UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Comparación de técnicas de generación de código

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Martí Pérez, Félix

Tutor/a: Ferri Ramírez, César

Cotutor/a: Hernández Orallo, José

CURSO ACADÉMICO: 2023/2024

Resum

En l'era digital, a mesura que augmenta la necessitat de codi, es produeix un canvi cap a plataformes sense codi que democratitzen el desenvolupament de programari en permetre als usuaris crear aplicacions sense coneixements tradicionals de programació a través d'interfícies intuïtives. La programació per exemples (PBE) amplia aquesta tendència generant codi a partir d'exemples d'entrada-eixida, simplificant la programació en inferir la lògica subjacent necessària. Aquest enfocament, arrelat en la Programació Inductiva (IP), millora l'accessibilitat i l'automatització de la programació derivant regles generals a partir d'exemples. Els grans models del llenguatge (LLM) han demostrat les seues capacitats generals en la generació de textos per a una àmplia gamma de tasques. En particular, aquesta tecnologia ha demostrat potents capacitats per a generar codi a partir d'exemples de llenguatge natural. No obstant això, la investigació s'ha centrat predominantment en la generació de codi a partir de descripcions en llenguatge natural, prestant relativament poca atenció a altres dominis, com la programació per exemples. Això suposa una gran oportunitat per a avaluar l'eficàcia dels LLMs en PBE. Aquest estudi examina la capacitat dels LLM per a generar codi a partir d'un conjunt de dades curades de tasques típiques i atípiques en Python i Haskell, amb l'objectiu d'avaluar el potencial i les limitacions de l'ús de LLM per a avançar en PBE.

Paraules clau: PBE, LLM, grans models del llenguatge, generació de codi

Resumen

En la era digital, a medida que aumenta la necesidad de código, se produce un cambio hacia plataformas sin código que democratizan el desarrollo de software al permitir a los usuarios crear aplicaciones sin conocimientos tradicionales de programación a través de interfaces intuitivas. La programación por ejemplos (PBE) amplía esta tendencia generando código a partir de ejemplos de entrada-salida, simplificando la programación al inferir la lógica subyacente necesaria. Este enfoque, arraigado en la Programación Inductiva (IP), mejora la accesibilidad y la automatización de la programación derivando reglas generales a partir de ejemplos. Los grandes modelos del lenguaje (LLM) han demostrado sus capacidades generales en la generación de textos para una amplia gama de tareas. En particular, esta tecnología ha demostrado potentes capacidades para generar código a partir de ejemplos de lenguaje natural. Sin embargo, la investigación se ha centrado predominantemente en la generación de código a partir de descripciones en lenguaje natural, prestando relativamente poca atención a otros dominios, como la programacion por ejemplos. Esto supone una gran oportunidad para evaluar la eficacia de los LLMs en PBE. Este estudio examina la capacidad de los LLM para generar código a partir de un conjunto de datos curados de tareas típicas y atípicas en Python y Haskell, con el objetivo de evaluar el potencial y las limitaciones del uso de LLM para avanzar en PBE.

Palabras clave: PBE, LLM, grandes modelos del lenguaje, generación de código

Abstract

In the digital age, as the need for code increases, there is a shift towards no-code platforms that democratise software development by enabling users to build applications without traditional programming skills through intuitive interfaces. Programming-by-Examples (PBE) extends this trend by generating code from input-output examples, simplifying programming by inferring the underlying logic required. This approach, rooted in Inductive Programming (IP), improves accessibility and automation of programming by deriving general rules from examples. Large Language Models (LLMs) have demonstrated general capabilities in text generation for a wide range of tasks. In particular, this technology has demonstrated powerful capabilities for generating code from natural language examples. However, research has predominantly concentrated on code generation from natural language descriptions, with relatively little attention paid to other domains, such as PBE. This presents a significant opportunity to evaluate the effectiveness of LLMs in PBE. This study examines the ability of LLMs to generate code from a curated dataset of typical and atypical tasks in Python and Haskell, with the aim of assessing the potential and limitations of using LLMs to advance PBE.

Key words: PBE, LLM, large language models, code generation

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

In the digital age, the role of code has become more critical than ever, driving innovation, automation, and the functionality of a vast array of applications and systems. As the complexity of coding increases, there has been a notable shift towards no-code platforms, which facilitate the democratisation of software creation by enabling users to build applications through intuitive interfaces without the necessity of traditional programming skills. These tools facilitate the creation of websites, management of data, and automation of processes by enabling users to interact with visual elements and templates, thereby reducing the barrier to entry for software development.

An intriguing extension of this trend is Programming-by-Examples (PBE), a paradigm where code is generated from a few provided examples of desired input-output pairs. This approach simplifies the programming process by enabling the system to infer the underlying logic required to transform inputs into outputs. One of the most prominent applications of PBE is seen in Microsoft Excels FlashFill, which allows users to manipulate data effortlessly by recognising patterns from a handful of examples and generating the corresponding code automatically.

The fundamental principle underlying PBE is inductive programming (IP), a branch of artificial intelligence that is concerned with the generation of programs from incomplete specifications, typically examples. The objective of IP is to model and solve problems by deducing the general rules that connect input and output data. This technique has demonstrated considerable potential in enhancing the accessibility and automation of programming.

However, the landscape of code generation is undergoing a rapid transformation through Large Language Models (LLMs). These models have transformed the field of text and code generation by genearting responses and scripts that are indistinguishable from those produced by humans. The question that arises is whether LLMs can match or even surpass the performance of traditional IP techniques in generating code from examples. Can the capabilities of LLMs, which have demonstrated exceptional performance in understanding and generating natural language, be effectively extended to the domain of programming by examples?

This study aims to address these questions by evaluating the performance of LLMs in the context of PBE. We examine the ability of LLMs to generate accurate code from a curated dataset of tasks that range from typical programming challenges to specially designed, atypical problems. By comparing the results across different programming languages such as Python and Haskell, we seek to understand the potential and limitations of LLMs in this domain.

Through this analysis, we hope to provide insights into the intersection of LLMs and IP.

1.1 Motivation

According to a study conducted by the OECD (Organization for Economics Comparation and Development) [54] between 2011 and 2015 on digital literacy in the workplace, participants were assessed at three levels. Level 1 included basic tasks such as deleting an email, while level 3 addressed "complex" tasks such as extracting the percentage of emails containing a keyword. The results found that in the US, 5 percent of the population could perform level 3 tasks, another 26 percent could perform level 2 tasks, and the remaining 69 percent were below level 2, with 26 percent of these people having no knowledge of using a computer at all.

Although, today, we can assume that these figures have increased, and that the number of people using a computer in the workplace has increased, there is still a considerable gap in the number of highly computer literate individuals. In addition, non-technical users are faced with repetitive tasks that could easily be automated by more advanced users.

Given this gap, there is a boom in no-code platforms. These platforms allow non-programmer users the possibility to build and manage programs solely through an interface, so that the user does not require programming knowledge to, for example, build a website or an application.

A particular case is programming by examples (PBE), a programming paradigm that synthesises the specifications of a program given user-defined input and output examples. In other words, the goal of programming by examples is, for example, given as input a list of first and last names and as output only the name of the first two positions in the list, the system must recognise that the user wants to extract the first position of each entry and generates the code that models that relationship, so that the user can extend it to the next items in the list.

Inductive programming (IP) is an area of research that brings together the areas of artificial intelligence and example programming to create systems capable of generating programs from incomplete specifications, such as example input/output pairs. The most prominent contribution in this field has been FlashFill [21], a technology presented by Microsoft and applied in Excel that allows combining, transforming and extracting data with just a few examples. With this tool, users without programming knowledge can now work with large volumes of data.

On the other hand, the field of natural language processing (NLP) with the advent of so-called large language models (LLMs) has undergone a revolution. These language models have demonstrated human-equivalent capabilities in text generation. Even more surprisingly, they have been able to generate code from abstract descriptions.

In light of these capabilities, an investigation into the potential of large language models (LLMs) for programming by example could prove to be a fruitful avenue of inquiry. Our objective in this work is not only to assess the viability of this technology in this context but also to gain insight into the reasoning capabilities of LLMs.

1.2 Objectives

This paper aims at a detailed evaluation of large language models for programming by examples. For this purpose, the following aspects should be covered:

- **Literature Review and Benchmark Creation**

  - *Conduct a Comprehensive Literature Review*:
    * Perform an exhaustive search of existing literature on PBE and LLMs.
    * Utilise academic conferences, and journals to gather relevant studies and reviews to provide a global picture of the field of PBE.
    * Identify current methodologies, datasets, benchmarks.
  - *Develop a Diverse Benchmark for Evaluation*:
    * Design a benchmark that includes tasks of varying complexity to reflect a wide range of scenarios.
    * Assess the impact of the number of examples on task definition and solution accuracy.
    * Ensure that tasks can be fully and unambiguously defined with a minimal set of examples.

- **Monitoring and Framework Establishment**
  - *Ongoing Monitoring of Advancements in LLMs*:
    * Keep abreast of new developments and publications related to LLMs and their applications in code generation.
    * Evaluate how recent advanced might influence ongoing research and methodology in PBE.
  - *Establish Model and Language Foundations*:
    * Identify and document the LLMs to be used, considering their strengths and weaknesses in relation to PBE.
    * Select programming languages for evaluation based on their prevalence in inductive programming systems, LLMs and other relevant metrics.
    * Justify the choice of models and languages in terms of their suitability for PBE tasks.

- **Detailed Evaluation and Analysis**
  - *Perform Comprehensive Evaluation*:
    * Develop a detailed evaluation framework to assess the performance of LLMs on the benchmark tasks.
    * Compare LLM performance across different tasks and programming languages to identify strengths and weaknesses.
  - *Conduct In-Depth Analysis of Results*:
    * Analyse the results to draw meaningful conclusions about the capabilities and limitations of LLMs in PBE.
    * Provide insights into areas for improvement and potential future directions for research.

1.3 Structure

The rest of the document is structured as follows:

- Chapter 2, State of the Art: A review of existing methodologies in program synthesis is conducted, with particular emphasis on inductive programming and the evolution of neural language models for code generation.

- Chapter 3, Problem Analysis: An analysis of the challenges of LLMs, considering technological choices for PBE, security, energy consumption, and legal frameworks.
- Chapter 4, Solution Design: Details on the experimental setup and methodologies used for evaluating LLM performance.
- Chapter 5, Final Results: The final evaluation results are presented, along with an insight into the findings.
- Chapter 6, Conclusion: A critical examination of the work presented and further possible extensions.
- Appendix C, Glossary: Provides definitions for technical terms and acronyms used throughout the document to aid understanding.

CHAPTER 2
State of the Art

Program synthesis is a research field that encompasses a wide variety of different technologies, implementations and applications. It aims at building programs based on user intentionality, where this intentionality can be formalised in a natural language description, constraints, a schema or input/output example pairs.

This is why we can classify program synthesis into three major problems [28] (Figure 2.1 shows a visual representation of this segregation):

- Deductive: Specific and structured descriptions. These systems include, for example, the generation of UML-based programs.
- Inductive: Using inductive reasoning to extract a generalisation about an incomplete problem, such as a subset of problem instances.
- Natural Language: The user's intentionality is described in natural language sentences.

In the first section, 2.1, we will focus on the field of induction-based program synthesis, trying to give a global and complete vision. Subsequently, 2.2 details the evolution of natural language generation technologies, focusing on the generation of code based on these systems.

2.1 Inductive Programming

Inductive programming (IP), also known as automatic programming, programming by examples (PBE) or inductive program synthesis (IPS), is a field that lies at the intersection between machine learning, programming and algorithms. Its evolution is as diverse as its many names and it is only in recent years that efforts have been made to unify the multiple lines of research that belong to this field. This is why Kitzelmann [35] and Flener et al. [19] encompass the different systems in the following problem-solving approaches:

- Analytical approach: Candidate programmes are constructed based on the input-output characteristics of the examples.
- Search-based approach: Hypotheses about candidate programmes are generated independently from the given specifications. Programme candidates are evaluated according to the given specifications and one or more of the best evaluated candidates are further developed.

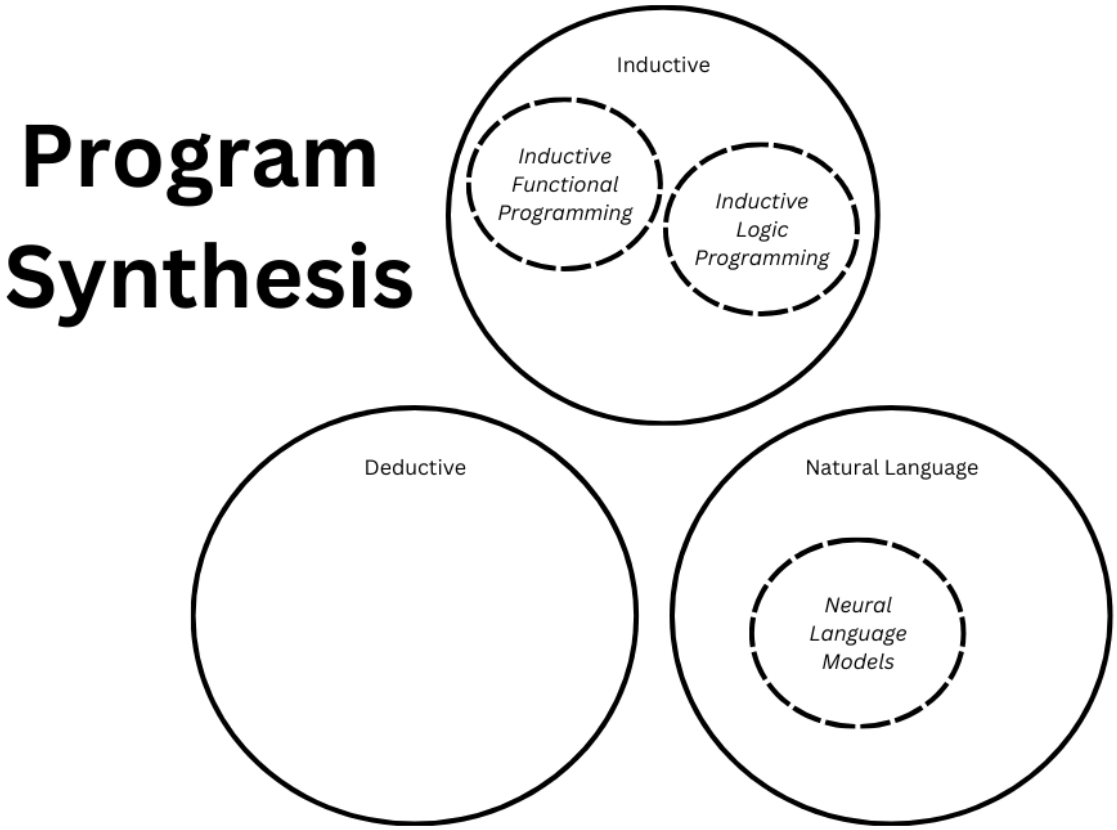

Figure 2.1: Venn diagram showing the sub-fields contained in the field of program synthesis.

As mentioned in a survey by Gulwani et al. [22], we can divide the field of IP into two main areas of study, inductive functional programming and inductive logic programming. In the following sections we will give an overview of both.

2.1.1. Inductive Functional Programming

Summers, in 1977, with his THESYS [75] system, presents the first system in the field of IP. In it, he analytically induced LISP functions of input/output examples. In particular, he saw that by using a few basic primitives and a fixed program grammar, a restricted class of recursive LISP programs satisfying the set of examples could be induced. Later, Biermann's method (1978) [5] and Igor1 (2006) [36] extended on Summers' work. Biermann's approach was to use fragments to speed up the exhaustive enumeration of regular LISP programs, while Igor 1 presents a more general system than Summers', where the scope of synthesised programs is larger thanks to the addition of a technique that allows the introduction of new auxiliary functions. It is important to note that all the methods described above suffer from strong restrictions in the general structure of the problem, such as the use of a small number of primitives and that the examples should be in order Also, synthesis is only limited to structural problems, where only the structure of the arguments matters, but not their content, as for example in inverting lists.

Subsequent systems attempt to solve the above problems by shifting the focus to a search problem in the solution space and start using more expressive languages. It extend the possible tasks to be modelled, however this increased the space of possible programs.

MagicHaskeller (2005) [33] [32], presents a function-forming system in Haskell, which uses an exhaustive search in the problem space. In order to bound this space, it uses only

higher-order functions and strong typing. Igor2 (2009) [34] [25] also uses Haskell, this system, in order to narrow down the search for programs, uses the analytical technique mentioned in the other systems, thus achieving an example-driven search.

In the last decade, there has been a trend towards more robust systems, capable of modelling more tasks and being able to define tasks with fewer examples. Another notable change has been in the use of programming language, where more recent systems have moved towards using the functional language ML (Meta Language) and its subsequent extensions, such as OCaml. ML is a more minimalist and efficient language compared to Haskell, given a simpler syntax and type system, it facilitates the design of inductive functional programming systems.

L2 o λ^2 (2015) [18] is perhaps one of the most noteworthy systems implemented in ML. This system generalises examples into hypotheses about the structure of the target program, based on a fixed set of primitives. It then employs deductive reasoning axioms to derive examples from the missing subexpressions within each hypothesis. The system uses a combination of enumeration and deduction to search for hypotheses that satisfy the given examples. This process is similar in Myth (2015) [56] which also makes use of ML, but looks directly for recursive functions (in contrast, λ^2 focuses on recursive data structures). SMyth (2020) [42] extends the Myth system and switches its implementation to the OCaml language. Burst (2022) [46] is also presented in OCaml as a more versatile system, being able to handle logic specifications as well.

Finally, the most recent system to date is SyRup (2023) [82]. This system makes use of a Domain Specific Language (DSL) which they call tiniyML, as it is a reduced version of ML. Furthermore, it proposes a trace-driven approach to address challenges related to ambiguity and generalisation in program synthesis.

2.1.2. Inductive Logic Programming

Compared to IFP, ILP is an area of research with more development and attention. In its years of evolution, it has had numerous reviews, which are very useful to get a picture of its progress at each stage of its development. The first of these reviews was presented in 1997 [66]. In this article, they introduce the foundations of the psychology that helped lay the groundwork of inductive logic programming. Subsequently, Muggleton [49] in 1999, Page et al. [57] in 2003, Zhang et al. [83] in 2022 and Cropper and Dumančić [12] in 2023, provide new approaches, possible improvements and limitations of the systems at that time. We can attribute some of the attention attracted to the field to these reviews, as they facilitate the entry of new researchers into the area.

The first formalisation of induction using logic is given by Plotkin [60] [58] [59] in 1970. In it, he proposes a generalisation based on positive examples, defined as base clauses. Specifically, to determine the minimum generalisation of these base clauses in relation to the background knowledge, composed of base literals. Plotkin's contributions continue to be influential in the field. At the end of the decade, Steven Vere continued Plotkin's work. During this time, Steven Vere introduces CONFUCIUS [79], a system that uses first-order logic for the representation of clauses.

In the early 1980s, the MARVIN system [67], introduced by Sammut, evolved from CONFUCIUS, incorporating both generalisation and specialisation. At the same time, Ehud Shapiro [71] introduced a framework for model inference and developed a top-down algorithm for deriving complete axioms from enumerated examples. Shapiro's work, influenced by inductive inference, introduced key concepts such as the Backtracing algorithm for identifying false clauses and a refinement operator for the specialisation of theories. He implemented these ideas in his MIs [70], system, initially focusing on Horn

clauses and then integrating them into his PhD thesis as part of a program debugging tool.

The late 1980s saw a surge in machine learning research within propositional logic frameworks, probably driven by the popularity of logic programming and Prolog. Wray Buntine [7] extended subsumption to address its limitations, while Stephen Muggleton developed DUCE [48] to generalise propositional variables. Muggleton and Buntine [51] introduced inverse resolution as a means of generalisation and predicate invention, which ignited further research interest. Early alternatives to inverse resolution emerged in systems such as FOIL [61], LINUS [39] and GOLEM [53] . FOIL employed a top-down refinement operator guided by information-based search heuristics, LINUS transformed ILP problems into a value-attribute representation for learning, and GOLEM, revisited Plotkin's concepts with efficiency improvements.

In 1990, Muggleton coined the term "inductive logic programming" and together with Pavel Brazdil, organised the first international workshop on inductive logic programming the following year. These workshops have since become an annual event, consolidating ILP as a research domain. Numerous systems have been developed and successfully applied in various fields. Recent theoretical advances include the invention of predicates, data mining, real number handling and adaptation to noisy examples, reflecting the expanding scope of the field.

Despite developments in other fields, there has continued to be an advance in example-based programming systems. Among these, perhaps the most influential system is Progol [50] introduced in 1995. It uses an A* heuristic for a top-down search (from general to specific), where examples are used to guide this search. In 2001, Aleph [73] a system based on Prolog, restricts the search space by initially constructing a specific clause based on the examples, then in the search it will ignoring any clause that is not more general than this specific clause. This results in a very efficient system with very good performance. Thanks to its simple implementation in a single Prolog document, it has proved to be a very popular system.

In the last decade, more powerful, yet simpler and more efficient systems have become available. In 2011, Aspal [11], a meta-level system that uses an ASP (answer set programming) solver for ILP problem solving was introduced. Although the implementation may be complex, it is actually one of the simplest systems, it uses statements to construct every clause that can belong to the hypothesis, adds a flag to each of this clauses and thus formulates an ASP problem about which flags should be set. Next, in 2015, Metagol [52] uses a Prolog meta-interpreter to build a test on the set of examples and extract a program from that test. Its implementation takes up just 100 lines of code in Prolog and is guaranteed to find the smallest program.

Popper [13] introduced in 2020, is the most recent ILP system to date. Popper brings together ASP and Prolog to propose an approach called *learning from mistakes*. Specifically, he proposes to divide learning into three parts: *generate*, where the system generates a hypothesis that satisfies a set of constraints; *test*, where the hypothesis is tested with the training examples; *constrain*, if the hypothesis fails, the system prunes the hypothesis space by adding new constraints. According to the authors evaluations, it is shown to be a powerful system in performance and with very low computation times. Still, the system contains limitations that are left open for future systems, such as improving the search space or the constraints.

2.2 Neural Language Models

The resurgence of neural networks, particularly deep learning, brought about a paradigm shift in NLP. Previous methods used rule based approaches for generating text, however this was not efficient, as capturing all use cases is intractable (curse of dimensionality), this specially applies to code.

The first mention of a neural language model, was in 2003, by Yoshua Bengio [4]. In his work he presents a system that fight this curse of dimensionality by using a feed-forward network to learn both the distributed representation for each word and the probability function of a word sequence. Even though its approach was interesting, and performance was good on simpler tasks, it failed in more complex ones. Mayor problems where that it could not handle variable-length context, did not capture long-term dependencies and suffered from training instabilities.

Recurrent Neural Networks (RNNs) [15] and LSTM [23] networks emerged as popular choices for modelling sequential data, including natural language and code. RNN [45] and LSTMs [20], as language models, were pivotal in addressing some of the shortcomings of earlier models, particularly in handling variable-length inputs and capturing longer-term dependencies.

Around 2014, researchers explored sequence-to-sequence (Seq2Seq) [76] models using RNNs, this models where ideal for tasks that involve converting sequences from one domain to another, where the lengths of the input and output sequences can vary. Early models like Code2seq [1], leveraged this approach for code generation. However they still faced challenges in managing long-range dependencies and generating accurate, contextually relevant code snippets.

The introduction of attention mechanisms marked a significant advancement in NLP. These mechanisms enabled models to focus on relevant parts of the input sequence more effectively during the decoding process. Bahdanau et al. [3] introduced attention mechanisms in Seq2Seq models, allowing for more effective alignment between input and output sequences. Attention mechanisms significantly improved the performance of code generation models by capturing dependencies more accurately, especially in long sequences.

The Transformer architecture, introduced in 2017 by Vaswani et al. [78], marked a breakthrough in NLP by replacing recurrence with self-attention mechanisms. Transformers enabled parallelisation of computation, making training faster and more efficient. Models based on the Transformer architecture, such as GPT [62] and BERT [14] , achieved remarkable results across a wide range of NLP tasks, including code generation.

The Transformer architecture also facilitated the development of larger pre-trained models, which could be fine-tuned on specific code generation tasks. This fine-tuning process allowed models to adapt to specific coding conventions, syntax, and semantics, leading to substantial improvements in code generation performance. Models like CodeBERT [17], which integrate representations of both natural language and code, exemplify the potential of fine-tuning pre-trained models for code generation.

The development of large language models has demonstrated their ability to generalise across different tasks, and these systems are currently being used as general-purpose systems capable of performing a wide range of tasks. "How General-Purpose Is a Language Model? Usefulness and Safety with Human Prompters in the Wild" [8] looks at the practical implications of more general language models, in real-world scenarios. This study examines the aforementioned general-purpose capabilities of these models by analysing their interaction with non-expert users. Despite the performance showed in

benchmarks, the research highlights significant gaps between theoretical capabilities and actual utility. The paper introduces a framework to evaluate the effectiveness and safety of language models when used directly by humans, focusing on the costs associated with prompt creation and output interpretation. Through empirical studies, the authors reveal that while models like GPT-3 can handle a diverse array of tasks, they often struggle with understanding nuanced human commands, leading to increased human effort.

Data wrangling and PBE, in contrast use more structured and well defined tasks, this avoids the variability present in human prompts. Both tasks share common goals as they aim to automate data manipulation and try taking complex operations more accessible to non-experts. However, they differ significantly in their scope and methodology. Data wrangling focuses specifically on cleaning and manipulating data. In contrast, PBE has a broader application in programming tasks, inferring program logic from user-provided input-output examples to generate executable code. While data wrangling produces transformed datasets, PBE creates actual programs or scripts.

"Can language models automate data wrangling?" [29] evaluates the problem of data wrangling for LLMs. In there findings, they show that LLMs can approximate the performance of specialised data wrangling tools. However, when considering these systems for real-world applications, performance should not be the only consideration. The paper argues that the main reason why it is not more widely used is because of limitations in cost, infrastructure, privacy issues or lack of training data.

In the period preceding the submission of this work, a paper addressing a similar topic was made available as a preprint on Arxiv. The paper, entitled "Is Programming by Example Solved by LLMs?" [41], similar to this work, aims at evaluating large language models for the task of PBE. In order to investigate the capabilities of large language models in the context of programming by example, the authors of the paper have constructed a dataset comprising a diverse array of tasks of preexisting tasks. This resulted in the creation of a dataset including numerical vectors, string manipulation macros, and graphics programs. Two different size models where fine-tuned for this specific task. The model was evaluated by varying the number of potential solutions that it could generate. The authors consider a task as resolved when one of this generated solutions does solve the specific task. The findings demonstrate that, as expected, LLMs demonstrate high performance when evaluated with the training data. However, the performance was found to be suboptimal when evaluated on holdout data. Furthermore, the findings indicate that the number of generated candidates should be in the order of 100 to obtain optimal results.

CHAPTER 3
Problem Analysis

This chapter, provides a comprehensive examination of the critical challenges surrounding the evaluation of LLMs for code generation from examples, offering insights into technological choices. In addition, we provide an overview of the challenges and concerns associated with this technology. More precisely, the chapter contains the following sections:

- Technological choices: This section looks at the various technical choices that need to be made for evaluating LLMs for programming by examples.
- Security: This section examines the security implications of using LLMs, in particular the phenomenon of "hallucination".
- Energy consumption: The environmental impact of LLMs is a major concern due to their high energy consumption during both training and deployment.
- Legal framework: The legal and ethical considerations of LLMs are becoming increasingly important as their use becomes more widespread. This section provides an overview of the current regulatory environment and the challenges posed.

3.1 Technological Choices

In order to conduct a comprehensive evaluation of large language models, it is necessary to consider a number of factors. It is of the utmost importance to delineate each technological decision that will be required. The following section identifies and analyses each of the technical choices in detail, providing insight into the factors that should be taken into consideration before deciding into a final configuration.

3.1.1. Programming Language

State-of-the-art models are trained on trillions of publicly available data found on the internet. Research has demonstrated that as these models increase in size, their general capabilities also tend to increase [31]. With this increase in size, the necessity arises for more data to train the models with [24]. In the domain of programming, as models increased in size and more training data was represented by code, large language models were able to also learn code generation. Currently, systems are able to generate quality code in any desired programming language.

# Ranking	Programming Language	Percentage (YoY Change)
1	Python	16.925% (-0.284%)
2	Java	11.708% (+0.393%)
3	Go	10.262% (-0.162%)
4	JavaScript	9.859% (+0.306%)
5	C++	9.459% (-0.624%)

Figure 3.1: Top 5 programming languages in GitHub.

However testing the performance of every existing programming language is impractical, so careful selection criteria should be used to choose one or more languages for evaluation.

Programming languages can be grouped into paradigms, each offering distinct principles and strategies for organising and writing code. In search of a programming language suitable for the task of programming by examples, certain aspects should be taken into consideration. Desired languages should facilitate the implementation of functions, making it easy to model and test them. It should also be convenient programming languages with a high-level of abstraction. The chosen programming language should be relatively simple, as the code implementation should not be a bottleneck. The objective is to capture and model the relationship between the input and output of the provided examples, rather than to asses the capacity of large language models to handle memory management or other non-relevant tasks.

It is also important to consider the representation of the programming language in the training data. As previously stated, these models are trained on a vast quantity of data. However, it is not expected that every programming language will be represented equally. As the specific training data for most LLMs is not publicly available, certain approximations can be made to account for this. The popularity of the programming language or the quantity of code available about it could be useful metrics for making an informed decision about which programming languages are more likely to be represented in the models data, thus resulting in the best performance.

Given that GitHub [1] is the greatest resource for code, it serves as a crucial resource for LLMs. When analysing language distribution across repositories, as seen in Figure 3.1 [2] , Python emerges as the most prevalent language, constituting nearly 17% of the platform's code-base. Consequently, Python's popularity in LLMs training data suggests it may yield optimal performance in evaluations.

On the contrary, IP systems typically operate within a range of specific programming languages. These systems commonly rely on languages such as Haskell, Prolog or other Domain-Specific Languages (DSL). To ensure a fair and meaningful comparison between IP systems and LLMs, it is imperative to standardise the programming language selection to encompass Turing-complete, general-purpose languages.

Haskell emerges as a particularly suitable choice due to its adoption in popular systems like Igor2 and MagicHaskeller. Its expressive and functional nature aligns well with

[1] https://github.com/
[2] https://madnight.github.io/githut/#/pull_requests/2024/1

the principles of inductive programming, facilitating the creation of elegant and concise solutions to complex problems.

Haskell is also a compelling choice for code generation with LLMs. Due to its modest representation on GitHub [3] , where it ranks at position 27 among programming languages, with a mere 0.132%. Its inclusion offers valuable insights in our evaluations. By examining potential correlations between task performance and language representation in LLM training data, we gain a deeper understanding of how linguistic and computational principles relate.

In conclusion, the selection of programming languages for evaluating LLMs performance in PBE requires careful consideration of multiple factors. The selected languages should facilitate function the implementation of functions, offer high-level abstraction, and be relatively simple to use. Additionally, the representation of these languages in the training data of LLMs is a crucial consideration. Python is identified as the best candidate due to its prevalence on platforms like GitHub, which may result in optimal performance in LLMs. Conversely, while Haskell is expected to be less represented in typical LLM training data, is widely used in inductive programming systems and offers a functional paradigm well-suited to the task. By including both Python and Haskell in the evaluation, we can gain insights into how language popularity and paradigm affect performance across different systems.

3.1.2. Large Language Models

Large language models are computational models based on artificial neural networks. In more precise terms, they can be categorised as generative models. These models learn statistical relationships from vast amounts of data and are then able to generate this data through the process of inference. In recent years, as both computing power and the quantity of data have tend to increase, these models have been able to achieve general-purpose language generation. As a consequence of this capabilities, a catalogue of different models has developed rapidly. This makes it challenging to select the most appropriate model(s) for evaluation. One might assume that selecting the best performance model could a straightforward process. However,given the generality of these models, no single metric can confidently indicate that a model performs better in all use cases. Therefore, it is necessary to group models and analyse the potential advantages and disadvantages of each.

On the one hand, we can identify models that have been designed with the ability to perform a wide range of tasks. These models are not limited to a specific task and are therefore capable of functioning as general assistants. This means that they can generate code, write a poem or an essay about the First Wold War. Research has demonstrated that this generality enhances the overall performance of the models [80], as evidenced by the fact that a models ability to write poems can enhance its ability to write essays. This is consistent with our understanding of human intelligence, which suggests that acquiring new skills will be beneficial in other domains. Still, this approach has limitations, as it is most effective with the largest and most computationally intensive models.

On the other hand, models designed for a specific task represent an interesting approach. These models are trained to perform in a specific task, such as code generation. This specificity will result in sub-optimal performance in any task that the model has not been trained for. In the aforementioned example, a model trained for code generation will exhibit sub-optimal performance when tasked with the generation of poems or essays. The main advantage of this specific type of models is that it is possible to achieve

[3] https://madnight.github.io/githut/#/pull_requests/2024/1

high levels of task performance with much smaller and less computationally intensive models. Depending on the intended use case, this approach can be beneficial as it enables edge computing, thus reducing latency and friction with the user.

Another crucual aspect to be taken into account is the accessibility of the models. The development of these models has been primarily driven by private research laboratories. This has resulted in the presentation of numerous models with licensing, restricting use cases, or even restricting the access of the model through an API (application programming interface). The advantage of closed models is in its easy of use, as they are usually integrated into a computing platform, making them accessible. However, this hard limit makes it so that use cases and modifications are impossible to perform.

In contrast, open-source models present themselves as models with open licensing and accessibility. The aforementioned models do not impose any restrictions on the use cases or modifications that can be performed. Nevertheless, these models are typically less powerful than their closed counterparts.

When the specific model(s) for our use case have been chosen. Another challenge arises in the manner of instructing the model to accurately perform the desired task. These instructions are formally known as prompts. A prompt consists of a natural language text description of the task that we want the model is to perform. The process of prompt engineering involves the creation of a well-defined text queries with the objective of obtaining a precise response. In order to achieve the desired outcome, the following is necessary to take into consideration. First, it should be taken into consideration effective strategies and techniques for well designed prompts [40] [9] [2]. Furthermore, taking this strategies and techniques into consideration, a methodological approach should be employed. This will facilitate the construction of increasingly complex prompts and direct the search towards an optimal solution.

The field of Large Language Models (LLMs) is advancing at an unprecedented rate, with state-of-the-art models being released almost monthly. Among these, OpenAI's Generative Pre-trained Transformer (GPT) series has proven to be particularly influential. These models have had a significant impact on natural language processing (NLP) by generating coherent and contextually relevant text in a range of applications. Although not explicitly designed for programming tasks, GPT models have demonstrated remarkable capabilities in this area.

GPT-4 [55] the latest in the series, has achieved ground-breaking results across a range of benchmarks and leader-boards [4], cementing its reputation as the most powerful and versatile model to date. It excels at a wide range of tasks, including complex reasoning and coding, making it particularly relevant to our goals. Its ability to generate nuanced and contextually appropriate text has set new standards in the field, demonstrating significant advances in areas such as contextual understanding and problem solving.

GPT-3.5 [6], the predecessor to GPT-4, remains an important milestone in the evolution of LLMs. While it may not match the performance metrics of GPT-4, GPT-3.5 serves as a critical benchmark for evaluating the challenges inherent in various NLP tasks. Its strength lies in its balance between performance and computational efficiency, offering an exceptional ratio of output quality to computational resources. This makes GPT-3.5 a valuable tool for applications where computational constraints are an issue.

Within our current scope, we recognise the potential value of exploring additional models, including open source alternatives or those specifically tailored for programming. However, for the purposes of this work, we have chosen to focus exclusively on GPT-4 and GPT-3.5. This decision allows for a focused analysis of their capabilities and

[4] https://arena.lmsys.org/

sets the stage for possible future evaluations of other models that may offer unique advantages or specialisations.

In conclusion, the selection of an appropriate LLM for a specific use case necessitates a balancing act between general-purpose capabilities, task-specific efficiency, and accessibility constraints. Once a model has been selected, it is essential to employ effective prompt engineering in order to fully exploit its capabilities. This approach, ensures that the chosen LLM can be effectively harnessed for the intended application, thereby maximising both performance and practicality. This study focuses on GPT-4 and GPT-3.5, which offer a strategic approach to leveraging their proven capabilities in natural language processing and code generation. This choice allows for a detailed examination of state-of-the-art general-purpose models in programming tasks, while also establishing a benchmark for potential future evaluations of other models.

3.2 Security

Despite the impressive capabilities of large language models, this technology is susceptible to a phenomenon known as "hallucinations." In the context of large language models, hallucinations refer to the generation of text that appears plausible but is factually incorrect or nonsensical [30]. This section will examine the nature of hallucinations, their underlying causes, and their security implications [64].

It is possible to differentiate between intrinsic and extrinsic hallucinations [44]. Intrinsic hallucinations occur when the model alters the content of the input, thereby creating a direct contradiction between the input and the output. To illustrate, if the input is "Paris is the capital of France" and the output is "Paris is the capital of Spain". Here, the output directly contradicts the factual content of the input. Extrinsic hallucinations occur when, based on the input alone, it is not possible to validate the output's validity. For example, if the input is "The company launched a new product" and the output is "The company launched a new product, which immediately became a global bestseller", the claim about becoming a global bestseller cannot be verified from the input alone and may be inaccurate.

The potential for hallucinations in LLMs introduces several security concerns [6]. A significant concern is the risk of misinformation and disinformation. As this technology becomes more widely used by regular users, and as its reliability increases, users trust in the technology will also increase. As a consequence of this trust, users will tend to pay less attention, overestimating the capabilities of the technology. This will result in an increase in the harm caused by hallucinations, as users will begin to consider the output of these models as the ground truth. This overestimation of capabilities can have significant consequences for the spread of misinformation.

Moreover, hallucinations can be exploited for phishing and social engineering attacks. The generation of convincing yet false content by LLMs has the potential to deceive users into divulging sensitive information or undertaking harmful actions. Another significant security concern is the issue of data privacy and confidentiality. The fabrication of information about individuals or organisations through hallucinations can result in breaches of confidentiality, causing reputational damage or violations of privacy.

The phenomenon of hallucinations in LLMs can be attributed to a number of factors. A primary cause is the limitations inherent in the training data. LLMs are trained on extensive datasets sourced from a variety of sources. If the datasets contain inaccuracies, the model may learn and subsequently reproduce these errors. Furthermore, biases and

data gaps can result in incomplete or unfair representation, which may lead to erroneous information being generated by the model when attempting to infer missing details.

It is also the case that prompt ambiguity plays a significant role in the generation of hallucinations. When a model is presented with an ambiguous or poorly constructed prompt, it may attempt to infer meaning and generate a response based on incomplete or unclear input. This often results in hallucinations.

Another factor that can influence the occurrence of hallucinations is overfitting. Overfitting occurs when the model becomes overly reliant on the data it has been trained on, preventing it from generalising to new, unseen data.

The final factor to be considered is the possibility of attacks that attempt to exploit the vulnerabilities of the model. These attacks are known as adversarial attacks. This comprises the introduction of minor alterations to the input data, which manipulate the model to generate erroneous or misleading outputs [72].

Addressing hallucinations in LLMs necessitates the implementation of both technical solutions and social awareness. One essential strategy is improving the quality of training data. Ensuring the accuracy and comprehensiveness of training datasets can serve to mitigate the occurrence of hallucinations. This necessitates the curating of datasets that minimise biases and include verified information.

Systems that incorporate human oversight, whereby human intervention is integrated into the training and evaluation of LLMs, can play a pivotal role in the identification and rectification of hallucinations. This approach combines the efficiency of automated systems with the discernment of human validation.

It is important to educate users about the limitations of LLMs and the potential for hallucinations, as this can help to mitigate over-reliance on these systems. The transparency of the development and deployment of LLMs enables users to critically assess the outputs, thereby promoting a more cautious approach to the information generated by these models.

The occurrence of hallucinations in large language models presents a significant challenge to their safe and reliable use. As these models become increasingly integrated into various sectors, it is of the utmost importance to gain an understanding of and to mitigate hallucinations. This is essential in order to utilise the benefits of these models while minimising the associated risks. Continued research and development efforts are necessary to enhance the accuracy and reliability of LLMs, ensuring their trustworthiness in applications where factual correctness is key.

3.3 Energy Consumption

Despite their potential, these models present significant energy concerns. This section explores the energy implications associated with LLMs, including their environmental impact, factors contributing to their high energy consumption, and potential mitigation strategies.

Deep learning models are inherently computationally intensive, requiring substantial resources for training and deployment. The environmental impact of these models is significant, primarily due to the considerable energy consumption involved in their operation. The training of an LLM is a multi-stage process, involving data preprocessing, model initialisation, and iterative training. Each of these stages contributes to the overall energy footprint.

[5] https://epochai.org/blog/tracking-compute-intensive-ai-models

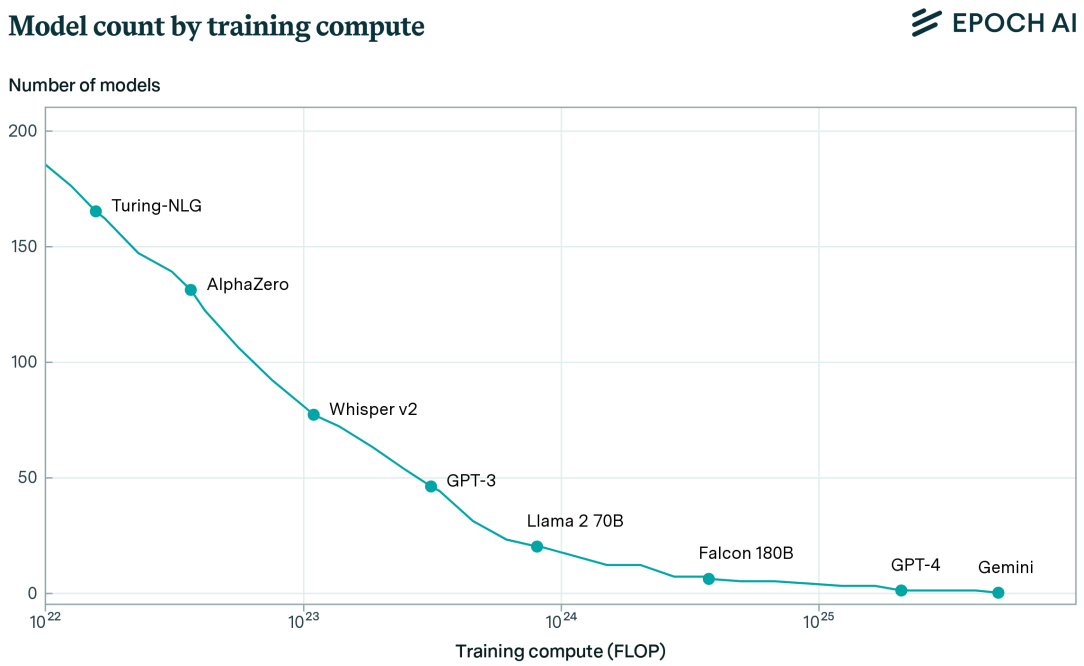

Figure 3.2: Number of models with training compute over different thresholds as of 2024 March 31. Source: EpochAI [5]

The carbon footprint of training large-scale models is particularly concerning. Currently, LLMs are trained on massive data centres, as this training necessitates the utilisation of substantial computational resources. The latest models are estimated to require 10^24 FLOPS (floating point operations per second) of training compute [63] (as shown in Figure 3.2). This compute is associated with 1,300 MWh of electricity, which is equivalent to the annual consumption of approximately 130 houses in a developed country [43].

The architecture employed in LLMs has a significant impact on their energy consumption. Transformer-based models, which underpin many state-of-the-art LLMs, rely on attention mechanisms that are computationally intensive. A significant factor is the sheer size and complexity of these models. Modern LLMs, contain billions of parameters and require substantial computational power. This appears to be an ongoing phenomenon. Research indicates that the rate of growth in computing power for frontier models will be fourfold per year, as evidenced by the observations that the compute required for training frontier models has grown by a factor of 4x per year [69]. This growth in computing power will be accompanied by a parallel increase in energy consumption.

Furthermore, the energy consumption associated with the training of these models is not the sole factor to be considered. The deployment of LLM in real-time applications, such as chatbots or language translation services, continues to consume energy. Although less energy-intensive than training, the continuous inference required for these applications contributes to the overall energy consumption and environmental impact [68].

The energy concerns associated with LLMs can be addressed through a combination of approaches, including improvements in model design, infrastructure efficiency, and operational practices. One effective strategy is the development of more energy-efficient model architectures. A growing number of researchers are tuning their attention to the optimising of algorithms with the aim of reducing computational complexity without

[6] https://epochai.org/blog/training-compute-of-frontier-ai-models-grows-by-4-5x-per-year

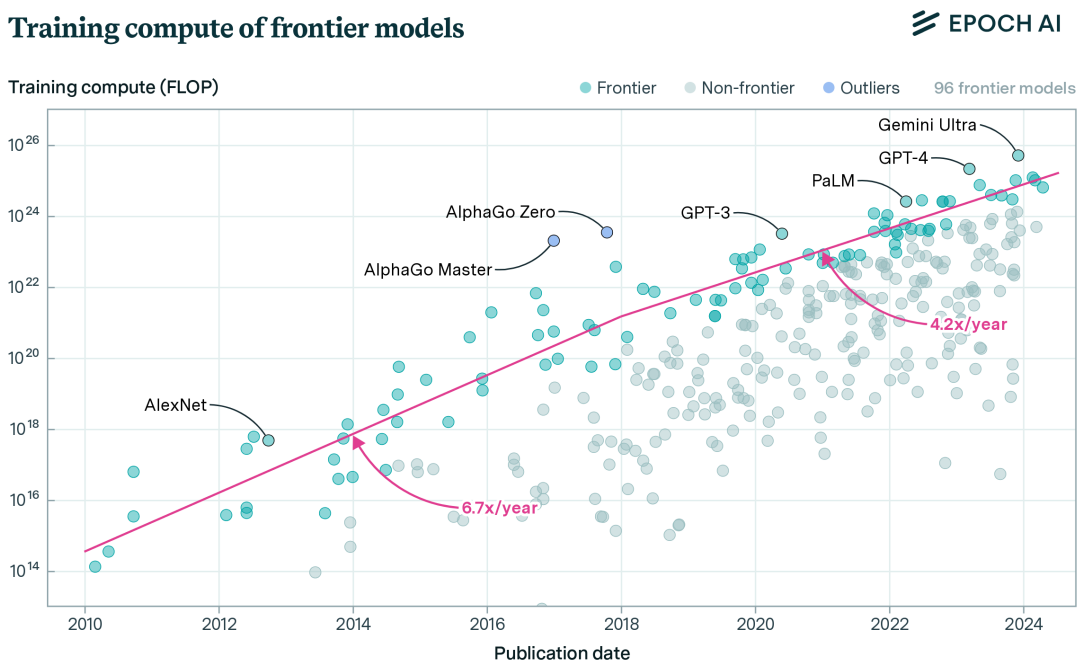

Figure 3.3: Trend of compute in deep learning models. Source: EpochAI [6]

compromising model performance. Techniques such as model pruning, quantisation, and knowledge distillation can significantly decrease the number of parameters and operations required, thus lowering energy consumption [74].

Infrastructure improvements also play a pivotal role in addressing energy concerns. The implementation of data centres optimised for energy efficiency and the utilisation of renewable energy sources, can result in significant reduction in the carbon footprint associated with LLM operations. The implementation of advances in cooling technologies and the utilisation of energy-efficient hardware can also contribute to a reduction in the environmental impact.

The energy concerns associated with Large Language Models present significant challenges to their sustainable deployment and operation. The substantial energy consumption involved in training and deploying these models has notable environmental impacts, primarily due to the high computational demands and reliance on energy-intensive infrastructure. Nevertheless, through the implementation of advanced model architectures, training techniques, and infrastructure efficiency, it is possible to mitigate these impacts. It is therefore imperative that continued research and innovation is perused in this area so environmental sustainability of this technology is ensured, enabling the benefits they offer while their footprint is minimised.

3.4 Legal Framework

The rapid integration of these models into diverse sectors has outpaced the development of a robust legal framework, leading to several legal and ethical controversies. These issues are particularly prominent concerning the training data used by LLMs and the ongoing efforts to regulate their use [38]. This section examines the current legal status of LLMs, controversies associated with their training data, and the evolving regulatory landscape, including initiatives such as the European Union's AI Act [10].

Currently, the legal framework governing LLMs is fragmented and still developing. Most promising laws do not specifically address the unique challenges posed by LLMs, resulting in a complex and often uncertain regulatory environment. In the United States, for instance, no federal legislation explicitly regulates LLMs. Instead, the regulatory impact on LLMs arises indirectly through existing laws related to data protection, intellectual property, and liability. Notable examples include the General Data Protection Regulation (GDPR) in the European Union [16]. These regulations impose requirements on transparency in data collection practices, necessitate user consent for data usage, and provide individuals with rights to access and delete their data. Consequently, these regulations affect how training data for LLMs is collected, stored, and utilised.

The training data employed for LLMs has become the main point of legal and ethical debates. LLMs are typically trained on extensive datasets compiled from various sources on the internet, including publicly available texts, proprietary content and personal data. This practice gives raise to a number of significant legal issues. A further concern is that many training datasets contain copyrighted materials, which raises questions about intellectual property rights. The unauthorised use of such materials has resulted in legal disputes over intellectual property rights, with several lawsuits being filed against developers for allegedly using copyrighted texts without permission from authors and publishers.

Furthermore, the computational capabilities of the models have also been a topic of concern within the legislative community. In light of the current trend, it is anticipated that an increase in computational power will result in enhanced model performance. This enhanced performance is accompanied by an augmented impact on society. In order to be able to control the influence of these models, some initiatives have been proposed to segregate models by the computational power. This would involve making a distinction should be made between less powerful systems with potentially less harmful capabilities and the most advanced models, which may have the most harmful capabilities.

In order to address these challenges, a number of regulatory initiatives have been implemented with the aim of establishing a clear and comprehensive legal framework for the field of artificial intelligence. The European Union's AI Act represents one of the most comprehensive regulatory initiatives to date. The European Commission proposed the AI Act in April 2021 with the intention of establishing a legal framework that would classify AI applications according to their respective risk levels. These levels were defined as follows: unacceptable risk, high risk, and low risk. Depending on their applications, LLMs could be categorised as high-risk, which would necessitate the implementation of stricter rules involving transparency, accountability, and safety.

The AI Act identifies several key areas of relevance to LLMs. In order to ensure transparency, the developers of this models are required to provide detailed information about the model's capabilities, limitations, and characteristics of the training data. This includes the disclosure of potential biases and the sources of training data, thereby informing users about the limitations and risks associated with LLM outputs. Risk management requires the implementation of thorough assessments to identify and mitigate potential harms. Theses assessments must include an evaluation of biases in training data and the implementing of safeguards against adverse effects.

In addition to the AI Act, other jurisdictions are investigating the potential for implementing regulatory frameworks. For instance, the United States of America is developing guidelines for the management of AI risks, with a particular focus on the implementation of appropriate safeguards [77]. In order to ensure the safe and responsible use of artificial intelligence, companies that utilise such technology are required to conduct a comprehensive assessment of the implications of their respective AI systems, as well as

to test and monitor them. Japan is also a pioneer in the field of legislation of artificial intelligence [47]. The approach taken is to encourage a human-centric approach to artificial intelligence. This is achieved by following seven principles: (1) Human-centric; (2) Education/literacy; (3) Privacy protection; (4) Ensuring security; (5) Fair competition; (6) Fairness, accountability, and transparency; and (7) Innovation .These efforts represent a growing recognition of the need for comprehensive legal frameworks to govern the deployment and operation of LLMs, addressing the unique challenges they pose.

The legal framework surrounding LLMs is currently characterised by the emergence of new regulations and the ongoing controversy surrounding their implementation. The issues related to the use of training data, including intellectual property rights, privacy, and bias, underscore the need for robust and clear regulations. As regulatory initiatives continue to evolve, collaboration among developers, users, and policymakers will be essential to creating a legal environment that balances innovation with ethical and legal responsibilities. This will ensuring the responsible and equitable deployment of LLMs.

CHAPTER 4
Solution Design

In light of the comprehensive analysis of the challenges surrounding this work addressed in Chapter 3, the following chapter will describe in detail each of the decisions that have been made and the approaches taken to tackle the problem. The chapter has been structured as follows:

- System Architecture: This section outlines the proposed solution for a pipeline to perform the evaluation. This pipeline will ensure a systematic and precise workflow.
- Methodology: This section outlines the methodology employed for the creation of a comprehensive and well-designed dataset, as well as a prompting strategy that, through incremental complexity, achieves satisfactory results.

4.1 System Architecture

In order to implement the code, a robust pipeline was developed with the objective of facilitating a systematic and precise workflow. This is depicted in Figure 4.1, which illustrates a flow chart of the final pipeline. This pipeline is composed of a directed acyclic graph, which features standard and decision nodes. Standard nodes represent the primary flow of operations, while decision nodes provide branching paths based on specific conditions.

The pipeline comprises three main components: dataset extraction, code generation, and code evaluation. Here's a step-by-step breakdown of the workflow:

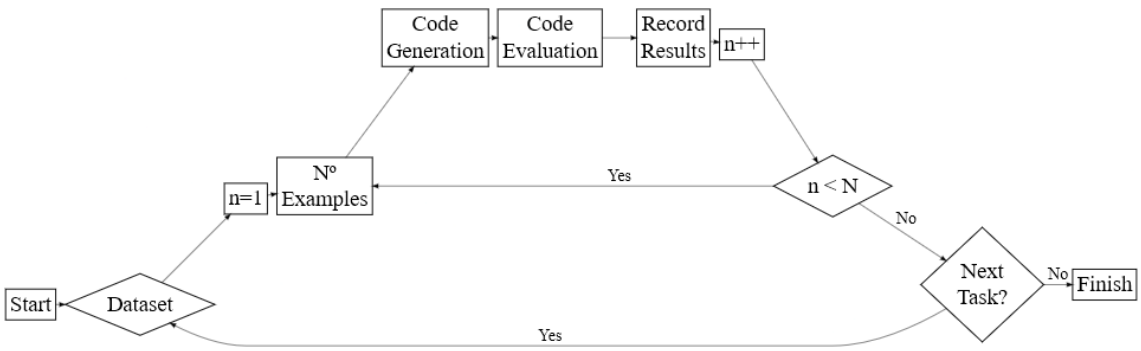

Figure 4.1: Code pipeline used for generation and evaluation of code produced by the large language models.

- Dataset Extraction: The pipeline begins with the conditional node *Dataset*, which selects the first problem from the dataset to be modelled.
- Example Selection: Next, the variable N is initialised to one. The node *N Examples* then extracts the first example associated with the selected problem.
- Code Generation: The *LLM* node generates the corresponding code based on the extracted example. This generated code is subsequently saved for further processing.
- Code Evaluation: The generated code is passed to the *Evaluate Code* node, where it undergoes a series of tests. The *Record Results* node records the outcomes of these evaluations, ensuring that the code is tested across all 20 examples within the problem set.
- Iteration: This process iterates for 9 steps, incrementing N by one in each iteration, until the condition $N \leq 10$ is satisfied. Once N exceeds 10, the pipeline moves to the next problem in the dataset, repeating the entire sequence.

This systematic approach allows us to address each problem in the dataset in a methodical manner, ensuring that every aspect of the code generation and evaluation is thoroughly covered. By rigorously testing the code across multiple examples and configurations, we ensure the reliability and consistency of our results.

The workflow not only standardises the process but also provides a framework for varying conditions and configurations, thereby enhancing the robustness of our solution. By utilising this structured pipeline, we are able to achieve consistent results across different problem scenarios, thereby validating the reliability of our code generation and evaluation process.

4.2 Methodology

4.2.1. Dataset

The main challenge in developing this work is to construct a dataset with a sufficiently diverse range of use cases. It's important not only to include different problems in the dataset, but also to consider the number of examples for each task and the order in which these examples are presented.

The dataset should contain an appropriate number of problems, but too many can obscure the relationships between them. The complexity of each problem is a key factor. Including problems of varying levels of difficulty is beneficial: simple problems help establish a baseline to determine whether basic relationships are being captured, while more complex, novel problems provide a higher benchmark for evaluating advanced models.

A significant challenge is to define problems in a way that categorises them by complexity. Determining whether a problem is 'easy' can be subjective, as complexity often depends on the evaluators perception. It is therefore desirable to establish a criterion for assessing the complexity of problems in the dataset.

Another critical issue is determining the appropriate number of examples needed to define a problem. It is valuable to assess how many examples are needed to capture the underlying relationships, and whether there is a correlation between the number of examples and model performance, or between difficulty and performance.

The order and quality of examples are also crucial considerations. Defining problems solely through examples is challenging, particularly in ensuring that examples are consistent and comprehensive. Examples should be informative and specific to the problem. For instance, the example *0,0=0* could illustrate addition, multiplication, or countless other problems.

An extensive search in inductive programming's literature was made to find a complete and comprehensive set of task that enabled the comparison of both systems. However, most systems we encountered used a limited number of tasks for their evaluations, which was not sufficient for a detailed comparison.

In their paper, Hofmann et al in [26] [27] present an evaluation frameworks for IP. This can be considered as the first unified framework for identifying strengths and weakness of IP systems, as well as one of the first comprehensive comparison of diverse IP systems. Subsequently, the University of Bamberg established a website that serves as a portal for the field of inductive programming[1]. The objective of this web page was to collect all research in the filed in a single repository. The project team presented an extension of Hofmann's dataset and new evaluations for newer models. As this represents the most comprehensive evaluation currently available, it was selected as the baseline for our dataset.

As shown in Figure 4.2, the dataset consists of 32 tasks of varying complexity, ranging from simple tasks as addition or determining even numbers, to more challenging tasks as Ackermann function or Fibonacci. These tasks represent common problems for which specific code implementations are readily available online.

As the dataset did not contain any specific criteria for the number of examples defining each task, we decided to extend each task to 20 input/output examples. We considered this quantity sufficient to cover a wide range of potential use cases, serving as an upper bound for the examples required to accurately capture task behaviour.

In addition to the mentioned dataset, we introduce 12 new tasks, each with their respective 20 input/output examples (shown in Figure 4.3). These additional tasks were designed to introduce nuances and complexities not present in the original set. While the relationships between inputs and outputs were intentionally created to be apparent to humans, they were deliberately made atypical to challenge existing coding paradigms. Consequently, no available code implementations for these tasks exist.

By deliberately crafting tasks with atypical relationships and deliberately avoiding preexisting code implementations, we challenge the systems to generalise effectively and adapt to unconventional patterns. Specially these tasks ensures that LLMs must rely only on their language comprehension and generation capabilities.

In summary, constructing a comprehensive dataset for evaluating inductive programming systems presents several challenges, as ensuring diversity in problem types and complexity to determining the optimal number and quality of examples. This was addressed by adopting a baseline dataset from the University of Bamberg, extending it to include 20 input/output examples per task, and introducing 12 additional atypical tasks. This carefully crafted dataset aims to provide a robust framework for comparing PBE. Additionally it asses the ability to handle both common and unconventional programming tasks. Ultimately, this approach seeks to offer a more nuanced and comprehensive evaluation of the task of PBE.

[1] https://www.inductive-programming.org/repository.html

Problem	Description
ack	Ackermann-Function
add	Addition
append	Append two lists
car	Head of a list
cdr	Tail of a list
drop	Drop the first n elements from a string starting at the front
eq	Equality of numbers
even	Even numbers
evenlist	List of even numbers
evenodd	Even < Odd simultaneously
evenpos	Filter even field positions from a list
evenslist (al	Only even numbers in a list
fib	Fibonacci numbers
geq	Greater than or equal
insert	Insert into string
last	Last element of a list
lasts	Combine the last elements of a list of lists to a new list
length	Lenght of a list
member	Test if element is member of a list
mult	Multiplication
multadd	Multiplication < Addition simultaneously
multfirst	Replace every list element with the head
multlast	Replave every list element with the last element
odd	Odd numbers
oddpos	Filter odd positions from a list
oddslist (all	Only odd numbers in a list
reverse	Reverse list
shiftl	Shift all list elements to the left
shiftr	Shift all list elements to the right
sort	Sort a list
sum	Sum over all list elements
swap	Swap first and last list element
switch	Switch elemnts in a list pair-wise starting up front
take-n	Take the n first elements from a list
weave	Sew two lists together

Figure 4.2: Dataset of programming problems associated with common tasks of varying difficulty, extracted from a public repository of the University of Bamberg.

Problem	Description	Example
add_even_prod	Multiplies all even numbers together and all odd numbers together, then returns the sum of these two sums.	1,2,3,4=1*3+2*4
char_shift	Given a character and a number, returns the character shifted forward in the alphabet by the given number of positions.	d,5=i
comulative_sum	Given a list of numbers, returns a list of the same length where each element is the cumulative sum up to that point.	12,34,56=12,46,102
entangle_strings	Given two strings, returns a new string where each position alternates between characters from the two input strings.	abc,cde=acbdce
length_non_repeting_char	Returns the length of the longest substring without repeating characters from the input string.	abracadabra=brac=4
mult_add	Multiply two numbers together, then adds the second one to the result.	1,2=4
mult_char	Multiply each character in a string by the given number.	ab,2=aabb
nested_factorial_sum	Computes the factorial for each individual digit of a number and then sums them.	312 = 3!+1!+2!=9
power_sum	Sums the list of numbers raised to their own powers.	1,2,3,4=1^1+2^2+3^3+4^4=288
symetrical_pair_prod	Given a list of numbers, calculates the sum of the product of each number with its symmetric counterpart in the list.	2,3,4,5,6=2*6+3*5+4*4
weighted_char_shift	Given a string and a list of numbers of the same length, shifts each character in the string by the corresponding number of positions in the list.	abc,[1,2,3]=bdf
mult_plus_one	Simple multiplication but adding 1 to the output	2,2=5

Figure 4.3: Custom dataset of complex and atypical programming problems.

4.2.2. Prompting

Having reached a point of finality with regard to the configuration and architecture of our systems, we are now in a position to proceed. As previously outlined in Subsection 3.1.2, the final remaining challenge is to define a prompt that can fully leverage the potential of LLMs.

To this end, we initiated the process by defining the system prompt. System prompts serve as a framework for guiding the overall context and constraints within which a model operates. These prompts are of paramount importance for the effective management of how a model interprets user inputs and generates outputs. By defining the model's role in advance, system prompts facilitate consistent, reliable, and contextually appropriate interactions, thereby enhancing the model's effectiveness. This ensures that the outputs are relevant and adhere to the specified guidelines.

With regard to the specific system prompt, we have opted for the following:

> System: You are an inductive programming agent, an autonomous entity tasked with inferring general rules or programs from a given set of input-output examples. It operates by analysing the provided examples to identify patterns or regularities and then formulates a generalised program that can accurately predict outputs for unseen inputs.

We think this is an effective prompt because it provides clear and specific guidance on the model's role and objectives. The prompt ensures a clear role for the model by explicitly defining its function as an "inductive programming agent." This approach is designed to facilitate the process of rule inference, which is of particular significance in the context of our desired tasks. Furthermore, by including input-output examples, we aim to encourage the model to utilise example-based learning. Finally, we attempt to indicate the aspects of the problem that the model should focus on in order to resolve it. In particular, we emphasise the importance of generating general programmes that will be applicable to new, unseen examples.

The previous system prompt was integrated with the subsequent two prompts in order to facilitate the initial evaluation. In light of the fact that code generation will be performed for both Python and Haskell, two prompts were crafted with the objective of aligning with the respective programming language while minimising the discrepancy between them. This led to the formulation of the following prompts:

> Python Prompt
>
> Given a list of examples which structure is f(input) = output, where given to a function f the input values, we get as output the value of this variable.
> Write in Python the function f that models the relationship between inputs and outputs, so given new inputs we can get the corresponding outputs.
> Avoid adding any additional functions as a main or a print.
> Model the following function f in Python:

Haskell Prompt

Given a list of examples which structure is f : input -> output, where given to a function f the input values, we get as output the value of this variable.
Write in Haskell the function f that models the relationship between inputs and outputs, so given new inputs we can get the corresponding outputs.
Avoid adding any additional functions.
Model the following function f in Haskell:

This prompts ensure that the structure of the examples is well defined, that the objective is clearly delineated and it is ensured that the generated code is aligned with the requirements.

In order to provide an empirical evaluation of the quality of our prompt, we present the obtained results in two distinct formats. The first format is a table which contains the average performance of tasks over all examples. The second format is a series of figures which demonstrate the performance of each task when varying the number of examples (contained in the appendix). The figures show a resulting graph for each of the problems contained in our dataset. Each individual graph displays the score for the generated code on the y-axis, with the number of examples defined for each problem displayed on the x-axis. Therefore, if a generated function is well defined and generalises to every example, the resulting score should be 100%. As the system was prompted with one example initially, then two examples, and so on until 10 examples, where a function was generated for each case, the x-axis shows each of the generated configurations. This results in a graph that depicts the performance of the generated function as the number of examples is varied. Furthermore, a red rectangle has been added added, which serves to differentiate and highlights the complex problems, as explained in Subsection 4.2.1.

The aforementioned results were employed to motivate the path to be followed in order to obtain the optimal prompt for the task.

Firstly, an analysis of the performance of the Python-generated functions is presented in Table 4.1 and Figures A.1 and A.3. Revealing that the outcomes of both GPT-3.5 and GPT-4 are largely comparable. This is at odds with our initial expectations, which were that as GPT-4 is a larger and more effective model, it should outperform GPT-3.5. However, the results for both models are not consistent and exhibit considerable fluctuations in performance as the number of examples varies. The observed inconsistency in the results precludes any further analysis aimed at identifying patterns that might be extrapolated from the data.

The ineffectiveness of the Haskell-generated functions is evident in Table 4.1. In this instance, none of the models are able to accurately capture any relationship.

Upon further analysis of the individual generated code, a clear trend emerged for both programming languages. In the case of Python, the fluctuations observed were a consequence of the generated code not fitting accurately to the prompt in some cases, this was caused by the inclusion of additional print statements, additional unnecessary functions or compilation errors. In Haskell, this same issue was more prevalent, with nearly all cases exhibiting compilation errors or the inclusion of additional, superfluous functions.

The outcomes of this results serve to reinforce the initial concerns regarding the prompt. In order to achieve useful results, it is of the utmost importance to align the model as closely as possible with the intended task. This highlights the necessity for the development of more effective prompting strategies.

Table 4.1: Mean Scores over number of examples for GPT-3.5 and GPT-4 on Python and Haskell using a simple prompt.

Function	Python Mean Scores(%)		Haskell Mean Scores(%)	
	GPT-3.5	GPT-4	GPT-3.5	GPT-4
ack	9.0	12.5	0.0	0.0
add	50.0	40.0	0.0	0.0
add_even_prod	11.0	4.5	0.0	0.0
append	60.0	40.0	4.5	4.5
car	86.5	49.5	0.0	0.0
cdr	31.5	57.5	5.0	4.5
char_shift	23.5	80.0	0.0	0.0
cumulative_sum	22.0	51.5	0.5	0.0
drop	54.0	45.0	17.5	35.0
entangle_strings	31.0	60.0	0.0	0.0
eq	89.5	87.5	0.0	0.0
even	28.0	27.5	0.0	0.0
evenlist	44.0	48.5	0.0	0.0
evenpos	9.0	29.0	17.5	20.0
fib	7.5	23.0	0.0	0.0
geq	49.0	59.0	0.0	0.0
insert	43.0	73.0	0.0	0.0
last	80.0	38.0	0.0	0.0
lasts	4.5	24.5	0.0	0.0
length	57.0	57.0	2.5	0.0
length_non_repeating_char	44.5	85.5	0.0	0.0
member	72.5	49.0	0.0	0.0
mult	67.5	90.0	0.0	0.0
mult_add	0.0	0.0	0.0	0.0
mult_char	56.0	35.0	0.5	0.0
mult_plus_one	22.5	0.0	0.0	0.0
multfirst	13.5	50.0	2.5	4.0
multlast	8.0	54.5	2.5	4.5
nested_factorial_sum	11.0	16.0	0.0	0.0
odd	60.5	30.5	0.0	0.0
oddpos	31.0	27.5	24.0	30.0
oddslast	57.0	70.0	0.0	0.0
power_sum	2.0	1.0	0.0	0.0
reverse	90	60	4.5	4.5
shiftl	81	36.5	2.5	3.5
shiftr	29.5	57	3.5	3.0
sort	79.5	28.5	5.0	5.0
sum	62.0	95.0	60.0	95.0
swap	32.5	28.5	3.5	4
switch	50.0	44.5	3.0	4.5
symmetrical_pair_prod	0.0	0.5	0.0	0.5
take	24.5	25.0	31.5	35.0
weave	30.0	76.0	5.0	4.5
weighted_char_shift	24.0	40.5	0.0	0.0

The principal issue with the previous prompt was the generation of code that aligned with the specified requirements. After a comprehensive evaluation, we determined that

the most prudent course of action would be to build upon our previous prompt by incorporating a code template for the model to utilise in the construction of its final solution. This approach is intended to enhance the consistency of our results, particularly in the case of Haskell, where previous outcomes have been less than satisfactory. This led to the formulation of the following prompts:

Python Prompt

User: Given a list of examples which structure is f(input) = output, where given to a function f the input values, we get as output the value of this variable.
Write in Python the function f that models the relationship between inputs and outputs, so given new inputs we can get the corresponding outputs.
Avoid adding any additional functions as a main or a print.
The code should have the following structure:
“‘python
def f(input):
  # your code here
  return output
“‘
Model the following function f in Python:

Haskell Prompt

User: Given a list of examples which structure is f : input -> output. Write in Haskell a function that captures the relationship between the inputs and outputs of the list.
The inputs and outputs should match the type defined in the in the function.
The code should have the following structure,
“‘haskell
f::
main :: IO ()
main = do input <- getLine
let ___ = map read (words input) :: [__]
let result = f ___ putStrLn (show result).
“‘
Model the following function f in Haskell:

Results for our new prompt where extracted, represented in the same layout as before (Table 4.2 and Figures A.5, A.6, A.7 and A.8). A general improvement can be seen over all results. Predominantly, Haskell (Figures A.6 and A.8) shows the most increase in performance. Both GPT-3.5 and GPT-4 do solve the task in specific configurations. Still, results are far from consistent and show undesirable instabilities.

On the other hand, Python (Figures A.5 and A.7) start promising results. Now some difference can be observed between GPT-3.5 and GPT-4. GPT-3.5 results show a minor improvement in stability and performance across tasks. Instead, GPT-4 shows a leap in performance and stability. The results show that GPT-4 consistently is able to solve: add, append, car, char_shift, cumulative_sum, entangle_string, eq, insert, last, length, member, mult, reverse, shiftl, shiftr, sort, sum, switch and weave.

Despite this results, improvements in performance and stability are still desirable.

Table 4.2: Mean Scores over number of examples for GPT-3.5 and GPT-4 on Python and Haskell using a more complex prompt that defines the desired code structure.

Function	Python Mean Scores(%)		Haskell Mean Scores(%)	
	GPT-3.5	GPT-4	GPT-3.5	GPT-4
ack	7.5	12.0	13.68	20.53
add	70.0	100.0	70.0	100.0
add_even_prod	2.0	0.5	7.5	12.0
append	70.0	100.0	78.0	53.5
car	80.0	99.5	29.5	76.0
cdr	56.0	73.0	30.5	32.0
char_shift	23.0	90.5	10.5	31.0
cumulative_sum	13.0	100.0	9.0	100.0
drop	52.0	55.5	47.0	26.5
entangle_strings	0.0	100.0	27.0	53.0
eq	45.0	100.0	32.5	100.0
even	45.0	40.0	40.5	47.0
evenlist	44.0	61.5	37.5	54.5
evenpos	7.0	34.0	39.5	56.0
fib	31.5	4.5	5.5	13.0
geq	41.0	58.0	16.5	59.5
insert	20.0	93.0	39.0	40.0
last	41.5	96.0	53.0	66.5
lasts	2.0	51.0	9.0	67.0
length	66.5	95.0	60.0	90.0
length_non_repeating_char	55.0	75.0	20.5	43.0
member	20.0	97.5	29.0	70.0
mult	34.0	100.0	3.5	66.5
mult_add	0.0	0.0	16.5	19.0
mult_char	21.0	63.0	28.0	35.0
mult_plus_one	8.0	42.5	7.5	18.5
multfirst	24.0	89.0	66.0	60.0
multlast	11.0	40.5	40.5	36.5
nested_factorial_sum	12.5	8.0	13.0	20.0
odd	55.5	70.0	34.0	75.5
oddpos	20.0	38.0	35.5	49.5
oddslist	30.5	65.0	41.0	72.0
power_sum	0.0	9.5	0.5	14.0
reverse	30.0	100.0	81.0	62.0
shiftl	36.0	91.0	41.0	23.0
shiftr	22.0	95.5	53.0	33.5
sort	51.0	83.0	28.5	40.0
sum	83.5	95.0	60.0	95.0
swap	22.0	42.5	39.5	47.5
switch	27.5	90.0	76.0	90.5
symmetrical_pair_prod	0.5	0.5	0.0	5.5
take	18.0	38.0	41.0	32.5
weave	20.0	92.0	45.5	70.0
weighted_char_shift	0.0	29.0	11.0	17.0

A great deal of research is currently being conducted with the objective of developing effective methods for leveraging the power of prompting. A novel approach, Zero-Shot-

Chain-of-Thought [37], has recently been introduced. This research demonstrates that the addition of a simple sentence, "Think step by step," can enhance the performance of zero-shot tasks. In the paper, the author states that *"we show that LLMs are decent zero-shot reasoners by simply adding "Let's think step by step" before each answer. Experimental results demonstrate that our Zero-shot-CoT, using the same single prompt template, significantly outperforms zero-shot LLM performances on diverse benchmarks"*.

Given the performance benefits demonstrated by Zero-Shot-Chain-of-Thought, we proceeded to implement these modifications to our previous prompt. This results in a combination of a description of the task, the desired code template with which to generate the output and the Zero-Shot-Chain-of-Thought strategy:

Python Prompt

User: Given a list of examples which structure is f(input) = output, where given to a function f the input values, we get as output the value of this variable.
Write in Python the function f that models the relationship between inputs and outputs, so given new inputs we can get the corresponding outputs.
Avoid adding any additional functions as a main or a print.
The code should have the following structure:
“‘python
def f(input):
  # your code here
  return output
“‘
Think step by step.
Model the following function f in Python:

Haskell Prompt

User: Given a list of examples which structure is f : input -> output. Write in Haskell a function that captures the relationship between the inputs and outputs of the list.
The inputs and outputs should match the type defined in the in the function.
The code should have the following structure,
“‘haskell
f::
main :: IO ()
main = do input <- getLine
let ___ = map read (words input) :: [__]
let result = f ___ putStrLn (show result).
“‘
Think step by step.
"Model the following function f in Haskell:

The results presented in Table 4.3 and Figures A.9, A.10, A.11 and A.12 are in accordance with the experiments conducted in the Zero-Shot-Chain-of-Thought paper. It can be observed that there has been an overall increase in stability and performance. In more precise terms, in Python, GPT-3.5 (Figure A.9) exhibits a reduction in overall fluctuations in the results. This is evidenced by the consistent resolution of certain problems that previously failed, including append, car, eq and shiftl. Given the good results observes with GPT-4 on Python (Figure A.11) using the previous prompt, the improvements achieved

in this case are comparatively less pronounced. Nevertheless, there has been a general improvement in stability and performance. The instances where the problem consistently solves have increased to include last. Haskell also demonstrates some improvements, although GPT-3.5 (Figure A.10) still exhibits excessive inconsistency in achieving clear results. In contrast, GPT-4 (Figure A.12) demonstrates the most significant improvements in comparison to previous results. In addition to the previous cases where the model did accurately solved the problem, this method enables the solution of further operations, including append, car, cumulative_sum, last, lasts, length, mult, multfirst, reverse swap and switch.

Nevertheless, this solution could be further enhanced, as fluctuations are still present, particularly in the case of code generation in Haskell using GPT-3.5.

Table 4.3: Mean Scores over number of examples for GPT-3.5 and GPT-4 on Python and Haskell using Chain of Thought in addition to defining the desired code structure.

Function	Python Mean Scores(%)		Haskell Mean Scores(%)	
	GPT-3.5	GPT-4	GPT-3.5	GPT-4
ack	11.5	14.0	4.74	1.05
add	70.0	100.0	13.5	100.0
add_even_prod	7.0	6.0	15.26	1.03
append	90.0	100.0	42.0	85.0
car	99.5	99.5	34.5	85.5
cdr	48.5	75.5	30.0	75.5
char_shift	17.5	70.0	10.0	13.0
cumulative_sum	49.0	100.0	11.5	80.0
drop	62.0	52.5	47.5	16.5
entangle_strings	32.0	95.5	9.5	58.5
eq	71.5	94.5	16.5	100.0
even	27.5	53.0	24.5	41.5
evenlist	38.0	62.0	36.5	54.5
evenpos	32.0	38.0	37.0	44.0
fib	8.0	42.5	22.0	31.5
geq	53.0	57.0	12.5	56.0
insert	10.5	93.0	29.0	60.0
last	71.0	96.0	39.0	95.0
lasts	5.0	86.5	11.0	76.5
length	84.5	95.0	18.5	80.0
length_non_repeating_char	39.0	84.0	19.5	42.5
member	68.5	100.0	34.0	50.0
mult	10.0	93.5	8.5	86.5
mult_add	0.0	0.0	17.0	13.5
mult_char	70.0	70.0	56.0	14.0
mult_plus_one	7.0	5.5	13.5	16.0
multfirst	27.5	89.0	30.5	91.0
multlast	13.0	45.5	22.0	27.5
nested_factorial_sum	14.5	4.0	8.5	16.0
odd	52.5	63.0	20.0	66.0
oddpos	35.0	28.0	32.5	37.5
oddslist	47.0	65.0	30.0	60.0
power_sum	0.5	1.0	3.0	9.0
reverse	77.0	100.0	71.5	100.0
shiftl	90.0	92.5	60.5	50.0
shiftr	25.5	99.0	35.0	82.0
sort	67.5	83.0	25.5	69.0
sum	55.5	95.0	44.5	95.0
swap	32.5	67.0	24.0	70.0
switch	40.0	95.0	13.5	100.0
symmetrical_pair_prod	9.0	0.5	5.0	0.5
take	20.0	33.5	37.5	42.0
weave	53.5	86.0	10.5	58.0
weighted_char_shift	0.0	60.0	9.5	0.0

As research in the field of prompt engineering has continued to develop, More sophisticated methodologies have been devised for optimising the full potential of LLM

systems. One of the promising strategies is flow engineering [81]. Flow engineering focuses on optimising the interaction between LLMs and task-specific workflows. This approach involves the structuring of LLM tasks as sequences of interconnected steps or "flows". Each step represents a discrete unit of the task that can be independently executed and refined. The division of complex tasks into manageable sub-tasks enables flow engineering to enhances the model's ability to handle intricate workflows with greater accuracy. This enables more precise control over the model's output and ensures that the model adheres to the specified task-specific guidelines in an accurate and consistent manner.

The efficacy of this strategy has led to the implementation of this approach to address complex tasks such as code generation. This resulted in the development of AlphaCodium [65], a framework that integrates prompt engineering with flow engineering. This framework enables the generation of code in an efficiently manner by structuring tasks into logical flows. This process involves the utilisation of LLMs to interpret and transform natural language prompts into sequences of actions that create code, ensuring the code aligns with user specifications and computational requirements. By applying principles of flow engineering, AlphaCodium enhances the process of code generation, enabling LLMs to generate coherent, functional code by following defined procedural steps. This integration of structured workflows and dynamic code generation shows how flow engineering enables LLM-driven code generation in complex technical domains.

In light of the framework proposed by AlphaCodium, we decided to build our own implementation tailored to our specific task. This resulted in the graph flow depicted in Figure 4.4. The implementation builds a simpler system than that proposed in AlphaCodium. Based on our previous prompt configuration, which combined the description of our task, the desired code template with which to generate the output and the Zero-Shot-Chain-of-Thought strategy, we feed this prompt to the model with its respective problem examples (prompt node). Given the code generated by the model (code generation node). Subsequently, the generated code is subjected to a code compilation node, which serves to identify any potential compilation errors. In the event of a compilation error in the generated code, the model is prompted again, this time with a prompt containing the initial prompt, the generated code solution for the prompt and the specific compilation errors. This stage is defined as the "reflecting" stage. In the event that no compilation errorThe objective of this stage is to ensure that the model is able to at least identify the relationship between the examples provided. Should the generated code be unable to function as intended with the provided examples, the model will be prompted again (reflect node), this time with a prompt containing the initial prompt, the generated code solution for the prompt and specific examples where the code did not generate the desired response. Nevertheless, if the generated code solution is found to be effective in addressing the given examples, the final solution will be saved and the execution will be terminated.

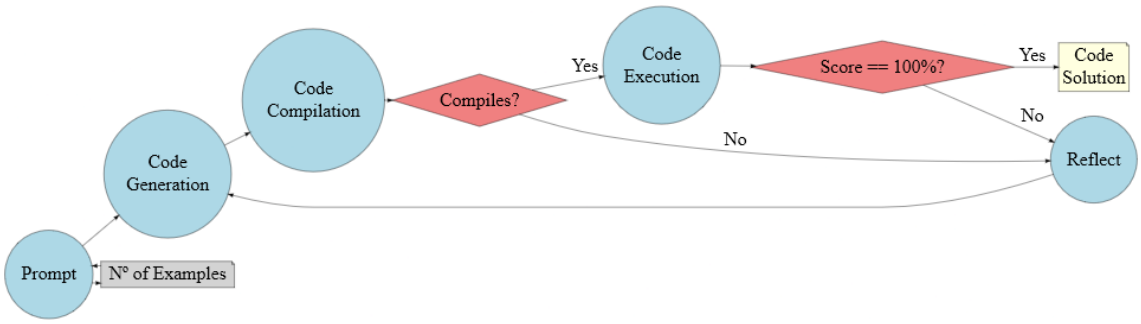

Figure 4.4: Proposed flow method.

The workflow was employed to generate code solutions for each configuration. A limitation of 15 iterations was imposed to prevent infinite looping, with each iteration of the code generation process by the LLM being considered as a full iteration of the associated graph.

This configuration is considered to be the final one. A more detailed analysis of the results will be presented in Section 5. Our research indicates that our proposed configuration represents the optimal approach for reliably utilising an LLM to capture the relationship between inputs and outputs provided by examples. By constructing a well-defined prompt that provides a detailed description of the task, the specific code structure that is desired, utilising Chain-of-Thought in addition to iteratively improving the code by checking for compilation errors and ensuring that it is able to work on the provided examples, we are able to accurately assess the full reasoning capabilities of large language models without the need for additional constraints.

CHAPTER 5
Final Results

In this section we show the results for our final configuration. An analysis will carried out to see if the results actually support our initial hypothesis. By further evaluating these results, we also hope to look for global patterns in the data that may provide insight into the behaviour of the models.

In this final stage of the evaluation process, we will be utilising the proposed dataset (Subsection 4.2.1). As detailed in Subsections 3.1.1 and 3.1.2, the generation will be evaluated for both Python and Haskell using both GPT-4 and GPT-3.5. The final prompt strategy will consist of a combination of flow engineering and prompting. In more precise terms, an autonomous agent has been designed, which, upon a first proposed solution, iteratively attempts to improve until it successfully generates a function that captures the relationship between the input-output examples seen by the model. The final prompt solution and the methodology employed to arrive at it are presented in Section 4.2.2.

Figures 5.1, 5.2, 5.3 and 5.4 illustrates the performance of the final configuration broken down by programming language, model and task. The graphs illustrate the performance of the generated code for the problems in the dataset. Each graph plots the code's score on the y-axis against the number of examples used to generate the code on the x-axis. A score of 100% represents the optimal performance, indicating that the code generalises well across all examples. The x-axis ranges from 1 to 10 examples, illustrating the function's performance as the number of examples used to generate the code increases. A red rectangle is used to highlight the distinction between simple and complex problems, as described in a specific subsection.

The results demonstrate an overall improvement in performance across all configurations. In the case of Python, GPT-3.5 exhibits an increase in performance and stability, with tasks being solved consistently across all examples. These include add, append, last, length, reverse, shiftl, sort and sum. In GPT-4, this trend is more pronounced. It is evident that the majority of tasks are being solved, with only a few instances where the model consistently fails to produce an output. In contrast, for Haskell, GPT-3.5 exhibits a decline in performance and an increase in variability in comparison to Python. A similar trend is observed in GPT-4 for Haskell, where a significant proportion of tasks are solved. However, Haskell exhibits greater fluctuations than Python.

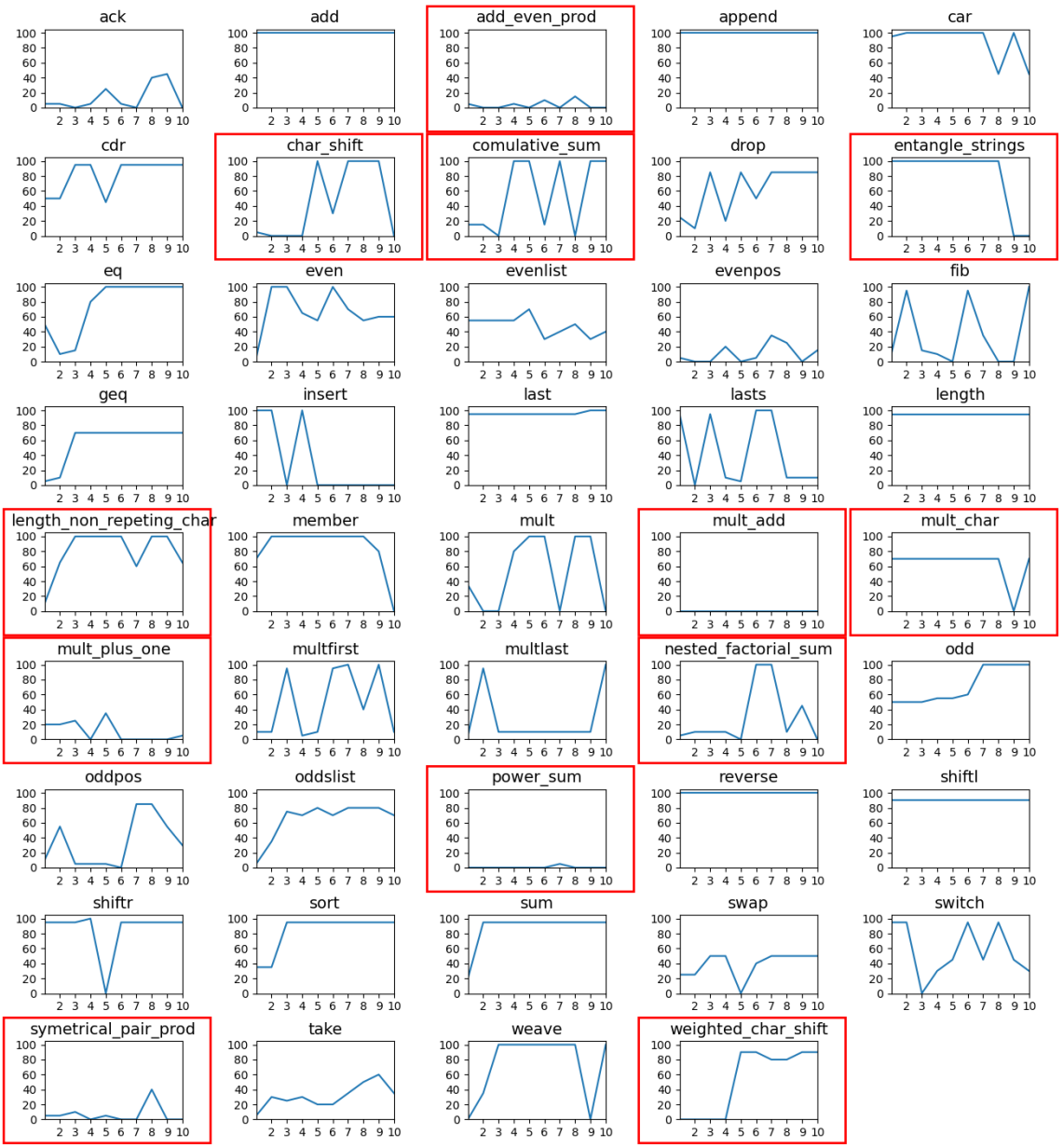

Figure 5.1: GPT-3.5 scores for each function in Python using our proposed coding agent. Red rectangles highlight the complex problems.

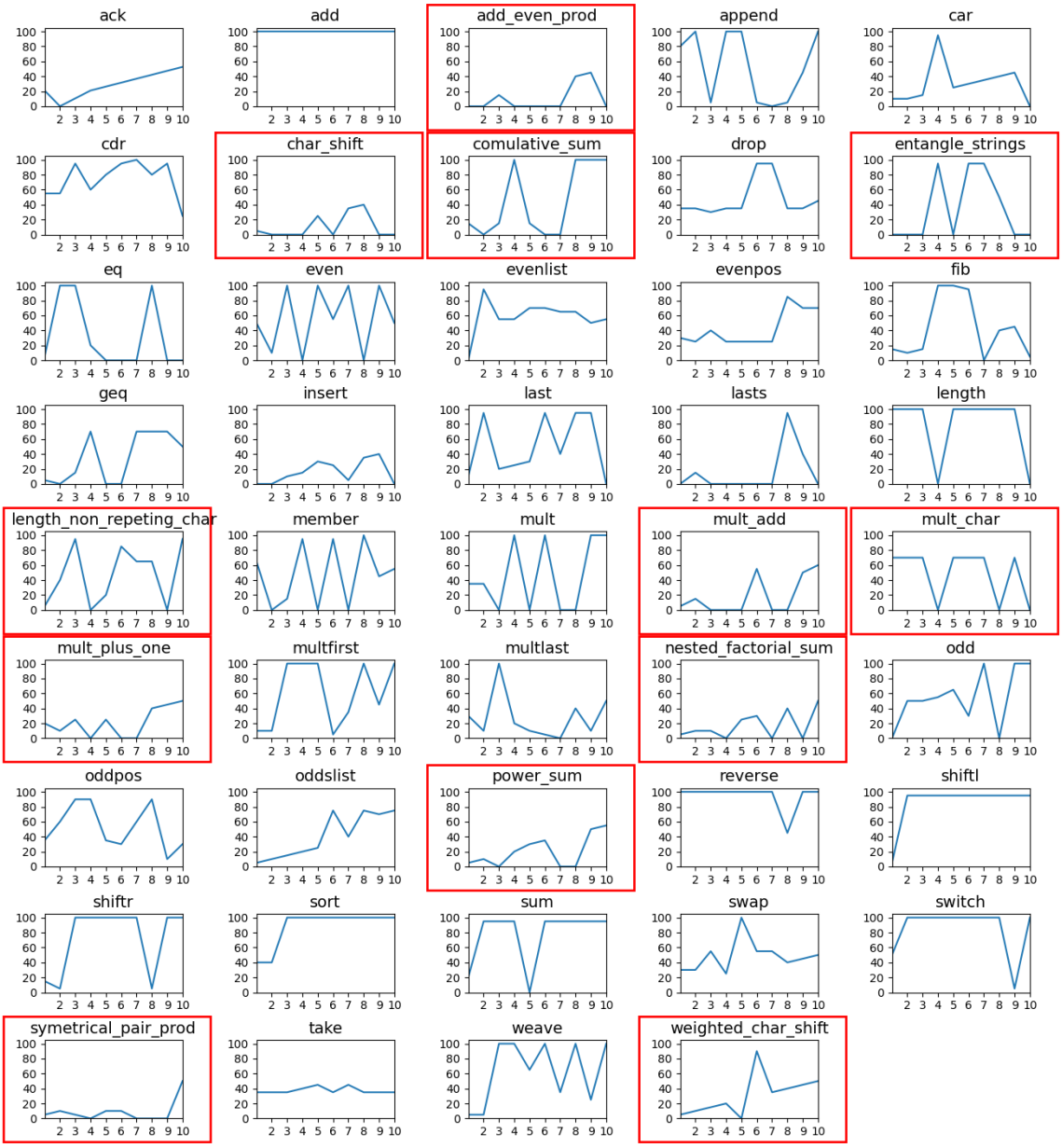

Figure 5.2: GPT-3.5 scores for each function in Haskell using our proposed coding agent. Red rectangles highlight the complex problems.

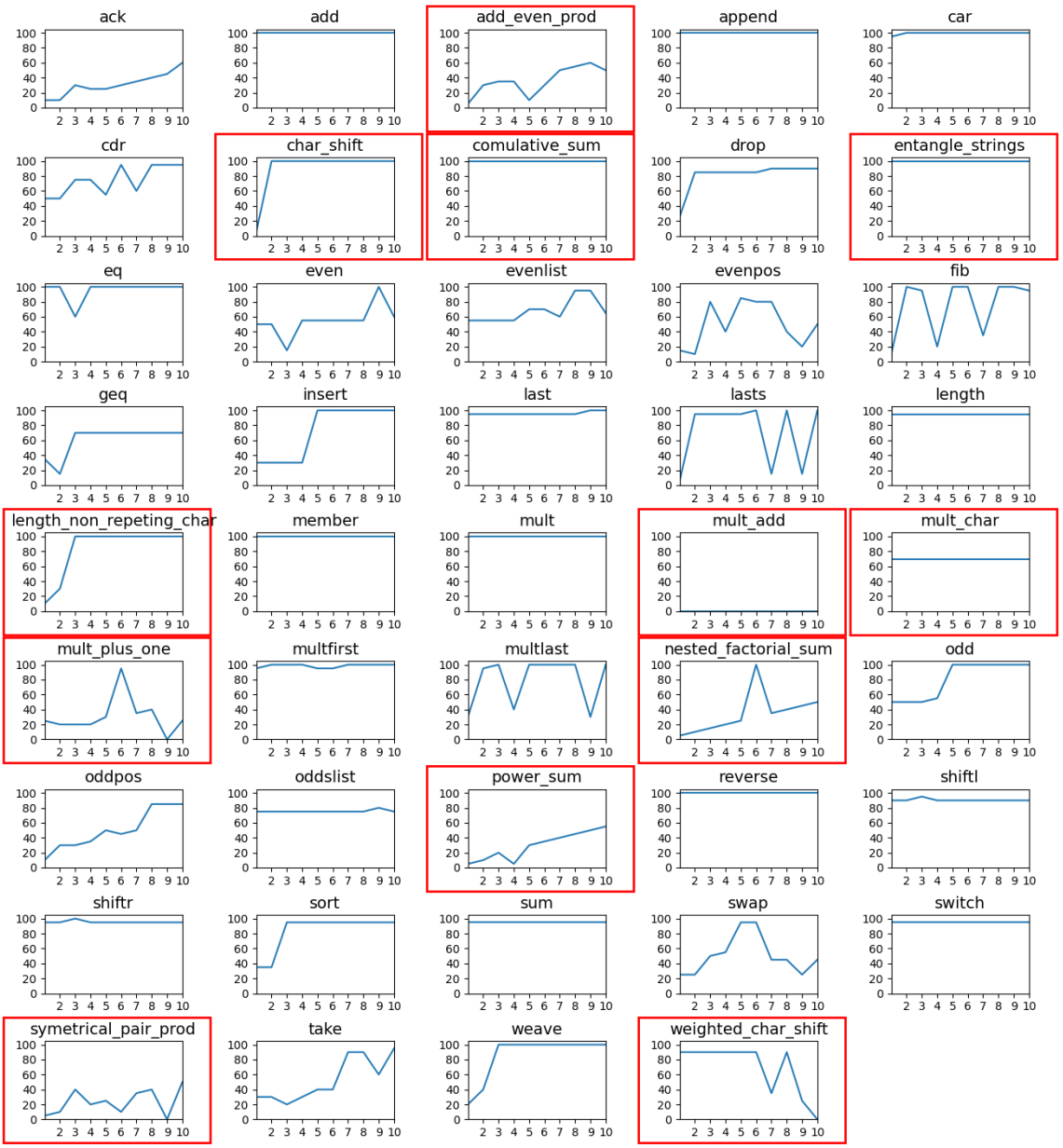

Figure 5.3: GPT-4 scores for each function in Python using our proposed coding agent. Red rectangles highlight the complex problems.

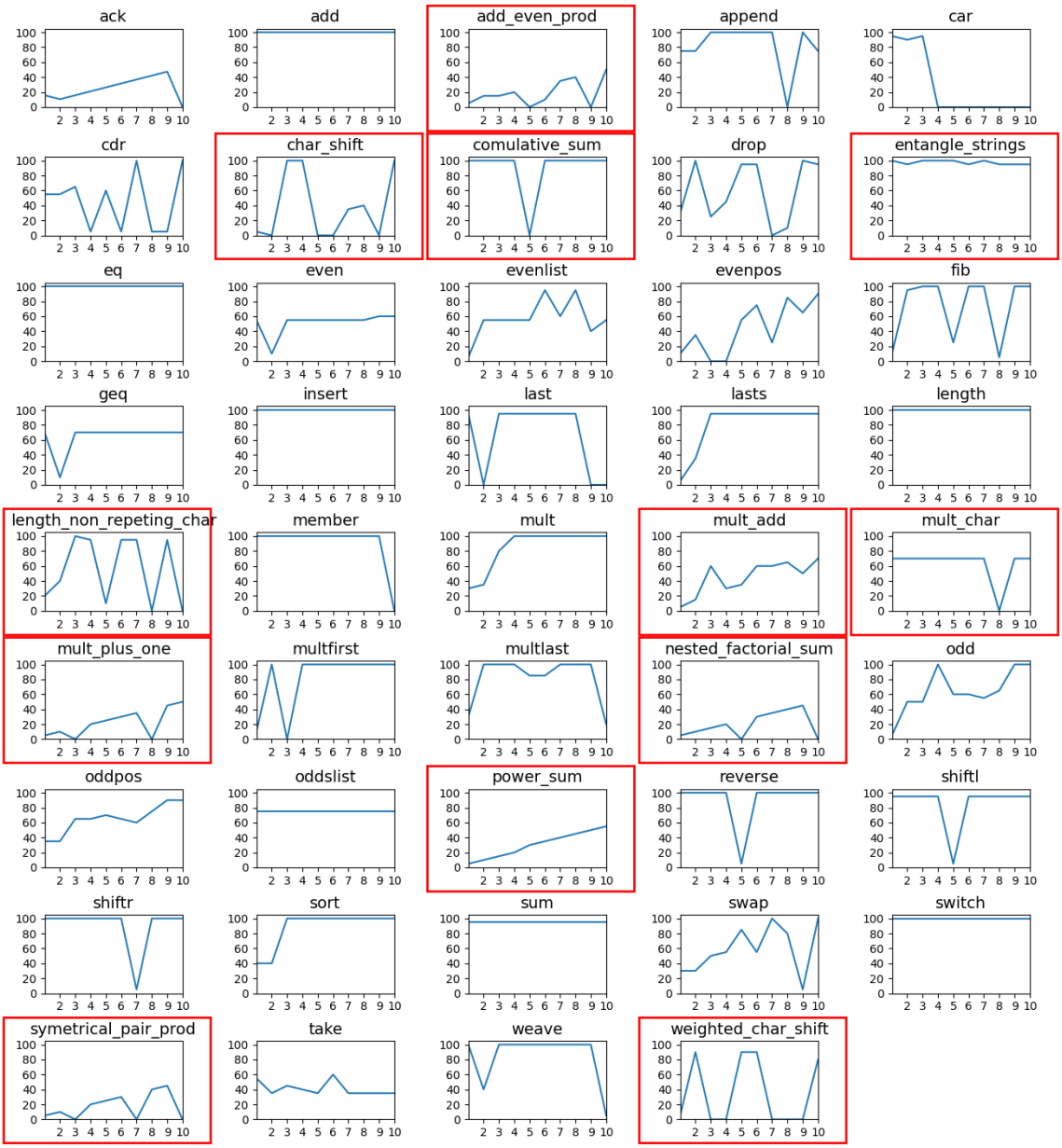

Figure 5.4: GPT-4 scores for each function in Haskell using our proposed coding agent. Red rectangles highlight the complex problems.

Although results per task provide valuable insight into the specific performance in each task, the figures themselves are too complex and contain too much information. This overwhelm of information makes it challenging to identify patterns in the data or global trends.

By calculating the mean performance per task for each configuration (Figure 5.5), interesting patterns emerge. The data presented in this figure clearly delineates two distinct tendencies. Firstly, the results show that both programming languages the result show a considerable improvement from the performance of GPT-3.5 to GPT-4. Additionally, the results demonstrate that, for both LLMs, Pythons configurations exhibits superior performance compared to Haskells configurations. This indicates that the best performance configuration is GPT-4 utilising Python. This findings corraborate our initial hypothesis. As anticipated, the superior capabilities of GPT-4 are reflected in its superior performance. Also, as Python is a more common programming language, it will definitely be

more overrepresented than Haskell in the training data of both models, thus resulting in superior performance in Python over Haskell.

However, the most striking findings emerge when examining the relationship between performance and the number of examples. Contrary to expectations, there is no discernible correlation between the two variables. It is noteworthy that, in all cases, there is no discernible increase in performance when the number of examples is augmented. This behaviour only appears at the one-to-two examples range. In this case, there is an improvement in all cases, with the exception of GPT-3.5 with Python.

The results demonstrate that LLMs show problems of saturation. This implies that if an LLM is unable to discern a relationship in the initial two examples, the addition of further examples will not result in the acquisition of any further insights. One might posit that this behaviour is analogous to that observed in humans, where the identification of a summation from two examples is relatively straightforward. However, the addition of numerous examples may prove to be a hindrance, rather than a benefit.

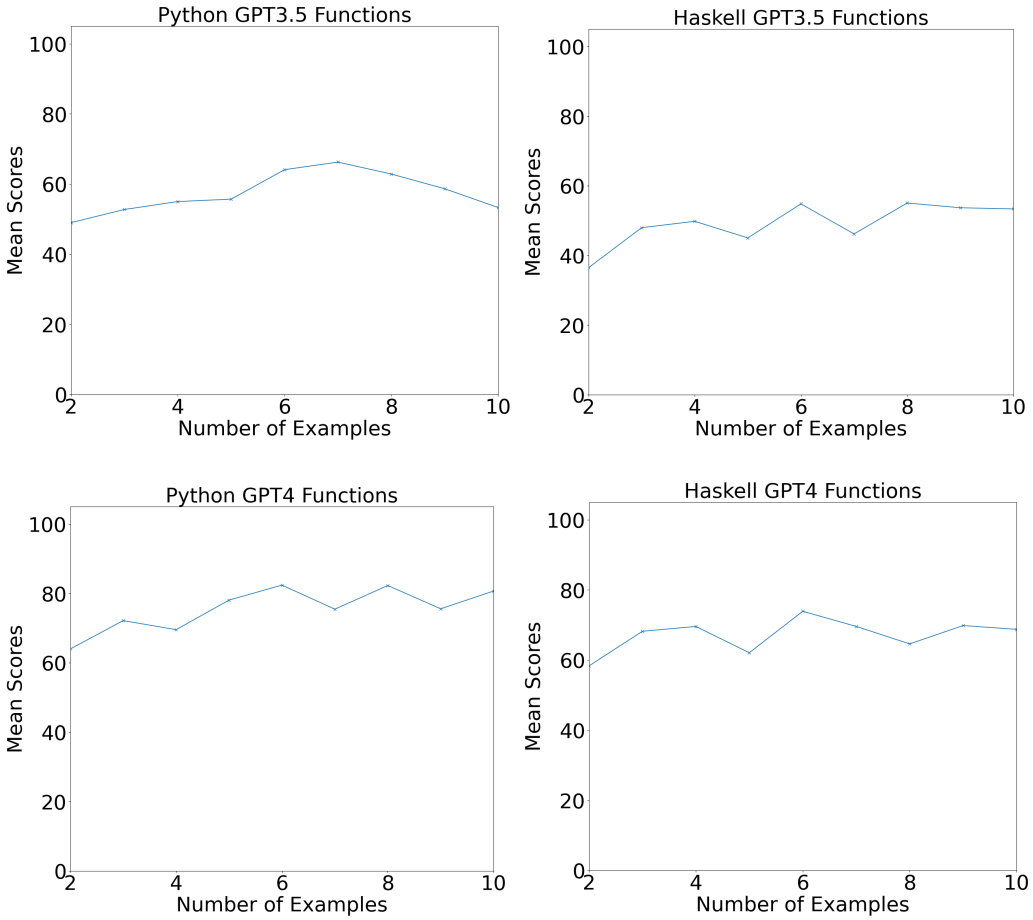

Figure 5.5: Comparison of mean scores over **all problems** for function generation in Python and Haskell using GPT-3.5 and GPT-4. The plots depict the performance trends for Python (top-left and bottom-left) and Haskell (top-right and bottom-right) across both model versions.

In light of these intriguing findings, it is worth further analysing the data in search for more valuable insights. The dataset includes both typical coding problems, such as member, add, and fibonacci, which are commonly encountered in learning environments, and atypical tasks that were specifically created for this study. It would be of interest to segregate the results in order to obtain more meaningful insights.

Figures 5.6 and 5.7 illustrate the mean performance over typical and atypical tasks, respectively. The outcomes of both experiments follow the same patterns as those observed when using all the functions. Both models saturate, and the correlations between performance of GPT-3.5, GPT-4, Haskell and Python remain consistent. A comparison of the two results reveals a significant distinction between typical and atypical problems. The tasks that are more commonly encountered result in a significantly higher performance when compared to the tasks that are defined as uncommon.

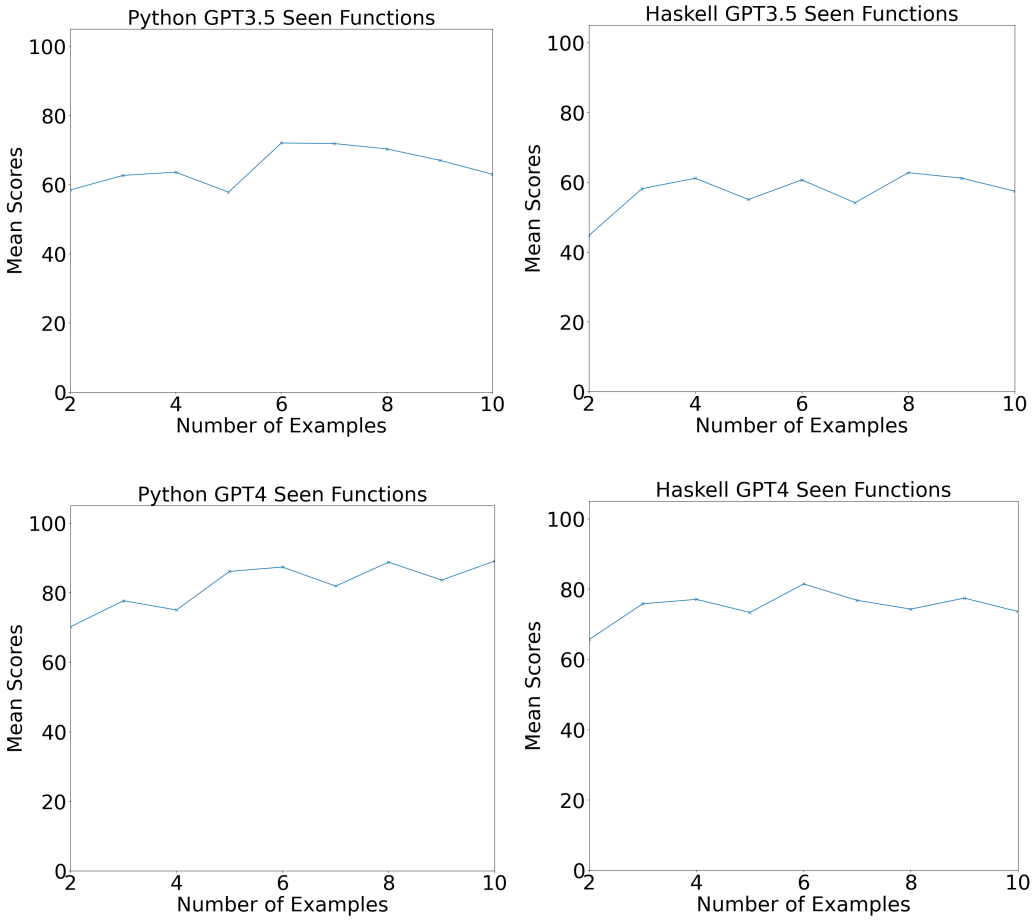

Figure 5.6: Comparison of mean scores over **typical problems** for function generation in Python and Haskell using GPT-3.5 and GPT-4. The plots depict the performance trends for Python (top-left and bottom-left) and Haskell (top-right and bottom-right) across both model versions.

Figure 5.7: Comparison of mean scores over **atypical problems** for function generation in Python and Haskell using GPT-3.5 and GPT-4. The plots depict the performance trends for Python (top-left and bottom-left) and Haskell (top-right and bottom-right) across both model versions.

The observed discrepancies in outcomes are indicative of promising outcomes. However, without a formal definition of complexity, it is not possible to state with confidence that the observed differences in performance between typical and atypical tasks are due to contamination. It is possible that the more atypical tasks are, in fact, more challenging to solve than the typical tasks. Due to time constraints, this topic could not be fully addressed in this work. However, we hope that this will provide motivation for future research in this area.

A comparative analysis between an IFP system and the final configuration could yield valuable insights. As previously stated in Section 2, MagicHaskeller is an IFP system that has shown good performance. Following an evaluation of a number of IFP systems conducted by the University of Bamberg[1], we opt to compare our approach with that of MagicHaskeller. In comparison to other sophisticated systems, such as Igor2, MagicHaskeller demonstrated superior overall performance.

Table 5.1 presents a comparison between MagicHaskeller and GPT-4, utilising both the Python and Haskell programming languages. The results demonstrate the percentage of instances where the model correctly identified the relationship. These scores represent the optimal performance, regardless of the number of examples employed. Due to time constraints, some tasks were not evaluated with MagicHaskeller.

[1] https://www.inductive-programming.org/repository.html

Table 5.1 shows the comparison between MagicHaskeller and GPT-4 using both Python and Haskell. The results show the percentage of examples where the model correctly captured the relationship. This scores where taken for the best performing case, without considering the number of examples. Due to time constrains, some tasks where not evaluated with MagicHaskeller.

In inductive programming systems, a search is conducted within the space of potential programs, resulting in either the successful identification of a program that models the relationship or an unsuccessful outcome. This is why there is an absence of intermediate values between 0 and 100 in the results.

A comparison of the two systems reveals instances where the IFP system is unable to solve certain tasks, while LLMs succeed. This is evident in tasks such as last, oddslist, sum, swap, and switch. These results suggest that, in terms of pure performance for these specific cases, LLMs outperform IFP systems. However, it is important to note that IFP systems offer other advantages, such as efficiency and lower time and cost complexity, which were not considered in this evaluation.

The assessment of the final configuration substantiates the initial hypothesis. The analysis indicates a notable enhancement in performance from GPT-3.5 to GPT-4, particularly in the context of Python, which consistently demonstrates superior performance compared to Haskell. This support the assumption that GPT-4 advanced capabilities contribute to its superior performance, and that Python's dominance in training datasets enhances its performance over Haskell. The study also reveals an interesting phenomenon: a saturation point for LLMs. It appears that beyond the initial one or two examples, additional data does not enhance performance. This saturation phenomenon indicates that LLMs exhibit the capacity to identify patterns at an early stage, with additional examples failing to enhance the clarity of relationships. Moreover, typical coding problems elicit superior performance compared to atypical ones. Nevertheless, it is challenging to infer the underlying cause of this phenomenon, which presents an opportunity for future research extensions. Furthermore, a comparison with the results obtained by MagicHaskeller has demonstrated that, at least in the context of this comparison, LLMs exhibit superior performance. It is, however, important to recognise that IFP systems offer distinct advantages in terms of efficiency, time, cost and complexity.

Table 5.1: Comparison of best performing IP systems vs our results for GPT-4 with Python and Haskell. Best resulting scores are chosen for both models.

Problem	MagicHaskeller(%)	GPT-4 Python(%)	GPT-4 Haskell(%)
ack	100	60	47
add	100	100	100
add_even_prod	Not evaluated	60	50
append	100	100	100
car	100	100	95
cdr	100	95	100
char_shift	Not evaluated	100	100
cumulative_sum	Not evaluated	100	100
drop	Not evaluated	90	100
entangle_strings	Not evaluated	100	100
eq	100	100	100
even	100	100	60
evenlist	100	95	95
evenpos	100	85	90
fib	100	100	100
geq	100	70	70
insert	Not evaluated	100	100
last	0	100	95
lasts	100	100	95
length	100	95	100
length_non_repeating_char	Not evaluated	100	100
member	Not evaluated	100	100
mult	100	100	100
mult_add	Not evaluated	0	70
mult_char	Not evaluated	70	70
mult_plus_one	Not evaluated	95	50
multfirst	100	0	100
multlast	100	100	100
nested_factorial_sum	Not evaluated	100	45
odd	100	100	100
oddpos	100	85	90
oddslist	0	80	75
power_sum	Not evaluated	55	55
reverse	100	100	100
shiftl	100	95	95
shiftr	100	100	100
sort	100	95	100
sum	0	95	95
swap	0	95	100
switch	0	95	100
symmetrical_pair_prod	Not evaluated	50	45
take-n	100	95	60
weave	100	100	100
weighted_char_shift	Not evaluated	90	90

CHAPTER 6
Conclusion

Given the objectives defined at the start of this work. This section will assess whether the objectives in fact have been met. This will be demonstrated by the evidence presented throughout the chapters of this work.

- **Literature Review and Benchmark Creation**
  - *Conduct a Comprehensive Literature Review*:
    - Perform an exhaustive search of existing literature on PBE and LLMs.
    - Utilise academic conferences, and journals to gather relevant studies and reviews to provide a global picture of the field of PBE.
    - Identify current methodologies, datasets, benchmarks.
  - *Develop a Diverse Benchmark for Evaluation*:
    - Design a benchmark that includes tasks of varying complexity to reflect a wide range of scenarios.
    - Assess the impact of the number of examples on task definition and solution accuracy.
    - Ensure that tasks can be fully and unambiguously defined with a minimal set of examples.

The concerns and challenges addressed in this point are further detailed in Subsection 4.2.1. Subsection 4.2.1 contains our proposed solution design with regard to the dataset. Looking back, it becomes evident that a benchmark was indeed identified successfully for our specific task. It was correctly identified that this benchmark, in fact, lacked more atypical and challenging problems, as it consisted only of common programming problems. This resulted in a dataset of 44 tasks, which can be consider to be complete and diverse enough.The process of defining a task, particularly in terms of its complexity, proved to be more challenging than initially anticipated. This is discussed in greater detail in Subsection 4.2.1. This represents an unaddressed topic within the scope of our work, which presents an opportunity for further extensions of this work. The number of examples that comprise a task was empirically determined. Initially, a range of 1 to 10 examples was selected as a starting point. Given the results obtained, here was no need to extend the number of examples beyond 10, as no correlations were observed between the number of examples and the performance in a task.

- **Monitoring and Framework Establishment**
  - *Ongoing Monitoring of Advancements in LLMs*:

    - Keep abreast of new developments and publications related to LLMs and their applications in code generation.
    - Evaluate how recent advanced might influence ongoing research and methodology in PBE.
  - *Establish Model and Language Foundations*:
    - Identify and document the LLMs to be used, considering their strengths and weaknesses in relation to PBE.
    - Select programming languages for evaluation based on their prevalence in inductive programming systems, LLMs and other relevant metrics.
    - Justify the choice of models and languages in terms of their suitability for PBE tasks.

We believe that both the resulting models and programming languages have been chosen on the basis of an informed decision, taking into account all the factors to be considered. In terms of models, GPT-4 has remained the reference in terms of performance throughout this work, and GPT-3.5 has been useful in proving our initial expectations and setting the baseline (for more information on why we chose these models, see Subsection 3.1.2). The decision to chose Python and Haskell as our final programming languages is explained in Subsection 3.1.1. As expected, Python gave the best performance. Haskell also behaved according to our expectations, being less represented in the training data, it shows worst performance than Python in all cases. It may be that it would have been interesting to include more programming languages, but we believe that this could be presented as a more comprehensive evaluation, specifically tailored to programming languages.

An evaluation of the follow-up in research literature during this work presents as a difficult objective to prove with evidence. However, if we carefully analyse the Section 4.2.2, this can indeed be seen. As part of our methodological approach to constructing the best fitting prompt for our use case, this resulted in the need to iteratively improve on previous results. This iterative improvement led to an extensive search for directions to proceed. In fact, AlphaCodium [65], the paper that inspired our final prompt design, was actually published during the time we were developing this work.

- **Detailed Evaluation and Analysis**
  - *Perform Comprehensive Evaluation*:
    - Develop a detailed evaluation framework to assess the performance of LLMs on the benchmark tasks.
    - Compare LLM performance across different tasks and programming languages to identify strengths and weaknesses.
  - *Conduct In-Depth Analysis of Results*:
    - Analyse the results to draw meaningful conclusions about the capabilities and limitations of LLMs in PBE.
    - Provide insights into areas for improvement and potential future directions for research.

The above detailed evaluation is included in the Section 5. This section, in addition to showing the final results, also extends to analysing global trends over all tasks and by complexity and compares our results with other IP systems. This gives an overview of the general performance of the given models and allows us to empirically hypothesise

about the behaviour of LLMs for the given task. However, an even more detailed and exhaustive evaluation would have been possible if the above-mentioned unresolved issues had been addressed.

Finally, it is worth noting the differences between our work and that of [41]. As mentioned in Section 2, our work also aims to evaluate LLMs for PBE. However, we believe that there are significant differences in the approach taken. In contrast to our implementation, the authors of the aforementioned paper decided to build a dataset using a variety of existing datasets, rather than creating a new one. This resulted in a dataset comprising a considerable number of tasks, including those related to numerical vectors, string manipulation, and graphical programs. In comparison, our dataset is less diverse in terms of both quantity and range of tasks. We believe that this reduced number of tasks facilitates a more straightforward comparison of performance in specific tasks. In their evaluations, they obtained results for GPT-4 and a fine-tuned system. The results demonstrated the efficacy of the model in generating solutions to given tasks, with the evaluation conducted on a set of N programs, where N ranged from 1 to 200 samples. Our approach, however, is superior to this brute-forcing method, as the generation of 200 code samples is both costly and time-consuming. In contrast, our iterative improvement strategy involved the generation of a maximum of 15 samples. A further shortcoming of the paper is the arbitrary definition of the number of examples to be included. This is determined on a task-by-task basis, without any clear criterion for making this decision.

In conclusion, we consider that the objectives of our work have been successfully achieved. For each objective, we have provided a detailed account of the evidence that supports our assertion and explained the reasons why we believe that this has been achieved in our work. We have also demonstrated how our approach differs from previous works in this domain.

6.1 Limitations and Future Work

This work demonstrates that, despite the promising capabilities demonstrated by large language models, programming by example remains an open problem. Further research on evaluation, with this work as a starting point, could prove beneficial. Furthermore, work on improving the alignment of these systems to the desired prompt should of help. In this section, we aim to provide insight into potential paths for improvement to this work and additional avenues for a more comprehensive evaluation.

Firstly, a number of potential enhancements can be made to the dataset employed. It is acknowledged that the dataset contains unresolved flaws that could potentially affect the performance of LLMs. In the course of our evaluations, no consideration was given to the order in which the examples were presented. Additionally, no detailed consideration was given to the specific examples used, nor to their informational value. Due to time constraints, this topic was not fully addressed; however, it is open to further investigation. Furthermore, improvements could be made to the presented dataset. One potential avenue for improvement would be to expand the scope of the problems included in the dataset. It is proposed that the evaluation be expanded by including additional example-based tasks in the form of graphs or logic problems. This would broaden the scope of the evaluation. Another potential avenue for improvement concerns the categorisation of each problem in terms of its complexity.

It would be possible to extend the evaluation to cover a greater number of models. We standby in the decision of only using GPT-3.5 and GPT-4 for this work. Nevertheless, a comparison of other models may be of interest. In particular, if such a comparison encompasses an extensive array of models, exhibiting a diverse range of sizes and gen-

eral capabilities or those specifically crafted for code generation. An alternative approach could be to construct a more specialised model as presented in [41], trained for the specific task of code generation using input-output examples. With regard to the issue of prompting, we believe that our proposed method has been sufficiently refined. Nevertheless, as novel strategies are developed and the alignment between the prompt and its intended purpose is enhanced, this could potentially yield intriguing new insights.

Although the approach is of interest, we believe that a more thorough evaluation using other programming languages will not yield any new insights. This is because we believe that as a benchmark, this is not sufficient for evaluating the performance of a model within different languages. Nevertheless, it could be a valuable addition to a more diverse existing benchmark.

In conclusion, this study demonstrates that despite the advances presented by LLMs, the domain of programming by example continues to present significant challenges. Our investigation highlights the necessity for further research in the evaluation of these systems, with this study serving as a preliminary point of departure. In conclusion, the proposed avenues for further investigation, namely dataset enhancement, model comparison and prompt refinement, provide a framework for future research. By addressing these avenues, we can advance towards more effective and accurate programming by example using LLMs.

6.2 Relation to Studies

The completion of this project would not have been possible without the acquisition of the skills and knowledge gained during the Bachelor of Computer Science degree at the Technical University of Valencia. The degree provided not only the technical skills required to complete this project, but also the development of soft skills that were applied across the duration of the project.

Soft skills have been of vital importance. The most crucial skill for this work has been problem-solving. Having the capacity to confidently address the problem by breaking it down into simpler tasks, then managing the pacification of these tasks and evaluating the time and effort required has been of great use. This skill has been largely covered in Project Management, thanks to its practical approach to the project cycle, which has helped to acquire essential skills for the management of any project size.

In addition, the acquisition of technical and specific skills has been of great benefit in this context. The knowledge acquired in Programming Languages, Technologies and Paradigms has enabled informed decision-making regarding the selection of appropriate programming languages, with an understanding of the distinctive paradigms and features that characterise each programming language. Machine Learning has proven invaluable in providing a robust foundational understanding of neural networks and supervised and semi-supervised learning techniques. In particular, Information Storage and Retrieval Systems have been of great use. his subject encompasses the study of data structures and algorithms employed for the management of unstructured information in large volumes of data, predominantly text data. Both Machine Learning and Information Storage and Retrieval Systems can be considered foundational technologies that have been employed in the construction of large language models. Intelligent Agents have been instrumental in the development of the final prompting strategy employed in the evaluations. The proposed method, which comprises flow engineering and a prompt, may be regarded as an agent that is capable of generating code and taking actions to rectify this code until a satisfactory result is achieved.

In addition to the aforementioned skills, it is also important to consider those that are often overlooked in the context of computer science. Firstly, programming skills developed throughout the degree, particularly in Introduction to Computer Science and Programming and Programming courses, where skills were acquired for the purpose of producing high-quality code. Finally, it is important to consider the development of writing skills, particularly those acquired in English for Computing (B2), where students were required to produce accurate written work in English related to Computer Science.

The integration of these diverse skills and knowledge areas exemplifies the comprehensive and multidisciplinary nature of the Computer Science degree. The curriculum is balanced, imparting critical technical competencies and emphasising essential soft skills and practical applications. The degree has fostered the development of problem-solving abilities, technical expertise, and communication skills, which have been pivotal in overcoming the challenges faced throughout this work and ensuring a well-rounded and effective approach to its completion. This foundation will undoubtedly serve as a robust platform for future endeavours in the field of computer science and beyond.

Bibliography

[1] Uri Alon et al. code2seq: Generating Sequences from Structured Representations of Code. 2019. arXiv: 1808.01400 [cs.LG].

[2] Xavier Amatriain. Prompt Design and Engineering: Introduction and Advanced Methods. 2024. arXiv: 2401.14423 [cs.SE]. URL: https://arxiv.org/abs/2401.14423.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. 2016. arXiv: 1409.0473 [cs.CL].

[4] Yoshua Bengio et al. "A neural probabilistic language model". In: J. Mach. Learn. Res. 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435.

[5] Alan W. Biermann. "The Inference of Regular LISP Programs from Examples". In: IEEE Transactions on Systems, Man, and Cybernetics 8.8 (1978), pp. 585–600. DOI: 10.1109/TSMC.1978.4310035.

[6] Tom B. Brown et al. Language Models are Few-Shot Learners. 2020. arXiv: 2005.14165 [cs.CL].

[7] Wray L. Buntine. "Generalized Subsumption and Its Applications to Induction and Redundancy". In: Artif. Intell. 36 (1986), pp. 149–176. URL: https://api.semanticscholar.org/CorpusID:27180464.

[8] Pablo Antonio Moreno Casares et al. "How General-Purpose Is a Language Model? Usefulness and Safety with Human Prompters in the Wild". In: Proceedings of the AAAI Conference on Artificial Intelligence 36.5 (June 2022), pp. 5295–5303. DOI: 10.1609/aaai.v36i5.20466. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20466.

[9] Kaiyan Chang et al. Efficient Prompting Methods for Large Language Models: A Survey. 2024. arXiv: 2404.01077 [cs.CL]. URL: https://arxiv.org/abs/2404.01077.

[10] European Commission. Artificial Intelligence Act. EUR-Lex. Proposal for a Regulation of the European Parliament and of the Council Laying Down Harmonized Rules on Artificial Intelligence (Artificial Intelligence Act) and Amending Certain Union Legislative Acts. 2021. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:52021PC0206.

[11] Domenico Corapi and Alessandra Russo. ASPAL. Proof of soundness and completeness. Tech. rep. Technical Report DTR11-5, Department of Computing, Imperial College, London, 2011.

[12] Andrew Cropper and Sebastijan Dumancic. "Inductive logic programming at 30: a new introduction". In: CoRR abs/2008.07912 (2020). arXiv: 2008.07912. URL: https://arxiv.org/abs/2008.07912.

[13] Andrew Cropper and Rolf Morel. Learning programs by learning from failures. 2020. arXiv: 2005.02259 [cs.AI].

[14] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

[15] Jeffrey L. Elman. “Finding structure in time”. In: Cognitive Science 14.2 (1990), pp. 179–211. ISSN: 0364-0213. DOI: https://doi.org/10.1016/0364-0213(90)90002-E. URL: https://www.sciencedirect.com/science/article/pii/036402139090002E.

[16] European Parliament and Council of the European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council. of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). OJ L 119, 4.5.2016, p. 1–88. URL: https://data.europa.eu/eli/reg/2016/679/oj (visited on 04/13/2023).

[17] Zhangyin Feng et al. CodeBERT: A Pre-Trained Model for Programming. 2020. arXiv: 2002.08155 [cs.CL]. URL: https://arxiv.org/abs/2002.08155.

[18] John K. Feser, Swarat Chaudhuri, and Isil Dillig. “Synthesizing data structure transformations from input-output examples”. In: SIGPLAN Not. 50.6 (June 2015), pp. 229–239. ISSN: 0362-1340. DOI: 10.1145/2813885.2737977. URL: https://doi.org/10.1145/2813885.2737977.

[19] Pierre Flener and Ute Schmid. “An introduction to inductive programming”. In: Artificial Intelligence Review 29.1 (Mar. 2008), pp. 45–62. ISSN: 1573-7462. DOI: 10.1007/s10462-009-9108-7. URL: https://doi.org/10.1007/s10462-009-9108-7.

[20] Alex Graves. “Generating Sequences With Recurrent Neural Networks”. In: ArXiv abs/1308.0850 (2013). URL: https://api.semanticscholar.org/CorpusID:1697424.

[21] Sumit Gulwani. “Automating string processing in spreadsheets using input-output examples”. In: POPL ’11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 317–330. ISBN: 9781450304900. DOI: 10.1145/1926385.1926423. URL: https://doi.org/10.1145/1926385.1926423.

[22] Sumit Gulwani et al. “Inductive programming meets the real world”. In: Commun. ACM 58.11 (Oct. 2015), pp. 90–99. ISSN: 0001-0782. DOI: 10.1145/2736282. URL: https://doi.org/10.1145/2736282.

[23] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: Neural Computation 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[24] Jordan Hoffmann et al. “Training compute-optimal large language models”. In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS ’22. New Orleans, LA, USA: Curran Associates Inc., 2024. ISBN: 9781713871088.

[25] Martin Hofmann. “IGOR2 - an analytical inductive functional programming system: tool demo”. In: PEPM ’10. Madrid, Spain: Association for Computing Machinery, 2010, pp. 29–32. ISBN: 9781605587271. DOI: 10.1145/1706356.1706364. URL: https://doi.org/10.1145/1706356.1706364.

[26] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. “A Unifying Framework for Analysis and Evaluation of Inductive Programming Systems”. In: 2009. URL: https://api.semanticscholar.org/CorpusID:17925691.

[27] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. "Analysis and Evaluation of Inductive Programming Systems in a Higher-Order Framework". In: Deutsche Jahrestagung für Künstliche Intelligenz. 2008. URL: https://api.semanticscholar.org/CorpusID:5111253.

[28] S. Jacindha, G. Abishek, and P. Vasuki. "Program Synthesis—A Survey". In: Computational Intelligence in Machine Learning. Ed. by Amit Kumar et al. Singapore: Springer Nature Singapore, 2022, pp. 409–421. ISBN: 978-981-16-8484-5.

[29] Gonzalo Jaimovitch-López et al. "Can language models automate data wrangling?" In: Mach. Learn. 112.6 (Dec. 2022), pp. 2053–2082. ISSN: 0885-6125. DOI: 10.1007/s10994-022-06259-9. URL: https://doi.org/10.1007/s10994-022-06259-9.

[30] Ziwei Ji et al. "Survey of Hallucination in Natural Language Generation". In: ACM Comput. Surv. 55.12 (Mar. 2023). ISSN: 0360-0300. DOI: 10.1145/3571730. URL: https://doi.org/10.1145/3571730.

[31] Jared Kaplan et al. Scaling Laws for Neural Language Models. 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361.

[32] Susumu Katayama. "MagicHaskeller: System demonstration". In: Approaches and Applications of Inductive Programming. 2011. URL: https://api.semanticscholar.org/CorpusID:253422402.

[33] Susumu Katayama. "Systematic search for lambda expressions". In: Symposium on Trends in Functional Programming. 2005. URL: https://api.semanticscholar.org/CorpusID:11179039.

[34] Emanuel Kitzelmann. "Analytical Inductive Functional Programming". In: Logic-Based Program Synthesis and Transformation. Ed. by Michael Hanus. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 87–102. ISBN: 978-3-642-00515-2.

[35] Emanuel Kitzelmann. "Inductive Programming: A Survey of Program Synthesis Techniques". In: Approaches and Applications of Inductive Programming. 2009. URL: https://api.semanticscholar.org/CorpusID:9340653.

[36] Emanuel Kitzelmann and Ute Schmid. "Inductive Synthesis of Functional Programs: An Explanation Based Generalization Approach". In: J. Mach. Learn. Res. 7 (Dec. 2006), pp. 429–454. ISSN: 1532-4435.

[37] Takeshi Kojima et al. "Large language models are zero-shot reasoners". In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS '22. New Orleans, LA, USA: Curran Associates Inc., 2024. ISBN: 9781713871088.

[38] Jinqi Lai et al. Large Language Models in Law: A Survey. 2023. arXiv: 2312.03718 [cs.CL]. URL: https://arxiv.org/abs/2312.03718.

[39] Nada Lavrac, Saso Dzeroski, and Marko Grobelnik. "Learning Nonrecursive Definitions of Relations with LINUS". In: Proceedings of the European Working Session on Machine Learning. EWSL '91. Berlin, Heidelberg: Springer-Verlag, 1991, pp. 265–281. ISBN: 354053816X.

[40] Alina Leidinger, Robert van Rooij, and Ekaterina Shutova. "The language of prompting: What linguistic properties make a prompt successful?" In: Findings of the Association for Computational Linguistics: EMNLP 2023. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 9210–9232. DOI: 10.18653/v1/2023.findings-emnlp.618. URL: https://aclanthology.org/2023.findings-emnlp.618.

[41] Wen-Ding Li and Kevin Ellis. "Is Programming by Example solved by LLMs?" In: 2024. URL: https://api.semanticscholar.org/CorpusID:270391839.

[42] Justin Lubin et al. "Program sketching with live bidirectional evaluation". In: Proc. ACM Program. Lang. 4.ICFP (Aug. 2020). DOI: 10.1145/3408991. URL: https://doi.org/10.1145/3408991.

[43] Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. "Estimating the carbon footprint of BLOOM, a 176B parameter language model". In: J. Mach. Learn. Res. 24.1 (Mar. 2024). ISSN: 1532-4435.

[44] Joshua Maynez et al. "On Faithfulness and Factuality in Abstractive Summarization". In: ed. by Dan Jurafsky et al. Online: Association for Computational Linguistics, July 2020, pp. 1906–1919. DOI: 10.18653/v1/2020.acl-main.173. URL: https://aclanthology.org/2020.acl-main.173.

[45] Tomas Mikolov et al. "Recurrent neural network based language model". In: Interspeech. 2010. URL: https://api.semanticscholar.org/CorpusID:17048224.

[46] Anders Miltner et al. "Bottom-up synthesis of recursive functional programs using angelic execution". In: Proc. ACM Program. Lang. 6.POPL (Jan. 2022). DOI: 10.1145/3498682. URL: https://doi.org/10.1145/3498682.

[47] Ministry of Economy, Trade and Industry (METI). Interim Report by the Conference toward AI Network Society. METI Press Release. Artificial Intelligence Governance in Japan Ver. 1.0: Interim Report by the Conference toward AI Network Society. 2022. URL: https://www.meti.go.jp/english/press/2022/0128_003.html.

[48] Stephen Muggleton. "Duce, an oracle-based approach to constructive induction". In: Proceedings of the 10th International Joint Conference on Artificial Intelligence - Volume 1. IJCAI'87. Milan, Italy: Morgan Kaufmann Publishers Inc., 1987, pp. 287–292.

[49] Stephen Muggleton. "Inductive Logic Programming: Issues, results and the challenge of Learning Language in Logic". In: Artificial Intelligence 114.1 (1999), pp. 283–296. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(99)00067-3. URL: https://www.sciencedirect.com/science/article/pii/S0004370299000673.

[50] Stephen Muggleton. "Inverse entailment and progol". In: New Generation Computing 13.3 (Dec. 1995), pp. 245–286. ISSN: 1882-7055. DOI: 10.1007/BF03037227. URL: https://doi.org/10.1007/BF03037227.

[51] STEPHEN MUGGLETON and WRAY BUNTINE. "Machine Invention of First-order Predicates by Inverting Resolution". In: Machine Learning Proceedings 1988. Ed. by John Laird. San Francisco (CA): Morgan Kaufmann, 1988, pp. 339–352. ISBN: 978-0-934613-64-4. DOI: https://doi.org/10.1016/B978-0-934613-64-4.50040-2. URL: https://www.sciencedirect.com/science/article/pii/B9780934613644500402.

[52] Stephen H Muggleton et al. "Meta-interpretive learning: application to grammatical inference". In: Machine learning 94 (2014), pp. 25–49.

[53] Stephen H. Muggleton and Cao Feng. "Efficient Induction of Logic Programs". In: International Conference on Algorithmic Learning Theory. 1990. URL: https://api.semanticscholar.org/CorpusID:14992676.

[54] OECD. Skills Matter. 2016, p. 160. DOI: https://doi.org/https://doi.org/10.1787/9789264258051-en. URL: https://www.oecd-ilibrary.org/content/publication/9789264258051-en.

[55] OpenAI et al. GPT-4 Technical Report. 2024. arXiv: 2303.08774 [cs.CL].

[56] Peter-Michael Osera and Steve Zdancewic. "Type-and-example-directed program synthesis". In: SIGPLAN Not. 50.6 (June 2015), pp. 619–630. ISSN: 0362-1340. DOI: 10.1145/2813885.2738007. URL: https://doi.org/10.1145/2813885.2738007.

[57] David Page and Ashwin Srinivasan. "Ilp: a short look back and a longer look forward". In: J. Mach. Learn. Res. 4.null (Dec. 2003), pp. 415–430. ISSN: 1532-4435.

[58] Gordon D. Plotkin. "A Further Note on Inductive Generalization". In: 2008. URL: https://api.semanticscholar.org/CorpusID:8230949.

[59] Gordon D. Plotkin. "A Note on Inductive Generalization". In: 2008. URL: https://api.semanticscholar.org/CorpusID:27884159.

[60] Gordon D. Plotkin. "Automatic Methods of Inductive Inference". In: 1972. URL: https://api.semanticscholar.org/CorpusID:118467251.

[61] J. R. Quinlan. "Learning logical definitions from relations". In: Machine Learning 5.3 (Aug. 1990), pp. 239–266. ISSN: 1573-0565. DOI: 10.1007/BF00117105. URL: https://doi.org/10.1007/BF00117105.

[62] Alec Radford and Karthik Narasimhan. "Improving Language Understanding by Generative Pre-Training". In: 2018. URL: https://api.semanticscholar.org/CorpusID:49313245.

[63] Robi Rahman, David Owen, and Josh You. Tracking Compute-Intensive AI Models. Accessed: 2024-06-15. 2024. URL: https://epochai.org/blog/tracking-compute-intensive-ai-models.

[64] G. Pradeep Reddy, Y. V. Pavan Kumar, and K. Purna Prakash. "Hallucinations in Large Language Models (LLMs)". In: 2024 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream). 2024, pp. 1–6. DOI: 10.1109/eStream61684.2024.10542617.

[65] Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering. 2024. arXiv: 2401.08500 [cs.LG]. URL: https://arxiv.org/abs/2401.08500.

[66] Claude Sammut. "The Origins of Inductive Logic Programming: A Prehistoric Tale". In: (July 1996).

[67] Claude Sammut and Ranan B. Banerji. "LEARNING CONCEPTS BY ASKING QUESTIONS". In: 1998. URL: https://api.semanticscholar.org/CorpusID:16642894.

[68] Siddharth Samsi et al. "From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference". In: 2023 IEEE High Performance Extreme Computing Conference (HPEC). 2023, pp. 1–9. DOI: 10.1109/HPEC58863.2023.10363447.

[69] Jaime Sevilla and Edu Roldán. Training Compute of Frontier AI Models Grows by 4-5x per Year. Accessed: 2024-06-15. 2024. URL: https://epochai.org/blog/training-compute-of-frontier-ai-models-grows-by-4-5x-per-year.

[70] Ehud Y. Shapiro. "Algorithmic Program Debugging". In: 1983. URL: https://api.semanticscholar.org/CorpusID:60699326.

[71] Ehud Y. Shapiro. "An algorithm that infers theories from facts". In: Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 1. IJCAI'81. Vancouver, BC, Canada: Morgan Kaufmann Publishers Inc., 1981, pp. 446–451.

[72] Erfan Shayegani et al. "Survey of Vulnerabilities in Large Language Models Revealed by Adversarial Attacks". In: ArXiv abs/2310.10844 (2023). URL: https://api.semanticscholar.org/CorpusID:264172191.

[73] Ashwin Srinivasan. "The aleph manual". In: (2001).

[74] Jovan Stojkovic et al. "Towards Greener LLMs: Bringing Energy-Efficiency to the Forefront of LLM Inference". In: ArXiv abs/2403.20306 (2024). URL: https://api.semanticscholar.org/CorpusID:268793445.

[75] Phillip D. Summers. "A Methodology for LISP Program Construction from Examples". In: J. ACM 24.1 (Jan. 1977), pp. 161–175. ISSN: 0004-5411. DOI: 10.1145/321992.322002. URL: https://doi.org/10.1145/321992.322002.

[76] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. 2014. arXiv: 1409.3215 [cs.CL].

[77] The White House. The White House Briefing Room. Executive Order on the Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence. 2023. URL: https://www.whitehouse.gov/briefing-room/presidential-actions/2023/10/30/executive-order-on-the-safe-secure-and-trustworthy-development-and-use-of-artificial-intelligence/.

[78] Ashish Vaswani et al. Attention Is All You Need. 2023. arXiv: 1706.03762 [cs.CL].

[79] Steven A. Vere. "Induction of Concepts in the Predicate Calculus". In: International Joint Conference on Artificial Intelligence. 1975. URL: https://api.semanticscholar.org/CorpusID:31594645.

[80] Jason Wei et al. Emergent Abilities of Large Language Models. 2022. arXiv: 2206.07682 [cs.CL]. URL: https://arxiv.org/abs/2206.07682.

[81] Yue Wu et al. AgentKit: Flow Engineering with Graphs, not Coding. 2024. arXiv: 2404.11483 [cs.AI]. URL: https://arxiv.org/abs/2404.11483.

[82] Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. "Trace-Guided Inductive Synthesis of Recursive Functional Programs". In: Proc. ACM Program. Lang. 7.PLDI (June 2023). DOI: 10.1145/3591255. URL: https://doi.org/10.1145/3591255.

[83] Zheng Zhang, Levent Yilmaz, and Bo Liu. "A Critical Review of Inductive Logic Programming Techniques for Explainable AI". In: IEEE Transactions on Neural Networks and Learning Systems (2023), pp. 1–17. DOI: 10.1109/TNNLS.2023.3246980.

APPENDIX A
Additional Figures

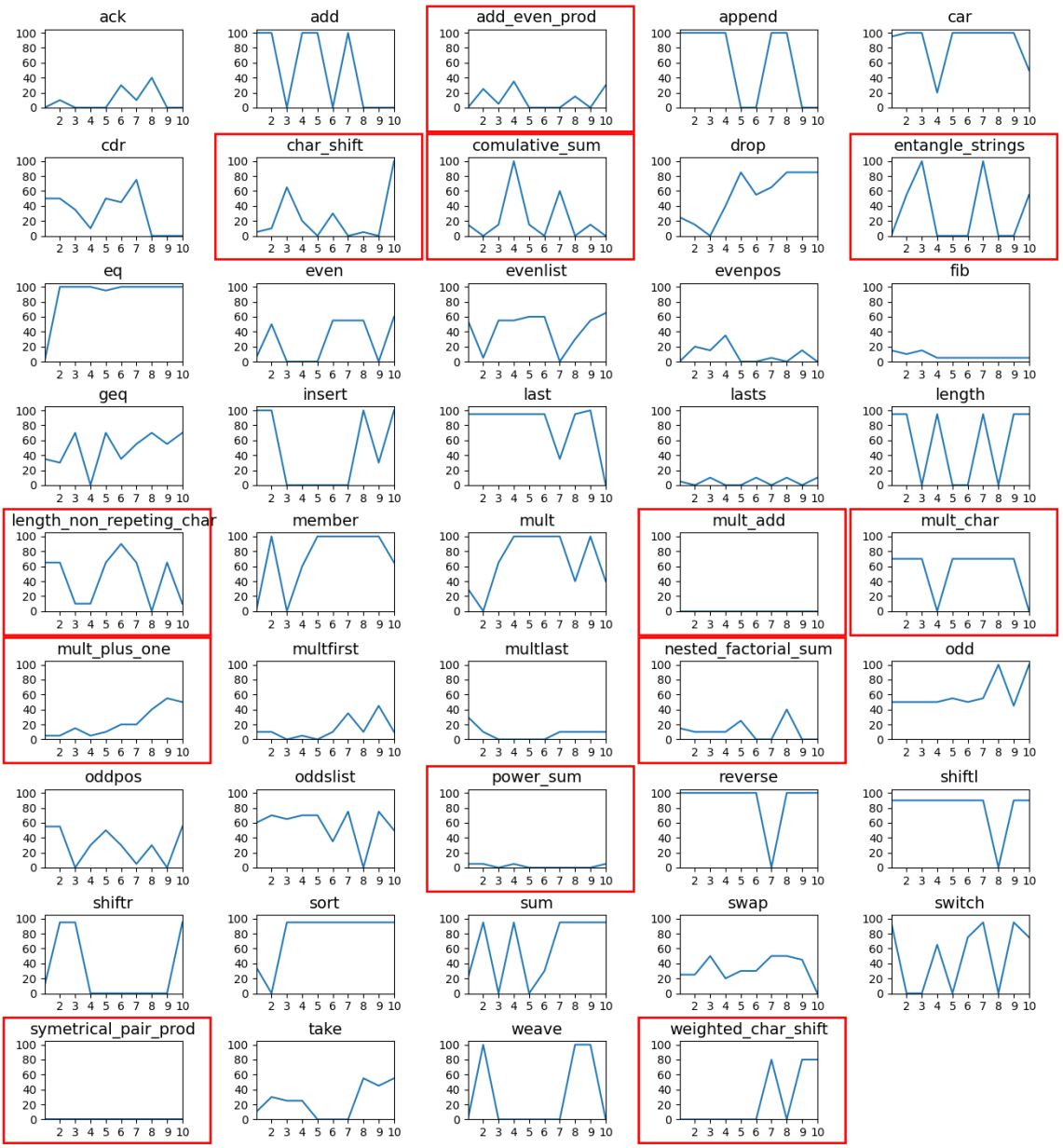

Figure A.1: GPT-3.5 scores for each function in Python using a simple prompt. Red rectangles highlight the complex problems.

Figure A.2: GPT-3.5 scores for each function in Haskell using a simple prompt. Red rectangles highlight the complex problems.

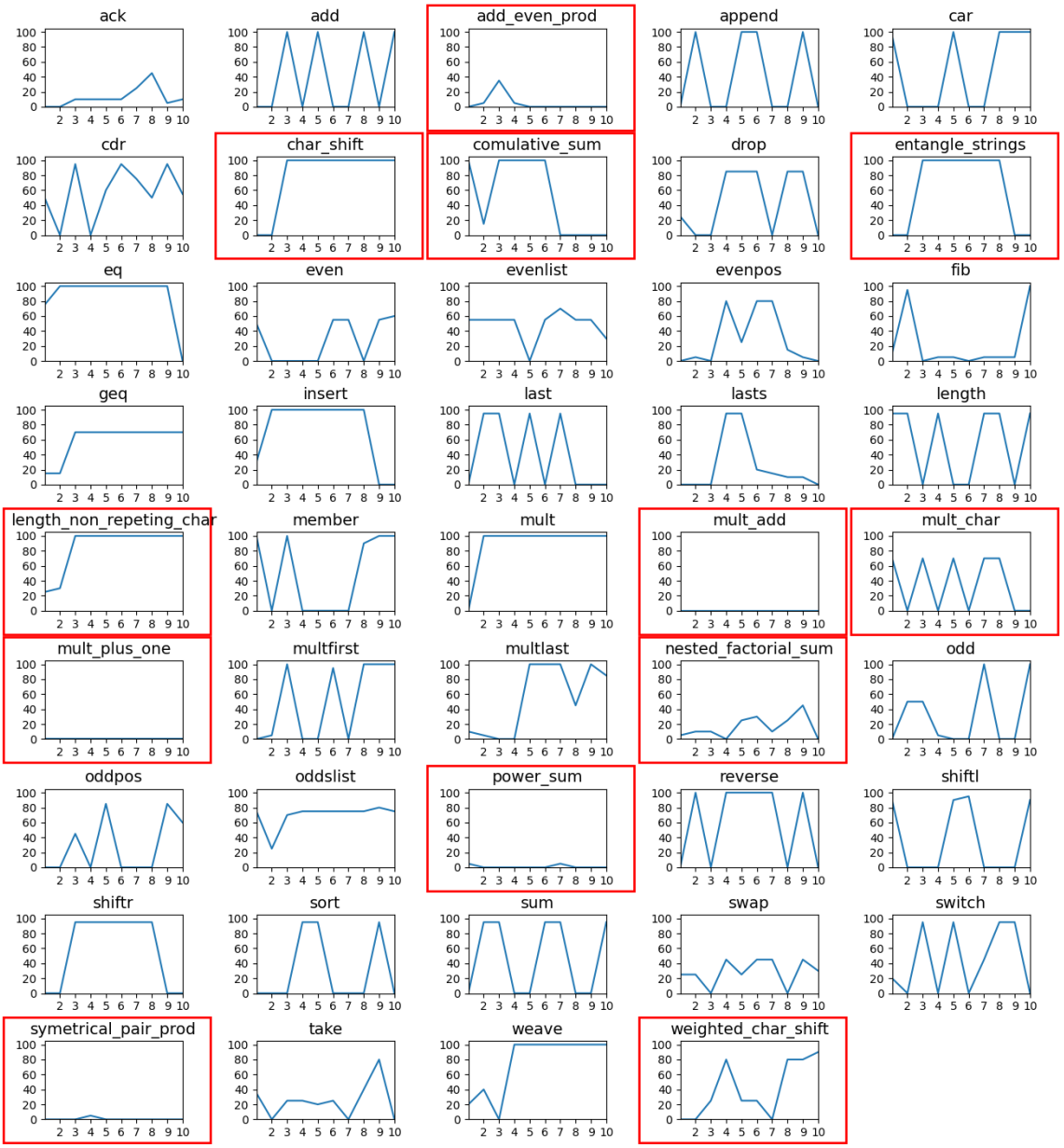

Figure A.3: GPT-4 scores for each function in Python using a simple prompt. Red rectangles highlight the complex problems.

Figure A.4: GPT-4 scores for each function in Haskell using a simple prompt. Red rectangles highlight the complex problems.

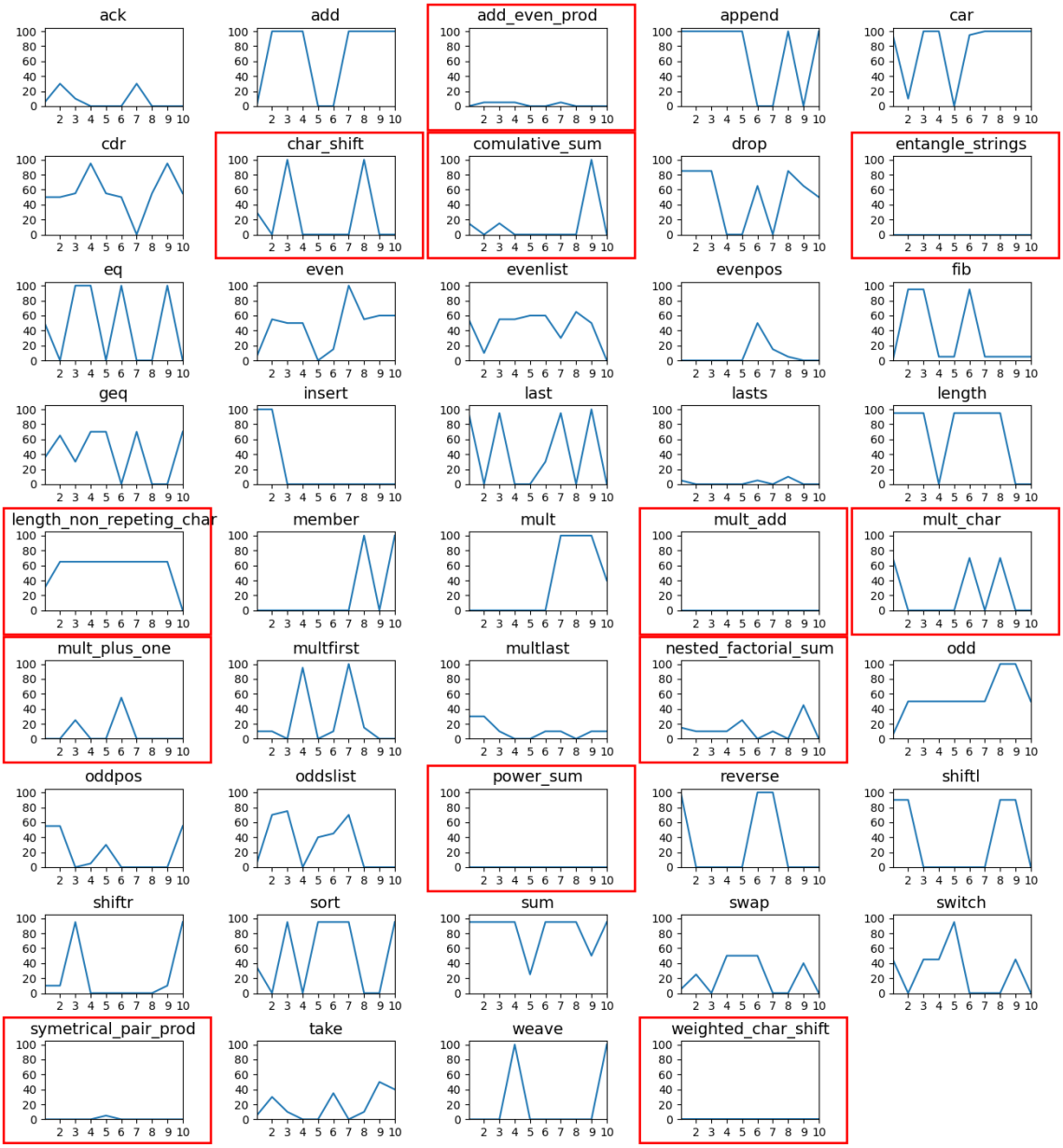

Figure A.5: GPT-3.5 scores for each function in Python using a more complex prompt that defines the desired code structure. Red rectangles highlight the complex problems.

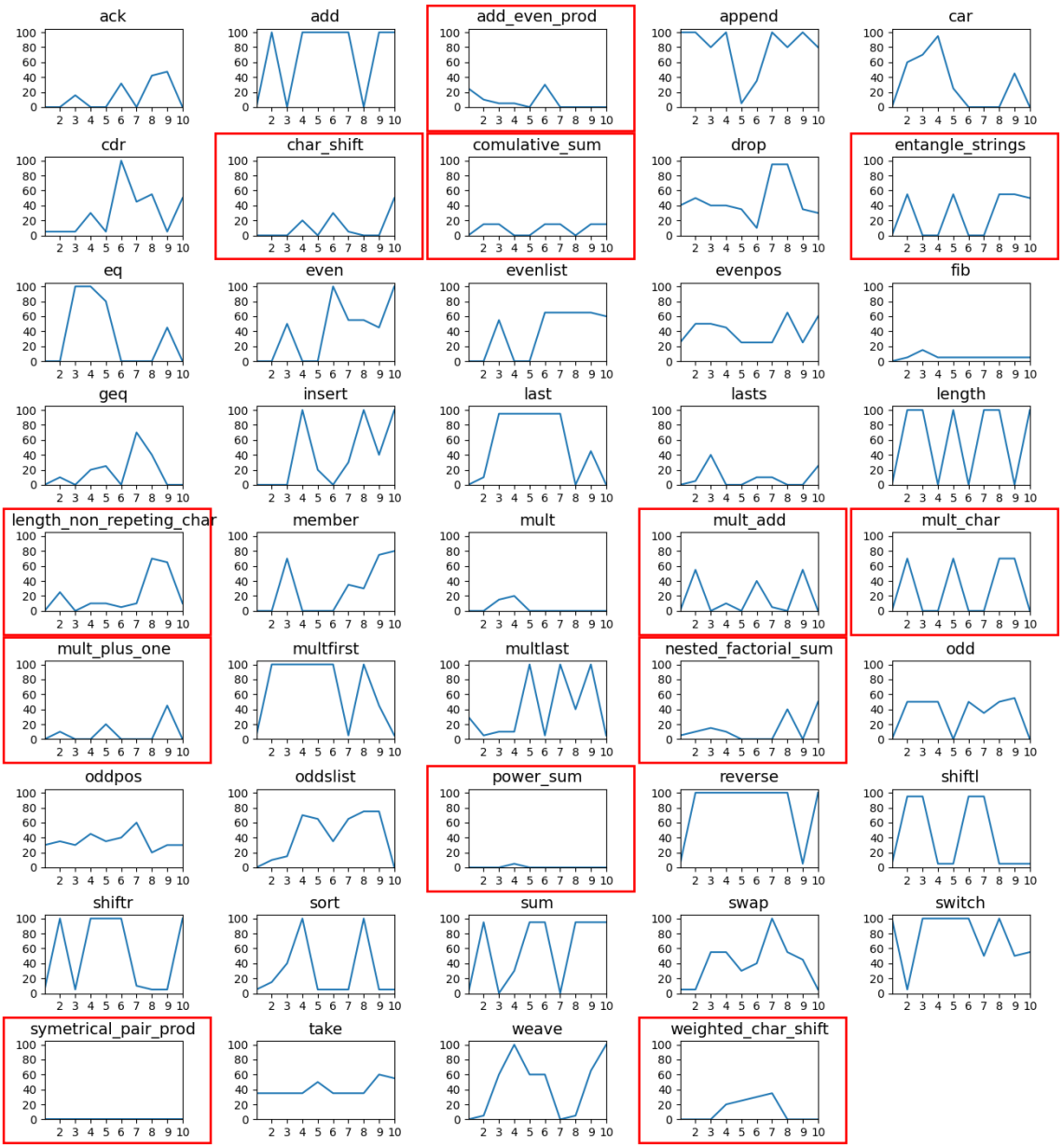

Figure A.6: GPT-3.5 scores for each function in Haskell using a more complex prompt that defines the desired code structure. Red rectangles highlight the complex problems.

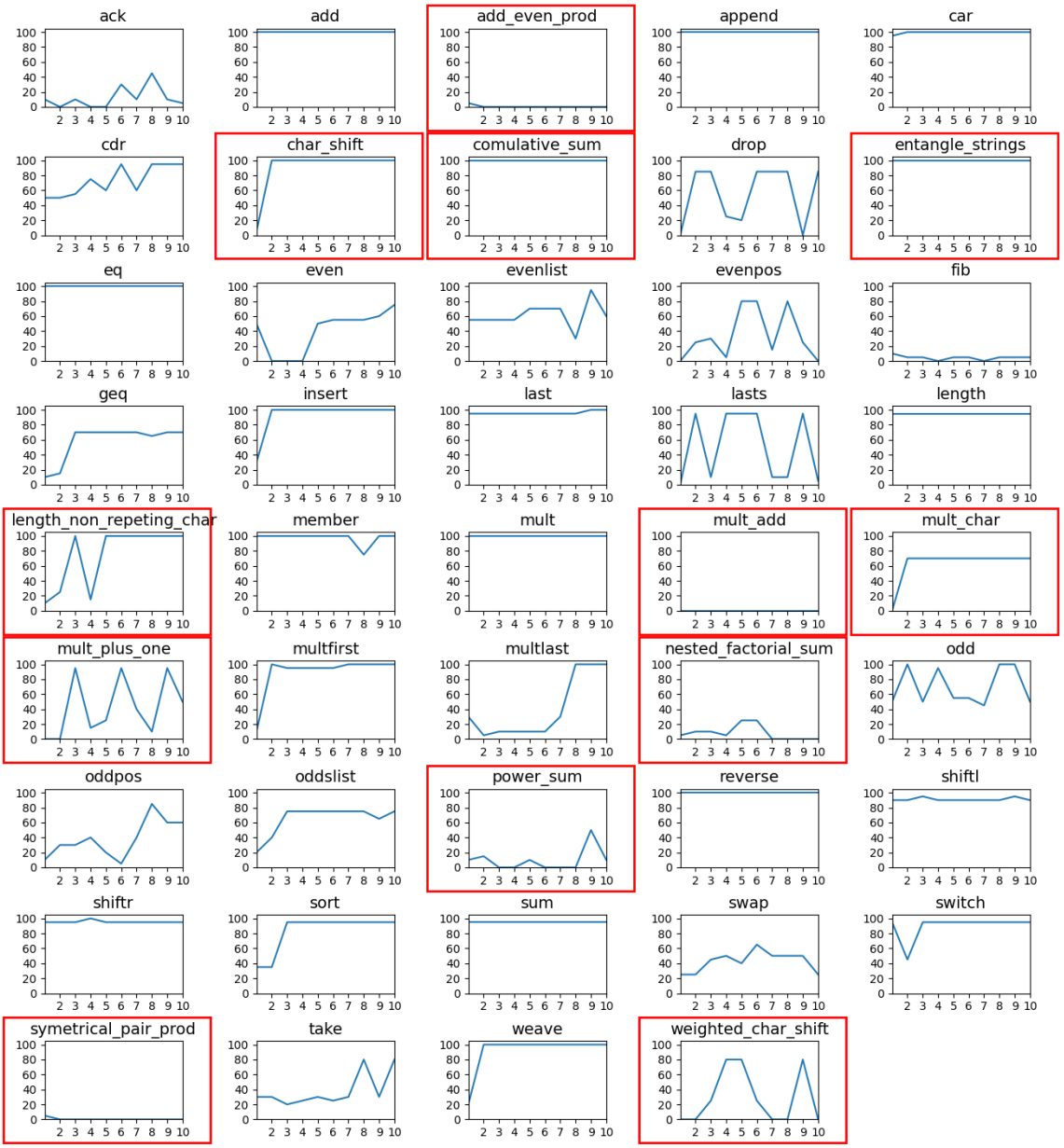

Figure A.7: GPT-4 scores for each function in Python using a more complex prompt that defines the desired code structure. Red rectangles highlight the complex problems.

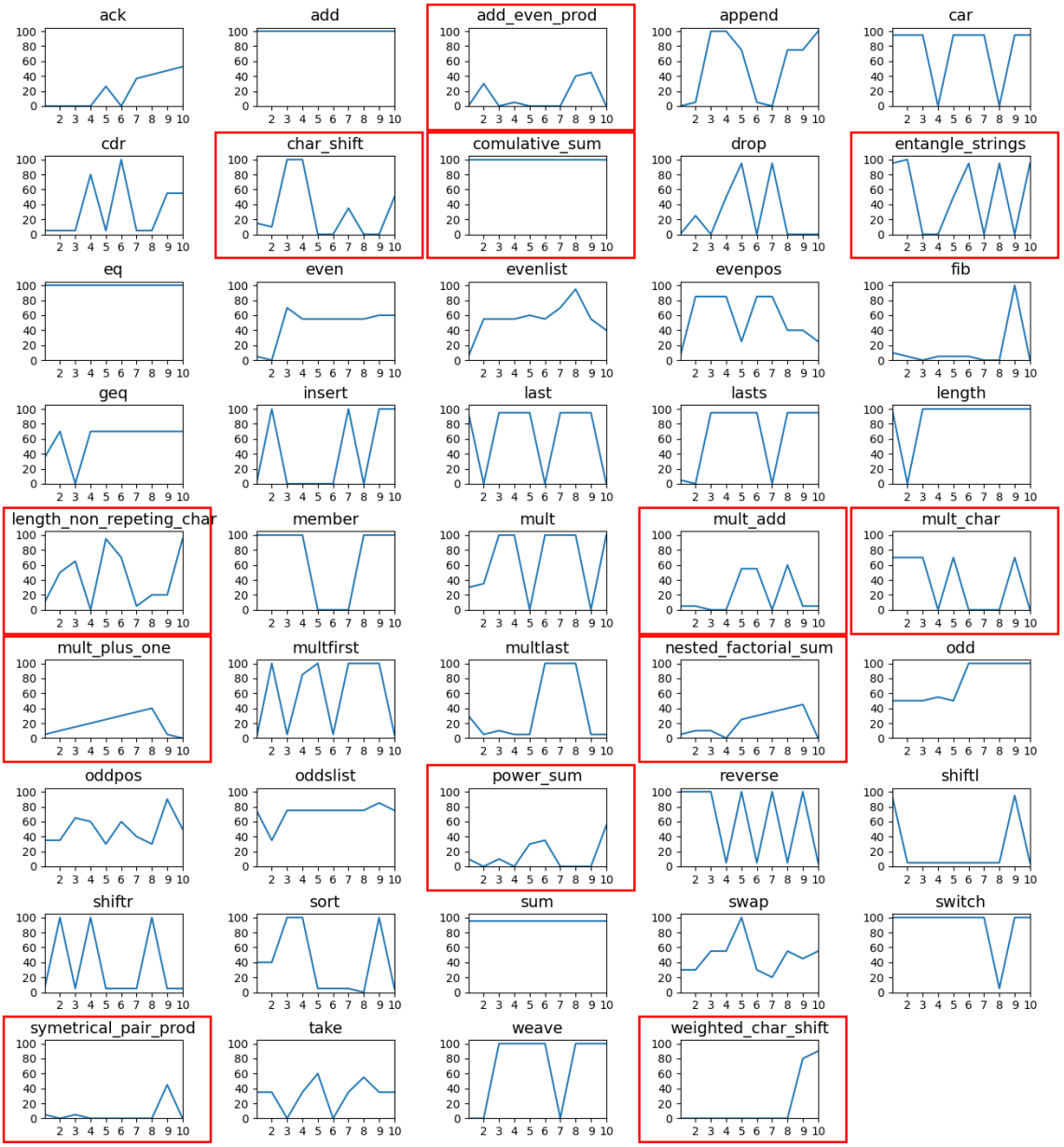

Figure A.8: GPT-4 scores for each function in Haskell using a more complex prompt that defines the desired code structure. Red rectangles highlight the complex problems.

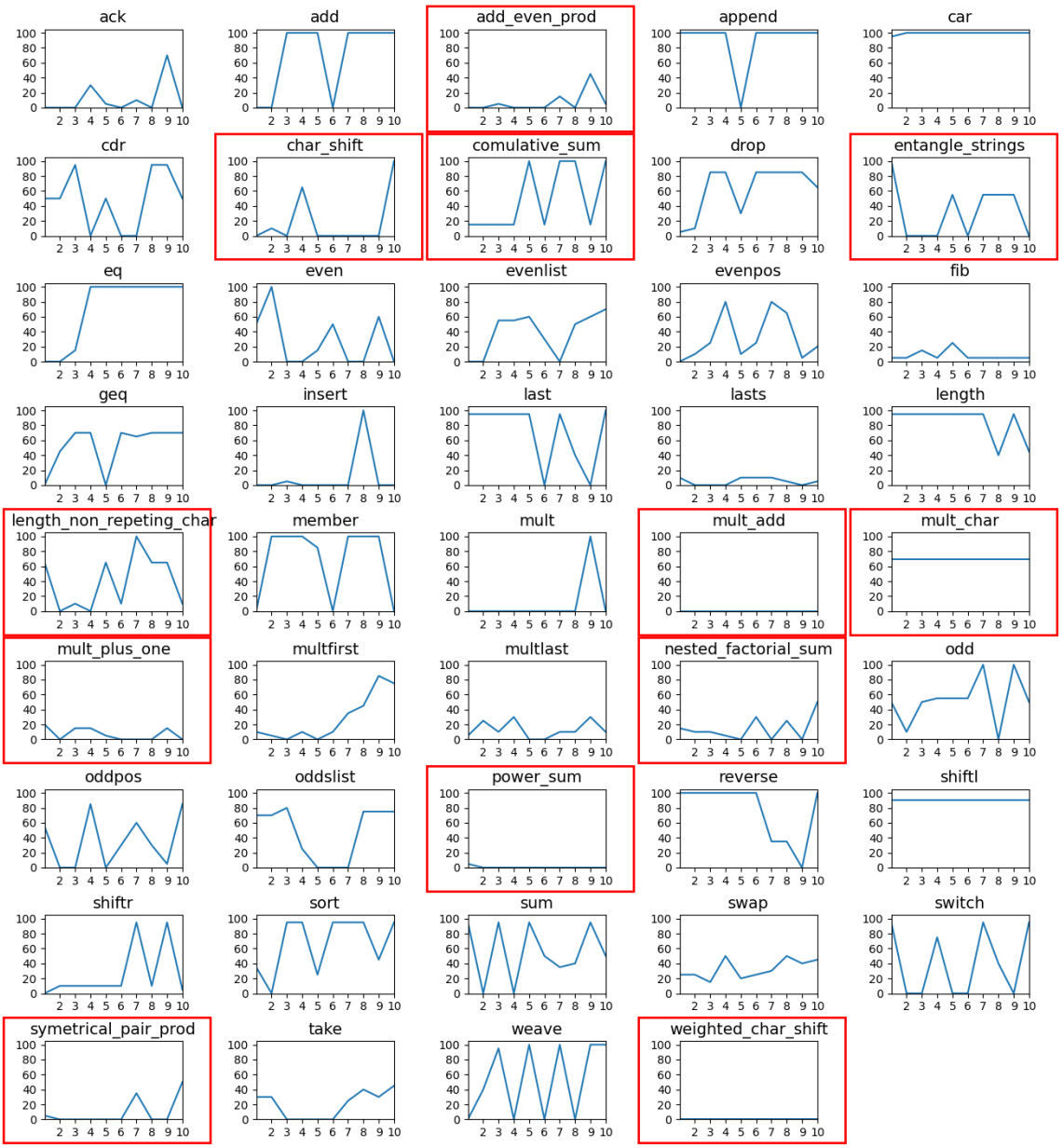

Figure A.9: GPT-3.5 scores for each function in Python using Chain of Thought in addition to defining the desired code structure. Red rectangles highlight the complex problems.

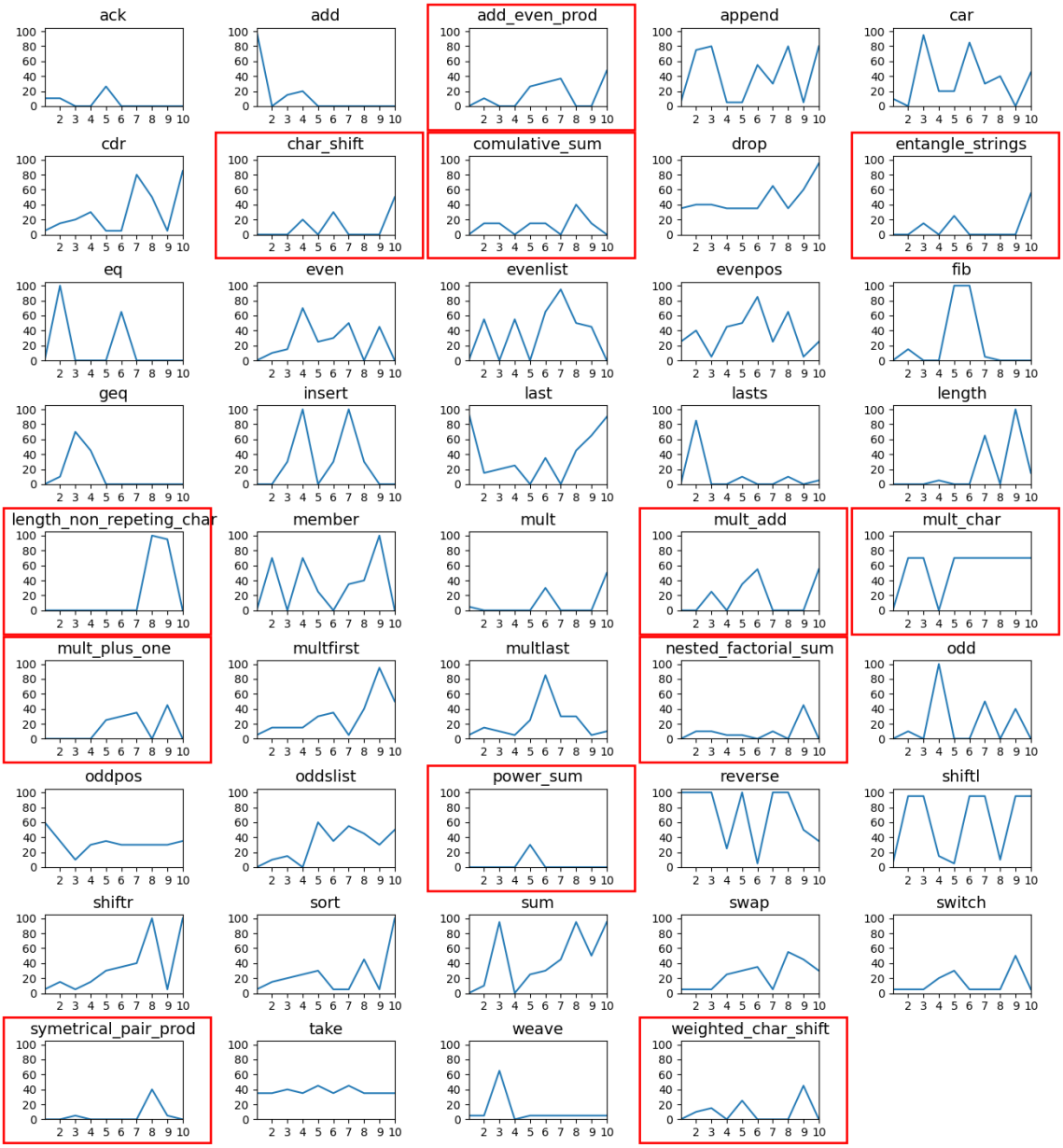

Figure A.10: GPT-3.5 scores for each function in Haskell using Chain of Thought in addition to defining the desired code structure. Red rectangles highlight the complex problems.

Figure A.11: GPT-4 scores for each function in Python using Chain of Thought in addition to defining the desired code structure. Red rectangles highlight the complex problems.

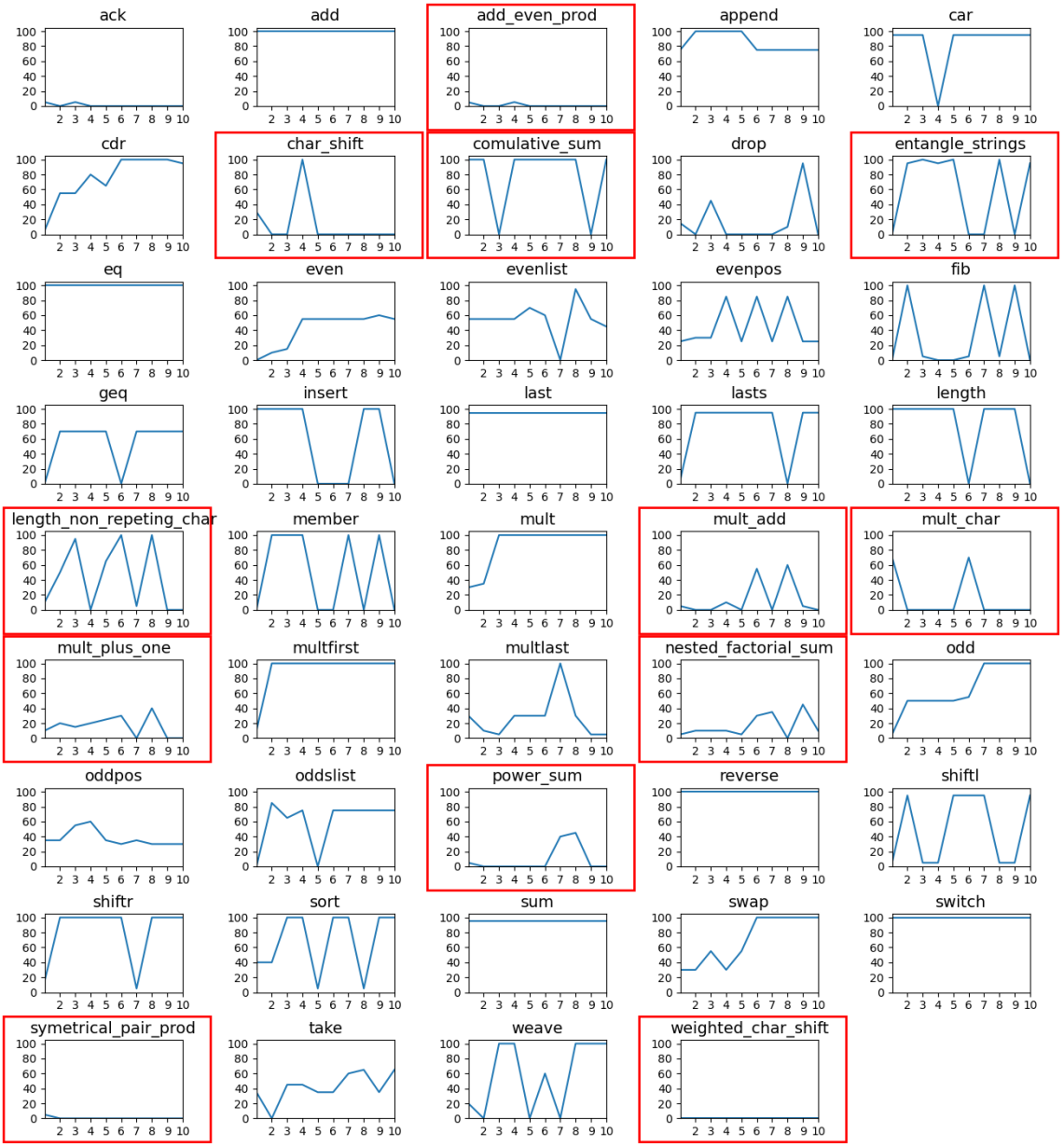

Figure A.12: GPT-4 scores for each function in Haskell using Chain of Thought in addition to defining the desired code structure. Red rectangles highlight the complex problems.

APPENDIX B
Sustainable Development Goals

Sustainable Development Goals	High	Medium	Low	Not Applicable
No Poverty				X
Zero Hunger				X
Good Health and Well-being				X
Quality Education				X
Gender Equality				X
Clean Water and Sanitation				X
Affordable and Clean Energy				X
Decent Work and Economic Growth	X			
Industry, Innovation, and Infrastructure	X			
Reduced Inequality				X
Sustainable Cities and Communities				X
Responsible Consumption and Production				X
Climate Action				X
Life Below Water				X
Life on Land				X
Peace, Justice, and Strong Institutions				X
Partnerships for the Goals				X

Table B.1: Relationship of the work with the Sustainable Development Goals.

The digital era has seen a surge in demand for software solutions, with a significant shift towards no-code platforms that democratise software development. This democratisation is critical to fostering inclusive economic growth and innovation, which is closely aligned with the United Nations Sustainable Development Goals, particularly in the areas of decent work and economic growth, and industry, innovation and infrastructure.

Traditional software development often requires extensive programming skills, which can be a barrier for many. By enabling users to create applications without deep technical knowledge, no-code platforms and tools such as Programming-by-Examples (PBE) expand opportunities for a wide range of people. This inclusivity helps to integrate non-traditional developers into the technology workforce, promoting decent work and economic opportunity for a wider range of people. For example, individuals in developing regions or from non-technical backgrounds can now participate in the digital economy and contribute to innovation and entrepreneurship without the steep learning curve associated with traditional programming.

The application of Large Language Models (LLMs) to PBE has the potential to significantly increase productivity. LLMs, which can generate code from natural language

examples, streamline the development process by automating repetitive and complex coding tasks. This automation can lead to shorter development cycles, enabling companies to innovate faster and more efficiently. By reducing the time and expertise required to develop software, LLMs enable a more dynamic and responsive approach to market needs, driving economic growth through accelerated innovation.

Large Language Models (LLMs) are increasingly recognised as one of the technological innovations of our time, and their use is spreading from specialist applications to mainstream users in a variety of domains. By rigorously evaluating and understanding the performance of LLMs, particularly in the context of Programming-by-Examples (PBE), we gain valuable insights into their capabilities and limitations. This rigorous evaluation not only improves our understanding of LLMs, but also drives further progress and innovation. By refining these systems through continuous research and development, we can maximise their potential to transform software development, making it more accessible, efficient and adaptable to different needs. Such advances ensure that LLMs remain at the forefront of technological innovation, contributing to the wider goals of sustainable economic growth and robust infrastructure development.

The development of intelligent programming tools such as LLMs for PBE supports the creation of a robust digital infrastructure. By helping to generate code from examples, LLMs contribute to the foundation of a more accessible and scalable digital ecosystem. This advancement supports the development of interoperable and efficient systems, which are critical components of modern digital infrastructure. By simplifying software development, LLMs help bridge the gap between technology and end-users, making sophisticated digital tools and services more accessible.

The exploration of Large Language Models for Programming-by-Examples is an important step towards achieving the Sustainable Development Goals of promoting decent work and economic growth, building resilient infrastructure, fostering innovation and sustainable industrialisation. By lowering barriers to software development, increasing productivity and supporting the creation of robust digital infrastructure, these technologies play a key role in fostering a more inclusive and innovative digital economy. As we continue to integrate LLMs into the programming landscape, their potential to democratise software development and drive sustainable growth will be instrumental in shaping the future of work and industry.

APPENDIX C
Glossary

Word	Definition
Benchmark	A standard or point of reference against which things may be compared or assessed.
PBE (Programming by Examples)	A programming paradigm where the system generates a program based on a set of input-output examples provided by the user.
IP (Inductive Programming)	A type of programming that involves learning general rules and patterns from specific examples or observations.
NLP (Natural Language Processing)	A field of AI focused on the interaction between computers and humans through natural language, enabling machines to understand, interpret, and generate human language.
LLM (Large Language Models)	Advanced machine learning models that are trained on vast amounts of text data to understand, generate, and manipulate human language.
UML	Unified Modeling Language, a standardized modeling language used to specify, visualize, construct, and document the artifacts of software systems.
IPS (Inductive Program Synthesis)	The process of automatically generating programs from examples and specifications through inductive reasoning.
LISP	A family of programming languages, known for their simple syntax and powerful features, widely used in artificial intelligence research.
Haskell	A standardized, general-purpose purely functional programming language with strong static typing and lazy evaluation.
ML	A general-purpose functional programming language known for its type inference and pattern matching. Often used as shorthand for "Machine Learning" as well.
OCaml	A functional programming language that extends the Caml dialect with object-oriented and imperative features.
Prolog	A logic programming language associated with artificial intelligence and computational linguistics, based on formal logic.
Python	A high-level, interpreted programming language known for its readability, simplicity, and wide applicability in various domains.

Continued on next page

Word	Definition
ASP (Answer Set Programming)	A form of declarative programming oriented towards difficult combinatorial search problems, based on the stable model (answer set) semantics of logic programming.
Meta-interpreter	A program that interprets another program, allowing for dynamic interpretation of different programming languages or paradigms.
Curse of dimensionality	A phenomenon where the difficulty of analysis and computation increases with the number of dimensions in the data, often leading to issues with overfitting and data sparsity.
API	Application Programming Interface, a set of protocols and tools for building and interacting with software applications, enabling communication between different software systems.
Prompt	A directive given to a model or system to perform a task, often used in the context of querying or guiding large language models.
FLOPS	Floating-Point Operations Per Second, a measure of a computer's performance, especially in fields requiring high computational power like simulations and scientific calculations.
GPT	Generative Pre-trained Transformer, a type of AI model developed by OpenAI, known for its capability to generate human-like text based on pre-training on a large corpus of text data.
GPT-3.5	A version of OpenAI's Generative Pre-trained Transformer model that offers advanced language understanding and generation capabilities, a step between GPT-3 and GPT-4.
GPT-4	The fourth generation of OpenAI's Generative Pre-trained Transformer, offering improved performance, context understanding, and generation abilities compared to its predecessors.