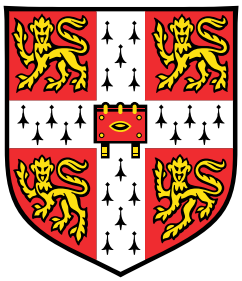
Function Constrained Program Synthesis



Patrick Hajali

Supervisor: Dr.Ignas Budvytis

Department of Engineering
University of Cambridge
2023 MLMI Cohort

This dissertation is submitted for the degree of
Master of Philosophy

Pembroke College August 2023

Declaration

I, Patrick Hajali of Pembroke College, being a candidate for the MPhil in Machine Learning and Machine Intelligence, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

This thesis makes use of the following open-source software:

- The HumanEval dataset and evaluation pipeline provided by OpenAI at github.com/openai/human-eval.
- The APPS dataset and evaluation pipeline, accessed at github.com/hendrycks/apps.
- OpenAI API calls to `gpt-4`, `gpt-3.5-turbo`, and `gpt-3.5-turb-16k`, along with the web-based ChatGPT Premium Membership (for access to GPT-4 and plug-ins), accessed at chat.openai.com.

Word-count: 14436

Patrick Hajali
August 2023

Abstract

Large language models (LMs) like GPT-4, GPT-3.5, and BART have demonstrated strong ability in comprehending, rectifying, and producing code, making them a powerful tool in the realm of automated programming. However, they face significant challenges when it comes to multi-step reasoning inherent in constructing intricate algorithms. While human developers excel in this domain by employing hierarchical decomposition - a method that divides complex tasks into more digestible sub-problems - LMs falter as they typically generate code in a linear sequence without such decomposition. After decomposing a complex-algorithm, humans implement modular sub-solutions to each task before weaving them back into a cohesive, overarching solution. A benefit of this approach is the creation of a collection of sub-functions which can be used in the development of future programs. Our goal is to integrate this strategy within the context of LM-guided program synthesis. A key requirement in doing this is to encourage LMs to use unseen (during training) functions provided in-context. The challenge, then, is to guide LMs in synthesising complex-algorithms using user-provided functions, while also promoting the generation of adaptable sub-functions as a natural by-product.

Current solutions, while evolving, present a host of limitations. Classical program synthesis approaches, like "library learning", lack the flexibility of generating programs in general-purpose languages and rely on strict formulations of the target algorithm, rather than a simple natural language specification. Although contemporary coding assistants, such as GitHub Copilot, are designed leverage code provided in-context, they lack autonomy and are unable to be restricted to a precise set of user-defined functions. Self-planning and decomposition-focused methods, which strive for structured code-generation, don't handle failures effectively and often don't accommodate user constraints.

Addressing these challenges, we introduce a new approach. We first constrains a language model to use only code provided in-context by the user. The constrained model is prompted to generate a target algorithm, and if the constrained LM fails to generate working code, we provide it with a modular sub-function, which is intended aid its future attempts at generating working code. We show that sub-function generation can be automated by

leveraging LMs, creating an autonomous end-to-end system. Altogether, the end result is enhanced integration of code provided in-context and improved coding capabilities, all while adhering to user-defined constraints.

Our findings demonstrate the potential of this approach. When constraining GPT-4 and GPT-3.5 to use only a set of 21 handwritten functions, our method achieved 81.2% and 72.5% pass@1 accuracy on the HumanEval dataset, respectively. Moreover, our generated functions displayed remarkable adaptability across varying tasks and even datasets. Additional assessments highlighted LMs' challenges in conforming to constraints, further emphasizing the usefulness of our method. In this work, we also demonstrate how GPT-4 can achieve 85.4% pass@1 accuracy on HumanEval (only 6% below current state-of-the-art) simply by prompting for structured output formatting, underscoring the need for "half-shot" evaluation to better gauge LMs' true coding capabilities.

In summation, this work presents an effective method to leverage the intelligence of pretrained LMs in program synthesis tasks constrained to using user-provided code.

Table of contents

Chapter 1

Introduction

In this thesis, our objective is to synthesise programs using code provided in-context. Specifically, given a set of functions and a high-level description of an algorithm, our goal is to automatically generate a program that implements this algorithm solely using the given functions and the fundamental operations inherent to the general-purpose programming language, such as Python, in which the program is to be generated. Concurrently, as part of our solution, we aim to produce modular, reusable code–in the form of sub-functions–that generalises to future tasks, rather than one-off solutions.

1.1 Motivation

Algorithm development often requires the breaking down of complex logic into functions. These functions play a pivotal role in promoting code modularity and reusability, ensuring efficient problem-solving while minimizing redundancy and decreasing the potential for bugs to be introduced.

Recent advances in large language models (LMs) such as ChatGPT (GPT-3.5) ([15]), GPT-4 ([47]), and Claude-2 ([4]) show promising capabilities for code comprehension, correction, and generation (Li et al. [42], Bubeck et al. [17]), but are limited in their ability implement complex, compositional programs (Dziri et al. [24]). One reason for this limitation is their inability to implement and reuse functions. To make use of unseen (during training) functions without retraining a LM, the functions must be provided within its immediate context.

Encouragingly, LMs' have demonstrated capabilities in using "tools", or API-calls, provided in-context (Schick et al. [55], Wu et al. [67], Shen et al. [57], Liang et al. [43]). This capability is closely analogous to generating calls to *functions* provided in-context. Moreover, recent developments have allowed for increased size of LMs' context windows (Chen et al.

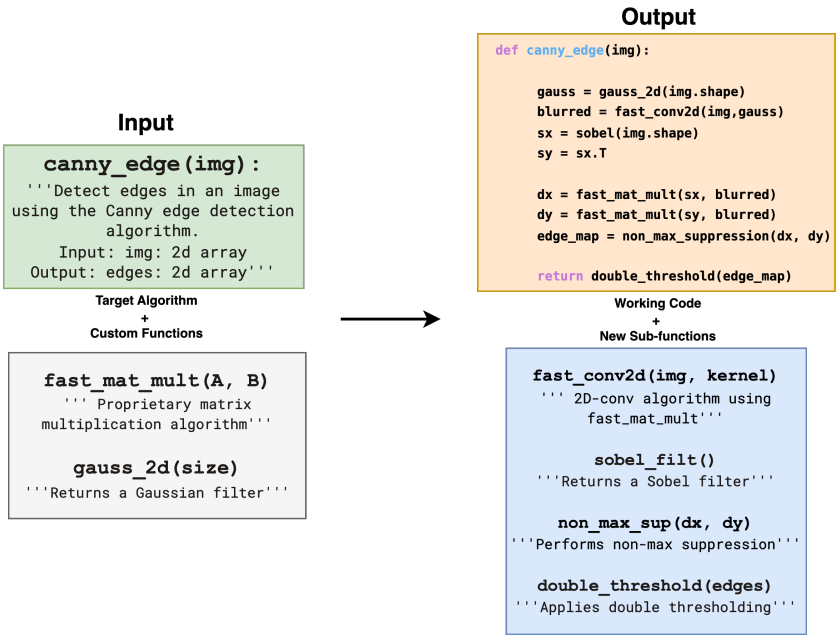


Fig. 1.1 **Example Input/Output of the Desired System.** The user provides a target algorithm description and custom functions. The system generates a solution using only the supplied functions, while also producing reusable sub-functions that are integrated into the output code and can be used for future tasks.

[20], Bertsch et al. [10], Martins et al. [46]). As a result, a large set of functions can be accommodated and presented in-context.

This motivates the development of a method wherein LMs are not only primed to use functions provided in-context but also engineered to yield reusable functions during their program synthesis attempts. There are many practical benefits to this system. For example, when a company aims to integrate a proprietary algorithm into its existing infrastructure, manual integration can be resource-intensive and prone to errors. In contrast, leveraging the proposed system offers an automated integration procedure.

1.2 Challenges

Designing a system that generates usable code while constrained to using user-specified functions presents several distinct challenges.

- A primary challenge is **ensuring that the system can handle scenarios where it is initially unable to formulate the desired algorithm using the given functions.** Specifically, there must be an effective mechanism to manage failures that remains consistent

with the user-imposed constraints. A simplistic approach, such as elevating the temperature or incorporating the error trace as input, might inadvertently cause the model to stray from these constraints.

- Another significant challenge is **ensuring the model can aptly use a user-provided function—even when encountering it for the first time—in context**, rather than reverting to similar functions with which it is more exposed to in training. The efficacy of this depends on factors like the model's size, its training set, and numerous other unpredictable variables. This consideration is paramount, especially considering that language models are trained on data sourced from the internet, inherently embedding them with human biases (Bender et al. [9], Weidinger et al. [65]). Such biases, when translated to code-generation, might manifest as a predisposition towards using certain functions, even if they're outdated or not preferred by the user.

- Key challenges for LMs also lie in **long-term planning and task decomposition** (Creswell and Shanahan [22]). Planning over a lengthy history and effectively exploring the solution space remain challenging for LMs, who are prone to hallucinations. LMs struggle to adjust plans when faced with unexpected errors, making them less robust compared to humans who learn from trial and error (Bang et al. [7]).

- Even if the LM successfully adheres to the imposed constraints and generates functional code, a challenge remains: **ensuring that the output is presented in a usable format** without requiring human intervention. Pre-trained LMs are fine-tuned to produce outputs in specific, non-standard formats. If these formats are not anticipated, it can lead to inadvertent errors, as demonstrated by Chen et al. [18].

1.3 Current Solutions

With these challenges in mind, recent works have proposed various solutions aimed at overcoming these hurdles.

Approaches leveraging classical program synthesis methods have introduced the concept of "library learning", exploring deductive algorithms which extract reusable components from an existing corpus of programs (Ellis et al. [25], Bowers et al. [13]). While these works produce reusable function libraries as a byproduct of program synthesis, they have some limitations inherent to classical synthesis methods. First, library learning relies on domain-specific languages (DSLs) rather than general-purpose code, like Python. This limits the generalisability of their solutions. Second, these approaches require a heavy amount of

input-output examples to properly specify the desired functions, whereas LMs can generate code from more flexible natural language specifications possibly combined with a few input-output examples.

Currently, LMs are often leveraged in code-generating tasks as coding "assistants" that suggest and debug lines of code in an auto-regressive manner. This approach (e.g., GitHub Copilot [30] and Amazon Codewhisperer [3]) invokes LMs to follow a limited range of context and produces code snippets in real-time to aid programmers. A coding-assistant, however, is designed to run alongside a human-programmer, and cannot be prompted to solve a number of problems on its own. Additionally, although this method often makes use of user-provided code, it does not allow the user to explicitly define which functions the LM can and cannot use.

In a more autonomous fashion, “agents", including AutoGPT [59], BabyAGI [69], GPTEngineer [5], and Aider [28], pose as a potential method to allow LMs to autonomously and iteratively construct complex programs, given simple user specifications. These agents can be initialised within a code-base and prompted to autonomously edit or generate code given the current context. Currently, however, none of the current methods integrate an effective way to handle cases when the model fails to produce working code. Failures are often handled by feeding the error into the intput and re-prompting–posing potential for divergence away from the constraints by swamping the context. Agent like GPTEngineer [5] avoid this by mandating periodic human intervention, taking away from the full autonomy of these systems. In general, these models have demonstrated unstable behavior due to the compound hallucinations that can occur.

Other attempts to enhance the coding prowess of LMs through novel styles of prompting have been numerous. For example, the self-planning method introduced by Jiang et al. [38], prompts LMs to bifurcate the code-generation process into clear planning and implementation stages as a nod to how human programmers operate. Yet, this method does not result in the creation of modular functions, nor does it have an efficient system to address scenarios where the model struggles to produce working code. It’s also worth noting that while planning aids structure, it isn’t synonymous with the hierarchical breakdown of an algorithm.

In a decomposition-focused approach, Parsel (Zelikman et al. [71]), presents a framework that allows LMs to decompose complex algorithmic reasoning tasks into hierarchical natural language descriptions in the unique "Parsel" language which can then be automatically compiled into executable code using the "Parsel" compiler. Interestingly, this approach generates and validates code for each sub-function within that decomposition, but it remains

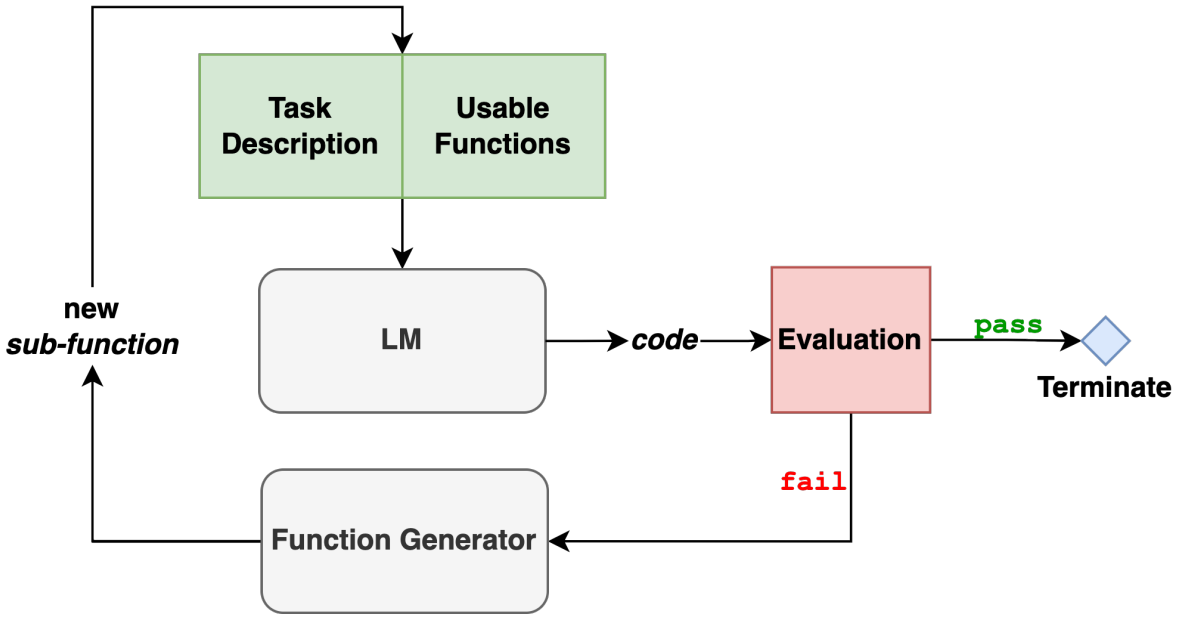


Fig. 1.2 **Approach.** The goal is to implement a target algorithm using only user-provided functions from a function bank along with basic Python operations. We constrain an LM to generate code under this limitation. When the generated code is non-functional, we employ another LM to propose and implement a new sub-function, or sub-skill. This sub-skill is appended to the set of usable functions. If the sub-skill does not adequately address deficiencies, this process repeats to sample a new sub-skills until the constrained model can integrate the new skills to generate working code. The core hypothesis is that supplying finely targeted skills missing from the model's capability will enable it to compose a solution using functions only from the user-defined function library.

limited by the quality of the decomposition initially proposed by the LM. This method also does not provide a way to constrain the model to user-provided code.

1.4 Approach

Our approach distinguishes itself from existing solutions in several key ways. We've expanded on the idea of *library learning*, extending its applicability to general-purpose programming languages, and ensuring a more usable framework for providing task descriptions. Instead of directing the LM to initiate problem decomposition right away, we embed decomposition as an iterative, hard-coded process in our method, allowing for a greater margin of error at each decomposition step. In contrast to platforms like Copilot, our strategy is designed to function autonomously. Crucially, we achieve all these enhancements without compromising the commitment to only using functions provided in-context.

The key steps of our approach are:

1. We constrain a LM to synthesise programs using only code provided in-context, by the user. For example, a LM may be tasked to produce a Canny-edge detection algorithm given custom `matrix_multiply` and `Gauss_2d` functions.

2. If the "constrained" LM fails to generate working code, we query a separate model to analyse the dysfunctional code and produce a new function, or "sub-skill", using only the given functions. For example, a proposed sub-skill may be `convolve_2d`.

3. The new sub-skill is added to the constrained model's set of valid code, with the intention of helping the constrained model produce working code.

Iterating the steps above provides a simple, yet effective method to solve a given task, while adhering to the user-defined constraints and simultaneously producing a bank of modular, re-usable functions transferable to new problems. Our approach also allows for cost-saving benefits, as a cheaper model can be used in attempts to generate the target algorithm (i.e., step 1) while leveraging the intelligence of a more costly model to propose new sub-skills (i.e., step 2), since producing new sub-skills requires less context than full generation of the target algorithm.

Using our method, we attain 81.2% and 72.5% pass@1 accuracy on the HumanEval [19] dataset when constraining GPT-4 and GPT-3.5 to answer all questions using only a predefined set of handwritten functions designed to emulate behaviors of typical Python Standard Library [50] functions. Furthermore, applying our method under similar constraints allow us to generate working solutions to 17.4% of challenges (232 questions) within a subset of the APPS [35] dataset using GPT-3.5. A notable outcome of our approach is the generation of sub-functions which exhibited utility across different tasks and datasets.

Along the way, we characterise GPT-4 and GPT-3.5's ability to adhere to user-provided constraints, and analyse the effects of proposing a constraint. In specific, we show that GPT-4 and GPT-3.5 perform 12% and 6% worse (pass@1 accuracy) on the HumanEval dataset when constrained to use replicated versions of common Python functions.

Notbably, we demonstrate that GPT-4 achieves 85.4% (+18.3% above baseline and -6% below current SOTA: Reflexion [58]) pass@1 accuracy on HumanEval simply by instructing the model to format its output in a structured way, and then parsing the model output. To address this, we propose a new code-evaluation paradigm defined "half-shot" evaluation, designed to better estimate the true coding-ability of LMs.

1.5 Contributions

The main contributions of this work are:

- Through quantitative experiments, we confirm and characterize the performance degradation when constraining language models to unseen code, even when all skills needed to solve the given tasks are available.
- We develop a method to generate relevant sub-skills for enhancing language model coding performance, without requiring human interference. We compare the abilities of LMs in sub-skill generation tasks to those of humans.
- We demonstrate that providing language models with targeted sub-skills can restore performance on tasks where models fail when constrained to use user-provided code.
- We highlight flaws in the current “zero-shot" method of evaluating language models on coding tasks. Addressing this issue, we propose and utilise a new “half-shot" paradigm aimed at establishing a less biased estimate of models’ coding-capabilities.

A shortened version of this thesis is intended to be submitted to the NeurIPS 2023 “ICBINB" Workshop.

1.6 Organisation of Report

The structure of this report is as follows:

- In Chapter 2, we discuss the foundational aspects of program synthesis and its associated domains. This chapter is included to provide necessary background and an understanding of the research landscape.
- In Chapter 3, we introduce our method constrained program synthesis. We detail the key steps to our solution and introduce metrics to assess skill-quality and evaluate generated code for functional correctness.
- In Chapter 4, we provide details of the experiments discussed in this report.
- In Chapter 5, we present our experiments and results.
- In Chapter 6, we conclude the report and provide a direction for future work.

Chapter 2

Background and Related Work

In this chapter, we discuss the foundational aspects of program synthesis and its associated domains. Our objective here is to provide necessary background for subsequent discussions and establish a contextual understanding of the research landscape.

- In Section 2.1 we introduce the problem of program synthesis. Here, we chart the evolution of traditional solutions and highlight the transition towards viewing program synthesis through the lens of natural language processing.
- In Section 2.2 we pivot to transformer-based methodologies, highlighting their significance in modern program synthesis. We discuss how LLMs code-generation abilities are being improved through prompt engineering, tools, and we also discuss agents.
- In Section 2.3 we introduce the related problem of program induction and discuss its limitations in the context of algorithm development.

2.1 Program Synthesis

Program Synthesis refers to the automated process of discovering programs within a specified programming language that adhere to a given set of constraints, often reflecting user intent (Gulwani et al. [33]).

In this section, we discuss the traditional techniques of program synthesis, and also introduce contemporary approaches that harness neural architectures. We then discuss a new perspective in viewing program synthesis as a natural language generation task, opening the door for program synthesis in general purpose languages.

2.1.1 Classical Methods

While modern methods frequently interpret user intent through natural language, expressing this intent in classical program synthesis techniques is often more complex and less direct (Biermann [12, 11]). One of the foundational approaches is deductive synthesis (Green [32], Manna and Waldinger [45]). This method requires formalising and then proving the existence of a program that satisfies a given specification through the use of formal logic. Following this proof, the program is then extracted. A key challenge with deductive synthesis has been that formulating the precise specification often becomes as labor-intensive and complex as manually writing the program. This led to the development of inductive synthesis methods that leverage inductive specifications, such as input-output examples, to alleviate some of these complexities (Shaw et al. [56]).

More modern approaches based on classical methods have employed probabilistic context-free grammars (PCFG) to generate a program's abstract syntax tree (AST) (Ji et al. [37]). This represents an essential step in modeling the underlying structure of the code. Allamanis et al. [2] utilized PCFG in the context of text-to-code retrieval, while Yin and Neubig [68] extended this concept to text-conditional code generation, highlighting the flexibility and adaptability of this approach in various domains within program synthesis.

Building upon these foundational methods, program synthesis experienced a paradigm shift with the introduction of neural techniques.

2.1.2 Applying Neural Architectures to Enhance Classical Methods

In recent years, the integration of neural architectures with program synthesis has marked a significant advancement in the field. Parisotto et al. [49] introduced a neuro-symbolic program synthesizer, utilizing an LSTM-based neural network to systematically guide the search across the space of lambda calculus expressions. This approach marries traditional symbolic reasoning with neural techniques, offering a more flexible and adaptive search strategy. Similarly, Balog et al. [6] employed a deep learning model to predict the likelihood of specific functions appearing in the source code. This predictive model subsequently informs and guides the search algorithm towards finding a solution, facilitating a more efficient search process.

The classic synthesis methods often necessitate the definition of a domain-specific language (DSL) over which the neural network searches to synthesize a program. In a novel approach, Ellis et al. [25] presented a method for simultaneously learning the DSL and neural search policy. Intriguingly, this methodology enables the model to produce a unique domain-

specific library of code for each task it is trained on, reflecting an adaptive and task-oriented learning process.

2.1.3 Viewing Program Synthesis as a Natural Language Generation Task

The perception of code as a language analogous to natural language has gained traction, leading to the formulation of what is known as the *naturalness hypothesis* (Allamanis et al. [1]). This hypothesis posits that code shares essential characteristics with natural language, and it can be studied and modeled using similar statistical techniques.

Im support of this, Hindle et al. [36] applied the n-gram model to model sequences of code. They provided empirical evidence that demonstrates code's repetitiveness, highlighting its suitability for statistical modeling. The model was expressed as:

$$p(c_1^L) = \prod_{i=1}^{L} p(c_i | c_{i-n+1}^{i-1}) \tag{2.1}$$

Here, c_1^L represents a sequence of code tokens of length L, c_i is the i-th code token, and n refers to the order of the n-gram model.

In the next section, we turn our attention to (no pun intended) a major evolution in language modeling annd synthesis techniques.

2.2 Transformer-Based Approaches

The rise of large-scale transformer-based language models, such as BERT and GPT (Devlin et al. [23], Radford et al. [53]), significantly impacted natural language generation. Leveraging the success of these architectures, similar transformer-based models were applied to program synthesis tasks (Feng et al. [26], Kanade et al. [39], Clement et al. [21]). These models, with their ability to capture long-range dependencies and intricate patterns within data, have shown promise in understanding and generating code.

In this section, we provide an overview of the transformer architecture. We proceed to discuss decoder-only models, with an emphasis on the GPT series which are used in this thesis. Subsequently, we highlight recent strategies to enhance LLMs without fine-tuning, covering prompting methods, in-context tool learning, and autonomous agents.

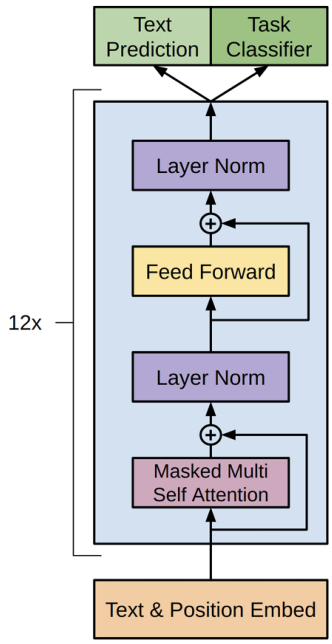


Fig. 2.1 **Decoder-only Transformer Architecture.** The diagram illustrates the decoder-only transformer architecture utilized in GPT models (Radford et al. [52], Brown et al. [16]). (Image credits: Liu et al. [44])

2.2.1 Transformer at a High-Level

The transformer architecture was first introduced by Vaswani et al. [63]. As originally proposed, the transformer is composed of two main components, the encoder and the decoder. The encoder's role is to process the input sequence and compress this information into a continuous representation that captures the relationships between the elements in the sequence. This is achieved through a series of self-attention mechanisms:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.2}$$

where Q, K, and V are the queries, keys, and values, respectively, obtained from a linear-transformation of the input (i.e., $W_k \cdot X$).

The decodes takes this continuous representation and, through a similar series of layers, generates the output sequence. It also employs self-attention, as well as attention over the encoder's output, to generate each element of the output sequence in a context-aware manner.

2.2.2 GPT Models

In this work, we utilize the GPT-3.5 and GPT-4 models developed by OpenAI. The Generative Pre-trained Transformer (GPT) was introduced by Radford et al. [52]. This model leveraged a decoder-only transformer architecture, as initially demonstrated by Liu et al. [44].

- **GPT-3:** GPT-3 Brown et al. [16] implemented significantly more parameters and introduced new training techniques, enabling more accurate and versatile language modeling. Codex (Chen et al. [19]) was developed by fine-tuning this model on code-completions.
- **GPT-3.5:** The Reinforcement Learning from Human Feedback (RLHF) method described by Ziegler et al. [73] was used to further develop GPT-3 by incorporating human feedback to fine-tune the models to behave in chat-scenarios.
- **GPT-4:** The latest model, GPT-4 OpenAI [47] is the SOTA model accross a wide range of tasks. The details of the model are not made public.

2.2.3 Improving Performannce of LLMs through Prompt Engineering

To extract optimal behaviour from pre-trained language models such as GPT-4, a line of recent work has focused on improving prompting methods. For example, Chain of Thought prompting, introduced by Wei et al. [64], has shown to improve reasoning abilities of language models on complex, multi-step tasks.

Similarly, developers have started to create software specifically designed to optimize prompts (Beauchemin [8], HegelAI [34]). This optimization aligns with the common practice of hyperparameter tuning in deep learning, applying the same principle to the construction and refinement of prompts.

2.2.4 In-Context Learning and Tools

Recent LLMs such as GPT-4 have shown emergent abilities to learn new concepts in-context, meaning at inference time. This has allowed users to place useful information in the prompt, to which the LM will employ. For example, recent work has shown a fine-tuned language model's ability to use “tools" provided in-context Schick et al. [55]. This is also possible using pre-trained language models Qin et al. [51]. Tools can be formulated as API calls. This is relevant to us, as invoking an API call mirrors calling a function.

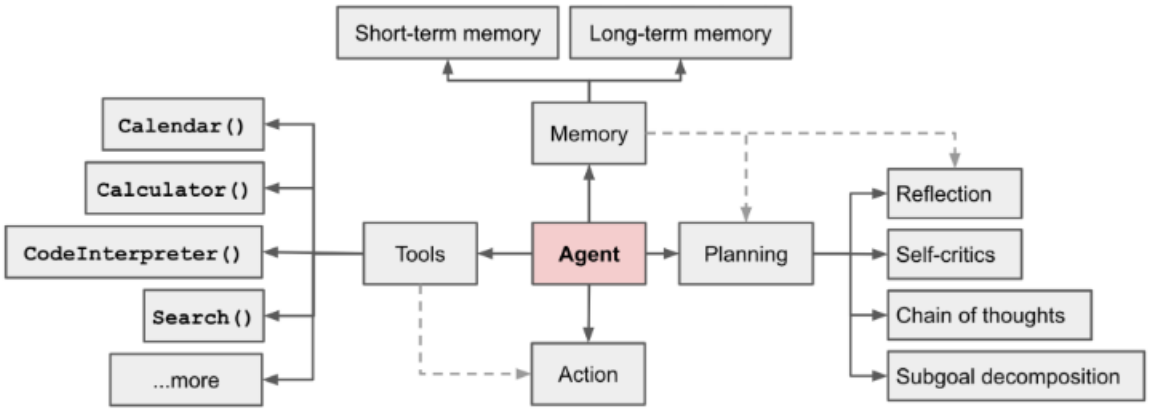


Fig. 2.2 **Overview of a LLM-powered autonomous agent.** (Image source: Weng [66])

2.2.5 Autonomous Agents

In this sub-section, we discuss how LLMs have been used to power autonomous agent systems (Significant-Gravitas [59], yoheinakajima [69]). We are interested in this because autonomous agent systems have been used to synthesise programs (AntonOsika [5]) and have the capability to influence drastic changes in how user-constrained program synthesis is done.

An autonomous agent system refers to an AI system that can operate and make decisions independently to accomplish goals, with minimal human intervention. The key components of such a system include:

- *Controller*, or "brain" of the system. In the context of this thesis, we consider the case which utilise an LLM as the controller. The LLM processes inputs, makes plans, calls APIs, and controls the overall system behavior.
- A *planning module* which Breaks down tasks into subgoals, orders actions, and allows the agent to refine strategies over time. This can be as simple as Chain of Thought prompting (Wei et al. [64]).
- *Memory* can be provided through LLM context, and long-term memory (if-needed) via an external database for storing and quickly retrieving information.
- *Tool use*, as discussed in Section 2.2.4.

The agent sets goals, makes plans, takes actions, handles complexity, improves over time, and maximizes autonomy using its reasoning and available tools/resources. A general

overview is provided in Figure 2.2. Although the stability and utility of agents such as AutoGPT (Significant-Gravitas [59]) has been questionable, agents designed to aid coding tasks have shown promise:

- **GPT Engineer** (AntonOsika [5]) is designed to create a whole repository of code given a task specified in natural language. The GPT-Engineer is instructed to think over a list of smaller components to build and ask for user input to clarify questions as needed.
- **Aider** (Gauthier [28]) is a coding-agent designed to as a command-line assistant which efficiently gains knowledge of a codebase and can be tasked to edit or implement new code. Aider has developed an efficient method to provide full “code-context" in the limited context of a language model by sending GPT a concise map of your whole git repository that includes all declared variables and functions with call signatures.

While agents like Aider are designed to work and build on-top of user-provided code, they do not allow for the user to fully constrain the model to the given codebase and do not directly present an effective way to handle cases when the model commits dysfunctional code.

2.3 Program Synthesis vs. Program Induction

In this section, we introduce the problem of program induction, an alternative approach towards algorithm development. Program synthesis and induction both seek to derive a program from input/output examples or user specifications. Program induction aims to create a model that itself functions as an algorithm, executing inputs like an algorithm, whereas program synthesis leverages models to generate programs. Though this work emphasizes program synthesis, we briefly review program induction efforts.

2.3.1 Recent Work in Program Induction

Many endeavors in program induction use neural architectures such as the Neural Turing Machine introduced by Graves et al. [31]. This architecture combines an LSTM with external memory, forming a trainable, differentiable Turing Machine. Zaremba and Sutskever [70] took a similar approach, training an LSTM to execute programs with remarkable accuracy.

Recent works in program induction have explored transformer-based models. Parez et al. [48] proved the Turing completeness of attention mechanisms (with slight modifications), the fundamental component of transformers. Giannou et al. [29] demonstrate that transformers, when looped (i.e., output is fed back in as input) can function as universal computers. In fact, they show that a looped transformer can emulate a basic calculator. Similarly,

Sindhi [60] proposed an approach for efficiently and precisely learning to combine previously trained skills for new tasks using coding transformers and hierarchical learning. Furthermore, Zhang et al. [72] investigated transformers' ability to emulate recursive functions, revealing that they learn approximations rather than true recursion.

2.3.2 Limitations

Program induction's limitations stem from its specificity. Although some methods yield interpretable programs (Kurach et al. [41], Gaunt et al. [27]), each task necessitates a unique neural network. Training such networks is often less efficient and less reusable compared to traditional code development.

Chapter 3

Method

In this chapter, we introduce our method to synthesise programs using only code provided in-context. The key steps, at a high-level, to our solution are:

1. Generate a prompt that both instructs the language model regarding the target algorithm and defines the user-specified "constraint." In this work, a "constraint" refers to the restriction that the model can only generate code containing calls to the specific set of functions provided by the user. Any other function calls outside this set are disallowed. This step is discussed in Section 3.1.

2. Use the generated prompt to sample code from the model. Evaluate the code using our “half-shot" method introduced in Section 3.4.1. If the code correctly implements the target algorithm while following the constraint, the problem is solved. Otherwise:

3. Generate a new prompt that informs a separate language model of the target algorithm, the constraint, and provides the non-working code. Instruct the model to produce a helpful "sub-skill," a term used in this work to refer to a Python function. The goal is for the model to propose a sub-skill that, if added to the set of allowed functions, will help the constrained language model produce working code. This step is discussed in 3.2.

4. Use the prompt generated in the previous step to sample a skill from a LM. Append this skill to the list of valid, user-provided functions. Details of this step are discussed in Section 3.3.

5. Repeat from Step 1.

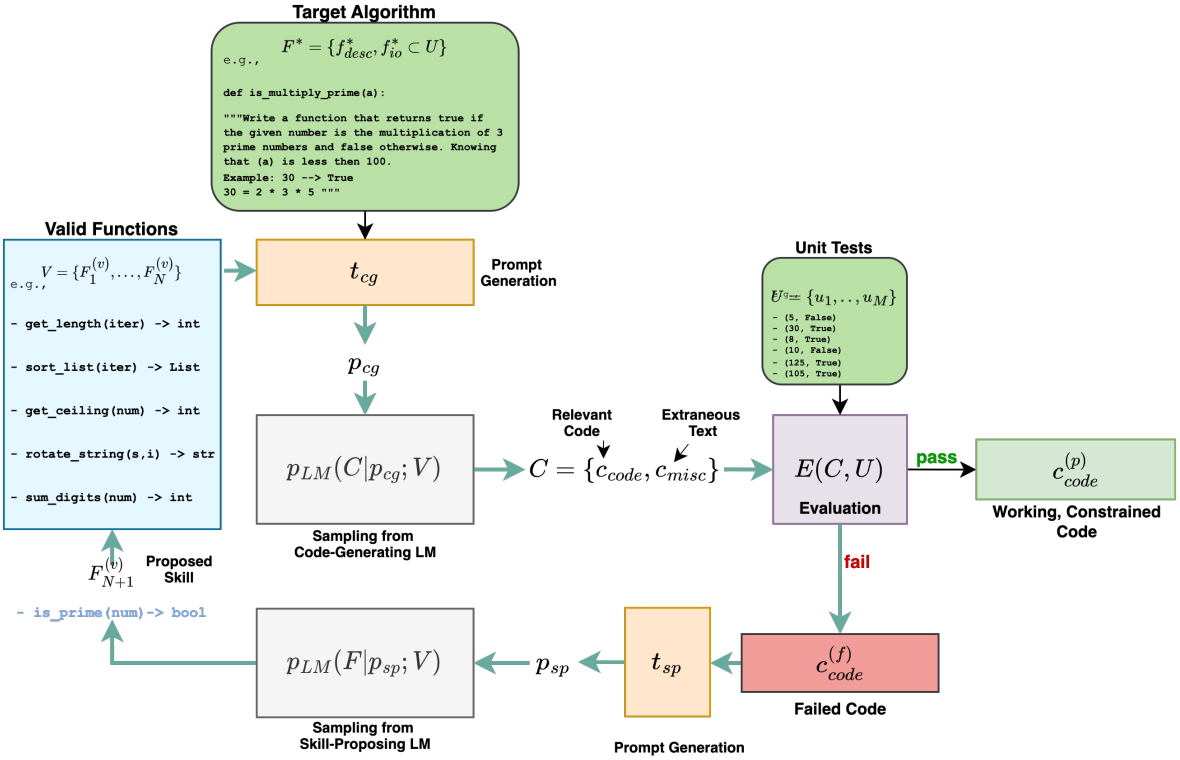


Fig. 3.1 **Method Overview.** Our goal is to implement a target algorithm F^* using only a set of user-provided functions V and basic Python operations. We constrain a language model to generate code under this constraint: $p_{LM}(C|p_{cg}; V)$ where p_{cg} is a dynamically-generated prompt. When the sampled code is non-functional, we employ another model $p_{LM}(F|p_{sp}; V)$ to implement a modular sub-skill, F_{N+1}. This sub-skill is added to V and designed to aid the constrained model in implementing F^*. If F_{N+1} does not adequately address deficiencies, this process repeats and new sub-skills are generated until the constrained model can generate working code. Sub-scripts CG and SP are used to distinguish between code-generation and skill-proposal, respectively. The core hypothesis is that supplying finely targeted skills not present in V will build a useful set of functions on-top of those provided in-context which will enable the constrained model to successfully generate F^*.

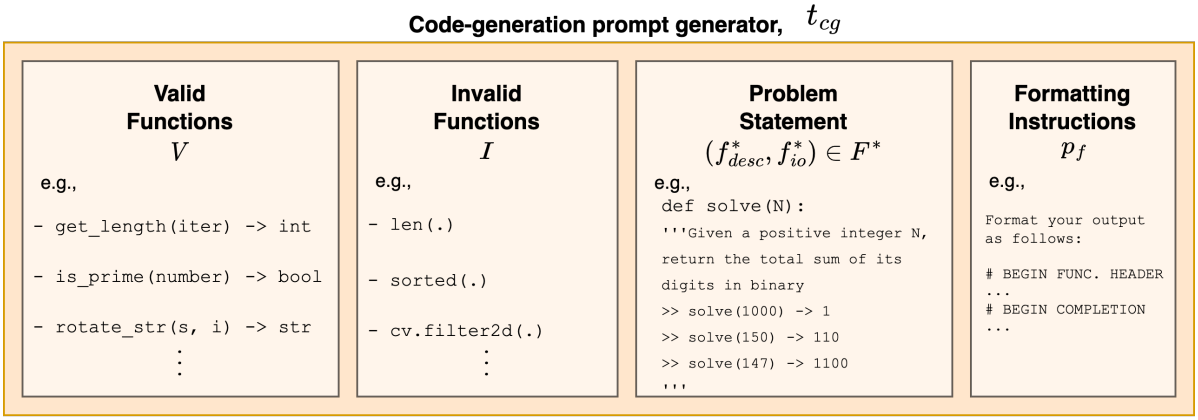


Fig. 3.2 **Prompt template used for constrained code-generation.** Key elements are specifying accessible valid functions V and restricting invalid ones I. Valid functions are provided through names, inputs/outputs, descriptions, and/or full code based on the optimal information level (see Fig. 3.4). Invalid functions, especially common training set ones like Python Standard Library, are listed to reinforce restrictions. Prompt structure is tuned by maximizing Usage Rate of $F \in V$ and minimizing Non-Compliance Rate of $F \in I$.

3.1 Constraining a Language Model to use Code Provided In-Context

In this section, we introduce a method to constrain a code-generating language model to use only a set of functions provided in-context by the user. We model code-generation as sampling code C from an LM formulated as a probabilistic function conditioned on prompt p_{cg}, which is generated by "prompt-generator" t_{cg}:

$$C \sim p_{LM}(C|p_{cg}) \tag{3.1}$$

$$p_{cg} = t_{cg}(F^*, V, I, p_f) \tag{3.2}$$

Here, t_{cg} takes as its inputs the target algorithm F^*, formatting-instruction p_f, the set of valid functions V, and the set of invalid functions I as an input. Although it may seem redundant to provide any information on any invalid functions since the model is already restricted to V (i.e., any $F \notin V$ is invalid), we discuss why this can be necessary later in this section. The subscript cg is used to denote that the specified prompt/prompt-generator is intended for code-generation.

Prompt generator, t_{cg}, (Fig. 3.2) is designed around two key objectives:

System Prompt

You are an intelligent programmer. You must complete the python function given to you by the user using only the functions they give you. And you must follow the format they present when giving your answer!

User Prompt

You have access to the following Python functions:

```
# VALID FUNCTIONS
{user_provided_functions}
```

You must complete the python function I give you using ONLY the given valid functions. You CANNOT use any of the following invalid functions:

```
# INVALID FUNCTIONS
{restriced_functions}
```

You must write the completion in the following form:

```
# FUNCTION HEADER
...
# START OF COMPLETION
...
```

You may only write your response in code/comments. Do not be verbose.
The function you are to complete is:

```
{func}
```

Fig. 3.3 **Example prompt for an OpenAI chat model (GPT4/3.5).** Constraints on valid and invalid functions are provided in the User Prompt. Alternatively, constraints can be specified in the Syten Prompt. Optimal placement depends on the model and should be tested. Using natural, commenting-style formatting instructions increases likelihood of correct formatting, as models have exposure to this. Unfamiliar formatting styles (e.g. @@!!Start Completion!!@@) may be ignored, especially at temperature=0.

- Maximise the likelihood that, for any target algorithm F^*, the model will generate working code C which calls the user-provided functions $F \in V$
- Minimise the likelihood that the model will invoke invalid functions I or, more generally, $F \notin V$, thereby restricting the model to functions allowed within the user-defined constraint.

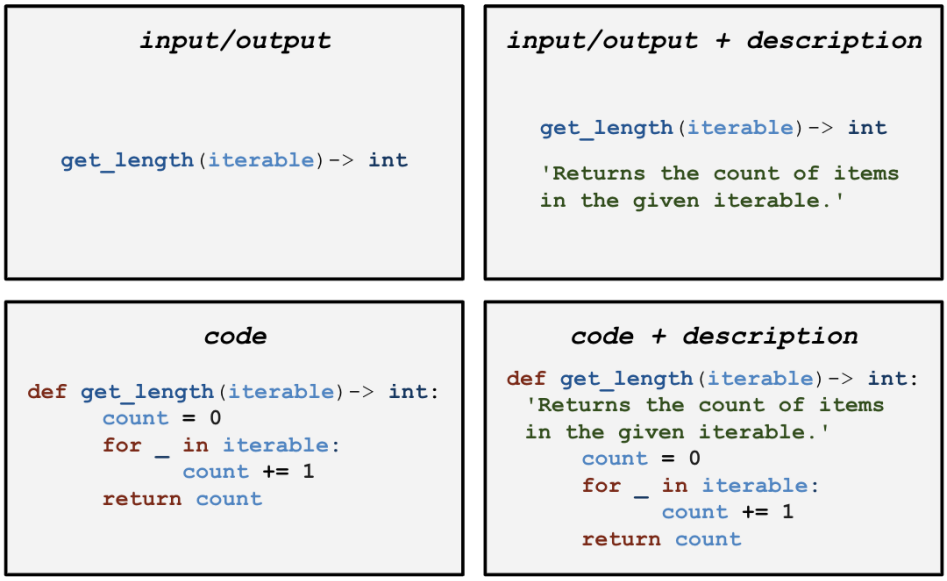


Fig. 3.4 **Information levels for provided functions.** At minimum, name and I/O-behavior must be given (top left). Optionally, a description can be added (top right). Alternatively, code without comments (bottom left) or with comments (bottom right) may be provided. The optimal information level depends on the model, task, and budget.

Providing valid functions. To accomplish these goals, prompt p_{cg} must provide useful information on the valid functions and clearly restrict invalid ones. We consider the case where all $F \in V$ can be provided directly in the prompt. An alternative approach is supplying an API to query valid functions as needed. However, if all functions fit within the model's context window, as in our experiments, using an API is redundant.

To encourage correct usage of a user-provided function $F \in V$, the prompt must include, at minimum, instructions for calling each function and input/output behaviour, i.e., $f_{name}, f_{io} \in F$. Additionally, providing a description of the function f_{desc} and/or full source code, f_{code}, may be beneficial. Refer to Figure 3.4 for an example of these elements.

The optimal level of information to provide per function depends on the specific model, task, and budget. To determine the appropriate level of detail, we introduce the *Utilisation Rate* (UR) metric in Section 3.4.2 of this report.

Restricting Non-Valid Function Calls. After providing function details, we specify that *only* functions in V may be called in generated code. To reinforce the constraint, a list of invalid functions, I, may also be explicitly provided. Listing invalid functions is useful when restricting use of highly exposed functions, i.e., Python Standard Library. Deciding which invalid functions to explicitly restrict depends on the task context. For example, when

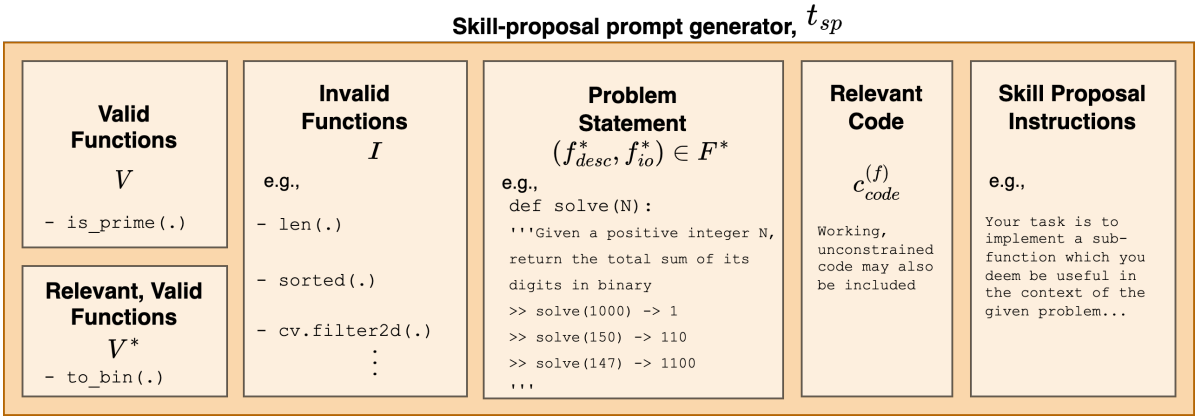


Fig. 3.5 **Skill-proposal prompt template.** Key elements are the failed code $c^{(f)}_{code}$. skill proposal instructions, and highlighted skills V^*. Working code that does not follow the required constraints may also be included. The aim of the prompt is to instruct the skill-proposal model to determine and implement a new sub-function F_{N+1} to provide to the constrained model.

generating computer vision code, explicitly prohibiting OpenCV (Bradski [14]) functions is useful. To guide selection, we define the *Non-Compliance Rate* (NCR) in Section 3.4.2.

3.2 Sub-skill generation

Code C sampled from $p_{LM}(C|p_{cg};V)$ is evaluated for functional-correctness. Evaluation is formulated as a function $E(C,U)$ which parses relevant code c_{code} from C and evaluates this against unit-tests U. The evaluation block tags the code with a pass (p) or fail (f) and returns the tagged code as its output. Further details on evaluation are discussed in Section 3.4.

When a constrained language model generates dysfunctional code $c^{(f)}_{code}$, for a given task F^*, our approach requires generating a new sub-skill which, if added to the set of valid functions, $V = \{F^{(v)}_1, ..., F^{(v)}_N\}$, is intended to increase the probability of sampling working code from the constrained language model:

$$p_{LM}(c^{(p)}_{code}|F^*, F_{N+1} \cup V) > p_{LM}(c^{(p)}_{code}|F^*, V) \tag{3.3}$$

Here, $c^{(p)}_{\text{code}}$ is working, constrained code and F_{N+1} is the new sub-skill. We define sub-skills as modular Python functions that can be invoked in an overarching function implementation.

System Prompt

Follow the user instructions and provide an implementation of what you deem to be the most useful sub-function.

User Prompt

I have the following docstring:
`{docstring}`

A correct completion to this function is:
`{unconst_comp}`

I constrained a language model to generate a new completion using only custom Python functions that I provided. I gave it access to the following functions:
`{custom_funcs}`

It generated the following incorrect completion:
`{const_comp}`

What new function would be useful to provide to the "constrained" language model to help it produce a working completion? Propose a completely new function. Only output code implementing the new function you propose. Only output executable code. Format your answer as follows:

BEGIN NEW-SUB FUNCTION ...

Fig. 3.6 **Example skill-proposal prompt for an OpenAI chat model (GPT4/3.5).** This illustrates querying for a single sub-skill function, though the prompt could be modified to request multiple skills. The problem statement and non-working constrained model are included to provide context. Working code which does not adhere to the constraints is provided in this example. The model is instructed to propose and implement the most useful new function to integrate.

If the coding-task is implementing a function, that function is the overarching skill. New sub-skills, when added to the collection of valid functions in the prompt (see Section 3.3 for details), aim to enable the constrained model to generate working code.

Sub-skills are proposed and implemented by sampling from a language model conditioned on prompt p_{sp}:

$$F \sim p_{LM}(F|p_{sp}) \tag{3.4}$$

$$p_{sp} = t_{sp}(c^{(f)}_{code}, F^*, V, I) \tag{3.5}$$

At minimum, the task description, F^*, valid functions V, and non-working code, $c^{(f)}_{\text{code}}$ are provided in the skill-generation process as context. The subscript sp denotes that the associated prompt/promp-generator are intended for skill-proposal.

We also consider the case where working, *non-constrained* code, f^*_{soln}, may be provided in the skill generation process:

$$p_{sp} = t_{sp}(c^{(f)}_{code}, f^*_{soln}, F^*, V, I) \tag{3.6}$$

It is crucial to note that, if included, the working code, f^*_{soln} does not meet the requirements imposed by the constraint. f^*_{soln} may be directly provided by a dataset (see Section 4.1) or produced by an unconstrained language model. Also note that the entire sub-skill generation process maintains the user specified constraints.

To measure the quality of a proposed sub-skill, we introduce Precise and General Utility (PU/GU) in Section 3.4.2.

3.3 Integrating Sub-Skills into New Code Generations

In the previous section, we outlined our method for generating useful sub-skills when a constrained model fails to produce working code. Here, we discuss how to effectively integrate new sub-skills into the constrained model's future code-generations.

To integrate a new sub-skill F_{N+1} into new code-generations we can simply add it to the set of valid functions V:

$$V \leftarrow \{F_{N+1}\} \cup V \tag{3.7}$$

and follow the approach discussed in Section 3.1. This alone, however, is not sufficient. Because F_{N+1} was implemented to aid specifically in solving task F^*, it is likely that F_{N+1} is more useful than other $F \in V$ in the context of solving F^*, and should be distinguished within prompt p, among $F \in V$. To do this, we introduce the following subset:

$$V^* \subset V \tag{3.8}$$

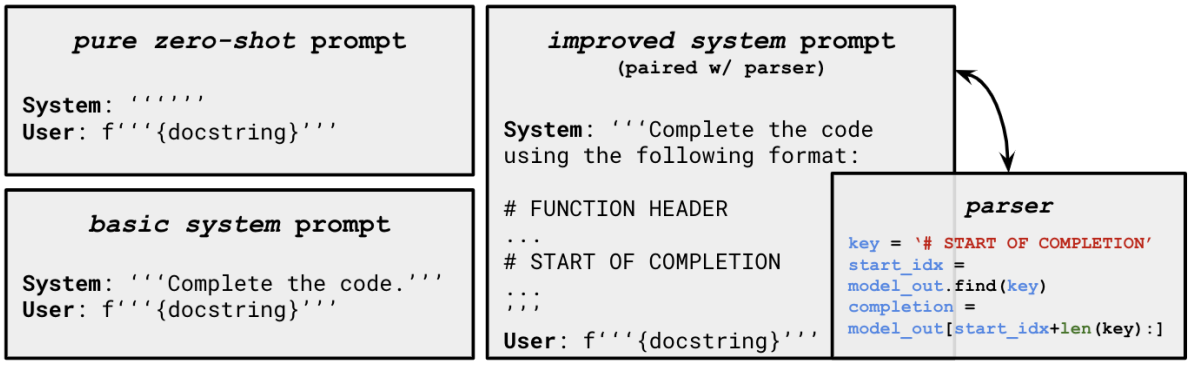


Fig. 3.7 **Levels of Formatting Guidance.** Basic zero-shot prompt (top left) only includes dataset prompt, i.e., `docstring`. Next level (bottom left) adds "Complete the code." to the system prompt. Final prompt (right) provides specific format instructions and parses outputs expecting that format. The aim is to reduce formatting-errors by introducing direct formatting instructions.

where V^* contains all $F \in V$ which are thought to be “particularly useful" in implementing the target algorithm F^*. We then formulate the new prompt p with respect to this distinction:

$$p_{cg} = t_{cg}(F^*, V^*, V, I, p_f) \tag{3.9}$$

Here, the updated prompt p_{cg} is used to condition the language model, from which a new solution to F^* is sampled.

Note that as $|V^*|$ increases, performance may decrease by overloading the context. Therefore, it is necessary to determine the optimal size for $|V^*|$ (for an example, see Figure 5.11).

3.4 Evaluation Framework

In this section, we motivate and describe our novel “half-shot" methodology for evaluating code-generations for functional-correctness (Subsection 3.4.1) and introduce the metrics used to assess the quality of proposed sub-skills (Subsection 3.4.2).

3.4.1 Evaluating Code Generations for Correctness

In this subsection, we first introduce a common metric used in functional evaluation of code. We then discuss how the standard evaluation procedure of language models on coding tasks can be misleading, as inconsistent output formatting penalizes models' scores. In the end, we introduce a new evaluation method leveraging *formatting instructions* and *output parsing* which provides a tighter estimate of a model's potential on programming datasets.

Metrics for Code Evaluation

Code generated by a language model can be evaluated for functional correctness through unit testing or, less commonly, for similarity to a canonical solution using match-based metrics like CodeBLEU (Ren et al. [54]). In this work, we focus solely on testing for functional correctness, aligning with standard procedures used by human developers to evaluate code.

When evaluating model generations for correctness, a useful metric is pass@k accuracy originally introduced by Kulal et al. [40]. k generations C per-problem $q_i \in Q$ are sampled from the model, $p_{LM}(C)$, and a problem is considered solved if any of the generations pass the associated unit-tests. Formally, for a set of questions Q, where each $q_i \in Q$ has an associated set of unit-tests U, the pass@k accuracy is defined as:

$$\text{“zero-shot” pass@k} = \frac{1}{|Q|}\sum_{i=1}^{|Q|} \mathbb{1}(\exists\, c_n \in C : u_j = \text{pass} \forall\, j) \tag{3.10}$$

$$= \frac{1}{|Q|}\sum_{i=1}^{|Q|} \mathbb{1}(\exists\, LM(p_i)_n \in C : u_j = \text{pass} \forall\, j) \tag{3.11}$$

Here, $C = \{c_1, ...c_k\}$ is the set of k generations sampled from model $LM(p_i)$ and p_i is the prompt associated with question i. In this context, “zero-shot” means that the only input to the model is prompt p_i provided by the dataset.

Standard Method of Evaluating Baseline Models

Typically, datasets like HumanEval and APPS (introduced in Section 4.1) provide a prompt describing the code to implement, which is fed directly to the model as the *only input* when evaluating for zero-shot performance. The model’s output is evaluated for functional correctness, as described above. However, this zero-shot evaluation methodology is problematic when models have inconsistent output formatting.

Specifically, top-performing language models are often finetuned to behave interactively, generating verbose, conversational responses rather than just completions. This causes variability in how they format code outputs by default - some models tend to wrap code in markdown formatting like “‘triple backticks”’ or may include non-executable explanatory text before or after the code. These extraneous characters often result in syntax errors when the output is evaluated, penalizing the model’s score regardless of the functionality of the code provided (Chen et al. [18]).

To accurately evaluate experimental results presented in Chapter 5, we need a strong baseline representing the true potential of existing models and approaches. If the baseline underperforms due to a suboptimal evaluation procedure, it may be the case that our approach's gains are simply a result of improved formatting rather than enhanced coding abilities (although this would be difficult to prove). At minimum, we want to ensure our approach provides quantitative value beyond what can be added from simply using a formatting prompt and parser.

Our Method of Evaluating Baseline Models

To do this, we introduce a modified evaluation method focused on establishing tight upper bound estimates for a models coding-ability. The rightmost prompt in Figure 3.7 illustrates our evaluation methodology on the *HumanEval* dataset for GPT-3.5 and GPT-4 models. While the optimal formatting prompt and its placement may vary based on the model and dataset, this demonstrates our general approach, which is outlined below:

1. Add a formatting-string to the prompt given by the dataset, instructing the model to formulate relevant code in as easy-to-parse format. The formatting-string cannot provide any domain specific informational advantages. For example, a formatting-string which uses the correct answer as an example on how the model should structure its output is not acceptable.

2. Parse the model output to extract the relevant code. The parser should strip non-code from model outputs before evaluation, minimizing syntax errors.

3. Evaluate the code using pass@k.

Combining all three steps, we formulate this evaluation method, calling it "half-shot" pass@k, as follows:

$$\text{"half-shot" pass@k} = \frac{1}{|Q|}\sum_{i=1}^{|Q|} \mathbb{1}(\exists\ c_i^* \in C : u_{ij} = \text{pass} \forall\ j) \tag{3.12}$$

$$= \frac{1}{|Q|}\sum_{i=1}^{|Q|} \mathbb{1}(\exists\ P(LM(p_i, p_f)_n, p_f) \in C : u_{ij} = \text{pass} \forall\ j) \tag{3.13}$$

In comparison to "zero-shot" pass@k, "half-shot" pass@k adds a formatting-string to the model input, denoted p_f and passes the model output through parser P. Note that the behavior of the parser is dependent on p_f.

Because this approach modifies the input to the model and parses the output, our evaluation method does not technically qualify as zero-shot even though no task-specific informational advantages are provided in-context. Nonetheless, we believe that "half-shot" evaluation provides a more accurate estimate of a language model's unaided performance on a coding task.

3.4.2 Metrics Used for Prompt Tuning

Evaluating Efficacy of Imposed Constraint

To quantify how responsive a constrained language model is to the provided constraints, we define Usage Rate (UR), Function-Specific Usage Rate (UR_F), and Non-Compliance Rate (NCR):

$$UR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \mathbb{1}(\exists F \in V : c_i \text{ calls } F) \tag{3.14}$$

$$UR_F(F) = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \mathbb{1}(c_i \text{ calls } F) \tag{3.15}$$

$$NCR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \mathbb{1}(\exists F \in I : c_i \text{ calls } F) \tag{3.16}$$

$$NCR_F(F) = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \mathbb{1}(c_i \text{ calls } F) \tag{3.17}$$

All metrics are evaluated across a set of tasks Q. The UR indicates how often the model uses any of the provided functions. The UR_F offers finer-grained insight into how specific functions F are being used.

The NCR indicates at what rate the model generates calls to invalid functions. Although the function agnostic NCR generally suffices for prompt optimisation, NCR_F may further reveal a model's exposure to specific functions during training, giving insight to take steps to further constrain or de-bias the model accordingly.

Evaluating Utility of a Sub-skill

To quantify the usefulness of a sub-skill F_{N+1}, for a single task and at-large, we introduce Precise and General Utility (PU/GU), respectively:

$$PU = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{\mathbb{1}(F_{N+1} \in V^*, c_{code} \text{ calls } F_{N+1}, c_{code} = \text{passed})}{\mathbb{1}(F_{N+1} \in V^*)} \tag{3.18}$$

$$GU = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{\mathbb{1}(F_{N+1} \in V^*, c_{code} = \text{passed})}{\mathbb{1}(F_{N+1} \in V^*)} \tag{3.19}$$

PU measures the rate at which the model generates working code which calls F_{N+1} when F_{N+1} is included as a distinguished sub-skill. GU more generally measures the rate at which the model generates working code when F_{N+1} is included as a distinguished sub-skill. High GU and low PU may indicate that although the model did not directly call the given sub-skill in its output, it may have learned useful information from it in-context.

Chapter 4

Experimental Setup

In this chapter we provide details related to the experiments presented in Chapter 5:

- In Section 4.1, we provide a description of the two datasets used, along with a sample question from each dataset (Figures 4.2, 4.3). In addition, we provide details of the all sub-datasets in Subsection (see 4.1.3) used in Sections „, of this report.
- In Section 4.2, we provide details of the prompts used in Section 5.2.1 of this report.
- In Section 4.3, we present the details of the set of functions V used to constrain the language model in our experiments. In Subsection 4.3.1 we offer a closer look at the five splits of functions presented in Figure 5.6.

4.1 Description of Datasets

In this work, we use two datasets: HumanEval (Chen et al. [19]) and APPS (Hendrycks et al. [35]).

4.1.1 HumanEval

HumanEval (HE) is designed to measure functional correctness for synthesizing programs from docstrings. It consists of 164 Python questions:

$$\text{HumanEval} = \{H_1, ..., H_{164}\} \tag{4.1}$$

Where each question H_i contains:

$$H = \{h_{\text{doc}}, h_{\text{io}}, h_{\text{soln}}\} \tag{4.2}$$

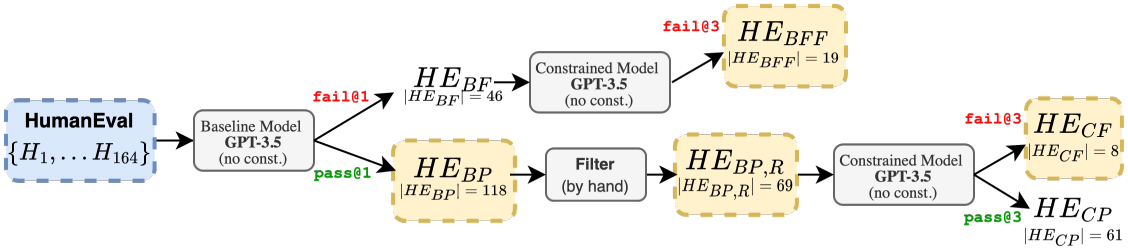


Fig. 4.1 **Sub-datasets extracted from HumanEval.** Relevant sub-datasets are highlighted in yellow. HE_{BFF} includes the questions that the original, unconstrained model failed to solve with just one attempt. On the other hand, HE_{CF} comprises questions that the baseline, unconstrained model answered correctly, but the constrained model could not solve. Both of these sub-datasets undergo a refinement process, where the constrained baseline model is allowed three more attempts at answering all questions (with $temp = 0.5$). This step ensures that any future successes in our experiments are more likely a result of genuine improvement rather than random chance.

where h_{doc} is the docstring prompt, h_{io} provides input/output examples to test correctness, and h_{soln} gives a reference solution. We use HumanEval because it is high-quality and handwritten, reducing exposure risk in models. The main drawback is the small-number of questions, which is why we also use APPS.

4.1.2 APPS

The Automated Programming Progress Standard (APPS) dataset consists of competition-level coding problems collected from different open-access coding websites such as Codeforces. APPS attempts to mimic how humans programmers are evaluated by posing coding problems in unrestricted natural language and evaluating the correctness of solutions.

$$\text{APPS} = \{A_1, ..., A_{10000}\}$$

$$H = \{a_q, a_{meta}, a_{io}, a_{solns}\}$$

Each problem contains a verbosely worded question, a_q, a rating (classified as introductory, intermediate, or competition level) contained in the metadata a_{meta}, input output pairs to test for functional correctness a_{io} and a number of canonical solutions a_{solns}.

Because APPS is compiled from open-access websites, it is suspected to have leaked into many language model's training data (Chen et al. [19]). We use APPS mainly to show our results on a large-scale.

TestID 0000

Question, a_q

An accordion is a string (yes, in the real world accordions are musical instruments, but let's forget about it for a while) which can be represented as a concatenation of: an opening bracket (ASCII code 091), a colon (ASCII code 058), some (possibly zero) vertical line characters (ASCII code 124), another colon, and a closing bracket (ASCII code 093). The length of the accordion is the number of characters in it. For example, [::], [:||:] and [:|||:] are accordions having length 4, 6 and 7. (:|:), {::}, [:],]:||:[are not accordions.
You are given a string s. You want to transform it into an accordion by removing some (possibly zero) characters from it. Note that you may not insert new characters or reorder existing ones. Is it possible to obtain an accordion by removing characters from s, and if so, what is the maximum possible length of the result?
—Input—
The only line contains one string s ($1 \le |s| \le 500000$). It consists of lowercase Latin letters and characters [,], : and |.
—Output—
If it is not possible to obtain an accordion by removing some characters from s, print -1. Otherwise print the maximum possible length of the resulting accordion.

Unit-Tests, a_{io}

|[a:b:|] -> 4
|]:[|:] -> -1
...

Canonical Solution, a_{soln}

...

Metadata, a_{meta}

Difficulty Interview
...

Fig. 4.2 **Example APPS question**. The question a_q is fed as the input to the LM. The output of the model is evaluated against unit tests, marked a_{io}. Each question in the APPS dataset has a difficulty rating: beginner, interview, or competition.

Task ID HumanEval/71

Prompt, h_{doc}

```
def triangle_area(a, b, c):
    '''
    Given the lengths of the three sides of a triangle.
       Return the area of
    the triangle rounded to 2 decimal points if the three
       sides form a valid triangle.
    Otherwise return -1
    Three sides make a valid triangle when the sum of any two
        sides is greater
    than the third side.
    Example:
    triangle_area(3, 4, 5) == 6.00
    triangle_area(1, 2, 10) == -1
    '''
```

Canonical Solution, h_{soln}

```
    if a + b <= c or a + c <= b or b + c <= a:
        return -1
    s = (a + b + c)/2
    area = (s * (s - a) * (s - b) * (s - c)) ** 0.5
    area = round(area, 2)
    return area
```

Unit Tests, h_{io}

```
def check(candidate):
    assert candidate(3, 4, 5) == 6.00
    assert candidate(1, 2, 10) == -1
    assert candidate(1, 1, 1) == 0.43
    assert candidate(2, 2, 10) == -1
```

Fig. 4.3 **Example HumanEval question.** The prompt h_{doc} contains a docstring providing a description of the target algorithm and a few examples. This is fed to the LM. The model's output is evaluated against unit tests h_{io} for functional correctness.

4.1.3 Sub-datasets

Four sub-datasets are extracted from HumanEval and used in our experiments. One sub-dataset, $APPS_{BP}$, is extracted from APPS. Refer to Fig. 4.1 for a visual guide explaining the extraction of the HumanEval sub-datasets.

HE_{BP}. The first sub-dataset, HE_{BP}, consists of 118 questions. These questions are sourced from those that the baseline model, GPT-3.5, can answer in one attempt using the "half-shot" pass@1 method (refer to 3.4), without any constraints. The aim of HE_{BP} is to pinpoint questions correctly addressed by the original unconstrained model.

$HE_{BP,R}$. The second sub-dataset, $HE_{BP,R}$, comprises 69 questions. These are extracted from HE_{BP} by manually identifying questions whose answers contain at least one free-standing function call, such as `len(.)` or `math.sqrt(.)`. The function calls identified serve as a reference to construct V_{rep}, as detailed in Fig. **??**. The primary goal of $HE_{BP,R}$ is to spot questions from the unconstrained model that feature a free-standing function call in their solutions. This set helps formulate V_{rep} (elaborated in Section 4.3).

HE_{CF}. The third sub-dataset, HE_{CF}, encompasses 8 questions. These are derived by testing the baseline GPT-3.5, but this time constrained on V_{rep} (refer to Sec. 4.3), on $HE_{BP,R}$ with a three-attempt limit, and collecting all the unanswered queries. HE_{CF} is designed to ensure that if our method improves a model's performance on a question from this set during tests, it is likely due to our specific improvements and not mere randomness. This dataset aids in gauging the effectiveness of introducing sub-functions.

HE_{BFF}. The fourth sub-dataset, HE_{BFF}, has 19 questions, obtained in a two-step manner:

1. Initially, we gather questions the baseline model, GPT-3.5, fails to answer in one attempt using the "half-shot" pass@1 method (as mentioned in 3.4), without constraints.

2. Following that, we test the constrained GPT-3.5 on V_{rep} (see Sec. 4.3 for V_{rep} specifics) on the previously collected questions, allowing for three tries, and select all the incorrectly answered ones.

The purpose of HE_{BFF} is to represent the questions that the original, unconstrained model couldn't solve in one attempt (as described in step 1). In our experiments, we want to decrease the likelihood that the questions in this dataset are passed just by random chance, rather than because of the improvements introduced by our method. To do this, we use the process in step 2 to refine and confirm the questions that were selected from step 1.

APPS$_{BP}$. $APPS_{BP}$ consists of 1329 questions from the APPS dataset which the unconstrained GPT-3.5 does not pass with one attempt (i.e., using “half-shot" pass@1). This sub-dataset is similar to HE_{BP}, but derived from APPS rather than HumanEval.

4.2 Prompt Details

In our experiments in Section 5.2.1, we present a number of different code-generation prompts, defined by different parameters of t_{cg}. The details are provided below.

Parameters of the Code-Generation Prompt Generator, t_{cg}

The tunable parameters, defined (for simplicity) as (x_1, x_2, x_3), are described below:

1. x_1 is a discrete parameter describing *where* in the prompt the connstraint is given. Since we are using GPT-3.5/GPT-4 (see ...) we have the option of including our constraints in the “user” prompt ($x_1 = U$) or “system" prompt ($x_1 = S$).

2. x_2 is a discrete parameter describing whether reinforcement of any invalid functions I is included ($x_2 = R$) or not ($x_2 = NR$ denoting “no-reinforcement").

3. x_3 is a discrete parameter defining the level of detail provided for a $F \in V$, as discussed in Sec. 3.1. All options must include the function name f_{name} and input/output behavior f_{io}. On top of these, we consider including:

   (a) Code f_{code} ($x_3 = C$)

   (b) Function description f_{desc} ($x_3 = D$)

   (c) Both code and description ($x_3 = C + D$)

The exact parameters of the five variants of t_{cg} considered in Sec. 5.2.1 and presented in Figures 5.3, 5.2 are given below:

1. (U, NR, C)

2. (U, R, C)

3. (S, R, C)

4. (U, R, D)

5. $(U, R, C + D)$

The five prompt-generators are crafted to gain the following insights:

- Comparing performance of (U, NR, C) to (U, R, C) provides insight on the effects of including a reinforcement of invalid functions.
- Comparing performance of (U, R, C) to (S, R, C) provides insight on whether including the constraint in the user or system prompt produces better results.
- Comparing (U, R, C), (U, R, D), and $(U, R, C + D)$ provides insight on the optimal amount of information to include for each $F \in V$

4.3 Details of Replicas

In this section, we provide details on the specific functions used to constrain the models in the experiments presented in Chapter 5. The models are restricted to a particular set of functions, denoted by V_{rep}. A detailed overview of these functions can be found in Fig **??**.

V_{rep} comprises 21 hand-written functions created to emulate common functions from the Python Standard Library ([cite]) and math library ([cite]). Hence, we call these functions “replicas." They are modeled on functions utilized by the unconstrained, baseline model during its evaluation on the HumanEval dataset (refer to Sec. 4.1.3). All replicas adhere to the following criteria:

1. A replica must maintain the same functionality as the original function it aims to imitate.
2. A replica must bear a distinct name from the corresponding original function, although the name should reflect its purpose. For instance, the replica mimicking `len` is named `get_length`.

In our experiments, we want to constrain the models to “replicas" in specific to ensure that we are not depriving the models of any essential resources they need to solve the problems in our dataset.

V_{apps} Whether a model is evaluated on HumanEval, HE_{BFF}, or HE_{CF}, the complete set of V_{rep} is used as the set of valid functions. However, for evaluations on the APPS dataset, we slightly relax the original constraint, resulting in a slightly different set of allowed functions, V_{apps}. Specifically, the functions:

Original Function	**Handwritten Replica**, $F \in V_{rep}$	**Num. Occurences**
len	get_length	35
int	cast_to_int	11
sum	compute_sum	10
sorted	sort_list	8
str	cast_to_string	7
set	create_set	7
max	get_maximum	6
list	create_list	6
bin	convert_to_binary	5
min	get_minimum	5
abs	absolute_value	4
float	cast_to_float	3
round	round_number	2
all	check_if_all_true	2
isinstance	check_if_instance	2
ord	get_unicode	1
math.ceil	get_ceiling	1
map	apply_func_to_iterable	1
math.sqrt	get_square_root	1
chr	convert_to_char	1
filter	add_to_list_if_func_is_true	1

Table 4.1 **Replicas and their corresponding original functions.**The second column lists all functions in V_{rep} to which the models are constrained to in our experiments. The corresponding original functions are provided in the left column. The 'number of occurrences' refers to the number of questions to which the original function appeared in the solutions generated by the unconstrained baseline model (GPT-3.5) when evaluated on HumanEval.

$$
\begin{aligned}
R = \{&\texttt{cast\_to\_int(.)},\\
&\texttt{cast\_to\_float(.)},\\
&\texttt{convert\_to\_char(.)},\\
&\texttt{create\_list(.)},\\
&\texttt{create\_set(.)},\\
&\texttt{apply\_func\_to\_iterable(.)}\}
\end{aligned}
\tag{4.3}
$$

are removed from V_{rep} and replaced with the corresponding functions from the PSL:

$$O = \{\texttt{int(.), float(.), char(.), list(.), set(.), map(.)}\} \tag{4.4}$$

Thus, the modified set V_{apps} is defined as:

$$V_{\text{apps}} = (V_{\text{rep}} / R) \cup O \tag{4.5}$$

The motivation behind this modification stems from the APPS dataset's frequent requirement for outputs in the "Standard-Input Format" used in competitive programming. This format mandates that the generated code reads test data as user input (e.g., utilizing the `input()` function), instead of adhering to a call-based format in which data is directly supplied as an argument to a function. Should an APPS question require a Standard-Input format output, the model is heavily conditioned by its training data (enough to override in-context restraints) to utilise functions like `int(.), float(.), char(.), list(.), set(.), map(.)` to process user input into a workable form. An alternative fix to ours might be to provide explicit instructions in the prompt, telling the model to use the standard input format and simultaneously guiding it to employ specific functions (such as our replicas) for reading and processing the input data. However, this level of overspecification gives the model an informational advantage that conflicts with our experiments. To maintain the integrity of our constraints, we have therefore chosen not to adopt this approach.

4.3.1 Splits of Replicas

In Figure 5.6, we present 5 different ways of splitting V_{rep} into 4 subsets. The details on what functions are included in each split is provided in Figure 4.9. Within each split, a group is formed in attempt to have all the replicas in each group account for roughly a quarter of the total "number of occurrences." The number of occurrences of a replica is defined as the number of questions in $HE_{BP,R}$ in which unconstrained GPT-3.5's solution contains a call to the origin function of the replica.

Group	Functions
1	int, sum, sorted, chr
2	str, set, list, max, abs
3	min, bin, float, isinstance, all, round, ord, map, filter, math.ceil, math.sqrt
4	len

Fig. 4.4 **Split 1**

Group	Functions
1	len, ord, map, filter
2	int, sum, sorted, bin
3	str, set, list, min
4	max, abs, float, isinstance, all, round, chr, math.ceil, math.sqrt

Fig. 4.5 **Split 2**

Group	Functions
1	int, sum, str, bin
2	len, math.ceil, math.sqrt
3	sorted, set, list, abs
4	max, min, float, isinstance, all, round, chr, ord, map, filter

Fig. 4.6 **Split 3**

Group	Functions
1	len, float, isinstance, round
2	int, set, bin, abs
3	sum, max, list, chr, all
4	sorted, str, min, ord, math.ceil, math.sqrt, map, filter

Fig. 4.7 **Split 4**

Group	Functions
1	map, filter, math.ceil, math.sqrt, chr, ord, round, all, isinstance, float
2	abs, bin, min, max
3	list, set, str
4	sorted, sum, int, len

Fig. 4.8 **Split 5**

Fig. 4.9 **Overview of function splits used in Fig. 5.6.** Groups were determined by splitting functions into roughly even groups *based on number of occurrences* (see 4.1).

Chapter 5

Experiments and Results

This chapter presents the experiments conducted and the related results. An overview of the chapter and key-findings within each section are provided below.

In Section 5.1 we provide a quantitative comparison of our “half-shot” evaluation method, first introduced in Section 3.4, against the standard zero-shot evaluation method; We show that simply by providing a LM with formatting-instructions and parsing its output, GPT-4 achieves 85.4% accuracy on HumanEval, performing worse than only other one publicly available method. This implies that **many works claiming to have improved LMs' coding-abilities may not provide value beyond what basic formatting and parsing provide.** To re-calibrate the scale, we propose that “half-shot" evaluation should be used to evaluate models' coding ability and establish better baselines.

In Section 5.2, we constrain a language model to “replicas” and present the following:

- An assessment of a language model's adherence to user-defined functions; Our findings demonstrate that **GPT-3.5 and particularly GPT-4 are capable of using functions provided in-context**, although they encounter difficulties adhering to instructions prohibiting the use of certain functions.
- An examination of the impact on model performance when constraining a model to unseen replicas of frequently used Python functions, across various coding challenges; We find that **GPT-4 and GPT-3.5 perform substantially worse (committing more logical errors) on coding-tasks when constrained** to use replicated versions of common Python functions masked under different names.

In Section 5.3, we present an evaluation of the efficacy of integrating functions into constrained language models to enhance performance; We show that proposing and integrating

	Zero-shot	Basic	Formatting-string with parser
GPT-4	67.1	73.7	85.4
GPT-3.5	34.1	70.0	71.3
Reflexion (GPT-4)	91.0	-	-
Parsel (GPT-4 + CodeT)	85.1	-	-
LLaMA 2	29.9	-	-

Fig. 5.1 ***HumanEval* accuracy with varying prompting methods.** See Fig. 3.7 for examples of the prompts. Pass@1 accuracy with *temperature*=0 is reported. Zero-shot prompt only includes dataset prompt, i.e., `docstring`. The "basic system prompt" adds "Complete the code." to the system prompt. The formatting-prompt/parser provides specific format instructions and parses outputs expecting that format. Both GPT-4 and GPT-3.5 improve markedly when stricter formatting instructions are used, indicating many zero-shot failures may be due to inaccurate formatting rather than coding inability. The highest performing model, Reflexion [58], is included for reference. Parsel [71] is included as an example of a code-generating procedure that provides no added performance value beyond what a formatting-string and parser can provide. LLaMA 2 [62] is included to emphasize the superiority of GPT-4/GPT-3.5 over open-source models on coding-tasks.

automatically generated functions allows a constrained model to solve problems it failed due to the constraint. Additionally, **providing GPT-generated sub-skills further enables constrained models to solve problems that even the unconstrained model initially failed to solve.**

Finally, in Subsections 5.3.1 and 5.3.2, we provide a comparison of functions generated by language models with those created by humans; We show that **GPT-4 and GPT-3.5 produce effective functions in a fraction of the time as human-developers** and GPT-generated functions may generalise better than handwritten functions when many functions are provided to a constrained LM at once.

5.1 Making the Case for "Half-Shot" Evaluation

In this section we compare standard "zero-shot" pass@1 to "half-shot" pass@1 presented in Section 3.4.1. While zero-shot evaluation does not allow for any information to be given to the LM other than the question prompt and evaluates the model's output directly, half-shot evaluation appends a formatting-string to the model's input and parses the output. We evaluated GPT-3.5 and GPT-4 pass@1 accuracy on the HumanEval dataset using three different prompts, shown in Fig. 3.7. While all prompts avoid giving informational advantages

related to the question, just the first prompt qualifies as “zero-shot” in the traditional sense because it provides only the question prompt and nothing else. Evaluating the model using the remaining two prompts falls under our definition of “half-shot” evaluation.

Our results indicate that GPT-3.5 is approaches its (estimated) upper-bound accuracy (near 71%) faster than GPT-4 (near 85%) as the prompts are improved, likely indicating that GPT-4 has a higher upper-bound accuracy than GPT-3.5. Under zero-shot conditions, GPT-3.5 appears much worse than its successor, scoring 33 points below GPT-4, while under “half-shot” evaluation, the gap narrows to only 14 points. This is likely due to the fact that GPT-4, by default (i.e., zero-shot conditions), tends to output its answers in an executable format more often than GPT-3.5. This can be caused by differences in fine-tuning.

The tighter estimate of each model’s performance gained from providing direct formatting instructions and parsing the output allows for a more precise comparison of their coding proficiency. Additionally, it provides a more robust baseline from which to compare potential gains introduce by methods like Parsel (Zelikman et al. [71]) and Reflection (Shinn et al. [58]). Under zero-shot conditions, the Parsel method provides gains that are non-existent under half-shot evaluation. **Because, in this work, we prefer to compare our method’s performance with a baseline that cannot simply be beaten by a formatting-string/parser, we use the half-shot evaluation paradigm.**

5.2 Constraining a Language Model to Replicas

In this experiment, both GPT-3.5 and GPT-4 are constrained to a set of functions denoted by V_{rep}. The set V_{rep} consists of 21 hand-written functions designed to emulate the behavior of specific functions in the Python Standard Library (PSL), but under different names. These emulated functions are referred to as "replicas," with each corresponding to an "origin function" in the PSL, as illustrated in Table 4.1.

In addition to V_{rep}, the models are also constrained using a similar set of functions, V_{apps}, when evaluated on the APPS dataset. Due to the similarity between V_{apps} and V_{rep}, we will not distinguish between these constraints in our discussion. For more details on why a modified constraint is used on the APPS dataset, see Section 4.1.3.

A comprehensive description of the replicas and their construction is provided in Section 4.3. It is critical to note that the replicas are derived from all origin functions found in unconstrained GPT-3.5’s solutions to questions in $HE_{BP,R}$. Thus, constraining the models to the replicas ensures that no necessary functions are withheld from the models when solving

questions in $HE_{BP,R}$, allowing us to constrain the models without intentionally limiting their ability to solve problems in the specific dataset.

Here is an overview of this section:

- In Subsection 5.2.1 we evaluate the models' adherence to the constraint for a range of different prompts generated by different settings of t_{cg}. Details of the prompts used are provided in Section 4.2.
- In Subsection 5.2.2, we present our findings showing that GPT-3.5 and GPT-4 perform worse on both HumanEval, APPS, and $HE_{BP,R}$ when constrained to the replicas. We investigate the reasons for a decrease in performance.

5.2.1 Assessing the Models' Ability to Follow the Constraint

Here, we asses the models' ability to (1) use the functions provided in-context (V_{rep}) and (2) *not* use any functions outside set V_{rep}.

Figure 5.2 presents the Utilization Rate (UR; defined in Sec. 3.4.2) for both GPT-3.5 and GPT-4 when constrained by the indicated prompts and evaluated on HumanEval, alongside an unconstrained control for comparison. GPT-4 is only evaluated on the top-three prompt configurations to reduce cost. The primary observation here is the frequent use of $F \in V_{\text{rep}}$, with both models generating calls to the in-context functions on the majority of questions.

A juxtaposition of the UR with the Non-Compliance Rate (NCR) in Figure 5.3 reveals that although the models use the functions provided in-context, they still often call invalid functions. This is unsurprising, as the replicated origin functions are some of the most widely used Python functions. This means that the models were heavily exposed during training to the functions we are attempting to restrict.

From these results, we observe that using a higher-performing models (i.e., GPT-4) is the most effective way to increase a constrained model's usage of the functions provided in-context. However, given that GPT-4 exhibits a similar NCR to GPT-3.5 in all cases, we conclude that **restricting a model from using functions to which it was heavily exposed during training presents a more challenging task than encouraging the use of functions provided in-context.**

We also gain insight on the optimal prompt configuration. In our case, the optimal prompt configuration is (U, R, C), which performs slightly below $(U, R, C+D)$ while consuming fewer tokens. Prompt (U, R, D) leads to the highest utilisation rate but results in reduced accuracy,

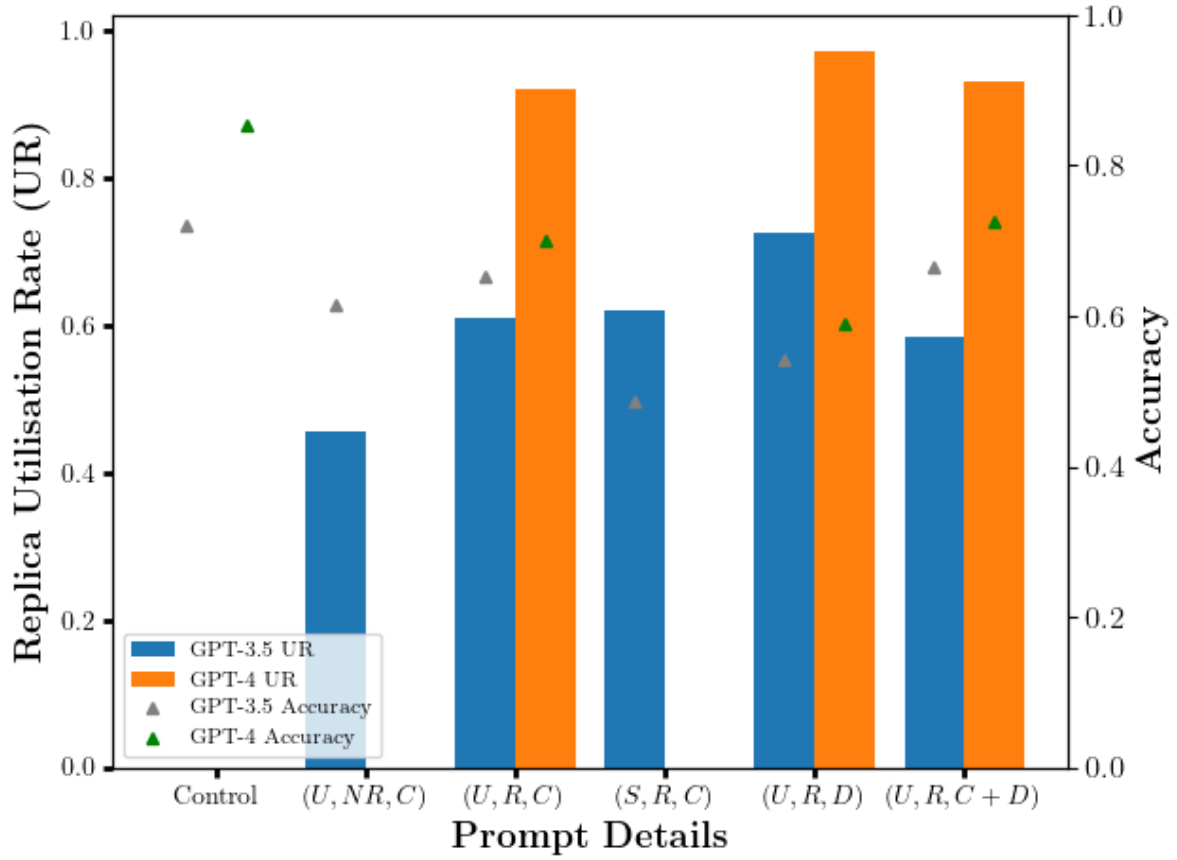


Fig. 5.2 **Function Utilisation Rate Across Different Prompt-Parameters and Models.** This figure illustrates the effect of model choice and prompt-configuration on the UR (left y-axis) of functions (replicas) in V_{rep} when answering HumanEval questions. A comparison is made between five specific prompt-configurations and an unconstrained control, for both GPT-3.5 and GPT-4. Overall accuracy on the dataset is represented on the right y-axis, and the goal is to maximize UR without compromising performance. The optimal prompt configuration is identified as (U, R, C), which slightly outperforms $(U, R, C + D)$ while using fewer tokens. In nearly all instances, the models utilise functions from V_{rep} in a majority of code-generations, showing LMs' ability to use unseen functions provided in-context. More details on the prompt configurations can be found in Sec. 4.2.

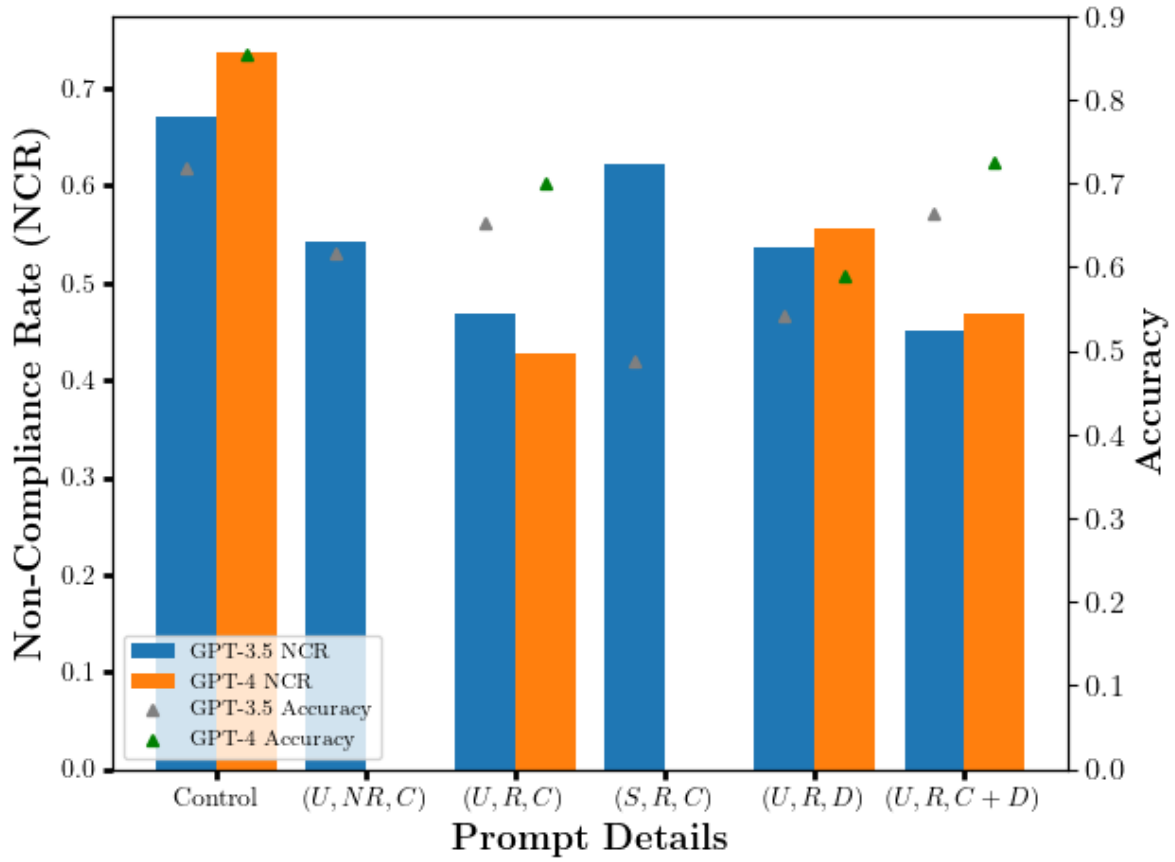


Fig. 5.3 **Non-Compliance Rate (NCR) Across Various Prompt Parameters and Models.** We compare the influence of different model choices and prompt configurations on the NCR of a constrained LM when answering HumanEval questions. Five specific prompt configurations are tested alongside an unconstrained control, which serves as a reference for understanding how often invalid functions are used without constraints. Even with the optimal model and configuration, the NCR remains above 50% of the control's rate, an expected outcome given the restriction of functions from the Python Standard Library, which are frequently encountered during training. A comparison between (U, NR, C) and (U, R, C) configurations further shows that explicitly listing invalid functions enhances compliance. In the best case, GPT-4 only marginally outperforms its predecessor, indicating that preventing models from using highly exposed functions is a more challenging task than encouraging them to use new functions provided in-context. For additional details on the prompt configurations, refer to Sec. 4.2.

	HumanEval	$\text{HE}_{\text{BP,R}}$	APPS_{BP}
GPT-4	85.4	94.2	-
GPT-4 Constrained	73.9	78.2	-
GPT-3.5	71.3	100	100
GPT-3.5 Constrained	65.8	75.3	10.4

Fig. 5.4 **Comparison of Model Performance With and Without Constraints.** This figure presents the "half-shot" pass@1 accuracy (with temperature set to 0) on HumanEval, $HE_{BP,R}$, and $APPS_{BP}$ datasets for both GPT-3.5 and GPT-4, evaluating their performance with and without constraints. While the constrained models are generally constrained to V_{rep}, they are specifically limited to V_{apps} when assessed on $APPS_{BP}$. For more details of the function sets and datasets, refer to Sec. 4.3 and 4.1.3. The results show a significant performance decline when constraints are applied even though all necessary functions are provided in V_{rep} and V_{apps}. Performance decrease is particularly significant in $HE_{BP,R}$ (25%) and $APPS_{BP}$ (-90%) datasets, which contain only questions the unconstrained GPT-3.5 model is able to solve. The drop is more pronounced for the $APPS_{BP}$ dataset, possibly because APPS questions are inherently more challenging than those in HumanEval, implying that the baseline unconstrained model's initial success on these questions might not be consistently replicable.

highlighting the need for an optimal balance between UR and correctness. In addition, **we find that providing the full source code f_{code} of the replicas appears to increase the model's usage of user-provided functions without compromising accuracy.** Adding descriptions does not significantly alter the outcome, possibly owing to the simplicity and self-descriptive nature of the replicas.

5.2.2 Evaluating the Impact of the Constraint on Model Performance

In the previous subsection, we analysed how well the models' are able to follow the imposed constraint. In this subsection, we asses the effects of the constraint on the models' performance on the HumanEval dataset.

Figure 5.4 provides the "half-shot" pass@1 accuracy of the constrained and unconstrained models across various datasets. A noticeable decline in performance becomes evident when comparing unconstrained models to their constrained counterparts. Specifically, for the $HE_{BP,R}$ dataset, where replicas encompass all required functions for accurate response generation, **the performance of GPT-3.5 declines by approximately 25% when constrained.** This suggests that the dip in accuracy arises not from the limitation of available tools. The decline in accuracy for the entire HumanEval dataset is less marked than in $HE_{BP,R}$. This is

Docstring

```
def cycpattern_check(a , b):
    '''
    You are given 2 words. You need to return True if the second
       word or any of its rotations is a substring in the first
       word
    cycpattern_check("abcd","abd") => False
    cycpattern_check("hello","ell") => True
    cycpattern_check("whassup","psus") => False
    '''
```

Unconstrained-Model Output

```
if len(a) < len(b):
    return False
if b in a:
    return True
for i in range(1, len(b)):
    if b[i:] + b[:i] in a:
        return True
return False
```

Constrained-Model Output

```
# Check if the length of b is
   greater than the length of a
if get_length(b)>get_length(a):
    return False
# Concatenate a with itself to
   check for rotations of b
a = a + a
# Check if b is a substring of
   a or any of its rotations
if b in a:
    return True
return False
```

Fig. 5.5 **Comparing Outputs from Constrained and Unconstrained GPT-3.5.** This figure shows the responses of GPT-3.5, in both constrained and unconstrained conditions, to a particular question involving the detection of cyclic patterns within strings. On the left, the Unconstrained-Model adheres to logic and a succinct approach. On the right, the Constrained-Model commits logical errors, beginning with the unnecessary concatenation of string a with itself. This discrepancy highlights the decrease in performance that may arise when a model is constrained to use replicas of common functions. The upper part of the figure shows the docstring, which outlines the specifications of the `cycpattern_check` function.

because questions present in HumanEval but absent in $HE_{BP,R}$ are either already failed by the unconstrained baseline or solvable without the necessity of a replica.

We attribute the 90% performance drop on the $APPS_{BP}$ dataset to the higher complexity of APPS questions compared to HumanEval. It is likely that the unconstrained model's baseline

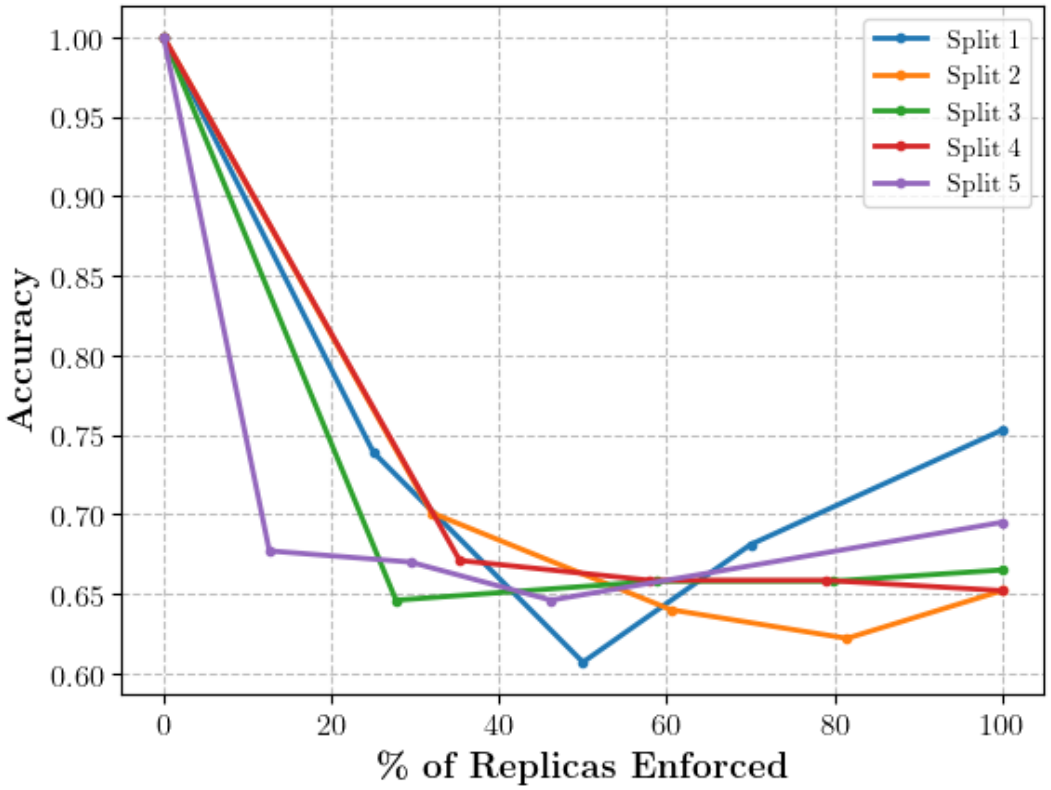


Fig. 5.6 **Analyzing Accuracy on the $HE_{BP,R}$ Dataset Relative to the Size of the Constraint.** We evaluate the accuracy of GPT-3.5 on the $HE_{BP,R}$ dataset with different amounts of replicas enforced. The percentage of replicas enforced is determined by the number of occurrences corresponding to the functions within the current subset of replicas V_{sub}. There are many ways to split the replicas into roughly even groups of four. Each "split" is a unique way. Detailed information about these splits can be found in Sec. 4.3.1. Critically, when a function from V_{rep} is not included in the enforced replicas, its Python Standard Library equivalent is not considered an invalid function. The most noticeable drop in performance occurs with the initial constraint of only a few replicas, suggesting that imposing a constraint, even a minor one, is what significantly impacts the model's performance. In certain scenarios, performance starts to rise as more functions are included, possibly due to the model learning from the code supplied for new replicas. When all replicas are enforced (100% on the x-axis), different accuracies are observed for different splits, creating a variability of about +/- 5% in accuracy. This variation stems from the fact that, for each split, the ordering of replicas in the prompt is different.

performance is inherently unstable for these questions. Minor variations in the prompt could sway its accuracy, making the performance decrease even more pronounced when constrained to replicas.

Figure 5.5 depicts a representative response of the model to a HumanEval question, both in its unconstrained and constrained states. In the vast majority of instances where the constrained model failed on $HE_{BP,R}$, the root cause is attributed to logical inaccuracies rather than syntactical oversights or misapplications of the replicas. A comprehensive analysis of all failure cases can be found in Appendix A.

Ablation

To investigate the reasons behind the observed performance decline, we examine GPT-3.5's accuracy on $HE_{BP,R}$. We vary the model's constraints using different fractions of the original replica set. Instead of defining these fractions simply as the count of replicas (e.g., 5 out of 21 total replicas representing roughly 25%), we define them based on the cumulative "number of occurrences" represented by the constrained replica subset. Refer to Section 4.3.1 for a comprehensive description. Notably, if an original replica is ablated and not included in V, its associated origin function is still treated as a valid function. This ensures that, regardless of the fraction of replicas enforced, all essential functions are available.

The most significant decline in performance occurs when only a small percentage of replicas are enforced. Interestingly, as the number of enforced replicas increases, the performance decrement is marginal, even improving in some instances. This implies that **it is the imposition of the constraint, rather than the magnitude of it, which causes a decrease the models' performance.** The increase in accuracy as more replicas are enforced in the constraint may be caused by the model learning in-context as more source code is provided for new replicas. Although the full set of functions in V_{rep} is accounted for in the 100% data point, the ordering of these functions within the prompt differs. This results in accuracy fluctuations of about +/- 5%, indicating the influence of prompt variability on model performance.

5.3 Providing Sub-functions to the Constrained Models to Recover Performance

In the prior experiment, we showed a performance degredation when models were constrained to replicas. Here, we demonstrate that while adhering to this constraint, introducing functions, as detailed in Sections 3.2 and 3.3, can ameliorate some of this performance loss. To achieve this, we employ GPT-3.5, GPT-4, and a human-expert to independently propose and implement functions for questions in HumanEval, HE_{CF}, HE_{BFF} and $APPS_{BP}$. These functions are then provided to the GPT-3.5 model, which is still constrained to replicas,

f^*_{soln} provided?	Skill-Proposing LM	**HumanEval**	**HE$_{\mathbf{CF}}$**	**HE$_{\mathbf{BFF}}$**	**APPS$_{\mathbf{BP}}$**
No	**GPT-4**	+7.3	+50	+42.1	-
	GPT-3.5	+6.7	+62.5	+10.5	+7.2
Yes	**GPT-4**	-	+62.5	+31.6	-
	GPT-3.5	-	+37.5	+15.8	-
	Human Expert	-	+100	-	-

Fig. 5.7 **Assessing Performance Gains when Providing Sub-Skills.** We evaluate constrained GPT-3.5's performance on various datasets when provided with a skill proposed by either GPT-3.5 or GPT-4, with and without access to unconstrained ground-truth reference code f^*_{soln}. Temperature was set to 0.2 for skill-proposing models, and 0 for the code-generating model. The displayed *gain* in accuracy refers to improvement relative to the constrained model operating without skills. Universal improvement across the HumanEval and APPS dataset when skills are introduced underlines the effectiveness of incorporating skills to enhance performance post-constraint. While the full recovery of performance lost to constraining is only achieved by the human expert, the measurable gain achieved from automated skill-proposal affirms the viability of our approach.

and its performance with the added functions is evaluated. Information on the datasets is provided below. For more details, see Section 4.1:

- HE_{CF} contains 8 HumanEval questions which were answered correctly by the unconstrained GPT-3.5 in a single attempt, but the constrained model failed even after three attempts.
- HE_{BFF} contains 19 HumanEval questions that neither the unconstrained nor the constrained GPT-3.5 model could answer, even after three attempts.
- $APPS_{BP}$ contains 1329 questions from the APPS dataset which the unconstrained GPT-3.5 model successfully solved in a single attempt.

Figure 5.7 presents the "half-shot" pass@1 accuracy improvement observed when functions are incorporated into the constrained GPT-3.5 model. The table evaluates the efficacy of functions proposed by GPT-3.5, GPT-4 and a human-expert, with an additional assessment based on the provision of ground-truth, unconstrained code f^*_{soln} to the proposing models. It should be noted that the human expert meticulously crafted the ideal functions tailored for the HE_{CF} dataset, serving as an example of how performance can be fully restored given optimal functions. A detailed comparison of human-crafted versus GPT-inferred functions unfolds later in this section. In figure 5.8, we present qualitative examples of the functions proposed for questions in HE_{CF} by GPT-4, GPT-3.5 and a human-expert. **The**

	2	84	89	79	154	110	65	82	PU	GU
get_decimal_part	Pass	F	F	F	F	F	F	F	1	0
sum_of_digits	F	F	F	F	F	F	F	F	0	0
join_list_to_string	F	F	F	F	Pass	F	F	F	0	0.12
format_binary_string	F	F	F	Pass	Pass	F	F	F	1	0.25
rotate_string	F	F	Pass	F	Pass	F	F	F	1	0.25
count_even_numbers	F	F	F	F	Pass	Pass	F	F	1	0.25
calculate_effective_shift	Pass	F	F	F	F	F	Pass	F	1	0.25
is_divisible	F	F	F	F	F	F	F	F	0	0

	2	84	89	79	154	110	65	82	PU	GU
get_decimal_part	Pass	F	F	F	Pass	F	F	F	1	0.25
sum_of_digits_binary	Pass	F	F	F	Pass	Pass	F	Pass	0	0.50
shift_character	F	F	Pass	F	Pass	F	Pass	F	1	0.37
add_extra_characters	F	F	F	F	Pass	F	F	F	0	0.12
is_rotation	F	F	F	F	F	F	F	F	0	0
count_even_elements	F	F	Pass	F	Pass	Pass	F	F	1	0.37
reverse_string	F	F	F	F	F	F	Pass	F	1	0.12
is_prime	F	F	F	F	Pass	F	F	Pass	1	0.25

	79	154	89	82	110	84	65	2	PU	GU
remove_bin_prefix	Pass	F	F	F	F	F	F	F	1	0.12
rotate_string	F	Pass	F	F	F	F	F	F	1	0.12
caesar_shift	F	F	Pass	F	F	F	F	F	1	0.12
is_prime	F	Pass	F	Pass	F	F	F	F	1	0.25
replace_from_another	F	F	F	F	Pass	F	F	F	1	0.12
sum_digits	Pass	F	Pass	F	F	Pass	F	F	1	0.37
is_shift_greater_than_len	F	Pass	F	F	F	F	Pass	F	1	0.25
get_floor	F	F	F	Pass	F	F	F	Pass	1	0.25

Fig. 5.8 **Skills Proposed for Questions in HE_{CF}.** Each row represents the name of the sub-skill implemented for a particular question in HE_{CF}, with the target question ID forming a diagonal relationship within the table. Upon including the skill as the only element of V^*, the constrained GPT-3.5 model was evaluated across the entire dataset (i.e., forming a row), with successful attempts marked as passes. Both Precise Utility (PU) and General Utility (GU) were computed for each sub-skill. (**top**) Sub-skills proposed by GPT-4 with access to unconstrained, ground truth code for reference. (**middle**) Sub-skills proposed by GPT-3.5 with no access to ground truth reference code. (**bottom**) Skills written by a human-expert. Note that the human expert invested considerable time and effort in creating skills that ensured successful passes on the targeted questions.

results highlight the ability of LM-generated functions in addressing challenges previously impassible by the constrained model (+62.4% on HE_{CF}). Impressively, it also resolves nearly half of the questions that even the unconstrained model initially faltered upon (+42.1% on HE_{BFF}) — all this while honoring the constraint. The adeptness of both models to propose productive functions without ground-truth code further highlights the potential generalisability of our approach.

Modularity

Figure 5.9 displays a key advantage of our methodology: the modularity of the proposed functions. **Many functions, despite being crafted with the intent of assisting a singular question, find relevance across diverse tasks.** Figure 5.9 illustrates the `is_prime` sub-skill, independently proposed and successfully used for three different questions spanning two datasets.

Cost-Saving

Presently, API calls to GPT-4 are roughly 20x more expensive than GPT-3.5. An advantage to our approach lies in its potential to harness the prowess of superior, albeit more costly models like GPT-4 without incurring the cost associated with using them as the main code-generating model. Employing GPT-4 in a skill-proposing capacity optimizes its knowledge extraction—evidenced by a +42% accuracy on HE_{BFF} versus GPT-3.5's +10%. Entrusting GPT-4 with complete coding tasks could elevate costs substantially, with the performance trade-off remaining an area of prospective exploration.

5.3.1 Comparison of Human and GPT-Generated Sub-Skills

The preceding section highlighted the utility of functions in enhancing the performance of constrained models. While previously we only considered functions crafted by an expert, this section delves into a comparison between functions created by language models and those conceived by *non-expert* humans.

For this analysis, three MLMI cohort students, GPT-4, and GPT-3.5 were assigned the task of formulating functions for the complete set of problems in HE_{CF}. The students were provided with the same prompt that was given to both GPT-4 and GPT-3.5. Half the problems (four) were accompanied by unconstrained ground-truth code as a reference, while the remaining problems were devoid of such reference. Subsequently, each batch of proposed functions was presented to the constrained GPT-3.5 model on a question-by-question basis, meaning

From APPS

Proposed Skill

```
def is_prime(n):
    'Checks and returns whether the given number is prime.
        '
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
```

Solution

```
def get_next_prime(n):
    next_prime = n + 1
    while not is_prime(next_prime):
        next_prime += 1
```

From HumanEval_82

Proposed Skill

```
def is_prime(n):
    """Check if a number
        is prime."""
    if n < 2:
        return False
    for i in range(2, int(
        n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Solution

```
def prime_length(string):
    length = get_length(
        string)
    return is_prime(length
        )
```

From HumanEval_75

Proposed Skill

```
def is_prime(n):
    """Check if a number is prime or not."""
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
```

Solution

```
def is_multiply_prime(a):
    factors = prime_factors(a)
    if get_length(factors) == 3:
        return check_if_all_true(apply_func_to_iterable(is_prime, factors))
    return False
```

Fig. 5.9 **Demonstrating Modularity**. This figure highlights the modularity inherent in our method by illustrating how a single skill (`is_prime`) was independently proposed (and successfully used) for three distinct questions across two different datasets. This shows that even though skills are proposed to aid only a specific question, they often possess utility that extends to other tasks.

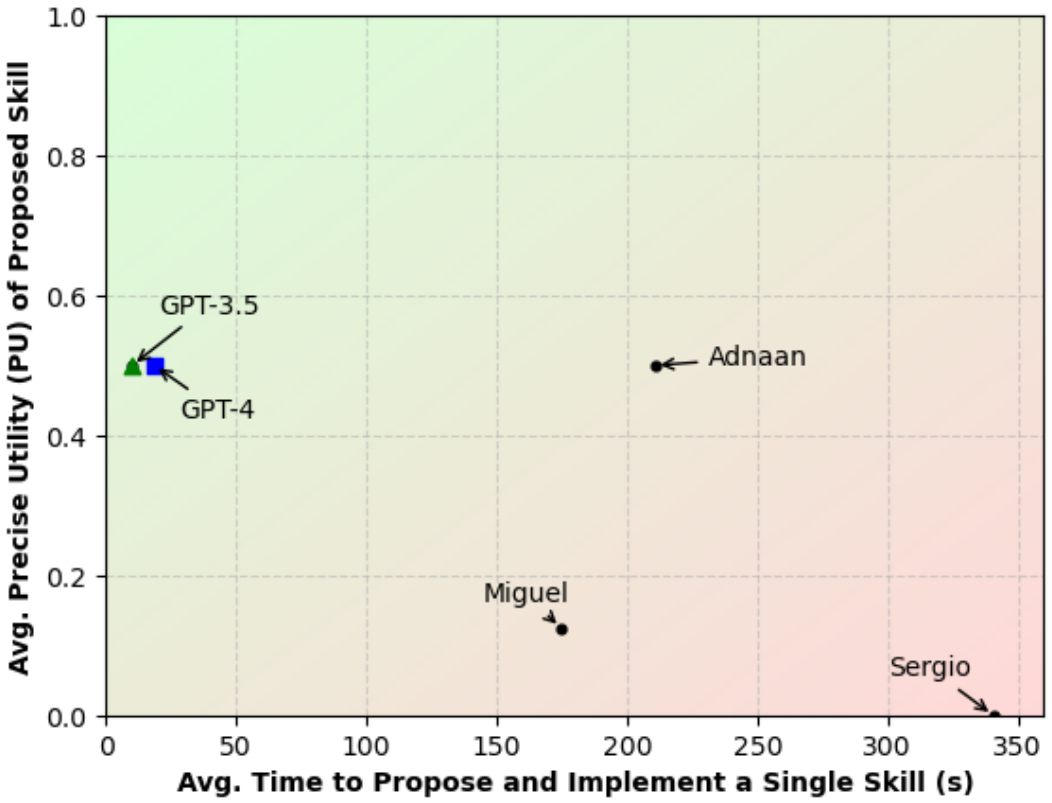


Fig. 5.10 **Comparing Skills Proposed by Language Models and Humans.** In this comparison, three students from the MLMI cohort, along with GPT-4 and GPT-3.5, were tasked with generating sub-skills for the eight problems within the HE_{CF} dataset. Both the Precise Utility (PU) of the skills and the time required for conception and implementation (i.e., writing the code) were evaluated. The reported averages reveal that human skill proposers, despite their above-average coding ability, produced skills comparable to or even less effective than those proposed by the LMs, while requiring considerably more time. Interestingly, the student who achieved the best results has had extensive experience working with GPT-4 and GPT-3.5, reflecting the influence of familiarity with these models on skill-proposal efficacy.

that only the sub-function F_i proposed for a certain question q_i was included in V^* only when code solving question q_i was sampled. The model was evaluated for “half-shot" pass@1 accuracy. The average Precise Utility (PU) of these proposed functions, a metric that evaluates whether the model effectively employed the given sub-function to correctly answer the question, was recorded, along with the time taken to propose and implement each sub-skill.

Figure 5.10 displays the outcomes. Optimal skill-proposers are located in the top-left quadrant, indicating fast proposal of high PU functions. In contrast, less effective proposers reside in the bottom right. Notably, every human participant required at least 15x longer

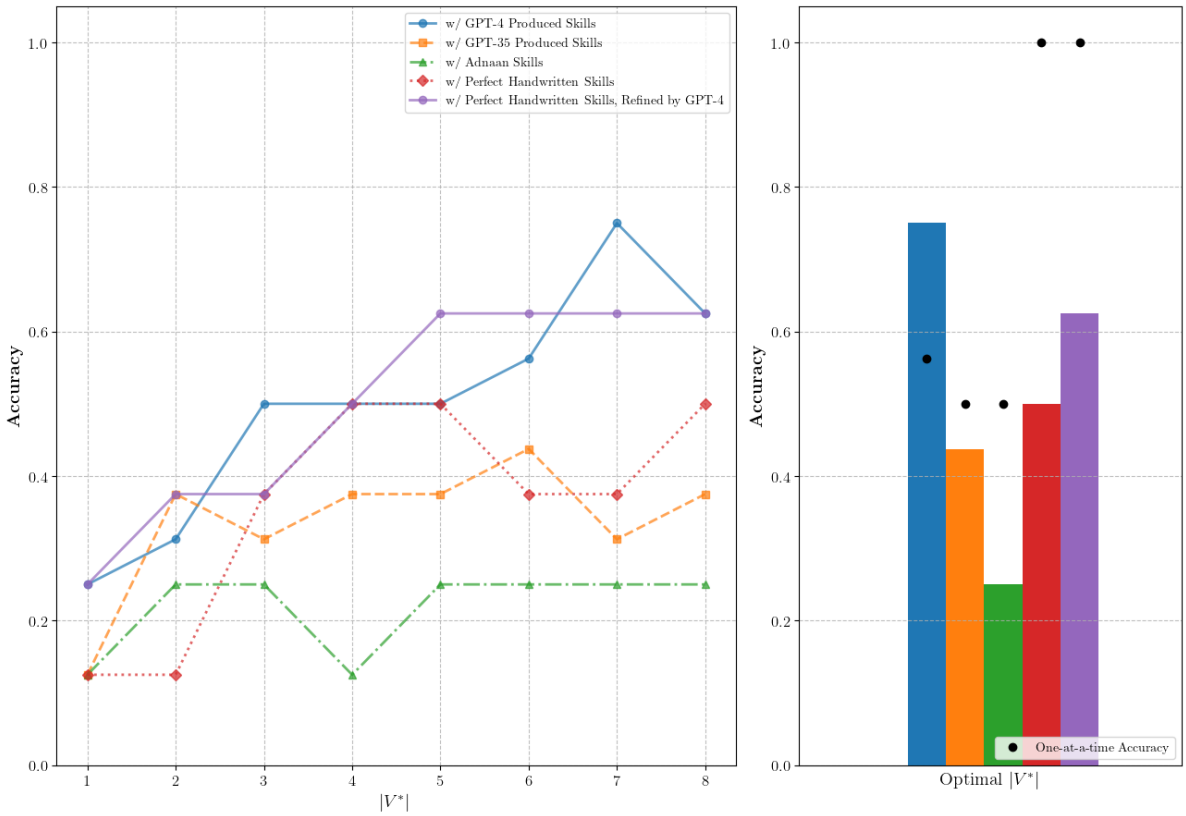


Fig. 5.11 **Impact of $|V^*|$ on Performance in HE_{CF}.** This figure analyzes the relationship between the number of functions included in V^* and performance on the HE_{CF} dataset, using functions proposed by GPT-4 (blue), GPT-3.5 (orange), and a human expert (red). The details of the proposed functions are provided in Figures 5.8. The batch of functions marked in purple are the perfect-handwritten skills after being refined by GPT-4. Results for sub-skills proposed by a non-expert human (Adnaan) are presented in green. The black dot on the right bar plot represents the naive expected accuracy when all functions are added ($|V^*| = 8$), matching the average accuracy across all 8 questions when only a single function is included in the constrained model's context when solving the related question. The model is only able to achieve 50% of its one-at-a-time accuracy when all human-written functions are provided (red bar plot). But when providing all functions generated by GPT-4, the model achieves 120% of its one-at-a-time accuracy (blue bar plot).

than either GPT-4 or GPT-3.5 to suggest a sub-skill, and their proposals were either equivalent or inferior in quality. **The primary takeaway is that, relative to the average human, language models can efficiently generate functions of comparable or superior quality in a fraction of the time.** Nonetheless, humans have the potential to propose superior functions to language models (i.e., human-expert) if they are prepared to invest significant time and expertise.

5.3.2 Determining the Optimal Number of Sub-skills to Provide, Per Question

In previous experiments, functions were provided to the constrained model per question. This means that for a given question (e.g., question ID 2 in HumanEval) only the sub-function generated with respect to the question (e.g., `get_decimal_part` in Figure 5.8) is provided to the constrained model on its subsequent at solving the given question. In this analysis, we explore how varying the number of functions in V^* influences model performance. As mentioned in Section 3.3, V^* represents a subset of valid functions highlighted in the prompt p_{cg}, deemed particularly relevant for a specific problem.

For this experiment, we use the 8 functions generated for each HE_{CF} question by GPT-4, GPT-3.5, and handwritten by a human-expert. Details of these functions are provided in Figure 5.8. We also include the functions generated by Adnaan, as discussed in the previous section. Note that in each set of 8 generated functions, each sub-function is proposed with the intention of supporting only one unique problem in HE_{CF}. For each set of 8 proposed functions, we vary the inclusion of these functions (from 1 to 8) within V^* and evaluate replica-constrained GPT-3.5 performance on all questions in HE_{CF}. When a sub-function is ablated from V^*, it is also made absent from V, meaning that the model is not provided the sub-function in any way.

Performance is compared against the average accuracy obtained when providing a sub-function solely to its corresponding HE_{CF} question. We call this the 'one-at-a-time-accuracy'. It is found by averaging across rows in the tables presented in Figure 5.8. The naive assumption is that when all functions are integrated into V^*, the accuracy should mirror the average 'one-at-a-time' configuration.

Interestingly, our results show that this is not the case. **We find that when the optimal amount of functions included (i.e., the $|V^*|$ which maximises accuracy), there is significant variability in how the model performs in comparison to the expected one-at-a-time accuracy.** For example, when using the "perfect" set of handwritten functions (deemed perfect because when the functions are fed one-at-a-time to their intended question, all questions pass; see bottom table in Figure 5.8) the model is only able to achieve 50% of its one-at-a-time accuracy when all functions are provided (red bar plot). But when providing functions generated by GPT-4, the model achieves 120% of its one-at-a-time accuracy (blue bar plot). In general, we see that GPT-generated functions, when provided to a constrained model alongside other functions, result in better performance than when handwritten functions are fed alongside other functions. Due to the small sample size of questions (8), this can easily be attributed to the stochastic nature of the constrained GPT-3.5 model (i.e.,

arbitrary variations in the prompt produce better/worse performance). However, if this is not the case, then this implies that the GPT models are formulating functions in a way that, when fed among a batch of other functions, allows the LM to better use the right function for the right problem.

To test the above hypothesis, we took each function in the set of perfect handwritten functions and asked GPT-4 to "Make this function more understandable." We then evaluated the results of this new set of refined functions, shown in purple. Although the refined set results in better performance than the non-refined set, it still only achieves 62% of its one-at-a-time accuracy. This results indicates that the functions in the handwritten set overfit the one-at-a-time scenario where they are the only sub-function being provided to the model. This means that they are perfectly crafted to trigger the model to output the correct solution when fed alone, but when combined with other functions (i.e., a variation in the prompt) the correct response isn't generated, **implying that the human-written functions do not generalise as well as GPT-generated functions to scenarios where multiple functions are provided at once.**

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this work, we have presented a method for synthesising programs which use only functions provided in-context. To do so, we have leveraged the capabilities of pre-trained LMs. Our approach does not only improve LMs' abilities to synthesise programs, but it also guides them in producing a modular, re-usable set of functions along the way. Our approach can be used to enable integration of proprietary or preferred functions into language models without retraining.

We introduced a technique to encourage models to adhere to constraints on allowed functions while also using provided functions effectively. Our results demonstrated that although GPT-3.5 and GPT-4 can generally use unseen functions supplied in-context, preventing use of highly exposed functions from training remains challenging. Nonetheless, we showed that GPT-4 attained 78.2% accuracy on $HE_{BP,R}$ when constrained to a set of 21 handwritten function replicas, compared to 94.2% when unconstrained. $HE_{BP,R}$ is a subset of HumanEval containing only questions that can be solved using the provided replica functions, confirming that the performance drop is not due to limiting necessary functions.

When constrained models fail to produce working programs, we presented an automated method to generate helpful sub-functions without human intervention. Our approach used a separate language model to analyse dysfunctional code and propose new functions to aid the constrained model. Experiments revealed that providing just one model-generated sub-function per question enabled constrained GPT-3.5 to solve over 60% of previously impassable HumanEval problems. When evaluated on the larger APPS dataset, supplying a single sub-function allowed the constrained model to solve 7.2% of questions it initially failed. In both cases, the model succeeded on these tasks when not constrained. Critically,

many sub-functions proposed in intermediary steps displayed general utility across diverse tasks.

We also introduced a "half-shot" evaluation paradigm that establishes tighter estimates of language models' unaided coding abilities by combining formatting instructions with output parsing. Under this evaluation, GPT-4 achieved 85% HumanEval accuracy, compared to 67% under standard zero-shot conditions. We brought into question the assumed value of a number of works which have proposed ways to improve general coding-abilities but did not surpass the "half-shot" evaluation mark.

In conclusion, this thesis makes valuable contributions towards using large language models for customisable program synthesis. We presented methods to constrain, enhance, and evaluate language models for code generation tasks using only user-provided functions. Our techniques allow pre-trained language models to be rapidly tailored for proprietary coding needs without compromising utility.

6.2 Future Work

There are many exciting directions for extending this work:

- Applying our approach to constrain models to non-replica function sets could demonstrate its utility for integrating real-world codebases or even algorithm discovery. As an example, constraining models to algorithm libraries like the one presented in [61] is a good starting point.
- Further investigation into optimal methods for communicating functions to language models is needed. Our initial, small-scale results imply that model-generated skills may generalise better than human-written skills when provided in batches, but larger experiments are required to confirm this trend.
- Testing context limits by progressively providing models with more in-context functions would reveal the maximum usable capacity before performance degradation occurs. Our initial results did not push the limits of the models' context windows.
- Experiments can be run to explore the trade-off between increasing model temperature and the model's adherence to a user-provided constraint. This is useful to help balance between creativity and rule-following when generating code.

- Evaluating our method's effectiveness when constraining models to proprietary functions from specific applications would demonstrate its practical value for integrating pre-existing components into larger programs.

A shortened version of this thesis is **intended to be submitted to the NeurIPS 2023 "ICBINB" Workshop.**

References

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *CoRR*, abs/1709.06182, 2017.

[2] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *International conference on machine learning*, pages 2123–2132. PMLR, 2015.

[3] Inc. Amazon Web Services. Codewhisperer. https://aws.amazon.com/codewhisperer/, 2023.

[4] Anthropic. Model card and evaluations for claude models, 2023.

[5] AntonOsika. gpt-engineer. https://github.com/AntonOsika/gpt-engineer, 2023.

[6] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.

[7] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity, 2023.

[8] Maxime Beauchemin. Promptimize. https://github.com/preset-io/promptimize, 2023.

[9] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? . In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '21, page 610–623, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383097. doi: 10.1145/3442188.3445922. URL https://doi.org/10.1145/3442188.3445922.

[10] Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew R Gormley. Unlimiformer: Long-range transformers with unlimited length input. *arXiv preprint arXiv:2305.01625*, 2023.

[11] Alan W. Biermann. Approaches to automatic programming. volume 15 of *Advances in Computers*, pages 1–63. Elsevier, 1976. doi: https://doi.org/10.1016/S0065-2458(08)60519-7. URL https://www.sciencedirect.com/science/article/pii/S0065245808605197.

[12] Alan W. Biermann. The inference of regular lisp programs from examples. *IEEE Transactions on Systems, Man, and Cybernetics*, 8(8):585–600, 1978. doi: 10.1109/TSMC.1978.4310035.

[13] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages*, 7(POPL):1182–1213, jan 2023. doi: 10.1145/3571234. URL https://doi.org/10.1145%2F3571234.

[14] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[15] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[17] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023.

[18] Lingjiao Chen, Matei Zaharia, and James Zou. How is chatgpt's behavior changing over time? *arXiv preprint arXiv:2307.09009*, 2023.

[19] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[20] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation, 2023.

[21] Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. Pymt5: multi-mode translation of natural language and python code with transformers. *CoRR*, abs/2010.03150, 2020.

[22] Antonia Creswell and Murray Shanahan. Faithful reasoning using large language models, 2022.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[24] Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Sean Welleck, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. Faith and fate: Limits of transformers on compositionality, 2023.

[25] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 835–850, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454080. URL https://doi.org/10.1145/3453483.3454080.

[26] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020. URL https://arxiv.org/abs/2002.08155.

[27] Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries, 2017.

[28] Paul Gauthier. Aider, 2023. URL https://github.com/paul-gauthier/aider.

[29] Angeliki Giannou, Shashank Rajput, Jy yong Sohn, Kangwook Lee, Jason D. Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers, 2023.

[30] Inc. GitHub and OpenAI. Github copilot. https://github.com/features/copilot, 2021.

[31] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014.

[32] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, page 219–239, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[33] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017. ISSN 2325-1107. doi: 10.1561/2500000010. URL http://dx.doi.org/10.1561/2500000010.

[34] HegelAI. Prompttools. https://github.com/hegelai/prompttools, 2023.

[35] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS, 2021. URL https://arxiv.org/abs/2105.09938.

[36] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 837–847. IEEE Press, 2012. ISBN 9781467310673.

[37] Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1143–1158, 2020.

[38] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. Self-planning code generation with large language model, 2023.

[39] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Pre-trained contextual embedding of source code. *CoRR*, abs/2001.00059, 2020.

[40] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. Spoc: Search-based pseudocode to code. *CoRR*, abs/1906.04908, 2019. URL http://arxiv.org/abs/1906.04908.

[41] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines, 2016.

[42] Jierui Li, Szymon Tworkowski, Yingying Wu, and Raymond Mooney. Explaining competitive-level programming solutions using llms, 2023.

[43] Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. *arXiv preprint arXiv:2303.16434*, 2023.

[44] Peter J. Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. Generating wikipedia by summarizing long sequences. *CoRR*, abs/1801.10198, 2018.

[45] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, mar 1971. ISSN 0001-0782. doi: 10.1145/362566.362568. URL https://doi.org/10.1145/362566.362568.

[46] Pedro Henrique Martins, Zita Marinho, and André FT Martins. Infinite memory transformer. *arXiv preprint arXiv:2109.00301*, 2021.

[47] OpenAI. Gpt-4 technical report, 2023.

[48] Jorge Parez, Pablo Barcela, and Javier Marinkovic. Attention is turing-complete. *Journal of Machine Learning Research*, 22(75):1–35, 2021. URL http://jmlr.org/papers/v22/20-302.html.

[49] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *CoRR*, abs/1611.01855, 2016.

[50] Python Software Foundation. Python standard library. https://docs.python.org/3/library/index.html, 2023. Online documentation.

[51] Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. Tool learning with foundation models, 2023.

[52] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding with unsupervised learning. 2018.

[53] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[54] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020. URL https://arxiv.org/abs/2009.10297.

[55] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.

[56] David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring lisp programs from examples. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'75, page 260–267, San Francisco, CA, USA, 1975. Morgan Kaufmann Publishers Inc.

[57] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580*, 2023.

[58] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

[59] Significant-Gravitas. Auto-gpt. https://github.com/Significant-Gravitas/Auto-GPT, 2023.

[60] Sahil Sindhi. Towards sample efficient and precise skill composition via coding transformers, 2023. supervised by Dr. Ignas Budvytis.

[61] TheAlgorithms. Python: All algorithms implemented in python. https://github.com/TheAlgorithms/Python/blob/master/DIRECTORY.md, 2023. GitHub repository.

[62] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu,

Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[64] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[65] Laura Weidinger, John Mellor, Maribeth Rauh, Conor Griffin, Jonathan Uesato, Po-Sen Huang, Myra Cheng, Mia Glaese, Borja Balle, Atoosa Kasirzadeh, Zac Kenton, Sasha Brown, Will Hawkins, Tom Stepleton, Courtney Biles, Abeba Birhane, Julia Haas, Laura Rimell, Lisa Anne Hendricks, William Isaac, Sean Legassick, Geoffrey Irving, and Iason Gabriel. Ethical and social risks of harm from language models, 2021.

[66] Lilian Weng. Llm-powered autonomous agents. *lilianweng.github.io*, Jun 2023. URL https://lilianweng.github.io/posts/2023-06-23-agent/.

[67] Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. Visual chatgpt: Talking, drawing and editing with visual foundation models. *arXiv preprint arXiv:2303.04671*, 2023.

[68] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696, 2017.

[69] yoheinakajima. babyagi. https://github.com/yoheinakajima/babyagi, 2023.

[70] Wojciech Zaremba and Ilya Sutskever. Learning to execute, 2015.

[71] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D Goodman, and Nick Haber. Parsel : A (de-)compositional framework for algorithmic reasoning with language models, 2022. URL https://arxiv.org/abs/2212.10561.

[72] Shizhuo Dylan Zhang, Curt Tigges, Stella Biderman, Maxim Raginsky, and Talia Ringer. Can transformers learn to solve problems recursively?, 2023.

[73] Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul F. Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *CoRR*, abs/1909.08593, 2019.

Appendix A

Supplementary Tables

Here, we provide:

- A detailed report of all questions failed by GPT-3.5 when constrained to the replicas. This is shown in Figure A.1.
- A list of all the replicas and their hand-written code, presented in Table A.1.

Table A.1 Hand-written code used for the Replicas. The corresponding function from the Python Standard Library is given in the left column.

Original function	**Custom re-implementation (Replica)**

len()	`def get_length(iterable):`
	`  count = 0`
	`  for _ in iterable:`
	`  count += 1`
	`  return count`

str()	`def cast_to_string(input):` `  return str(input)`
chr()	`def convert_to_char(input):` `  return chr(input)`
float()	`def cast_to_float(input):` `  return float(input)`
int()	`def cast_to_int(input):` `  return int(input)`

list()	`def create_list(iterable=None):`
	`  if iterable is None:`
	`  return []`
	`  lst = []`
	`  for item in iterable: lst.append(item)`
	`  return lst`

set()	`def create_set(iterable=None):`
	`  s = {}`
	`  if iterable:`
	`  for element in iterable: s[element] = None`
	`  return s.keys()`

Table A.1 Hand-written code used for the Replicas. The corresponding function from the Python Standard Library is given in the left column.

Original function	Custom re-implementation (Replica)

isinstance()	`def check_if_instance(obj, class_or_tuple):`
	`  if not isinstance(class_or_tuple, tuple):`
	`  class_or_tuple = (class_or_tuple,)`
	`  for cls in class_or_tuple:`
	`  if type(obj) == cls or type(obj) in cls.__subclasses__():`
	`  return True`
	`  return False`

sorted()	`def sort_list(iterable, key=None, reverse=False):`
	`  lst = list(iterable)`
	`  if key is None:`
	`  compare = lambda a, b: a > b`
	`  else:`
	`  compare = lambda a, b: key(a) > key(b)`
	`  for i in range(len(lst)):`
	`  for j in range(len(lst) - 1):`
	`  if compare(lst[j], lst[j + 1]):`
	`  lst[j], lst[j + 1] = lst[j + 1], lst[j]`
	`  if reverse:`
	`  lst = lst[::-1]`
	`  return lst`

all()	`def check_if_all_true(iterable):`
	`  for element in iterable:`
	`  if not element:`
	`  return False`
	`  return True`

min()	`def get_minimum(*args):`
	`  if len(args) == 1:`
	`  args = args[0]`
	`  if not args:`
	`  raise TypeError('expected at least 1 arguments, got 0')`
	`  min_val = args[0]`
	`  for arg in args:`
	`  if arg < min_val:`
	`  min_val = arg`
	`  return min_val`

max()	`def get_maximum(*args):`
	`  ...`
	`  return max_val`

bin()	`def convert_to_binary(n):`
	`  if n < 0:`
	`  return '-' + convert_to_binary(-n)`
	`  result = ''`
	`  while n:`
	`  result = ('1' if n & 1 else '0') + result`
	`  n >>= 1`
	`  return '0b' + result if result else '0b0'`

sum()	`def compute_sum(iterable, start=0):`
	`  total = start`
	`  for item in iterable:`
	`  total += item`
	`  return total`

round()	`def round_number(number, ndigits=None):`
	`  if ndigits is None:`
	`  return int(number + 0.5) if number >= 0 else int(number - 0.5)`
	`  else:`
	`  factor = 10.0 ** ndigits`
	`  return int(number * factor + 0.5 if number >= 0 else number * factor - 0.5) / factor`

math.ceil()	`def get_ceiling(number):`
	`  integer_part = int(number)`
	`  if number == integer_part:`
	`  return integer_part`
	`  if number > 0:`
	`  integer_part += 1`
	`  return integer_part`

Table A.1 Hand-written code used for the Replicas. The corresponding function from the Python Standard Library is given in the left column.

Original function	Custom re-implementation (Replica)

math.sqrt()	`def get_square_root(input, precision = 0.00001):`
	`  guess = input / 2.0`
	`  while True:`
	`  better_guess = (guess + input / guess) / 2.0`
	`  if abs(guess - better_guess) < precision:`
	`  return better_guess`
	`  guess = better_guess`

ord()	`def get_unicode(char):`
	`  if len(char) != 1:`
	`  raise TypeError("Error." % len(char))`
	`  return int.from_bytes(char.encode('utf-8'), byteorder='big')`

map()	`def apply_func_to_iterable(function, iterable):`
	`  result = []`
	`  for item in iterable:`
	`  result.append(function(item))`
	`  return result`

abs()	`def absolute_value(number):`
	`  if number < 0:`
	`  return -number`
	`  else:`
	`  return number`

filter()	`def add_to_list_if_func_is_true(function, iterable):`
	`  result = []`
	`  for item in iterable: if function(item): result.append(item)`
	`  return result`

Task ID	Desc. of Failure	Failure Categorization
144	casts a string fraction to a float instead of splitting numerator and denominator first	logic
65	does not reverse string if shift % length = 0 (should always reverse when shift > len)	logic
124	assumes string formatting is correct (can be split on the '-') ; does not account for leap years	logic
149	sorts by string value rather than string length and alphabetically	logic
114	restarts sub-array if element is negative rather than if it is less than current element	logic
79	does not remove the 0b prefix from the convert_to_binary func.	logic (didn't understand custom func well)
142	does not include the case where index is not 3 or 4	logic
121	selects odd position elements instead of even by starting at index 1 instead of 0	logic
89	does not mod by 26 to ensure staying within alphabet	logic
97	needs to separate modulo operations by a parentheses	logic
154	does not correctly check rotations of b	logic
99	always rounds negative numbers toward 0, even when decimal part is greater than or equal to -0.5	logic
110	returns "yes" prematurely after checking if lst2 has even elements	logic
84	converts char to bin then bin to int instead of char to int	logic
70	does not recompute min and max value within the while loop	logic
82	used get_ceiling() when floor should've been used	logic
90	does not account for multiple smallest elements (i.e. [1,1] case) originally avoided by converting list to a set	logic

Fig. A.1 Detailed report of failures when prompted to use custom sub-functions.