efficiency can improve (Alioto et al., 2018). This extreme rate of increased computing power of LLMs is mainly due to the development focus being on improving accuracy while neglecting the development of more energy-efficient models (Bender et al., 2021). LLM being energy inefficient is a

problem because energy production is known to cause many environmental problems such as pollution, ecosystem degradation, acid rain, etc. (Dincer, 1999).

Energy consumption related to the usage of the services provided by LLMs should also be considered an ecological concern. There is also an additional economical consideration with LLMs energy consumption as OpenAI is providing their LLM service, ChatGPT for free, meaning the company has a substantially negative cash flow<sup>4</sup>. This is especially the case as ChatGPT has seen an influx of users trying out the model after its public release.

## 7.5 Future Work

The task specification method of serialization was not included in the study, as it was deemed too time-consuming to be included in the experiment. Serialization is when complex problems are broken down into simpler sub-problems that are solved separately (Reynolds and McDonell, 2021). One of the methods considered for performing a study on serialization is to include human subjects. Multiple approaches could be used, such as having participants generate code using the task specification methods and asking them to score it based on specific criteria in a survey format similar to what was mentioned in the alternative methods. The second type could be observing the subject's approach to solving a problem without receiving prompt engineering guidance, this could be performed as a case study. Both of these alternative approaches would provide valuable insight into how humans use prompts. The survey approach also gives researchers the opportunity to investigate specific criteria based on the aim of the research and further the field of prompt engineering.

Future research could expand the experiment by investigating more programming tasks, and programming languages; and having a larger sample size by generating more solutions with the same prompts. In this study every prompt is used to generate ten solutions, it would be desirable for every prompt to be used a hundred times to improve the generalizability of the results. Because the data collection and evaluation were performed manually, there were fewer gathered results, compared to having the data automatically gathered via API and evaluation performed using unit tests. Therefore, future research could be done utilizing an API, with an existing dataset that is suited for this type of evaluation such as HumanEval, which contains 164 programming problems, human-written solutions, unit tests, and prompt strings. The benefit of using this dataset is that the results would be more comparable to previous works such as Chen et al. (2021). The dataset could also be used to test whether different prompt engineering techniques could be used to modify the prompt strings and see if they have an impact on code generation.

Another way to expand the experiment would be to include more variations to the prompt strings to get a more generalized conclusion as to whether ChatGPT generates different solutions depending on the wording of the prompt. The study could also be expanded by combining the task specification methods, using for example two methods at the same time, and then comparing the different combinations with each other.

This study used ChatGPT-3.5, while future research could conduct a similar experiment on other LLMs, such as the GPT-4 version of ChatGPT. The results from these experiments

---

4

<https://www.theinformation.com/articles/openais-losses-doubled-to-540-million-as-it-developed-chatgpt>

could be compared and used to measure the differences between LLMs code generation capabilities. This would be similar to the comparison made with the related works, Finnie-Ansley et al. (2022), Denny et al. (2022), Chen et al. (2021), and Mukherjee et al. (2021). The task specification methods that were used in the experiment had some influence over the code generation, but other models could perhaps be more receptive and show different results.

Many methods can be used to measure code similarity, such as BLEU and CodeBLEU, which use ground truth. These scores are commonly used when evaluating LLMs and could be used in future works that use a suitable dataset such as HumanEval similar to Chen et al. (2021). For something more in line with this study's use of code similarity, it is possible to compare bytecode rather than the syntax since comments and variable names affect the score of the current approach. The research value lies in advancing the understanding and knowledge of how effective prompts are and how much non-determinism affects the results. It also gives insight into how the model could potentially interpret and construct a solution to a problem based on its training data.

As the research field of code generation develops a systematic literature review could be conducted as mentioned in the alternative methods. The research question could focus on what types of prompt engineering techniques are used in code generation research and what results they produce. A study on this topic would be beneficial for other researchers in the field of code generation and potentially also for software developers that use LLMs.

## References

Alioto, M., De, V. and Marongiu, A., 2018. *Energy-quality scalable integrated circuits and systems: Continuing energy scaling in the twilight of Moore's law*. IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 8(4), pp.653-678.

Austin, P.C., 2009. *Balance diagnostics for comparing the distribution of baseline covariates between treatment groups in propensity-score matched samples*. Statistics in medicine, 28(25), pp.3083-3107.

Bavarian, M., Jiang, A., Jun, H., Pondé, H., 2022, *New GPT-3 capabilities: Edit & insert*. OpenAI. <https://openai.com/blog/gpt-3-edit-insert> (Accessed 2023-04-17)

Bender, E.M., Gebru, T., McMillan-Major, A. and Shmitchell, S., 2021. *On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?* 🦜. Proceedings of the 2021 ACM conference on fairness, accountability, and transparency, pp.610-623.

Berndtsson, M., Hansson, J., Olsson, B. and Lundell, B., 2007. *Thesis projects: a guide for students in computer science and information systems*. Springer Science & Business Media.

Beurer-Kellner, L., Fischer, M. and Vechev, M., 2022. *Prompting Is Programming: A Query Language For Large Language Models*. arXiv preprint arXiv:2212.06094.

Bicycle Cards. 2023. *LEARN TO PLAY Blackjack*. <https://bicyclecards.com/how-to-play/blackjack/> (Accessed 2023-04-20)

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A. and Agarwal, S., 2020. *Language models are few-shot learners*. Advances in neural information processing systems, 33, pp.1877-1901.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G. and Ray, A., 2021. *Evaluating large language models trained on code*. arXiv preprint arXiv:2107.03374.

Chowdhary, K.R., 2020. *Natural language processing*. Fundamentals of artificial intelligence, pp.603-649.

Cohen, A., 2011. *Fuzzywuzzy: Fuzzy string matching in python*. SeatGeek <http://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/> (Accessed 2023-04-26)

Costa, L.D.F., 2021. *Further generalizations of the Jaccard index*. arXiv preprint arXiv:2110.09619.

Davis, A., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledebner, G., Reynolds, P., Sitaram, P. and Ta, A., 1993. *Identifying and measuring quality in a software requirements specification*. In [1993] Proceedings First International Software Metrics Symposium, pp.141-152. IEEE.

Dehaerne, E., Dey, B., Halder, S., De Gendt, S. and Meert, W., 2022. *Code Generation Using Machine Learning: A Systematic Review*. IEEE Access.

Denny, P., Kumar, V. and Giacaman, N., 2022. *Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language*. arXiv preprint arXiv:2210.15157.

Dincer, I., 1999. *Environmental impacts of energy*. Energy policy, 27(14), pp.845-854.

Finnie-Ansley, J., Denny, P., Becker, B.A., Luxton-Reilly, A. and Prather, J., 2022. *The robots are coming: Exploring the implications of openai codex on introductory programming*. Australasian Computing Education Conference, pp.10-19.

GitHub, 2022. *GitHub Copilot · Your AI pair programmer*. GitHub.  
<https://github.com/features/copilot> (Accessed 2023-03-30)

Harrison, Chase., 2022. *LangChain, Prompt Templates*.  
<https://langchain.readthedocs.io/en/latest/modules/prompts.html> (Accessed 2023-03-01)

Hiemstra, D. (2009). *Language Models*. Liu, L. and Özsu, M.T. (eds) Encyclopedia of Database Systems. Springer, Boston, MA.

Hughes, A., 2023. *ChatGPT: Everything you need to know about OpenAI's GPT-3 tool*. BBC Science Focus. <https://www.sciencefocus.com/future-technology/gpt-3/> (Accessed 2023-01-27)

IBM, 2023a. *What is Artificial Intelligence (AI)?*. IBM.  
<https://www.ibm.com/topics/artificial-intelligence> (Accessed 2023-03-03)

IBM, 2023b. *What is Machine Learning?*. IBM.  
<https://www.ibm.com/topics/machine-learning> (Accessed 2023-03-03)

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y.J., Madotto, A. and Fung, P., 2023. *Survey of hallucination in natural language generation*. ACM Computing Surveys, 55(12), pp.1-38.

Karpathy, A., Abbeel, P., Brockman, G., Chen, P., Cheung, V., Duan, Y., Goodfellow, I., Kingma, D., Ho, J., Houthoof, R., Salimans, T., Schulman, J., Sutskever, I., Zaremba, W., 2016. *Generative Models*. OpenAI. <https://openai.com/research/generative-models> (Accessed 2023-04-17)

Khan, H.A., 2003. *A visual basic software for computing Fisher's exact probability*. Journal of Statistical Software, 8, pp.1-7.

Khan, T., 2023. *What are tokens and how to count them?*. OpenAI.  
<https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them> (Accessed 2023-03-02)

Kim, H.Y., 2017. *Statistical notes for clinical researchers: Chi-squared test and Fisher's exact test*. Restorative dentistry & endodontics, 42(2), pp.152-155.

Kojima, T., Gu, S.S., Reid, M., Matsuo, Y. and Iwasawa, Y., 2022. *Large language models are zero-shot reasoners*. arXiv preprint arXiv:2205.11916.

le Cessie, S., Goeman, J.J. and Dekkers, O.M., 2020. *Who is afraid of non-normal data? Choosing between parametric and non-parametric tests*. European journal of endocrinology, 182(2), pp.E1-E3.

Lee, A., 2023. *What Are Large Language Models Used For?*. Nvidia.  
<https://blogs.nvidia.com/blog/2023/01/26/what-are-large-language-models-used-for/>  
(Accessed 2023-02-17)

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H. and Neubig, G., 2023. *Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing*. ACM Computing Surveys, 55(9), pp.1-35.

Floridi, L. and Chiriatti, M., 2020. *GPT-3: Its Nature, Scope, Limits, and Consequences*. Minds and Machines, 30(4), pp.681-694.

Merritt, R., 2022. *What Is a Transformer Model?*. Nvidia.  
<https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/>  
(Accessed 2023-02-17)

Mukherjee, R., Wen, Y., Chaudhari, D., Reps, T., Chaudhuri, S. and Jermaine, C., 2021. *Neural program generation modulo static analysis*. Advances in Neural Information Processing Systems, 34, pp.18984-18996.

OpenAI, 2022. *ChatGPT: Optimizing Language Models for Dialogue*. OpenAI.  
<https://openai.com/blog/chatgpt/> (Accessed 2023-01-27)

OpenAI, 2023a. *GPT-4 Technical Report*. arXiv preprint arXiv:2303.08774.

OpenAI, 2023b. *OpenAI API Documentation*. OpenAI.  
<https://platform.openai.com/docs/introduction> (Accessed 2023-03-03)

OpenAI, 2023c. *Terms of use*. OpenAI. <https://openai.com/policies/terms-of-use> (Accessed 2023-03-24)

Perrigo, B., 2023. *Exclusive: OpenAI Used Kenyan Workers on Less Than \$2 Per Hour to Make ChatGPT Less Toxic*. TIME.  
<https://time.com/6247678/openai-chatgpt-kenya-workers/> (Accessed 2023-04-12)

Radford, A., Narasimhan, K., Salimans, T. and Sutskever, I., 2018. *Improving language understanding by generative pre-training*. OpenAI.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. and Sutskever, I., 2019. *Language models are unsupervised multitask learners*. OpenAI blog, 1(8), p.9.

Ragkhitwetsagul, C., Krinke, J. and Clark, D., 2018. *A comparison of code similarity analysers*. Empirical Software Engineering, 23, pp.2464-2519.

Reynolds, L. and McDonell, K., 2021. *Prompt programming for large language models: Beyond the few-shot paradigm*. Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems, pp.1-7.

Salian, I., 2018. *SuperVize Me: What's the Difference Between Supervised, Unsupervised, Semi-Supervised and Reinforcement Learning?*. Nvidia.

<https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/>

(Accessed 2023-02-17)

Susnjak, T., 2022. *ChatGPT: The End of Online Exam Integrity?*. arXiv preprint arXiv:2212.09292.

The Exploratorium. 2000. *The Brain Explorer: Tic-Tac-Toe*.

[https://www.exploratorium.edu/brain\\_explorer/tictactoe.html](https://www.exploratorium.edu/brain_explorer/tictactoe.html) (Accessed 2023-03-23)

University of Skövde, 2020. *Object Oriented Programming G1F (IT401G)*. University of Skövde.

<https://www.his.se/en/education/courses/informatics/object-oriented-programming-g1f-it401g/> (Accessed 2023-04-28)

Vailshery, L.S., 2022. *Most used programming languages among developers worldwide as of 2022*. Statista.

<https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/> (Accessed 2023-04-17)

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. *Attention is all you need*. Advances in neural information processing systems, 30.

Vasylykiv, B., 2023. *Limitations And Ethical Considerations Of Using ChatGPT*. Incora.

<https://incora.software/insights/chatgpt-limitations> (Accessed 2023-02-17)

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J. and Schmidt, D.C., 2023. *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*. arXiv preprint arXiv:2302.11382.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. *Experimentation in Software Engineering*. Springer Science & Business Media.

Zirpoli, C.T., 2023. *Generative Artificial Intelligence and Copyright Law*. Congressional Research Service.



# Appendix A - Prompts

Table of prompts used to conduct the experiment.

Treatment	Prompt
<b>Signifier</b> Java Tic-Tac-Toe	Create a Java application for the game Tic-Tac-Toe
	Write a program in Java for the game Tic-Tac-Toe
	Using Java implement the game Tic-Tac-Toe
<b>Signifier</b> Java Blackjack	Create a Java application for the game Blackjack
	Write a program in Java for the game Blackjack
	Using Java implement the game Blackjack
<b>Signifier</b> Python Tic-Tac-Toe	Create a Python application for the game Tic-Tac-Toe
	Write a program in Python for the game Tic-Tac-Toe
	Using Python implement the game Tic-Tac-Toe
<b>Signifier</b> Python Blackjack	Create a Python application for the game Blackjack
	Write a program in Python for the game Blackjack
	Using Python implement the game Blackjack
<b>Demonstration</b> Java Tic-Tac-Toe	This is a demonstration of the game Luckycard: [Appendix C - Luckycard Java] Write the code for Tic-Tac-Toe:
	This is a demonstration of the game Hangman: [Appendix D - Hangman Java] Write the code for Tic-Tac-Toe:
	This is a demonstration of the game ConnectFour: [Appendix E - ConnectFour Java] Write the code for Tic-Tac-Toe:
<b>Demonstration</b> Java Blackjack	This is a demonstration of the game Luckycard: [Appendix C - Luckycard Java] Write the code for Blackjack:
	This is a demonstration of the game Hangman: [Appendix D - Hangman Java] Write the code for Blackjack:
	This is a demonstration of the game ConnectFour: [Appendix E - ConnectFour Java] Write the code for Blackjack:

<b>Demonstration</b> Python Tic-Tac-Toe	This is a demonstration of the game Luckycard: [Appendix C -Luckycard Python] Write the code for Tic-Tac-Toe:
	This is a demonstration of the game Hangman: [Appendix D - Hangman Python] Write the code for Tic-Tac-Toe:
	This is a demonstration of the game ConnectFour: [Appendix E - ConnectFour Python] Write the code for Tic-Tac-Toe:
<b>Demonstration</b> Python Blackjack	This is a demonstration of the game Luckycard: [Appendix C -Luckycard Python] Write the code for Blackjack:
	This is a demonstration of the game Hangman: [Appendix D - Hangman Python] Write the code for Blackjack:
	This is a demonstration of the game ConnectFour: [Appendix E - ConnectFour Python] Write the code for Blackjack:
<b>Memetic proxy</b> Java Tic-Tac-Toe	Create a Java application for the game Tic-Tac-Toe as if you were a computer science student in their first year
	Create a Java application for the game Tic-Tac-Toe as if you were a senior software developer
	Create a Java application for the game Tic-Tac-Toe as if you were a computer science teacher giving an example for first year computer science students
<b>Memetic proxy</b> Java Blackjack	Create a Java application for the game Blackjack as if you were a computer science student in their first year
	Create a Java application for the game Blackjack as if you were a senior software developer
	Create a Java application for the game Blackjack as if you were a computer science teacher giving an example for first year computer science students
<b>Memetic proxy</b> Python Tic-Tac-Toe	Create a Python application for the game Tic-Tac-Toe as if you were a computer science student in their first year
	Create a Python application for the game Tic-Tac-Toe as if you were a senior software developer
	Create a Python application for the game Tic-Tac-Toe as if you were a computer science teacher giving an example for first year computer science students
<b>Memetic proxy</b> Python	Create a Python application for the game Blackjack as if you were a computer science student in their first year

Blackjack	Create a Python application for the game Blackjack as if you were a senior software developer
	Create a Python application for the game Blackjack as if you were a computer science teacher giving an example for first year computer science students
<b>Constraints</b> Java Tic-Tac-Toe	Create a Java application for the game Tic-Tac-Toe, the game must be object-oriented
	Create a Java application for the game Tic-Tac-Toe, the code should be reliable and scalable
	Create a Java application for the game Tic-Tac-Toe, the game handles invalid user input such as numbers and characters, no code is unused, and the code contains no errors
<b>Constraints</b> Java Blackjack	Create a Java application for the game Blackjack, the game must be object-oriented
	Create a Java application for the game Blackjack, the code should be reliable and scalable
	Create a Java application for the game Blackjack, the game handles invalid user input such as numbers and characters, no code is unused, and the code contains no errors
<b>Constraints</b> Python Tic-Tac-Toe	Create a Python application for the game Tic-Tac-Toe, the game must be object-oriented
	Create a Python application for the game Tic-Tac-Toe, the code should be reliable and scalable
	Create a Python application for the game Tic-Tac-Toe, the game handles invalid user input such as numbers and characters, no code is unused, and the code contains no errors
<b>Constraints</b> Python Blackjack	Create a Python application for the game Blackjack, the game must be object-oriented
	Create a Python application for the game Blackjack, the code should be reliable and scalable
	Create a Python application for the game Blackjack, the game handles invalid user input such as numbers and characters, no code is unused, and the code contains no errors
<b>Meta Prompt</b> Java Tic-Tac-Toe	Appendix F - Java Tic-Tac-Toe A
	Appendix F - Java Tic-Tac-Toe B
	Appendix F - Java Tic-Tac-Toe C
<b>Meta Prompt</b> Java Blackjack	Appendix F - Java Blackjack A
	Appendix F - Java Blackjack B

	Appendix F - Java Blackjack C
<b>Meta Prompt</b> Python Tic-Tac-Toe	Appendix F - Python Tic-Tac-Toe A
	Appendix F - Python Tic-Tac-Toe B
	Appendix F - Python Tic-Tac-Toe C
<b>Meta Prompt</b> Python Blackjack	Appendix F - Python Blackjack A
	Appendix F - Python Blackjack B
	Appendix F - Python Blackjack C

# Appendix B - Software Requirements Specification

## Tic-Tac-Toe

### User interaction

- The user can control at least one of the players.
- User is told which players (markers) turn it is.
- User is prompted to input a value for the row and column. (Alternatively: a number for position)
- The game board is printed after a player places a marker. (This includes the final move of the game)
- The game result is printed after a win or tie.

### Game logic

- A 3x3 board which contains markers exists.
- The board starts empty.
- Players alternate placing markers.
- A turn only ends if a valid marker has been placed.
  - A marker cannot be placed on an occupied position.
  - A valid move has to be within the confines of the game board.
- It is possible to win by placing 3 markers connected to each other either vertically, horizontally, or diagonally.
- If all positions on the board are occupied and no player has won, the game is a tie.
- The player whose markers achieved the win condition is the winner. (If X wins but the stated winner is O then it is a failure)

### Input validation

- For `scanner.nextInt()` or `int(input())` the program needs to handle invalid integer inputs without crashing. (It is acceptable that the program crashes due to a string input)
- The player can reenter a new value if the previous input was invalid. (Or a standard value is used)

### Unused code

- A declared method should be used at least once.
- A declared variable should be used at least once.
- Parameters in method/function should be used at least once.

### Code errors

- A variable should not be directly accessed outside its scope. (No variable access error)
- The code should compile at runtime free from syntax errors.
- The game should not crash when accessing a function, variable, or object.
- The game should be semantically correct.

- An example of incorrect code would be a function which assumes empty cells are defined as "" when they are defined as " ", thus resulting in incorrect program behavior that prevents runtime evaluation.
- Actual-parameter type: Are the actual-parameter types in an API call consistent with the corresponding formal-parameter types?

## Blackjack

### User interaction

- User is shown their hand.
- The user is shown the dealer's hand with the first card face up and the second card hidden. (If the hidden card can be inferred due to hand value being printed it fails)
- The owner of the card is shown.
- The User is prompted to input a value either hit or stand.
- If the player hits, their new card is displayed.
- If the player busts the game is over.
- If the player stands, the dealer's hidden card will be revealed.
- If the dealer hits, their new card is displayed.
- If the dealer stands(hand value >16), the winner is printed.
- If the player or dealer busts it is printed.

### Game logic

- A deck of 52 cards is used. (More cards are fine so long as it is a complete deck)
- The deck is shuffled.
- Cards are taken from the top of the deck. (The same card is not taken twice)
- The value of an active hand can be at most 21 points, if it exceeds this value it is "bust".
- The value of an ace is 11 unless that would result in a bust in which case it is worth 1.
- The face cards are worth 10 points (jack, queen, king).
- All other cards are worth their rank (10 is worth 10, 7 is 7, etc).
- Valid rank values are: 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack (J), Queen (Q), King (K), Ace (A).
- Valid suit values are: Hearts (♥), Diamonds (♦), Clubs (♠), Spades (♣). (No stated suit is fine so long as there are 4 of each card)
- At the start of the game the player receives 2 cards.
- At the start of the game the dealer receives 2 cards, one is hidden (American rules). Alternative European rules dealer receives 1 card and will draw the second after the player stands.
- If the player hits, one card will be drawn from the deck and added to the player's hand.
- If the player stands, it will be the dealer's turn.
- If the value of the dealer's hand is below 17 the dealer hits.
- If the value of the dealer's hand is above 16 the dealer stands.
- If no one goes bust the hand with the highest value wins.
- If both hands have the same value it is a push (tie). Alternative house rules dealer wins.
- If the game has rounds it is expected that the hands are emptied of cards, so that the rounds are self-contained.

## **Input validation**

- If an invalid input was entered the program should not crash. (This includes scanners and inputs that expect int but received a string input)
- The player can reenter a new value if the previous input was invalid. (Alternative: standard value either stand or hit)

## **Unused code**

- A declared method should be used at least once.
- A declared variable should be used at least once.
- Parameters in method/function should be used at least once.

## **Code errors**

- A variable should not be directly accessed outside its scope. (No variable access error)
- The code should compile at runtime free from syntax errors.
- The game should not crash when accessing a function, variable, or object.
- The game should be semantically correct.
  - An example of incorrect code would be a function which assumes empty cells are defined as "" when they are defined as " ", thus resulting in incorrect program behavior that prevents runtime evaluation.
  - Actual-parameter type: Are the actual-parameter types in an API call consistent with the corresponding formal-parameter types?

# Appendix C - Luckycard

## Java

```
//Import necessary libraries
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;

public class Luckycard {
    private static Deck deck;

    public static void main(String[] args) {
        deck = new Deck();
        deck.shuffle();

        System.out.println("Welcome to LuckyCard game!");
        Scanner scanner = new Scanner(System.in);

        // Loop until the deck is empty or the user quits
        while (!deck.isEmpty()) {
            // Prompt the user to play again or quit
            System.out.print("\n===== > Press ENTER to play again or "q" to quit:
");

            // Read the user input
            String input = scanner.nextLine();

            // If the user enters "q", break out of the loop
            if (input.equals("q")) {
                break;
            }
            // Otherwise, play another round
            play();
        }

        // Print a message indicating that the deck is empty
        System.out.println("Deck is empty. Thank you for playing and welcome back!");
    }

    // Define a method to play a round of the game
    private static void play() {
        System.out.println("\n----- Playing a game round");
        // Draw three cards from the deck to play a game round
        Card card1 = deck.drawCard();
        Card card2 = deck.drawCard();
        Card card3 = deck.drawCard();

        // Print the value and bonus of each card drawn
        System.out.println("Card 1: " + card1 + " → Value = " + (card1.getValue() +
card1.getBonus()));
        System.out.println("Card 2: " + card2 + " → Value = " + (card2.getValue() +
card2.getBonus()));
        System.out.println("Card 3: " + card3 + " → Value = " + (card3.getValue() +
card3.getBonus()));
    }
}
```



```

        // Determine if the player has won or lost the game round based on the values of
the cards.
        int value1 = card1.getValue() + card1.getBonus();
        int value2 = card2.getValue() + card2.getBonus();
        int value3 = card3.getValue() + card3.getBonus();
        if (value3 > value1 && value3 < value2 || value3 < value1 && value3 > value2) {
            System.out.println("\nYou win!");
        } else {
            System.out.println("\nYou lose.");
        }
    }
}

enum Suit {
    DIAMONDS, CLUBS, HEARTS, SPADES
}

enum Rank {
    ACE(1), TWO(2), THREE(3), FOUR(4), FIVE(5), SIX(6), SEVEN(7), EIGHT(8), NINE(9),
    TEN(10), JACK(11), QUEEN(12),
    KING(13);

    private int value;

    private Rank(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

class Deck {
    private ArrayList<Card> cards;

    // Create a new deck of cards and add each card to it
    public Deck() {
        cards = new ArrayList<Card>();
        for (Suit suit : Suit.values()) {
            for (Rank rank : Rank.values()) {
                cards.add(new Card(suit, rank));
            }
        }
    }

    // Shuffle the deck to randomize the order of the cards
    public void shuffle() {
        Collections.shuffle(cards);
    }

    // Check if the deck is empty to determine if the game should continue or end
    public boolean isEmpty() {
        return cards.isEmpty();
    }
}

```

```

    public Card drawCard() {
        if (isEmpty()) {
            return null;
        }
        return cards.remove(0);
    }
}

class Card {
    private Suit suit;
    private Rank rank;

    public Card(Suit suit, Rank rank) {
        this.suit = suit;
        this.rank = rank;
    }

    public int getValue() {
        return rank.getValue();
    }

    public int getBonus() {
        if (suit == Suit.DIAMONDS) {
            return 4;
        } else if (suit == Suit.CLUBS) {
            return 6;
        } else if (suit == Suit.HEARTS) {
            return 8;
        } else {
            return 10;
        }
    }

    public String toString() {
        return rank + " of " + suit;
    }
}
}

```

## Python

```

#Import necessary libraries
import random

class Suit:
    DIAMONDS = "Diamonds"
    CLUBS = "Clubs"
    HEARTS = "Hearts"
    SPADES = "Spades"

class Rank:
    ACE = 1
    TWO = 2
    THREE = 3

```

```

FOUR = 4
FIVE = 5
SIX = 6
SEVEN = 7
EIGHT = 8
NINE = 9
TEN = 10
JACK = 11
QUEEN = 12
KING = 13

def __init__(self, value):
    self.value = value

class Card:
    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank

    def getValue(self):
        return self.rank.value

    def getBonus(self):
        if self.suit == Suit.DIAMONDS:
            return 4
        elif self.suit == Suit.CLUBS:
            return 6
        elif self.suit == Suit.HEARTS:
            return 8
        else:
            return 10

    def __str__(self):
        rank_names = {1: "ACE", 2: "TWO", 3: "THREE", 4: "FOUR", 5: "FIVE", 6: "SIX", 7:
"SEVEN", 8: "EIGHT", 9: "NINE", 10: "TEN", 11: "JACK", 12: "QUEEN", 13: "KING"}
        return f"{rank_names[self.rank.value]} of {self.suit}"

class Deck:
    def __init__(self):
        self.cards = []
        # Create a new deck of cards and add each card to it
        for suit in [Suit.DIAMONDS, Suit.CLUBS, Suit.HEARTS, Suit.SPADES]:
            for rank in [Rank.ACE, Rank.TWO, Rank.THREE, Rank.FOUR, Rank.FIVE, Rank.SIX,
Rank.SEVEN, Rank.EIGHT, Rank.NINE, Rank.TEN, Rank.JACK, Rank.QUEEN, Rank.KING]:
                self.cards.append(Card(suit, Rank(rank)))

    def shuffle(self):
        # Shuffle the deck to randomize the order of the cards
        random.shuffle(self.cards)

```

```

def isEmpty(self):
    # Check if the deck is empty to determine if the game should continue or end
    return not bool(self.cards)

def drawCard(self):
    if self.isEmpty():
        return None
    return self.cards.pop(0)

def play():
    print("\n----- Playing a game round")
    # Draw three cards from the deck to play a game round
    card1 = deck.drawCard()
    card2 = deck.drawCard()
    card3 = deck.drawCard()
    # Print the value and bonus of each card drawn
    print(f"Card 1: {card1} → Value = {card1.getValue() + card1.getBonus()}")
    print(f"Card 2: {card2} → Value = {card2.getValue() + card2.getBonus()}")
    print(f"Card 3: {card3} → Value = {card3.getValue() + card3.getBonus()}")

    # Determine if the player has won or lost the game round based on the values of the
    cards.
    value1 = card1.getValue() + card1.getBonus()
    value2 = card2.getValue() + card2.getBonus()
    value3 = card3.getValue() + card3.getBonus()
    if (value3 > value1 and value3 < value2) or (value3 < value1 and value3 > value2):
        print("\nYou win!")
    else:
        print("\nYou lose.")

# Main program
deck = Deck()
deck.shuffle()

print("Welcome to LuckyCard game!")

# Loop until the deck is empty or the user quits
while not deck.isEmpty():
    # Prompt the user to play a game round or quit
    choice = input("Press 'p' to play a game round, or 'q' to quit: ")
    if choice == 'p':
        play()
    elif choice == 'q':
        print("Thanks for playing!")
        break
    else:
        print("Invalid input. Please try again.")

```

# Appendix D - Hangman

## Java

```
import java.util.Scanner;

public class Hangman {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String[] words = { "prompt", "programming", "artificial", "intelligence",
"evaluation" };
        String word = words[(int) (Math.random() * words.length)];
        char[] guesses = new char[word.length()];
        for (int i = 0; i < guesses.length; i++) {
            guesses[i] = '_';
        }
        int incorrectGuesses = 0;
        System.out.println(new String(guesses));
        while (incorrectGuesses < 6 && new String(guesses).contains("_")) {
            System.out.println("Guess a letter:");
            String input = scanner.nextLine().toLowerCase();
            try {
                char letter = input.charAt(0);
                if (!Character.isLetter(letter)) {
                    throw new IllegalArgumentException("Invalid input. Please
enter a letter.");
                }
                boolean found = false;
                for (int i = 0; i < word.length(); i++) {
                    if (word.charAt(i) == letter) {
                        guesses[i] = letter;
                        found = true;
                    }
                }
                if (!found) {
                    incorrectGuesses++;
                    System.out.println("Incorrect! You have " + (6 -
incorrectGuesses) + " guesses left.");
                }
                System.out.println(new String(guesses));
            } catch (IllegalArgumentException e) {
                System.out.println(e.getMessage());
            }
        }
        if (new String(guesses).contains("_")) {
            System.out.println("Sorry, you ran out of guesses. The word was " + word);
        } else {
            System.out.println("Congratulations! You guessed the word " + word + "
correctly.");
        }

        scanner.close();
    }
}
```

# Python

```
import random

words = ["prompt", "programming", "artificial", "intelligence", "evaluation"]
word = random.choice(words)
guesses = ["_"] * len(word)
incorrect_guesses = 0

print(" ".join(guesses))

while incorrect_guesses < 6 and "_" in guesses:
    try:
        letter = input("Guess a letter: ").lower()
        if len(letter) != 1 or not letter.isalpha():
            print("Invalid input. Please enter a single letter.")
            continue

        found = False
        for i, c in enumerate(word):
            if c == letter:
                guesses[i] = letter
                found = True

        if not found:
            incorrect_guesses += 1
            print(f"Incorrect! You have {6 - incorrect_guesses} guesses left.")

        print(" ".join(guesses))
    except:
        print("Something went wrong try again!")

if "_" in guesses:
    print(f"Sorry, you ran out of guesses. The word was {word}.")
else:
    print(f"Congratulations! You guessed the word {word} correctly.")
```

# Appendix E - ConnectFour

## Java

```
import java.util.Scanner;

public class ConnectFour {
    private char[][] board; // 2D array to represent the game board
    private int numRows; // number of rows on the game board
    private int numCols; // number of columns on the game board
    private char currentPlayer; // current player (either 'X' or 'O')

    public ConnectFour(int numRows, int numCols) {
        this.numRows = numRows;
        this.numCols = numCols;
        board = new char[numRows][numCols];
        currentPlayer = 'X';
        initializeBoard();
    }

    private void initializeBoard() {
        // set every cell on the board to '-'
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
                board[i][j] = '-';
            }
        }
    }

    private void printBoard() {
        // print out the game board
        for (int i = numRows - 1; i >= 0; i--) {
            for (int j = 0; j < numCols; j++) {
                System.out.print(board[i][j] + " ");
            }
            System.out.println();
        }
        System.out.println();
    }

    private boolean isValidMove(int col) {
        // check if the given column is a valid move
        if (col < 0 || col >= numCols) {
            return false;
        }
        if (board[numRows - 1][col] != '-') {
            return false;
        }
        return true;
    }

    private void makeMove(int col) {
        // make a move in the given column
        for (int i = 0; i < numRows; i++) {
            if (board[i][col] == '-') {
```

```

        board[i][col] = currentPlayer;
        break;
    }
}

private boolean checkForWin() {
    // check for a win on the game board
    // check rows
    for (int i = 0; i < numRows; i++) {
        for (int j = 0; j <= numCols - 4; j++) {
            if (board[i][j] == currentPlayer && board[i][j + 1] ==
currentPlayer && board[i][j + 2] == currentPlayer
                && board[i][j + 3] == currentPlayer) {
                return true;
            }
        }
    }
    // check columns
    for (int j = 0; j < numCols; j++) {
        for (int i = 0; i <= numRows - 4; i++) {
            if (board[i][j] == currentPlayer && board[i + 1][j] ==
currentPlayer && board[i + 2][j] == currentPlayer
                && board[i + 3][j] == currentPlayer) {
                return true;
            }
        }
    }
    // check diagonals
    for (int i = 0; i <= numRows - 4; i++) {
        for (int j = 0; j <= numCols - 4; j++) {
            if (board[i][j] == currentPlayer && board[i + 1][j + 1] ==
currentPlayer
                && board[i + 2][j + 2] == currentPlayer && board[i +
3][j + 3] == currentPlayer) {
                return true;
            }
        }
    }
    for (int i = 0; i <= numRows - 4; i++) {
        for (int j = numCols - 1; j >= 3; j--) {
            if (board[i][j] == currentPlayer && board[i + 1][j - 1] ==
currentPlayer
                && board[i + 2][j - 2] == currentPlayer && board[i +
3][j - 3] == currentPlayer) {
                return true;
            }
        }
    }
    return false;
}

private boolean checkForDraw() {
    // check if the game is a draw
    for (int i = 0; i < numRows; i++) {
        for (int j = 0; j < numCols; j++) {
            if (board[i][j] == '-') {

```



```

        return false;
    }
}
return true;
}

private void switchPlayer() {
    // switch the current player
    if (currentPlayer == 'X') {
        currentPlayer = 'O';
    } else {
        currentPlayer = 'X';
    }
}

public void play() {
    Scanner scanner = new Scanner(System.in);
    boolean gameOver = false;
    while (!gameOver) {
        // print out the game board
        printBoard();
        // get the current player's move
        System.out.println("Player " + currentPlayer + ", please enter your move
(0- " + (numCols - 1) + "): ");
        try {
            int col = scanner.nextInt();
            // validate the move
            if (!isValidMove(col)) {
                System.out.println("Invalid move. Please try again.");
                continue;
            }
            // make the move
            makeMove(col);
            // check for a win
            if (checkForWin()) {
                System.out.println("Player " + currentPlayer + " wins!");
                gameOver = true;
            } else if (checkForDraw()) {
                System.out.println("Game over. It's a draw.");
                gameOver = true;
            } else {
                // switch the player
                switchPlayer();
            }
        } catch (Exception e) {
            System.out.println("Invalid input");
            scanner.nextLine();
        }
    }
    // print out the final game board
    printBoard();
    scanner.close();
}

public static void main(String[] args) {
    ConnectFour game = new ConnectFour(6, 7);
}

```

```
        game.play();
    }
}
```

## Python

```
class ConnectFour:
    def __init__(self, numRows, numCols):
        self.numRows = numRows
        self.numCols = numCols
        self.board = [['-' for _ in range(numCols)] for _ in range(numRows)]
        self.currentPlayer = 'X'

    def initializeBoard(self):
        for i in range(self.numRows):
            for j in range(self.numCols):
                self.board[i][j] = '-'

    def printBoard(self):
        for i in range(self.numRows - 1, -1, -1):
            for j in range(self.numCols):
                print(self.board[i][j], end=' ')
            print()
        print()

    def isValidMove(self, col):
        if col < 0 or col >= self.numCols:
            return False
        if self.board[self.numRows - 1][col] != '-':
            return False
        return True

    def makeMove(self, col):
        for i in range(self.numRows):
            if self.board[i][col] == '-':
                self.board[i][col] = self.currentPlayer
                break

    def switchPlayer(self):
        if self.currentPlayer == 'X':
            self.currentPlayer = 'O'
        else:
            self.currentPlayer = 'X'

    def checkForWin(self):
        # check for a horizontal win
        for row in range(self.numRows):
```

```

        for col in range(self.numCols - 3):
            if self.board[row][col] != '-' and self.board[row][col] ==
self.board[row][col + 1] == self.board[row][col + 2] == self.board[row][col + 3]:
                return True

    # check for a vertical win
    for row in range(self.numRows - 3):
        for col in range(self.numCols):
            if self.board[row][col] != '-' and self.board[row][col] == self.board[row +
1][col] == self.board[row + 2][col] == self.board[row + 3][col]:
                return True

    # check for a diagonal win (top-left to bottom-right)
    for row in range(self.numRows - 3):
        for col in range(self.numCols - 3):
            if self.board[row][col] != '-' and self.board[row][col] == self.board[row +
1][col + 1] == self.board[row + 2][col + 2] == self.board[row + 3][col + 3]:
                return True

    # check for a diagonal win (bottom-left to top-right)
    for row in range(3, self.numRows):
        for col in range(self.numCols - 3):
            if self.board[row][col] != '-' and self.board[row][col] == self.board[row -
1][col + 1] == self.board[row - 2][col + 2] == self.board[row - 3][col + 3]:
                return True

    # no win found
    return False

def checkForDraw(self):
    # check if the board is full
    for row in range(self.numRows):
        for col in range(self.numCols):
            if self.board[row][col] == '-':
                return False

    # board is full, game is a draw
    return True

def play(self):
    input_func = input
    gameOver = False
    while not gameOver:
        self.printBoard()
        col = input_func("Player {}, please enter your move (0-{}):
".format(self.currentPlayer, self.numCols - 1))
        try:
            col = int(col)

```

```
        if not self.isValidMove(col):
            print("Invalid move. Please try again.")
            continue
        self.makeMove(col)
        if self.checkForWin():
            print("Player", self.currentPlayer, "wins!")
            gameOver = True
        elif self.checkForDraw():
            print("Game over. It's a draw.")
            gameOver = True
        else:
            self.switchPlayer()
    except ValueError:
        print("Invalid input")

    self.printBoard()

if __name__ == '__main__':
    game = ConnectFour(6, 7)
    game.play()
```

## Appendix F - Meta Prompts

### Java Tic-Tac-Toe

#### Prompt variation A

**Prompt:** Write a prompt that can be used by ChatGPT to create a simple Java application for the game Tic-Tac-Toe

#### Response:

Create a Java application for the classic game of Tic-Tac-Toe, which allows two players to take turns placing their marks (X or O) on a 3x3 grid. The game should check for a winner after each turn and display the winner if there is one, or declare a tie if the grid is filled without a winner. The application should also allow the players to play again if they wish, resetting the grid for a new game.

#### Prompt variation B

**Prompt:** Write a prompt that can be used by ChatGPT to create a simple Java application for the game Tic-Tac-Toe

#### Response:

Write a program in Java that allows two players to play a game of Tic-Tac-Toe. The program should prompt the players to enter their moves (as coordinates on a 3x3 grid) and display the game board after each move. The program should also check for a winning condition (three in a row, column, or diagonal) and declare the winner if one is found. If the game ends in a tie, the program should display a message indicating that the game is over. Additionally, the program should handle invalid user input (e.g., entering a move outside of the 3x3 grid or trying to place a piece on a square that is already occupied) and prompt the user to enter a valid move.

#### Prompt variation C

**Prompt:** Write a prompt that can be used by ChatGPT to create a simple Java application for the game Tic-Tac-Toe

#### Response:

Write a Java program that allows two players to play Tic-Tac-Toe against each other on the command line. The program should display a 3x3 grid of cells where players can enter their moves. The game should continue until one player wins or the game ends in a tie.

Here are the basic requirements for the program:

- The program should prompt the first player to enter their move (e.g. "Player 1, enter row and column number:").
- The program should validate the move entered by the player and ensure that the cell is empty. If the move is invalid, the program should display an error message and prompt the player to enter a valid move.

- The program should alternate between the two players until one player wins or the game ends in a tie.
- The program should detect when one player has won and display a message announcing the winner.
- The program should detect when the game ends in a tie and display a message announcing the tie.
- The program should allow the players to choose whether to play again or quit after the game is over.

## Java Blackjack

### Prompt variation A

**Prompt:** Write a prompt that can be used by ChatGPT to create a simple Java application for the game Blackjack

### Response:

Create a Java application that simulates the game of Blackjack between the user and the computer dealer. The game should start with the user being dealt two cards and the dealer being dealt one card face up and one card face down. The user should be able to choose whether to hit (receive another card) or stand (keep their current hand). If the user's hand exceeds 21, they lose the game. Once the user stands, the dealer should reveal their face down card and continue to hit until their hand exceeds 17. If the dealer's hand exceeds 21, the user wins. If both the user and dealer have hands under 21, the hand with the highest value wins. The game should end when either the user or dealer wins or when the user chooses to quit.

### Prompt variation B

**Prompt:** Write a prompt that can be used by ChatGPT to create a simple Java application for the game Blackjack

### Response:

Create a Java application that simulates a game of Blackjack between a player and a dealer. The program should start by prompting the player for their name and then dealing two cards to both the player and the dealer. The player's cards should be displayed to them, and the dealer's first card should be displayed face down.

The player should then be prompted to either "hit" or "stand". If the player chooses to hit, they should be dealt another card, and their total score should be updated. If the player chooses to stand, the game should proceed to the dealer's turn.

The dealer should continue to hit until their score is at least 17. At this point, the dealer should reveal their face-down card and display their total score.

The winner of the game should be determined by comparing the player's and dealer's scores. If the player's score is higher than the dealer's score and is less than or equal to 21, the player wins. If the dealer's score is higher than the player's score and is less than or equal to 21, the dealer wins. If both the player and the dealer have the same score, it's a tie.

### **Prompt variation C**

**Prompt:** Write a prompt that can be used by ChatGPT to create a simple Java application for the game Blackjack

### **Response:**

Write a simple Java application for the game Blackjack. The application should have the following features:

1. The game should be played between the player and the dealer.
2. The player should be able to choose their bet amount at the start of each game.
3. The player should be dealt two cards at the start of each game.
4. The dealer should be dealt one card at the start of each game.
5. The player should be able to hit (draw another card) or stand (keep their current hand) during their turn.
6. The dealer should continue to hit until they have a hand value of 17 or greater.
7. The game should end when either the player or the dealer reaches a hand value of 21, or when both the player and dealer stand and the hand values are compared to determine the winner.
8. The game should keep track of the player's balance and adjust it based on the outcome of each game.
9. The game should allow the player to play again or quit at the end of each game.

## **Python Tic-Tac-Toe**

### **Prompt variation A**

**Prompt:** Write a prompt that can be used by ChatGPT to create a simple Python application for the game Tic-Tac-Toe

### **Response:**

Write a Python program that allows two players to play a game of Tic-Tac-Toe on the console. The program should start by printing out an empty 3x3 board. Players should then take turns entering their moves by specifying the row and column of the square they want to mark. The program should validate the moves to ensure that the square is empty and that the input is within the range of the board. After each move, the program should check for a win or a tie and print out the updated board. The game should continue until one player wins or the board is full, at which point the program should declare the winner or a tie and ask if the players want to play again. The program should handle errors gracefully and allow the players to quit at any time.

### **Prompt variation B**

**Prompt:** Write a prompt that can be used by ChatGPT to create a simple Python application for the game Tic-Tac-Toe

**Response:**

"Create a Python program that allows two players to play Tic-Tac-Toe against each other on the command line. The program should ask for player names, display the current board state after each turn, and determine the winner or a tie. The board should be represented as a 3x3 grid of spaces that is initially empty. Players should take turns placing Xs and Os on the board until one player wins or the board is full. If a player chooses an already-occupied space, the program should ask them to choose another one. At the end of each game, the program should ask the players if they want to play again."

**Prompt variation C**

**Prompt:** Write a prompt that can be used by ChatGPT to create a simple Python application for the game Tic-Tac-Toe

**Response:**

Create a Python program that allows two players to play a game of Tic-Tac-Toe on the command line. The program should:

- Display an empty Tic-Tac-Toe board at the start of the game.
- Prompt the first player to enter their move (either 'X' or 'O') and the position they would like to place their marker on the board (using row and column numbers).
- Verify that the chosen position is valid (i.e., not already occupied) and display an error message if it is not.
- Update the board to show the first player's move and display the updated board.
- Repeat the above steps for the second player, using the opposite marker ('O' if the first player chose 'X' and vice versa).
- Continue taking turns until one player has won or the game ends in a tie.
- At the end of the game, display a message indicating who won or if the game ended in a tie.
- Ask the players if they want to play again and start a new game if they do.

Feel free to add any additional features or customization to the game.

## Python Blackjack

**Prompt variation A**

**Prompt:** Write a prompt that can be used by ChatGPT to create a simple Python application for the game Blackjack.

**Response:**

Create a Python application for the game of Blackjack that allows the user to play against the computer. The application should include the following features:



- The game should start with the player and the computer being dealt two cards each. The player's cards should be visible, but only one of the computer's cards should be visible.
- The player should be able to choose whether to "hit" (receive another card) or "stand" (keep their current hand).
- The computer should automatically hit until their hand is worth at least 17 points.
- Aces should be worth 1 or 11 points, whichever is more advantageous for the player.
- The game should end when either the player or the computer has a hand worth 21 points, or when both have chosen to stand. If the player wins, the game should display a congratulatory message. If the computer wins, the game should display a message indicating the player's loss. If there is a tie, the game should display a message indicating a push.
- After the game has ended, the player should be given the option to play again or to exit the game.

### **Prompt variation B**

**Prompt:** Create a prompt that can be used by ChatGPT to write a program in Python for the game Blackjack.

### **Response:**

Write a Python program that allows a user to play the game of Blackjack against a computer dealer. The game should follow the standard rules of Blackjack, where the goal is to have a hand value of 21 or as close to 21 as possible without going over. The program should:

1. Allow the user to place a bet at the beginning of each round.
2. Deal two cards to the player and two cards to the dealer.
3. Display the player's cards and the dealer's face-up card.
4. Allow the player to "hit" (take another card) or "stand" (keep their current hand).
5. If the player's hand goes over 21, the game should end and the player loses their bet.
6. If the player chooses to stand, the dealer should reveal their face-down card and hit until their hand value is 17 or higher.
7. If the dealer's hand goes over 21, the player wins their bet.
8. If neither the player nor the dealer busts, the hand with the higher value wins.
9. At the end of each round, the program should ask the user if they want to play again.

Note that face cards (jacks, queens, and kings) count as 10 and aces can count as either 1 or 11, depending on which value would be more advantageous for the player.

### **Prompt variation C**

**Prompt:** Write a prompt that can be used by ChatGPT to implement the game Blackjack using Python

### **Response:**

Write a program that allows the user to play a game of Blackjack against the computer. The program should start by prompting the user for their name and welcoming them to the game. The rules of the game should be explained to the user.

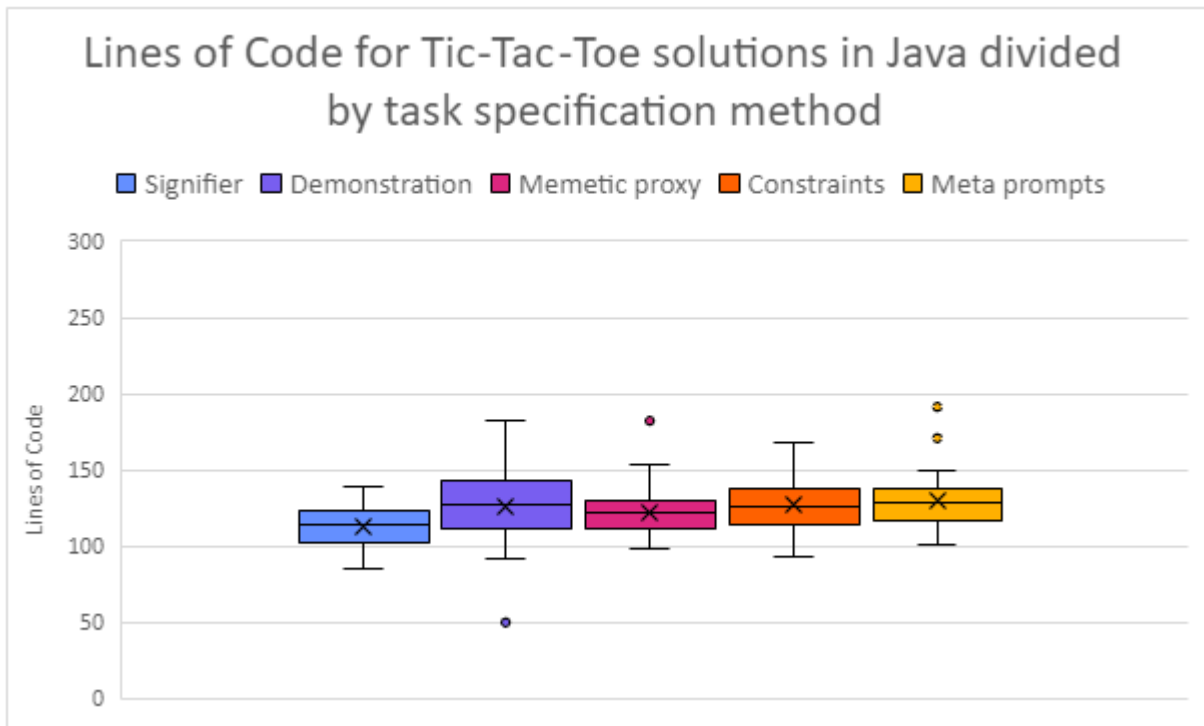
The program should then ask the user how many decks of cards they would like to play with and create a deck of cards accordingly. The deck should be shuffled before the game starts.

The game should then begin. The user should be dealt two cards, and the computer should be dealt two cards (one face up and one face down). The user should be prompted to hit or stand. If they choose to hit, they should be dealt another card. If they choose to stand, the game should move on to the computer's turn.

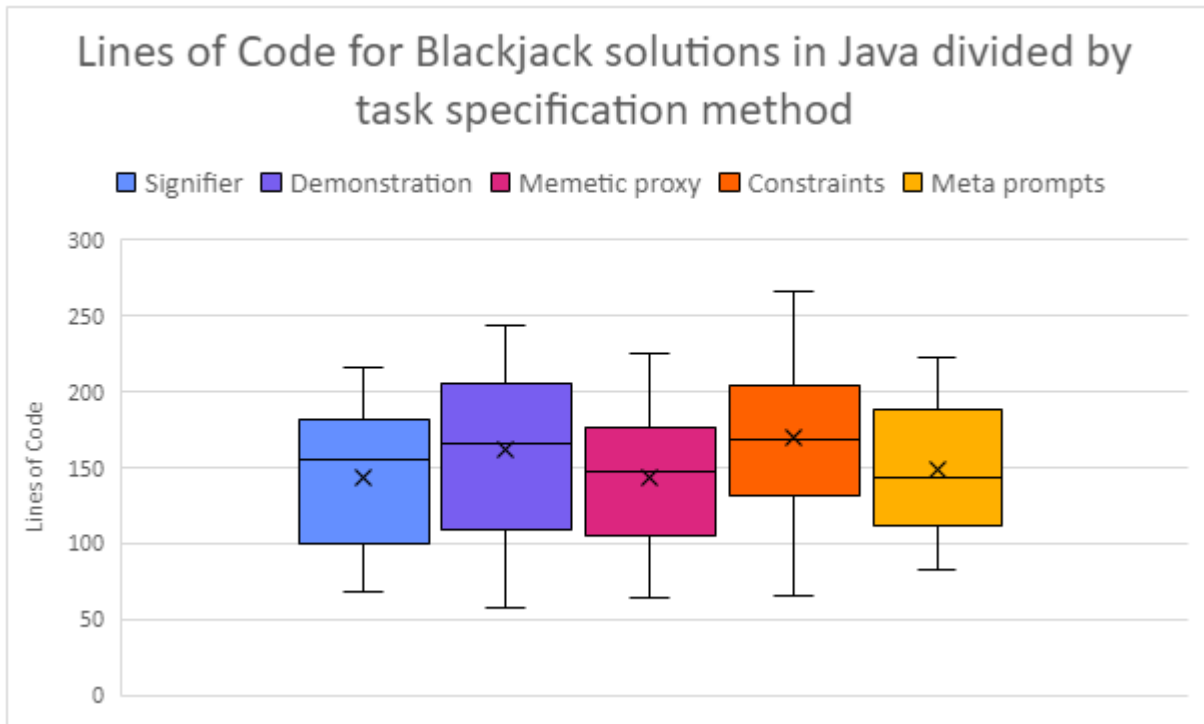
The computer's turn should begin once the user has chosen to stand. The computer should continue to hit until their hand is worth at least 17. Once the computer has finished their turn, the program should determine the winner and display the result.

The program should keep track of the user's bankroll and allow them to place bets on each hand. The user should be prompted to enter their bet at the beginning of each hand. If the user wins, their bankroll should be increased by the amount of their bet. If the user loses, their bankroll should be decreased by the amount of their bet. The game should continue until the user decides to quit or runs out of money.

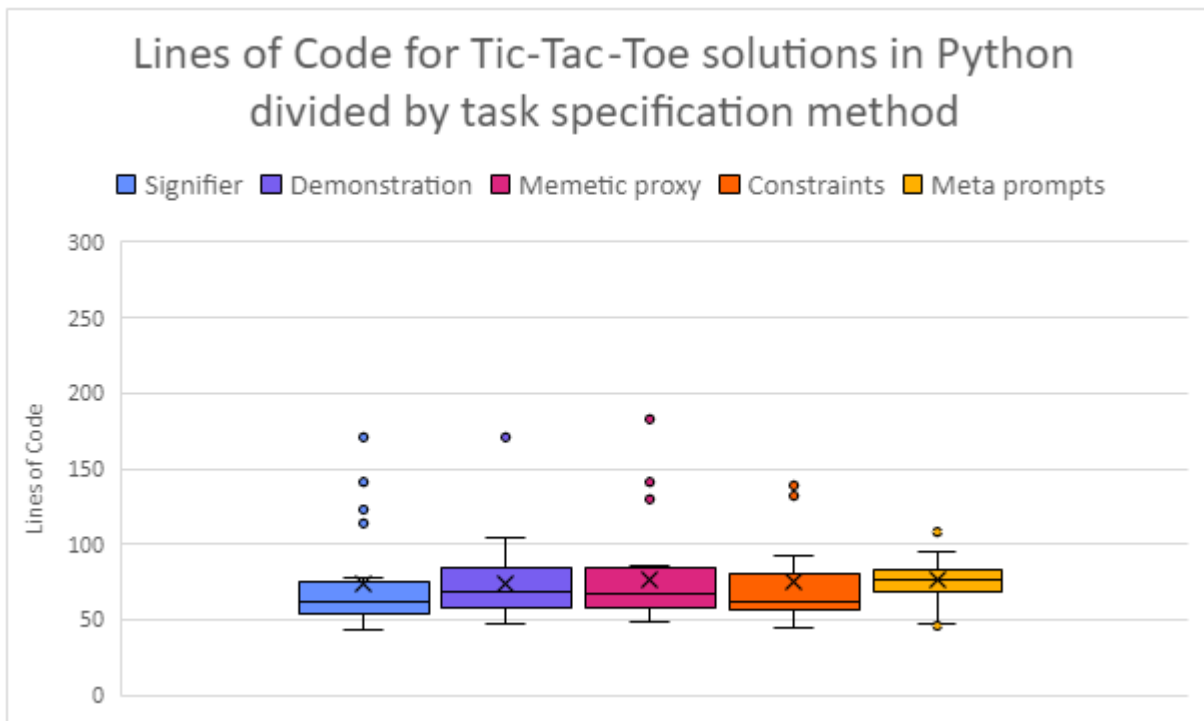
## Appendix G - Lines of Code Figures



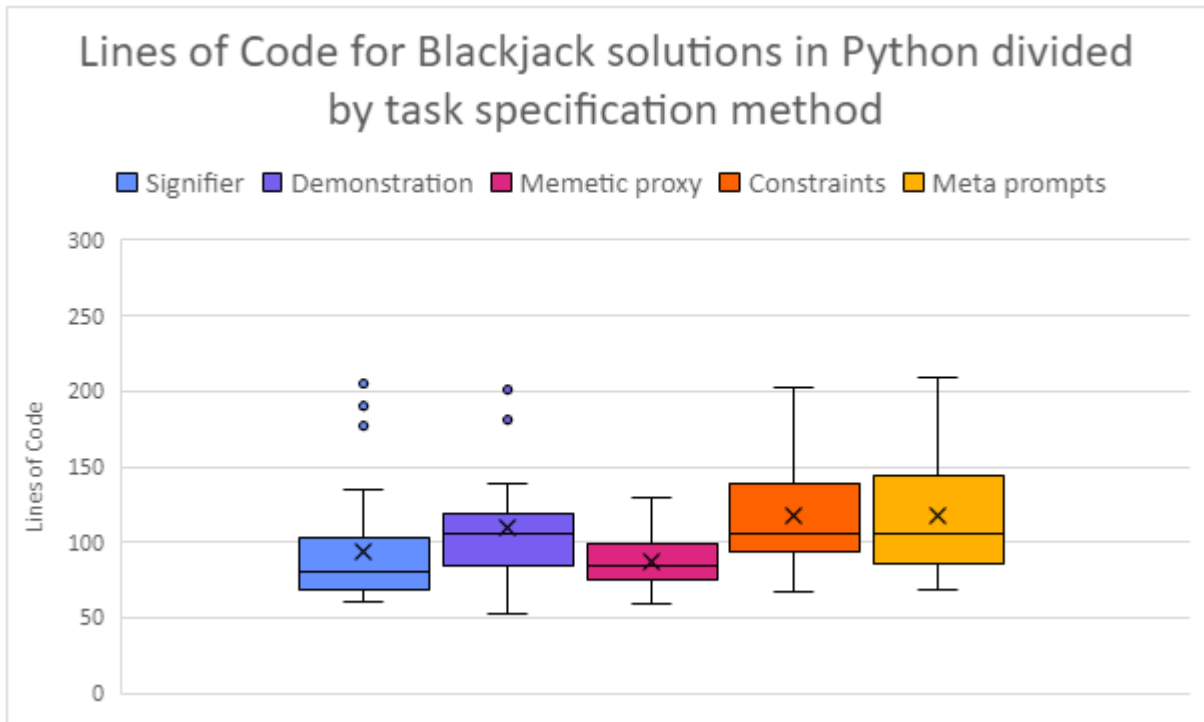
**Figure G.1:** Box plots displaying the five-number summary of lines of code for solutions generated using the chosen task specification methods for the programming task Tic-Tac-Toe in Java.



**Figure G.2:** Box plots displaying the five-number summary of lines of code for solutions generated using the chosen task specification methods for the programming task Blackjack in Java.



**Figure G.3:** Box plots displaying the five-number summary of lines of code for solutions generated using the chosen task specification methods for the programming task Tic-Tac-Toe in Python.



**Figure G.4:** Box plots displaying the five-number summary of lines of code for solutions generated using the chosen task specification methods for the programming task Blackjack in Python.

Figure G.1, Figure G.2, Figure G.3, Figure G.4 displays box plots, showing the five-number summary of Lines Of Code (LOC) for code generated using different task specification methods for programming tasks Tic-Tac-Toe and blackjack in programming languages Java and Python. The five-number summary includes the smallest observation, first quartile, median, third quartile, and largest observation (Austin, 2009). Additionally the average LOC is shown as an “X” in the box plot. The graphs show a relatively consistent spread when comparing task specification methods, for Blackjack in Java there is a higher spread while Tic-Tac-Toe in Python has the smallest spread with a few outliers. Tic-Tac-Toe in Java has a relatively consistent spread, and lastly Blackjack in Python has the most inconsistent spread among the task specification methods as memetic proxy has a small spread while constraints and meta prompts have a high spread.